An Introduction To Digital Signal Processors

Bruno Paillard, Ph. D. - P.eng. - Professor
Génie électrique et informatique
Université de Sherbrooke
January 27 2002

INTRODUCTION 5

1. Computers and microprocessors 5

2. Application areas of the microprocessor 7

3. Examples of embedded systems 8

MICROPROCESSORS 9

1. Applications and types of microprocessors 9

2. Evolution of the software development tools and methodologies 13

3. Present frontiers 14

4. Criteria for choosing a microprocessor 15

5. Building blocks of an embedded system 17

6. Program execution 21

SYSTEM ARCHITECTURE 23

1. Von Neumann architecture 23

2. Harvard architecture 25

3. Pros and cons of each architecture 25

4. Busses, address decoding and three-state logic 26

5. Memories: technology and operation 44

INTRODUCTION TO THE SIGNAL RANGER DSP BOARD 51

1. Hardware features 51

2. Software tools 51

BINARY REPRESENTATIONS AND FIXED-POINT ARITHMETIC 55

1. Bases and positional notations 55

2. Unsigned integer notation 57

3. Signed Integer notation using a separate sign bit 61

4. 2's complement signed notation 62

5. **Offset binary notation** 68

6. **Fractional notations** 69

7. **Floating point notation** 73

8. **BCD notation** 75

9. **ASCII Codes** 77

SOFTWARE DEVELOPMENT TOOLS 79

1. **Cross-development tools** 79

2. **Code development methodology** 82

3. **Code testing** 88

THE TMS320C5402: ARCHITECTURE AND PROGRAMMING 91

1. **Architecture** 91

2. **CPU** 93

3. **Operands and addressing modes** 107

4. **Branch instructions** 118

5. **Repeats** 120

6. **Other conditional instructions** 121

7. **The stack** 122

8. **Functions** 127

9. **Interrupts** 145

ON-CHIP PERIPHERALS OF THE TMS320C5402 157

1. **Introduction** 157

2. **Timers** 158

3. **Host Port Interface (HPI)** 158

4. **Multi-channel Buffered Serial Ports (McBSPs)** 158

5. **Direct Memory Access (DMA)** 159

6. **BIO and XF signals** 160

7. **Clock generator** 160

8. **Wait-state generator** 161

SOFTWARE DEVELOPMENT METHODOLOGY FOR EMBEDDED SYSTEMS 163

1. Introduction 163

2. Avoiding problems 165

3. Modular programming practices 166

4. Structured programming 168

5. Managing peripherals: interrupts vs. polling 179

6. High-level language development 191

SIGNAL RANGER'S DEVELOPMENT TOOLS 193

1. Introduction 193

2. Mini-debugger 193

3. Communication kernel 200

APPENDIX A – SCHEMATICS OF THE SIGNAL RANGER DSP BOARD 208

Introduction

- 1. Computers and microprocessors
- 2. Applications areas of the microprocessor
- 3. Examples of embedded systems.

1. COMPUTERS AND MICROPROCESSORS

Charles Babbage invented the concept of computer in the mid 19th century, but the concept had to wait for the development of vacuum tube electronics in the 1930s to achieve its full potential.

John W. Mauchly and J. Presper Eckert at University of Pennsylvania's Moore School of Electrical Engineering developed one of the first electronic computers between 1942 and 1946. Called ENIAC (Electronic Numerical Integrator and Computer), it was originally used to calculate ballistic tables for the military. With 17 468 vacuum tubes and 100 feet of front panel, this 30 tons mighty machine was capable of doing 5000 additions and 300 multiplications a second. Although it is less than 1/10 000th the computational speed found in a modern cellular phone, due to its "all electronic" design, it was the fastest computer in use at the time. Later models were used for nuclear physics and aerodynamics research, two fields where the super-computer is still a tool of the trade today.

Although not publicised at the time, the first electronic computer was actually built by Tommy Flowers, an electronics engineer in the British secret service, during the Second World War. This computer called Colossus was used by the British secret service to decipher German military codes. Because of the secrecy that surrounded these operations it was not recognized as the first electronic computer until recently.

Initially computers were used to carry out numerical computations, but nowadays they are used in many applications from music players to flight control systems.

A computer performs its task by the sequential execution of elementary binary operations called "instructions". These elementary instructions may represent the addition of two numbers for instance, or a comparison between two numbers, or the transfer of a binary word from one memory location to another. They are assembled in a complete "program" that defines the global task carried out by the computer.

The part of the computer that executes the instructions is called the Central Processing Unit (CPU).

A microprocessor is a silicon chip that implements the complete Central Processing Unit of a computer. Silicon photolithographic and etching processes are used to mass-produce microprocessors.

The development of the microprocessor in the 1970's represents a major milestone in the history of electronics and computer systems. It enabled the development of low cost computers, which in time became "personal computers". It also spawned the field of "embedded systems", in which a microprocessor is used to control an electronic device or subsystem. Nowadays, nearly all customer, scientific and industrial electronics incorporate microprocessors.

The paternity of the microprocessor is still debated today. In 1971, Intel introduced the 4004, which included all the elements of a 4-bit CPU. The same year Texas Instruments introduced the TMS1802NC. Both microprocessors were originally intended to support the functions of a desk calculator. The TMS1802NC was not very flexible. Its program, in particular, was stored in a Read Only Memory, which contents were permanently etched on the silicon chip. Modifying the program required the development of new lithography masks. Texas Instruments received a patent in 1973 for the development of the microprocessor.

Intel continued its development effort and produced the 8008 in 1972, the 8080 in 1974, and the 8086 in 1978. These microprocessors were the precursors of today's Pentiums.

Several companies followed in Intel and Texas Instrument's footsteps. Motorola with the 6800 and Rockwell with the 6502 are two examples.

The first years, microprocessors were not very differentiated and these first machines were used equally in computer systems and in embedded systems. For instance Rockwell's 6502 was used in a number of embedded system development boards, such as the KIM and the AIM65. It was also the CPU of one of the first personal computers: the PET produced by the Commodore company.

Later, looking for increased performance, as well as new markets, microprocessor manufacturers specialized their designs. The first micro-controller, the TMS1000 from Texas instruments was introduced in 1974. Microcontrollers not only possess a CPU on the silicon chip, but also integrate a number of peripherals (memory, parallel ports analog to digital converters…etc.). In essence they constitute complete microcomputers integrated on the same chip of silicon. The addition of peripherals to the core CPU makes microcontrollers particularly efficient in embedded systems applications where cost, size and power consumption must be kept low. For instance a microwave oven control unit was one of the first applications targeted by the TMS1000 microcontroller. In the 1980s Intel introduced the 8748 microcontroller family. This family integrated many peripherals, including a program memory that was erasable and reprogrammable by the developer. These characteristics lowered the development cost of microcontroller systems, and enabled the use of microcontrollers in low-volume embedded applications.

The 1980s also saw the introduction of the first Digital Signal Processors. Introduced in 1983 by Texas Instruments, the TMS320C10 was a microprocessor specifically designed to solve digital signal processing problems. Prior to its release, signal processing was mostly the domain of analog electronics. Digital signal processing applications were few and usually required high-cost complex machines that were only viable in aerospace or military applications. The introduction of the DSP ushered the establishment of digital signal processing as one of the core disciplines of Electrical Engineering. Digital Signal processing progressively replaced analog signal processing in applications that range from control to telecommunications. This "digital migration" is still in progress today, and affects applications of ever-decreasing cost. Digital signal processing implements techniques and technologies that are much more advanced and entail a lot more complexity than its analog counterpart. However the flexibility allowed by programming, the precision and inherent stability of the processing parameters, as well as the possibility of very complex and adaptive processes, nearly impossible to

implement in analog form, combined to the very low cost of today's microprocessors make this approach unavoidable.

The differentiation brought about by the integration of specialized peripherals onto the microprocessor's silicon chip produced devices that are extremely specialized. Some microcontrollers for instance are specifically designed for applications such as communications protocols (Ethernet, USB, etc.). Others still are specifically designed for use in electric motor drives, and so on…

The benefit of such specialization is the production of very efficient designs, in terms of cost, size and power consumption. On the other hand it forces the developer to learn and master an ever-increasing variety of CPUs. This difficulty must not be under-estimated. The time investment needed to get familiar with a new microprocessor and its software development tools is so great that it is often a larger obstacle to the adoption of this microprocessor by the developer, than the technical limitations of the microprocessor itself. Many developers will go through untold contortions to keep working with a microprocessor with which they are already familiar. On the manufacturer's side, the introduction of a new family of microprocessors is a very complex and costly operation, and the adoption of the microprocessor by the market is far from guaranteed.

To circumvent this problem some manufacturers are introducing new types of microcontrollers that incorporate programmable logic and programmable analog subsystems. This is the case of the PsoC (Programmable System on a Chip) family from Cypress. The integration of programmable sub-systems gives tremendous flexibility to the product. The developers can, to a great extent, specialize the microcontroller themselves by designing their own set of peripherals tailored to the application.

2. APPLICATION AREAS OF THE MICROPROCESSOR

The most obvious application of the microprocessor is the computer. From the super-computer used for scientific computation to the pocket personal computer used for word processing or Internet browsing, computers have become an essential tool in our everyday life.

The world of computer systems however accounts only for a small fraction of the total number of microprocessors deployed in applications around the world. The vast majority of microprocessors are used in *embedded systems*. Embedded systems are electronic devices or sub-systems that use microprocessors for their operation. The low cost of microprocessors, combined to the flexibility offered by programming makes them omnipresent in areas ranging from customer electronics to telecommunication systems. Today, it is actually very difficult to find electronic devices or sub-systems that do not incorporate at least one microprocessor. Needless to say, the vast majority of the engineering and design activity related to microprocessor systems is in the area of embedded systems.

By contrast to computer systems, the program of an embedded system is fixed, usually unique, and designed during the development of the device. It is often permanently stored in a Read Only Memory and begins its execution from the moment the device powered on. Because it is fixed and usually resides in a Read Only Memory, the software of an embedded system is often called *firmware*.

3. EXAMPLES OF EMBEDDED SYSTEMS

At home, the microprocessor is used to control many appliances and electronic devices: microwave oven, television set, compact-disk player, alarm system are only a few examples. In the automobile microprocessors are used to control the combustion of the engine, the door locks, the brake system (ABS brakes) and so on… Microprocessors are used in most measuring instruments (oscilloscopes, multi-meters, signal generators…etc). In telecommunication systems microprocessors are used in systems ranging from telephone sets to telephone switches. In the aerospace industry they are used in in-flight navigation systems and flight control systems.

Microprocessors are even used in very low-cost applications such as wristwatches, medical thermometers and musical cards.

Even in computer systems, *embedded microprocessors* are used in pointing devices, keyboards, displays, hard disk drives, modems… and even in the battery packs of laptop computers!

© BRUNO PAILLARD

Microprocessors

- 1. Applications and types of microprocessors
- 2. Evolution of tools and software development methodologies
- 3. Present frontier
- 4. Criteria for choosing a microprocessor
- 5. Building blocks of an embedded system
- 6. Program execution

1. APPLICATIONS AND TYPES OF MICROPROCESSORS

Today, microprocessors are found in two major application areas:

- Computer system applications
- Embedded system applications

Embedded systems are often high-volume applications for which manufacturing cost is a key factor. More and more embedded systems are mobile battery-operated systems. For such systems power consumption (battery time) and size are also critical factors. Because they are specifically designed to support a single application, embedded systems only integrate the hardware required to support this application. They often have simpler architectures than computer systems. On the other hand, they often have to perform operations with timings that are much more critical than in computer systems. A cellular phone for instance must compress the speech in real time; otherwise the process will produce audible noise. They must also perform with very high reliability. A software crash would be unacceptable in an ABS brake application.

Digital Signal Processing applications are often viewed as a third category of microprocessor applications because they use specialized CPUs called DSPs. However, in reality they qualify as specialized embedded applications.

Today, there are 3 different types of microprocessors optimized to be used in each application area:

- **Computer systems:** General-purpose microprocessors.
- **Embedded applications:** Microcontrollers
- **Signal processing applications:** Digital Signal Processors (DSPs)

In reality the boundaries of application areas are not as well defined as they seem. For instance DSPs can be used in applications requiring a high computational speed, but not necessarily related to signal processing. Such applications include computer video boards and specialized co-processor boards designed for intensive scientific computation. On the other hand, powerful general-purpose microprocessors such as Intel's i860 or Digital's Alpha-chip are used in high-end digital signal processing equipment designed for algorithm development and rapid prototyping.

The following sections list the typical features and application areas for the three types of microprocessors

1.1. General purpose microprocessors

1.1.1. Applications

- *Computer systems*

1.1.2. Manufacturers and models

- *Intel:* *Pentium*
- *Motorola:* *PowerPC*
- *Digital:* *Alpha Chip*
- *LSI Logic:* *SPARC family (SUN)*

... etc.

1.1.3. Typical features

- *Wide address bus allowing the management of large memory spaces*
- *Integrated hardware memory management unit*
- *Wide data formats (32 bits or more)*
- *Integrated co-processor, or Arithmetic Logic Unit supporting complex numerical operations, such as floating point multiplications.*
- *Sophisticated addressing modes to efficiently support high-level language functions.*
- *Large silicon area*
- *High cost*
- *High power consumption*

1.2. Embedded systems: Microcontrollers

1.2.1. Application examples

- *Television set*
- *Wristwatches*
- *TV/VCR remote control*
- *Home appliances*
- *Musical cards*
- *Electronic fuel injection*
- *ABS brakes*
- *Hard disk drive*
- *Computer mouse / keyboard*
- *USB controller*
- *Computer printer*
- *Photocopy machine*
- *... etc.*

1.2.2. Manufacturers and models

- *Motorola:* *68HC11*
- *Intel:* *8751*

- *Microchip:* *PIC16/17family*
- *Cypress* *PsoC family*
- *… etc.*

1.2.3. Typical features

- *Memory and peripherals integrated on the chip*
- *Narrow address bus allowing only limited amounts of memory.*
- *Narrow data formats (8 bits or 16 bits typical)*
- *No coprocessor, limited Arithmetic-Logic Unit.*
- *Limited addressing modes (High-level language programming is often inefficient)*
- *Small silicon area*
- *Low cost*
- *Low power consumption.*

1.3. Signal processing: DSPs

1.3.1. *Application examples*

- *Telecommunication systems.*
- *Control systems*
- *Attitude and Flight control systems in aerospace applications.*
- *Audio/video recording and play-back (Compact-disk/MP3 players, video cameras…etc.)*
- *High-performance hard-disk drives*
- *Modems*
- *Video boards*
- *Noise cancellation systems*
- *… etc.*

1.3.2. *Manufacturers and models*

- *Texas Instruments:* *TMS320C6000, TMS320C5000...*
- *Motorola:* *56000, 96000...*
- *Analog devices:* *ADSP2100, ADSP21000...*
- *... etc.*

1.3.3. Typical features

- *Fixed-point processor (TMS320C5000, 56000...) or floating point processor (TMS320C67, 96000...)*
- *Architecture optimized for intensive computation. For instance the TMS320C67 can do 1000 Million floating point operations a second (1 GIGA Flop).*
- *Narrow address bus supporting a only limited amounts of memory.*
- *Specialized addressing modes to efficiently support signal processing operations (circular addressing for filters, bit-reverse addressing for Fast Fourier Transforms…etc.)*
- *Narrow data formats (16 bits or 32 bits typical).*
- *Many specialized peripherals integrated on the chip (serial ports, memory, timers…etc.)*
- *Low power consumption.*
- *Low cost.*

1.4. Cost of a microprocessor

Like many other electronic components, microprocessors are fabricated from large disks of mono-crystalline silicon called “wafers”. Using photolithographic processes hundreds of individual microprocessor chips are typically fabricated on a single wafer. The cost of processing a wafer is in general fixed, irrespective of the complexity of the microprocessors that are etched onto it.

The silicon surface occupied by a microprocessor on the wafer is dependant on several factors. The most important factors being its complexity (number of gates, or transistors), and the lithographic scale indicating how small a gate can be.

At first glance, for a given fabrication process and a given lithographic scale, it would seem that the cost of a microprocessor is roughly proportional to its surface area, therefore to its complexity.

However things are a little bit more complex. In practice the wafers are not perfect and have a number of defects that are statistically distributed on their surface. A microprocessor whose area includes the defect is generally not functional. There are therefore always some defective microprocessors on the wafer at the end of the process. The ratio of good microprocessors to the total number fabricated on the wafer is called the “fabrication yield”. For a fixed number of defects per square inch on the wafer, which is dependant on the quality of the wafer production, the probability of a microprocessor containing a defect increases non-linearly with the microprocessor area. Beyond a certain surface area (beyond a certain complexity) the fabrication yield decreases considerably. Of course the selling price of the good microprocessors is increased to offset the cost of having to process and test the defective ones. In other words the cost of a microprocessor increases much faster than its complexity and surface area.

1.5. Power consumption of a microprocessor

As we discussed earlier, some microprocessors are optimized to have a low power consumption. Today almost all microprocessors are implemented in CMOS technology. For this technology, the electric current drawn by the microprocessor is almost entirely attributable to the electric charges used to charge and discharge the parasitic input capacitance of its gates during transitions from 0 to 1 and 1 to 0. This charge loss is proportional to the following factors:

- The voltage swing (normally the supply voltage).
- The input gate capacitance.
- The number of gates in the microprocessor.
- The average number of transitions per second per gate.

The input gate capacitance is fairly constant. As manufacturing processes improve, the lateral dimensions of transistor gates get smaller, but so does the thickness of the oxide layers used for the gates. Typical values are on the order of a few pF per gate.

With thinner oxide layers, lower supply voltages can be used to achieve the same electric field. A lower supply voltage mean that less charge is transferred during each transition, which leads to a lower current consumption. This is the main factor behind the push for decreasing supply voltages in digital electronics.

© BRUNO PAILLARD

The current consumption is also proportional to the number of gates of the microprocessor (its complexity), and to the average number of transitions per second (its clock frequency).

At identical fabrication technology and lithographic scale, a microprocessor optimized for power consumption is simpler (less gates), and operates at a lower clock frequency than a high-performance microprocessor.

For a given processor, the developer can directly adjust the trade-off between computational speed and power consumption by appropriately choosing the clock frequency. Some microprocessors, even offer the possibility of dynamically adjusting the clock frequency through the use of a Phase-Locked-Loop. It allows the reduction of the clock frequency during less intensive computation periods to save energy. Such systems are especially useful in battery-operated devices.

2. EVOLUTION OF THE SOFTWARE DEVELOPMENT TOOLS AND METHODOLOGIES

In the field of embedded systems, software development tools and methodologies can often appear to be lacking in sophistication, compared to the tools used in the field of computer systems. For instance, many embedded applications are developed entirely in assembly language. Software that is developed in high-level languages is often developed in C, and rarely in C++, even today. When schedulers and multi-tasking kernels are used, they are much simpler than their counterparts found in the operating systems of modern-day computers.

This apparent lack of sophistication may be attributed to several factors:

- An embedded system is usually designed to run a unique application. Providing a uniform set of standardized “operating system” services to software applications is therefore much less critical than it is for a computer system designed to run multiple applications that all have similar needs. In fact in many embedded systems, “operating system” functions like peripheral drivers and user-interface are custom-designed, along with the core functionality of the device.
- The software of an embedded system is generally much less complex than the software that runs on a computer system. The difficulties of embedded system development usually lie in the interaction between hardware and software, rather than in the interaction between software and software (software complexity).
- Most embedded systems are very specific devices. Their hardware resources have been optimized to provide the required function, and are often very specific and very limited. This makes the use of complex operating systems difficult. Indeed, to be efficient, a computer’s operating system often relies on considerable resources (memory, computational speed… etc.). Furthermore, it expects a certain level of uniformity and standard in the hardware resources present in the system.
- For embedded systems, real-time execution constraints are often critical. This does not necessarily mean that the computational speed must be high, but rather that specific processes must execute within fixed and guaranteed time limits. For instance, for a microcontroller driving an electric motor, the power bridge protection process (process leading to the opening of all transistors) must execute within 1 microsecond following the reception of a short-circuit signal, otherwise the bridge may be damaged. In such a situation, the execution of code

within a complex and *computationally hungry* operating system is obviously not a good choice.

- The development of code in a high-level language is often less efficient than the development in assembly language. For instance, even with the use of an optimizing C compiler, designed specifically for the TMS320VC5402 DSP for which it generates code, the development of a filter function can be 10 to 40 times slower when developed in C than when it is optimized in assembly language. For one thing a high-level language does not (by definition) give the developer access to low-level features of the microprocessor, which are often essential in optimizing specific computationally intensive problems. Furthermore, the developer usually possesses a much more complete description of the process that must be achieved than what is directly *specifiable* to a compiler. Using this more complete information, the developer can arrive at a solution that is often more efficient than the generic one synthesized by the compiler.
- In the field of computer systems, software development tools and operating systems are designed so that the developer does not have to be concerned with the low-level software and hardware details of the machine. In the field of embedded systems, the developer usually needs to have a fine control over low-level software, and the way it interacts with the hardware.
- Finally, for an embedded system, the execution of the code must be perfectly understood and controlled. For instance, a rollover instead of a saturation during an overflow can be disastrous if the result of the calculation is the attitude control signal of a satellite launch vehicle. This exact problem led to the destruction of the ESA's "Arianne V" rocket during its first launch in June 1996. In situations such as this one, the use of a high-level language such as C, for which the behaviour during overflow (rollover or saturation) is not defined in the standard, may be a bad choice.

3. PRESENT FRONTIERS

Today, the traditional frontiers between embedded systems and computer systems are getting increasingly blurred. More and more applications and devices have some attributes of an embedded system and some attributes of a computer system. A cellular telephone for instance may also provide typical "computer system" functions such as Internet connexion. On the other hand, a PC may provide typical "embedded system" functions such as DVD playback.

In many cases these devices are designed using a hybrid architecture in which a general-purpose microprocessor executes software under the control of a complex operating system, and in which special purpose microprocessors (DSPs for instance) are used as peripherals, and execute function-specific software (or firmware) typical of an embedded system. These architectures may even be implemented at the integrated circuit level where the general-purpose microprocessor and the DSP reside on the same silicon chip.

In other cases, microcontroller-based systems are enhanced by specialized peripherals to rigidly support traditional computer-system functions in a limited way. For instance the use of specialized peripherals to support a TCP/IP modem connexion can provide Internet access to a microwave oven controller.

More than 30 years after the development of the first microprocessors, the field is still in profound evolution.

© BRUNO PAILLARD

4. CRITERIA FOR CHOOSING A MICROPROCESSOR

The choice of a microprocessor for a given application is probably one of the most difficult tasks facing the systems engineer today. To take this action correctly, the engineer must know the whole range of microprocessors that could be used for the application, well enough to precisely measure the pros and cons of each choice. For instance, the engineer may be required to evaluate if a particular time-critical task can execute within its time limits for each potential choice of microprocessor. This evaluation may require the development of optimized software, which obviously entails a very good knowledge of the microprocessor and its software development tools, and a lot of work! In practice, the investment in time and experience necessary to know a microprocessor well is very high. Most engineers try to make their choice within the set of devices with which they are already familiar. This approach is sub-optimal, but reflects the reality of systems design. In this field, even seasoned designers do not have an in-depth experience of the very wide variety of microprocessors that are on the market today.

To help in the choice, most manufacturers offer “development kits” or “evaluation kits”. These kits allow the engineer to design and test software, without having to design the entire custom hardware required by the application. Manufacturers often also offer software examples and libraries of functions that allow the engineer to evaluate typical solutions without having to design the software. The availability of such tools and information biases the choice toward one manufacturer, or one family of microprocessors. On the other hand it is often seen by design engineers as legitimate criteria on which to base their choice.

The following general criteria may be applied for the choice of a microprocessor:

1 **Capacity of the microprocessor** (It is advisable to take some margin because the problem often evolves in the course of the design).

Logical criteria:

- Instruction set functionality.
- Architecture, addressing modes.
- Execution speed (not just clock frequency!)
- Arithmetic and Logic capabilities.
- Addressing capacity.

Physical criteria:

- Power consumption.
- Size.
- Presence of on-chip peripherals – Necessity of support circuitry.

2 **Software tools and support**: Development environment, assembler, compiler, evaluation kit, function libraries and other software solutions. These may come from the manufacturer or from third-party companies.

3 **Cost**

4 **Market availability**

5 **Processor maturity**

When evaluating the capacity of a microprocessor for a given application, the clock frequency is a good criterion to compare microprocessors in the same family, but should not be used to compare microprocessors from separate manufacturers, or even from separate families by the same manufacturer. For one thing, some microprocessors process complete instructions at each clock cycle, while others may only process part of an instruction. Furthermore the number and complexity of the basic operations that are contained into single instructions vary significantly from one microprocessor family to the next.

The instruction set and addressing modes are important criteria. Some microprocessors have very specialized instructions and addressing modes, designed to handle certain types of operations very efficiently. If these operations are critical for the application, such microprocessors may be a good choice.

The capability of the Arithmetic and Logic Unit (ALU) is usually a critical factor. In particular the number of bits (or resolution) of the numbers that can be handled by the ALU is important. If the application requires operations to be carried out in 32-bit precision, a 32-bit ALU may be able to do the operation in a single clock cycle. Using a 16-bit ALU, the operation has to be decomposed into several lower-resolution operations by software, and it may take as much as 8 times longer to execute. Another important factor is the ability of the ALU to carry out operations in fixed-point representation only (fixed-point processors), or in fixed-point and floating-point representations (floating-point processors). Fixed-point processors can perform floating-point calculations in software, but the time penalty is very high. Of course there is a trade-off between the capability of the ALU and other factors. A very wide (32 or even 64-bits) floating-point ALU accounts for a significant portion of the microprocessor's silicon area. It has a considerable impact on the cost and power consumption of the CPU, and may prohibit or limit the integration of on-chip peripherals.

Most embedded applications only require limited amounts of memory. This is why most microcontrollers and DSPs have limited address busses. For those few embedded applications that need to manage very large amounts of memory, the choice of a general-purpose microprocessor designed for computer systems and having a very wide address bus may be appropriate.

As mentioned earlier, by carefully choosing the clock frequency, the designer can directly adjust the power consumption. If power consumption is an important factor one avenue may be to choose a microprocessor designed for its low power. Another, often competitive, solution may be to choose a more complex and more powerful microprocessor, and have it work at a lower frequency. In fact the latter solution provides more flexibility because the system will be able to cope should unanticipated computational needs appear during the development.

The presence of on-chip peripherals is an important factor for embedded designs. It can contribute to lowering the power consumption, and the size of the system because external peripherals can be avoided. These factors are especially important in portable battery-operated devices.

The availability of good development and support tools is a critical factor. The availability of relevant software functions and solutions in particular can cut down the development time significantly and should be considered in the choice.

Cost is a function of the microprocessor's complexity, but it is also a function of the microprocessor's intended market. Microprocessors that are produced for mass markets are always less expensive (at equal complexity) than microprocessors produced in smaller quantities. Even if the application is not intended for a large volume, it may be good to choose a microprocessor that is, because the design will benefit from the lower

© BRUNO PAILLARD

cost of the microprocessor. On the other hand, if the application is intended for large volume it is highly advisable to make sure that the processor will be available in large enough quantities to support the production. Some microprocessors are not intended for very high-volume markets and, as with any other electronics component, availability can become a problem.

Finally the processor's maturity may be a factor in the choice. Newer processors are often not completely functional in their first release. It can take as long as a year or two to discover and "iron-out" the early silicon bugs. In this case the designer has to balance the need for performance with the risk of working with a newer microprocessor.

5. BUILDING BLOCKS OF AN EMBEDDED SYSTEM

Figure 2-1 describes the architecture of a typical microprocessor system. It is worth noting that this generic architecture fits computer systems such as PCs, as well as embedded systems such as a microwave oven controllers or a musical cards.

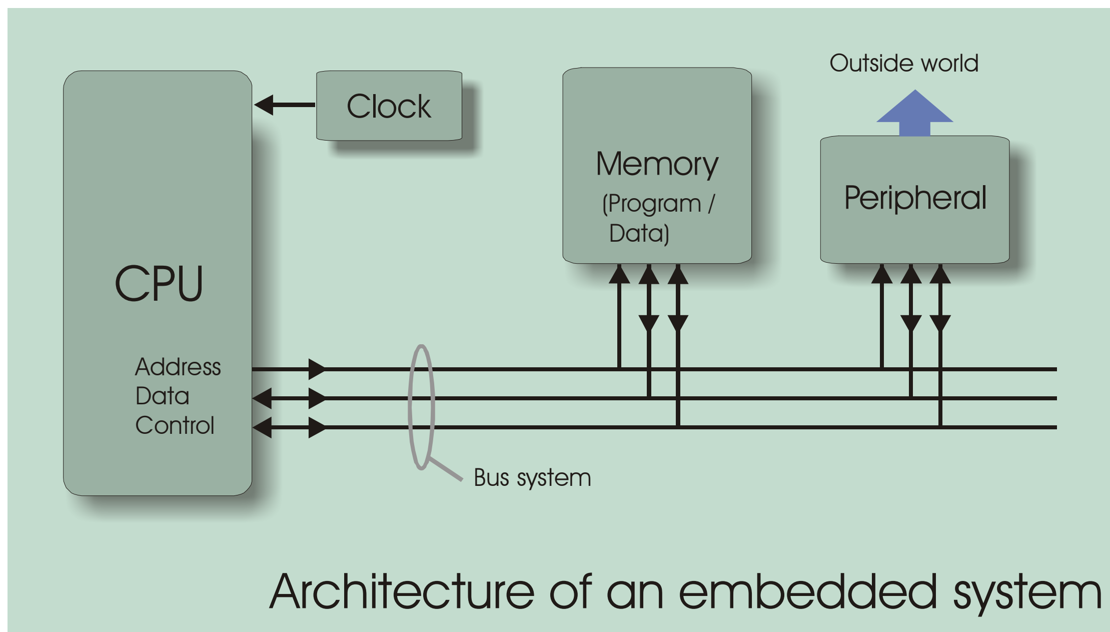


Figure 2-1

<u>Microprocessor (CPU):</u>

The CPU is a sequential logic machine that reads instructions in memory and executes them one after the other. A clock sequences the various operations performed by the CPU.

The CPU:

- **Executes** the instructions found in memory.
- **Performs** the calculations and data processing operations specified by the instructions.
- **Initiates** data exchanges with peripherals (memory, parallel ports, …etc.)

Before the development of the first microprocessor in 1971 the multi-circuit structure that used to perform this role in early computers was called Central Processing Unit (CPU). Today the words CPU and microprocessor are often used interchangeably. However, the term microprocessor is generally reserved to CPUs that are completely integrated on a single silicon chip. For microcontrollers, which also include peripherals on the same silicon chip, the term CPU describes exclusively the part of the device that executes the program. It excludes the peripherals.

Clock:

The clock circuit is usually implemented by a quartz oscillator. The oscillator may be part of the microprocessor, or may be external. The quartz crystal itself is always external. In low cost designs where frequency stability is not an issue, a low-cost ceramic resonator may replace the quartz. The clock sequences all the operations that are performed by the CPU. The signal from the oscillator may be divided-down by programmable dividers, or it may be multiplied by an adjustable factor using a Phase-Locked-Loop (PLL). Dividing or multiplying the clock frequency provides a way to dynamically adjust the computational speed and the power consumption of the microprocessor. Multiplying the oscillator frequency is also essential in modern designs because it is very difficult (and therefore expensive) to produce stable quartz resonators at frequencies above 100MHz. The PLL therefore enables the use of a lower frequency quartz resonator to produce the high clock frequencies required by today's microprocessors.

Memory:

Memory circuits are used to:

- **Store** program instructions.
- **Store** data words (constants and variables) that are used by the program.
- **Exchange** these data words with the CPU.

Peripherals:

Peripherals provide services to the CPU, or provide a connexion to the outside world. Memory circuits are a special case of peripherals. Any electronic circuit connected to the CPU by its bus system is considered to be a peripheral.

Bus system:

The bus system is the network of connexions between the CPU and the peripherals. It allows instructions and data words to be exchanged between the CPU and its various peripherals.

The following example describes the use of a microprocessor system in a balance control application (figures 2-2 and 2-3).

© BRUNO PAILLARD

Figure 2-2 *"Bikeman at DSPWorld 1997" Photo: Cory Roy*

The microprocessor used in this application is a TMS320C50 from Texas Instruments.

The system has 3 inputs that are brought to the CPU via 3 peripherals:

- A Pulse Width Modulation (PWM) signal follows the position of a radio-control joystick. The width of the pulses transmitted by the radio-control system represent the trajectory input of the pilot. The signal is transmitted to a peripheral called a *timer* that measures the width of its pulses, and provides this information to the CPU.
- A vibrating gyrometer is placed in the head of the cyclist. It provides an analog signal proportional to the rate of roll of the bicycle around the axis passing through the contact points of the wheels. This analog signal is measured by a peripheral called an Analog to Digital converter that provides this information to the CPU.
- An optical sensor placed in the gear train of the bicycle provides a square wave with a frequency proportional to the speed of the bicycle. This frequency is measured and provided to the CPU by another timer.

The system has one output that is generated by the CPU:

- A PWM signal is generated by a third timer, and sent to the servomotor that controls the position of the handlebars.

The system works as follows:

- Thirty times a second the gyrometer output is measured by the Analog to Digital Converter and is sent to the CPU.
- The CPU integrates this rate of roll signal to produce the absolute angle formed by the bicycle frame and the road.
- The CPU calculates the position that the handlebars should take to counteract any unwanted variation of this angle (to stabilize the angle), and to properly achieve the pilot's trajectory input. If the pilot wants to go in a straight line for instance, and the bicycle is falling to the right, the CPU temporarily turns the handlebars to the right to put the bicycle into a right trajectory. The centrifugal force due to this trajectory then stops the fall to the right and swings the bicycle back to a straight position. When the bicycle is straight, the handlebars are repositioned in the middle to maintain this trajectory. If the pilot wants to turn left the CPU temporarily turns the handlebars to the right. The centrifugal force due to the right trajectory puts the bicycle in a left fall. Once the bicycle has attained the correct left trajectory, the CPU places the handlebars in an intermediate left position to maintain this trajectory. Finally when the pilot wants to get out of the left turn, the CPU temporarily increases the angle of the handlebars to the left. The centrifugal force swings the bicycle back to the right. When the bicycle reaches a vertical angle, the handlebars are put back in the center to provide a stable straight trajectory.
  Because the system is naturally unstable, very little force has to be exerted on the handlebars to control balance and trajectory. By perfectly controlling the position of the handlebars, the system uses the centrifugal force to do the hard work.
- The position of the handlebars that is calculated by the CPU is sent to a timer that generates the PWM signal that drives the servomotor.
- The speed of the bicycle is measured to adapt the reaction of the handlebars to the speed of the bicycle. When the bicycle is going fast, the reaction on the handlebars is small. When it is going slowly, the amplitude of the reaction increases.

For such a system an automatic balance and trajectory control system is essential, because the bicycle is small and its reaction time is so short that no human pilot could react in time.

The simplified architecture of the system is as follows:

© BRUNO PAILLARD

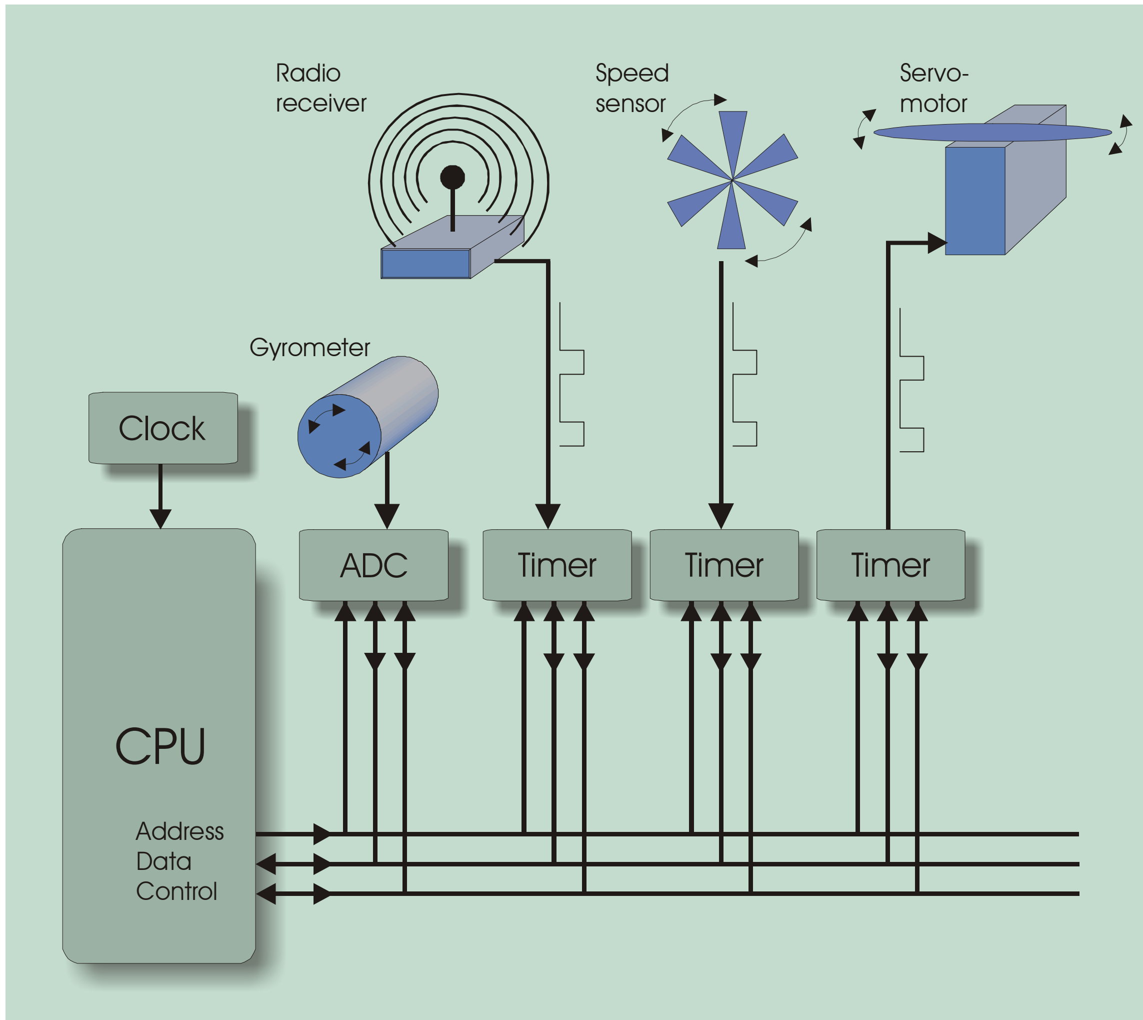


Figure 2-3

6. PROGRAM EXECUTION

The CPU executes instructions that are stored in memory. These instructions describe elementary operations, such as reading a peripheral, adding two values, or writing a result back into memory. The sequence of many instructions constitutes a program.

Its clock sequences the CPU operations. At each clock cycle, the CPU carries out specific elementary operations that lead to the execution of the instructions. Two phases can be identified in the operation of the CPU:

1. **The Fetch Cycle**:
   During the fetch cycle the CPU reads the instruction word (or words) from the memory. The fetch cycle is always a read from memory.
2. **The Execute Cycle**:
   During the execute cycle the CPU carries out the operation indicated by the instruction. This operation can be a read from memory or from a peripheral, a

write to memory or to a peripheral, or can be an internal arithmetic or logic operation that does not require any exchange with a peripheral.

These two phases do not always take the same time. The length of the fetch cycle varies with the number of instruction words that must be fetched from memory. For many microprocessors, the length of the execute cycle varies with the complexity of the operation represented by the instruction. However, from the time of its reset on, the CPU always alternates between fetch cycle and execute cycle.

Some microprocessors are able to do both cycles at once. They fetch the next instruction while they execute the present one.

© BRUNO PAILLARD

System architecture

- 1. Von Neumann architecture
- 2. Harvard architecture
- 3. Pros and cons of each architecture
- 4. Busses, address decoding and three-state logic
- 5. Memories: technology and operation

The system's architecture represents the way that the elements of the system are interconnected and exchange data. It has a significant influence on the way the CPU accesses its peripherals. Every type of architecture requires special hardware features from the CPU (in particular from its bus system). Every microprocessor is therefore designed to work in a specific type of architecture and cannot usually work in other types.

Every type of architecture includes:

- At least one CPU
- At least one peripheral
- At least one bus system connecting the CPU to the peripherals.

1. VON NEUMANN ARCHITECTURE

In 1946, well before the development of the first microprocessor, John Von Neumann developed the first computer architecture that allowed the computer to be programmed by codes residing in memory. This development took place at University of Pennsylvania's Moore school of Electrical Engineering, while he was working on the ENIAC – one of the first electronic computers. Prior to this development, modifying the program of a computer required making modifications to its wiring. In the ENIAC, program instructions were stored using an array of switches, which represented an early form of Read Only Memory. The Von Neumann architecture is the most widely used today, and is implemented by the majority of microprocessors on the market. It is described in Figure 3-1. For this architecture, all the elements of the computer are interconnected by a single system of 3 busses:

- **The Data Bus** Transports data between the CPU and its peripherals. It is bi-directional. The CPU can read or write data in the peripherals.
- **The Address Bus** The CPU uses the address bus to indicate which peripherals it wants to access, and within each peripheral which specific register. The address bus is unidirectional The CPU always writes the address, which is read by the peripherals.

- **Control Bus** This bus carries signals that are used to manage and synchronize the exchanges between the CPU and its peripherals. For instance the signal that indicates if the CPU wants to read or write the peripheral, as well as the signal used to synchronize the exchange are part of the control bus. Since the CPU initiates and manages all data exchanges with the peripherals, the main signals of the control bus originate at the CPU and are sent to the peripherals. However special lines of the control bus, such as interrupt or wait-state signals carry information from the peripherals to the CPU.

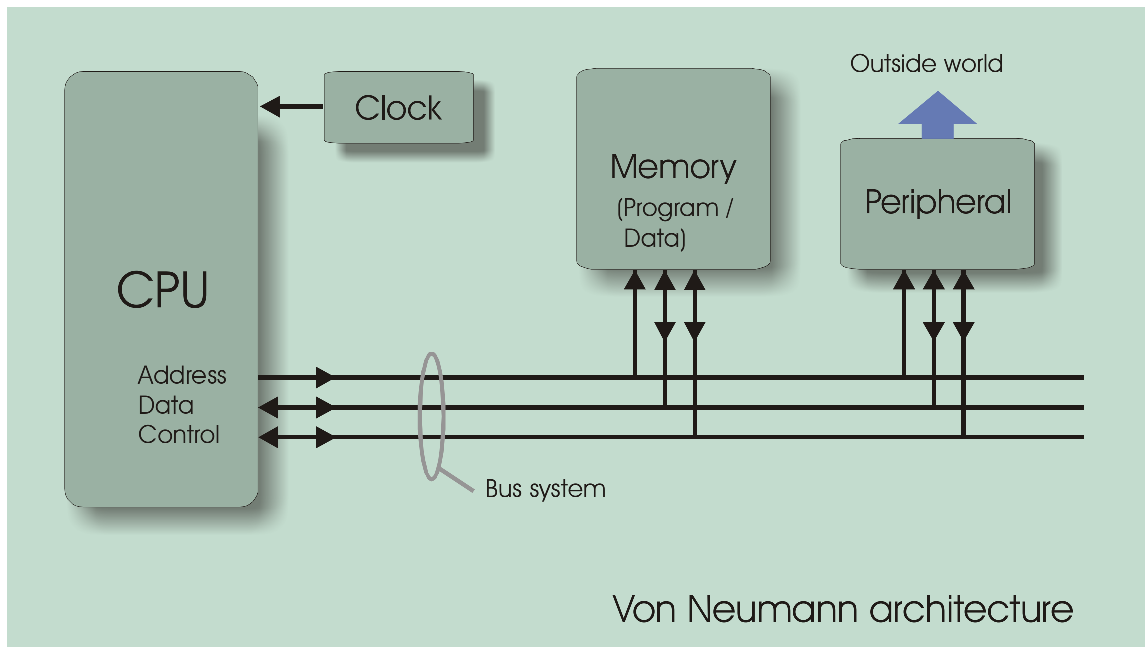


Figure 3-1

The main characteristic of the Von Neumann architecture is that it only possesses 1 bus system. The same bus carries all the information exchanged between the CPU and the peripherals, including the instruction codes as well as the data processed by the CPU.

2. HARVARD ARCHITECTURE

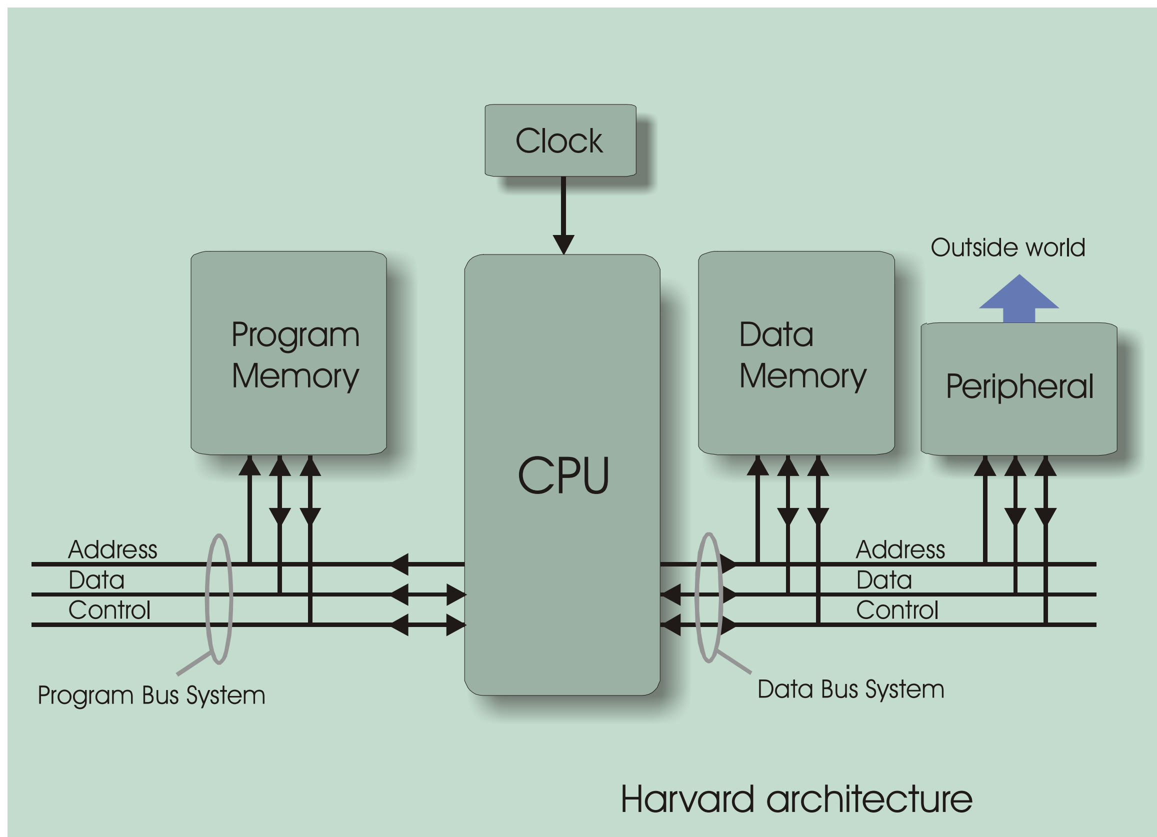


Figure 3-2

The Harvard architecture – as the name implies – was developed at Harvard University. By contrast to the Von Neumann architecture, it uses two separate bus systems to transport the instruction codes and the data being processed by the CPU.

- **The Program Bus System:** Is used exclusively to transport instruction codes from the program memory to the CPU during the fetch cycle.
- **The Data Bus System:** Is used exclusively to transport data from/to the CPU, to/from the memory and peripherals.

***Note**: One word of caution about a confusing terminology is in order: The Program Bus System and the Data Bus System are complete bus systems composed of an address bus, a data bus and a control bus. Very often the Program Bus System and Data Bus System are simply called Program Bus and Data Bus. When discussing a "data bus", it is therefore important to put it in context and determine if it applies to a data bus system comprising an address bus, a data bus and a control bus, or if it is simply a data bus.*

3. PROS AND CONS OF EACH ARCHITECTURE

- Since it possesses two independent bus systems, the Harvard architecture is capable of simultaneously reading an instruction code, and reading or writing

a memory or peripheral as part of the execution of the previous instruction. It has a speed advantage over the Von Neumann architecture.

- The Harvard architecture is also safer, since it is not possible for the CPU to mistakenly write codes into the program memory and therefore corrupt the code while it is executing.
- However, the Harvard architecture is less flexible. It needs two independent memory banks (one for program and another one for data). These two resources are not interchangeable. For an embedded system that always runs the same application, this is not a significant problem since the program and data memory needs can be easily anticipated and the required memory resources can be optimized for the application. A computer system however may run a wide variety of applications, some requiring large program memories, and others requiring large data memories. The Von Neumann architecture is better suited for this type situation, because program and data memories are interchangeable, and it will lead to a better usage of the memory resources.

Over the years, these architectures have been enhanced to provide more performance. For instance, an *enhanced Harvard* architecture may have separate bus systems for program memory, data memory, and input/output peripherals. It may also have multiple bus systems for program memory alone, or for data memory alone. These multiple bus systems increase the complexity of the CPU, but allow it to access several memory locations simultaneously, thereby increasing the data throughput between memory and CPU.

4. BUSSES, ADDRESS DECODING AND THREE-STATE LOGIC

In all architectures, it is always the CPU that initiates exchanges with the peripherals. Data is always exchanged between the CPU and a peripheral. It is never exchanged directly between 2 peripherals. For this reason, data transfers are always described from the CPU's point of view.

Definition:	**Read operation:**	The CPU reads a register in a peripheral.

Definition:	**Write operation:**	The CPU writes a register in a peripheral.

4.1. Read and Write operations

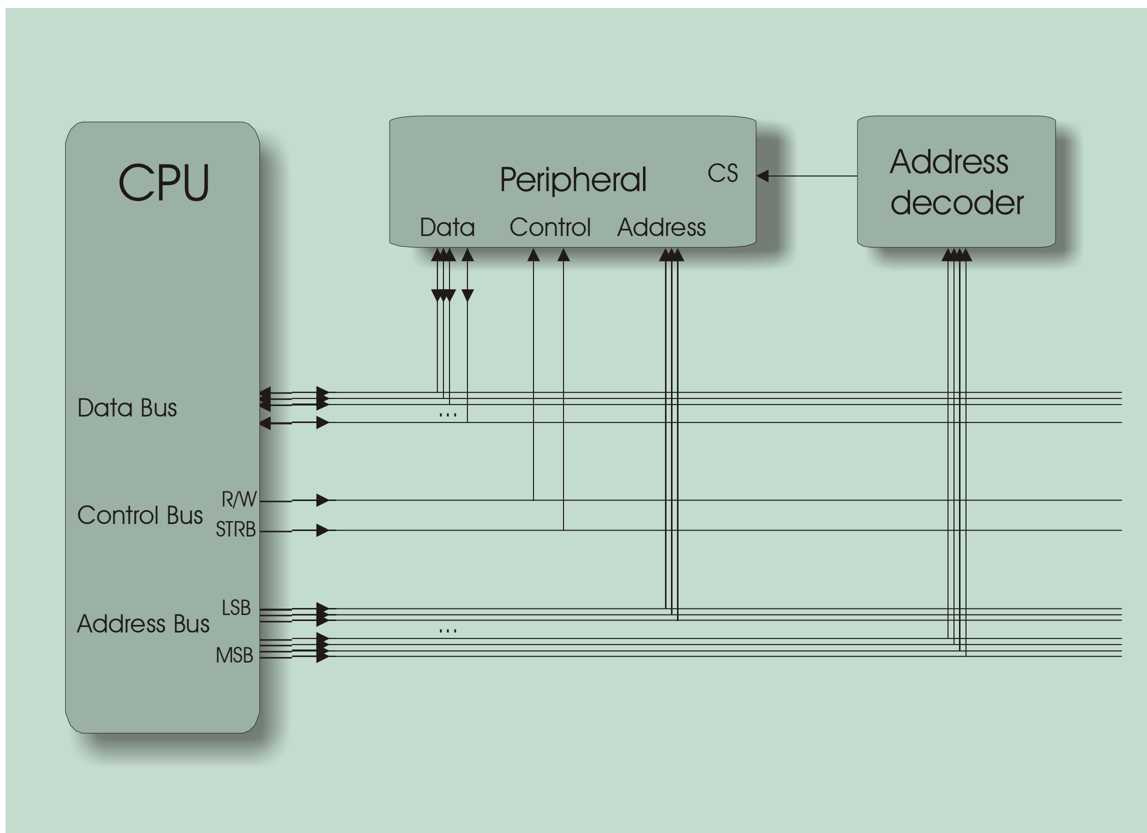


Figure 3-3

Figure 3-3 represents the connexion pattern between the CPU and a peripheral. The peripheral could be a memory, a parallel port or any other type.

Definition:	**Interface:** The way the CPU is interconnected to a peripheral is also called an *interface*.

Definition:	**Bus width:** The width of a bus corresponds to the number of bits that constitute the bus. Each bit is transported by an individual track on the printed-circuit-board.

Definition:	**N-Bit microprocessor:** A microprocessor is designated "N-bit microprocessor" if it is able to process N-bit-wide data words in a single operation. Normally, it also represents the width of its data bus, because it allows it to read and write words N bits at a time in the peripherals. The wider the data bus the more complex the Arithmetic and Logic Unit inside the CPU, and the more efficient the CPU is to process large data words. Many small microcontrollers are only 8-bit processors. Their data bus is therefore 8-bits wide. Intermediate performance processors are 16-bit processors. For instance the TMS320VC5402 is in this category. Its data bus is 16-bits wide. High-performance general-purpose microprocessors are often 32, or even 64 bits microprocessors.

Definition:	**Register:** A register is an array of N flip-flops that store information within a peripheral, or which outputs drive some functions of the peripheral. Each flip-flop stores one bit of information. When the CPU accesses a register, each flip-flop in the register is connected to the corresponding bit in the data bus. Each data transfer between the CPU and the peripheral is therefore done N-bits (the whole register) at a time. It is not possible for the CPU to access individual bits (or flip-flops) in a peripheral's register. All peripherals include at least one register, but many include more. A memory obviously contains a lot of registers (usually many thousand). Each register represents a **word** of data.

Usually the peripherals are "adapted" to the CPU. For instance a 16-bits CPU is usually interfaced to memories having 16-bit registers. However, this is not always the case, and peripherals can be interfaced to a CPU even though their registers have fewer bits. For instance, an 8-bit peripheral can be interfaced to a 16-bit CPU. In that case, every time the CPU writes to the peripheral, only 8 of the 16 bits being written (usually the lower 8 bits) actually go to the peripheral's register. The 8 other bits are not connected and are of no consequence to the peripheral. When the CPU reads the peripheral, only 8 bits of the word being read are actually updated by the peripheral. The 8 other bits are random and should be discarded by the CPU.

***Note**: The TMS320VC5402, which is the subject of the following chapters, is a 16-bit DSP. Therefore it has a 16-bit data bus. It also has a 16-bit address bus.*

Figures 3-4 and 3-5 show idealized timings for the read and write cycles in the data space for a TMS320VC5402 processor. They do not show data *setup* and *hold* timings for the signals that are described. For the sake of clarity, these chronograms do not show the effect of Wait-States, and they do not show the data-space select signal. These chronograms apply to accesses to memory circuits, as well as any other peripheral interfaced in the data space of the CPU.

4.1.1.Read Cycle

A complete read cycle takes only one CPU clock period. For a TMS320VC5402 running at 100MHz, a clock period lasts for 10 nano-seconds. It is performed as followed:

At the beginning of the cycle the data bus is assumed to be in a high-impedance state (see following sections).

- On the falling edge of the CPU clock, the CPU presents a binary code on the address bus (A0-A15). This code specifies the particular register being accessed among all the registers of all the peripherals present in the system. This code is called an **address**. The address carries two types of information:
  - It carries the identity of the peripheral being accessed among all the peripherals present in the system. By a process called **Address Decoding**, the specific peripheral being accessed is recognized by a portion of the address code, and selected for the exchange. Usually, the upper bits of the address code carry the identity of the peripheral.
  - The address also carries the identity of the register being accessed within this specific peripheral. Usually the lower bits of the address code carry this information. These bits are directly read by the selected peripheral, which internally selects the corresponding register for the transaction.

- At the same time, the CPU presents the direction of the access (1=Read) on the R/W line of the control bus.
- At the same time, the low state on the STRB (strobe) signal directs the peripheral to put the data word to be read on the lines of the data bus. It usually takes a few nano-seconds for the peripheral to actually drive the bus, and for the logic levels to stabilize.
- At the end of the cycle, on the next falling edge of the clock, the rising edge of the STRB signal indicates the capture of the data word by the CPU. At this time the data word is well established and stable on the bus.
- The rising edge of the STRB signal also directs the peripheral to relinquish the bus (put it back to a high-impedance state). It usually takes a few nano-seconds for the bus to actually go back to high-impedance.

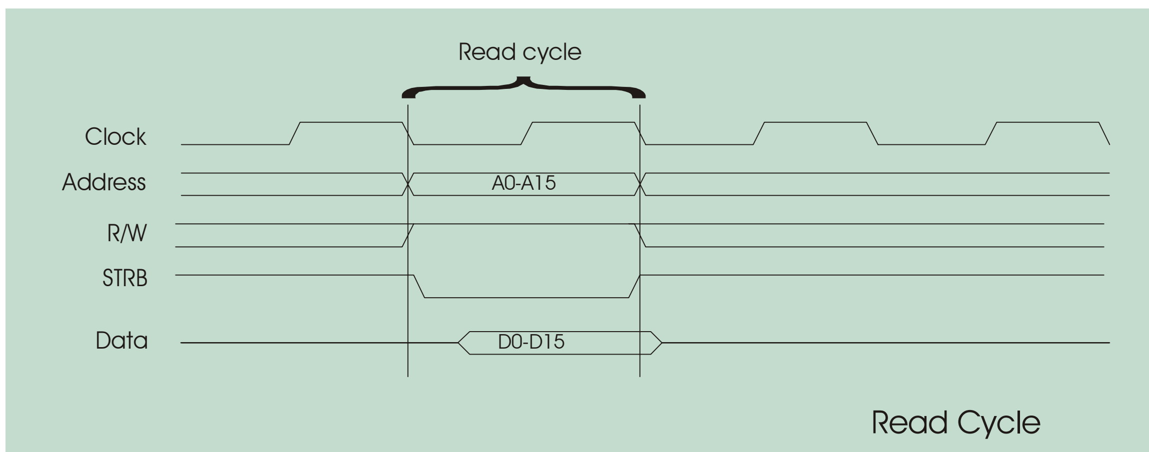


Figure 3-4

4.1.2. Write Cycle

On the TMS320VC5402 the write cycle takes a little bit longer. It normally takes 3 clock cycles, but can be shortened to 2 clock cycles when back-to-back writes are being performed.

At the beginning of the cycle the data bus is assumed to be in a high-impedance state (see following sections).

- On the falling edge of the CPU clock, the CPU presents the address (A0-A15) on the address bus. As is the case with a read, this address is used to select a specific peripheral in the system, and select a specific register in this peripheral.
- Half a period later, the CPU presents the direction of the access (0=Write) on the R/W line of the control bus.
- Half a period later, the low state on the STRB (strobe) signal indicates to the peripheral that the CPU has placed its data bus into low-impedance output, and is driving the data word on the lines of the data bus.
- One clock cycle later, the rising edge on the STRB signal directs the peripheral to capture the data present on the data bus. At this time the data word is well established and stable on the bus.

- The data word is held on the bus for half a cycle, to insure that levels do not change during the transition, while the peripheral is capturing the data word.
- The cycle formally ends half a cycle later, on the falling edge of the clock signal.

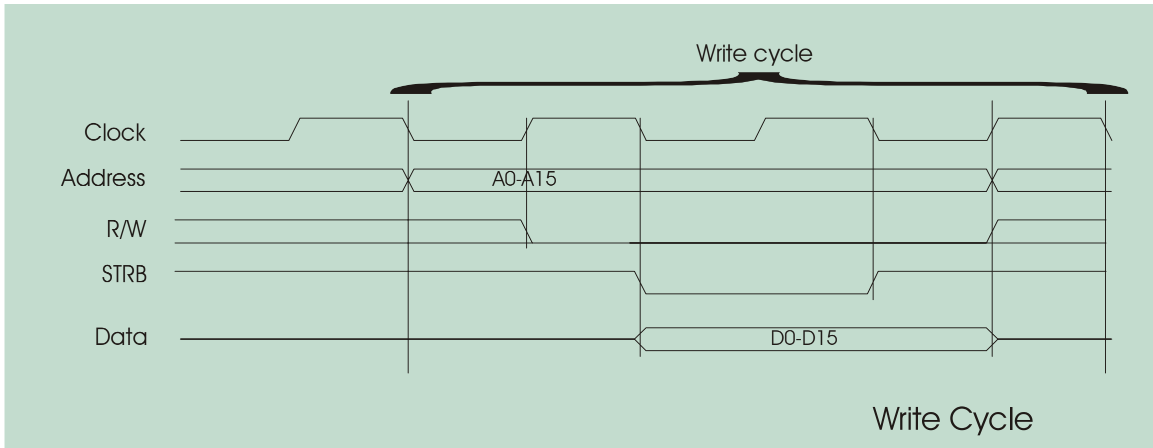


Figure 3-5

Note: *For the TMS320VC5402, the preceding chronograms only apply to accesses to off-chip memory devices or peripherals. Accesses to on-chip RAM can be performed much faster. Accesses to the on-chip "Dual-Access RAM" in particular can be performed at a rate of two accesses per clock cycle. Furthermore, they only apply to accesses in the Program and Data spaces of the DSP. Accesses to the I/O space of the DSP actually differ slightly from those to the Program and Data spaces. For more detailed information about external accesses, the reader should refer to the following documents from Texas Instruments: SPRU131f – TMS320C54x DSP Reference set – Volume 1: CPU and Peripherals; and SPRS079e – TMS320VC5402 Fixed Point Digital Signal Processor.*

4.2. Address bus - Address decoding

To access a register within a peripheral, the CPU must indicate the identity of the peripheral, and the identity of the register to be accessed within the peripheral. This process uses a single address code, which transports both pieces of information. A process called **address decoding** allows the peripheral to be selected from a portion of the address code.

Definition:	**Address decoding:** Process by which a specific peripheral is selected from a portion of the address code present on the address bus.

Definition:	**Addressing Capacity:** For an M-bit address bus, the CPU can distinguish between 2^M individual address codes. Each individual address may or may not correspond to a register in a peripheral. However, the total of all 2^M addresses codes is called the Addressing Capacity of the processor. Processors with wider address busses have a higher addressing capacity, and can manage larger amounts of memory and peripherals.

 © BRUNO PAILLARD

In practice in any given system, every address code does not correspond to a register of a peripheral. Of all the possible address codes, usually large numbers do not select any peripheral. If the CPU tries to read from an address that does not select a peripheral, the data bus stays in a high-impedance state during the read, and the word read by the CPU is random. If the CPU tries to write to an address that does not select a peripheral, the data written does not end-up being captured by any peripheral. Although such operations are meaningless, they do not cause any hardware problems.

On the other hand, in some systems several addresses may select the same register in the same peripheral. These are called **aliases**. Having aliases does not serve a particular purpose. It is usually done to simplify the address decoding logic. In such a case, the software designer will simply choose one address among all the aliases to access the particular register within the particular peripheral.

Definition: **Memory map:** The memory map, or **addressing map**, represents graphically a map of all the possible addresses. The map shows the address zones that select the various peripherals present in the system (not just the memory circuits). It may also show address zones that present a specific interest to the software designer. The memory map is a document produced by the hardware designer, and is used by the software designer.

Definition: **Address space:** Harvard architectures have several bus systems to exchange data between various peripherals and the CPU. Each bus system gives access to a specific set of peripherals that usually cannot be accessed using another bus. The same address may select a register in a program memory on the program bus system for instance, and may select an unrelated peripheral on the data bus system. Each bus system gives access to a specific address space. For instance, the TMS320VC5402 has 3 address spaces: The program space in which resides the program memory; the data space in which reside memories used for storing variables, as well as some peripherals; the I/O space in which may reside additional input/output peripherals. Each address space has its own memory map.

Usually a memory map does not show individual registers but rather, it indicates the limits of various zones occupied by peripherals, or zones that are of particular interest to the software designer.

Figure 3-6 represents the memory map of the Signal Ranger board. The CPU of this board is a TMS320VC5402, which has a program space, a data space, and an I/O space. The I/O space is not represented because it contains no peripherals.

This memory map does not distinguish between “on-chip” and “off-chip” peripherals. One particularity of the TMS320VC5402 is that the same “on-chip” RAM circuit (addresses 0080_H to $3FFF_H$) can appear in both the program space and the data space. This RAM circuit is decoded in both spaces. Therefore it can be used to store data and variables, as well as instruction codes. However, the “off-chip” memory present at addresses 4000_H to $FFFF_H$ is only decoded in the data space. Therefore instruction codes cannot be executed from this memory.

The sections marked “Vectors”, “Kernel” and “MailBox” are memory sections that are occupied by a piece of software called “communication kernel”. The communication kernel enables communications between the PC and the DSP board. The corruption of

these sections with other data or code would disable communications until the next reset of the board.

The Read Only Memory called “BootLoader” in the program space contains code that is executed at the reset of the CPU. It is this bootloader code that allows the PC to load and execute the communication kernel.

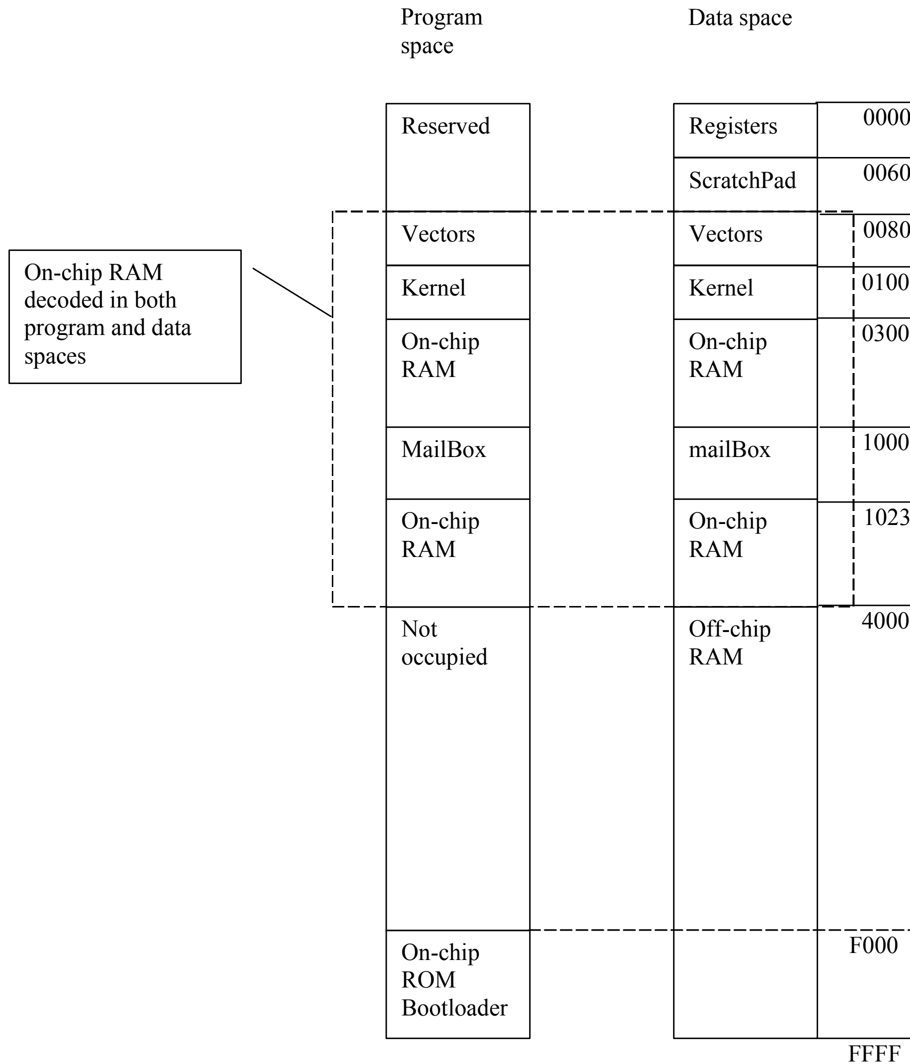


Figure 3-6

4.2.1. How address decoding works

The registers that the CPU may access are physically located within various memory or peripheral circuits. The CPU selects a specific register within a specific peripheral by writing an address code on the address bus. One portion of this code (usually the upper bits) specifies the peripheral, and another portion (usually the lower bits) specifies the register within the peripheral.

For a peripheral with 2^L registers (generally the number of registers within a peripheral is a power of 2), it takes L bits to select a particular register from the 2^L possible choices. Such a peripheral would have L address inputs that are used for this purpose. These L

© BRUNO PAILLARD

inputs are generally connected to the L lower bits of the CPU's address bus. When accessing this peripheral, the L lower address bits indicated by the CPU are used to select one register among the 2^L that the peripheral contains.

Additional circuits called **Address Decoders** are used to select specific peripherals from all the peripherals present in the system. An address decoder is a combinatory circuit that "observes" a certain number of upper bits of the address bus, and selects a specific peripheral when it recognizes a unique bit pattern. The peripheral is selected by activating its "Chip-Select" (CS) or "Chip-Enable" (CE) input. Every peripheral in the system has its own address decoder, which activates its chip-select input on a unique bit pattern of the upper address bits.

Figure 3-7 shows a possible interface between a CPU and two peripherals.

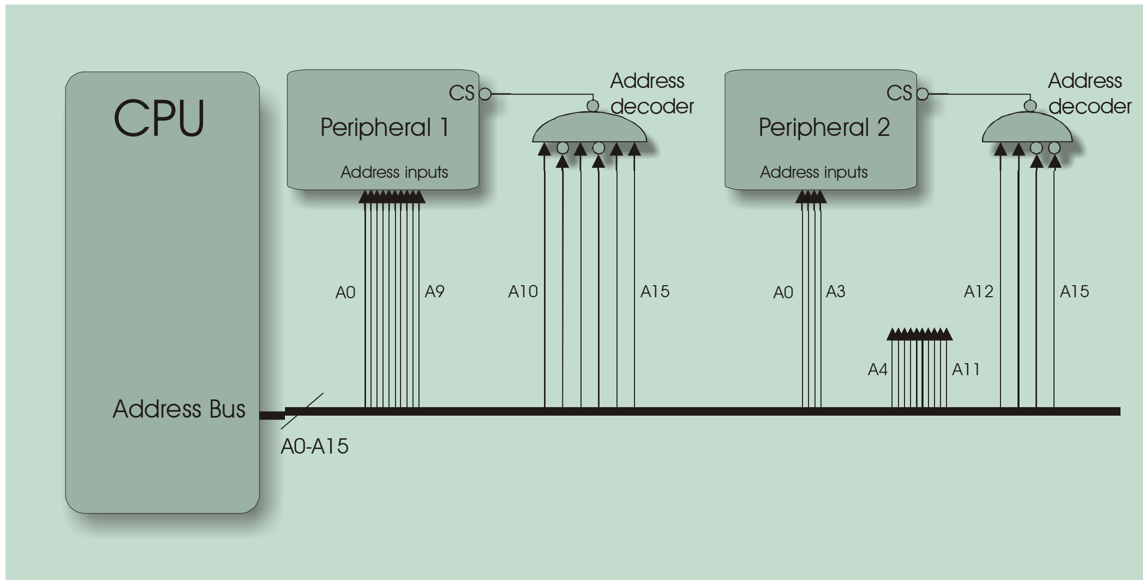


Figure 3-7

- **Peripheral No 1:**

  Bits A0 to A9 of the address bus are used to select a specific register within the peripheral. From this, we can infer that the peripheral has a maximum of 2^{10} = 1024 internal registers.

  The remainder of the address bits (bits A10 to A15) are used to select the peripheral. A specific combination of these bits (110101) uniquely selects this peripheral.

  The decoder recognizes all addresses of the form:

  1101 01xx xxxx $xxxx_b$.

  The first address of the zone that selects the peripheral (the zone "occupied" by the peripheral) in the memory map is obtained by replacing all x by 0s. The last address is obtained by replacing all x by 1s. The zone occupied by the peripheral goes from:

  1101 0100 0000 0000_b ($D400_H$) to

  1101 0111 1111 1111_b ($D7FF_H$).

In this case all the bits of the address bus are used either to select the peripheral, or to select a register within the peripheral. This is called a **complete decoding**.

- **Peripheral No 2:**

  Bits A0 to A3 of the address bus are used to select a register. This peripheral must therefore have a maximum of 2^4 = 16 registers.

  Bits A12 to A15 are used to select the peripheral through its address decoder. A specific combination of these bits (0011) uniquely selects this peripheral.

  The decoder for peripheral 2 recognizes all addresses of the form:

  0011 xxxx xxxx $xxxx_b$.

  The zone occupied by the peripheral in the memory map goes from:

  0011 0000 0000 0000_b (3000_H) to

  0011 1111 1111 1111_b ($3FFF_H$).

  In this case, bits A4 to A11 of the address bus are not used to select the peripheral or to select a register within the peripheral. These bits are of no consequence to the decoding process. This is called a **partial decoding**.

In the case of a partial decoding, every bit that is not used in the selection of the peripheral, or in the selection of a register within the peripheral, can take any value without influencing the decoding process. All these unused bits lead to **equivalent addresses**, or **aliases** that can equally select the same register of the same peripheral. In figure 3-7, since there are 8 unused bits in the decoding of peripheral No2, for each register in the peripheral, there are 2^8 = 256 separate addresses that can select this register.

The zone occupied by peripheral No 2 (from 3000_H to $3FFF_H$) is 4096 addresses long. It is actually composed of 256 consecutive alias zones (one alias zone for each combination of the 8 unused bits) of 16 addresses each. The first register of the peripheral can equivalently be accessed at addresses 3000_H, 3010_H, 3020_H…etc. The second register can equivalently be accessed at addresses 3001_H, 3011_H, 3021_H…etc.

Partial decoding has no purpose in itself. It is done only to simplify the logic of the address decoding circuit. Indeed a complete decoding for peripheral No2 would have required a combinatory circuit with 12 inputs, which may be more expensive than the 4-input decoder used instead. Partial decoding wastes space in the memory map because the same registers of the same peripherals occupy many addresses in the map. However, if there is a lot of address space to spare in the system (which is often the case in embedded designs), this solution may be perfectly acceptable.

Note: *As can be seen from the preceding example, the process of address decoding is such that the first address of a zone occupied by a peripheral always has its lower address bits at 0. The last address always has its lower bits at 1. Therefore the lower boundary of the zone occupied by any peripheral in the memory map always lies at an address that is a power of two. The upper boundary always lies at an address that is a power of two minus one.*

Note: *In principle it would be possible for a particular combination of upper address bits to select more than one peripheral. In practice this situation would lead to a conflict between the selected peripherals. It is the responsibility of the hardware designer to ensure that this situation does not happen. This situation could easily be observed in the memory map by an overlapping of the address zones of the offending peripherals.*

© BRUNO PAILLARD

Note: *The Chip-Select inputs of peripherals are usually active low. This convention comes from the fact that in TTL logic –early microprocessor systems were implemented in TTL technology - a floating input is naturally pulled to a logic high. In TTL, the high level was therefore usually defined as the inactive one. Nowadays most logic is usually implemented in CMOS technology, for which floating inputs do not default to a logic high. However the convention stayed.*

4.2.2. Technological aspects of address decoding

An address decoder is a combinatory circuit that works as follows:

- The circuit "observes" certain upper bits of the address bus.
- When it recognizes a specific code on those bits, it activates its output that is connected to the chip-select input of the corresponding peripheral. By proper wiring, or a proper programming of the decoder, the hardware designer sets the specific code recognized by each decoder in the system.

In practice two technologies may be employed to implement address decoders:

- The decoder may be implemented using off-the-shelf combinatory logic circuits. For this solution decoders/demultiplexers are usually preferred to off-the shelf NAND gates because the latter do not give the designer much flexibility in choosing the code that activates the peripheral.

  Circuits such as the 74HC138 (see figure 3-9) may be used for this purpose.

- The decoder may also be implemented using programmable logic devices (PLDs or CPLDs).

  Programmable logic devices may be a little more expensive than decoders/demultiplexers (although some of them are particularly inexpensive nowadays), and may draw more power. Power consumption is often a bigger issue than cost. However, they present the following advantages:

  - More inputs are usually available, which facilitates the design of complete decoders.
  - They are programmable, and the address zones for each peripheral may be modified during the course of the design, or upgrade of the system, without having to modify the printed circuit board.
  - If complex enough circuits are used, it may be possible to integrate other interface logic that may be required in a particular design. This additional interface logic is often called **glue logic**.

In either case more than one decoder may generally be fitted in one physical circuit.

The time delay between the presentation of the address and the selection of the peripheral is usually a critical factor in the timing of the access. Because of this address decoders are usually not constructed using large combinations of off-the-shelf gates. Figure 3-8 shows an example of such a bad practice for peripheral No1. In this example, the time delay between the presentation of the address and the activation of the peripheral is determined by the longest path (5 gates) between any address bit and the chip-select.

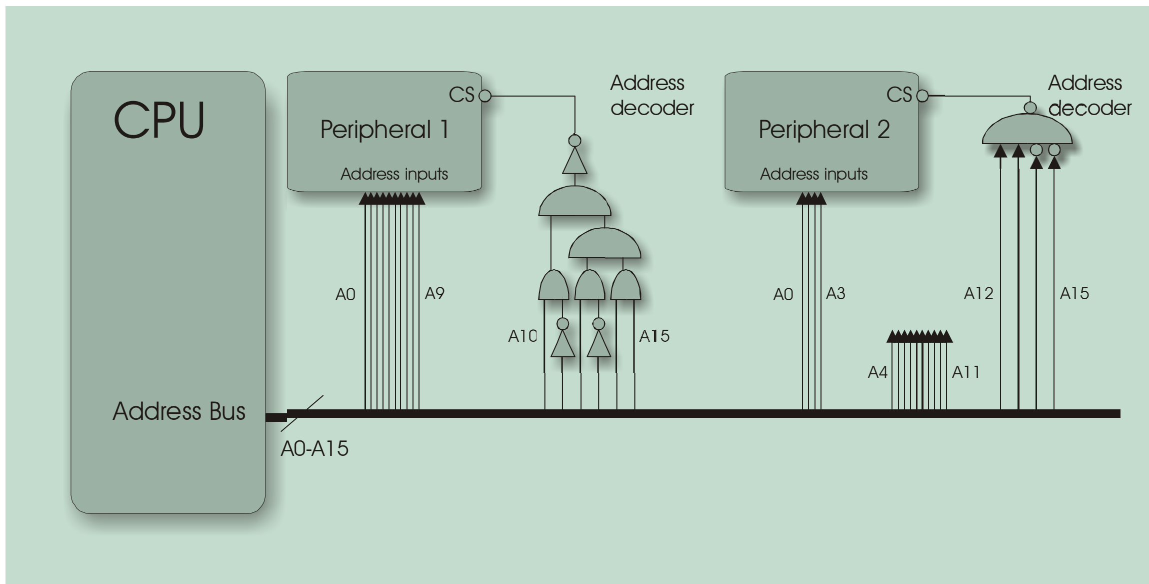


Figure 3-8

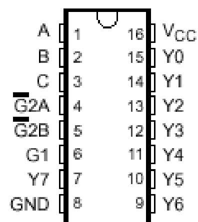
Figures 3-9A and 3-9B from Texas Instruments show the data sheets of a classic decoder/demultiplexer: The 74HC138. A decoder/demultiplexer has N inputs and 2^N outputs. It activates the output whose number corresponds to the binary code on its inputs. By carefully choosing to which address bits the inputs are connected, and which of the outputs selects the peripheral, the designer can choose precisely the address zone occupied by the peripheral. The 74HC138 has 3 inputs and 8 outputs.

Figure 3-10 show how it is used in a system, decoding addresses of the form:

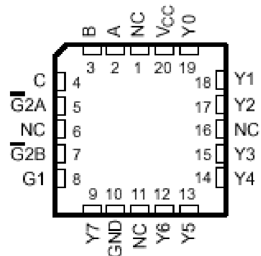
010x xxxx xxxx $xxxx_b$ (4000_H to $5FFF_H$)

- **Designed Specifically for High-Speed Memory Decoders and Data Transmission Systems**
- **Incorporate Three Enable Inputs to Simplify Cascading and/or Data Reception**
- **Package Options Include Plastic Small-Outline (D), Thin Shrink Small-Outline (PW), and Ceramic Flat (W) Packages, Ceramic Chip Carriers (FK), and Standard Plastic (N) and Ceramic (J) 300-mil DIPs**

SN54HC138 . . . J OR W PACKAGE
SN74HC138 . . . D, N, OR PW PACKAGE
(TOP VIEW)



SN54HC138 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

description

The 'HC138 are designed to be used in high-performance memory-decoding or data-routing applications requiring very short propagation delay times. In high-performance memory systems, these decoders can be used to minimize the effects of system decoding. When employed with high-speed memories utilizing a fast enable circuit, the delay times of these decoders and the enable time of the memory are usually less than the typical access time of the memory. This means that the effective system delay introduced by the decoders is negligible.

The conditions at the binary-select inputs at the three enable inputs select one of eight output lines. Two active-low and one active-high enable inputs reduce the need for external gates or inverters when expanding. A 24-line decoder can be implemented without external inverters and a 32-line decoder requires only one inverter. An enable input can be used as a data input for demultiplexing applications.

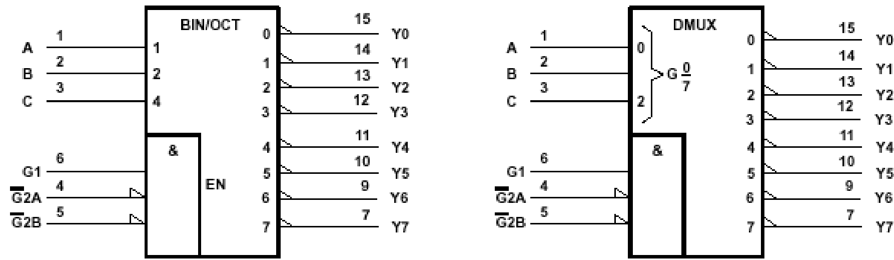
The SN54HC138 is characterized for operation over the full military temperature range of –55°C to 125°C. The SN74HC138 is characterized for operation from –40°C to 85°C.

Figure 3-9A

FUNCTION TABLE

INPUTS						OUTPUTS							
ENABLE			SELECT										
G1	$\overline{G2A}$	$\overline{G2B}$	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	L	H	H	H	H	H	L	H	H	H	H
H	L	L	H	L	L	H	H	H	H	L	H	H	H
H	L	L	H	L	H	H	H	H	H	H	L	H	H
H	L	L	H	H	L	H	H	H	H	H	H	L	H
H	L	L	H	H	H	H	H	H	H	H	H	H	L

logic symbols (alternatives)†



† These symbols are in accordance with ANSI/IEEE Std 91-1984 and IEC Publication 617-12.
Pin numbers shown are for the D, J, N, PW, and W packages.

Figure 3-9B

© BRUNO PAILLARD

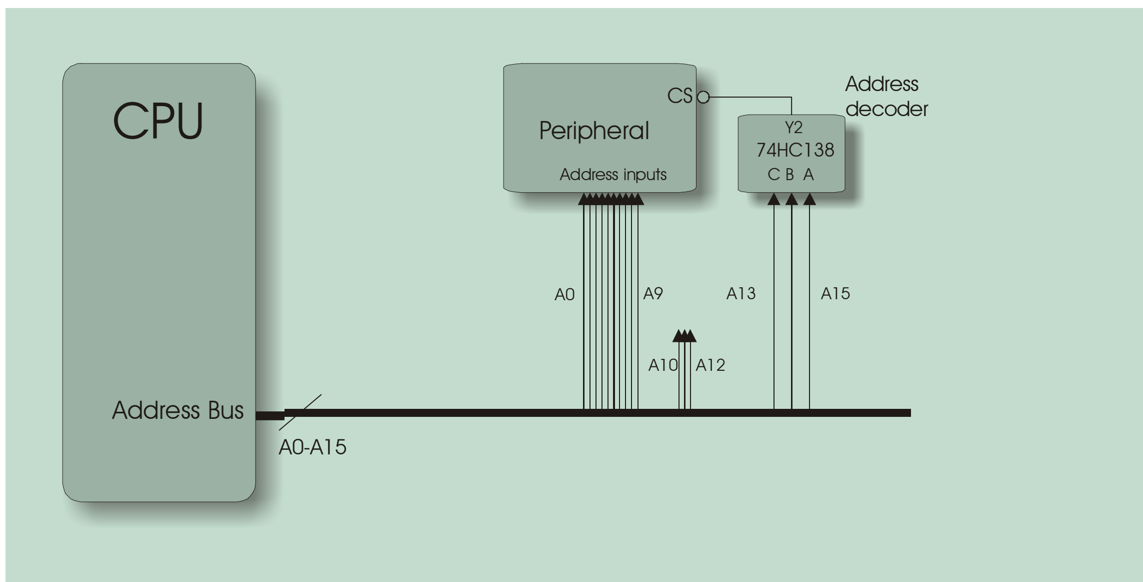


Figure 3-10

4.3. Data Bus – 3-state logic

The data bus is a group of tracks on the printed-circuit board that connect all the peripherals to the CPU, and which transport data words between the peripherals and the CPU. Each track of the data bus transports one bit of the data words exchanged. All the peripherals are connected "in parallel" on the data bus.

Since it is always the CPU that initiates exchanges with the peripherals, data words cannot be exchanged directly between peripherals. They can only be exchanged between the CPU and a peripheral. For instance, moving the contents of a memory location to another memory location would require the CPU to first read the initial memory location, and then write the data word back to the destination address. If this operation were done in external memory, where the chronograms of figures 3-4 and 3-5 apply, the complete process would take at least 3 clock cycles.

All the data pins of the peripherals are connected in parallel to the same tracks of the data bus. Since some (in fact most) of these peripherals can be read by the CPU, they have the capability to drive the bus using a data output.

When the CPU writes to peripherals, they present inputs on their data pins. This causes no problems because many inputs can be connected in parallel. However, when the CPU reads a peripheral, only this one should present outputs on its data pins. Otherwise the data outputs of many peripherals would be connected together. This would obviously lead to conflicts between these outputs, which could lead to the destruction of the output stages of any (or all) of the peripherals.

To avoid this situation, all the peripherals that can be read by the CPU (all those that can drive the data bus using an output) actually use 3-state buffers to connect to the bus. 3-state buffers are output stages that can be put in a high-impedance (disconnected output) mode of operation.

Note: *Three-state buffers are actually called so because their outputs are capable of driving a 0 state, driving a 1 state, and being put in high-impedance, which can be seen as three individual states.*

Figure 3-11 shows the internal output logic of a peripheral that can be read by the CPU. The 3-state output buffers are almost always in the high-impedance state (disconnected from the data bus). The only time where they are connected is when:

- The peripheral is selected (CS = 0), and
- The peripheral is being read by the CPU (R/W = 1), and
- The CPU triggers the transaction via the STRB signal (STRB = 0).

In these conditions only, the peripheral puts its outputs in low-impedance, and drives the data bus with the data requested by the CPU.

Since only one peripheral can be selected at a time, there is no risk of conflict between the outputs of separate peripherals, even though these outputs are all connected together via the data bus.

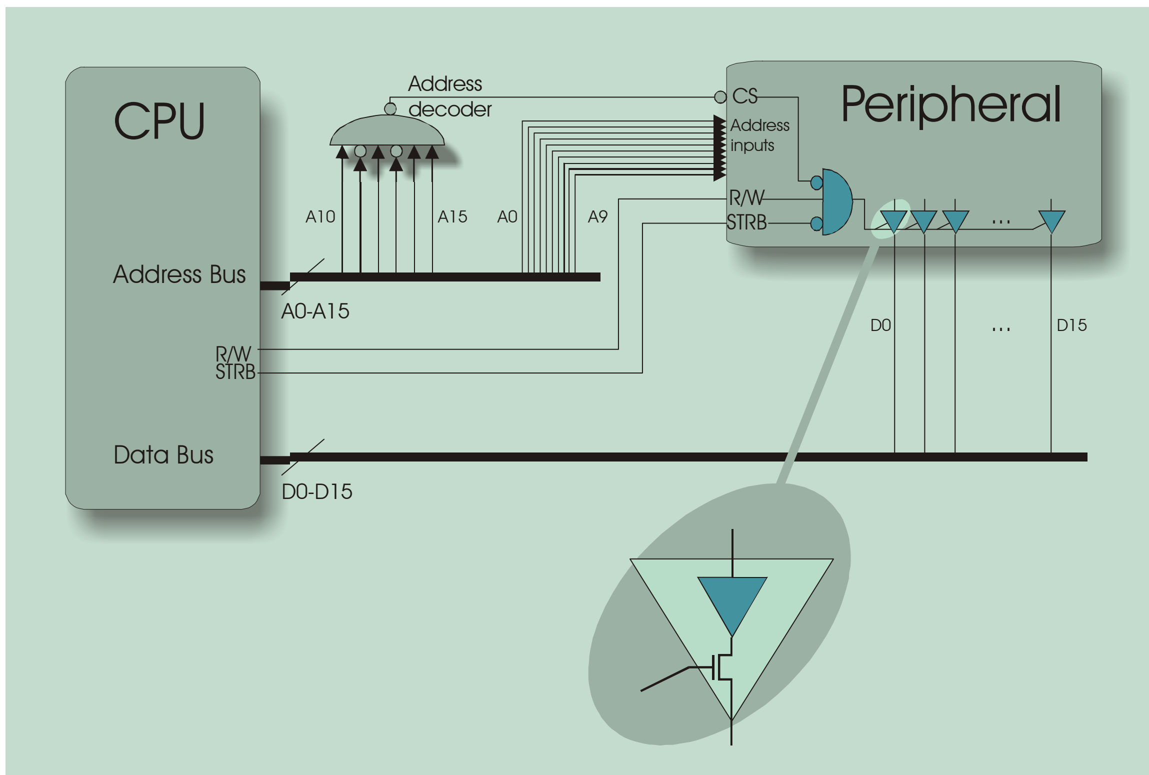


Figure 3-11

As an example, figure 3-12 shows the data sheet of a RAM circuit (from Cypress) having a capacity of 64K 16-bit words. The data sheet shows that each data pin on the device (I/O1 to I/O16) is internally connected to inputs, as well as outputs of the memory.

Such a memory can be interfaced to a TMS320VC5402 as follows:

- The Chip-Enable (CE) signal from the memory is connected to the output of the address decoder.

© BRUNO PAILLARD

- The Write-Enable (WE) signal from the memory is connected to the R/W signal of the CPU. The WE signal selects outputs when it is low and inputs when it is high. It has the same function as a R/W signal.
- The Output-Enable (OE) signal from the memory is connected to ground. This signal is used in situations when it is necessary to disable outputs using an additional signal. This may be the case in some architectures to force the memory to relinquish the bus to allow its access by another device. It is not useful here.
- The Byte-Low-Enable and Byte-High-Enable (BLE and BHE) signals from the memory are both connected to the STRB signal on the CPU. These signals perform the function of triggering the transaction during read and write cycles. There are two of them simply to allow each memory word to be accessed as two independent 8 bit words if so desired. Connecting BLE and BHE together allows the accesses to be performed 16 bits at a time.

64K x 16 Static RAM

Features

- 3.3V operation (3.0V–3.6V)
- High speed
  - — t_{AA} = 10/12/15 ns
- CMOS for optimum speed/power
- Low Active Power (L version)
  - — 576 mW (max.)
- Low CMOS Standby Power (L version)
  - — 1.80 mW (max.)
- Automatic power-down when deselected
- Independent control of upper and lower bits
- Available in 44-pin TSOP II and 400-mil SOJ
- Available in a 48-Ball Mini BGA package

Functional Description

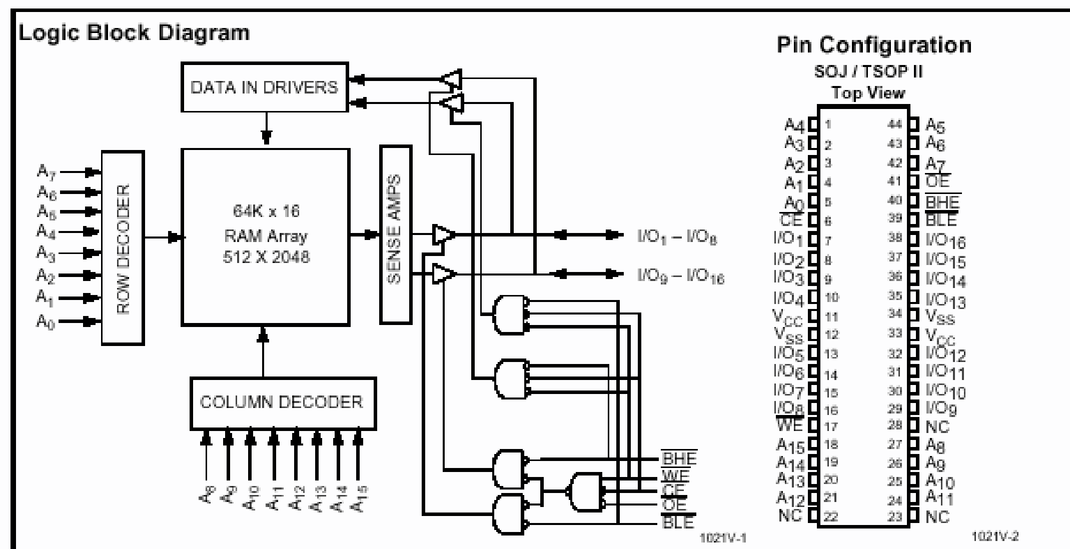
The CY7C1021V is a high-performance CMOS static RAM organized as 65,536 words by 16 bits. This device has an automatic power-down feature that significantly reduces power consumption when deselected.

Writing to the device is accomplished by taking Chip Enable ($\overline{CE}$) and Write Enable ($\overline{WE}$) inputs LOW. If Byte Low Enable ($\overline{BLE}$) is LOW, then data from I/O pins (I/O$_1$ through I/O$_8$), is written into the location specified on the address pins (A_0 through A_{15}). If Byte High Enable ($\overline{BHE}$) is LOW, then data from I/O pins (I/O$_9$ through I/O$_{16}$) is written into the location specified on the address pins (A_0 through A_{15}).

Reading from the device is accomplished by taking Chip Enable ($\overline{CE}$) and Output Enable ($\overline{OE}$) LOW while forcing the Write Enable ($\overline{WE}$) HIGH. If Byte Low Enable ($\overline{BLE}$) is LOW, then data from the memory location specified by the address pins will appear on I/O$_1$ to I/O$_8$. If Byte High Enable ($\overline{BHE}$) is LOW, then data from memory will appear on I/O$_9$ to I/O$_{16}$. See the truth table at the back of this data sheet for a complete description of read and write modes.

The input/output pins (I/O$_1$ through I/O$_{16}$) are placed in a high-impedance state when the device is deselected ($\overline{CE}$ HIGH), the outputs are disabled ($\overline{OE}$ HIGH), the $\overline{BHE}$ and $\overline{BLE}$ are disabled ($\overline{BHE}$, $\overline{BLE}$ HIGH), or during a write operation ($\overline{CE}$ LOW, and $\overline{WE}$ LOW).

The CY7C1021V is available in 400-mil-wide SOJ, standard 44-pin TSOP Type II, and in 48-ball mini BGA packages.



Selection Guide

			7C1021V-10	7C1021V-12	7C1021V-15
Maximum Access Time (ns)			10	12	15
Maximum Operating Current (mA)	Commercial		210	200	190
		L	160	150	140
Maximum CMOS Standby Current (mA)	Commercial		5	5	5
		L	0.500	0.500	0.500

Cypress Semiconductor Corporation • 3901 North First Street • San Jose • CA 95134 • 408-943-2600
October 18, 1999

Figure 3-12

4.4. Wait-States

A fast CPU like the TMS320VC5402 is capable of performing very fast accesses to external peripherals. At the fastest, accesses can be performed in 1 clock cycle, i.e. 10ns!

For some peripherals, such an access cycle may be too fast to provide a reliable exchange of data. Referring to the read chronogram in figure 3-4 for instance, For some peripherals it may take more than 10 ns to drive the requested word of data on the data bus. This is not counting the delay from the time the address appears on the address bus to the actual selection of the peripheral. To allow the CPU to be interfaced to slower peripherals and address decoding logic, **wait-states** can be inserted in the read and write cycles. Wait-states simply increase the length of the read and write cycles by a certain number of clock cycles, and slow them down enough for the access to be performed reliably. The length and the type of wait-states must be adjusted for the speed and behaviour of each type of external peripheral. Three types of wait-states can be used:

- **Software wait-states:** Software wait-states are the easiest to use because they require no additional signals in the interface logic. The CPU has a wait-state generator that can be adjusted to increase the length of each read and write cycle by up to 14 clock cycles. Read and write cycles can be slowed down to 140ns. This allows a direct interface to most peripherals, even the slowest ones. The wait-state generator provides great flexibility to specify different numbers of wait-states for separate address spaces and memory locations, therefore allowing different access speeds to be used with different peripherals.

***Note**: In the Signal Ranger board, the external memory device requires one wait state to be reliably accessed. The communication kernel that is loaded shortly after reset configures the wait-state generator for one wait-state in the address zone corresponding to this memory.*

- **Hardware wait-states:** Hardware wait-states can be used to lengthen the read and write cycles indefinitely. To support hardware wait-states, the peripheral must have a WAIT output notifying the DSP when it is able to respond to the access. This output is connected to the READY input of the DSP. When hardware wait-states are used, the DSP initiates the access and freezes its bus signals. Bus signals stay frozen until the peripheral resumes the access by activating its WAIT output. This type of wait-states is used to interface the CPU with peripherals that may not be able to respond to an access at all at certain times. This situation happens for instance when trying to access dynamic RAM circuits, which are not accessible when they are in their refresh cycle. The process allows the access cycle to be stretched until the RAM circuit completes its refresh cycle and becomes accessible again.

- **Bank switching wait-states:** When the CPU makes back-to-back accesses to memory locations that physically reside within the same memory device, these accesses can be very fast because the same device stays selected between the accesses. The address decoding delay, as well as any internal memory-select delay, is only seen during the first access to the device. When CPU back-to-back accesses cross an address boundary between two separate devices, the first access to the new device may take longer, to account for the delay of deselecting the previous device and selecting the new one. In this situation, it may be necessary to add a wait-state to the access cycle. The wait-state generator can be adjusted to add a wait-state automatically when

back-to-back accesses cross the address boundaries between separate devices. It allows the specification of address boundaries in the system with great flexibility.

5. MEMORIES: TECHNOLOGY AND OPERATION

Definition:	**Memory:** A memory is a device containing several of registers that can be accessed by the CPU. These registers can be read for all types of memories, and can be read and written for Random Access Memories (RAM).

Each register in a memory circuit has a fixed number of bits. This describes the **width** of the memory. Generally a memory device is chosen so that its width matches the width of the data bus to which it is interfaced. 16-bit wide memories are generally chosen to be interfaced with a 16-bit CPU like the TMS320VC5402. A memory contains a certain number of registers. This describes the **length**, or **depth** of the memory. The **size** or **capacity** of a memory may be described in terms of number of bits (width x length). This figure is indicative of the silicon area occupied by the memory; therefore it is closely related to cost and power consumption. The size of a memory may also be described in terms of number of registers. This information is more relevant to the software designer who handles data in complete words. For instance, the memory in figure 3-12 has a width of 16 bits, and a length of 64K words (65 536 words). Depending on the context in which it is used, its size may be given as 64K words, or as 1 M bits (1 048 576 bits).

5.1. Memory types

Two broad classes of memories exist:

- **Random Access Memory:** (RAM). This is memory that the CPU can read and write.
- **Read Only Memory:** (ROM). This is memory that the CPU can only read. The CPU can “program” or “burn” certain types of ROM memories, but this is a process much different and more complex than performing a simple write cycle.

The memory described in figure 3-12 is a RAM.

Note: *Bulk storage media like hard drives or CDROM drives are not considered memory circuits here. Only memory devices that are peripherals of a CPU (in the sense that they are interfaced to the bus system of a CPU) are considered memory circuits or memory devices. Bulk storage devices are usually complete microprocessor systems that contain a CPU and memory circuits themselves.*

5.2. Elements of technology

Each of the two broad classes of memories discussed above can itself be subdivided into several types, based on technology. Table 3-1 presents the major types:

Random Access Memory (RAM) (Read and write)	Read Only Memory (ROM) (Read only)
SRAM (Static RAM)	**Fuse ROM** (Is not in use anymore)
DRAM (Dynamic RAM)	**Masked ROM** (Is programmed by the manufacturer during one of the lithography steps used in the production of the device)
	EPROM (Erasable by exposure to ultra-violet radiation, and programmable using special equipment)
	OTPROM (Identical to EPROM, but does not have an erasure window)
	EEPROM – FLASH ROM (Electrically erasable and reprogrammable)

Table 3-1

5.2.1. ROMs

- **Fuse-ROMs**
  Fuse-ROMs were among the first programmable ROMs that could be programmed in production as well as in development. They are no longer in use today. They used to be constructed from large arrays of diodes, as shown in figure 3-13. Figure 3-13 shows the simplified example of an 8 bytes memory. When an address code is presented on the A0-A3 address lines, the internal address decoder selects the corresponding line of the matrix and brings it to a high level. Each matrix line corresponds to a register, and each column represents one bit in the register. When a diode is present at the junction between a line and a column, the corresponding bit is read as 1 on the data line. When the diode is absent, the bit is read as 0. Fuse-ROMs were sold fully populated with a diode at each junction. By using calibrated current pulses (and a programming logic not shown in the figure), diodes could be selectively fused (or burned), leaving a 0 in the data pattern stored in the memory. Fuse-ROM programs could be modified to turn more 1s into 0s (fusing more diodes), but of course 0s could never be turned back into 1s. Even though fuse-ROMs are no longer in use, the term **Fusing** or **Burning** is still applied to the process of programming a programmable ROM. The term **Fuse-Map** is applied to the data pattern in a programmable ROM, even today.

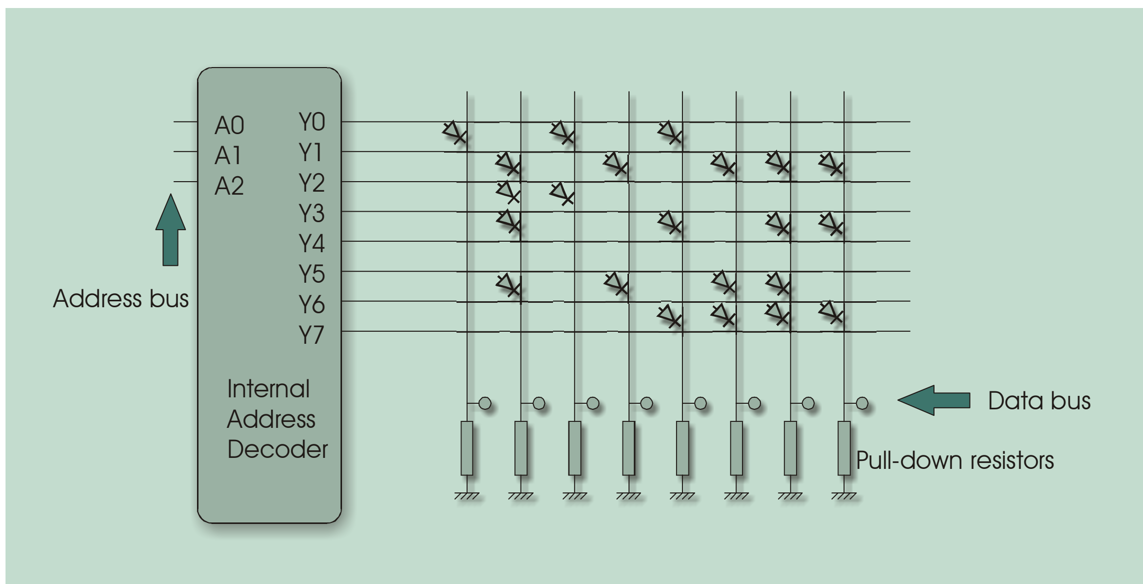


Figure 3-13

- **Masked ROM:**
Masked ROMs are programmed during one of the many lithographic steps of an integrated circuit's manufacturing process. They are usually included with a CPU as part of a microcontroller, and are programmed at the same time the device is manufactured. A special lithographic mask is used for the programming. Masked ROMS may contain code or data that are useful to many applications. For instance, the TMS320VC5402 contains a masked ROM that stores a bootloader code that is used at the reset of the DSP for any application. Masked ROMs can also contain custom code that makes up a specific application. Today, most CPU manufacturers provide tools and procedures that enable embedded systems manufacturers to submit and include their custom code into the silicon chip of the microcontroller. This saves cost, and space in the production of the system. However, since the tooling cost of designing custom lithographic masks, and setting up a production of custom CPUs is quite high, these savings are only justifiable in very high-volume productions.

- **EPROM** (Erasable Programmable Read Only Memory)
A few years ago these memories were the most widely used ROMs in the industry. For these memories, bits are stored by trapping charges behind tunnel barriers. The storage cell is implemented in the form of the conductive "floating gate" of a MOS transistor, which is completely encased into electrically insulating oxide layers. The charges require a high energy to tunnel across the oxide layers, and once behind, have a very low probability of crossing back. In fact the probability is so low that EPROMs normally have retention times in the tens of years. EPROMs are programmed using a special equipment called an EPROM programmer. The programmer provides high voltage (12-15V) calibrated pulses that can selectively transform memory bits from 1s to 0s. Once programmed, 0s cannot be programmed back into 1s using the programmer. The EPROM needs to be erased by exposing it to ultraviolet radiation of a specific wavelength for a certain amount of time (usually 10 to 20 minutes). The ultraviolet light provides the energy necessary for the charges to cross back the oxide barrier. For that purpose EPROMs have a quartz window transparent to ultraviolet light. Normally, once programmed, a label is placed over the window to minimize its exposure to

© BRUNO PAILLARD

ambient ultraviolet light. Without the label, an EPROM would loose some data in a few days if exposed to direct sunlight, or in a few months if exposed to indoors lighting. Camera flashes are a great source of ultraviolet light. I can say from experience that it is a very bad idea to take a picture of your prototype if the EPROM window is not protected.

EPROMs can be used in the development stage, because they can easily be erased, reprogrammed and the new code can be readily tested. An EPROM programmer only costs a few thousand dollars. An eraser, which is nothing more than an ultraviolet tube in a box, only costs a hundred dollars or so. EPROMs can also be used in small production runs. In either case, they need to be mounted in a socket, because the programming and erasure procedures cannot be performed directly on the printed-circuit board.

Figure 3-14 shows a simplified view of a cell storing one bit of data in an EPROM. The cell is essentially a MOS transistor with a special floating gate that is used to trap charges. The threshold of the transistor changes depending on the charge that is trapped in the floating gate. Therefore during a read cycle, when the transistor is selected using its control gate, the amount of current that passes through it can be used to determine if the bit is 0 or 1. The control gate is also used to generate the large electric field necessary for the charges to tunnel into the floating gate during the programming of the cell.

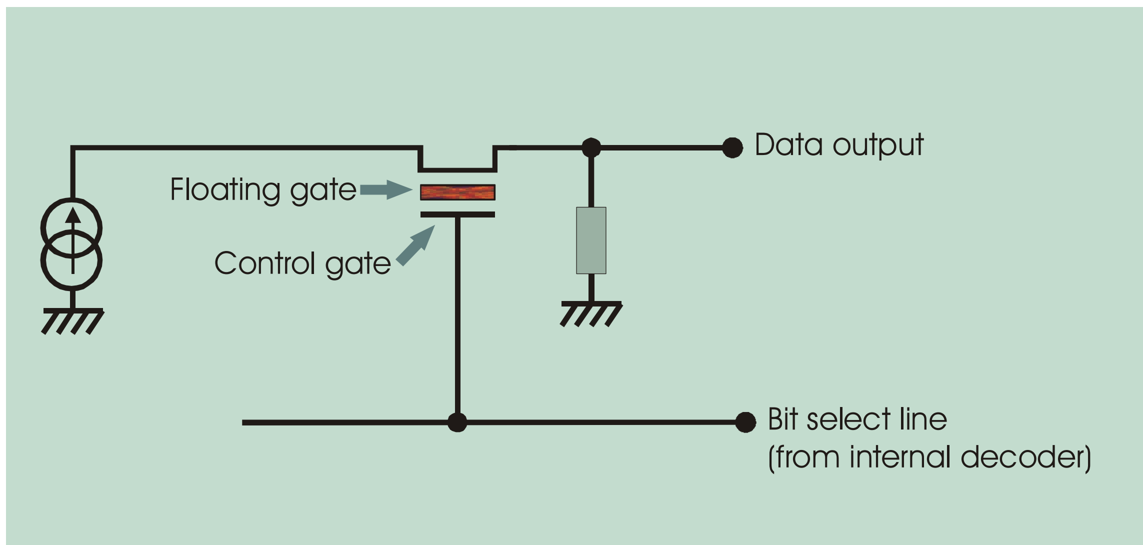


Figure 3-14

- **OTPROM**
  One-Time Programmable Read Only Memory. An OTPROM is nothing more than an EPROM that is encapsulated in a low-cost plastic package that does not have the quartz window required for erasure. Since its package is low-cost, it is much less expensive than an EPROM. It is normally used for small to medium production runs, for which the cost of a masked ROM is prohibitive, but where the code is stable enough to not require erasure.

- **EEPROM**
  Electrically Erasable Programmable Read Only Memory. This technology represents an improvement over the older EPROM technology. The storage cell structure is similar, but ultraviolet exposure is not required to erase it. High-voltage calibrated pulses are used for both programming and erasure.

Nowadays, EEPROMs contain voltage pumps that generate these high voltages (12-15V) directly from the single 3.3V or 5V logic supply of the memory. This means that the memory can be erased and programmed in-situ, by the CPU to which it is interfaced. It does not have to be placed into an EPROM programmer. Programming is still a complex and long operation (it can take up to several milliseconds to program one word, compared to the few nanoseconds it takes to write a word in RAM). However it presents several advantages:

- Since the EEPROM can be programmed in-situ, it does not have to be mounted in a socket. Therefore the circuit assembly is much less expensive, and more reliable (less sensitive to vibrations and contact oxidation).
- It can often be integrated on the same silicon chip as the CPU. Many microcontrollers and DSPs do integrate a certain amount of EEPROM.
- In development, the erasure and programming cycle is much simpler than for an EPROM, and does not require any special equipment. Erasure and programming are usually carried out by the target CPU itself, under control of special programming software.
- Since the CPU of the target system itself can program the EEPROM, it can be used to store various system parameters. Measuring instruments, for instance, can use an EEPROM to store calibration values. Since it is a ROM, these parameters do not disappear when the system is powered-down.
- They allow the embedded code to be upgraded, even after the system has been deployed (what is called field-upgradeability). This is the case for many computer peripherals (modems, video cards…etc.) Even the BIOS of most PCs today are located in EEPROMs and can be field-upgraded by the consumer.

EEPROMs are the most widely used types of ROMs in the industry today. They are used in the development stage, as well as in production for small, medium, and sometimes even high-volume productions.

- **FLASH memory:**
  Sometimes called FLASH ROM, they are a variant of the EEPROM memory. The differences between FLASH and EEPROM are mostly technological, namely:
    - FLASH memories are much faster to program than EEPROMs, Each word of EEPROM may take as much as 5 to 10 ms to program, while each word of FLASH usually takes 5 to 10 μs.
    - The cell size is much smaller for FLASH memories than for EEPROMs (around 5 μm for FLASH, versus around 50 μm for EEPROMs). Therefore FLASH memories usually have much larger capacities than EEPROMs.
    - Erasure is performed in bulk (the whole memory, or a complete section at a time), while it can be performed one word at a time for EEPROMs

5.2.2. RAMs

- **DRAM**
  Dynamic Random Access Memory has the simplest structure of all RAM memories. Figure 3-15 shows a simplified view of the storage cell of a Dynamic

RAM. For this type of RAM, bits are stored in capacitors, and a storage cell only requires the use of one transistor. Since the storage capacitor is not perfect, its charge leaks with time. It is therefore necessary to restore (or refresh) the charges periodically. Every few milliseconds, a refresh cycle has to be performed. The refresh cycle is simply performed by presenting all row addresses in sequence on the address lines of the memory. Earlier systems had the CPU perform this operation, which cost precious execution time. Newer systems employ a RAM controller that performs this operation autonomously. In any case, the RAM is not accessible during its refresh cycle, which is one of its disadvantages. Another disadvantage is that the refresh cycle consumes current, even when the RAM is not being accessed. Its great advantage is that since it requires only one transistor per storage cell, very large RAM sizes can be achieved on relatively small silicon areas. It is the lowest cost memory per bit in the industry today. This is the type of memory that is used in computer systems for instance.

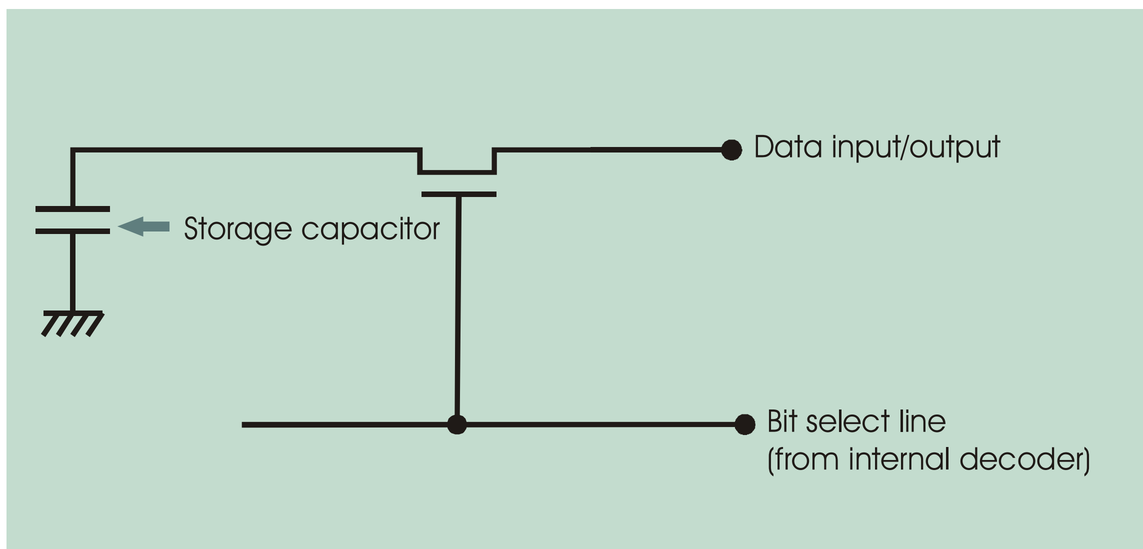


Figure 3-15

- **SRAM**:
  Static Random Access Memory. The basic storage cell structure in a static RAM is a flip-flop. Figure 3-16 shows the structure of an RS flip-flop (which is one of the simplest kind), as well as the schematic of the NAND gate that is used to make the flip-flop. From figure 3-16, it is clear that the basic storage cell uses at least 8 transistors. At equal capacity a static RAM uses much more silicon area than a dynamic RAM. It is therefore usually much more expensive. From another point of view, at equal silicon area (and price) a static RAM has much less capacity than a dynamic RAM.
  One advantage of static RAMs is that the storage structure is inherently stable. They do not require refresh cycles, and can be accessed at all times. For embedded systems access time is usually a bigger issue than size, and static RAMs are often preferred.
  Static RAMs are implemented in CMOS technology, for which power consumption is proportional to the average number of gate transitions per second. There are no gate transitions when the RAM is not being accessed, therefore static RAMs can keep their data with very little supply current. Most static RAMs have a standby mode that minimizes power consumption when the

device is not being accessed. In fact for some devices, power consumption is so low in this mode that it can stay powered by a small lithium battery for years. Some manufacturers actually make memory devices that include a standby battery in the same package. Such devices may be used to keep system parameters, which must not be lost when the system is powered down. Although EEPROM are increasingly preferred for this purpose.

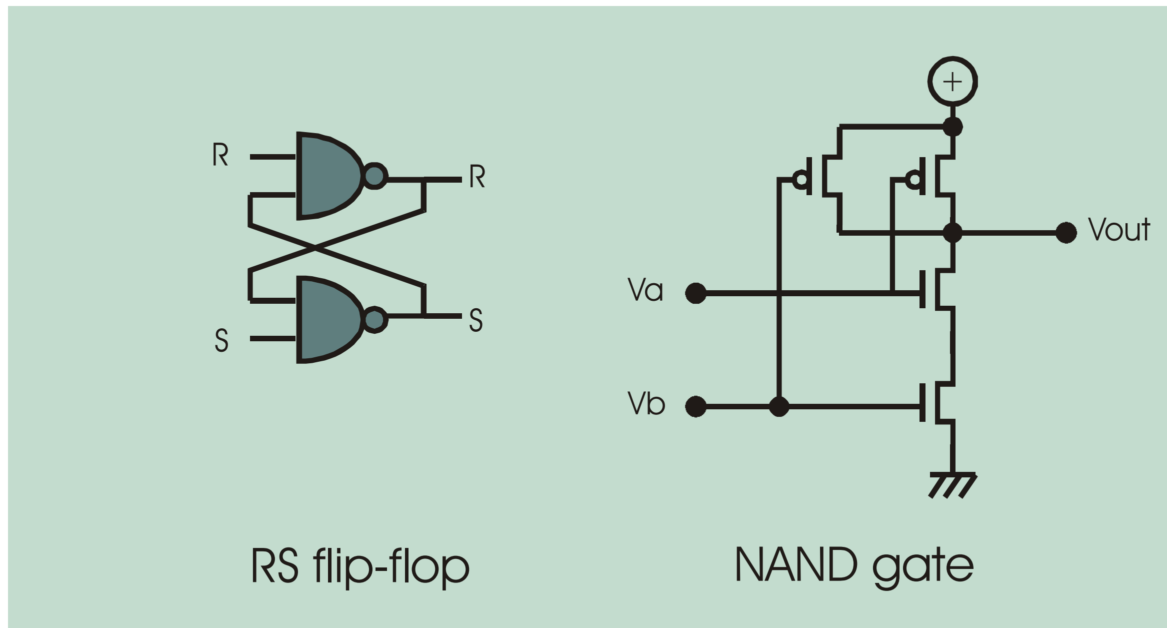


Figure 3-16

5.3. Paralleling of memory devices

The system designer usually chooses memory devices that have a width equal to the width of the CPU's data bus. However, in some case several narrower memory devices may be connected in parallel to cover the complete width of the data bus. For instance, one 16-bit wide memory device can be interfaced to a 16-bit CPU. Equivalently 4 memory devices, each having 4-bit registers, could be used. In this case, the same address decoder selects all the devices, and each device is interfaced to a portion of the data bus. The CPU sees the whole group as a single, wider memory device.

© BRUNO PAILLARD

Introduction To The Signal Ranger DSP Board

- 1. Hardware features
- 2. Software tools

1. HARDWARE FEATURES

The Signal Ranger DSP board is a development board that has been designed for multi-channel digital signal processing. It features a TMS320VC5402 fixed-point DSP. This 16-bits DSP has a Harvard architecture, and is capable of executing 100 Million instructions per second (100 MIPS).

The DSP has 16 K words of on-chip dual access RAM (RAM that can be accessed twice per CPU cycle). Even though its architecture is Harvard, the on-chip RAM is decoded in both the data and program spaces. It is the only RAM on the board that is decoded in the program space. Therefore it is the only place where code can be executed.

An external memory device adds 48 K words to the data space of the CPU, for a total of 64K words. The external RAM is only decoded in the data space, and is accessed with one wait-state (2 CPU cycles per access). The address map of the board is described in figure 3-6.

The DSP is interfaced to 8 Analog Interface Circuits (AICs). Each AIC includes an Analog to Digital Converter (ADC) and a Digital to Analog Converter (DAC). These AICs can sample analog signals at up to 22 K samples/s (80 K samples/s without anti-aliasing filters), with a 15-bit resolution. They are arranged as two banks of 4 AICs. Each bank is interfaced to one serial port (McBSP) of the DSP.

The board is connected to a PC via a USB port. The USB port provides power supply to the board, as well as a 0.5 Mbit/s data pipe.

The board uses an independent USB controller (Cypress AN2131). For the user, this means that even a complete crash of the DSP software does not compromise board communications, and the DSP can always be reset with no hardware intervention. This is a nice feature for a development board because the DSP code might not always be stable. In fact the USB controller is a microcontroller itself, and the DSP is interfaced as one of its peripherals via its Host Port Interface (HPI). This type of architecture where the DSP is a peripheral of another “Host” CPU is very popular nowadays.

2. SOFTWARE TOOLS

Software tools come from several sources:

2.1. DSP code development tools

DSP code can be developed using the Code Composer Studio development environment. This development environment is provided by Texas Instruments, and contains the main following tools:

- Optimizing C compiler
- Assembler
- Linker
- Simulator

These tools allow the development of DSP code in C++, C, or assembly language, and generate an executable file that can be downloaded to the DSP using the debugger.

2.2. Debugging tools

2.2.1. Mini-debugger

This tool has been developed at Université de Sherbrooke. It consists in a user-interface running on the PC, as well as a communication kernel running on the DSP. The mini-debugger can perform the following operations:

- Load an executable code into DSP memory.
- Launch the execution of DSP code from any given entry point.
- Force a branch to a given address, even if the DSP is already executing user code.
- Read or write the CPU registers in real time (during the execution of DSP code).
- Read or write memory sections in real time (during the execution of DSP code). The data display format for memory reads and writes can be chosen from a variety of formats (signed, unsigned, decimal, hexadecimal, 16-bits, 32-bits, floating-point…etc.). Data can even be plotted graphically, which is interesting to observe the signals that are processed by the DSP in real time.

All accesses can be performed using the symbolic names (variables, labels) used in the source code.

The communication kernel is a small (256 words) code that executes on the DSP and supports data exchanges between the PC and the DSP. It is automatically loaded when the mini-debugger is run, or when the reset button is pressed. It has been designed to interfere very little with the user code executing on the DSP; in particular, all user code that executes under interrupts has precedence over the kernel.

More details can be found in the chapter “Signal Ranger development tools”.

2.2.2. In-circuit emulator

The Signal Ranger board has a JTAG connector that can be with an emulator pod, such as an XDS510pp from Texas Instruments. It is supported by the Code Composer Studio development tools. Using these tools, it is possible to execute the DSP code under control of the user. The user can perform step-by step execution, or read and write memory locations and CPU registers. More information on emulating tools can be found in the chapter “Software Development Tools”.

© BRUNO PAILLARD

2.3. PC code development tools

The mini-debugger is a nice tool to test DSP code interactively using a standard user interface. However, many DSP applications require the use of a custom user interface on the PC. For these applications various software interfaces, developed at Université de Sherbrooke, are provided. These interfaces allow a designer to develop PC applications that can access and use the DSP board.

- Several Dynamic Link Libraries (DLLs) are provided to develop PC applications using any development environment that supports the use of DLLs. Microsoft's Visual Studio is an example. Using these DLLs, the PC application can load DSP code, execute it, and communicate with the board at high and low levels.
- An interface library is provided to develop PC applications using the LabVIEW development environment from National Instruments. Using this library, the PC application can load DSP code, execute it, and communicate with the DSP board.
- Complete documented code examples are provided to accelerate the development learning curve.

Binary Representations and Fixed-Point Arithmetic

- 1. Bases and positional notations
- 2. Unsigned integer notation
- 3. Signed integer notation using a separate sign bit
- 4. 2's complement signed notation
- 5. Offset binary notation
- 6. Fractional notations
- 7. Floating point notation
- 8. BCD notation
- 9. ASCII codes

1. BASES AND POSITIONAL NOTATIONS

A base is a set of symbols, associated to a positional notation rule. In a positional notation, every position (digit) in a number is attributed a power of the base. For integer unsigned numbers, the power begins at 0 for the rightmost digit, and is incremented by one for each successive position. The base number indicates the number of symbols used in the representation.

Note: *In this document, the base used for the representation is indicated by a suffix following the number. For instance* 12_d *represents the number 12 expressed in base 10.* 12_H *represents the number 12 expressed in hexadecimal.* 10_b *represents the binary number 10.*

Table 5-1 shows the various symbols used in the three most common bases used in microprocessor development: Decimal, Hexadecimal and Binary.

Base 10	Base 2	Base 16 (Hexadécimal)
0	0	0
1	1	1
2		2
3		3
4		4
5		5
6		6
7		7
8		8
9		9
		A
		B
		C
		D
		E
		F

Table 5-1

1.1. Base 2 - Binary

Base 2 (binary) is used in the field of microprocessor systems because these systems are built from logic circuits that have two states. Data elements processed by these systems are **binary words**. Binary digits are called **bits**. 8-bit binary words are called **bytes**. Groups of 4 bits are sometimes called **nibbles**.

Definition: **LSB** Least Significant Bit. This represents the rightmost bit in a binary word.

Definition: **MSB** Most Significant Bit. This represents the leftmost bit in a binary word.

Note: *In an N-bit binary word, bits are usually numbered from 0 (the LSB) to N-1 (the MSB), rather than from 1 to N.*

1.2. Number of bits, dynamic range and resolution

An N-bit binary word can represent a total of 2^N separate values. In the case of the unsigned integer notation, these values normally range from 0 to 2^N-1. For other types of binary notations the dynamic range may be different. For instance for 2's complement signed integers the dynamic range spans -2^{N-1} to 2^{N-1}-1.

The resolution of the notation represents the smallest increment that can be described using the notation. For integer binary notations, the resolution is always one (the smallest difference that can be represented between two integer numbers cannot be smaller than one. However, as will be discussed later, fractional binary notations allow finer resolutions.

© BRUNO PAILLARD

Obviously, when using binary numbers having a fixed number of bits, the resolution and the dynamic range of the numbers represented vary in opposite directions: When the resolution is increased (made finer), the dynamic range is reduced, and vice-versa.

Definition:	**Dynamic Range** For a given binary notation, the dynamic range represents the range of values that can be represented using this notation. It is usually specified by giving the smallest number and the largest number that can be represented using this notation. For instance, the 16-bit unsigned integer notation, has a dynamic range that spans values from 0 to 65535_d.

Definition:	**Resolution** For a given binary notation, the resolution is defined as the smallest increment between two numbers that can be represented using the notation. Integer notations have a resolution of one.

1.3. Base 16 - Hexadecimal

Binary is the base of choice to describe data and processes in the field of microprocessor design. However, when large values need to be represented, a binary number may use a large number of bits. The notation rapidly becomes unpractical. To simplify the notation, base 16 (hexadecimal) is often used. Base 16 has the advantage that 16 it is a power of 2. Every digit in base 16 corresponds to a group of 4 bits in base 2. Conversions between base 16 and base 2 are therefore easy to do and intuitive.

To convert from binary to hexadecimal, the bits of the binary word are grouped by packets of 4. These 4 bits packets are called nibbles. Each nibble is then converted to an hexadecimal symbol. There are 16 symbols in the hexadecimal base: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. To convert back into binary, each hexadecimal digit is simply replaced by its corresponding 4-bit binary pattern.

1.3.1. Conversion examples

```
1101  0110  0010  1010                    Base 2 -> Base 16
D     6     2     A

F     5     A     3                       Base 16-> Base 2
1111  1010  1010  0011
```

2. UNSIGNED INTEGER NOTATION

2.1. Conversion from base 2 to base 10

To convert from base 2 to base 10, one simply has to add the value of each bit (0 or 1) multiplied by the power of 2 corresponding to the bit position.

Ex:	1	0	1	1	0	1_b
=	$1x2^5$ +	$0x2^4$ +	$1x2^3$ +	$1x2^2$ +	$0x2^1$ +	$1x2^0$
=	45_d					

2.2. Conversion from base 10 to base 2

Converting from base 10 to base 2 requires successively dividing quotients of the original number by 2, using an integer division, and keeping the rests. The procedure is represented graphically in figure 5-1. The procedure can easily be demonstrated by working backwards from the last division, and showing that each bit contributes to the initial decimal number by a power of 2 corresponding to its position.

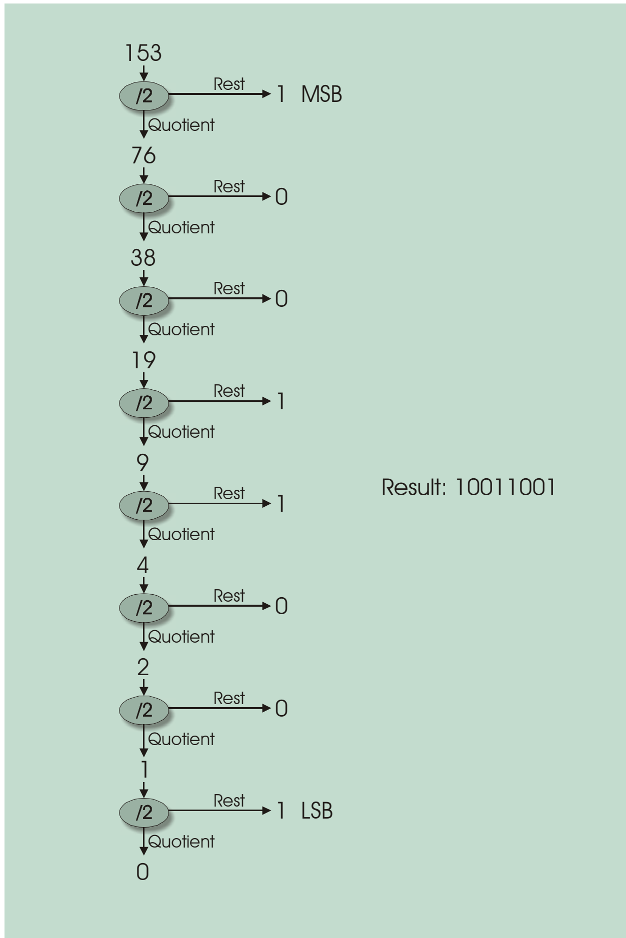


Figure 5-1

2.3. Unsigned integer arithmetic

2.3.1. Addition

The addition of two unsigned integer binary words is performed using the same technique as for a decimal addition. The addition is performed bit-by-bit, beginning by the LSB (leftmost bit). At each stage when the result is greater than 1, a carry is generated and propagated to the next stage. The carry of the first stage is set to zero.

Example:

```
  1001        9d
  1011 +     11d
 ----------
 10100       20d
```

 © BRUNO PAILLARD

A one-bit adder can be implemented using only combinatory logic. Such an adder would have 3 inputs:

- Bit input No1 (bit from first operand)
- Bit input No2 (bit from second operand)
- Carry input (carry propagated from the previous stage)

It would have two outputs:

- Bit output (result bit)
- Carry output (carry propagating to the next stage)

The table below shows the truth table of this 1-bit adder:

Input 1	Input 2	Input carry	Output	Output carry
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 5-2

Figure 5-2 shows an example of implementation of a 4-bits adder. It also shows one possible implementation of the 1-bit adder using combinatory logic. A 40-bit wide circuit of this type is implemented in the Arithmetic and Logic Unit of the TMS320VC4502.

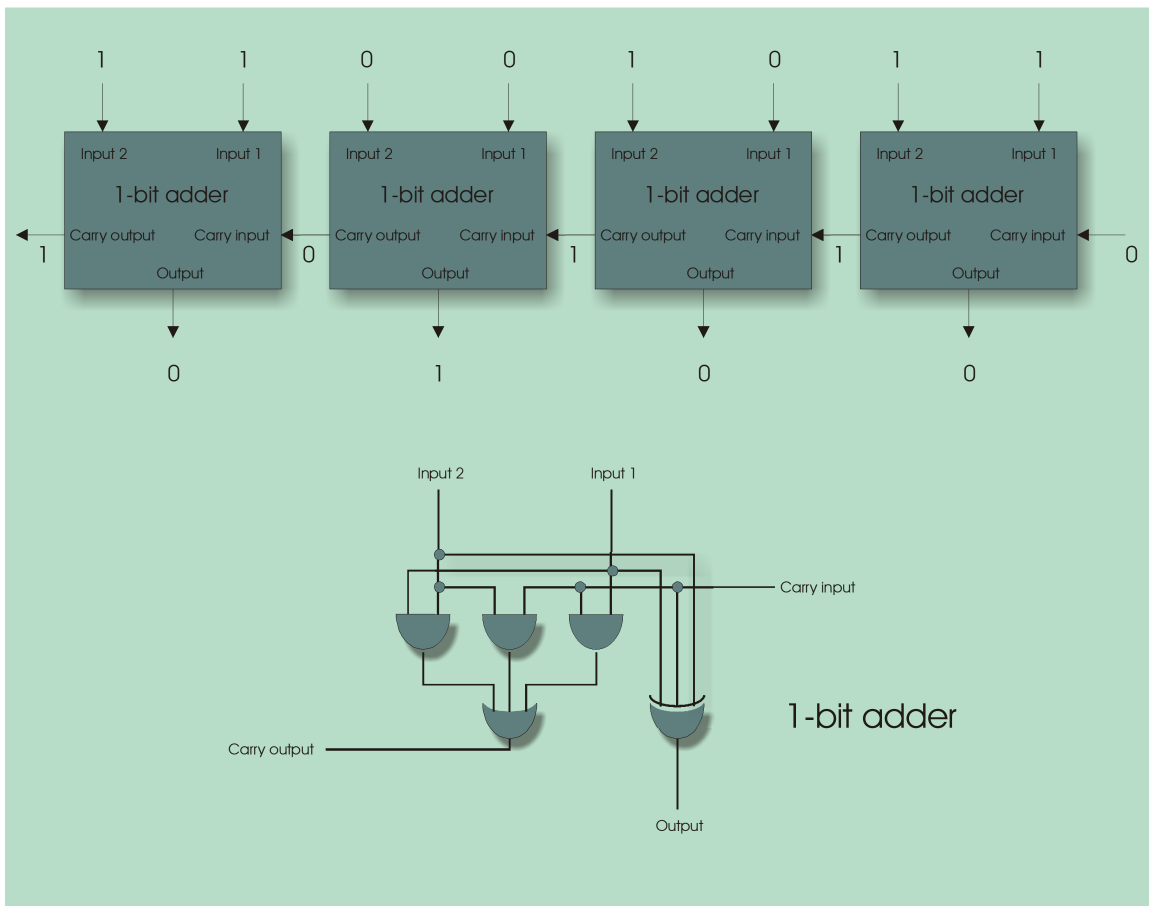


Figure 5-2

***Note**: Usually the notation is the same, and uses the same number of bits for the operands, and the results of an operation. This is due to the fact that the Arithmetic and Logic Unit of a CPU usually works with a fixed number of bits. For the TMS320VC5402, the operations are normally performed using 40-bit arithmetic.*

2.3.2. Overflow

The overflow is defined as follows:

Definition:	**Overflow** An overflow occurs when the result of an operation is outside the dynamic range of the notation, whatever the notation,.

The notion of overflow depends on the notation. For an N-bit unsigned notation, an overflow is declared when a result is greater than 2^{N-1}. For instance, for a 16-bit unsigned notation the overflow limit is 65535.

During an unsigned addition for instance, an overflow is declared whenever a carry is generated by the last stage of the adder. This carry would normally propagate to the bit position directly to the left of the MSB, which does not exist in the notation.

© BRUNO PAILLARD

2.3.3. Multiplication by N

A binary multiplication by N is also performed using the same technique as for a decimal multiplication. The two operands are positioned one above the other, and every bit of the multiplicand is successively multiplied by every bit of the multiplier, beginning with the LSB. For every new bit of the multiplier, the partial result is shifted left by one bit position. All the partial results are added to form the final result. The process is somewhat simpler than in base 10 however, because each digit of the multiplier can only take two values: 0 -> the partial result is zero, 1 -> the partial result is a copy of the multiplicand.

Example of a multiplication:

```
    1101           13d              Multiplicand
    1011 x         11d              Multiplier
------------------
    1101                            Partial results
   1101    +
  0000     +
 1101      +
------------------
10001111           143d             Final result
------------------
```

2.3.4. Multiplication by 2

Multiplication by 2 is a simpler special case. The result is obtained by left-shifting all the bits of the multiplicand by one bit position. 0 replaces the LSB. When the notation has a fixed number of bits (which is usually the case when the operation is performed by a CPU), the MSB is lost after the left shift. For an unsigned notation, an overflow is declared if the lost MSB was 1.

Example of a multiplication by 2:

```
  0010110101001   x2
     <-         <-0
= 0101101010010
```

2.3.5. Division by 2

A division by 2 is obviously obtained by the reverse process. The binary word is right shifted by one bit position. 0 replaces the MSB, and the LSB is lost (in fact the LSB represents the rest of the integer division by 2).

Example of a division by 2:

```
    0010110101001      /2
0 ->    ->
=   0001011010100
```

3. SIGNED INTEGER NOTATION USING A SEPARATE SIGN BIT

The first solution that comes to mind to define a signed notation is to add a separate sign bit to the unsigned binary word. The MSB may used as the sign bit (0=positive, 1=negative).

Example of an 8-bit signed notation using a separate sign bit:

```
Sign = 0 (positive)
|
00101001    = +41d


Sign = 1 (negative)
|
10101001    = -41d
```

This notation allows the representation of values from $-2^{N-1}-1$ to $2^{N-1}-1$. One problem of this notation is that there are two separate codes to represent 0:

- 10000000_b (-0)
- 00000000_b (+0)

Because it has considerable advantages when performing binary arithmetic, another signed notation called **2's complement notation** is usually preferred. The separate sign bit notation is not frequently used. One such use is to represent the mantissa of floating point numbers.

4. 2'S COMPLEMENT SIGNED NOTATION

Two's complement notation is best described using the concept of a circular counter that rolls over to zero past the maximum value. Such a counter is represented in figure 5-3 for a 16-bit counter. After $FFFF_H$, the counter rolls over to 0000_H.

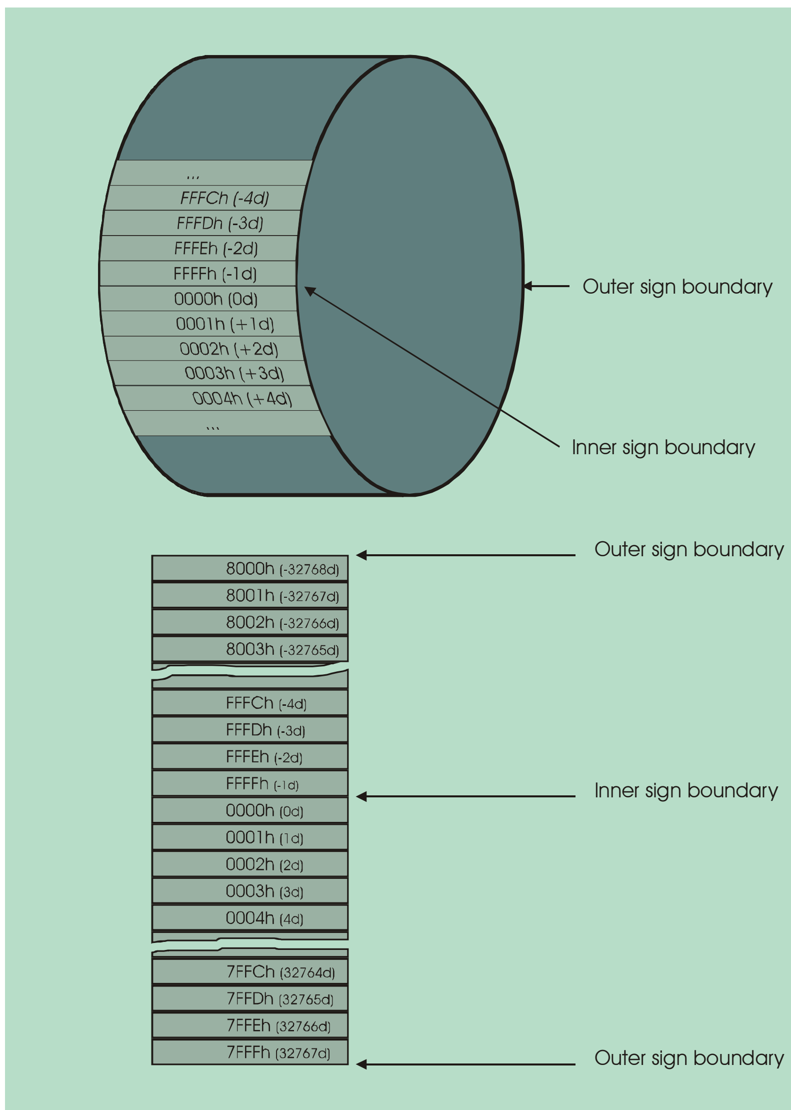


Figure 5-3

The code 0000_H is defined as the representation of the value 0_d. The code just after it (0001_H) is naturally defined as the representation of $+1_d$, 0002_H represents $+2_d$…etc. The code just before 0000_H ($FFFF_H$) is logically defined as the representation of –1d. By the same token, $FFFE_H$ represents $–2_d$ …etc.

The limits of the representation are defined to give as many (or nearly as many) positive values as negative values. For the above 16-bit notation, the dynamic range goes from $–32768_d$ (8000_H) to +32767 ($7FFF_H$).

More generally, for an N-bit notation, the dynamic range spans $–2^{N-1}$ to $+2^{N-1}-1$.

The 2's complement notation might take some time to get used to. Here are some pointers:

- One characteristic of this notation is that all positive numbers are represented as they would be in the unsigned notation. However their MSB can only be zero. Negative numbers on the other hand all have their MSB at 1. For this reason, in 2's complement notation the MSB is called *sign bit*, even though it does not represent a separate sign bit per se.

- Small positive numbers have many 0s to the right of the 0 MSB, while small negative ones have many 1s to the right of the 1 MSB.
- Large positive numbers have only a small number of 0s (sometimes none at all) between the MSB and the first 1 to its right. While large negative number have only a small number of 1s (sometimes none at all) between the MSB and the first 0 to its right.

4.1. Changing the sign of a number

To obtain the opposite of a positive number, the usual technique is described as follows:

- First each bit of the number is complemented (this is also called taking the 1's complement of the number). For instance, to take the opposite of 0000000000010110_b (22_d), each bit is first complemented.

```
0000000000010110        22d
1111111111101001
```

The number that is obtained is found on the circular counter at a location symmetrical of the code 0000000000010110b, relative to the inner sign boundary on figure 5-3.

The opposite of the number 22_d is really located at the position symmetrical of the code 0000000000010110b, relative to the 0 code, rather than to the inner sign boundary. To correct this a value of 1 is simply added to the code to obtain the required result.

```
1111111111101001       -23d
               1 +
1111111111101010       -22d
```

Note: *Since the 2's complement notation is based on the notion of circular counter, one must know precisely the number of bits used in the notation to interpret code values properly. For instance :*
the code 0111_b in 2's complement notation on 4 bits, represents the positive value $+7_d$.
In 2's complement notation on 3 bits it represents the negative value -1_d.

The number of bits used to perform calculations in the Arithmetic and Logic Unit of a CPU is usually known and fixed, therefore the interpretation of data values doesn't normally cause any problems. However in a few situations this little detail can cause considerable problems. Conversions between a 16-bit signed notation and a 32-bit signed notation for instance require close attention. A DSP like the TMS320VC5402 has features to help perform such of conversions, however the software designer should always pay close attention to the way the operation is carried out, to avoid any surprises.

Also, most scientific pocket calculators capable of calculating in decimal and hexadecimal or binary perform their calculations assuming a fixed number of bits. However leading zeroes are often not displayed so the user does not have a clear indication of the number of bits used for the notation. For instance, a typical case is described as follows:

The user performed a calculation in hexadecimal and the result displayed on the calculator screen is $FFFC_H$. The user assumes a 16-bit notation, however the calculator used a 32-bit notation for the calculation. Upon conversion to base 10, the result should

have been -4_d. However, the calculator will display $+65532_d$. When using pocket calculators to convert from hexadecimal or binary to decimal, close attention should be paid to the width of the notation in order to avoid any unpleasant surprises.

4.2. Advantages of the 2's complement notation

The main advantage of the 2's complement notation is that additions and subtractions can be performed using the same adder logic used to perform unsigned additions. From the point of view of CPU complexity, this is a great advantage because it leads to a simplification of the Arithmetic and Logic Unit, which leads to less silicon area, less power consumption and lower CPU cost.

The following example describes 3 operations that are performed in 2's complement notation on 8 bits, using only an addition operator. Subtractions are performed by taking the opposite of one of the operands. It should be noted that the last two operations would have generated an overflow, had they been interpreted as unsigned operations. However in the 2's complement notation, the generation of a carry by the last stage of an addition does not indicate an overflow.

```
   Examples:

   X = 83d
   Y = 11d
   X + Y = 94d

01010011           83d
00001011 +         11d
-------------------
01011110           94d

   X - Y = 72
01010011           83d
11110101 +        -11d
-------------------
01001000           72d

   Y - X = -72
00001011           11d
10101101 +        -83d
-------------------
10111000          -72d
```

4.3. Overflow

Following the definition of overflow that was given earlier, an overflow occurs in an N-bit 2's complement notation when a result is greater than $2^{N-1}-1$, or lower than -2^{N-1}. In 16-bit notations the limits are –32768 and +32767.

Figure 5-4 describes the case of an addition of two large positive operands. Both operands are sufficiently large that the result crosses the outer sign boundary to the right of the circular counter. This is called a **rollover**, because the result that should have been positive rolls over to the negative values.

Figure 5-5 describes the opposite case of an addition between two large negative operands. In this case also, the operands are sufficiently large to cross the outer sign boundary and cause a rollover.

When the result of an addition crosses the inner sign boundary a carry is generated. However no overflow is detected in 2's complement notation.

A rollover usually has catastrophic consequences on a process, because a result that should have been very large and positive suddenly becomes a very large negative one, in the other direction, a result that should have been very large and negative suddenly becomes a very large positive one. If the result is the command signal of a system, for instance, this system will react in the direction opposite to what should have been achieved, and with great amplitude.

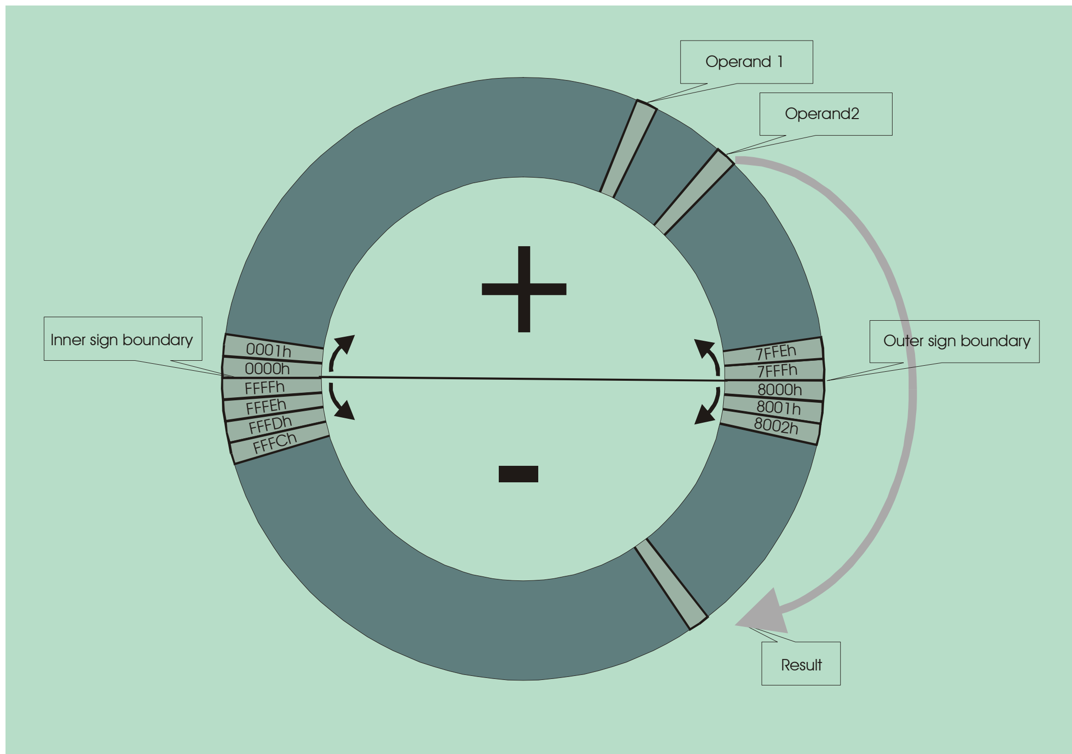


Figure 5-4

 © BRUNO PAILLARD

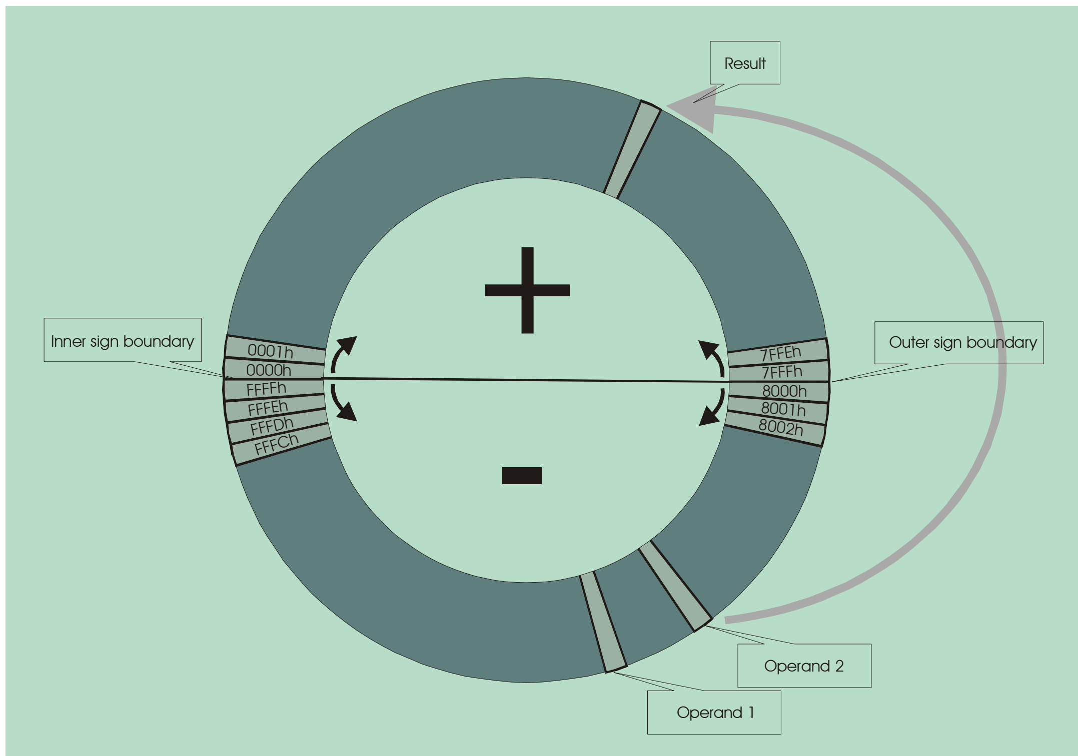


Figure 5-5

***Note**: An overflow during an addition in 2's complement can only happen when two very large positive operands, or two very large negative operands are added. It can never happen during the addition of a positive operand and a negative operand, whatever their magnitude.*

4.4. Saturation or rollover

To avoid a rollover, most operations performed in 2's complement should detect an overflow and saturate the result to the most positive or most negative value that can be represented in this notation. For instance in figure 5-4, the result should have been saturated to $+32767_d$. It should have been saturated to -32768_d in figure 5-5. Most CPUs allow the detection of overflows and steps can usually be taken in software to saturate the results. The Arithmetic and Logic Unit of the TMS320VC5402 allows the results to be saturated automatically in hardware, during the operation. This saturation is performed at no cost in terms of execution time.

4.5. Multiplication by 2 in 2's complement signed notation

A multiplication by 2 is performed in exactly the same way, whether it is in unsigned or in 2's complement signed notation. All bits are shifted left by one position. The MSB is lost, and 0 replaces the LSB. The only difference lies in the detection of an overflow. Since the MSB is also the sign bit in 2's complement notation, an overflow is detected if the sign bit that is lost is replaced by a sign bit of a different value. For a positive number, this happens when the bit to the right of the MSB is 1. In this case the multiplication by 2 of a positive number seems to produce a negative result. Conversely, an overflow happens when multiplying a negative number when the bit to the right of the MSB is 0.

In this case the multiplication by 2 of the negative number seems to produce a positive result.

4.6. Division by 2 in 2's complement signed notation

A division by 2 in 2's complement is performed in a slightly different way than in unsigned notation. The bit pattern is still shifted right by one bit position. The LSB is still lost. However the sign bit must be preserved. A 0 do not replace the sign bit, as is the case in unsigned notation. Instead the MSB is replaced by a copy of itself. This is called **Sign Extension**.

***Note**: The process of shifting a binary word to the left or to the right in order to multiply or divide it by a power of two is called an **Arithmetic Shift**. This distinguishes the operation from a **Logical Shift**, which is used to simply shift the bit pattern. There is no difference between an arithmetic and a logical shift to the left, except for the detection of an overflow. However, an arithmetic shift to the right performs a sign extension, which a logical shift to the right does not.*

Ex: For a positive number:

```
0010110101001      /2
   ->
0001011010100      /2
   ->
0000101101010      /2
   ->
0000010110101
       ...
```

Ex: For a negative number:

```
1010110101001      /2
   ->
1101011010100      /2
   ->
1110101101010      /2
   ->
1111010110101
       ...
```

5. OFFSET BINARY NOTATION

Offset binary notation is really a variant of the 2's complement notation. It has much of the same properties. This notation is obtained by translating the binary codes of the 2's complement notation by half a turn in the circular counter. For a 16-bit offset binary notation, the code representing 0_d is 8000_H. The code representing $+32767_d$ is $FFFF_H$, and the code representing -32768_d is $7FFF_H$.

Figure 5-6 shows the circular counter for this notation in 16 bits.

This notation is sometimes used by Analog to Digital Converters (ADCs) because it simplifies their hardware. It is also used to encode the exponent of a number in the IEEE floating-point notation standard.

© BRUNO PAILLARD

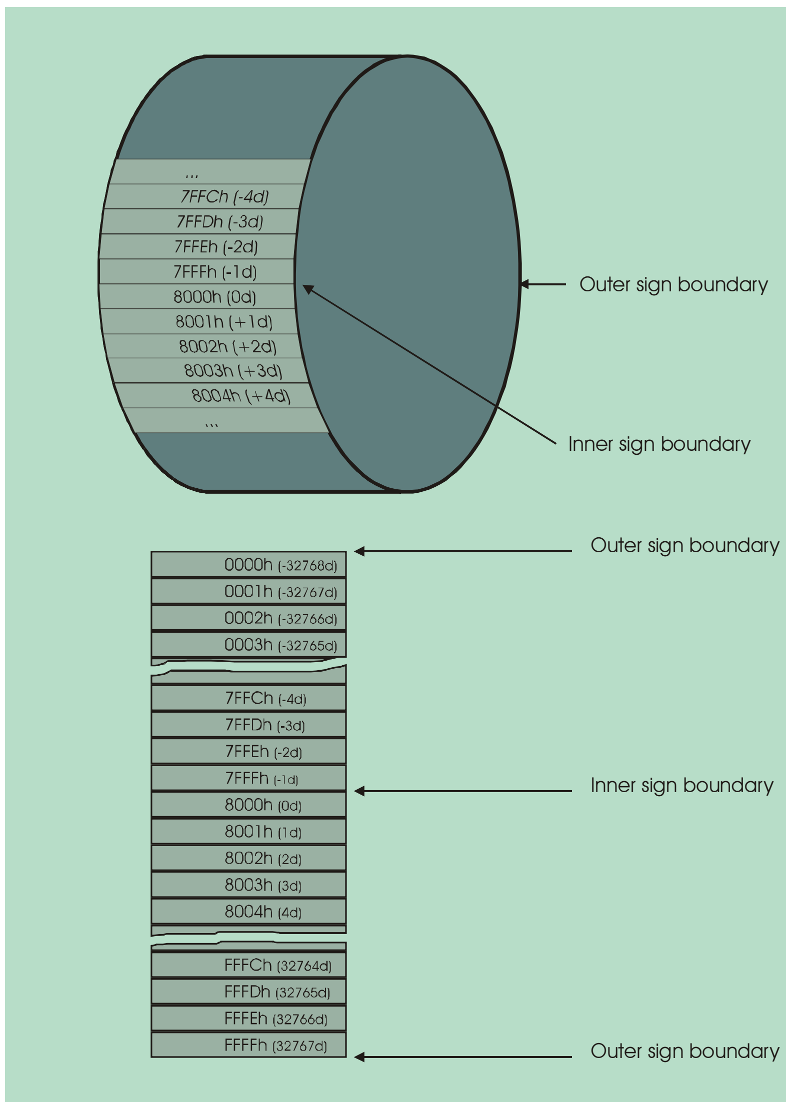


Figure 5-6

6. FRACTIONAL NOTATIONS

6.1. Fractional point

All the integer notations that we have seen so far assumed the position of the fractional point (the binary point) to be directly to the right of the LSB. To represent numbers with fractional parts, one simply has to assume that the binary point is located somewhere to the left of this position, dividing the binary code into an integer part to the left of the binary point, and a fractional part to its right. *Assume* is a key word here, because as we will see not much else distinguishes a fractional notation from an integer one.

The weights of bits that are to the right of the binary point are still powers of 2, however they are negative ones: $2^{-1}=1/2$, $2^{-2}=1/4$…etc.

For instance in the following example of an unsigned number, if the binary point is assumed to be to the left of the MSB, the value of the number is 0.6875_d.

```
1           0           1           1
x2^-1       x2^-2       x2^-3       x2^-4
0.5   +     0     +     0.125 +     0.0625
=     0.6875d
```

Fractional (binary) points can be used in exactly the same way for 2's complement numbers as they are for unsigned numbers. The assumed position of the binary point sets the dynamic range of the notation, as well as its resolution. Every time the binary point is shifted by one bit position to the left, the dynamic range and the resolution are both divided by two.

Processing fractional numbers does not require many specialized functions in the CPU. For the most part, fixed-point CPUs such as the TMS320VC5402 process integers and fractional numbers in exactly the same way. The interpretation of the results and their scale lies almost entirely on the software designer.

Fixed-point is often mistakenly understood as meaning *integer*. In fact *fixed-point* means that the position of the fractional point is assumed to be fixed and to be the same for the operands and the result of an operation. Fixed-point CPUs perform their calculations following this assumption. When the assumption is true, there is no difference at the CPU level between an integer and a fractional operation. However, when the binary point is not assumed to be at the same location for all operands, one of these operands may need to be scaled (shifted) by the required power of two before the operation, in order to align its binary point to that of the other operand. When the binary point of the result is not assumed at the same location as that of the operands, the result may have to be scaled by the required power of two after the operation. Shifting the values of operands or results to scale them and align their assumed binary points to prescribed positions is therefore a very common practice when performing fixed-point arithmetic. For this reason, the Arithmetic and Logic Unit of the TMS320VC5402 has a function allowing the arithmetic shift of any binary value by an adjustable amount, in just one cycle. In fact the shift can often be combined with another operation in the same cycle, essentially costing nothing in terms of execution time.

6.2. « Q » notation

The fractional notation can be applied to the 2's complement notation. The dynamic range and resolution are divided or multiplied by powers of two as the fractional point is shifted along the binary word.

For instance when the binary point is located to the right of the LSB (integer notation) a16-bit 2's complement signed notation has a dynamic range from -32768_d to $+32767_d$. When the fractional point is to the left of the LSB, the dynamic range goes from -16384_d to $+16383.5_d$, and the resolution is brought down to 0.5.

A **Qm.n** naming convention is used to describe 2's complement fractional notations. For this naming convention:

- *m* represents the number of bits to the left of the binary point (the number of points allocated to the integer part of the number), excluding the sign bit.
- *n* represents the number of bits to the right of the binary point (the number of bits allocated to the fractional part of the number).

The naming convention does not take the MSB of the number (sign bit) into account. A Qm.n notation therefore uses m+n+1 bits.

A Qm.n notation has a dynamic range that goes from -2^m to $+2^m$. The resolution is 2^{-n}.

© BRUNO PAILLARD

For instance:

- **Q0.15** represents a fractional 2'scomplement notation that has a dynamic range going from -1_d to $+1_d$. Its resolution is 1/32768. Q0.15 is often denoted **Q15** in short. In Q15 notation the code 8000_H represents -1_d, and the code $7FFF_H$ represents $+0.99996948_d$ (almost $+1_d$). The code $FFFF_H$ represents -0.00003052_d, and 0001_H represents $+0.00003052_d$. So compared to the 2's complement integer notation, everything is 32768_d times smaller.
- **Q3.12** represents a fractional 2'scomplement notation that has a dynamic range spanning -8_d to $+8_d$. Its resolution is 1/4096.
- **Q0.31** represents a fractional 2'scomplement notation that has a dynamic range spanning -1_d to $+1_d$. Its resolution is 1/2 147 483 648. Q0.31 is often denoted **Q.31** in short.

6.3. Scale of representation

There is no difference at the CPU level between a fractional and an integer representation. The difference is based on the concept of scale, which is almost completely in the head of the designer. The [implicit] placement of the binary point at various bit positions along the data word allows the scale of representation to be adjusted in factors of two. Compared to an integer notation, the values represented in a fractional notation are simply divided by a scale factor that depends on the position of the binary point, and therefore is a power of two (in Q15, the scale factor is 32768, in Q1.14 the scale factor is 16384…etc.).

Since the position of the binary point is implicit, and the scale factor is in the head of the designer, why not use an arbitrary scale that is not a power of two? For instance, the designer could decide that 16-bit 2's complement numbers between 8000_H and $7FFF_H$ really represent decimal values between -5_d and $+5_d$. In this case the scale factor would simply be 5/32768.

In practice any scale can be used to represent integer or fractional values. The software designer must simply keep track of the scales used at each step of a particular calculation, so that this calculation makes sense. Since the basic operations that are performed by the CPU normally assume the same scale for the operands and the result, the designer may have to use scaling operations at some point (or points) to adjust the scale of intermediate results as required. Changing the scale of an intermediate result may be required for a variety of reasons. The most common, is to make the best use of the available dynamic range. If the numbers are too small compared to the dynamic range, they will lack resolution. If they are too large, operations will often end up in overflows. There is therefore a trade-off to be reached between resolution and overflow. Adjusting the scale at each step of a complex calculation is very much akin to adjusting amplifier gains in an electronic signal chain, in order for the signals to stay above the noise level, while trying to avoid saturations at the same time.

Changing the scale of a number by a power of two is a very fast operation on any CPU, because it can be accomplished by a simple arithmetic shift of the bit pattern. Therefore software designers have a tendency to always use scale factors that are powers of two (scales that are defined by their "Q" denomination). A DSP is usually able to perform a multiplication by any value in a very short time (usually just one cycle). This greatly facilitates the use of arbitrary scale factors for the representation of fixed-point numbers. The TMS320VC5402 has this capability, and the cost of an arbitrary scaling in terms of execution time is only one cycle. This processor also has the capability to combine an

arithmetic shift to other arithmetic operations in the same cycle. So the cost of scaling by a power of two is usually free!

6.4. Integer and fractional multiplications

6.4.1. Unsigned integer multiplication

When two N-bit unsigned numbers are multiplied, the dynamic range of the result usually requires a 2N-bit notation.

For instance an unsigned 16-bit integer can range between 0 and 65535. The multiplication of two such numbers can range between 0 and 4 294 967 296, which requires a 32-bit notation.

For this reason, the Arithmetic and Logic Unit of processors that are able to multiply 16-bit numbers, usually has the capability to handle at least 32 bit- numbers, to be able to represent multiplication results without any loss of resolution. The ALU of the TMS320VC5402 actually has 40-bit accumulators, which provide an 8-bit margin that is used during multiply-accumulate operations that are common in signal processing problems.

6.4.2. Signed integer multiplication

The situation is slightly different when two signed numbers are multiplied. For instance when two 16-bit 2's complement are multiplied. Each number has a dynamic range that spans –32768 to +32767. The result of the multiplication of two such numbers only has a dynamic range requiring a 31-bit 2's complement notation. In other words, if the result is represented using 32 bits, its two MSBs are always equal and carry the same information. This is like having two sign bits.

6.4.3. Signed fractional multiplication

The Q15 notation is used very often in fixed-point digital signal processing. For this notation, the dynamic range spans -1_d to $+1_d$. When two Q15 numbers are multiplied, and the result is represented in 32-bit notation, it is natural to assume that the result is represented in Q31 notation, which also has a dynamic range spanning -1_d to $+1_d$.

However, the 32-bit result of the multiplication of two Q15 numbers should normally be interpreted as a Q1.30 result. This is for the same reason that two sign bits appear in a signed integer multiplication. To transform the result into Q31 notation, it must be left-shifted by one bit, which eliminates one of the sign bits. The TMS320VC5402 has a special mode that allows its ALU to automatically perform the left shift when Q15xQ15 - Q31 fractional multiplications are performed. This mode should only be used in this situation, because it will lead to a result that is twice what it should be if it is used when multiplying integer numbers.

6.5. Rounding and truncation

Let's look closely at a fractional 2's complement notation such as the Q13.2 notation. This notation has a dynamic range that spans -8192.0_d to $+8191.75_d$, with a 0.25_d resolution.

To obtain the integer part of a number using this notation, the first idea that comes to mind is to set the two fractional bits (the two LSBs) to zero. The effect of this operation is such that:

© BRUNO PAILLARD

- For a positive number, the value is adjusted to the integer number just below the initial value. For instance, the code 0005_H (1.25_d) is replaced by the code 0004_H (1.0_d).
- For a negative number, the fractional value is adjusted to the integer number also just below the initial value. For instance, the code $FFFB_H$ (-1.25_d) is replaced by the code $FFF8_H$ (-2.0_d).

This operation is called a **truncation**. It is very easy to implement, because is only requires setting the fractional bits to zero. This can be done efficiently by performing the AND of the bit pattern with a mask pattern that contains 0s at the bit locations corresponding to the fractional bits. It can usually be done in one cycle on any CPU. However, the result does not conform to the intuitive operation of extracting the integer part, because on negative numbers the result of a truncation is the integer number just below (whose absolute value is just above) the original fractional number. In the preceding example the truncation of $–1.25_d$ would have produced $–2_d$.

If +0.5 is added to the number prior to the truncation, the operation produces the integer number that is closest to the initial fractional number, whatever its sign. $+0.5_d$ is a binary code that contains all zeroes, except at the bit location just to the right of the binary point. The binary code representing $+0.5_d$ depends on the notation. For instance, in Q13.2, $+0.5_d$ is represented by the code 0002_H. For instance, using the two previous examples:

- 1.25_d (0005_H) + 0.5_d (0002_H) = 1.75_d (0007_H). The truncation of 0007_H produces 0004_H (1.0_d), which is the integer number closest to the initial value 1.25_d.
- -1.25_d ($FFFB_H$) + 0.5_d (0002_H) = -0.75_d ($FFFD_H$). The truncation of $FFFD_H$ produces $FFFC_H$ (-1.0_d), which is the integer number closest to the initial value -1.25_d.

It is easy to verify that this operation (addition of 0.5, followed by truncation) always produces the integer number closest to the initial fractional number. This operation is called **rounding**.

Rounding is a little bit more complex than truncation, but still is quite easy to perform at the CPU level. On the TMS320VC5402, there is even an ALU function to perform the operation in just one cycle.

The operation of extracting the integer part of a signed number is the most complex of all. It requires taking the absolute value of the number (testing its sign and taking the opposite if it is negative), then truncating, then changing the sign again (recovering the initial sign) if the initial number was negative. Even for processors that are optimized for high-performance calculation, such as the TMS320VC5402, this takes many cycles. Unfortunately, this is the behaviour that is defined as the default in the C language when converting from floats to integers. Not only is it often less interesting than rounding from a calculation point of view, it also cost more execution cycles on most CPUs.

7. FLOATING POINT NOTATION

For fixed-point notations, the scale of representation (or equivalently the position of the binary point) is implicit, and assumed to be fixed during calculations.

In the **floating-point** notation the position if the binary point is explicitly represented in the data word. An exponent field within the data word represents this position, or equivalently the scale of representation.

There are several standards used to represent floating-point numbers. Nowadays, the IEEE standard is the most widely used, and defines floating-point numbers of various widths (32 bits, 64 bits…etc.).

Figure 5-7 shows the example of an IEEE 32-bit floating-point number. It consists of three separate fields:

- **The mantissa**: Represents the fractional part of the number. The mantissa is represented on 23 bits in unsigned binary notation. It is normalized, which means that the bit pattern is left-shifted to eliminate all non-significant leading 0s. The exponent is corrected as necessary to take into account the normalization of the mantissa. The binary point is assumed to be to the left of bit No22. The MSB is therefore always one.
- **The sign**: A separate sign bit is applied to the mantissa (not the exponent).
- **The exponent**: The exponent represents the power of two that must be applied to the mantissa, to completely describe the number. The exponent is represented on 8 bits, in signed offset-binary notation. Exponents between –128 and +127 can be represented using this notation.

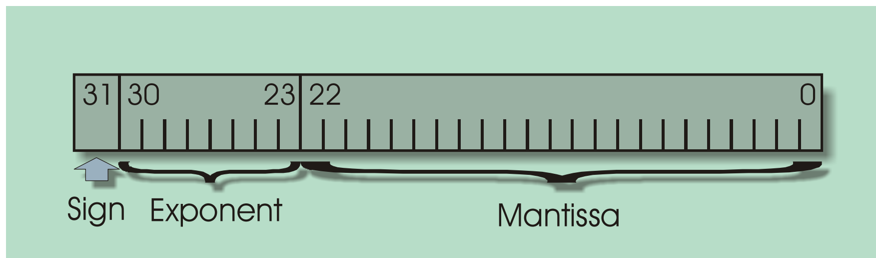


Figure 5-7

One advantage of the floating-point notation is that it allows the representation of a very wide dynamic range, without requiring as many bits as an equivalent integer or fractional notation. The trade-off between resolution and overflows is therefore easier to manage than for fixed-point notations. This notation also has disadvantages. For instance when adding two numbers of very different scales, many of the LSBs of the smaller number may fall below the resolution of the larger one, losing much of its own resolution. In extreme cases, all of its bits may fall below the resolution of the larger number and nothing is added. This problem often occurs when implementing digital filters having a large number of coefficients, because after the accumulation of a large number of product terms, the partially accumulated result can becomes very large relative to the individual product terms. Further product terms are added with a very poor resolution, and sometimes may not be accumulated at all.

Some CPUs have functions in their ALU to operate directly on floating-point data, as well as fixed-point data. Such CPUs are called floating-point CPUs. For such CPUs, the added complexity of the ALU accounts for a significant portion of the silicon area of the processor. These machines have a higher cost, or at equal cost pack less peripherals on their silicon area (less RAM capacity for instance) than equivalent fixed-point processors.

On the other hand, it is always possible to implement floating-point arithmetic by software on a fixed-point CPU. This is done at a tremendous cost in terms of

computational time however. A floating-point multiplication that can be done in one cycle on a floating-point DSP has an execution time in the tens of cycles on a fixed-point DSP.

In practice, the trade-off between complexity and power consumption and cost is such that if the use of fast floating-point arithmetic is critical to the application, it is usually advantageous to use a floating-point CPU. Fortunately, many signal processing problems do not require floating-point arithmetic if adequate care is given to the problem of scaling throughout the process. Some problems are even better served with fixed-point arithmetic.

8. BCD NOTATION

8.1. Description

The Binary Coded Decimal (BCD) notation is really a decimal notation that uses the symbols of binary. In BCD, a 4-bit binary number represents each decimal digit.

Number	Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 5-3

For instance, the number 193_d is represented by the following binary code :

```
1          9          3d
0001       1001       0011b      =     000110010011b
```

8.2. Uses of the BCD notation

Because it is a decimal notation rather than a binary one, BCD is used in calculations that benefit from being perform directly in decimal. For instance pocket calculators often work in BCD.

One advantage of BCD is that the results of calculations are easy to convert into decimal. This is advantageous in many embedded systems that must display these results, and have limited hardware and software resources to do the conversion. Examples range from gas pumps to weight scales, digital pressure gauges, digital thermometers…etc.

A disadvantage of the BCD notation is that it makes a fairly inefficient use of the available bits. For instance, an unsigned 12-bit BCD notation can only represent

numbers between 0_d and 999_d. Using an unsigned binary notation, numbers between 0 and 4095_d can be represented with the same number of bits.

Another disadvantage is that calculations are not done in BCD the way they are done in binary. BCD calculations are more complex, and require correction steps so that the result stays in BCD. For instance, the addition of two BCD digits requires a correction step to take into account the effect of the decimal carry. The following example shows this correction process when adding 8_{BCD} to 5_{BCD}.

1000_b	8_d	1st operand
0101_b +	5_d +	2nd operand
1101_b	13_d	Binary result
0110_b +	6 +	Correction
0001 0011_b	13_{BCD}	BCD result

The result of the first binary addition gives 1101_b (D_H). This result is outside the bounds of a BCD digit. It must be corrected by adding 6, and by propagating the decimal carry to the next BCD digit. The schematic of an elementary one digit BCD adder is given in figure 5-8.

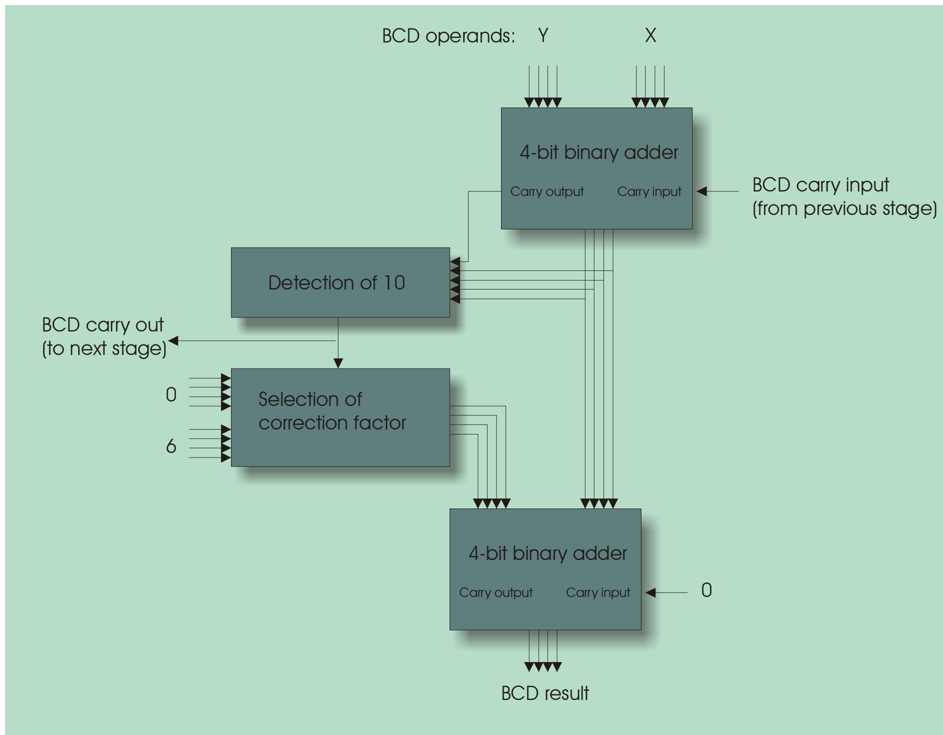


Figure 5-8

© BRUNO PAILLARD

9. ASCII CODES

Microprocessor systems do not just process numbers. They also need to process alphanumeric characters. The most widely used notation to represent alphanumeric characters is the ASCII code (American Standard Code for Information Interchange). The ASCII code was originally developed in 1968, mostly to support data exchanges in large information networks, such as those that were used by the news agencies at the time.

Originally there were many national variants of the ASCII code (German, Danish, Canadian, Spanish, French…etc.). However, because a common standard must be used to exchange data between different countries, most national variants have been abandoned in favour of the standard character set.

The **standard ASCII** character set in table 5-4 represents each character by a 7-bit code. The set includes uppercase and lowercase alphabetical characters. It also includes special and punctuation symbols, as well as control characters that are designed to perform an operation on the terminal on which they are received. Carriage Return (CR) for instance causes the cursor to jump to the next line; Back Space (BS) causes the cursor to go back one character. BEL causes a bell (or another sound) to ring.

In the mid 1980s, the ISO 8859 extended the standard ASCII character set by 128 characters (it uses 8 bits instead of 7). The ISO 8859 character set (or **Extended ASCII**) allows the representation of accentuated characters necessary in many languages. While the lower 128 characters of the ISO 8859 character set is the same as the standard ASCII set, the ISO 8859 standard defines various sets for the upper 128 characters. In other words, the encoding of accentuated and national-specific characters varies from system to system, as can be experienced by users trying to send or receive emails containing such characters.

Because extended ASCII does not solve the problem of compatibility between the various nationally specific character sets, a new standard called **Unicode** has been established in 1991. The standard uses 16 bits to represent alphanumeric characters. However, Unicode is not widely used. Unicode alphanumeric data takes twice as much space as ASCII data. Furthermore, Unicode conversion tables are significantly larger than a simple ASCII table. Unicode is well adapted for use in computer systems having substantial hardware and software resources. In many embedded systems that have only limited resources, the ASCII code is the norm.

There is no special function in a CPU to handle or process ASCII characters. All character processing is performed using the normal functions of the CPU and the ALU.

9.1. Character strings

ASCII codes can be assembled into character strings that represent complete words or sentences. The most widely used technique for constructing a character string consists in placing ASCII codes in successive memory locations, followed by a termination character designating the end of the string. The termination character is system-dependant. Some systems use the NULL character (00_H). Others use the End-Of-Text character (04_H).

The development tools for the TMS320VC5402 use the NULL character as the string terminator.

For instance, the following byte sequence: 48_H-65_H-$6C_H$-$6C_H$-$6F_H$-00_H represents the character string “Hello”. The presence of the 00_H character at the end of the string is

very important, without it the following memory locations would be interpreted as part of the string, up to (and it can be a long time…) the next encounter of the code 00_H.

NUL 00	DLE 10	SP 20	0 30	@ 40	P 50	` 60	p 70
SOH 01	DC1 11	! 21	1 31	A 41	Q 51	a 61	q 71
STX 02	DC2 12	" 22	2 32	B 42	R 52	b 62	r 72
ETX 03	DC3 13	# 23	3 33	C 43	S 53	c 63	s 73
EOT 04	DC4 14	$ 24	4 34	D 44	T 54	d 64	t 74
ENQ 05	NAK 15	% 25	5 35	E 45	U 55	e 65	u 75
ACK 06	SYN 16	& 26	6 36	F 46	V 56	f 66	v 76
BEL 07	ETB 17	' 27	7 37	G 47	W 57	g 67	w 77
BS 08	CAN 18	(28	8 38	H 48	X 58	h 68	x 78
HT 09	EM 19	) 29	9 39	I 49	Y 59	i 69	y 79
LF 0A	SUB 1A	* 2A	: 3A	J 4A	Z 5A	j 6a	z 7a
VT 0B	ESC 1B	+ 2B	; 3B	K 4B	[5B	k 6b	{ 7b
FF 0C	FS 1C	, 2C	< 3C	L 4C	\ 5C	l 6c	| 7c
CR 0D	GS 1D	- 2D	= 3D	M 4D	] 5D	m 6d	} 7d
SO 0E	RS 1E	. 2E	> 3E	N 4E	^ 5E	n 6e	~ 7e
SI 0F	US 1F	/ 2F	? 3F	O 4F	_ 5F	o 6f	del 7f

Table 5-4

© BRUNO PAILLARD

Software Development Tools

- 1. Cross development tools
- 2. Code development methodology
- 3. Code testing

1. CROSS-DEVELOPMENT TOOLS

Microprocessor software is developed using cross-development tools. The term "cross-development" is used to indicate that the development is done on one system (often a PC), while the code is developed to be executed on another system (namely the **target system**).

Definition:	**Target system**: The target system is the system for which the code is developed. In our case, the target system is the Signal Ranger board, and the target CPU is the TMS320VC5402.

Definition:	**Host system**: The host system is the system that supports the development tools. Often the host system also supports communications with the target system, in order to download code into it, and test it. In our case, the host system is a PC.

Definition:	**Cross-development tools**: Cross-development tools include all the software that runs on the host system to allow the development and test of the target system's software.

Cross-development tools always include:

- **A text editor** The text editor allows the developer to write the target code.
- **An assembler** The assembler transforms the target assembly code into machine code.

Cross-development tools can also include:

- **A linker** In modular development tools, a linker is used to select the locations in target memory of the various sections of code and data. The linker also allows the resolution of symbolic references in the code, to produce the final executable code. Absolute assemblers do not need a linker, but most professional modular development tools do.

- **A compiler** A compiler transforms a high-level language code description (such as a C or C++ program) into low-level assembly language.
- **A simulator** A simulator allows the execution of code to be simulated on the host system. It provides much of the same functions as an emulator or debugger. It is more limited because it usually is not capable of simulating the hardware of the target system other than the processor and memory. It is used when the developer has no access to the target system.
- **A loader** A loader performs the loading of the executable code into the RAM memory of the target system. Once the code is loaded into RAM, it can be executed.
- **A FLASH or EEPROM programmer** This host software programs the EEPROM or FLASH ROM of the target system with the executable code that has been developed. Nowadays, a FLASH or EEPROM programmer relies only on the target hardware, and on a communication link between the host and target systems to perform the programming *in-situ*. No specialized programming equipment is required, as used to be the case when EPROMs were the norm. Once the EEPROM is programmed, the target code can be executed from it.
- **A debugger** A debugger allows the execution of the code on the target system in a controlled manner. For instance, it allows step-by-step execution, the observation and modification of CPU registers and RAM locations during execution…etc. A debugger may include a loader, or the two can be separate tools.
- **An emulator** An emulator provides much of the same functions as a debugger. The difference is that in the case of a debugger, these functions are supported by software running on the target processor. In the case of an emulator, these functions are supported by specialized hardware on the target system. Nowadays, emulator functions are supported by an *instrumentation hardware* that is part of the CPU. This hardware facilitates the development of code directly on the target system. Also, because these functions are supported by hardware, they generally interfere less with the user code than in the case of a debugger.

***Note**: The development environment used to develop code for the TMS320VC5402 is called **Code Composer Studio**. This software is provided by Texas Instruments, and includes most of the tools described above. Because the communication with the target system is very dependant on the target system's hardware, the loader, the FLASH programmer, and the debugger are generally provided by the company that manufactures the target system, rather than by the company that manufactures the CPU or provides the rest of the development tools. In our case a mini-debugger for the Signal Ranger board, including a loader, has been developed by Université de Sherbrooke and is provided for code testing.*

Figure 6-1 presents the hierarchy of the development tools:

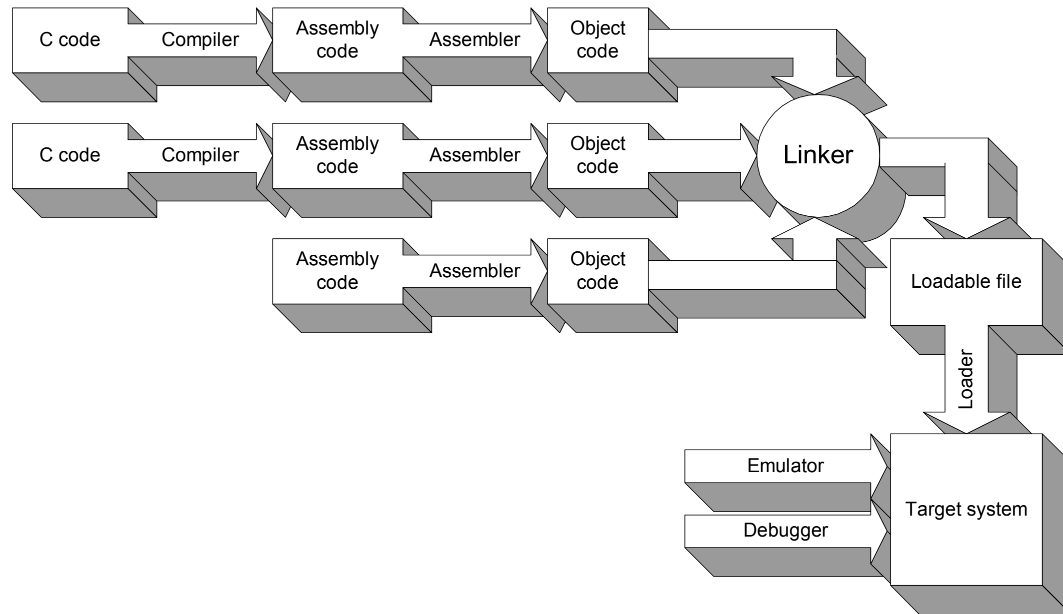


Figure 6-1

Definition:	**Source code:** The source code (or source file) is a text file containing the high-level language (usually C or C++) or assembly language description of the code. The software designer generates the source code using a text editor. The source code for a complete application may be distributed on several files called **modules**. In figure 6-1, the boxes named C code and Assembly code are all source files. Three modules are represented in the figure.

Definition:	**Object code**: Object code (or object file) represents machine code that has been assembled by the assembler. The object code contains the machine codes that will be loaded into the system's memory prior to execution. However, the location in memory of this machine code has not yet been decided, and all the symbolic references have not been resolved.

Definition:	**Executable code**: The executable code, **executable file**, or **loadable file** is a file containing all the information necessary to load the machine codes into the memory of the target system. This file contains at least the binary values to be loaded, and the load addresses. In may also contain symbolic information such as the names of the symbols used in the source code and their values. Such symbolic information is useful in a debugger. It helps the debugger provide access to the variables and functions defined in the code by using their symbolic names. Code Composer Studio generates an executable file in the COFF format (Common Object File Format). This file contains loading information, as well as very complete symbolic information. The linker generates the executable code from the various object files of the project.

2. CODE DEVELOPMENT METHODOLOGY

2.1. High-level language development

For a variety of reasons, most of which were discussed in Chapter 2, it is extremely rare that embedded software can be entirely developed in a high-level language. Software modules that are critical, either because of code size, execution time, control of the execution, or other reasons are often developed in assembly language. Development in assembly language constitutes the minimum skill that the embedded designer must possess. Of course some software may be developed in a high-level language such as C or C++. Often the shell of the code, the user interface, modules where complex logic must be implemented, and more generally all non-critical modules may benefit from being written in a high-level language. Among other things, it allows developers unfamiliar with the CPU to understand these modules. In large projects developed by teams of designers, it allows the development of non-critical modules to be undertaken by developers that do not possess the specialized skills necessary to develop in assembly language. However, the modules developed in a high-level language must always be interfaced with (call and be called by) code developed in assembly language. This means that to be able to develop code in C, the software designer must not only be skilled at C and assembly language development, but must also know in every detail all matter pertinent to the interfacing of C and assembly. Such matter includes the C calling conventions, how the C language uses the CPU registers, how the C language manages the stack and heap, context protection conventions… etc. In short, the designer must have a pretty deep working knowledge of the C compiler.

Writing code in a high-level language always represents an additional difficulty in the context of embedded software design.

When a software module is written in a high-level language, a **compiler** is used to transform this code into assembly code. The assembler then processes the assembly code exactly as if the developer had generated it directly.

2.2. Assembly language

The CPU executes **instructions codes** that are stored in memory in the form of binary codes. These codes contain **Opcodes** that define the operation to be performed and **Operands** that define the entities on which the operations are done. Operands can define memory addresses, constants, CPU registers…etc. For the TMS320VC5402, a complete instruction code (Opcode + Operand(s)) may be expressed by one or several 16-bit words.

The set of all instructions that a CPU can execute is called its **instruction set**.

***Note**:* *The complete instruction set of the TMS320VC5402 is described in the help menu of Code Composer Studio, under “Contents – TMS320C54x Mnemonic instruction set summary”*

There are two aspects to an instruction or program:

- The first aspect is the textual description that the designer uses to write the program. This is the **assembly source code**. In the assembly source code, the instructions are described by their **mnemonic** name. A mnemonic name, or

mnemonic, is a short name (usually 3 or 4 letters) that describes the basic operation performed by the instruction.

- The other aspect is the physical description of the program in the memory of the target system. This is this description that is used by the CPU to execute the program. This description is called **machine code**.

The **assembler** is the main software tool that transforms the assembly source code into machine code, although the linker also has a role to play in the transformation.

The relationship between the assembly code and the machine code is very direct. This is why writing in assembly language provides the developer with so much control over the execution. However, the relationship is not one-to-one. An opcode represents not only the mnemonic of the instruction, but also the way the instruction accesses its operands. This is called the **addressing mode**. Most instructions support several addressing modes. Depending on the addressing mode that is used, the same mnemonic can be represented by a number of opcodes.

2.3. Disassembly

Code composer studio provides a disassembler as part of the simulator. This disassembler reads instruction codes in memory, and presents the developer with the corresponding assembly source code. Because of certain ambiguities that the disassembler cannot resolve, the assembly code that is presented may not be exactly the same as the original source code. However, it is code that would execute in exactly the same way.

For the TMS320VC5402, an instruction is sometimes represented by more than one memory word. In such a case, the first memory word is the opcode, the following word is an operand. For the disassembly to be done properly, it is imperative that the first address specified as the starting point of the code represent an opcode. Otherwise, the disassembler may begin disassembling an operand, and much of the rest of the code will not make sense.

2.4. Link

Two major advantages of modern modular development tools such as Code Composer Studio are:

- They allow the use of symbols in the source code, to represent variables (memory locations), constants or labels (program addresses).
- They allow the development to be split into separate source modules. Each module can be independently compiled and assembled, and all modules can then be linked together to produce the complete executable file.

To support these two essential features, the development is done in two separate steps: **Assembly**, then **Link**.

2.4.1. Object file

Lets take the example of an application written in two separate modules called Module_A, and Module_B. The function *Add32* is defined in Module_A, and referenced (called) in Module_B. Similarly, the variable *Var16* is defined and allocated in Module_B, but referenced (accessed) by the code of Module_A.

When Module_A is assembled, the assembler is able to transform most of the assembly instructions into machine code. However, the assembler does not know the exact address of the variable *Var16*. Similarly, when Module_B is assembled, the assembler does not know the address of the function *Add32*, and cannot replace the symbolic call address by the absolute address in the machine code of Module_B.

More generally, the loading memory addresses of the various modules (code and variables) are not yet decided at the time of the assembly. Any branch or call cannot be completely transformed into machine code, because the branch address is not known at that time. The instruction codes corresponding to accesses to variables cannot be fully assembled because the addresses of the variables are not known either.

The result of the assembly pass is an **object code**, or **object file**. An object file contains the machine codes corresponding to most of the instructions. However the absolute loading address of the code in the target system's memory has not been determined yet. All references to program addresses (such as branches or calls) are not resolved at this point. The addresses of variables in the target memory space have not been determined, so all references to these variables are not resolved either. The object file contains no information about the absolute loading addresses of code or variables, and all the instruction codes referencing program or variables addresses are not fully defined. However, the object code contains all the information necessary to resolve the references once all the pieces are linked together. For this reason, it is also called **relocatable code**.

Once all the modules have been assembled, they are transmitted to the linker. From indications provided by the developer, the linker determines the memory locations of all the sections of code and variables that make-up the project. It then combines all the symbolic references found in the various object files to resolves all the symbols (calculate their value), and finally transforms the code into an absolute (**relocated**) executable code. The executable code contains the binary words that constitute the machine codes, and their loading addresses, including complete instruction codes for branches, calls, and accesses to variables that were not finalized in the object code. It also may contain symbolic information in the form of a table listing all the symbols used in the source code, and their value.

The linker allows the developer to determine the location of code and variables in the memory of the target system. Different targets have different amounts of memory, located at different addresses. The developer must provide information to the linker, indicating which memory blocks are available in the system, and in which memory block each section of code or variables should be loaded. Code Composer Studio has a *visual linker* that allows the developer to specify this using a graphical interface.

***Note**: Some low cost development tools do not provide a separate linker. The link operation is performed directly at the end of the assembly operation. These tools are called **absolute assemblers**. A separate linker can work on separate object modules, and resolve symbolic references between the modules. On the other hand, an absolute assembler must generate the executable code directly after the assembly. For the link pass to have access to all the symbolic references, the object code must be located in a unique module. Absolute assemblers do not support the modular development of code.*

2.4.2. Symbols

Symbols are used in the assembly code to define constant values, addresses of static variables and labels. Symbols are handled by the linker, which resolves their values. All

© BRUNO PAILLARD

modular development tools that provide a separate linker recognize 2 basic types of symbols:

- **Local symbols**: By default symbols are local to the module in which they are defined. This means that they are not recognized outside of this module. When a reference to a local symbol is made in a module, the assembler expects to find the symbol declaration in the same module. Otherwise it generates an error. When 2 local symbols with the same name are declared in 2 separate modules, they are assumed by the linker to represent 2 separate entities.
- **Global symbols**: When symbols are declared global, they are recognized across all the modules of the project. Global symbols can be declared in one module and referenced in another. Global symbols usually represent the names of functions and static variables that are used by several modules. To be global, a symbol must be declared so in the code using the appropriate directive. When a reference to a global symbol is made in a module and the symbol declaration is not in the same module, the assembler assumes that the declaration is in another module and does not generate an error. The linker will generate an error however if it does not find the symbol declaration in any module.

2.4.3. Sections

To be able to specify separate load addresses for separate code and data segments, the developer can group them into separate sections. Sections are defined by using appropriate directives in the source code. Some sections have standard names, such as ".text" for code, ".bss" for uninitialized data or ".data" for initialized data. The developer can also choose specific names for project-specific sections. In each source module, the developer may specify several sections, and chooses the sections to which belong each code or data segment.

The linker processes each section that it finds in each source module. These are called **input sections**. Sections from separate source modules may have the same name. For instance, each source module generally includes a section named ".text" that contains code. The linker groups all the sections that have the same name in separate modules, and constructs **output sections**. Output sections are code or data segments that can be assigned a specific location in the target memory.

2.4.4. Link files

The developer must specify to the linker the loading addresses of each output section that it generates. In Code Composer Studio, this is done using a graphical interface called **Visual Linker,** through a drag-and-drop process. Two files support the operation of the visual linker:

- **A memory description file**: This file describes the memory map of the target system. It specifies which types of memory reside at which addresses, and in which memory spaces. Occasionally, it may specify that some memory zones are reserved, and should not be used to load code or data. This file is target-system-specific. For the Signal Ranger board, it is called *SignalRanger.mem*.
- **The link "recipe" file**: This file contains all the information specifying where each output section resides in the memory map of the target

system. This file is project-specific. In Code Composer Studio, it has a ".rcp" extension. The visual linker generates the file, following the graphical specifications given by the developer.

2.5. Edition of assembly code

Assembly code is written using strict syntax rules. These rules are described in Code Composer Studio's online help, under *Content - Code Gen Tools – Assembler – Assembler Description – Understanding the Source Statement Format.*

2.5.1. Format of an instruction line

Each line of code (instruction or directive) is divided into fields. Each field contains a specific element of the code. Fields are separated by spaces or tabs. The following fields are defined:

- **Label field**: Begins in column 1. This field is optional, and may contain a symbol that represents the address of the instruction that follows on the same line. Labels may have up to 32 characters, and must not begin with a number.

***Note**: The developer must be careful not to write a mnemonic in column 1, because it would be interpreted as a label by the assembler.*

- **Operation field**: Begins just after the label field. This field may contain a mnemonic or a directive.
- **Operand field:** Begins just after the operation field. It may contain one or several operands, as required by the instruction or directive. Operands are separated by commas, rather than spaces or tabs.
- **Comment field:** Begins just after the operand field if there is one, but must absolutely be separated by a semicolon. All the text following the semicolon is considered to be a comment, and is disregarded by the assembler. A comment can begin in column 1 if the first character is an asterisk or a semicolon. In this case the whole line is disregarded. This is useful to separate the code into different visual segments.

2.5.2. Constants

The assembler supports the specification of constants in the following formats:

- **Decimal** Default format Ex. 10
- **Hexadecimal** The prefix "0x" must be added Ex. 0x10
- **Binary** The suffix "b" or "B" must be added Ex. 10b
- **Octal** The suffix "q" or "Q" must be added Ex. 10q
- **Floating-point** A floating-point number is expressed by using a fractional point, and if necessary an exponent using the prefix "e" or "E Ex. 1.10E-2.
- **ASCII** An alphanumeric character between quotes represents the corresponding 8-bit ASCII code. Ex. 'A'

© BRUNO PAILLARD

- **Character string** A succession of alphanumeric characters between double quotes represents a character string. Ex. “Hello”
  The assembler adds the NULL (00_H) termination code at the end of the string.

2.5.3. Symbols

Symbols may be used to represent labels (branch or call addresses), variables or constants. The following limitations apply to the definition of symbols:

- All symbols except labels may have up to 200 characters. Labels are limited to 32 characters.
- Uppercase and lowercase are differentiated.
- The first character of a symbol may not be a numeral.
- A symbol may not contain spaces or tabs.
- Some symbolic names are reserved and may not be used by the developer. Reserved names obviously include instruction mnemonics, directives, and other symbols that are used by the assembler.

2.5.4. Expressions

In some cases it might be useful to perform operations on constants to generate new constants that depend on the initial ones. Such operations are called **expressions**, and can be defined using a number of arithmetic and Boolean operators. The list of available operators is described in Code Composer Studio’s on-line help, under: *Contents – Code Generation Tools – Assembler – Assembler Description – Expressions.*

***Note**: It is important to recognize that expressions are performed on constants, and generate new constants, both of which must be completely resolved before the code is executed. In other words, an expression is evaluated at assembly time, rather than at execution time. Expressions are used to define relationships between constants, and to allow the calculation of some constants to be automatically derived from others, by the assembler. A modification to an initial constant can then lead to the automatic recalculation of all the constants that are derived from it through the use of expressions.*

2.5.5. Directives

Directives provide information or symbolic commands to the assembler in order to specify assembly or link parameters. No machine code is generated in response to a directive. There are directives to:

- Define sections and associate code or data to them.
- Allocate (and sometimes initialize) static variables.
- Define local and global symbols.
- Control the appearance of the source code.
- Apply conditions to the assembly of some code sections
- Define macros.

All the directives and their syntax are described in Code Composer Studio's online help in *Contents – Generation tools – Assembler – Assembler Directives*. The most common tasks defined by directives are:

- Define sections (".text", ".sect", ".usect"…etc.)
- Define symbols (".set"…etc.)
- Declare global symbols (".global", ".def", ".ref"… etc.)
- Allocate and initialize variables in memory (".int", ".long", ".word", ".space"…etc.)
- Allocate uninitialized variables in memory (".bss"…etc.)

2.5.6. Macros

The concept of a macro is very simple. A macro represents a segment of code using a symbolic name. After the macro definition, each time the assembler encounters the macro symbol in the code, it replaces it by the corresponding code segment. A macro can represent one or several lines of codes.

Macros are used to define code segments that are used often in the code. They can also be used to define complex functions that are constructed from several instructions.

Macros can use arguments that are declared to add flexibility to their utilization. The arguments specified when the macro is invoked replace the arguments declared in its definition. Of course macro arguments are resolved at assembly time. They are completely defined at execution time.

There is often some confusion between a macro invocation and a function call. The two concepts are very different. The code of a function only appears once in the code, even if the function is called from several places. On the other hand, the code of a macro is copied in the program every time the macro is invoked. The CPU does not branch anywhere in response to a macro, it simply executes the code of the macro that is copied in place every time the macro is invoked. Macros and functions may have arguments. However, in the case of a function, arguments are dynamic entities that take variable values during the execution. Arguments of macros are constants that are resolved at assembly time and are fixed at execution time.

The syntax of a macro is defined in Code Composer Studio's online help, under *Contents – Code Generation Tools – Assembler – Macro Language*

3. CODE TESTING

Three types of hardware and software tools are provided to test embedded code:

- Simulators.
- Evaluation boards and debuggers.
- In-circuit emulators.

The function of each of these tools is essentially the same: Allow the execution and testing of the embedded code in a controlled environment.

© BRUNO PAILLARD

3.1. Simulator:

A simulator is an entirely software tool that executes on the host system. It simulates the execution of the code on the target CPU. A simulator allows step-by-step execution, breakpoints, the observation and modification of memory locations and CPU registers during the execution… etc.

Its functionality is limited in the sense that it cannot simulate the hardware that is specific to the target system. At its most basic level, it simulates the CPU and program and data memory. More sophisticated simulators may also simulate in a limited way the operation of on-chip peripherals, but nothing more. It is a great tool to develop and test pure calculation code. However it is poorly adapted to test code that manages specific target peripherals.

3.2. Evaluation board and debugger

Systems manufacturers provide **evaluation boards** or **evaluation kits** to allow developers to familiarize themselves with the CPU and development tools used in the target system, as well as to develop code. Such boards are usually very flexible and contain hardware resources that can support a variety of applications. Evaluation boards can be used to develop the entire target code, to the extent that the on-board hardware allows it. Some evaluation boards include a prototyping area that can be used to interface target-specific peripherals that are required for the development and test of the code.

If the on-board hardware allows it, an evaluation board can even be used to produce a prototype of the target system that can be helpful in the early stages of system design.

Work on an evaluation board is often valuable, because it allows the development of the target software to proceed concurrently to the development of the target hardware. It can be used to validate certain aspects of the hardware/software design, and it may even lead to improvements of the hardware design early in its development stage (which is always the best time…)

A debugger is often provided with the evaluation board to allow the downloading and test of the code on the evaluation board. A debugger allows much of the same functions that are generally provided by a simulator or an emulator: step-by-step execution, breakpoints, the observation and modification of memory locations and CPU registers during the execution… etc. These functions, as well as communication with the host system are supported by software that executes on the evaluation board, as well as on the host system. On the evaluation board, this software uses up some of the target resources, such as CPU time, memory, interrupts…etc. A good debugger uses these resources sparingly, in order to minimize the interference between the debugger and the tested code. However, it is impossible to completely avoid using resources, and the developer must always cope with some degree of interference.

The Signal Ranger board is an evaluation board. It is provided with a mini-debugger that can be used to test DSP code.

3.3. In-circuit emulator

Historically, in-circuit-emulators were designed using discrete logic and/or microprocessors, to completely replicate the operation of a CPU. These systems could be inserted in place of the target CPU through the use of a socket. They allowed the developer to probe signals in the emulator, or to stop and resume code execution (breakpoints, step-by-step…) using the emulator's hardware.

This solution had considerable disadvantages. The cost of the emulator hardware was high. There were power supply problems, because the power consumption of the emulator was often higher than the consumption of the original CPU. Also, as CPU speeds increased, it became increasingly difficult to maintain the integrity of the signals that had to be transported by a long connector. Line reflections, ground bounce and cross-talk problems unavoidably increased with higher clock frequencies.

Nowadays, most microprocessors include an instrumentation logic that is integrated on the chip. This logic that is added to the core functionality of the CPU by the manufacturer, allows the probing and modification of various CPU registers and memory locations. It also allows the execution to be stopped and resumed, which supports step-by-step execution and breakpoints. The JTAG (Joint Test Access Group) standard, established in 1985 by European manufacturers, specifies a set of CPU signals (and behaviours) that are used to access these test functions of the CPU. Today, most CPU manufacturers follow the JTAG standard.

This solution has the following advantages:

- It allows the code to be executed in a controlled manner, by providing all the functionality usually found in a debugger (step-by-step, breakpoints…etc.)
- It allows the debugging of the code to occur in real time, and in the real-life environment of the target system. It may facilitate the detection and correction of problems that only occur on the real target system. Timing problems related to the operation of target-specific peripherals, may only appear in a real-life situation for instance.
- Being implemented in hardware, the debugging functions interfere much less with the target application than when they are implemented in software, such as in a debugger. There is still some amount of interference, for instance stopping the execution of the code obviously has an impact on the behaviour of the system. However, the interference is really minimized compared to a software debugger.
- The cost impact on the target hardware is really limited to the presence of a standard header on the Printed Circuit Board, to allow the connection of a host to the JTAG signals. The cost is so low that the same target hardware used to develop the code is often used in production. There is no need to design a lower-cost version that would not support JTAG testability in production.
- In fact JTAG can also be used in production to test the hardware.
- On the host side, a low-cost connection port is often used to control the JTAG signals from a host system such as a PC. Most of the complexity is implemented in the form of software running on the host PC.
- In systems that contain FLASH or EEPROM, the JTAG connection can also be used to program the memory during development, as well as in production.

© BRUNO PAILLARD

The TMS320C5402: Architecture And Programming

- 1. Architecture
- 2. CPU
- 3. Operands and addressing modes
- 4. Branch instructions
- 5. Repeats
- 6. Other conditional instructions
- 7. The stack
- 8. Functions
- 9. Interrupts

1. ARCHITECTURE

The TMS320VC5402 is a 16-bit fixed-point DSP that can run at up to 100MHz. It includes the following on-chip peripherals:

- 16 K words of RAM. This 16-bit wide RAM can be accessed twice per cycle. It can be mapped into both the data space and the program space. Because it is so fast, we strongly recommend executing the DSP code from it, and to use it to store often accessed variables and constants.
- 4 K words of ROM. This masked ROM contains a bootloader code that executes when the DSP is reset. The masked ROM also includes sine tables that may be used in signal processing applications.
- 2 synchronous serial ports. These high-speed serial ports are called Multi-Channel Buffered Serial Ports (McBSPs). They support a variety of serial interface standards, and are especially useful to interface analog input/output devices to the DSP.
- A Host Port Interface (HPI). This device allows the DSP to be interfaced as the peripheral of another (host) processor. It gives the host processor access to the on-chip RAM of the DSP, which can be used to support communications between the DSP and the host. On the Signal Ranger board, the DSP is interfaced as the peripheral of an AN2131 USB controller. The USB controller is a small microcontroller that is used to manage the board’s power supply, and support communications with the PC.
- 2 timers. These 2 timers can generate adjustable frequency periodic signals that may be used to drive some types of peripherals, such as ADCs and DACs. The timers can also be used to precisely measure the time between software events.

- A Direct Memory Access (DMA) controller. The DMA controller can manage direct data exchanges between several peripherals, including serial ports and memory. The DMA controller can synchronize exchanges between peripherals without any DSP intervention.
- General-purpose I/O signals. The BIO input and XF output can be used to read and write digital signals under software control.
- A clock generator. The clock generator is driven by a low-frequency (around 10MHz) crystal oscillator. It includes a software programmable PLL that can be used to dynamically (while the DSP is running) adjust the main DSP clock frequency. The trade-off between computational speed and power consumption can be adjusted dynamically, depending on the requirements of the application. At the fastest, the DSP can be run at 100MHz.
- A wait-state generator. This peripheral is used to adjust the amount of wait-states introduced when the DSP accesses external devices. Various amounts of hardware or software wait-states can be introduced, depending on the type of peripheral that is being accessed.

The TMS320VC5402 has an advanced Harvard architecture that has independent program and data busses. Internally the CPU has one program bus and three separate data busses. This multiplicity of busses allows it to perform complex operations such as a multiplication in just one cycle. A 1-cycle multiplication requires the CPU to access the opcode and two operands at the same time (in parallel).

To minimize its pin count, the TMS320VC5402 has only one external bus interface. This interface supports accesses to devices mapped in program space and in data space. A third space is also defined as the I/O space, which has different timings optimized for input/output peripherals. Since there is only one external bus system, three exclusive space-selection signals are provided to indicate which particular space is targeted by the access.

- **PS (program space):** This signal is activated for Fetch (read) cycles in external memory. It is also activated during the execution of special instructions that can read or write the program memory.
- **DS (data space):** This signal is activated during read and write cycles in external data memory. Many instructions support reads or writes in the data space.
- **IS (I/O space):** This signal is activated during read and write cycles in the I/O space. Two special instructions support these accesses.

Only one external access can be performed at each clock period. In fact many external devices require wait-states, which further slow down external accesses. In practice, since external accesses are much slower than internal accesses, it is recommended to limit such accesses to what is absolutely necessary. Whenever possible, the code should be executed from the on-chip memory. Frequently accessed variables should also be stored in on-chip memory. On the Signal Ranger board, the only external peripheral that is interfaced on the bus system of the TMS320VC5402 is the 64 K word external memory. This memory is mapped in the data space of the DSP (no code can be executed from it), and its accesses require one wait-state.

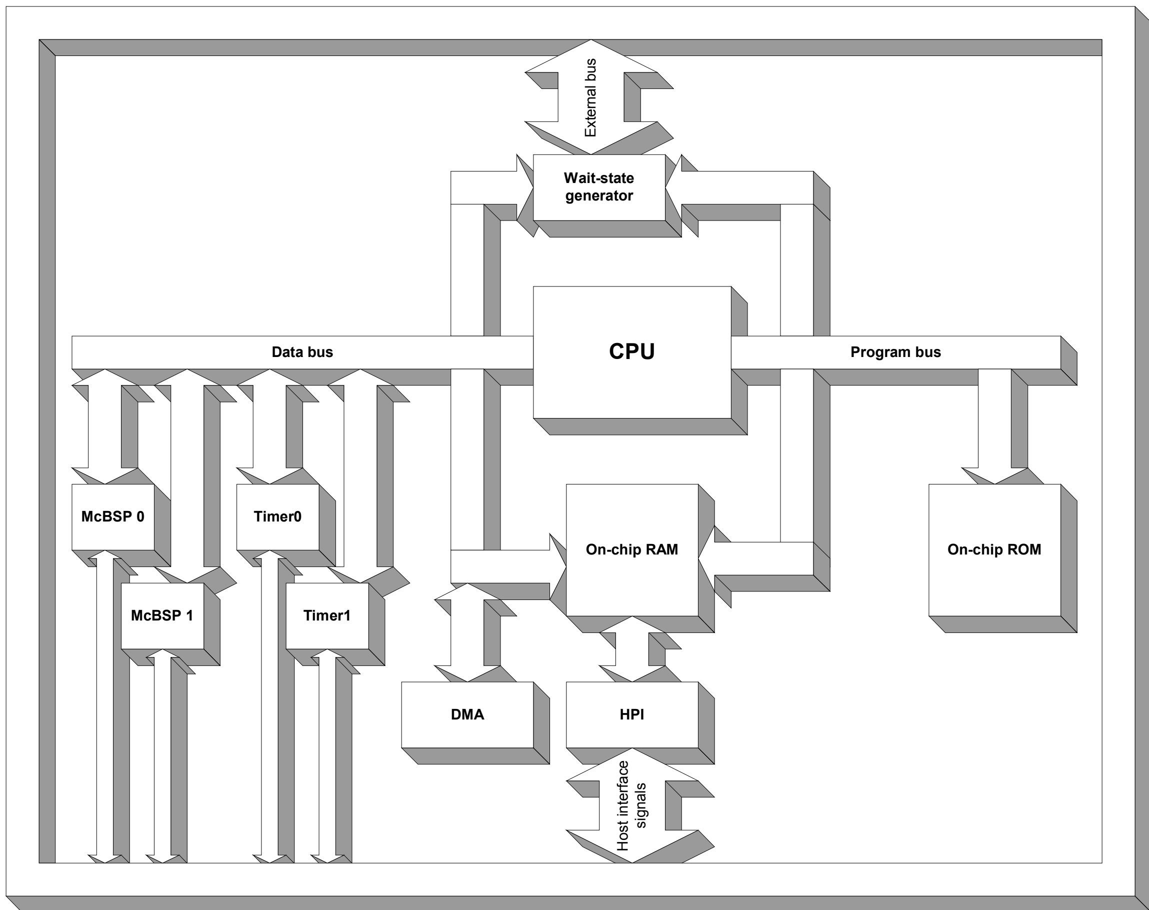


Figure 7-1

Figure 7-1 shows a simplified description of the architecture of the TMS320VC5402.

Note: *Even though its architecture is Harvard, the program bus and the data bus can both access the same on-chip RAM. This RAM is* ***overlaid*** *in the address map of the program space and the address map of the data space. This memory can therefore be used to store instruction codes, as well as data.*

2. CPU

Figure 7-2 shows a simplified description of the architecture of the CPU of the TMS320VC5402.

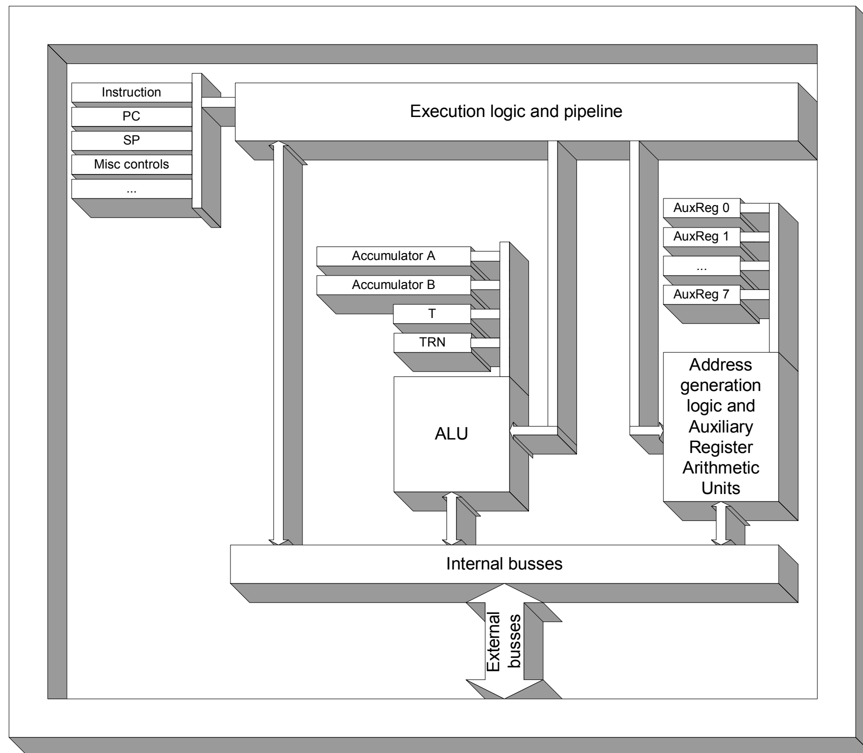


Figure 7-2

2.1. Instruction execution logic

The execution logic includes elements that can read and decode instructions codes and sequence their execution. The execution of a complete instruction takes at least the six following steps. Each step takes 1 clock cycle (10ns) to complete.

- **Pre-fetch**: The content of the program counter (PC register) is placed on the address bus of the program bus system. The PC is incremented afterwards.
- **Fetch**: The instruction at the specified memory address is read by the CPU and placed in the instruction register.
- **Decode**: The instruction is decoded.
- **Access**: If the instruction needs to read operands, the addresses of up to 2 operands are placed on the address busses of up to 2 data bus systems.
- **Read**: If the instruction needs to read operands, these operands are read in parallel on up to 2 data bus systems.
  If the instruction needs to write an operand, its address is placed on the address bus of one of the data bus systems.
- **Execute**: The instruction is executed. If the instruction needs to write an operand, the operand is placed on the data bus in this cycle.

***Note**: When wait-states are introduced during read and write cycles to an external device, these extra cycles are added to the six discussed above, and delay the execution of the instruction by the corresponding number of cycles.*

2.1.1.1.Pipeline

To accelerate the execution of instructions a **pipeline** is used. The pipeline allows the CPU to execute all six execution steps in parallel, on six successive instructions. For instance, while it performs the pre-fetch cycle of one instruction, the CPU also performs the fetch cycle of the previous instruction, the decode cycle of the instruction before it…etc.

On average, the CPU normally executes 6 instructions in parallel, which adds up to one instruction per clock cycle (1 instruction every 10ns). Figure 7-3 shows the progression of instruction cycles in the pipeline.

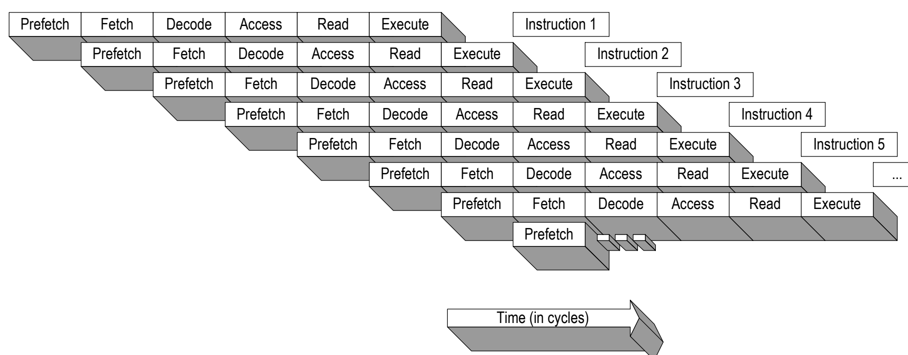


Figure 7-3

The execution time of every instruction that is specified in Texas Instrument’s documentation takes into account the effect of the pipeline. This execution time does not represent the time it takes from the pre-fetch to the end of the execute cycle. Rather, it represents the increment in execution time incurred by adding this instruction to the rest of the code. With this definition, most instructions of the TMS320VC5402 execute in a single clock cycle.

The execution of some instructions may take more than 6 pipeline steps. For instance some instructions are coded on 2 16-bit words, rather than 1. In such a case fetching the extra instruction word takes an extra cycle. As discussed above, the introduction of wait-states during reads and writes to external devices can also add extra cycles to the execution time. To allow these longer steps, without losing pipeline consistency, wait cycles where the CPU does nothing are inserted in the pipeline.

***Note**: Any instruction takes at least as many cycles to execute, as there are 16-bit words in its instruction code. To insure their execution in a single cycle, most instructions can be completely coded (opcode and operands) on a single 16-bit word. For some instructions, or some addressing modes this is not possible, and the complete instruction may be coded on two, or even three separate 16-bit words. Such instructions cannot execute in less than two, resp.*

three execution cycles. It is recommended to use preferentially instructions and addressing modes that can be completely encoded on a single 16-bit word. This is especially true for instructions that are used often, and in time-critical sections of the code.

2.1.2. Pipeline exceptions

Other pipeline conditions can also cause problems. Most instructions are executed in the execute cycle of the pipeline. However, some instructions, depending on the registers they access, or depending on the particular addressing mode that they use, may execute in earlier cycles. This problem can lead to situations where an early-execution instruction does not have the result it expects from the previous instruction, at the time when this result should be used in its execution. Most problems occur when the execution of a particular instruction depends on the contents of a CPU register that should have been updated by the previous instruction. If this instruction executes too early in the pipeline, the register may not have been updated in time by the previous instruction. These situations are called **pipeline exceptions**. Most pipeline exceptions are managed directly by hardware in the CPU. The execution logic simply inserts as many wait cycles as necessary in the pipeline, to keep the sequence of instructions consistent. Exceptions that are managed in hardware do not cause much trouble, other than slowing down the execution by a few cycles.

Some pipeline exceptions however are not managed by hardware. These are called **unprotected pipeline exceptions**. Unprotected exceptions should be managed by software. A simple software solution is to insert NOP (No Operation) instructions in the code between the interfering instructions. Usually one or two NOP instructions are enough to delay the second instruction long enough for it to obtain the expected result from the previous one. Another more efficient solution is to rearrange the sequence of instructions to insert other useful instructions between the interfering instructions. Whatever the solution, the real problem is the detection of unprotected exceptions. Exceptions arise from very complex situations, and are difficult to anticipate. Furthermore, they produce a behaviour that is often difficult to understand and debug. Code Composer Studio's assembler has a feature that can detect unprotected exceptions and generate warnings. These warnings are not enabled by default, but should always be enabled by the developer. Pipeline exception warnings can be enabled in the *Project* menu, under, *Build options – Diagnostics*.

2.1.3. Repetition logic

The execution logic allows the efficient repetition of one or several instructions. The end-of-repetition test is implemented in hardware inside the execution logic, and does not take any execution time. This feature can be used to implement loops that can avoid the end-of-loop test. This leads to very efficient loops where only the useful part of the loop takes execution time. It is especially interesting during loops that repeat many times a very simple operation, because in such a case the end-of-loop test can contribute for a substantial portion of the execution time. In fact for many typical signal-processing operations, this can accelerate the execution time by a factor two or even four.

2.2. CPU registers

The CPU hardware resources include several registers that are used to hold specific data or perform specific operations. These registers are part of the internal logic of the CPU.

© BRUNO PAILLARD ING.

- **A program counter (PC)**: This16-bit register always holds the address of the next instruction in program memory. It is normally incremented at each execution cycle, except in the case of branches or calls. Its contents are placed on the address bus during the pre-fetch cycle. Branches or calls reload the counter with a new address value, which causes the program to branch at the new address.
- **A stack pointer (SP)**: This 16-bit register is used to manage the stack. The stack is an area in memory used to keep critical information during function calls and interrupts.
- **Two accumulators (A et B)**: The accumulators are 40-bit registers. They contain operands and results of arithmetic and logic instructions. They have a large number of bits (32 bits + 8 guard bits), which allows them to hold the results of 16-bitx16-bit multiplications without loss of resolution. The 8 guard bits allow them to accumulate a large number of product terms without overflow. This feature is essential in the implementation of dot products or digital filters, in which sums of a large number of product terms must be performed. These registers are closely associated to the main ALU of the DSP.
- **A temporary register (T):** This 16-bit register is used to hold temporary operands and results of some instructions, such as multiplications. In other words, it takes the role of a smaller additional accumulator.
- **Three repeat management registers (BRC, RSA, REA)**: These 3 16-bit registers hold the start address (RSA), end address (REA), and repeat counter (BRC) that are used to manage the repetitions of groups of instructions (block-repeats).
- **Three control and status registers (ST0, ST1, PMST)**: These 3 16-bit registers hold several bits and bit fields that are used to control the various modes of operations of the CPU, and to indicate the status of its features.
- **Eight auxiliary registers (AR0-AR7)**: These 8 16-bit registers usually hold the addresses of variables accessed by the CPU in the indirect addressing mode. In effect, they are used as pointers to these variables. Two small ALUs called the Auxiliary Register Arithmetic Units (ARAU0, ARAU1) are associated with these registers to perform automatic address calculations (increments, decrements… etc.). In practice, these registers can also be used to perform simple 16-bit arithmetic.
- **An interrupt mask register (IMR)**: This 16-bit register holds the interrupt masks of all the on-chip and off-chip interrupt sources (peripherals).
- **An interrupt flag register (IFR)**: This 16-bit register holds the interrupt flags of all the on-chip and off-chip interrupt sources (peripherals).
- **A transition register (TRN)**: This 16-bit register holds the transition history during the Viterbi decoding. This register is used by very specialized instructions.
- **A circular addressing register (BK)**: This 16-bit register holds the buffer size used in circular addressing.

2.3. Arithmetic and Logic Unit (ALU)

The logic within the CPU that performs arithmetic calculations and Boolean operations is called the Arithmetic and Logic Unit. The ALU often represents a significant portion of the silicon area of the microprocessor. In the TMS320VC5402, the ALU includes a multiplier that can perform a multiplication on 16-bit operands, and produce 32-bit result, in just one cycle. In fact it can also add the product term to a partial accumulation result in the same cycle…!

The outputs of the ALU (the results of arithmetic or logic instructions) are generally placed in one of the accumulators. Exceptionally, some instructions are able to write a result directly in memory. At least one input of the ALU (one of the operands of arithmetic or logic instructions) is usually held in an accumulator.

***Note**: Texas Instruments does not consider the multiplier as being part of the ALU, but rather a separate function of the CPU's logic. However, because it is functionally related to the ALU, we chose to present it as part of the ALU.*

The ALU can perform the following operations:

2.3.1. Additions/subtractions

The ALU can perform signed (2's complement) and unsigned additions and subtractions on 32 bits. The results can be stored on 40 bits. Additions and subtractions are normally performed in one cycle.

The operands can be automatically scaled by an adjustable arithmetic left or right shift, in the same cycle.

2.3.2. Multiplications

The ALU can perform signed or unsigned multiplications in just one cycle. Operands are on 16 bits, and the result is on 32 bits. The multiplier is coupled to an adder, so that multiplication results can be accumulated to a partial result in one of the 40-bit accumulators in the same cycle. This is very useful to implement dot products and digital filters, which are sums of many product terms.

2.3.3. Logical shifts

The ALU can perform logical shifts of 32-bit operands in one cycle. The operands can be shifted by an adjustable amount (from 16 bits to the right, to 15 bits to the left). During the shift, the last outgoing bit is placed in the carry bit (C). Logical shifts to the right never perform a sign extension. Incoming bits are always zero.

2.3.4. Arithmetic shifts

Arithmetic shifts are very similar to logical shifts. There are 2 differences:

- A sign extension is performed during shifts to the right, if the Sign Extension Mode control bit (SXM) is set.
- The result is saturated during shifts to the left if an overflow is detected, and the Overflow Mode control bit (OVM) is set.

© BRUNO PAILLARD ING.

2.3.5. Rotations

Rotations are similar to logical shifts. There are 3 differences:

- The bit pattern is only shifted by one position. Operands cannot be rotated several bit positions at a time.
- The outgoing bit is placed in the carry bit (C), or in the test control bit (TC). This allows tests and conditional branches to be conditioned by the state of the outgoing bit.
- The content of the carry bit (C) or the test control bit (TC) is re-injected into the incoming bit location. In other words, the bit pattern is shifted circularly through C or TC.

2.3.6. Boolean operations

The ALU can perform AND, OR, Exclusive OR, and NOT operations on 32-bit operands in one cycle. When performing a Boolean operation between two operands, each bit of the result is calculated by performing the Boolean operation on the corresponding bits of the operands. In other words, these are bitwise operations. The operands of Boolean operations can be automatically shifted by an adjustable amount in the same cycle. In this case, the shift is always logical (no sign extension is performed on right shifts, no saturation is performed on left shifts).

2.3.7. Special operations

The ALU has special features that accelerate certain types of calculations that are common in signal processing problems. The entire calculations are not performed in one cycle, but the ALU can usually perform each basic step of the calculation (which usually is a complex operation) in a single cycle. The following types of calculations are explicitly supported:

- Calculation of Finite Impulse Response (FIR filters), and dot products. The DSP can calculate and accumulate each product term in a single cycle.
- Filtering using linear phase (symmetrical) filters. The DSP is capable of taking advantage the symmetry of the impulse response, calculating this filter in only half the time it would take to calculate a complete FIR filter.
- Calculation of a Euclidian distance (sum of squared difference terms). The DSP can accumulate each squared difference term in a single cycle.
- Adaptive filtering. The DSP is able to implement each individual term of the LMS (Least Mean Square) equation in a single cycle.
- Viterbi decoding. The DSP can implement each step of the Viterbi decoding in a single cycle.

The ALU has the following additional features:

2.3.8. Automatic sign extension

The ALU always works in 2's complement signed notation. Its ability to perform unsigned operations is obtained by carrying out the operations on wider representations

than the operands, and by avoiding a sign-extension on the operands if necessary. The following details are worth noting.

- Additions and subtractions are always performed on 40 bits. Both accumulators, as well as the inputs of the ALU are really 40-bit wide. However, operands residing in memory are only 16 or 32-bit wide. Operands that are loaded into an input of the ALU, or into an accumulator may, or may not, be sign extended. This sign extension determines if the result is signed or unsigned. To perform unsigned operations, the operands are simply loaded into the accumulators or into the input of the 40-bit ALU without sign extension (their 8, resp. 24 upper bits are set to zero for 32, resp. 16 bits operands). In effect they appear to be positive in 40-bit 2's complement signed notation. To perform a signed operation, the operands are sign extended before being loaded into the accumulators or into the input of the ALU (their 8, resp. 24 upper bits are set to be identical to the MSB of the 16-bit or 32-bit operand). The operation itself is always carried out in signed notation.
  The state of the Sign Extension Mode bit (SXM) controls whether sign extension is performed or not when loading operands into the accumulators or into the inputs of the ALU. This bit is contained in the ST1 register discussed later. It also plays a role in the sign extension that can occur during arithmetic shifts.
- Even though multiplications appear to work on 16-bit operands, they are really performed on 17-bit operands. Because the operands are internally represented on 17 bits rather than 16, unsigned operands can be represented by zeroing bit No 16. Signed operands are represented by sign-extending bit No16. The multiplication itself is always performed in 2's complement signed notation.
  The choice between sign-extending the operands or not is not decided by the state of the SXM bit, as it is for additions and subtractions. Rather, there are specific signed and unsigned multiplication instructions. Signed multiplications perform the sign extension of the operands, while unsigned multiplications place a zero in bit No 16. This process is done before the actual multiplication is performed.

The fact that arithmetic operations are always performed in signed notation in the ALU explains some of its behaviours. For instance it is perfectly legal to subtract two unsigned numbers (if SXM is 0 both operands are loaded in the ALU without sign extension), but still obtain a negative result if the first operand is smaller than the second.

2.3.9.Fractional multiplication

As we saw in chapter 5, the result of a multiplication must be left-shifted by one position when the operands are in Q15 notation, and the result is interpreted as a Q31 number. The multiplier of the TMS320VC5402 has a special mode of operation that can automatically perform the shift. The shift is performed or not, depending on the state of the fractional mode (FRCT) bit contained in the ST1 register discussed below. The bit must be set for Q15xQ15->Q31 multiplications, and must be cleared for integer multiplications, otherwise the results would appear to be twice as large as they should be.

2.3.10.Automatic arithmetic shift of an operand

The logic of the ALU that performs logical and arithmetic shifts (the **Barrel Shifter**) can perform those shifts on the operands of additions, subtractions, logical instructions, and certain memory read and write instructions. In these cases, the shit is performed in the

same cycle as the operation itself. This function is useful to scale numbers before or after the operation. For instructions that support this function, the shift is invoked by specifying the required shift number as a parameter of the instruction. Multiplication instructions do not support the shift. However, in this case, multiplication operands can be shifted when they loaded into the accumulators, and results held in accumulators can be shifted when they are written to memory.

When these automatic shifts are performed during arithmetic operations, the upper bits of the operands may be sign-extended depending on the state of the Sign Extension Mode (SXM) bit. When automatic shifts are performed during logical operations, sign extension is never performed regardless of the state of SXM.

2.3.11.Automatic saturation of results

The ALU offers several ways to saturate a binary result.

- The results of 2's complement signed arithmetic operations (additions, subtractions, multiplications, arithmetic left shifts) can be automatically saturated in case of overflow, instead of producing a rollover. For this, the Overflow Mode (OVM) bit of the ST1 register must be set. When it is set, the 40-bit result residing in one of the two accumulators is saturated to the highest or lowest representation in 32-bit 2's complement signed notation. For instance:
  The number 0900000000_H would be saturated to $007FFFFFFF_H$ (+2 147 483 647d).
  The number $FE00000000_H$ would be saturated to $FF80000000_H$ (-2 147 483 648d).
  Automatic saturations are performed in the same cycle as the operation itself.

***Note**: An overflow during a multiplication can only happen in the following conditions: The two 16-bit operands being multiplied are 8000_H and 8000_H, and the fractional mode (FRCT) is set. In this case, the 40-bit result is saturated to $007FFFFFFF_H$ instead of giving 0080000000_H. It should be noted that this "overflow" does not produce a rollover in the 40-bit accumulator, even if the result is not saturated, since it is represented on 40-bits (and not 32), and the 8 upper bits are zero. A rollover would only happen if the 8 upper bits were discarded, such as would be the case if the lower 32 bits were stored in memory without precautions.*

- Even if the overflow mode (OVM) is not active, it is always possible to force a saturation to occur in the accumulator by using the SAT instruction. This instruction works in exactly the same way as discussed above. It takes one cycle.

- Certain memory write instructions can perform a saturation of the accumulator content before the write. This type of saturation is slightly different than the saturation discussed above. It is still performed on 32 bits (on the 32 lower bits of the accumulator). However, it can be performed in signed or unsigned notation depending on the state of the SXM (Sign Extension Mode) bit. For instance:
  The number: 09000000000_H would be saturated to $00FFFFFFFF_H$, if SXM=0.
  The number: 09000000000_H would be saturated to $007FFFFFFF_H$, if SXM=1.
  To obtain this saturation during the write, the Saturation on Store (SST) bit of the PMST register must be set. This saturation is performed in the same cycle as the memory write.

- Some CPUs in the TMS320C5000 family (not all of them) can perform a saturation during an intermediate multiplication, used in a multiply-accumulate

type of operation (instructions MAC, MAS… etc.). This saturation is performed automatically when the Saturation on Multiply (SMUL) bit of the PMST register is set to 1. To work properly, the OVM bit must also be set to 1. In this case, the behaviour is the same as what would be obtained by performing the multiplication and the addition/subtraction separately, and saturating the result at each step. As discussed above, the result of a multiplication cannot really create a rollover. However, this mode of operation is provided to conform to standard calculation techniques developed for the compression of audio signals.

2.3.12.Rounding

The ALU is capable of automatically rounding values stored in the accumulator. The rounding is performed as follows:

- The 40-bit value contained in one of the accumulators is assumed to be represented in Q23.16 notation (binary bit to the right of bit No 15).
- The rounding is obtained by adding 0.5_d (0000008000_H) to the content of the accumulator, and truncating the 16 lower bits.

Certain instructions can perform this rounding automatically on the result of an arithmetic operation. For instructions that support this, the rounding is performed in the same cycle as the operation itself.

A rounding instruction can be forced by using the RND instruction.

***Note**: The operation of the RND instruction is flawed in silicon revisions of the TMS320VC5402, up to rev. D. The document SPRZ155c “TMS320VC5402 Digital Signal Processor Silicon Advisories” should be consulted for a workaround.*

2.3.13.Dual 16-bit mode of the ALU

In addition to the normal mode of operation of the ALU, which performs a single 40-bit operation in one cycle, the ALU has a special mode that allows it to perform two 16-bit operations in one single cycle (for instance two parallel additions, or two parallel subtractions. This special mode is useful in some situations, such as Viterbi decoding.

2.4. The control and status registers

The CPU includes 3 registers that can control the various modes of operation of the ALU, and the CPU, and indicate the status of various operating conditions.

- **ST0 and ST1**: Contain various bits controlling the modes of operation of the ALU, as well as ALU status bits.
- **PMST**: Contains CPU control and configuration bits.

2.4.1.ST0

15–13	12	11	10	9	8–0
ARP	TC	C	OVA	OVB	DP

- **DP** (Data Page Pointer): These 9 bits constitute the Data Page Pointer, which contains the base address used in DP-based direct addressing mode.

- **OVA, OVB**: These 2 bits indicate that an overflow (in 2's complement signed notation) occurred during the execution of an arithmetic instruction. Most arithmetic instructions affect either OVA, or OVB, depending on the accumulator that is used to hold the result. Either bit is set to 1 during an overflow on the corresponding accumulator, even if the result has been saturated by the OVM mode. Once an overflow is detected in an accumulator, and the corresponding bit is set to 1, it stays set until the bit is used as the condition of a conditional instruction (a conditional branch for instance), or until it is explicitly reset.
- **C** (Carry): This bit is modified by additions and subtractions. It is set to 1 when an addition generates a carry to the left of bit No31, or when a subtraction generates a borrow. C is also modified by shifts, rotations, and other special instructions.
- **TC** (Test Control): This bit holds the result of the ALU's test instructions. It is normally used as a condition in conditional instructions (conditional branches for instance). It can also be set by certain shift instructions.
- **ARP** (Auxiliary Register Pointer): This 3-bit field constitutes a pointer to one of the 8 auxiliary registers. It is used in indirect addressing mode, in the "compatibility" mode (CMPT bit set to 1). This addressing mode is offered to provide a certain level of compatibility with previous generations of DSPs. It is rarely used in recently developed code because of its lack of flexibility.

2.4.2. ST1

15	14	13	12	11	10	9	8	7	6	5	4–0
BRAF	CPL	XF	HM	INTM	0	OVM	SXM	C16	FRCT	CMPT	ASM

- **ASM** (Accumulator Shift Mode): Certain instructions that perform arithmetic operations can apply an arithmetic shift to an operand before the operation. For some of these instructions the shift value is specified directly in the opcode of the instruction. For others, the shift value is specified using the 5 bits of the ASM field. The ASM field can specify shifts from –16 (16 to the right) to +15 (15 to the left).
- **CMPT** (Compatibility): This bit sets the "compatibility" feature of the indirect addressing mode. This feature is offered to provide a certain level of compatibility with previous generations of DSPs. It is rarely used in recently developed code because of its lack of flexibility.
  - CMPT = 0 : Normal indirect addressing mode.
  - CMPT = 1 : Compatibility mode.
- **FRCT** (Fractional): This bit enables the arithmetic left shift that is required when multiplying two Q16 numbers, and the result should be in Q31 notation.
  - FRCT = 0: No left shift is performed on the result of the multiplication (integer multiplication).
  - FRCT = 1: The result is left shifted by one bit position. (fractional Q16xQ16->Q31 multiplication).
- **C16**: This bit allows the ALU to work in the dual 16-bit mode of operation. Normally the ALU works in 40-bit mode. However,

when C16 is set some arithmetic instructions can perform 2 16-bit operations in the same cycle.

- C16 = 0: Normal 40 bits mode of operation.
- C16 = 1: Dual-16-bit mode of operation.

- **SXM** (Sign Extension Mode): This bit enables or disables the Sign Extension Mode. A sign extension may be performed on the operands of arithmetic instructions when they are loaded into accumulators or into the inputs of the ALU. It can also be performed during arithmetic shifts to the left. In both cases, sign-extension is performed when SXM is 1, and upper bits are reset to zero when SXM is 0.
  - SXM = 0: No sign-extension is performed (upper bits are reset).
  - SXM = 1: Sign-extension is performed.

- **OVM** (Overflow Mode): This bit enables or disables the automatic saturation that can be carried out in case of overflow during an arithmetic operation. The saturation can be carried out during most arithmetic instructions, such as additions, subtractions, multiplications and arithmetic left shifts. Overflowing numbers are saturated to the largest positive or negative number that can be represented in 32-bit 2's complement notation: $007FFFFFFF_H$ or $FF80000000_H$. The 8 guard bits are set to the state of the sign bit.
  - OVM = 0: No saturation (rollover on overflow)
  - OVM = 1: Saturation on overflow.

- **INTM** (Interrupt Mask): This bit enables or disables all maskable interrupts. It works in inverse logic.
  - INTM = 0: All maskable interrupts are enabled.
  - INTM = 1: All maskable interrupts are disabled.

- **HM** (Hold Mode): This bit allows a choice between two behaviours of the CPU when its HOLD input activated. An external device normally uses the HOLD input of the DSP to take control of the external bus of the DSP. When HOLD is activated, the DSP puts all its external busses (Address, Data, Control) in high impedance. At this point, the external device can take control of the busses. This other device can be another processor sharing the DSP's busses, a Direct Memory Access controller, or any other type of device that can act as a bus master.
  - HM = 0: During a hold, the CPU places all its busses in high impedance but keeps executing code from its internal memory. Execution only stops if an external access is required.
  - HM = 1: During a HOLD, the CPU places all its busses in high impedance and stops all execution.

- **XF** (External Flag): The TMS320VC5402 provides a general-purpose output that can be used to drive external logic. The XF bit is used to control the state of the output. On the Signal Ranger board, this output is used to drive the reset inputs of the Analog Interface Circuits.

- **CPL** (Compiler Mode): The TMS320VC5402 supports two variants of the direct addressing mode: Direct addressing based on the Data Page Pointer (DP), and direct addressing based on the Stack Pointer (SP). The direct addressing mode based on SP is especially useful to allocate dynamic variables and function arguments on the stack. Such operations are commonly implemented in the code generated by high-level language compilers. Code Composer Studio's C/C++ compiler in particular always uses this variant. The direct addressing mode based on DP is offered to provide some level of

© BRUNO PAILLARD ING.

compatibility with earlier DSPs. It does not have any particular advantages.
- CPL = 0: Direct addressing mode on DP.
- CPL = 1: Direct addressing mode on SP.

- **BRAF** (Block Repeat Active Flag): This bit indicates if a block-repeat is active or not.
  - BRAF = 0: No block-repeat in progress.
  - BRAF = 1: A block repeat is in progress.

***Note**: This bit can be written, however, modifying this bit can interfere with the normal execution of code.*

2.4.3.PMST

15–7	6	5	4	3	2	1	0
IPTR	MP/$\overline{MC}$	OVLY	AVIS	DROM	CLKOFF	SMUL†	SST†

- **SST** (Saturation on Store): This bit enables or disables the automatic saturation that can be performed when writing the contents of an accumulator to memory. This function is only available for certain memory write instructions.
  - SST = 0: No saturation during memory writes.
  - SST = 1: Accumulator contents are saturated during memory writes.

- **SMUL** (Saturation on Multiply): Certain models in the C5000 family of DSPs provide an intermediate saturation function at the end of a multiply, during multiply-accumulate (MAC, MAS…etc.) instructions. If enabled, the intermediate result is saturated in case of overflow. For this function to be fully functional, the OVM bit must also be set.
  - SMUL = 0: Saturation occurs on intermediate multiplications.
  - SMUL = 1: Saturation is disabled on intermediate multiplications.

- **CLKOFF** (Clock Off): This bit enables or disables the clock output of the DSP (ClkOut). The clock output is usually disabled to minimize the RF noise that it generates. Disabling the ClkOut output does not stop the clock, only the external output. The DSP will keep on running.
  - CLKOFF = 0: ClkOut output is active.
  - CLKOFF = 1: ClkOut output is inactive.

- **DROM** (Data ROM): This bit is used to make the DSP's bootloader ROM visible in the DSP's data space. This on-chip ROM contains code that is executed when the DSP is reset. It is mapped from $F000_H$ to $FFFF_H$ in the program memory space of the DSP at reset. When it is mapped in the data space using the DROM bit, all accesses in this address range in the data space are directed to this on-chip ROM and no access cycle is generated on the external bus. External peripherals that may be mapped in the same address range in the data space will not respond. There are no obvious advantages to make this ROM visible in the data space of the DSP, except to access and handle the sine tables that it contains.
  - DROM = 0: On-chip ROM is visible in the data space.
  - DROM = 1: On-chip ROM is not visible in the data space.

- **AVIS** (Address Visibility): When this bit is set, the addresses of all internal bus accesses are presented on the external address bus. This is useful

for tracing on-chip accesses in a debugging phase. However, this function increases the power consumption of the DSP, and also increases the RF noise generated by the DSP.

- AVIS = 0: The state of the external address bus is held at its last value during internal accesses.
- AVIS = 1: The state of the external address bus reflects the state of the internal address bus during on-chip accesses.

- **OVLY** (RAM Overlay): When this bit is set, the on-chip RAM is visible in the program space of the DSP, and the DSP is able to execute code from it. Otherwise, it is only visible in the data space. It is strongly suggested to overlay the RAM in the program space, and execute code from it, since this dual access RAM is much faster than any external RAM could be. This provides the highest performance.
  - OVLY = 0: The on-chip RAM is only visible in the data space only.
  - OVLY = 1: The on-chip RAM is visible in both the data space and the program space.

***Note**: The first 128 words of RAM represent the internal registers of the CPU and on-chip peripherals. This section of the RAM is never mapped into program space regardless of the state of the OVLY bit.*

***Note**: For the Signal Ranger board, the only place where code can be executed from is the internal RAM. The OVLY bit must therefore always be set.*

- **MP/MC** (Microprocessor/Microcontroller): This bit is used to make the DSP's bootloader ROM visible in the program space of the DSP. This on-chip ROM contains code that is executed when the DSP is reset. It is mapped from $F000_H$ to $FFFF_H$ in the program space of the DSP at reset. When it is mapped in the program space using the MP/MC bit, all accesses in this address range in the program space are directed to the on-chip ROM and no access cycle is generated on the external bus. External peripherals that may be mapped in the same address range in the program space will not respond. Normally this ROM is not useful beyond the reset of the DSP, and can be disabled after the reset. When it is disabled, accesses in its address range are directed to the external bus, which allows the DSP to access external peripherals that may be mapped there.
  - MP/MC = 0: The on-chip ROM is visible in the program space.
  - MP/MC = 1: The on-chip ROM is not visible in the program space.
- **IPTR** (Interrupt Pointer): The interrupt vector table is located in the address range $FF80_H$ to $FFFF_H$ in program space by default. However, this table can be dynamically moved after reset to any 128-word zone in the program space. The 9 bits of the IPTR field adjust the 9 upper bits of the beginning address of the vector table. The 7 lower bits of the beginning address are always zero; therefore the interrupt table always starts at an address that is a multiple of 128.

***Note**: The communication kernel of the Signal Ranger board places IPTR at 001_H. The vector table is therefore in the address range 0080_H to $00FF_H$ when the kernel is executing.*

2.5. Visibility of the CPU registers in memory

To facilitate certain operations, all the CPU registers are mapped in memory between addresses 0000_H and $007F_H$ in data space. These registers can therefore be read or modified as if they were memory locations. Some of these registers are more than 16-bit wide. In this case they are decomposed into several 16-bit wide fields, and these fields are mapped at successive addresses. For instance each 40-bit accumulator is decomposed into 3 16-bit words: The lower bits (A_L, B_L) contain the lower 16 bits of the accumulators. The higher bits (A_H, B_H) contain the upper 16 bits of the accumulators, the guard bits (A_G, B_G) contain the 8 guard bits of the accumulators in their 8 lower bits and their 8 upper bits are not used.

2.6. Address generation logic

The data address generation logic of the TMS320VC5402 (DAGEN) is used to generate addresses of operands in the data space. It contains two address generation units. These units, called the Auxiliary Register Arithmetic Units (ARAU0, ARAU1) are really small ALUs, dedicated to performing address calculations on auxiliary registers (additions, subtractions, increments, decrements…etc.). The presence of these Auxiliary Register Arithmetic units allows the CPU to process in sequence all the elements of a data array very efficiently. For instance it can perform an operation on a data element of an array (possibly using its ALU), and in the same cycle increment the address to point to the next data element. The fact that there are two of these ARAUs even allows the CPU to manage two data arrays at the same time. This is important for complex operations such as digital filters for instance, where one data array might contain the impulse response of the digital filter, and the second array would contain the signal samples.

In practice the Auxiliary Register Arithmetic Units can also be used to perform simple arithmetic, unrelated to address calculations. The ARAUs perform 16-bit unsigned arithmetic on the auxiliary registers. To perform operations in this notation, it is often advantageous to use them rather than the ALU, because they do not require using an accumulator (which at any time may be used for other purposes), and because it is often more difficult to perform 16-bit arithmetic with the 40-bit ALU.

3. OPERANDS AND ADDRESSING MODES

Addressing modes represent the ways instructions access their operands in memory. Addressing modes are therefore closely related to the instructions themselves.

3.1. Data types

The TMS320VC5402 can handle data types in two widths:

- It can transfer 16-bit words to/from memory.
- It can transfer 32-bit words to/from memory. However, since the DSP's data busses are only 16-bit wide, 32-bit words are stored at two consecutive addresses. The MSB is stored at the first address, and the LSB is stored at the next one. The first address must be even. The on-chip memory can support two reads per cycle, therefore 32-bit data words can be read in just one cycle when they reside in the on-chip memory. For writes, or when the data words reside in off-chip memory, the MSB is transferred first, and the LSB is transferred next.

Only the first transfer address (the one corresponding to the MSB) is specified as the operand in the instruction code.

***Note**: Most instructions handle 16-bit data, only a small number of instructions directly handle 32-bit data.*

3.2. Addressing modes

The TMS320VC5402 provides seven addressing modes:

- Immediate addressing
- Absolute addressing
- Direct addressing (based on SP/DP)
- Indirect addressing
- Accumulator addressing
- Memory-mapped register addressing
- Stack addressing.

Every addressing mode is not supported by every instruction. Each instruction usually supports a subset of the addressing modes. Instructions that handle multiple operands may support a separate addressing mode for each operand.

At the assembly code level, the developer specifies a particular addressing mode using a unique syntax. At the machine code level, each opcode represents the instruction, as well as the particular addressing mode(s) that it uses. In other words, the same instruction used with different addressing modes is represented by different opcodes.

3.2.1. Immediate addressing

Immediate addressing is used to handle constant data. At the machine code level, this constant is specified as part of the instruction code. It is defined at link time. Since it is part of the machine code, it normally cannot be modified. In many instances, the machine code is stored in ROM, which precludes its modification.

Such an immediate constant can be represented on 3, 4, 8, 9, or 16 bits. When it is defined on 16-bits, the instruction code is represented using two consecutive 16-bit words: the opcode, and its constant operand. In this case, the instruction takes an extra cycle to execute. Otherwise, the opcode and the constant are represented in the same 16-bit word.

At the assembly code level, the developer uses a “#” prefix to specify immediate addressing. This symbol may be omitted if there is no ambiguity, such as for instructions that do not support any other addressing mode.

Ex:

```
LD #0x0080,A            ; This instruction loads the constant 80H
                        ; into accumulator A.
```

3.2.2. Absolute addressing

Absolute addressing is used to handle data that resides at a constant address in memory. There are four types of absolute addressing:

- **Dmad**: (Data Memory Address). Is used to access operands stored in the data space.
- **Pmad**: (Program Memory Address). Is used to access operands stored in the program space.
- **PA**: (Port Addressing). Is used to access operands in the I/O space.
- ***(lk)**: *(lk) symbolizes the content of an address represented by a long (16-bit) constant. This addressing mode is technically a variant of the indirect addressing mode. However, we chose to present it with the other variants of absolute addressing because it is functionally similar.

Whatever the variant, the absolute addressing mode specifies the address of the operand as a 16-bit word that is part of the instruction code. The constant address is defined at link time. Since it is part of the machine code it normally cannot be modified. In many instances, the machine code is stored in ROM, which precludes its modification.

At the assembly code level, the developer does not need a particular syntax to specify the absolute addressing.

Examples:

Dmad

```
MVKD  0x1000,*AR5       ; The content of address 1000H is stored at
                        ; the address pointed to by AR5. In this
                        ; example only the first operand is
                        ; defined in absolute addressing. The
                        ; second is defined in indirect
                        ; addressing, which is studied later.
```

Pmad

```
MVPD  0x1000,*AR5       ; The content of address 1000H is stored at
                        ; the address pointed to by AR5. In this
                        ; example only the first operand is
                        ; defined in absolute addressing. The
                        ; second is in indirect addressing that is
                        ; studied later.
```

PA

```
PORTR 0x1000,*AR5       ; The content of address 1000H is stored at
                        ; the address pointed to by AR5. In this
                        ; example only the first operand is
                        ; defined in absolute addressing. The
                        ; second is indirect addressing that is
                        ; studied later.
```

The choice between each of the 3 variants above is implicit, and closely related to the use of a particular instruction. For instance the MVDK instruction only handles operands in the data space. The MVPD only handles operands in the program space. The instruction PORTR only handles data in the I/O space.

The *(lk) syntax is similar to the syntax used in C to define the "contents of a pointer".

Ex: ***(lk)**

```
LD    *(0x1000),A       ; The content of address 1000H is loaded
                        ; into accumulator A.
```

Absolute addressing is used when the address of the operand is constant and fixed at link time. Variables that have constant addresses are called **static variables**.

Definition:	**Static variable:** A static variable is a variable stored at a fixed address in memory. Its address is defined at link time and is represented by a constant in the machine code.

3.2.3. Direct addressing

Direct addressing allows the CPU to access operands by specifying an offset from a base address that is defined in a CPU register. Two registers may be used for this, which defines to two variants of direct addressing.

- **DP-based direct addressing**: DP (Data Pointer) is a 9-bit field contained in the ST0 register. The address of the operand is obtained by concatenating the 7 lower bits of the instruction code, to the 9 bits of DP, to form a complete 16-bit address. This addressing mode is used in the same situations as the absolute addressing mode. It can be more efficient because only 7 address bits have to be specified in the instruction code. The instruction code (including opcode and operand) can be represented by a single 16-bit word. The corresponding instruction can therefore be executed in one cycle less than an equivalent instruction using a 16-bit absolute operand. However the initialization of DP is necessary and costs execution time. This addressing mode is useful when accessing several variables residing in the same 128-word memory block. In such a case, the initialization of DP can be done only once to be able to access all the variables residing in the same block.

  In practice this addressing mode is difficult to use efficiently. Indeed, since the address is obtained by concatenating, rather than adding, a base address to an offset, the 128-word block that the content of DP defines is always aligned between addresses that are multiples of 128. DP should be re-initialized every time an access is done beyond the boundaries defined by the present DP value. In practice, the exact physical addresses of variables are not defined at the time the assembly code is written, but rather at link time. It is therefore very difficult for the developer to anticipate the accesses crossing block boundaries, and to insert the necessary instructions to re-initialize DP at the proper times. This addressing mode is offered to provide some level of compatibility with earlier DSPs. However, the absolute *(lk) addressing mode is often used instead of DP-based direct addressing, even though it is less efficient. The SP-based variant of direct addressing is used often, but in totally different situations.

- **SP-based direct addressing**: SP (Stack Pointer) is a 16-bit register that is used to manage the stack. The stack and its management are discussed in section 7 of this chapter. The address of the operand is obtained by adding (not concatenating) the 7 lower bits of the instruction code, to the 16 bits of SP, to form a complete 16-bit address. Using this addressing mode, the DSP can access data residing in any of the 128 consecutive addresses beginning at the

© BRUNO PAILLARD ING.

address contained in the stack pointer. Because the 7 lower bits of the instruction code are added, rather than concatenated, to the address contained in SP, the problem of page boundaries that exists in the DP-based direct addressing mode variant is eliminated. The SP-based variant of direct addressing in very useful to access temporary variables and function arguments that have been dynamically allocated on the stack. A C compiler commonly allocates temporary variables on the stack. Such C variables, called **automatic variables,** are declared within C functions or within brackets, and their existence (their "scope") is limited to the portion of code in which they are declared. Among other things, this addressing mode has been introduced to provide a better support of function arguments and automatic variables in C. Its use is not limited to the support of C functions however, and it is advantageous to use it in any case where dynamic variables should be allocated and accessed on the stack.

Definition: **Dynamic variable:** A dynamic variable is a variable that can be allocated at a specific address in memory, during the execution of the code, and can be de-allocated when the variable is not used anymore. After it has been de-allocated, the memory location can be used for other purposes. Dynamic variables are used to store temporary results. They are commonly allocated on the stack, and SP-based direct addressing is used to access them.

The Compiler mode (CPL) bit of the ST1 CPU register determines if DP-based direct addressing or SP-based direct addressing is used. This decision is made at execution time, rather than at assembly time. Observation of opcodes alone cannot determine which variant is being used. The name of this bit (CPL) indicates that the SP-based variant of the direct addressing mode is always used by code generated by the C compiler. However, it is also very common to use this variant in code that is developed directly in assembly language.

Ex: SP-based direct addressing

```
ADD   0x0005,B          ; When this instruction executes, the 7
                        ; lower bits of the first argument (0005H)
                        ; are added to the contents of the stack
                        ; pointer. The content of the resulting
                        ; address is then added to the content of
                        ; accumulator B, and the result of the
                        ; addition is returned to accumulator B.
                        ; The operand that is added to B is
                        ; therefore a variable that resides 5
                        ; addresses higher than the address
                        ; contained in SP.
```

At the assembly code level, no particular syntax is used to specify direct addressing. This is possible because there is never any ambiguity between direct addressing mode and absolute addressing mode (instructions that support the direct addressing mode do not support the absolute one and vice-versa. However, when the SP-based direct addressing mode is being used, the following syntax may be used for clarity:

```
ADD   *SP(5), B
```

***Note**: It is worth noting that the use of the above syntax does not force the instruction to execute in the SP-based variant of the direct addressing mode. This decision is taken at execution time, depending on the state of the CPL bit. In other words, the above instruction will execute in DP-based direct addressing mode (even though it seems to refer to the stack pointer) if the CPL bit is zero at execution time.*

3.2.4. Indirect addressing

The indirect addressing mode is the most versatile, the most efficient, and probably the most widely used of all the addressing modes that the TMS320VC5402 provides. It includes numerous variants. Some of these variants are designed to efficiently support advanced signal processing functions such as filtering and Fast Fourier Transforms.

The indirect addressing mode uses auxiliary registers to hold the addresses of (point to) operands in memory. Since the contents of auxiliary registers can be modified during execution, this addressing mode allows the CPU to access variables whose addresses are variable themselves. One obvious use of the indirect addressing mode is to access dynamic variables that may be allocated at execution time. Compared to SP-based direct addressing mode, the indirect addressing mode does not require these dynamic variables to be allocated on the stack. A C compiler uses the indirect addressing mode to support dynamic allocation on a heap (*malloc*-type functions). More generally, a C compiler uses the indirect addressing mode to support pointer-based accesses to variables.

The auxiliary registers used in indirect addressing mode are connected to two Auxiliary Register Arithmetic Units that can perform address calculations in parallel to (in the same cycle as) operations performed by the ALU. It is possible to increment or decrement a pointer, to point to all the elements of an array in sequence. Such address modifications do not cost anything in terms of execution time. They are performed in parallel to the execution of the instruction. Array processing represents another typical use of the indirect addressing mode.

The instruction code specifies the number of the auxiliary register used to point to the operand, and a possible operation performed on this auxiliary register. Since there are only 8 auxiliary registers, and 15 possible operations, this specification only requires the use of 7 bits of the instruction code (3 bits for the auxiliary register, and 4 bits for the operation). The complete instruction code (opcode + indirect operand) is usually expressed in a single 16-bit word. So it may even be advantageous to use the indirect addressing mode, instead of the absolute addressing mode, to access static variables that are used often. In this case, it takes a few cycles to initialize an auxiliary register to the address of the variable being accessed. Once the initialization is done however, accesses in indirect addressing mode will generally execute one cycle faster than similar instructions in absolute addressing mode, because the absolute addressing mode needs one extra word of instruction code to specify the complete 16-bit address of the operand.

***Note**: Two variants of the indirect addressing mode exist. The first one is obtained when the compatibility mode bit (CMPT) of the ST1 register is 0. We choose to only study this mode here. The second "compatibility" mode is obtained when the CMPT bit is 1. This second mode is much more complex, and less flexible than the first one. It is offered to provide some level of compatibility with earlier DSPs. We do not recommend its use in newly developed code.*

© BRUNO PAILLARD ING.

At the assembly code level, indirect addressing is invoked by using an asterisk, similar to the C “content of pointer” notation. A possible address calculation may be added to the notation.

Ex :

```
MVPD  0x1000,*AR5-      ; In this example, the content of address
                        ; 1000H is added to the memory location
                        ; “pointed to” by AR5. AR5 is then
                        ; decremented. In this example, only the
                        ; second operand (*AR5-) is accessed in
                        ; indirect addressing mode. The first one
                        ; is accessed in absolute addressing mode.
```

Depending on the instruction, 2 separate types of indirect addressing modes exist:

- **Single operand instructions**: These instructions access only one operand in indirect addressing mode. They may however, access another operand using another addressing mode. The MVPD instruction above is an example of single operand indirect addressing.
- **Dual operand instructions**: These instructions access two operands in indirect addressing mode.

3.2.4.1.Register modifications in single operand indirect addressing mode

The following register modifications are allowed:

- ***ARx** No modification, ARx holds the address of the operand.
- ***ARx+** The content of ARx is incremented after the access.
- ***+ARx** The content of ARx is incremented before the access.
- ***ARx-** The content of ARx is decremented after the access.
- ***ARx+0** The content of AR0 is added to ARx after the access.
- ***ARx-0** The content of AR0 is subtracted from ARx after the access.
- ***ARx(lk)** The 16-bit constant lk is added to ARx to produce the address of the operand. ARx is not modified. This addressing mode allows increments and decrements other than 1. It also allows the use of an offset, without modifying the base address contained in ARx. This may be useful to access specific members of a structure. Since lk is coded in the instruction code, it cannot be modified at execution time.
- ***+ARx(lk)** The 16-bit constant lk is added to ARx before the access. This is similar to the *ARx(lk) modification, except that ARx is modified before the access.
- ***ARx+%** The content of ARx is incremented with circular propagation, before the access. The circular propagation allows the address to be rolled back within the boundaries of an array, when the modification brings it outside of the array. This modification allows the efficient implementation of circular buffers. It is used, among other things, to implement digital filters. The array boundaries are specified using the BK register.

- ***ARx-%** The content of Arx is decremented with circular propagation before the access. This is similar to *ARx+%, except that the auxiliary register is decremented.
- ***+ARx(Ik)%** The 16-bit constant Ik is added to ARx with circular propagation, before the access. This is similar to *ARx+%, except that an arbitrary constant is added to the auxiliary register. This modification requires the use of an extra instruction word, to specify the 16-bit arbitrary constant. It costs an extra execution cycle.
- ***ARx+0%** The content of AR0 is added to ARx with circular propagation, after the access. This is similar to *+ARx(Ik)%, except that the value added is held in AR0, and that the modification occurs after the access. Since the value added is held in AR0, rather than specified by a 16-bit constant, the use of this modification does not cost an extra execution cycle.
- ***ARx-0%** The content of AR0 is subtracted from ARx with circular propagation, after the access. This is similar to *ARx+0%, except that the content of AR0 is subtracted rather than added.
- ***ARx+0B** The content of AR0 is added to ARx with bit-reverse propagation, after the access. The bit-reverse propagation is very specific. It provides the special access order that is used in an array to perform the “butterfly” operations found in spectral transforms such as Fast Fourier Transforms and Sine or Cosine transforms.
- ***ARx-0B** The content of AR0 is subtracted to ARx with bit-reverse propagation, after the access. This is similar to *Arx+0B, except that the content of AR0 is subtracted, rather than added.

3.2.4.2.Registers and register modifications in dual operand indirect addressing mode

Instructions that need to access two operands in indirect addressing mode only have 8 bits in the instruction code to specify these two operands. With these 8 bits, they need to specify two separate auxiliary registers, and two separate register modifications. Compared to the 7 bits used to specify one register and one modification in single operand indirect addressing, this represents only 2 bits for the specification of the register and 2 bits for the specification of the modification. Consequently, only a subset of the auxiliary registers, and a subset of register operations can be used in dual operand indirect addressing mode. The available registers are:

- **AR2**
- **AR3**
- **AR4**
- **AR5**.

The allowed register modifications are:

- ***ARx** No operation, ARx contains the address of the operand.
- ***ARx+** The content of ARx is incremented after the access.
- ***ARx-** The content of ARx is decremented after the access.

- ***ARx+0%** The content of AR0 is added to ARx with circular propagation, after the access.

***Note**: Even though it is functionally similar to absolute addressing, the *lk addressing mode is implemented as a special variant of the single operand indirect addressing mode. For this variant, the address of the operand is fetched in the instruction word following the opcode, rather than in an auxiliary register. In practice the significance of this is that all the instructions supporting single operand indirect addressing also support the *lk addressing mode.*

***Note**: The structure of the instruction code (opcode + operand) is very similar for direct addressing mode and single operand indirect addressing mode. In practice this means that most instructions supporting the direct addressing mode also support the single operand indirect addressing mode, and vice-versa.*

3.2.5. Accumulator addressing

Accumulator addressing is functionally very similar to indirect addressing, except that accumulator A is used to contain the address of the operand, rather than an auxiliary register. This addressing mode is only used by a few special instructions, such as the READA, WRITA instructions that perform transfers to/from the program space.

Ex:

```
READA *(AR5)        ; In this example, the content of the address
                    ; pointed to by accumulator A (in the program
                    ; space) is transferred to the memory address
                    ; contained in AR5. In this example, only the
                    ; first operand is implicitly defined in the
                    ; accumulator-addressing mode. The destination
                    ; operand specified by *AR5 is defined in
                    ; indirect addressing.
```

3.2.6. Memory-mapped register addressing

Memory-mapped register addressing is used to access efficiently the CPU and on-chip peripheral registers of the TMS320VC5402. It is functionally similar to absolute addressing, except that the upper 9 bits of the address that is accessed are assumed to be 0s, and are not specified in the instruction code.

Only the first 128 addresses of the CPU's memory map can be accessed using this addressing mode. The TMS320VC5402 has all of the CPU and on-chip peripheral registers mapped in the first 128 words of the memory space. This addressing mode is used exclusively to access these registers.

Since only the 7 lower bits of the address are specified in the instruction code, the complete code, including opcode and operand, can be represented using a single 16-bit word. Instructions that access registers using the memory-mapped register addressing mode usually execute in one cycle less than similar instructions using the absolute addressing.

At the assembly code level, no particular syntax is needed to specify direct addressing. This is possible because there is never any ambiguity between the memory-mapped register addressing mode and the absolute addressing mode (instructions that support the memory-mapped register addressing mode do not support the absolute one and vice-versa).

Ex:

```
LDM   0x0025,A          ; In this example, the content of address
                        ; 0025H is stored into accumulator A. In
                        ; the TMS320VC5402, 0025H is the address of
                        ; the PRD register of one of the on-chip
                        ; timers.
```

3.2.7. Stack addressing

The stack is used to automatically store return addresses during function calls and interrupts. A few instructions use the stack addressing mode to directly push and extract the contents of CPU registers or memory locations onto/from the stack. The operation of the stack and these instructions are described later in this chapter.

***Note**: The stack addressing mode should not be confused with the SP-based direct addressing mode. The former is used to directly push and extract data onto/from the stack. The latter is used to access variables that have been dynamically allocated on the stack.*

3.2.8. Operand notation

Texas Instrument's documentation uses some abbreviations to describe the types of operands and the addressing modes that are used by each instruction.

3.2.8.1. Accumulators

- **src - dst** Represents accumulators A or B. **src** is used when the accumulator is used to hold an operand, or to hold an operand and the result. **dst** is used when the accumulator is only used to hold a result.

3.2.8.2. Absolute operand

- **dmad** Represents a 16-bit operand address in the absolute addressing mode, in the data space.
- **pmad** Represents a 16-bit operand address in the absolute addressing mode, in the program space.
- **PA** Represents a 16-bit operand address in the absolute addressing mode in the I/O space.
- ***(lk)** Represents a 16-bit operand address in the *lk addressing mode.

3.2.8.3. Immediate addressing mode

- **#lk – lk** Represents a 16-bit constant operand in the immediate addressing mode. The "#" sign may be omitted when there is no ambiguity.
- **#K - K** Represents a 2, 5, or 8-bit constant operand in the immediate addressing mode. The "#" sign may be omitted when there is no ambiguity.

© BRUNO PAILLARD ING.

3.2.8.4.Memory-mapped register addressing

- **MMR** Represents an operand (the name of a register) in the memory-mapped register addressing mode.

3.2.8.5.Direct and indirect addressing modes

- **Smem** Represents a 16-bit operand in the direct and single-operand indirect addressing modes.
- **Lmem** Represents a 32-bit (long) operand in the direct and single-operand indirect addressing modes. A long operand is stored as two consecutive words in memory.
- **Sind** Represents an operand in single-operand indirect addressing mode. This notation is used for the few instructions that support only the indirect addressing mode.
- **Xmem - Ymem** Represent operands in dual-operand indirect addressing mode.

3.2.8.6.Miscellaneous

- **SBIT** Describes a 4-bit operand representing a bit number, in immediate addressing mode.
- **N** Describes a 1-bit operand representing the state of a bit in immediate addressing mode.
- **Cond - CC** Describes a 2, 4, or 8-bit operand representing a condition (a branch condition for instance) in immediate addressing mode.
- **SHIFT** Describes a 5-bit operand representing a shift value in immediate addressing mode.
- **SHIFT1** Describes a 4-bit operand representing a shift value in immediate addressing mode.
- **T** Represents the temporary register TREG. TREG is used in some instructions that involve multiplications
- **TRN** Represents the transition register TRN. TRN is used by a few instructions used in the Viterbi decoding.
- **DP** Represents the DP field of the ST0 register. DP is used in the DP-based direct addressing mode.
- **ASM** Represents the 5-bit ASM field of the ST1 register. ASM is used by a few instructions to specify a shift value.
- **TS** Represents the 6 lower bits of the T register. A few instructions use this field of the T register to specify a shift value.
- **ARx** Represents an auxiliary register.

4. BRANCH INSTRUCTIONS

Branch instructions allow the code to branch to a new address. This is done by reloading the Program Counter (PC) with a new address. The next fetch is then performed at the new address, and the program continues executing from there.

There are two types of branch instructions:

- **Unconditional branch instructions**: They perform the branch unconditionally:
  - B: Reloads the PC with an immediate 16-bit address. Since the operand is immediate, the branch address is constant.
  - BACC: Reloads the PC with the 16 lower bits contained in the A or B accumulator. Since the contents of the accumulator can be modified during the execution, this instruction provides a way to branch to a dynamic address (an address that can be chosen at execution time).
- **Conditional branch instructions**: They perform the branch only if a simple or compound condition is met. Otherwise, the execution continues at the next instruction. Conditional branch instructions are used to implement tests, and loops. There are two conditional branch instructions:
  - BC: Reloads the PC with an immediate 16-bit address if a simple or compound condition is met. Possible conditions are described later in this section. They usually represent arithmetic tests performed on the content of an accumulator.
  - BANZ: Reloads the PC if the content of a chosen auxiliary register is not zero. This instruction is especially useful to implement FOR loops.

4.1. Branch conditions

Branch conditions are evaluated by the conditional branch instruction, to determine if the branch should occur or not. They usually represent the result of a simple or compound arithmetic test performed on the content of an accumulator. Table 7-1describes all the branch conditions that are available. Table 7-2 describes the groups and categories that apply to form compound conditions.

These individual conditions may be combined using the following rules:

- Compound conditions must all be chosen from group 1, or from group 2 exclusively. Conditions from both groups cannot be combined.
- When conditions are chosen from group 1, only one condition can be chosen from each category. Up to two conditions can therefore be used in this case. The tested accumulator must be the same for both conditions.
- When conditions are chosen from group 2, only one condition can be chosen from each category. Up to three conditions can therefore be used in this case.

© BRUNO PAILLARD ING.

Condition	Description	Operand
A = 0	Accumulator A equal to 0	AEQ
B = 0	Accumulator B equal to 0	BEQ
A ≠ 0	Accumulator A not equal to 0	ANEQ
B ≠ 0	Accumulator B not equal to 0	BNEQ
A < 0	Accumulator A less than 0	ALT
B < 0	Accumulator B less than 0	BLT
A ≤ 0	Accumulator A less than or equal to 0	ALEQ
B ≤ 0	Accumulator B less than or equal to 0	BLEQ
A > 0	Accumulator A greater than 0	AGT
B > 0	Accumulator B greater than 0	BGT
A ≥ 0	Accumulator A greater than or equal to 0	AGEQ
B ≥ 0	Accumulator B greater than or equal to 0	BGEQ
AOV = 1	Accumulator A overflow detected	AOV
BOV = 1	Accumulator B overflow detected	BOV
AOV = 0	No accumulator A overflow detected	ANOV
BOV = 0	No accumulator B overflow detected	BNOV
C = 1	ALU carry set to 1	C
C = 0	ALU carry cleared to 0	NC
TC = 1	Test/control flag set to 1	TC
TC = 0	Test/control flag cleared to 0	NTC
$\overline{\text{BIO}}$ low	$\overline{\text{BIO}}$ signal is low	BIO
$\overline{\text{BIO}}$ high	$\overline{\text{BIO}}$ signal is high	NBIO
none	Unconditional operation	UNC

Table 7-1

Group 1		Group 2		
Category A	**Category B**	**Category A**	**Category B**	**Category C**
EQ	OV	TC	C	BIO
NEQ	NOV	NTC	NC	NBIO
LT				
LEQ				
GT				
GEQ				

Table 7-2

5. REPEATS

The TMS320VC5402 has two functions that allow it to repeat code in a loop without having to test for the end of the loop at each iteration. When they are used, only the useful part of the loop costs execution time. This is especially advantageous in cases where many iterations are carried out for a loop representing only a few execution cycles. In such a case an end-of-loop test would greatly increase the overall execution time, sometimes doubling or even tripling it.

- **Repeat single instruction (RPT, RPTZ):** Either of these 2 instructions allows the repetition of the instruction immediately following it, from 1 to 65536 times. The operand of RPT or RPTZ represents the number of iterations minus 1. It can be expressed by an immediate 8 or 16-bit constant, or can be the content of a memory location, accessed in direct or indirect addressing mode. The RPTZ instruction resets the A or B accumulator before beginning the loop, while the RPT instructions does not.

***Note**: The number of iterations actually performed is N+1 (where N is the value of the immediate, direct or indirect specified operand). These instructions therefore always perform at least 1 iteration.*

***Note**: It is not possible to nest several loops using the RPT or RPTZ instructions.*

***Note**: The repetition of a single instruction is uninterruptible. The CPU treats the whole repetition as if it was a single instruction.*

***Note**: A few instructions that normally take several cycles to execute, only take one when they are repeated. The list of these instructions is provided in Texas Instrument's document SPRU131f, at table 6.17.*

***Note**: Some instructions cannot be repeated. The list of these instructions is provided in Texas Instrument's document SPRU131f, at table 6.18. The repetition of such instructions is often not useful (an operation performed entirely on an accumulator for instance would lead to the same operation performed over and over if it was repeated). In other cases it does not make sense (repetition of a branch instruction for instance). However, the developer should be*

aware of this because Code Composer Studio's assembler does not give any warning when an attempt is made to repeat an instruction that is not repeatable.

- **Repeat block instruction (RPTB):** This instruction allows the repetition of the group of instructions that follows it, from 1 to 65536 times. The operand of RPTB specifies the address of the last word of the last instruction of the group (the last instruction if it is coded on a single 16-bit word). The first instruction of the group is always the instruction following the RPTB. The number of iterations is specified by the content of the Block Repeat Counter (BRC) register plus 1. This register must be initialized prior to the execution of RPTB. The execution of RPTB automatically loads the Repeat Start Address (RSA), and Repeat End Address (REA) registers. It then sets the Block Repeat Active Flag (BRAF) of the ST1 register, to initiate the repetition.

***Note**: The number of iterations actually performed is always N+1 (where N is the content of the BRC register). This instruction therefore always performs at least 1 iteration.*

***Note**: Contrary to the repeat single process, the repeat block process can be interrupted. It is advantageous to use it instead of a repeat single, even to repeat only one instruction, in cases where the loop must be interruptible. This may be the case when the repetition of a large number of iterations could delay an interrupt loo long.*

***Note**: Contrary to the repeat single process, all instructions can be repeated using the RPTB instruction.*

***Note**: A RPT or RPTZ loop can be nested within a RPTB loop.*

***Note**: It is possible in principle to nest RPTB loops within one another. To do this, the contents of the BRC, RSA, and REA registers must be properly saved and restored, and the BRAF bit of ST1 must be handled properly at each level. In practice however, these operations are complex, and cost execution time. So much so that it is easier, and just as efficient to simply use a repeat-block for the inner loop, and build normal loops using conditional branch instructions (BANZ for instance) for the outer loops.*

6. OTHER CONDITIONAL INSTRUCTIONS

The instruction set of the TMS320VC5402 includes several conditional instructions, in addition to the conditional branches. Their execution can always be replaced by the execution of a similar unconditional instruction, coupled to a conditional branch. However, these conditional instructions often provide a more efficient way of performing the operation.

Their are conditional instruction to perform:

- Conditional branches.
- Conditional function calls.
- Conditional function returns.
- Conditional memory writes.

In addition to these conditional instructions, there is a "Conditional Execution" (XC) instruction that can be used to conditionally execute any instruction following it. Two variants exist, to allow the conditional execution to apply to the execution of instructions

coded on 1, or 2 16-bit words. In certain situations this conditional execution instruction may produce a slightly more efficient code than what would be obtained using a conditional branch instruction.

***Note**: The conditional instructions can use all the single and compound conditions described in tables 7-1 and 7-2, except the conditional memory writes that can only use the subset composed of arithmetic conditions on accumulators.*

7. THE STACK

7.1. What is a stack

The stack is a space in the system RAM that is used to store temporary data used by the program. In particular, it is used to automatically store return addresses during function calls and interrupts. Its operation is therefore closely related to these operations. The stack can also be used to allocate, use, and de-allocate dynamic variables that may be needed by the application.

The stack is physically located in the system RAM, outside the CPU. However, it is managed by a special CPU register called the Stack Pointer (SP).

7.2. How a stack works

The stack is nothing more than a group of consecutive RAM addresses that are accessed with a **Last-In-First-Out** (**LIFO**) logic. Following this access logic, contents are extracted from the stack in the reverse order than when they were stored. For instance if three variables V1, V2, and V3 are stored on the stack in this order, they will be retrieved in the order V3, V2, and V1. The stack pointer automatically implements this LIFO logic.

There are instructions to store register or memory contents on the stack (this is called **pushing** data on the stack), as well as retrieve data from the stack (this is called **extracting, pulling,** or **popping** data from the stack).

These instructions are:

- **PSHD, PSHM**: Push the content of a memory location, resp. a CPU register on the stack.
- **POPD, POPM**: Extract a word from the stack and store it in a memory location, resp. a CPU register.

At any time, the stack pointer contains the address of the last word occupied on the stack. The stack fills-up from high to low addresses. This means that every time a word is pushed on the stack, the stack pointer is decremented. Every time a word is extracted, the stack pointer is incremented.

Definition:	**Top of the stack:** At any time, the address pointed to by the stack pointer is called the top of the stack. This represents the address of the last word allocated or pushed on the stack.

© BRUNO PAILLARD ING.

Figure 7-4 describes a push operation using the instruction PSHM AR1. The stack pointer is assumed to contain $7FFF_H$ before the push. The instruction executes in two steps:

- The stack pointer (SP) is decremented from $7FFF_H$ to $7FFE_H$. This is equivalent to allocating a new memory location on the stack. In effect $7FFE_H$ becomes the address of the last word occupied on the stack.
- The content of the auxiliary register AR1 is written at the address $7FFE_H$ in memory, which represents the top of the stack.

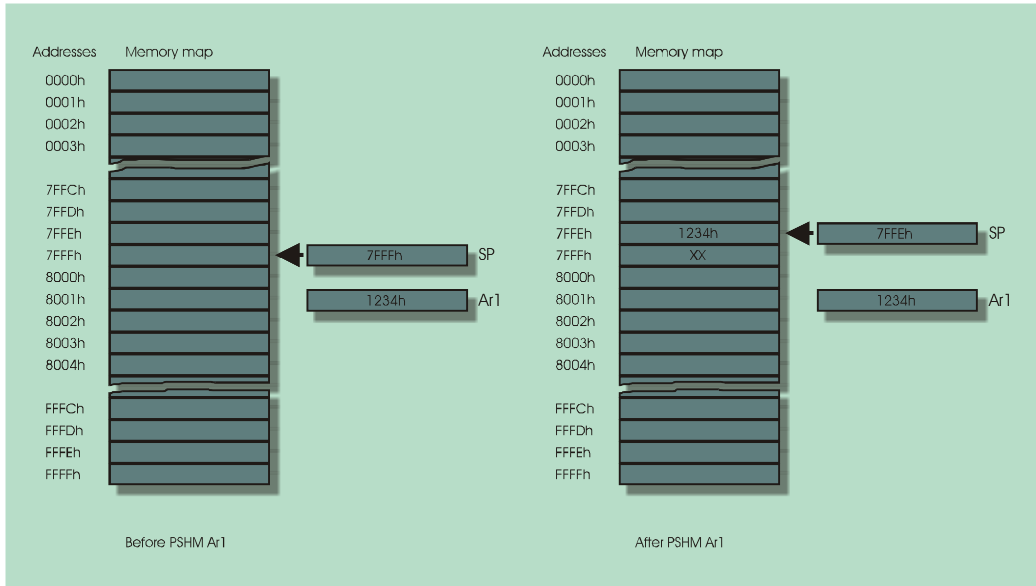


Figure 7-4

Figure 7-5 describes an extraction operation using the instruction POPM AR2. This instruction extracts a word from the stack and stores it in auxiliary register AR2. The stack pointer is assumed to contain $7FFE_H$ before the extraction. The instruction executes in two steps:

- The content of the address pointed to by SP (the last address occupied on the stack) is read and stored into AR2.
- SP is incremented from $7FFE_H$ to $7FFF_H$. This is equivalent to de-allocating the corresponding address from the stack.

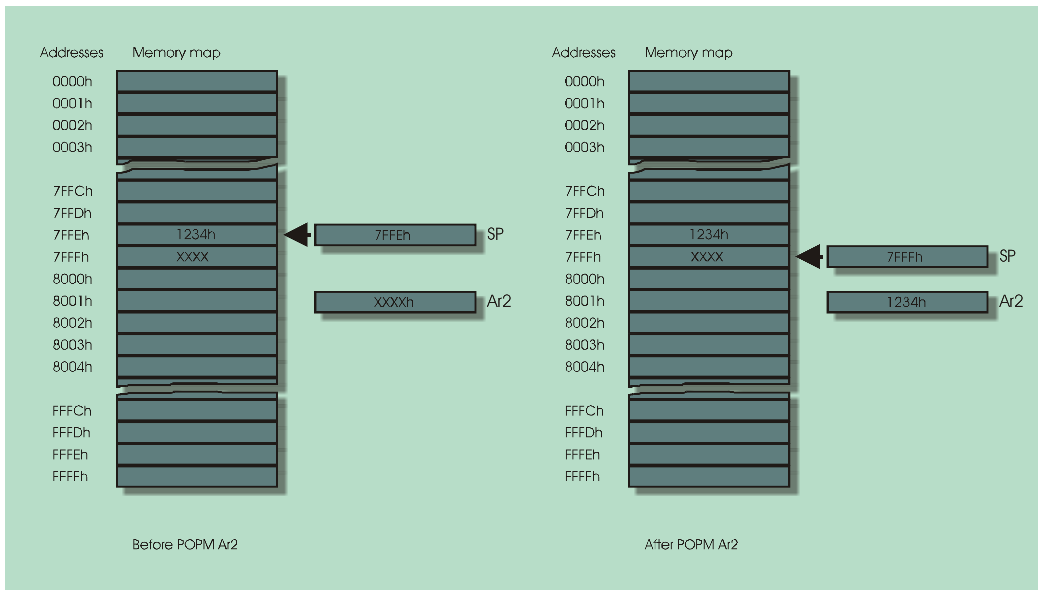


Figure 7-5

***Note**: After the execution of the POPM instruction the word that has been extracted from the stack is not erased from memory. However, this address should be considered to be de-allocated and not containing any useful data. Indeed, interrupts use the stack automatically, and may place the return address of an interrupt into this memory location at a time that is asynchronous and unpredictable from the main code execution.*

It is clear from the operation of the stack that the stack pointer should be initialized at the highest possible address of a RAM zone that is not used for any other purpose. This way, the stack will be able to grow without overwriting any useful data. This stack pointer initialization can be done using the STM instruction that allows an arbitrary value to be loaded into the stack pointer. The initialization must be done before any use of the stack is attempted. Since function calls and interrupts use the stack automatically, the stack pointer must be properly initialized before any function or interrupt is entered. Otherwise, the function or interrupt would write its return address at a random memory location, may be corrupting a useful data or instruction code in the process. Usually the stack pointer is initialized in the first few cycles after the reset of the CPU.

***Note**: Signal Ranger's communication kernel initializes the stack at address $3FFF_H$. This is the highest address of the on-chip RAM. Code and variables are normally loaded into on-chip RAM starting at the lowest possible address. This way the stack has the maximum space to grow.*

As the stack fills-up, data words are stored at lower and lower addresses. There is a possibility that, at some point, the stack data will begin to overwrite memory zones that are used to store other data or code. This situation is called a **stack collision**. As one can imagine, this situation is not too healthy, and usually leads to the code becoming unstable or crashing. There is no system in the CPU to detect and protect against stack collisions. It is entirely the responsibility of the software designer to make sure that the space reserved for the stack is large enough to allow it to grow without interfering with other code or data. This task is usually difficult to accomplish because some processes,

 © BRUNO PAILLARD ING.

such as interrupts, use the stack in ways that are difficult to anticipate. The worst-case scenario, which allows the stack to grow the most, may be difficult to assess. The detection and correction of problems that stem from stack collisions also pose great difficulties. Because of the random element introduced by interrupts, stack collisions do not consistently produce the same behaviour when they occur. The problem may not even be detected at the time of the stack collision. For instance in the case of a corruption of code by the stack, the problem may not even be detected until the corrupted section of code is executed. In any case, the problem of stack collisions is a classic "bug" that should receive the attention it deserves from the software designer.

7.3. Dynamic allocation on the stack

In addition to its automatic use during function calls and interrupts, the stack is often used to allocate dynamic variables that may be required to store temporary results. Such dynamic variables can be de-allocated when the space is no longer required. Dynamic allocation provides a much more efficient use of the available RAM, because once de-allocated the memory locations can be safely used for other purposes.

The TMS320VC5402 includes two functions designed specifically to support dynamic allocation on the stack:

- **The instruction FRAME**: This instruction allows the stack pointer to be directly incremented or decremented. For instance the instruction `FRAME #-10` decrements the stack pointer by 10. The instruction `FRAME #10` increments it by 10.
  Decrementing the stack pointer represents an operation of allocation on the stack. Indeed, even if nothing is written in the corresponding memory locations, these addresses become unavailable to other processes that use the stack (function calls, interrupts…etc.). In other words they are "reserved".
  Incrementing the stack pointer represents an operation of de-allocation of memory addresses on the stack. Indeed, memory addresses that are lower than the present position of the stack pointer become available to other processes on the stack. In other words they are freed.
- **The SP-based direct addressing mode**: This addressing mode allows instructions to access memory words that are in a 128-word zone beginning at the present position of the stack pointer. Addresses that are higher than the content of the stack pointer represent words that have been pushed or allocated on the stack. This addressing mode allows the access of the 128 words that have been pushed or allocated most recently on the stack.

Most high-level languages commonly use the stack to allocate various types of temporary variables. For instance, the C language uses the stack to allocate **automatic variables**. Automatic variables are variables that are declared within C functions or within brackets. Their existence (their *scope*) is limited to the portion of code in which they are declared.

The assembly code generated by a C compiler normally allocates automatic variables on the stack at the beginning of the portion of code that represents their scope, and de-allocates them at the end of this portion of code.

Let's study the following portion of C code for instance:

```
int   input, output;

void test()
{
      int X1;
      int X2;

      X1 = input * 2;
      X2 = input +3;
      output = X1 + 1;
}
```

In this portion of code, *input* and *output* are static variables that are reserved at fixed addresses in RAM.

The following assembly code represents what the C compiler generates from the above C code. The original C statements have been placed in comment fields above the corresponding assembly blocks.

```
_test:
      FRAME     #-2
;----------------------------------------------------------------------
;     X1 = input * 2;
;----------------------------------------------------------------------
      LD        *(_input),A
      STL       A,#1,*SP(0)
;----------------------------------------------------------------------
;     X2 = input +3;
;----------------------------------------------------------------------
      LD        #3,A
      ADD       *(_input),A
      STL       A,*SP(1)
;----------------------------------------------------------------------
;     output = X1 + 1;
;----------------------------------------------------------------------
      LD        *SP(0),A
      ADD       #1,A
      STL       A,*(_output)
      FRAME     #2
      RET
```

- The C statements “int X1”, and “int X2” are translated into the assembly instruction “FRAME #-2”. This instruction allocates 2 words, corresponding to the automatic variables X1 and X2, on the stack. To perform the allocation, the stack is simply decremented by 2. It now points to the address of the newly allocated variable X1. The variable X2 is located at the following address ((SP) + 1).
- The C statement “X1 = input*2” is implemented by loading the content of the static variable *input* into accumulator A (instruction “LD *(input), A”), and then writing the content of accumulator A into the automatic variable X1 with an arithmetic left shift (instruction “STL A, #1, *SP(0)”). The variable X1 is accessed in SP-based direct addressing, with an offset of zero (argument “*SP(0)”). This argument points to the address contained in the stack pointer.

- The C statement “X2 = input + 3” is implemented by loading 3 into accumulator A (instruction “LD #3,A”), then adding the content of the static variable *input* to A (instruction “ADD *(input),A”), and finally writing the content of accumulator A at the address of the automatic variable X2 (instruction “STL A,*SP(1)”). X2 is accessed in SP-based direct addressing mode, with an offset of one. This argument points to the address following the one pointed to by the stack pointer.
- The C statement “output = X1 +1” is implemented by loading the content of X1 into accumulator A, (instruction “LD *SP(0),A”), then adding one to accumulator A (instruction “ADD #1,A”), and finally writing the content of accumulator A to the address of the static variable *output* (instruction “STL A,*(output)”). X1 is once again accessed in SP-based direct addressing using an offset of 0.
- Finally the instruction « FRAME #2 » de-allocates both X1 and X2. The 2 corresponding addresses become available to other processes using the stack.

***Note**: In order for the direct addressing mode to work in its “SP-based” variant, the CPL bit of the ST1 register must be set prior to the execution of this code.*

The C compiler uses the process described above to allocate dynamic variables. The same process can be used to allocate dynamic variables when writing code in assembly language. The advantage of using dynamic allocation to store temporary results is that the locations can be de-allocated when they are not required anymore. After being de-allocated, the addresses are made available to other processes. They can be allocated again to contain other temporary variables. Or they can be used on the stack by function calls, interrupts, or PUSH instructions. Dynamic allocation on the stack should be considered any time a temporary variable is needed in the code.

8. FUNCTIONS

A function is a block of code that performs a particular operation. To execute this block of code, the program branches to the beginning of the function. After the function is completed, the program returns to the location it was called from and continues executing from there. A function usually implements a certain operation on input variables, and provides a result in output variables. Input and output variables are called **arguments** of the function.

Writing functions allows the developer to decompose a complex problem into a number of smaller, simpler ones. Each elementary process is encapsulated into a function. More complex functions can be constructed from simpler ones. In effect, this provides a way to develop the code in a modular fashion.

In practice, most of the code that constitutes an application is part of a function at one level or another. To distinguish the various calling levels, we call **calling code**, or **caller**, the higher-level code that calls a function. We call **function**, the code that is called. The caller may itself be part of a function at a higher calling level.

The same function can be called many times, from various locations in the program. However the actual code of the function resides at a single location in program memory. In addition to provide modularity to the development of code, using functions also provides significant savings in terms of program memory size.

8.1. Call and return

The calling code branches to the entry point of a function using the CALL instruction. This instruction works exactly the same as a branch instruction, such as B, except that the content of the program counter (PC) at the time of the call is automatically pushed on the stack. At the time of the branch the CPU is ready to perform the next fetch. The PC already contains the address of the opcode that follows the CALL instruction. This is the **return address** that is pushed on the stack. At the end of the function, a return instruction (RET) extracts the last word from the stack, and loads it into the PC. This word should normally be the return address that was pushed during the call. Loading the return address into the PC has the effect of forcing the program to branch back to the caller, at the instruction directly following the CALL instruction.

The TMS320VC5402 has three instructions that can be used to call a function:

- CALL: Implements a simple call at a constant address. The argument of the call is an immediate address. The entry point of the function is therefore a constant.
- CALA: Implements a simple call at the address contained in the lower 16 bits of accumulator A or B. This allows the call to occur at an address that can be dynamically modified at execution time. It can be used to choose a particular function from a predetermined list. It can also be used to call a function that may be dynamically relocated in the program memory.
- CC: Implements a conditional call at a constant address. It is similar to CALL, except that a condition can be applied to the call.

These 3 instructions are functionally similar to the 3 branch instructions: B, BACC and BC, except that they also push the return address on the stack.

8.2. Example of a function

The following example describes two code segments: a caller and a called function. For each segment the column on the left represents the addresses where the instruction codes reside in memory.

- The caller initializes the memory locations at addresses 2000_H and 2001_H with the values 1111_H and 2222_H.
- The caller then calls the function *AddInv*, which performs a particular operation on these memory locations, and stores the result of the operation at address 2002_H.
- Finally the caller loads the content of address 2002_H and resumes its processing.

The *AddInv* function performs the following operation:

- It adds the contents of memory locations at addresses 2000_H and 2001_H. In this example, the 40-bit result stored in accumulator A is 0000003333_H.
- It then complements the bit pattern. The result is $FFFFFFCCCC_H$.
- It finally stores the lower 16 bits of the accumulator ($CCCC_H$) into address 2002_H, and returns to the caller.

To understand how the stack is used, let's assume that the stack pointer points to $3FFA_H$ just before the call.

© BRUNO PAILLARD ING.

Caller

```
0302H ST    #0x1111, *(#0x2000)
0305H ST    #0x2222, *(#0x2001)
0308H CALL  AddInv
030AH LD    *(#0x2002),B
```

Function

```
AddInv:

040DH LD    *(#0x2000),A
040FH ADD   *(#0x2001),A
0411H CMPL  A
0412H STL   A, *(#0x2002)
0414H RET
```

Figure 7-6 describes the evolution of the program counter and stack pointer during the function call:

- Just before the execution of CALL AddInv the PC already points to the next instruction at address $030A_H$. The stack pointer points to $3FFA_H$.
- The execution of the call pushes the content of the PC on the stack and branches to the entry point of the function at address $040D_H$. After the call, the PC then contains $040D_H$. The stack pointer has been decremented to $3FF9_H$, and this address (the top of the stack) now contains the return address $030A_H$.
- *AddInv* executes up to the RET instruction.
- Just before the execution of RET, the PC contains the address of what would be the next instruction if there was no return (0415_H). However this instruction will never be fetched.
- The execution of the return extracts the return address from the stack, and loads it into the PC. After the return the PC then contains $030A_H$. The stack pointer is incremented to $3FFA_H$, which de-allocates memory address $3FF9_H$.
- The CPU will then fetch and execute the instruction at address $030A_H$, which is LD *(#0x2002),B in the calling code. This instruction is the one just following the call. The caller has resumed its execution after the return.

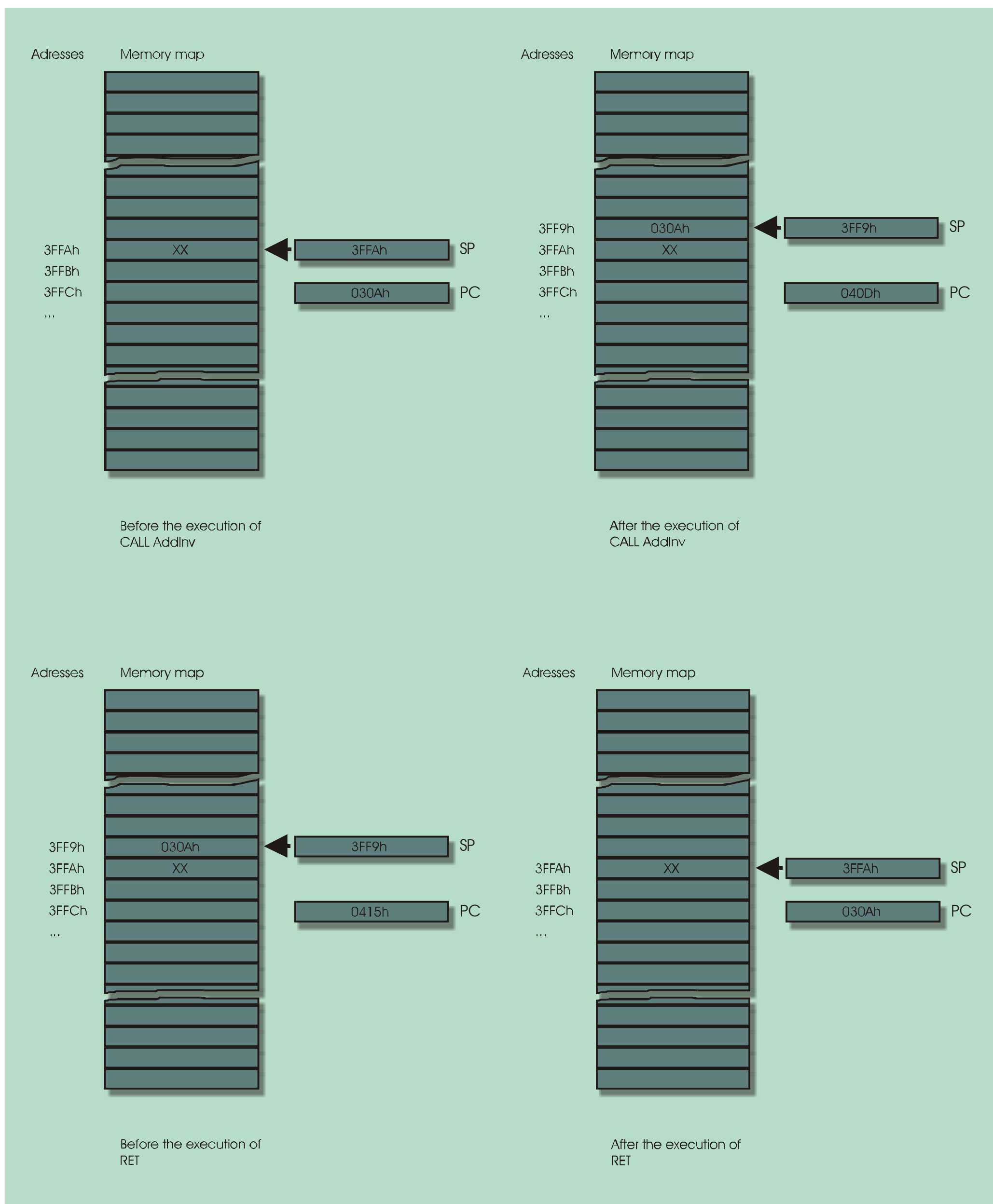


Figure 7-6

8.3. Passing arguments to a function

To work properly together, the caller and the function must exchange input and output arguments. The process of exchanging arguments is called passing arguments. In the above example, all the arguments are passed using static variables. The two input arguments are stored at addresses 2000_H and 2001_H. The output argument is stored at address 2002_H. Both the caller and the function must know the addresses of the arguments.

This argument passing technique poses a few problems. In particular, if the function is modified to use arguments at other addresses, the caller (or callers) must also be

modified. A dependency is therefore created between caller and function. Such a dependency should be avoided because it complicates the modular development of the code. Ideally, we would like to be able to develop the various modules and software layers of an application as independently from each other as possible.

The only case where arguments are passed using static variables is when they are passed to interrupt routines. In this situation, this is the only way to pass arguments, and developers accept the consequences.

Normally, arguments are passed to functions using one of four techniques described below that do not pose the above problem.

These four techniques include two “vehicles”:

- **Passing arguments using registers**
  Input arguments are placed in certain CPU registers by the caller, before the call. Output arguments are placed in certain registers by the function before the return.

- **Passing arguments on the stack**
  The caller allocates input and output arguments on the stack before the call. It then writes the proper values to the newly allocated input arguments. The function reads input arguments and writes output arguments on the stack. After the return, the caller then de-allocates the input and output arguments from the stack

For each of these two vehicles the arguments can be passed in two forms:

- **Passing arguments by value**
  It is the values of the arguments themselves that are exchanged between the caller and the function. The caller must copy the input arguments from their memory locations to the registers or to the stack before the call. The function works on copies of the input arguments and generates copies of the output arguments. After the return, the caller must handle the output arguments or copy them to their proper memory locations. The function itself does not know where the original arguments reside in memory.

- **Passing arguments by address**
  It is the addresses of the arguments in memory that are exchanged between the caller and the function. The caller must write the addresses where the arguments reside in memory, to registers or to the stack before the call. The function then reads the input arguments in memory, processes them and writes the output arguments in memory.

***Note**: Functions usually require several arguments. In practice the developer may very well choose a different technique for passing each argument. A particular technique is often chosen depending on the type and function of each argument. The following examples show the pros and cons of each technique.*

8.3.1. Passing arguments using registers

The following example shows a case where the arguments are passed in registers.

Caller

```
0302H        ST    #0x1111, *(#0x2000)
0305H        ST    #0x2222, *(#0x2001)
0308H        LD    *(#0x2000), A
030AH        STM   #0x2001, AR4
030CH        STM   #0x2002, AR5
030EH        CALL  AddInv
0310H        LD    *(#0x2002),B
             …
```

Function

```
AddInv:
0313H        ADD   *AR4, A
0314H        CMPL  A
0315H        STL   A, *AR5
0316H        RET
```

The code does the following:

- The caller initializes the memory locations at addresses 2000_H and 2001_H with the values 1111_H and 2222_H.
- The caller loads the first input argument (the content of address 2000_H) into accumulator A. This first input argument is therefore **passed by value**.
- The caller loads the address of the second input argument (2001_H) into the auxiliary register AR4. This second argument is therefore **passed by address**.
- The caller loads the address of the output argument (2002_H) into the auxiliary register AR5. This argument is therefore **passed by address**.
- The caller calls *AddInv*.

The function *AddInv* begins execution:

- The content of the memory location pointed to by AR4 is added to accumulator A (accumulator A already contains the first input argument).
- Accumulator A is complemented.
- The result of the operation is written to the memory location pointed to by AR5.
- The function returns to the caller. The caller then uses the result that is found at address 2002_H.

© BRUNO PAILLARD ING.

8.3.2. Passing arguments on the stack

The following example shows a case where the arguments are passed on the stack. The operation performed by the function is the same as before.

Caller

```
0302H        ST    #0x1111, *(#0x2000)
0305H        ST    #0x2222, *(#0x2001)
0308H        FRAME #-3
0309H        LD    *(#0x2000), A
030BH        STL   A, *SP(0)
030CH        ST    #0x2001, *SP(1)
030EH        ST    #0x2002, *SP(2)
0310H        CALL  AddInv
0312H        FRAME #3
0313H        LD    *(#0x2002),B
             …
```

Function

```
AddInv:
0316H        MVDK  *SP(2),AR4
0318H        MVDK  *SP(3),AR5
031AH        MVDK  *SP(1),A
031BH        ADD   *AR4, A
031CH        CMPL  A
031DH        STL   A, *AR5
031EH        RET
```

The code does the following:

- The caller initializes the memory locations at addresses 2000_H and 2001_H with the values 1111_H and 2222_H.
- The caller allocates 3 words on the stack (instruction FRAME #-3).
- The caller writes the content of address 2000_H (the first input argument) on top of the stack. To do this, the caller uses the SP-based direct addressing mode. This argument is **passed by value**.
- The caller writes the address of the second input argument (address 2001_H) on the stack, with an offset of 1 relative to the stack pointer (*SP(1)). This argument is **passed by address**.
- The caller writes the address of the output argument (address 2002_H) on the stack, with an offset of 2 relative to the top of the stack (*SP(2)). This argument is passed by address.
- The caller calls *AddInv*.

The function *AddInv* begins execution:

- The address of the second argument is read from the stack and loaded into auxiliary register AR4. This allows the CPU to later access the argument using the indirect addressing mode based on AR4.

***Note**: The address of this second argument is read using an offset of 2, relative to the content of SP (the top of the stack). At this time, the function has been called, and the return*

address has been pushed on top of the stack. SP now points to the return address, rather than to the first argument, as was the case before the call. The second argument is accessed with an offset of 2, rather than 1.

- The address of the output argument is read from the stack and loaded into auxiliary register AR5. Again, the output argument is accessed with an offset of 3, rather than 2, because of the presence of the return address on the top of the stack.
- The value of the first input argument is read from the stack and loaded into accumulator A.
- The function adds accumulator A to the second argument. The second argument is accessed in indirect addressing mode, based on AR4.
- Accumulator A is complemented.
- The result of the operation is stored at the address of the output argument. The output argument is accessed in indirect addressing mode, based on AR5.
- The function returns to the caller.
- The caller then de-allocates all the arguments that had been allocated on the stack. And processes the result.

Figures 7-7 to 7-16 show the evolution of the stack and the relevant registers during the execution of the code.

***Note**: The code described above uses the direct addressing mode based on SP. For the direct addressing mode to work in its SP-based variant, the CPL bit of the ST1 register must be set. Otherwise the instructions that use direct addressing (STL A, *SP(0) – ST #0x2001, *SP(1), … etc.) will work in the DP-based variant, and obviously will not give the expected result! We did not show the instructions that set the CPL bit. We assume that this was done prior to the execution of the code presented in the example.*

***Note:** At the return of the function, it is very important that the caller de-allocate the arguments from the stack. Otherwise the stack will stay “corrupted” by these 3 values and the other processes that have stored data on the stack will not be able to retrieve it.*

© BRUNO PAILLARD ING.

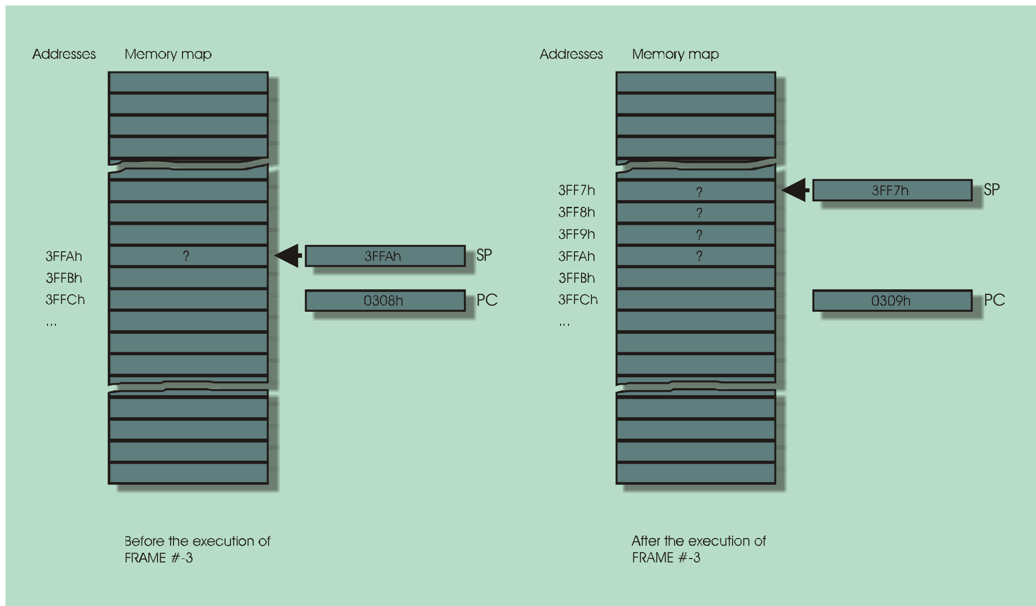


Figure 7-7

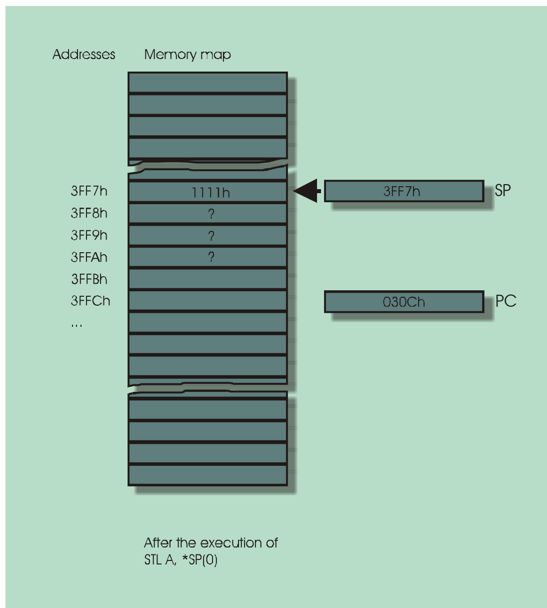


Figure 7-8

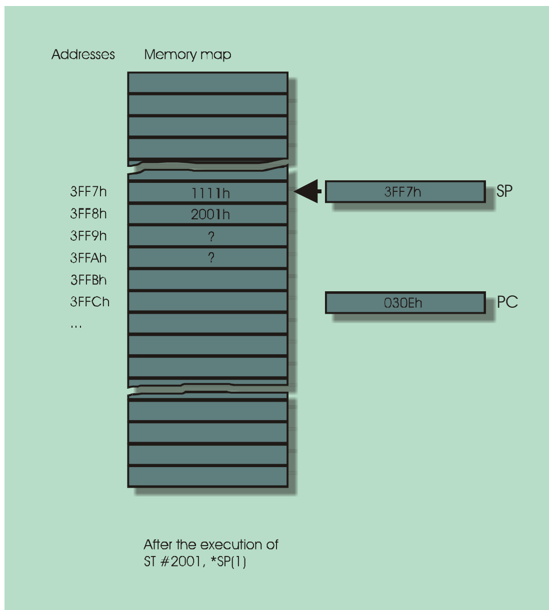


Figure 7-9

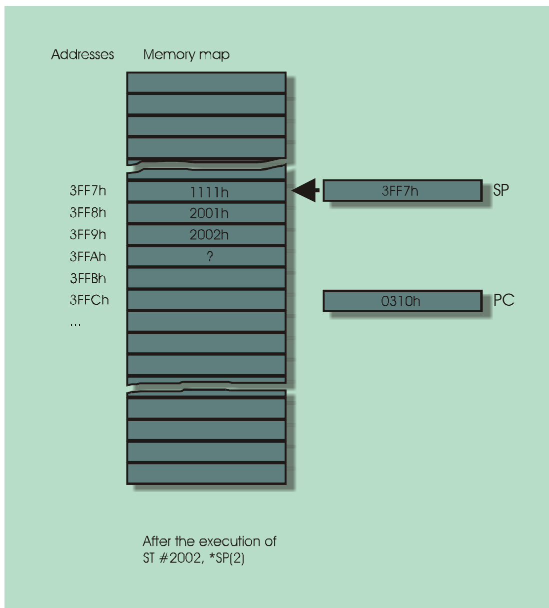


Figure 7-10

© BRUNO PAILLARD ING.

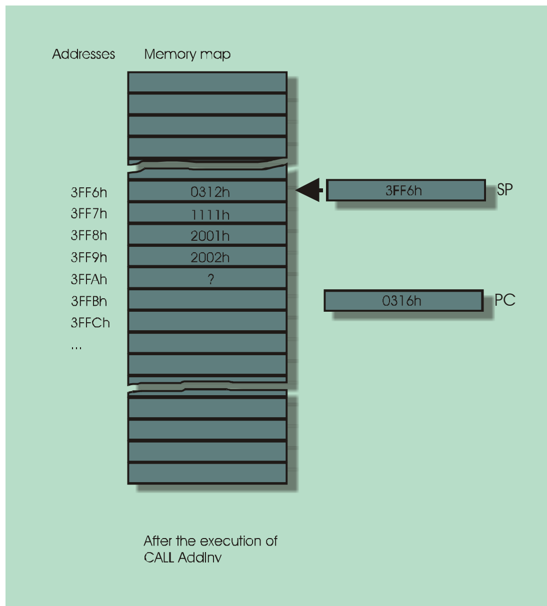


Figure 7-11

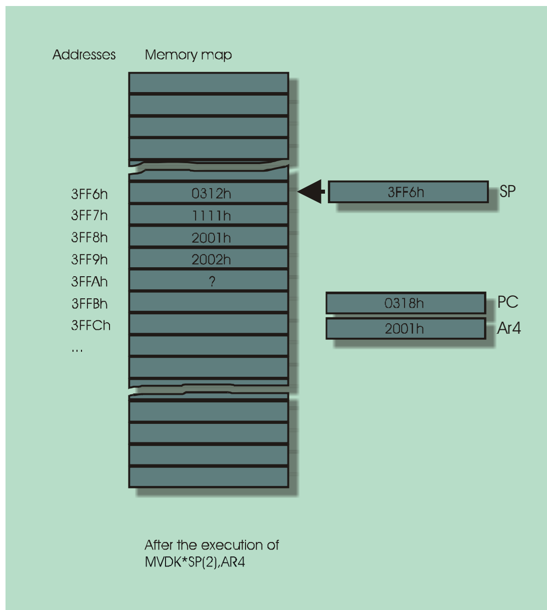


Figure 7-12

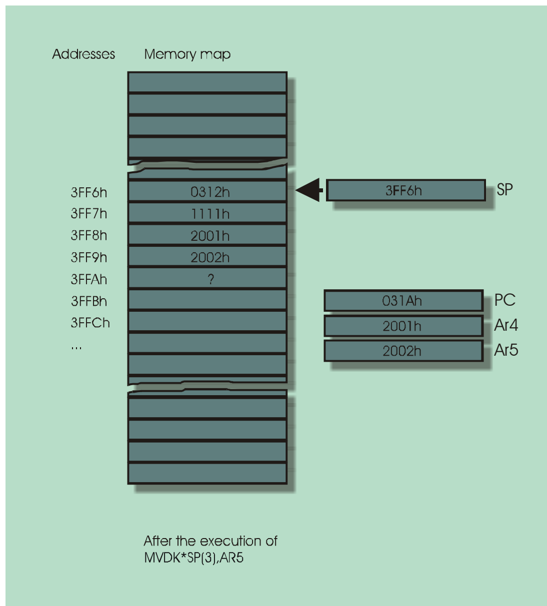


Figure 7-13

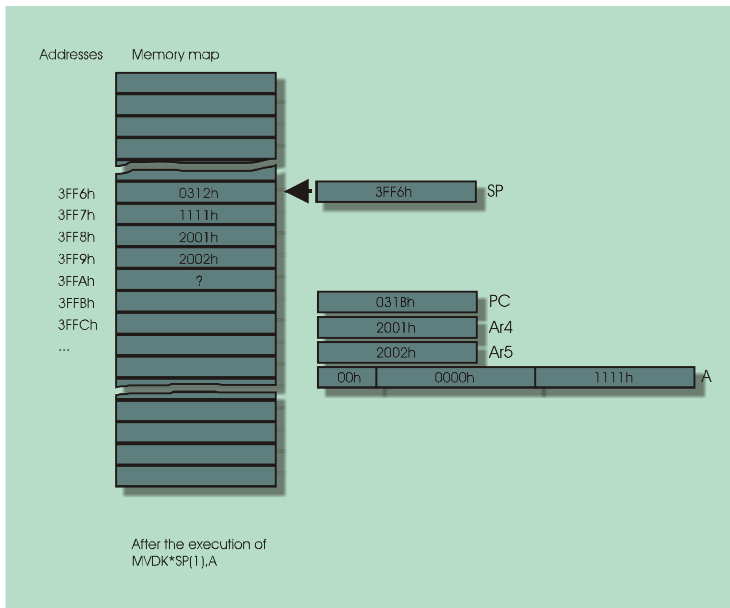


Figure 7-14

© BRUNO PAILLARD ING.

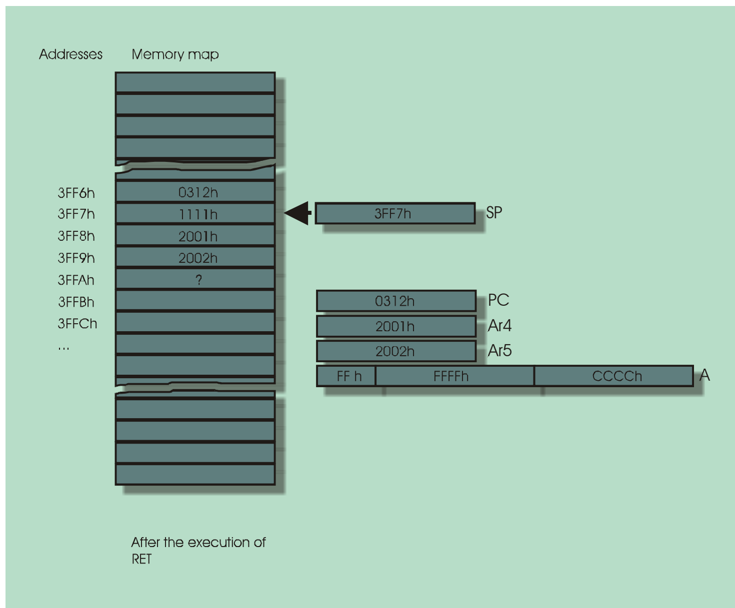


Figure 7-15

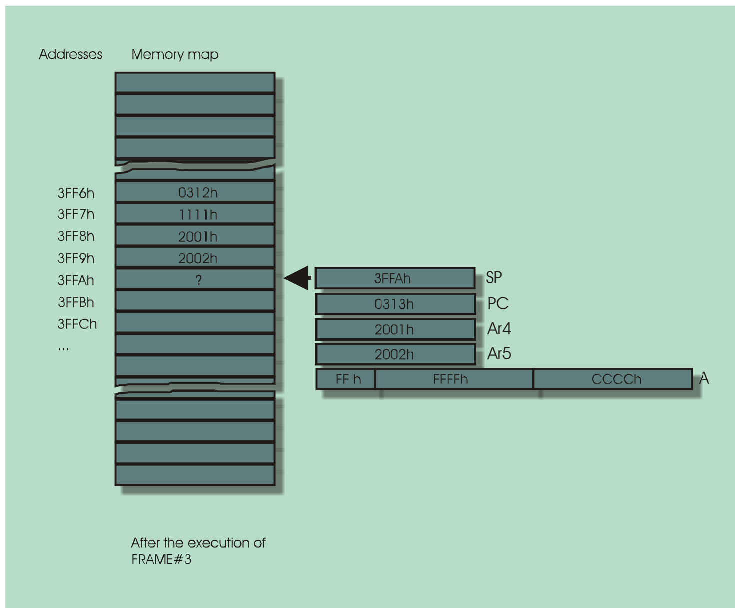


Figure 7-16

8.3.3. Criteria for choosing the argument-passing technique

8.3.3.1. Passing arguments using registers vs. Passing arguments on the stack

Using CPU registers to pass arguments is very efficient. It often requires fewer execution cycles than using the stack. The values of the arguments often have to be in registers to be processed by the CPU. When the arguments are passed on the stack it is necessary to read them from the stack and load them into registers for processing. When they are passed in registers, they are already there.

However the small number of registers available to pass arguments limits this technique to some degree. Don't forget that some CPU registers may already be used for other purposes! When large numbers of arguments are used, they are generally passed on the stack.

When arguments are passed using registers, accumulators are generally used to pass values and auxiliary registers are generally used to pass addresses.

8.3.3.2. Passing arguments by value vs. passing arguments by address

Passing arguments by value is usually more efficient than passing them by address. When arguments are passed by address, these addresses must first be read to be able to access the values. However, passing arguments by value may be less efficient in the following cases:

- When the arguments are large arrays or structures, passing them by value makes no sense because it would require passing every single element of the array or the structure. When passing arguments by address in the case of an array or structure, only the beginning address of the array or structure is passed, which saves execution time and memory size.
- For output arguments, passing by address may be simpler than passing by value, because the function is then able to update the output arguments directly at their location in memory, rather than passing back a value to the caller, which will then have to copy it to its appropriate address.

© BRUNO PAILLARD ING.

8.4. Register protection

Let's take a look at the following code. This code reads N words from a source address in memory, adds 2 to each word, and writes them back to a destination address in memory.

Caller

```
0302H        STM   #3, AR4
0304H        STM   #0x2000, AR5
0306H        STM   #0x2100, AR6

0308H Loop:  FRAME #-2
0309H        MVKD  AR5, *SP(0)
030BH        MVKD  AR6, *SP(1)
030DH        CALL  MoveOne
030FH        FRAME #2
0310H        MAR   *AR5+
0311H        MAR   *AR6+
0312H        BANZ  Loop, *AR4-
             …
```

Function

```
MoveOne:

0315H        MVDK  *SP(1), AR4
0317H        MVDK  *SP(2), AR5
0319H        LD    *AR4, A
031AH        ADD   #2, A
031CH        STL   A, *AR5
031DH        RET
```

Note: *This code is particularly inefficient. Nevertheless, it provides a good illustration of the necessity of protecting registers*

The caller performs the following operations:

- Loads the value 3 in AR4. AR4 is used as a loop counter, and is initialized to 3, to count 4 (3+1) iterations.
- Loads the address 2000_H into AR5. AR5 points to the source address in memory.
- Loads 2100_H into AR6. AR5 points to the destination address in memory.
- Begins executing the loop, which will be iterated 4 times.
- Within the loop, the caller performs the following operations:
  - Allocates 2 words on the stack to pass arguments to the function *MoveOne*.
  - Passes the source and destination addresses to the function using the stack. These two addresses reside in registers AR5 and AR6 originally, and the caller simply copies those contents to the 2 words that have been allocated on the stack.
  - Call the function *MoveOne*. The function reads the word at the source address, adds 2, and writes the result back to the destination address.

- De-allocates the 2 words that have been allocated on the stack.
- Increments the source and destination addresses.
- Tests the loop counter AR4, and iterates if it is not 0. AR4 is decremented after the test. The loop is iterated 4 times.

The function performs the following operations:

- Reads the source address on the stack and loads it into AR4.
- Reads the destination address on the stack and loads it into AR5.
- Loads the source memory content into accumulator A.
- Adds 2 to accumulator A.
- Writes accumulator A to the destination address.
- Returns to the caller.

***Note**: The code described above uses the direct addressing mode based on SP. For the direct addressing mode to work in its SP-based variant, the CPL bit of the ST1 register must be set. Otherwise the instructions that use direct addressing will work in the DP-based variant, and obviously will not give the expected result! We did not show the instructions that set the CPL bit. We assume that this was done prior to the execution of the code presented in the example.*

When testing the above code, one quickly realizes that it is not functional. The reason for that is that AR4 is used by the caller to count loops, and is also used in the function to point to the source address. When the function returns to the caller, AR4 contains the last source address rather than the loop count. In other words, the loop counter has been corrupted by the call.

This is a very usual situation, since callers and functions both use CPU registers. There are only a limited number of CPU registers, and the same register is often used at both levels for different purposes.

To solve this problem, the appropriate register contents are saved on the stack before their use in the function, and restored before they are used again in the caller. This is called **register protection**, **context protection** or **context save**.

For instance, the following *MoveOne* function protects AR4 and AR5 at entry, and restores them just before the return. This way it does not appear to the caller that the content of AR4 and AR5 have been modified by the function. The code is functional.

© BRUNO PAILLARD ING.

Function

```
MoveOne:
0315H      PSHM  AR4
0316H      PSHM  AR5
0317H      NOP
0318H      MVDK  *SP(3), AR4
031AH      MVDK  *SP(4), AR5
031CH      LD    *AR4, A
031DH      ADD   #2, A
031FH      STL   A, *AR5
0320H      POPM  AR5
0321H      POPM  AR4
0322H      RET
```

The PSHM and POPM instructions are frequently used to protect the registers.

***Note**: The NOP instruction between PSHM AR5 and MVDK *SP(3), AR4 is used to solve an unprotected pipeline conflict.*

8.4.1. Register protection conventions

Various conventions may be used to protect the registers. In the above example the responsibility of protecting the registers is placed on the function. The registers are pushed on the stack at the function's entry point. They are restored just before the return.

Three conventions can be used:

- The function can have the responsibility of protecting the registers. This is the case in the example above.
- The caller can have the responsibility of protecting the registers.
- The responsibilities can be mixed, being placed on the caller for some registers, and on the function for others.

The following example describes the code of the above example, which has been modified so that the responsibility of protecting the register is placed on the caller.

Caller

```
0302H          STM   #3, AR4
0304H          STM   #0x2000, AR5
0306H          STM   #0x2100, AR6
0308H  Loop:   PSHM  AR4
0309H          PSHM  AR5
030AH          PSHM  AR6
030BH          FRAME #-2
030C           MVKD  AR5, *SP(0)
030EH          MVKD  AR6, *SP(1)
0310H          CALL  MoveOne
0312H          FRAME #2
0313H          POPM  AR6
0314H          POPM  AR5
0315H          POPM  AR4
0316H          MAR   *AR5+
0317H          MAR   *AR6+
0318H          BANZ  Loop, *AR4-
               …
```

Function

```
MoveOne:
031BH          MVDK  *SP(1), AR4
031DH          MVDK  *SP(2), AR5
031FH          LD    *AR4, A
0320H          ADD   #2, A
0322H          STL   A, *AR5
0323H          RET
```

8.4.1.1.Why is a convention necessary?

A convention is established to allow the code to be developed in a modular fashion. When code is developed at the caller level, the developer should not have to worry about the function being called. When developing code at a function level, the developer should not have to worry about what happens at the caller level. This design methodology greatly simplifies the development of complex applications, because by minimizing the dependencies between caller and function, it minimizes the possibility of bugs arising from these dependencies. It also simplifies the work of the software developer, who can concentrate on small modules at a time, and does not have take complex dependency relationships into account. Furthermore, in complex application designs, the engineer who develops the function level may not develop the caller level. Adopting a convention for the protection of registers will minimize the information that these 2 engineers must share.

8.4.1.2.Responsibility to the function

When the function has the responsibility of protecting registers, it must protect all registers that are used and modified in the function.

© BRUNO PAILLARD ING.

***Note**:* *Registers that are used to pass arguments are not included in the protection convention.*

8.4.1.3.Responsibility to the caller

When the caller has the responsibility of protecting registers, it must protect all registers that are used by the caller, and that should not be corrupted by the function.

8.4.1.4.Mixed convention

The conventions described above often lead to unnecessary protections. For instance in the above example, when the caller has the responsibility of protecting the registers, it protects AR4, AR5, and AR6, even though the function does not use AR6. However, this must be done, because otherwise the developer working on the caller would have to inspect the code of the function every time it is called, which is exactly what the convention is supposed to avoid. In many cases, such inspection is even impossible, because the function may already have been assembled and its source code may not be available to this developer.

A mixed convention is more complex, but it can minimize the unnecessary protections. For example, let’s assume that for auxiliary registers AR0 to AR3, the responsibility of protection goes to the caller and for registers AR4 to AR7 it goes to the function.

- The caller assumes that registers AR0 to AR3 may be corrupted by the function, However, AR4 to AR7 will be protected by the function if necessary. Therefore by using AR4 to AR7 as much as possible for its operations across the call, the caller does not have to protect them.
- The function knows that it does not have to protect AR0 to AR3. Therefore by using these registers as much as possible for its operations, it does not have to protect them.

If the caller can work with AR4 to AR7 exclusively, and the function can work with AR0 to AR3 exclusively, neither level has to protect anything. In practice, some callers may need to use some of AR0 to AR3, and some functions may need some of AR4 to AR7. In these cases, either level will simply protect the registers that are not covered by protection at the other level. This strategy greatly minimizes the required protections, and offers both modularity and efficiency. The C compiler of Code Composer Studio uses such a mixed strategy for the context protection. Details of this strategy can be found in Code Composer Studio’s on-line help, under *Code generation tools – Optimizing C/C++ compiler/Runtime environment/Register conventions.*

9. INTERRUPTS

Interrupts provide a way for peripherals to request the attention of the CPU. Certain peripherals require the CPU to perform management tasks immediately after a hardware event. Interrupts are used to signal the CPU that these peripherals need to be serviced quickly.

Let’s illustrate the concept using the remote control bicycle described in chapter 2. An Analog to Digital Converter (ADC) is used to measure the gyrometer output signal. Every time the ADC completes the conversion of a new voltage sample, it sends an interrupt to the CPU. The CPU stops its current processing, and enters an **Interrupt Service Routine** (ISR) that does the following:

- Reads the digital sample that is stored in an internal register of the ADC.
- Calculates the new position of the handlebars to insure the stability of the bicycle.
- Writes this value to the PWM generator that drives the servomotor connected to the handlebars.

The CPU can then return to the code it was executing prior to the interrupt.

In this case, the CPU has a limited time to respond to the interrupt. If it delays the service too long, the ADC may have completed the conversion of a new sample, and the previous one will be lost.

An Interrupt Service Routine is very similar to a function. In particular, once the ISR has completed its execution, the program returns to the code that it was executing prior to the interrupt. For this, the ISR uses the stack to store the return address, exactly as it does for a function. The main difference however is that the ISR is triggered by an electrical signal send from a peripheral to the CPU, rather than by the execution of a call instruction.

Since an ISR is not called, there is no *calling code* per se. The code that is executed in response to interrupts is sometimes called **background code**, since it represents actions that are performed automatically, and independently of the main code (in the background). The main code, which is periodically interrupted by the background code, is therefore called **foreground code**.

A CPU usually has several interrupt inputs, which can be connected to various peripherals. The TMS320VC5402 has several external interrupt inputs, which can be used to receive interrupts from external peripherals. Several of its on-chip peripherals also have the ability to generate interrupts.

There are 2 types of interrupts:

- **Maskable interrupts**: Maskable interrupts can be disabled (masked). When they are the CPU simply ignores them. Maskable interrupts are generally used to service peripherals.
- **Non-maskable interrupts**: As the name implies, the CPU cannot ignore non-maskable interrupts. They may also be used to service peripherals, but are often used to regain control of a crashed code.

9.1. Maskable interrupts

9.1.1. Triggering mechanism

Figure 7-17 represents the various elements in the microprocessor-system that contribute to the triggering of an interrupt. For a TMS320VC5402, the peripheral may be on-chip or off-chip. In the latter case, only the interrupt input line is represented in the figure. In the case of an on-chip peripheral, the peripheral and the CPU are internally connected, and the interrupt signal is not brought to a pin of the DSP.

© BRUNO PAILLARD ING.

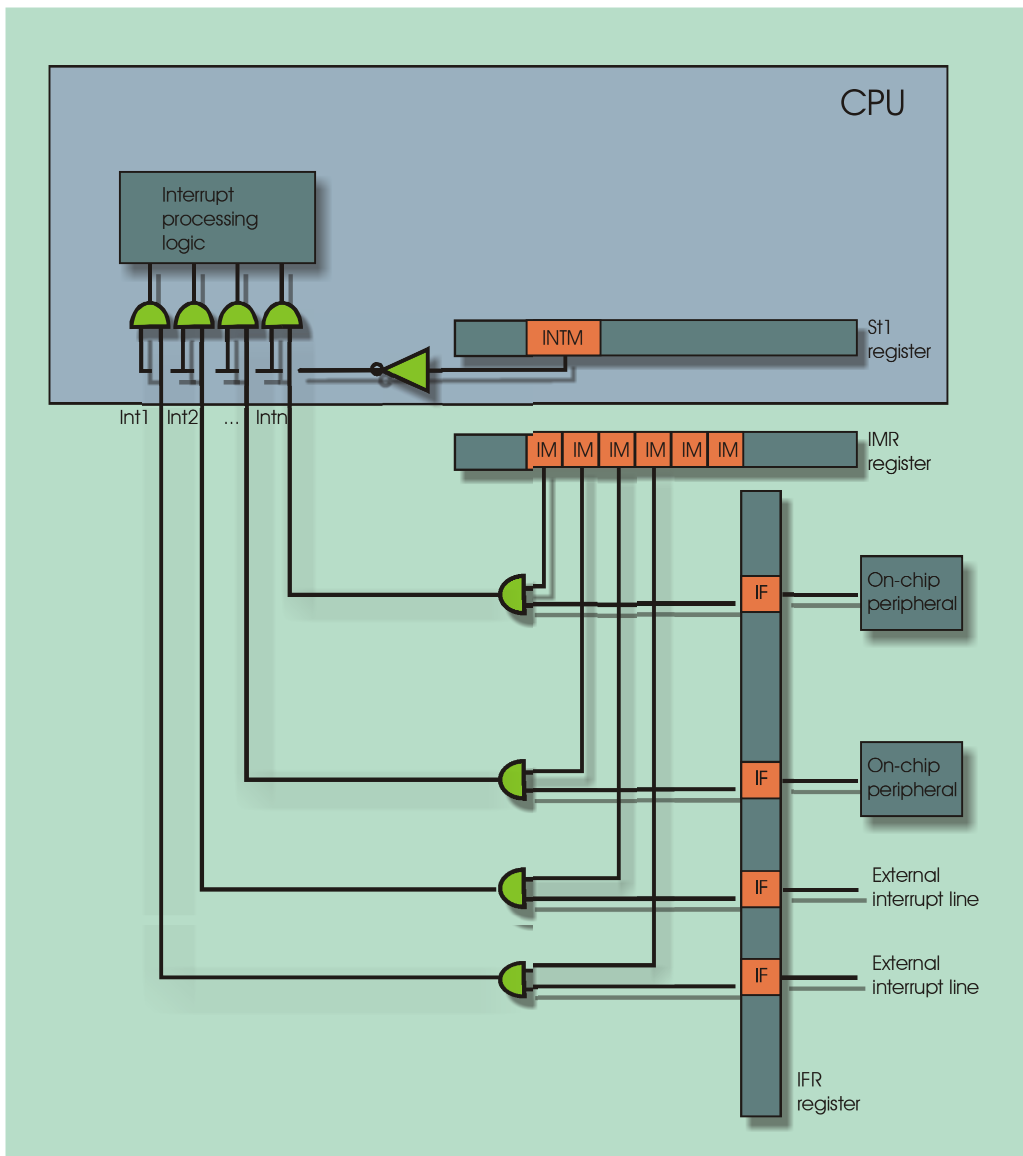


Figure 7-17

The lines called INTn (Interrupt n) in the figure carry the interrupt signals. For most microprocessors, interrupt inputs are active low. For some microprocessors, such as the TMS320VC5402, a falling edge triggers the interrupt. In this case, once triggered, the interrupt signal may stay low without continuously triggering the interrupt. For other microprocessors, the interrupt is triggered by a low state. In this case the interrupt signal must be brought back up to avoid continuously triggering the interrupt.

To avoid false interrupts that could be triggered by noisy signals, the interrupt lines of the TMS320VC5402 must be sampled low for at least 2 cycles, after having been high, for the interrupt to be qualified. This allows the CPU to filter out transient pulses that may be due to signal reflections, "ground bounce" or other hardware problems that would diminish the integrity of these signals. This increases the reliability of the triggering mechanism at the cost of a small 1-cycle delay in the processing of the interrupt.

Even though it is not shown in the figure, the peripherals are also connected to the CPU by one of its bus systems.

Each interrupt line goes through 2 AND gates between the peripheral and the interrupt processing logic. These gates are used to block (mask) the interrupts. The other input of each gate is connected to an interrupt mask. The interrupt signal is allowed to go through when the state of both masks is properly set.

- **The global interrupt mask** (called INTM) allows or masks all interrupts. In effect it allows the CPU to respond to interrupts. The INTM bit is contained in the ST1 register. It is active low (0 enables interrupts, 1 disables interrupts). This bit can be set or reset using the instructions SSBX INTM (to set it), and RSBX INTM (to reset it).

***Note**: Even though the ST1 register can be read or written, this does not apply to the INTM bit. The INTM bit can only be modified using the SSBX and RSBX instructions. In particular it is possible to push and pop ST1 on the stack. However, popping ST1 from the stack has no effect on INTM (INTM stays unchanged).*

- **Local interrupt masks** (called IM in figure 7-17) are associated to each individual interrupt source. Local masks can enable or disable interrupts selectively. They are active high (0 disables interrupts, 1 enables interrupts). All the local masks are contained in the IMR register.

When the CPU qualifies an interrupt signal from a peripheral, a flip-flop called the **interrupt flag** is set to one. The flag being at 1 indicates that an interrupt has been requested and is pending. All the interrupts flags are contained in the IFR register of the CPU. When an interrupt signal is qualified, the corresponding interrupt flag goes to 1 regardless of the states of the local and global masks. If the local mask is set to 1, and the global mask is 0, the CPU responds to the interrupt. Otherwise the interrupt is latched (its flag stays at 1) and will be triggered whenever the masks allow it. Since the CPU can read the IFR register, it can determine if interrupts are pending, even when they are masked. The CPU can also reset any flag to cancel the corresponding interrupt request.

***Note**: To reset an interrupt flag, the IFR register must be written with a 1 at the bit position of the desired flag. Writing a 1 resets the flag, and writing a 0 does not modify it. This unusual logic does not allow an interrupt to be triggered simply by writing to the IFR register. Furthermore, it allows selected flags to be reset, while others are left untouched in a single write operation.*

When an interrupt flag is set to 1 and both masks allow the interrupt to be taken, the CPU does the following operations:

1. It completes the execution of the current instruction. Loops implemented with the RPT (repeat single) instruction are considered as if they were single instructions. In this case, the CPU completes the whole loop.
2. The CPU accepts the interrupt and resets the corresponding interrupt flag.
3. The CPU then pushes the present content of the PC on the stack. This represents the return address of the interrupt service routine.

© BRUNO PAILLARD ING.

4. The CPU then sets the INTM global mask to 1, in effect masking all other maskable interrupts. This process insures that the CPU will not respond to an interrupt from within another one. Maskable interrupts cannot normally be nested within one another.

Note: *The developer may authorize the nesting of maskable interrupts if required. To do this, the INTM bit must simply be reset at the beginning of the interrupt service routine. From this point on, the CPU will respond to other interrupts even though it is presently servicing one.*

5. The CPU branches to a specific address called an **interrupt vector**. Each interrupt source has its own interrupt vector in program memory. Interrupt vectors are located at fixed addresses. Usually a branch instruction to the entry point of the interrupt service routine is placed at the vector's address.
6. The interrupt service routine (ISR) executes. Usually this routine performs management tasks for the peripheral.
7. At the end of the ISR, a special "return and enable" instruction (RETE) extracts the return address from the stack and loads it into the PC. It also restores the INTM bit to its original 0 value. At this point the CPU becomes interruptible again.

Note: *From the moment it takes the interrupt and clears the interrupt flag, the CPU is sensitive to the capture of a new interrupt. If a new interrupt is captured after the flag has been reset, it goes back to 1and the new interrupt becomes pending. The new interrupt service routine is not triggered immediately because the INTM bit is normally at 1 until the end of the present ISR. However, this new interrupt will be taken as soon as the return from the present ISR is executed, and the INTM is restored to 0.*

9.1.2. The TMS320C5402's interrupt sources

The TMS320VC5402 has a total of 17 interrupt sources:

- There are 4 external interrupt pins that trigger an interrupt on a falling edge, followed by at least 2 cycles at a low state. They are called INT0, INT1, INT2, and INT3.
- The on-chip Timer0 can generate interrupts (called TINT0).
- The on-chip Timer1 can generate interrupts (called TINT1)
- The Host Port Interface (HPI) can generate interrupts (called HPINT).
- The serial port 0 (McBSP0) can generate 2 separate interrupts (called BXINT0 and BRINT0).
- The serial port 1 (McBSP1) can generate 2 separate interrupts (called BXINT1 and BRINT1).
- The Direct Memory Access (DMA) system can generate 5 separate interrupts (called DMAC0 to DMAC4).

Note: *3 of these sources share the flags, the masks, and the interrupt vectors. Accordingly there are only 14 flags, 14 masks and 14 vectors for all the maskable interrupts. For these 3 sources, the selection is done according to the state of a configuration bit that usually resides in one of the two competing peripherals.*

Figure 7-18 shows the contents of the IFR register that contains all the interrupt flags of the TMS320VC5402.

15-14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	DMAC5	DMAC4	BXINT1 or DMAC3	BRINT1 or DMAC2	HPINT	INT3	TINT1 or DMAC1	DMAC0	BXINT0	BRINT0	TINT0	INT2	INT1	INT0

Figure 7-18

Figure 7-18 shows the contents of the IMR register that contains all the local interrupt masks of the TMS320VC5402.

15-14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Resvd	DMAC5	DMAC4	BXINT1 or DMAC3	BRINT1 or DMAC2	HPINT	INT3	TINT1 or DMAC1	DMAC0	BXINT0	BRINT0	TINT0	INT2	INT1	INT0

Figure 7-19

9.1.3. Interrupt vectors

The interrupt vectors table of the TMS320VC5402 resides at a fixed address in the program space of the DSP. Each entry in the table consists of 4 consecutive words. When the CPU takes the interrupt, it loads the address of the corresponding vector in the PC, which makes it branch to the first word of the vector. In rare cases, the code of the ISR is short enough that it can reside completely in these 4 words. Most of the time however, a branch instruction is placed there, redirecting the execution to the real entry point of the ISR.

Note: *It is imperative that the interrupt vector be initialized before the corresponding interrupt is triggered. Otherwise, the DSP begins executing the random code that resides at the vector's location as soon as the interrupt is triggered. This usually leads to an immediate and complete crash of the application. In an embedded application, interrupt vectors often reside in ROM, and are initialized when the ROM is programmed. In some cases however, such as in the Signal Ranger board, the interrupt vectors may reside in RAM, and must then be properly initialized before interrupts are triggered.*

Note*:* *In cases where an interrupt may be triggered but no service code should be executed, a simple return and enable (RETE) instruction can be placed at the vector's location. This instruction promptly returns to the code that was executing prior to the interrupt, without doing anything else.*

Table 7.3 lists the addresses of all interrupt vectors (maskable and non-maskable). 2 sets of addresses are indicated: at reset, and after the execution of the communication kernel for the Signal Ranger board. For more information on the latter, refer to the section on reset below.

No Interruption	Priorité	Nom	Vecteur au reset	Vecteur pour Signal Ranger
0	1	**RS**/SINTR	FF80$_H$	0080$_H$
1	2	**NMI**/SINT16	FF84$_H$	0084$_H$
2	2	SINT17	FF88$_H$	0088$_H$
3	2	SINT18	FF8C$_H$	008C$_H$
4	2	SINT19	FF90$_H$	0090$_H$
5	2	SINT20	FF94$_H$	0094$_H$
6	2	SINT21	FF98$_H$	0098$_H$
7	2	SINT22	FF9C$_H$	009C$_H$
8	2	SINT23	FFA0$_H$	00A0$_H$
9	2	SINT24	FFA4$_H$	00A4$_H$
10	2	SINT25	FFA8$_H$	00A8$_H$
11	2	SINT26	FFAC$_H$	00AC$_H$
12	2	SINT27	FFB0$_H$	00B0$_H$
13	2	SINT28	FFB4$_H$	00B4$_H$
14	2	SINT29	FFB8$_H$	00B8$_H$
15	2	SINT30	FFBC$_H$	00BC$_H$
16	3	**INT0**/SINT0	FFC0$_H$	00C0$_H$
17	4	**INT1**/SINT1	FFC4$_H$	00C4$_H$
18	5	**INT2**/SINT2	FFC8$_H$	00C8$_H$
19	6	**TINT0**/SINT3	FFCC$_H$	00CC$_H$
20	7	**BRINT0**/SINT4	FFD0$_H$	00D0$_H$
21	8	**BXINT0**/SINT5	FFD4$_H$	00D4$_H$
22	9	**DMAC0**/SINT6	FFD8$_H$	00D8$_H$
23	10	**TINT1/DMAC1**/SINT7	FFDC$_H$	00DC$_H$
24	11	**INT3**/SINT8	FFE0$_H$	00E0$_H$
25	12	**HPINT**/SINT9	FFE4$_H$	00E4$_H$
26	13	**BRINT1/DMAC2**/SINT10	FFE8$_H$	00E8$_H$
27	14	**BXINT1/DMAC3**/SINT11	FFEC$_H$	00EC$_H$
28	15	**DMAC4**/SINT12	FFF0$_H$	00F0$_H$
29	16	**DMAC5**/SINT13	FFF4$_H$	00F4$_H$

Table 7-3

9.1.4. Register protections for interrupt routines

Interrupt service routines modify registers as functions do. However in the case of an interrupt, protecting the registers can clearly not be the responsibility of the foreground code, since there is no way to anticipate the triggering of the interrupt. Protecting the registers is therefore always the responsibility of the interrupt routine.

Note*:* *An interrupt service routine must protect all the registers that it uses, without exception. In particular this means that if an ISR uses a repeat block (RPTB instruction) it must protect the Block Repeat Counter (BRC), the Repeat Start Address (RSA), the Repeat End Address (REA), and the ST1 register that contains the Block Repeat Active Flag (BRAF bit). When these registers are restored before the return, the BRC register must be restored before the ST1 register. Otherwise the BRC register is 0 when the BRAF bit is restored, since the repeat is presumably finished when the code returns from the interrupt. BRC being 0 automatically clears the BRAF bit. If the foreground code happened to be executing a repeat block, it won't resume its execution upon return.*

9.1.5. Interrupt latency

When a peripheral interrupts the CPU it usually needs to be serviced quickly. In practice, the CPU does not respond immediately to an interrupt. The delay between the setting of the interrupt flag and the execution of the interrupt service routine is called the **interrupt latency**.

The interrupt latency includes the following elements:

- The CPU must complete the execution of all the instructions that have already been fetched at the time of the interrupt (all the instructions that are in the pipeline, except those that are in the pre-fetch and fetch phases). This delay is generally short (a few cycles) and depends on the contents of the pipeline at the time of the interrupt.
- If a repeat-single is in the pipeline (RPT instruction), the CPU must complete all the iterations of the repeat. This delay may be very long if many iterations have to be completed. In cases where the repeat might be too long for the proper management of the peripheral, a repeat-block can be used instead of the repeat-single. A repeat block (instruction RPTB) takes a little bit longer to initialize, but executes just as fast as a repeat-single once it is iterating. Contrary to a repeat-single, a repeat-block is interruptible.
- The CPU will not take an interrupt until both the global and the local mask allow it. If an interrupt is triggered while the CPU is executing a section of code in which the local or global masks disable interrupts, this section will have to be completed before the interrupt routine can be executed. This is especially true of maskable interrupts. When a maskable interrupt is executing, the global mask normally disables interrupts. An interrupt that would be triggered while the CPU is executing another one would have to wait until the CPU returns from the other one, to be executed.

The latency of an interrupt is generally a very critical constraint. For most peripherals using interrupts, waiting too long to respond to an interrupt may cause the peripheral to be mismanaged. In many cases this may cause the crash of the code, or at least cause the application to be temporarily non-functional. There is a limit to the interrupt latency that can be tolerated for the management of any peripheral. The software designer must make sure that latency constraints can be met for every peripheral. This usually means estimating the worst-case latencies for every interrupt. This is not an easy thing to do, since many combinations of events will lead to varying amounts of latency, and all combinations should be taken into account. Also, the latency of certain interrupts might be influenced by the code of other interrupts, which might not yet be written at the time the estimate is needed.

Latency problems are a classic source of bugs. Such bugs are known to be difficult to detect and correct, because there is an element of randomness to their existence. For instance the service of one peripheral may only be missed if its interrupt occurs at the beginning of the execution of another ISR, which will cause a long enough delay. Such a situation might be very rare, and therefore very difficult to detect. A good designer will not wait to see if and when the code crashes, but rather will make a list of tolerable latencies for each peripheral, and compare it against precise estimates of the worst-case situation for each one. The worst-case latency that any one peripheral might face may be difficult to estimate; however this is a very important task.

 © BRUNO PAILLARD ING.

9.2. Non-maskable interrupts

The TMS320VC5402 supports 2 types of non-maskable interrupts:

- **Hardware interrupts** (NMI and Reset).
- **Software interrupts** (instructions INTR and TRAP).

Non-maskable interrupts work in much the same way as maskable interrupts. However, they have neither local nor global mask. The CPU always responds to a non-maskable interrupt.

9.2.1. Hardware interrupts

Hardware non-maskable interrupts are not normally used to manage peripherals. Rather, they are used to gain control of the CPU at power-up, or to regain control of a crashed code. In the case of a crashed code, a maskable interrupt would be of little use, because the random code being executed might have masked it inadvertently. A non-maskable interrupt however will always be able to force the execution to branch to a known section of the code.

The TMS320VC5402 has two hardware interrupts: Reset and NMI. They are triggered by a low pulse on the RS, and NMI DSP inputs respectively.

9.2.1.1. Reset

The reset interrupt is normally triggered by a special reset signal at power-up. The reset signal can also be activated at any time to force the DSP to be re-initialized. The reset signal initializes most of the CPU registers and logic, and triggers the reset interrupt. The reset interrupt branches to the reset vector in the interrupt vector table. The reset vector then branches to the entry point of the initial code.

For the CPU to be properly reset, the reset vector must already contain the proper branch instruction at power-up, and this branch instruction should point to valid initial code. For this the reset vector, and the code it points to, can only be located in ROM. If they were located in RAM their contents would be random at power-up.

Since all vectors are in the same memory zone, they all are located in ROM and cannot be modified. For the TMS320VC5402, the interrupt vectors are located at addresses $FF80_H$ to $FFFF_H$ in the program space. This is a section of the DSP's on-chip ROM. The reset interrupt branches to a special starting code called the **bootloader**, which also resides in the on-chip ROM. This code has been designed by Texas Instruments and allows user code to be loaded and executed in RAM from various hardware interfaces (external memory, serial ports, HPI… etc.). As soon as the user code is executing the bootloader code is not needed anymore. Furthermore, the user code will usually implement interrupts of its own, and cannot use the fixed interrupt vectors that reside in the on-chip ROM. To solve this problem, the interrupt vector table can be relocated at the beginning of any 128-word zone in the program space. This is done by modifying the 9-bit IPTR field of the PMST register. The relocation can be performed at any time during the execution of code. This dynamic relocation capability gives the system enough flexibility to adapt to various memory configurations.

***Note**: On the Signal Ranger board, the bootloader code is used to load a communication kernel in the on-chip RAM. This communication kernel is then used to manage data and control communications between the DSP and the PC. The communication kernel relocates the interrupt vectors at addresses 0080_H to $00FF_H$ in the on-chip RAM of the DSP.*

The user code that is loaded using the communication kernel can use and initialize the interrupt vectors at this location, but must not modify interrupt No25 (HPINT), which is used by the kernel.

9.2.1.2.NMI

The NMI interrupt is generally used to regain control of a crashed code. It is a high-priority interrupt. However, it does not perform all the initializations that the reset does. For this purpose it is a little bit gentler than a reset. However, because it does not reinitialize everything, it may not be able to regain control in all circumstances.

When a Reset or an NMI is used to regain control of a crashed code, the CPU should not return from its ISR. Such an action would branch back to the uncontrolled code and defeat its purpose. Rather, the interrupt is used to branch to a known section of code from which the execution continues. The stack may be re-initialized, along with any relevant variables. In effect there is no ISR.

9.2.2.Software interrupts

Software interrupts behave like hardware interrupts, except that they are triggered by the execution of special instructions, rather than by hardware events. In that sense, they are really similar to function calls.

The TMS320VC5402 supports two types of software interrupts: The TRAP instruction, and the INTR instructions. Both instructions can be used to trigger the execution of any interrupt in table 7-3. The INTR instruction sets the INTM bit, which masks all maskable interrupts, while TRAP does not. Because of this, INTR is a bit closer to a real interrupt than TRAP.

Software interrupts may be used to switch between various tasks in multi-tasking environments.

9.2.3.Availability of the stack

Interrupts, like functions, use the stack to temporarily store their return address. However, unlike functions interrupts are not called, rather they are triggered by hardware events. If interrupts were to be triggered before the stack had been initialized, their return address would be saved at a random memory location, possibly corrupting other data or code.

To avoid this situation, maskable interrupts are always masked when the DSP gets out of reset. The stack pointer should always be initialized before any interrupt is unmasked.

Non-maskable interrupts can obviously not be masked, even at reset. Of all the non-maskable interrupts, the reset does not pose any problem, since it leads to the fresh restarting of the code anyway. Software interrupts can easily be avoided before the stack pointer is initialized. The NMI interrupt however can be triggered by an electrical signal on the NMI input, and could arguably pose a problem if it was triggered between the reset and the initialization of the stack. It is the responsibility of both the hardware and the software designers to make sure that the NMI interrupt, if it is connected to any external logic, cannot be triggered after the release of the reset signal and before the initialization of the stack pointer.

© BRUNO PAILLARD ING.

9.2.4. Nesting of interrupts

Contrary to functions, interrupts are not normally nested within one-another. When the CPU takes an interrupt, it sets the INTM bit, which masks the other interrupts for the duration of the service routine. Other interrupts that are triggered while the CPU is processing an ISR are latched and executed as soon as the CPU returns from the present one. In principle, the software designer can allow the nesting of interrupts. To achieve this, the INTM bit should simply be reset at the beginning of the ISR. This practice is not recommended however. It seldom increases the performance of the code because nesting interrupts does not accelerate their execution. It greatly complicates things, in particular so many more combinations of events must be considered when estimating interrupt latencies.

Non-maskable interrupts do interrupt maskable ones. However, since they are used in exceptional circumstances, this does not generally pose any problems.

9.2.5. Priorities of maskable interrupts

Each maskable interrupt has a specific priority level. The priority level of every interrupt of the TMS320VC5402 is indicated in table 7-3. Lower numbers have higher priority. A higher priority interrupt will not interrupt a lower priority one. However, if both interrupts are triggered simultaneously, or if they are triggered during a time when the CPU is uninterruptible, the higher priority interrupt will be taken first when the CPU becomes interruptible again. The lower priority interrupt will be taken when the CPU returns from the higher priority ISR.

9.2.6. Passing arguments to an interrupt service routine

Data must often be exchanged between interrupt routines and foreground code. However, since the foreground code has no control over the triggering of the interrupts, passing arguments is more difficult for interrupts than it is for functions.

The foreground code cannot use CPU registers to pass arguments to interrupts. Since the triggering time of interrupts cannot be anticipated by the foreground code, registers would have to be allocated permanently for that purpose. Registers are too precious a commodity for them to be permanently allocated to a single task.

For the same reason, the stack cannot be used to pass arguments either.

The only remaining option is to pass arguments using static variables. Even though we said earlier that this solution posed problems, it is accepted here because there are no other options.

9.2.7. Corruption of static variables by an interrupt routine

If an interrupt routine exchanges data with the foreground code using static variables, both modules may read and write the variables at any time. Since the execution of the interrupt routine is not synchronized to the foreground code, it can be triggered at the exact time when the foreground code is handling the variables. When this happens, the interrupt routine may modify the contents of the variables while they are being used by the foreground code, in effect intruding on, and possibly corrupting, the foreground process. The foreground code has several ways to protect itself from such intrusions when they can cause problems:

- The foreground code can set the global mask (INTM) for the duration of the section that handles the shared static variables. Such a section of code is called

atomic because it executes as a whole and cannot be interrupted. Atomic sections should remain short, because they increase the latency of all interrupts. They should be limited to the essential section of code that should not be interrupted. This solution also lacks generality because the interrupts are enabled at the end of the atomic section, whether or not they were enabled initially.

- A better solution would be to push the IMR register, which contains all the local flags, on the stack. Then clear the IMR register, which would mask all interrupts. Then execute the atomic section of code, and finally restore the IMR register from the stack. This solution has more generality. It restores the state of the interrupt logic exactly as it was before the atomic section.
- A still better solution in this case would be to push IMR on the stack. Then only mask the single interrupt that may cause the intrusion. Then execute the code that handles the shared variables, and finally restore IMR from the stack. This solution has the advantage of not blocking the other interrupts during the critical section of foreground code.

***Note**: One solution that may come to mind to perform the operation described at the second point would be to push ST1 (which contains the global mask bit INTM) on the stack, then set INTM to mask all interrupts, then execute the atomic section of code, and finally restore ST1 from the stack. This is not possible however because the INTM bit can only be modified using the SSBX and RSBX instructions. Restoring ST1 from the stack would restore all its bits except INTM. The solution described at the second point does exactly the same thing by operating on IMR instead.*

On-Chip Peripherals of the TMS320C5402

- 1 Introduction
- 2 Timers
- 3. Host Port Interface (HPI)
- 4. Multi-channel Buffered Serial Ports (McBSPs)
- 5. Direct Memory Access (DMA)
- 6. BIO and XF signals
- 7. The clock generator
- 8. The Wait-State generator

1. INTRODUCTION

This chapter provides a brief description of the peripherals that are found on the TMS320VC5402. For a more detailed description, the reader should refer to the document SPRU 131 *TMS320C54x DSP reference set, volume 1: CPU and Peripherals* from Texas Instruments.

The TMS320VC5402 includes 7 types of peripherals:

- Timers 0 and 1
- Host Port Interface (HPI)
- Multi-channel Buffered Serial Ports (McBSPs) 0 and 1
- Direct Memory Access (DMA)
- Generic I/O signals (BIO and XF)
- Clock generator
- Wait-state generator

All the registers controlling these peripherals are accessible in page 0 of the data space. More specifically, they reside between addresses 0020_H and 0058_H. They can be accessed efficiently using the memory-mapped data addressing mode.

In reality, only the first 4 subsystems are peripherals in the usual sense: “Devices that are interfaced to the CPU through its bus system, and which provide services to the CPU”. The generic I/O signals, the clock generator, and the wait-state generator are partly integrated into the core logic of the CPU, and go beyond the traditional definition of a peripheral.

2. TIMERS

The TMS320VC5402 has 2 timers : Timer 0 and Timer 1. These are very simple devices that provide the following functions:

- Generation of periodic signals of adjustable frequency. This function is useful to generate appropriate clock signals for all kinds of peripherals, such as Analog to Digital and Digital to Analog Converters.

Note: *On the Signal Ranger board, the output of Timer 0 is used to generate the master clock signal for the 8 Analog Interface Circuits (AICs).*

- Triggering of software events at periodic intervals. Each timer can generate interrupts at regular intervals. This can be used to trigger specific processes. One example of this function is the switching of tasks in preemptive multi-tasking kernels.
- Precise measurement of the duration of software processes. For instance the timers can be used to precisely measure the duration of functions and ISRs.

3. HOST PORT INTERFACE (HPI)

The Host Port Interface is a device that allows the DSP to be interfaced to another (host) CPU. This is an essential element in many modern architectures, where the DSP is a peripheral of a general-purpose microprocessor, and provides it with specialized services, such as high-speed calculations and signal processing functions.

The HPI gives the host CPU access to all of the on-chip RAM of the DSP. Through the HPI, the host is also able to send an interrupt the DSP, and receive an interrupt signal from the DSP. Through the HPI, the host can control the DSP and communicate with it. For instance, at reset the host can download executable code into the on-chip RAM of the DSP and launch its execution. While the code is being executed by the DSP, the host can read and write locations in the on-chip RAM, to implement some form of communication with the DSP.

Note: *On the Signal Ranger board, the HPI is used to interface the DSP to the USB controller. The USB controller, which is a small microcontroller that implements USB communications with the PC, in effect sees the DSP as one of its peripherals.*

4. MULTI-CHANNEL BUFFERED SERIAL PORTS (MCBSPS)

The TMS320VC5402 has 2 McBSPs that can be used to interface several devices to the DSP, or to provide processor-to-processor communications in multi-DSP parallel architectures.

The McBSPs are very sophisticated serial ports that implement a large variety of serial interface protocols. They provide a direct (glueless) interface with many serial peripherals in the industry. The support of such a large number of serial standards accounts in a high part for their complexity. Among the serial standards supported by the McBSPs are:

- The standard serial protocol used by Texas Instruments for interfacing I/O devices such as Analog Interface Circuits.
- The Serial Peripheral Interface (SPI) standard developed by Motorola for the interfacing of peripherals. Many peripherals in the industry, such as ADCs, DACs and serial ROMs, implement the SPI standard.
- The Inter-IC sound bus (I^2S) standard developed by Philips for the interfacing of audio devices (ADCs, DACs, audio codecs, audio processors…etc.)
- The T1/E1 framing standard used in telecommunications applications.
- The Audio Codec 97 (AC97) standard developed by Intel for the interfacing of audio devices in the personal computer area.
- The ISDN-Oriented Modular Interface rev.2 (IOM-2) developed by European telecommunication companies for the interfacing of communication devices (Codecs, Modems, …etc.).

Each McBSP provides a bi-directional serial bus that can be used to interface multiple peripherals, signaling at up to 50Mb/s. They include 2 levels of transmit buffers and 3 levels of receive buffers. These buffers reduce the latency constraints on the DSP that manages them. The McBSPs can be connected to the DMA (see below), which can provide automatic transfer of data to/from the on-chip memory of the DSP, further simplifying the DSP management task.

Note: *On the Signal Ranger board, each McBSP is used to interface the DSP to a bank of 4 Analog Interface Circuits (AICs), each one consisting of an ADC and a DAC. Through its McBSPs, the DSP has access to 8 analog inputs and 8 analog outputs.*

5. DIRECT MEMORY ACCESS (DMA)

The TMS320VC5402 has a Direct Memory Access controller. This device allows direct data exchanges between peripherals, without intervention from the CPU. For instance, it can be used to automatically transfer data words from one memory location to another. During DMA transfers the CPU is free to carry out other tasks. DMA transfers do not interfere with data movements on the internal busses that are controlled by the CPU.

The DMA can transfer single words, or complete blocks at a rate of one word per 4 CPU cycles. It has 6 independent channels that can control up to 6 transfers in parallel. The 6 channels share the bandwidth.

The DMA can send interrupts to the CPU to notify it that a transfer is completed.

DMA transfers can be initiated by the CPU, or by particular hardware events such as:

- Reception or transmission on a McBSP.
- Timer interrupt.
- Interrupt from one of the on-chip peripherals.

One of the best uses of the DMA is the automatic transfer of data between the McBSPs and the on-chip RAM. Data words can be transferred to/from the McBSPs, one word at a time, and be automatically stored in memory. This greatly simplifies the CPU's management of the McBSPs.

***Note**: On the TMS320VC5402 the DMA does not have access to the external bus. It does not allow automatic data exchanges with external peripherals.*

6. BIO AND XF SIGNALS

The TMS320VC5402 has 1 general-purpose input pin (BIO), and 1 general-purpose output pin (XF). These pins can be used to interface the DSP to specialized external logic.

For instance, on the Signal Ranger board, the XF output is used to control the reset signals of all 8 Analog Interface Circuits. The logic level on the XF pin is decided by the state of the XF bit in the ST1 register.

The BIO input can be used to sense a logic level on external logic devices. The level on the BIO input pin can be used as a condition of most conditional instructions.

7. CLOCK GENERATOR

The clock generator of the TMS320VC5402 supports the choice of an on-chip or external oscillator. An internal programmable Phase Locked Loop (PLL) allows the fine adjustment of clock frequency, while the CPU is in operation.

An external oscillator is useful in applications where the precision and stability requirements of the clock are out of the ordinary. For normal applications, we recommend the use of the internal oscillator for 3 reasons:

- It only requires an external crystal, which is much less expensive than a complete external oscillator.
- An external oscillator may be more precise, but its power consumption is usually much higher than the consumption of the internal oscillator.
- The clock input of the TMS320VC5402 works in 1.8V logic, and is not tolerant to 3.3V signals. Most external oscillators on the market today provide 5V or 3.3V signals, and are difficult to properly interface to the DSP's clock input.

***Note**: On the Signal Ranger board the DSP uses a 10MHz crystal connected to the on-chip oscillator. From this base frequency, the PLL synthesizes the 100MHz clock required by the CPU.*

The PLL of the clock generator allows the CPU clock frequency to be finely adjusted. The PLL allows the base crystal frequency to be multiplied by integer numbers. In addition to the PLL, a programmable divider provides fractional frequency adjustments. The use of a PLL has several advantages:

- It allows the generation of a high (100MHz) clock frequency from a relatively low crystal frequency. Crystals in the 10MHz range are easier to manufacture with good precision. They are more common, and less expensive than crystals in the 100MHz range.
- It allows the adjustment of the clock frequency to be carried out under control of the software that executes on the DSP. For CMOS devices, such as most microprocessors today, there is a direct proportionality between frequency of operation and power consumption. Lowering the clock frequency when the application does not require high computational speed allows the CPU to save energy. This is an essential function for most mobile battery-operated devices.

© BRUNO PAILLARD ING.

***Note**: On the Signal Ranger board, the PLL is adjusted to a factor of 10. With a 10MHz crystal, the PLL synthesizes a 100MHz clock frequency, which is the maximum for the TMS320VC5402. The clock frequency can be adjusted at any time by the user code.*

***Note**: The PLL can achieve clock frequencies higher than 100MHz, from the 10MHz crystal frequency. Obviously this is not recommended, since the DSP is not guaranteed to work reliably at frequencies above 100MHz.*

8. WAIT-STATE GENERATOR

A fast CPU like the TMS320VC5402 is capable of performing very fast accesses to external peripherals. At the fastest, accesses can be performed in 1 clock cycle, i.e. 10ns!

For some peripherals, such an access may be too fast to provide a reliable exchange of data. Referring to the read chronogram in figure 3-4 for instance, a particular peripheral may take more than 10 ns to drive the requested data word on the data bus. This is not counting the delay from the time the address appears on the address bus to the peripheral actually being selected. To allow the CPU to be interfaced to slower peripherals, **wait-states** can be inserted in the read and write cycles. Wait-states simply increase the length of the read and write cycles by a certain number of clock cycles, and slow them down enough for the access to be reliable. The length and the type of wait-states must be adjusted for the speed and behaviour of each type of external peripheral.

Three types of wait-states can be used:

- **Software wait-states:** Software wait-states are the easiest to use because they require no additional signals in the interface logic. The CPU has a wait-state generator that can be adjusted to increase the length of each read and write cycle by up to 14 clock cycles. Read and write cycles can be slowed down to 140ns. This allows a direct interface to most peripherals, even the slowest ones. The wait-state generator provides great flexibility to specify separate numbers of wait-states for separate address spaces and memory locations, therefore allowing different access speeds to be used with different peripherals.

***Note**: In the Signal Ranger board, the external memory device requires one wait state to be reliably accessed. The communication kernel that is loaded shortly after reset configures the wait-state generator for one wait-state in this memory zone.*

- **Hardware wait-states:** Hardware wait-states can be used to lengthen the read and write cycles indefinitely. To support hardware wait-states, the peripheral must have a *wait* output notifying the DSP when it is able to respond to the access. This output is connected to the READY input of the DSP. When hardware wait-states are used, the DSP initiates the access and freezes its bus signals. Bus signals stay frozen until the peripheral resumes the access by activating its wait output. This type of wait-states is used to interface the CPU with peripherals that may not be able to respond to an access at all at certain times. This situation happens for instance when trying to access dynamic RAM circuits, which are not accessible when they are in their refresh cycle. The process allows the access cycle to be stretched until the RAM circuit completes its refresh cycle and becomes accessible again.

- **Bank switching wait-states:** When the CPU makes back-to-back accesses to memory locations that physically reside within the same memory device, these accesses can be very fast because the same device stays selected between the accesses. The address decoding delay, as well as any internal memory select delay, is only seen during the first access to the device. When back-to-back accesses cross an address boundary between two separate devices, the first access to the new device may take longer, to account for the delay of deselecting the previous device and selecting the new one. In this situation, it may be necessary to add a wait-state to the access cycle. The wait-state generator can be adjusted to add a wait-state automatically when back-to-back accesses cross the address boundaries between separate devices. It allows the specification of address boundaries in the system with great flexibility.

© BRUNO PAILLARD ING.

Software Development Methodology For Embedded Systems

- 1 Introduction
- 2 Avoiding problems
- 3 Modular programming practices
- 4 Structured code
- 5 Handling peripherals: Interrupts vs. Polling
- 6 High-level language development

1. INTRODUCTION

When a high-level language is used for the development of code, the structure of the language itself often forces the developer to build the code in a very structured manner. Code that is structured and organized is more reliable, easier to debug, and easier to manage in the long term (maintain, modify…etc.).

In the context of embedded systems, the code is often developed directly in assembly language for reasons of efficiency and flexibility. The developer cannot rely on the language to impose structure to the code. Rather the developer must make a conscious effort to achieve a good structure. Good structure and modular design practices is often what distinguishes an inexperienced developer from a professional one.

Assembly programming poses no difficulty when projects are small and simple. However, as projects become larger and more complex, bugs can appear not only within the various software functions that make-up the project, but can also result from unexpected interactions (or interferences) between those functions. This effect can be amplified if the development is carried out by a team of developers.

In the field of embedded systems, reliability is often much more critical than for computer systems. Many embedded systems operate equipment in life-critical applications. One can only think of the consequences that a software crash would have in an automobile Anti-locking Brake System. Even if human life is not directly at risk, loss of life can be an indirect consequence of software failure. An aircraft instrument malfunction, due to a software problem for instance can have dire consequences. A simple PH-meter giving an erroneous reading in a chemistry lab may indirectly put lives at risk.

Insuring the reliability of embedded software accounts in a large part for its cost. The industry estimates that an average time of 1H30 is spent on the development of each line of assembly code. Obviously the estimate takes into account more than writing the instruction itself. Quality testing at the module and integration levels often accounts for a significant portion of this time. The time required for debugging may also play a major role.

Even though software quality control and testing tools and methodologies are receiving a lot of attention recently, the most significant improvement in software quality doesn't come from using sophisticated testing tools and methods, but rather from having the right attitude when writing the code.

In complex and large projects it is imperative to thoroughly think the code before writing the first instruction line. The difficulty of writing assembly code does not come from choosing the right instruction or the right addressing mode (even though it may appear this way to a beginner!). Rather it comes from specifying, designing and structuring the code. Anticipating and avoiding unwanted interactions between modules is both difficult and critically important. Good modular development practices greatly help in minimizing those interactions.

The trial-and error methodology is especially not recommended, because many details may be overlooked and cause serious problems in the long run. With some luck, software bugs may lead to problems that appear often. In this case they will be discovered soon and have a better chance of being corrected. In less fortunate situations symptoms only appear very rarely, which makes the bugs so much more difficult to correct. The software should be properly planned, and then implemented according to the plan.

Code written in assembly language must usually conform to the following constraints:

1. **Program and data must not take a large space in memory**
   This constraint is often present because microprocessor systems designed for embedded applications are often small systems with few resources. RAM and ROM capacity is often limited for a variety of reasons, including cost, power consumption, size…etc. The code must make good use of the little memory available.

2. **The code must execute efficiently**
   In a time when microprocessors seem to be manufactured with ever-increasing computational speed and performance, it may seem that a day will come soon when they will be so powerful that they will be able to support almost any application, no matter how inefficiently the code is written. This view is very misleading. For one thing the pool of potential applications is not finite. Rather, new applications often appear as soon as new processors are able to support these applications within the constraints of market and technology (cost, size, productibility, marketability…etc.). Many scientists and research labs are working on applications that only make sense with the processors that are anticipated a few years from now. So even the best performance offered on the market today is not enough, and will probably never be!
   If the code is written to execute efficiently and take full advantage of all the resources of the processor, it will often require a smaller, less expensive processor, which could give the manufacturer a real market advantage.

3. **The code must have low power consumption**
   It might seem strange to discuss the power consumption of the software. However it is really a combination of the two previous constraints. Software that uses memory space efficiently needs less capacity, and less capacity directly translates into less power consumption.
   Software that is written to execute efficiently requires a slower clock to do the same as a less efficient software. As we saw in chapter 2, for microprocessors implemented in CMOS technology (almost all microprocessors today) there is a direct proportionality between clock frequency and power consumption. Running the microprocessor at slower clock frequencies directly translate into power

© BRUNO PAILLARD ING.

savings. For battery-operated mobile applications, this is a critical factor. In many mobile embedded systems, in which the CPU accounts for the most significant portion of the power consumption, an N-fold improvement in the efficiency of the code directly translates into the same N-fold increase in battery life.
For aerospace applications, code inefficiency may translate into extra weight that has to be carried in the form of batteries.

4. **The code must be easy to modify and maintain**
To achieve this, the code must be easy to understand. Simple straightforward solutions should be used rather than clever ones, even if it sometimes means writing less efficient code.
In addition, the code should be written in a modular fashion, with good structure and hierarchy. If the modules are fairly independent from each other, a module may be independently tested, corrected or replaced without having to modify the rest of the code. Modularity greatly facilitates the maintenance of the code.

5. **In most projects the code must be developed within tight deadlines**
The time that is available to complete the project is not unlimited. To conform to the often-tight deadlines, the developer may choose proven solutions rather than sophisticated ones that may carry more risk, even sometimes at the cost of efficiency.

Constraints 5 and 6 often go against constraints 1, 2, and 3, and conspire to lower the efficiency of the code. Design is a balancing act and it is often difficult to weight appropriately these and other design constraints. As a matter of policy, the priority is usually given to constraints 4 and 5, because the advantages of writing a straightforward code that is easy to understand and maintain often outweigh the small loss of efficiency that may sometimes be incurred. Using “programming tricks” to increase efficiency should only be considered when all other solutions have failed.

However, it should also be noted that the balance between these five design constraints is very different in the context of embedded systems from what it is in the context of computer systems. In embedded systems, constraints 1, 2 and 3 are usually given more importance than in the context of computer system. This is one reason why the design of code in assembly language is so common in embedded systems.

2. AVOIDING PROBLEMS

The debugging phase is usually the one that carries the most risk, and is the most difficult to plan in a large software development project. Minimizing its impact on the project takes careful planning.

In a complex project, many hours of debugging time can often be saved simply by:

- Carefully (and completely) reading the relevant documentation. This is especially true when writing code to manage peripherals. The data sheets of the peripherals should be read in their entirety (this includes footnotes!). In many cases small details that were documented but overlooked cause all the problems.
- Carefully planning and designing the code at high-level.
- Writing specifications for the various modules of code that have to be developed.

- Implementing the code as close as possible to the original specification.
- Properly documenting any specification changes that have to be made.
- Thoroughly and individually testing the various modules of code before their integration in a complex application.

3. MODULAR PROGRAMMING PRACTICES

One of the most widely used practices of modern programming is the organization of code in a hierarchy of well-defined modules.

A complex software application is not built from CPU instructions. Rather, simple functions are assembled into more complex ones, which in turn are assembled into even more complex ones… and so on. At every level, modules are developed that are single functions, or groups of functions performing related tasks. At every level modules should be simple.

Modular development has several advantages:

- At each level of the development and for each module, the software designer only has to focus on a relatively simple piece of code. This simplicity makes it easier to understand and to anticipate the relationships and interactions that this module could have with the rest of the code.
- The relative simplicity of the modules makes them easy to test completely and in all possible conditions and configurations. The set of conditions underlying the execution of a single module is usually very limited. By contrast, it is very difficult to reproduce (and even to anticipate) all the conditions that may underlie the execution of a complex application.
- If modules are written to be general enough, they may be used in more than one place in a project, and sometimes in even more than one project. The reusability of the code is a great advantage, not only because using recycled code saves development time. Recycled code has normally been tested and qualified and usually provides a greater degree of reliability than newly developed code.

At the level of the assembly code, as well as at high-level such as in C or C++, this modular hierarchy is simply supported by the function call mechanism. However, to take the most advantage of modular programming, a few general principles must also be followed:

3.1. Independence:

Modules should be developed so that they are as independent as possible from each other. Modularity only simplifies the development if the relationships between the modules are not intricate and difficult to assess. Ideally, developers should be able to modify, correct, or even completely replace any module without having to touch (or even worry about) the rest of the code.

- Independence is often obtained by insuring that each module only allocates and handles the resources that it needs. For instance a section of code that requires a certain interrupt to be temporarily disabled should not act on the global mask, which would disable all interrupts, and possibly disrupt the operation of other modules.

- Sharing resources is also a great source of unwanted interference between modules. For instance, using the same static variable to serve two separate tasks in separate modules might seem to save memory space, but it also creates unwanted interference between the two modules.
- Some resources are unavoidably shared between modules. CPU time is an obvious example. Shared resources should be used as sparingly as possible by the modules that need them. For instance an interrupt service routine should only contain the code that is absolutely essential to its function, because the longer it takes to execute, the longer it blocks other interrupts.

3.2. Self-sufficiency:

Modules should be self-sufficient. For instance, a module that requires for its own execution that a shared resource be in a certain state, or configured a certain way, should not rely on the fact that another module has already configured the resource correctly for its operation. Relying on actions taken by another module is a dangerous practice. If this other module is modified or eliminated, the required configuration might not be correctly implemented for our module. In some instances this practice may seem inefficient because it leads to the same actions being unnecessarily repeated by various modules. However it greatly contributes to the minimization of unwanted interference between modules, further insuring their independence.

***Note**: In order to minimize interferences with other modules, a module that modifies the state or configuration of a shared resource should, as much as possible, return it to its initial state after its operation is completed.*

3.3. Consistency:

Modules should be developed in a consistent manner. Consistency contributes to the compatibility of the various modules, and facilitates their integration into a complex application.

For instance, argument passing conventions, and context protection conventions, should be established and applied across the project. Such calling conventions greatly facilitate the developer's task because when code is developed at the caller level, the developer does not have to worry about the function being called. When developing code at a function level, the developer does not have to worry about the caller level.

3.4. Generality:

To be able to reuse them as often as possible, modules should be as versatile and as general as possible.

- For instance, instead of using a constant to adjust the behavior of a function, passing the value as an argument might make the function useful in other circumstances.
- When writing the set of functions required to handle a certain peripheral (what is called the peripheral's **driver**), the developer should consider writing functions to support all the features and operating modes of the peripheral, not just the functions required by the present application or project. This is more work, but it will pay-off when the same driver can be reused in another project.

***Note**:* *It is clear that function calls are at the base of modular design. Even though the general term "module" is used in the discussion, the four principles discussed above should be equally applied to the development of functions, which are elementary modules.*

4. STRUCTURED PROGRAMMING

4.1. What is structured programming

Structured programming is a concept that has been introduced in the 1970's to help the software designer cope with the complexity of large applications. Most contemporary high-level programming languages, such as Pascal, Fortran, C, C++…etc., force the developer to follow the principles of structured programming. They are **structured languages**. Even though the term is often used to describe high-level languages, the principles of structured programming can (and should) be applied to the development of code in assembly language as well.

Structured programming takes the concepts of modular and hierarchical design and applies them using a limited set of rules and basic structures.

To qualify as *structured*, a program must be constructed from basic structures having only one entry point and one exit point. In addition, each module should be built exclusively from 3 basic types of structures, discussed in the following section.

4.2. Flowcharts

Flowcharts are used to describe the execution of a program. They represent the cause-and-effect relationships between the various sections of code. Since a processor executes code sequentially, flowcharts also implicitly represent the execution sequence. Flowcharts are tools that are used to design and document the code. In the design stage, they are extremely useful to build the code in a structured manner.

Because the main information that is conveyed by a flowchart is the sequencing of the various sections of code, they are poorly adapted to represent parallel software processes. However, even when parallel processes are implemented in a software system, at the lowest level the processor always executes the code sequentially. Flowcharts can always be used to describe low-level code.

Figure 9-1 describes the primary symbols used to create flowcharts. These are:

- **The flow lines and flow arrows**.
  Flow lines connect the various boxes and indicate the sequence of events.
- **The beginning and end terminals**.
  The beginning and end terminals are used to indicate the entry and exit points of a flowchart.
- **The process box**.
  The process box is used to encapsulate a simple or complex process. The description of the process is written in the box.
- **The decision**.
  The decision symbol represents the implementation of a binary test (a test that can only have two results). Each possible result leads to a specific process. This

seems to be restrictive, however in assembly language basic tests are always binary. Multi-output tests can be implemented by combining several binary tests.

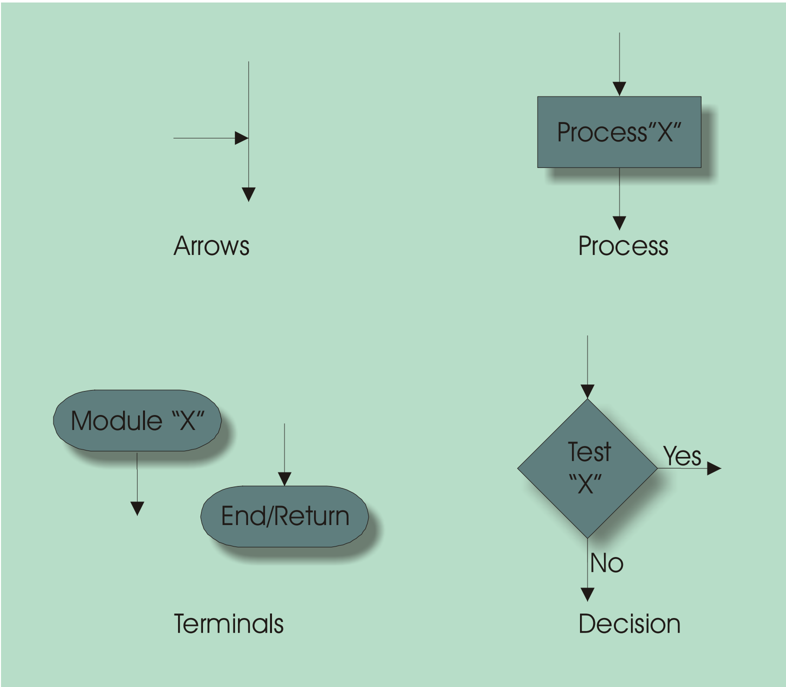


Figure 9-1

Figure 9-2 describes secondary symbols used in flowcharts. These are:

- **Breaking and continuation connectors**.
  They are used to symbolize a direct connection between two far-away points.
- **Off-page connectors**.
  They are used to two connect points of the same flowchart that reside on separate pages.

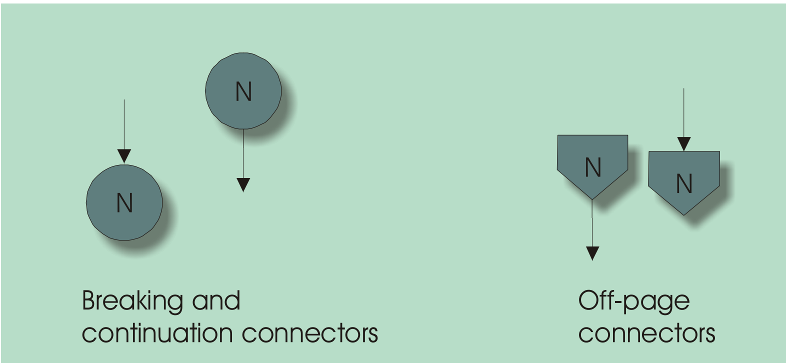


Figure 9-2

Even though these secondary symbols are available, we strongly advice against their use. The main purpose of a flowchart is to present the sequence of a program in a clear and simple way. When a flowchart is so complex that it has to run on several pages, or

that breaking and continuation symbols have to be used, it is time to rethink the presentation of the flowchart (and possibly also the organization of the code).

Just as the program should be designed using a modular hierarchical strategy, the same strategy should be applied to the presentation of the flowchart. The top-level of the flowchart should be simple and present only high-level processes. If needed, complex processes in the top level should be expanded and presented in following flowcharts, usually on separate pages. Complex processes in these flowcharts may be further expanded in still other flowcharts… and so on. Modular construction allows the design to be simple at every module of every hierarchical level. If the design is modular, there is no reason for any flowchart describing the design to be complex.

Figure 9-3 presents an example of a flowchart. In this flowchart, the first process box could be expanded into the flowchart in figure 9-4.

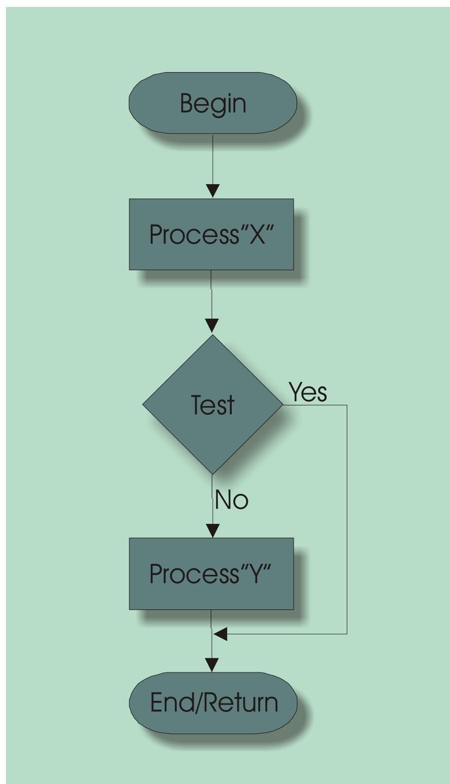


Figure 9-3

Figure 9-4 presents an example of unstructured code. Loops are intertwined, and 4 or 5 different paths lead to some points of the flowchart. This flowchart is very difficult to follow.

© BRUNO PAILLARD ING.

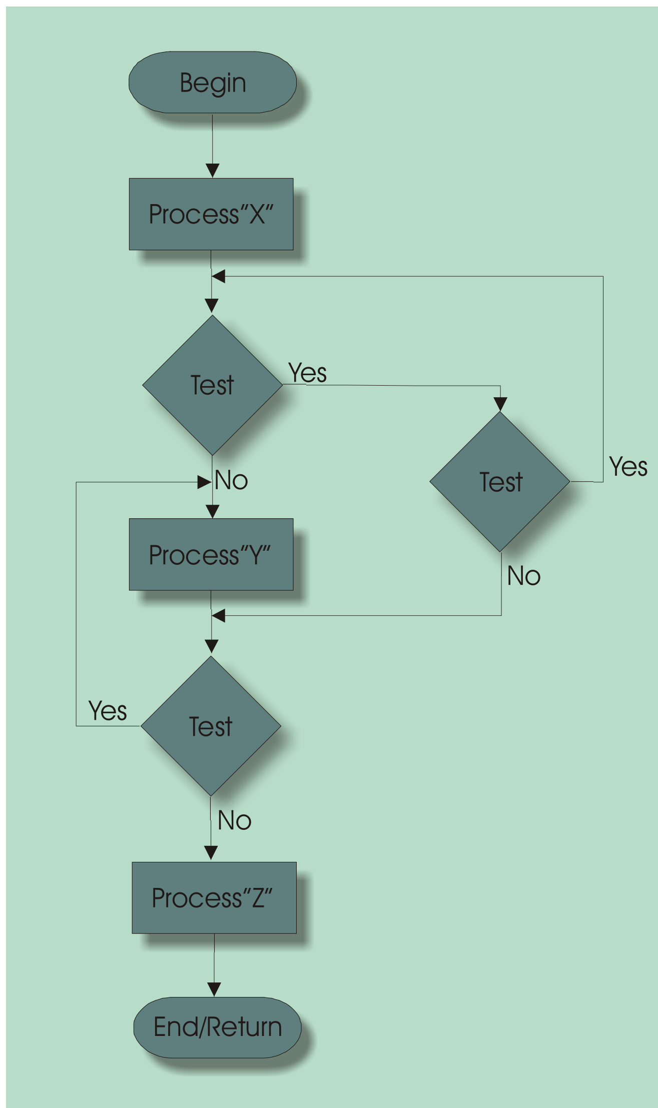


Figure 9-4

4.3. Basic structures

Structure programming defines only 3 types of elementary structures. All the code should be constructed from only these 3 types:

- **The SEQUENCE**:
  The SEQUENCE is the simplest structure and is presented in figure 9-5. It depicts several processes executing in sequence.
- **The IF-THEN-ELSE**:
  The IF-THEN-ELSE structure is presented in figure 9-6. This structure allows the execution of either of 2 processes, conditionally to the result of a test.
- **The WHILE**:
  The WHILE structure is presented in figure 9-7. This structure allows the execution of a process in a loop, as long as the result of a test is true. As soon as the result of the test becomes false, the execution exits the loop.

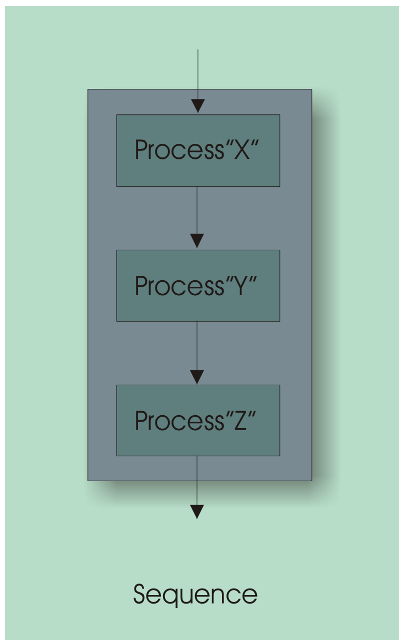


Figure 9-5

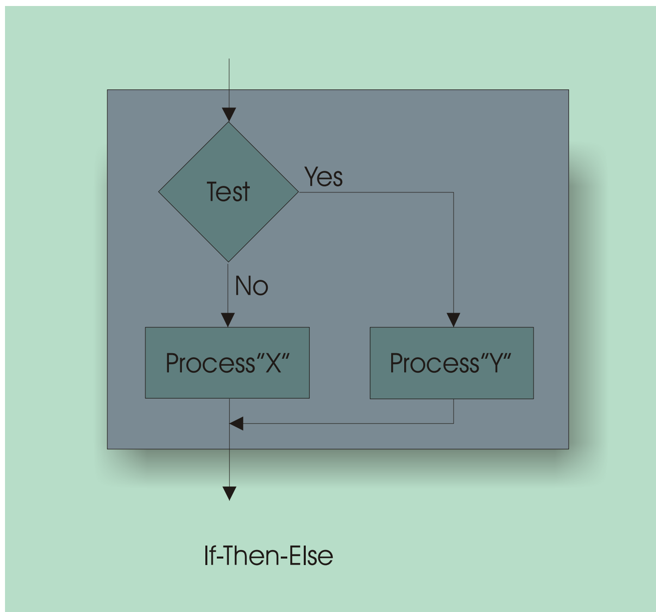


Figure 9-6

© BRUNO PAILLARD ING.

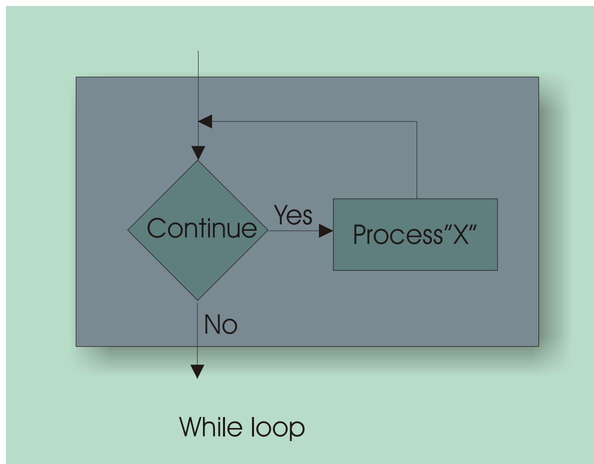


Figure 9-7

Even though they are not absolutely needed, a few useful secondary structures are usually accepted. These are:

- **The DO-UNTIL**:
  The DO-UNTIL structure is represented in figure 9-8. It is similar to the WHILE structure. However the test is performed only after the process is executed once.
- **The FOR loop**:
  The FOR loop is a special case of a WHILE structure, in which the test is based on the value of a loop counter. The FOR loop executes a fixed number of iterations.
- **The CASE**:
  The CASE structure is presented in figure 9-9. This structure can be implemented by a cascade of IF-THEN-ELSE structures. This is actually the case when this structure is implemented in assembly language.

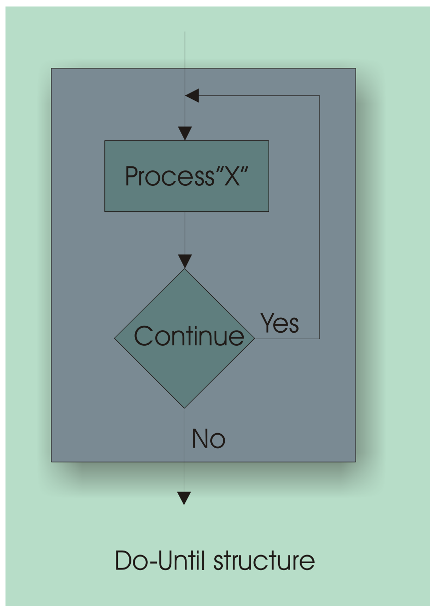


Figure 9-8

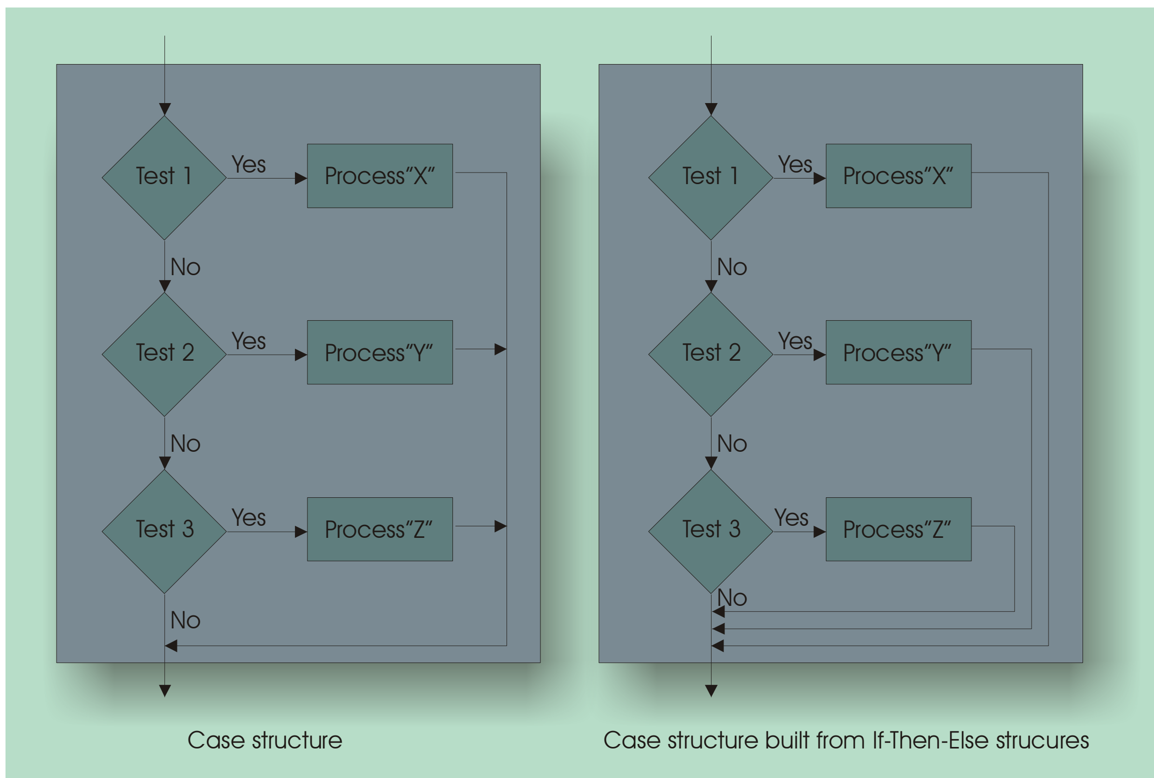


Figure 9-9

© BRUNO PAILLARD ING.

Every basic structure has only one entry point and only one exit point. This is the main reason why structured programming keeps the execution sequence simple and easy to follow. Using only these basic structures in the design of a program may seem restrictive. However, almost any complex operation can be accomplished by combining these basic structures at different hierarchy levels.

Figure 9-10 shows an example of a complex operation built in a structured manner. Individual structures have been identified at several levels by different shades of grey. Each process box in the figure could be further detailed by a complete flowchart presented in another figure.

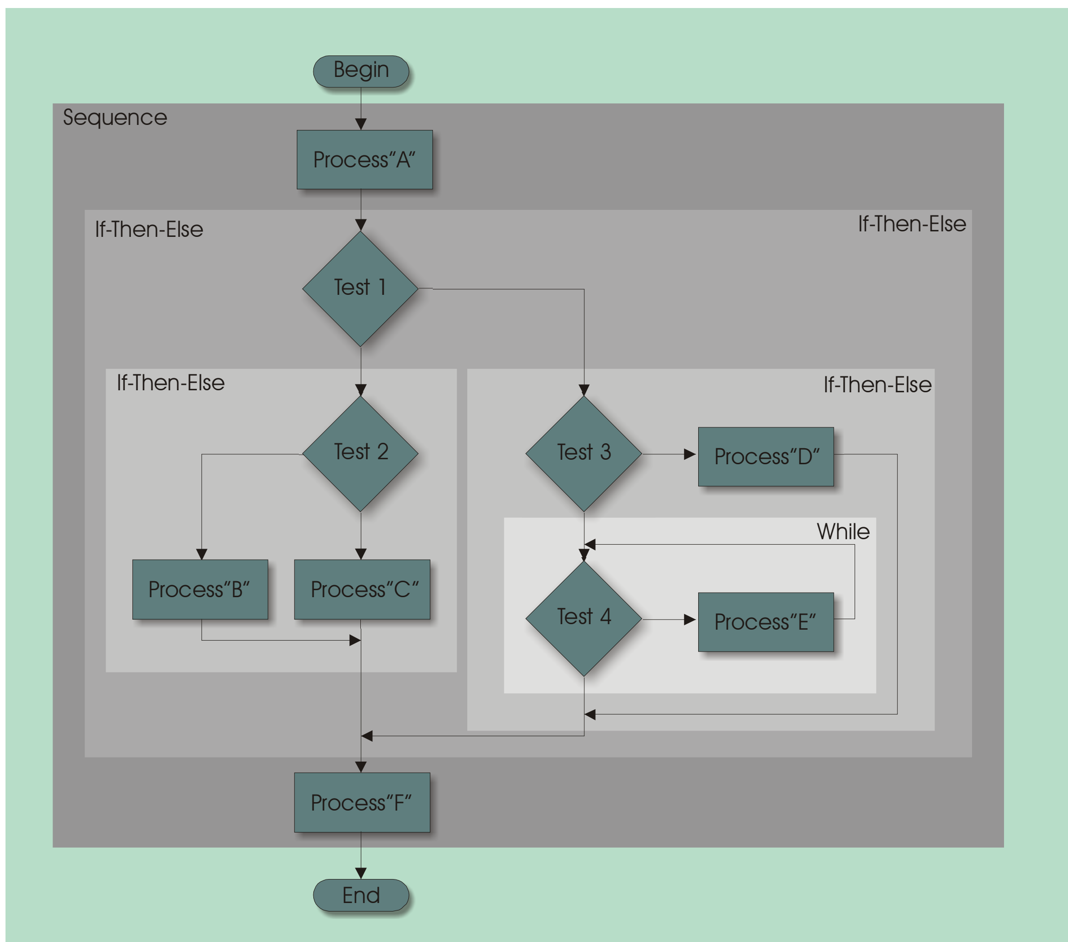


Figure 9-10

By contrast, the flowchart that was presented in figure 9-4 cannot be decomposed into basic structures as this one.

***Note**: Transforming a program that does not conform to the rules of structured programming, into a structured one is very difficult. It usually involves analyzing the program in detail, and constructing a structured one from scratch to do the same thing. It is generally much easier to design the program to be structured from the beginning.*

4.4. Implementation of the basic structures on the TMS320C5402

The following sections give TMS320VC5402 assembly language examples of the basic structures.

4.4.1. WHILE loops

This type of loop implements the test at the beginning of the loop. There are two ways to implement a while loop:

- A conditional branch instruction (BC) can be used to implement the test. This instruction allows many conditions to be tested. It is especially useful to test arithmetic conditions in an accumulator (>0, <0, =0… etc.).
- Another type of conditional branch instruction (BANZ) can be used to count a specific number of loops. This instruction tests the content of an auxiliary register and branches if the content is not zero. A modification of the auxiliary register (usually a decrement) can be performed in the same cycle. This instruction is very useful to implement FOR loops. An example of a FOR loop using the BANZ instruction was seen in section 8-4.

The *InvArray* function described in the following example reads successive memory contents beginning at a given address. If the value is not zero, its opposite is taken and written back to the same address. If the value is zero, the process is stopped.
The beginning address is passed to the function by the auxiliary register AR4.

```
InvArray:
      Loop:
0302H       LD     *AR4,A
0303H       BC     Stop,AEQ
0305H       NEG    A,A
0306H       STL    A,*AR4+
0307H       B      Loop
      Stop:
0309H       RET
```

The function does the following:

- Loads accumulator A with the first value at the address pointed to by AR4.
- Implements the end-of-loop test (is accumulator A zero?)
- If not the opposite of A is calculated.
- Writes the content of A back to the original address pointed to by AR4. Increments AR4, which now points to the following address.
- Branches unconditionally to the beginning of the loop.

4.4.2. FOR loops

FOR loops are a special case of WHILE loops in which the end-of-loop test is the completion of a predetermined number of iterations.

There are several ways of implementing FOR loops on the TMS320VC5402:

 © BRUNO PAILLARD ING.

- If there is only one level of nesting, and if the loop only contains one instruction, the repeat-single (RPT) instruction can be used to implement the loop very efficiently. With this solution, the end-of-loop test is performed in hardware by the CPU and does not cost any execution time. This solution should be used carefully when the number of iterations is large, because the loop cannot be interrupted and may lead to long interrupt latencies.
- If there is only one level of nesting, but the loop contains multiple instructions, the repeat-block (RPTB) instruction can be used to implement the loop. Repeat-blocks take a few more cycles to setup, but the end-of-loop test does not cost execution time either. Repeat-blocks can be interrupted.
- When several loops need to be nested within each other, a repeat-single or repeat-block may be used at the innermost level. A BANZ conditional branch instruction, using an auxiliary register as a loop counter, may be used for the outer loops. In principle, repeat-blocks may be nested within each other, but all the repeat registers need to be protected at each nesting level. The cost of this protection in terms of execution cycles, as well as the complexity of the operation makes this solution unattractive.

The *InvArray* function presented in the following example does an operation similar to the previous one, except that it carries out a fixed number of iterations. The number of iterations is passed to the function by the auxiliary register AR5. The loop is implemented using the RPTB instruction.

```
InvArray:

0302H        MAR   *AR5-
0303H        MVMD  AR5,BRC
0305H  Loop: RPTB  Stop-1
0307H        LD    *AR4,A
0308H        NEG   A
0309H        STL   A,*AR4+
030AH  Stop: RET
```

The function does the following:

- Decrements AR5, which contains the number of iterations. The RPTB instruction (as well as the RPT instruction) always iterates once more than the specified number. The minimum number of iterations is 1. It is achieved by specifying 0 iterations.
- Loads the content of AR5 (the number of iterations minus 1) into the block-repeat counter (BRC register).
- Executes the RPTB instruction, which implements the loop.

***Note**: The argument of the RPTB instruction represents the last word of the last instruction of the repeated block. To represent this address using a symbol, the simplest way is to use a label pointing to the next instruction (the label Stop is used in the example), and to use the value of the symbol minus 1 as the argument of the RPTB instruction. The use of a label pointing directly to the last instruction of the block only works if this last instruction has only one word. Indeed in this case the label would point to the first word of the last instruction, rather than the last word of the last instruction.*

The next example implements the test at the beginning of the loop, as the FOR loop normally requires. This is important in cases where 0 loops may be specified. In practice, the specified number of loops may be a variable that is the output of another function, and can sometimes be 0. This technique is also useful when loops need to be nested.

```
InvArray:

0302H Loop: BANZ  T1,*AR5-
0304H       B     Stop
0305H T1:   LD    *AR4,A
0307H       NEG   A
0308H       STL   A,*AR4+
0309H       B     Loop
030BH Stop: RET
```

4.4.3.IF-THEN-ELSE structure

The code below shows an example of an IF-THEN-ELSE. In the example, the content of the memory address 2000_H is tested to determine if it is positive or not. If it is positive the function implements a loop that divides N successive memory addresses by 2, starting at the address specified in AR4. If not, the N memory contents are multiplied by 2. In both cases, N is specified by the content of AR5. The loop is implemented using the BANZ instruction.

The function does the following:

- Loads the content of address 2000_H into accumulator A.
- Tests the sign of accumulator A, and conditionally branches to Div2, or to Mul2.
- At Div2, implements a loop that divides N successive memory locations by 2, starting at the address pointed to by AR4. The division is implemented as an arithmetic shift to the right.
- At Mul2, implements a loop that multiplies by 2 N successive memory locations, starting at the address pointed to by AR4. The multiplication is implemented as an arithmetic shift to the left.

```
Div_Mul:

0302H       LD    *(#0x2000), A
0304H       BC    Div2, AGT

0306H Mul2: BANZ  T1,*AR5-
0308H       B     Stop
030AH T1:   LD    *AR4,A
030BH       SFTA  A,1
030CH       STL   A,*AR4+
030DH       B     Mul2

030FH Div2: BANZ  T2,*AR5-
0311H       B     Stop
0313H T2:   LD    *AR4,A
0314H       SFTA  A,-1
0315H       STL   A,*AR4+
0316H       B     Div2

0318H Stop: RET
```

This is only an example, in practice the same code structure can be used, with a

different tests and processes other than Div2, and Mul2.

5. MANAGING PERIPHERALS: INTERRUPTS VS. POLLING

There are two techniques that can be used to manage peripherals:

- Interrupt-driven management.
- Polling-driven management.

5.1. Polling-driven management

Polling-driven management is easier to implement. It consists in disabling the peripheral's interrupts using the local interrupt mask, and reading the peripheral's interrupt flag at regular intervals. When the flag is set, the service function is simply called using a CALL instruction.

The execution sequence and flowchart for this technique are given in figure 9-11.

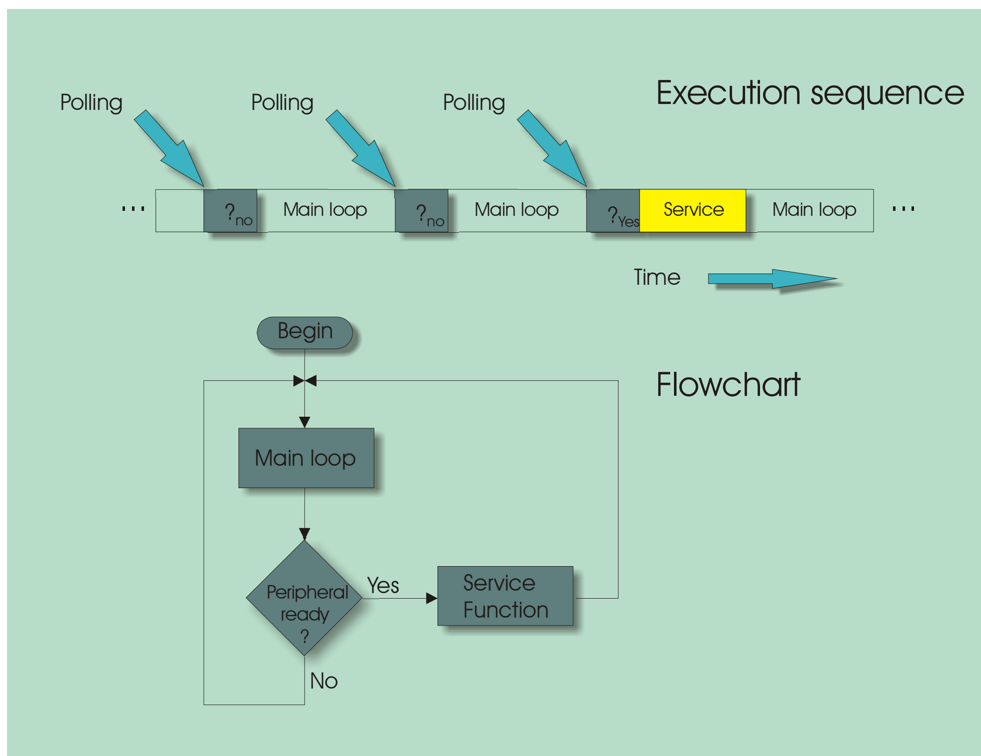


Figure 9-11

The main loop is a never-ending loop that performs its foreground functions, and periodically tests the peripheral's interrupt flag. The service latency can be as long as the time spent between two consecutive tests of the flag.
The main advantage of this technique is that the service of the peripheral occurs at times that are synchronized to the execution of the main loop. The execution sequence is much more deterministic than when interrupts are used. Its main disadvantage is that the service latency may be long if the main loop has a lot of processing to do between the tests of the flag. Also, the regular polling of the flag costs execution time, especially if the flag has to be polled often.

In some cases, it may be possible to anticipate the setting of the interrupt flag. For instance, in some Analog to Digital Converters, a conversion is initiated by writing to a specific control register of the ADC. After being initiated by software, the conversion takes a fixed amount of time to complete, at which point the ADC requests the interrupt to signal that it is ready to be serviced. In such a case, the CPU can begin continuously polling the flag, immediately after initiating the conversion. In this case, the service latency can be very short, typically only a few cycles.

This type of situation is described in figure 9-12.

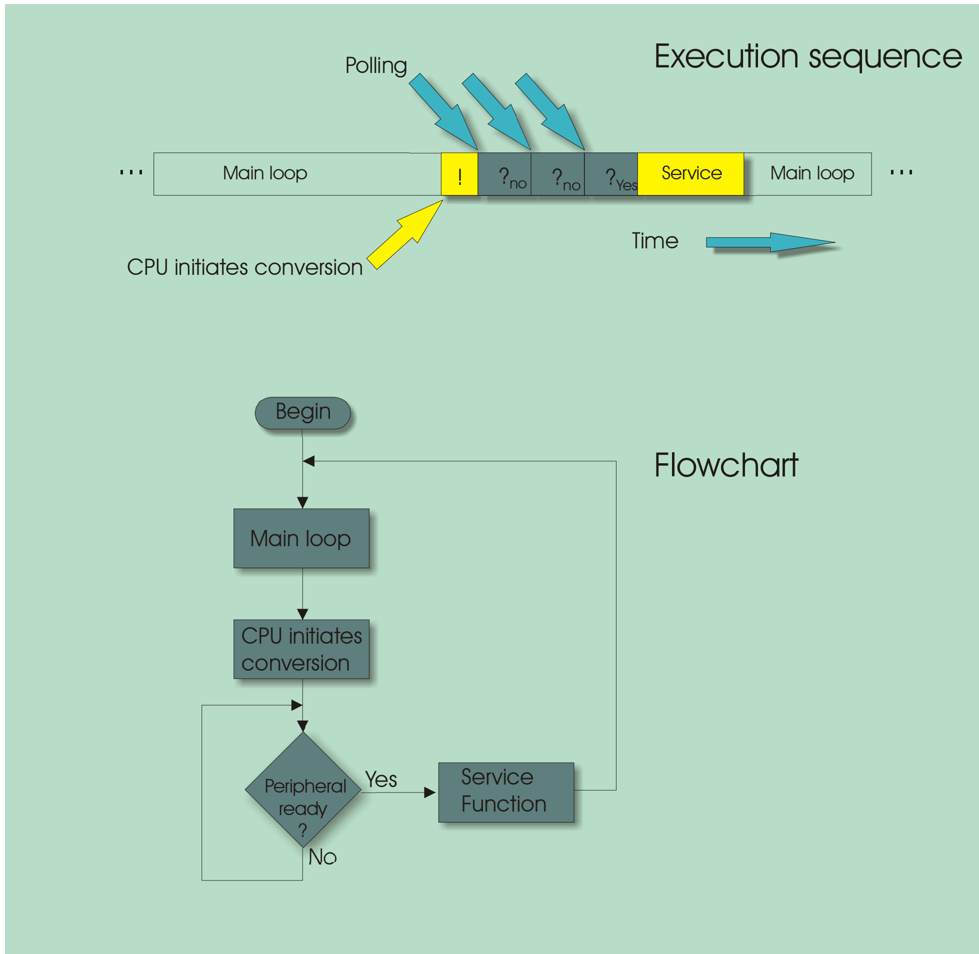


Figure 9-12

In situations where the CPU cannot do anything else until the peripheral has been serviced, polling-driven management is also a good choice.

5.2. Interrupt-driven management

Interrupt driven management is advantageous when it is difficult to anticipate the service request of the peripheral, and when the service latency must be short. In this case, the local mask is set to enable the interrupt, and whenever the interrupt flag is set, it triggers the interrupt service routine. Figure 9-13 describes the execution sequence and flowchart in this case.

 © BRUNO PAILLARD ING.

Note: *A flowchart is poorly adapted to describe asynchronous processes, such as interrupts. In figure 9-13 we used the flowchart symbols in a "creative way", to try to represent the sequence of events. However this is clearly not a standard flowchart.*

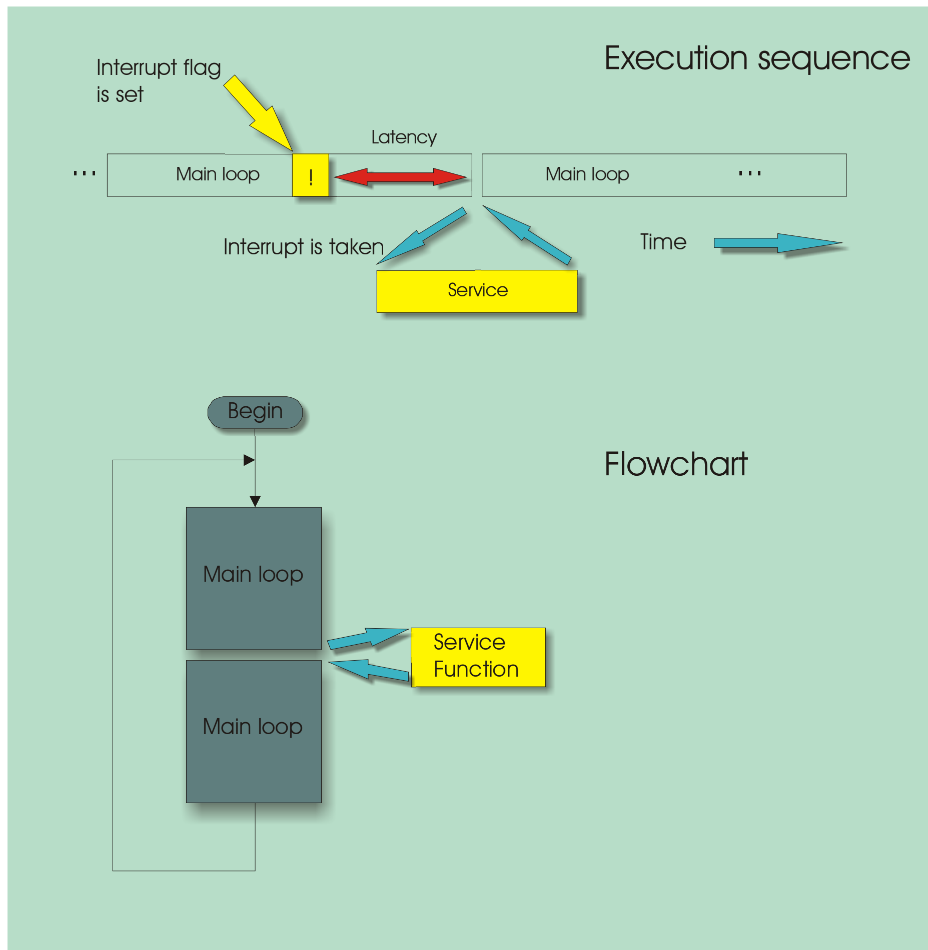


Figure 9-13

Interrupt-driven management is more complex because interrupts are triggered asynchronously from the foreground code. The situation can become very complex when the CPU has several peripherals to manage using interrupts. Of particular concern is the fact that each peripheral must be serviced within a finite delay. Each ISR may block the other ones, and is triggered at times that are very difficult (if not impossible) to anticipate.

5.2.1.Interrupt latency

Four components contribute to the latency of an interrupt:

- Before taking the interrupt, the CPU must complete the execution of all the instructions in the pipeline. This delay depends on the pipeline contents at the time of the interrupt. It is generally short, on the order of a few cycles.
- If a repeat-single instruction (RPT) is in the pipeline, the CPU must compete all the iterations of the loop before taking the interrupt. This delay can be quite long if there are many iterations to perform. In situations where this delay would be

too long, the use of a repeat-block (RPTB) is recommended instead of a repeat-single. The repeat-block takes a few more cycles to setup, but is interruptible.

- Some portions of the foreground code may temporarily disable interrupts. If the interrupt flag is set while the interrupts are disabled, the interrupt will not be serviced until interrupts are enabled again. To minimize this component of the latency the sections of code that disable interrupts should be as short as possible.
- Finally, the CPU might already be executing another interrupt service routine when an interrupt flag is set. In this case the new interrupt will not be taken until the present service routine completes. To minimize this component of the latency, interrupt service routines should be as short as possible. They should only contain the code that is essential to their function.

The last 3 components of the latency are generally the most significant ones.

5.2.2. Code required for interrupt-driven management

The code required to manage a peripheral is called **a peripheral driver**. When a peripheral driver uses interrupts, it usually includes the following software components:

- **The code of the interrupt vector**. It is normally a branch to the entry point of the interrupt service routine. It is essential that this branch instruction be loaded at the correct vector address before the interrupt is triggered. Otherwise the code will crash as soon as the interrupt is triggered. The branch instruction is usually loaded at the correct address before running the entire application. This is the case if the code is loaded into RAM using a loader, or simply when the code is programmed into a ROM. This is called a **static initialization**. In some cases it may be advantageous to have some setup code perform the initialization. This is used in situations where the entry point of the service routine needs to be modified during the execution. This is called **dynamic initialization**.
- **The interrupt service routine**. The interrupt service routine performs all the required operations on the peripheral, and returns to the caller using a return-and-enable instruction (RETE).
- **Peripheral initialization and interrupt setup code**. This code is usually neatly packaged into a single initialization function. This function includes in sequence:
    - The initialization of the peripheral.
    - If necessary an initialization of the static variables that are used to manage the peripheral. The static variables used to pass arguments between the foreground code and the ISR for instance.
    - A reset of the interrupt flag. The interrupt flag may have been set for some time before this initialization and may not signal a recent request. Resetting it at this time insures that the interrupt will not be taken as soon as it is enabled.
    - The unmasking of the local interrupt mask. At this time the developer should consider that interrupts may be taken at any time, because the global mask may already have been enabled for the service of another peripheral.

 © BRUNO PAILLARD ING.

- The unmasking of the global interrupt mask. This operation may be unnecessary because the global mask may already have been enabled for the service of another peripheral. However it is generally implemented for the sake of the module's self-sufficiency.

- **Code to stop the management of the peripheral**. Although it is not always required by a particular project, this function may be created to add value and generality to the driver.
- **Support functions**: When necessary, a peripheral driver also includes support functions that are used to access the peripheral in the foreground, modify its configuration, handle the arguments that are passed between the foreground code and the interrupt routine… etc.

5.2.3. Example of interrupt-driven management

The following example uses the Timer 1 of the TMS320VC5402 to trigger an interrupt every millisecond. The ISR increments a counter. When the value reaches $FFFF_H$ it rolls over to 0000_H. This provides a simple "tic" function that can be used to support various foreground functions that need an absolute time reference.

```
;---------------------------------------------------------------------------
;
;     TIC.asm
;
;     This code uses Timer 1 to trigger an interrupt every 1 ms.
;     It is used to increment a 16-bit "tic" counter.
;
;     This code has been developed to execute on the Signal Ranger DSP board.
;
;---------------------------------------------------------------------------

;---------------------------------------------------------------------------
;
;      MACRO: Sends an acknowledge to the PC (required by the PC interface)
;
;---------------------------------------------------------------------------
Acknow      .macro
      STM   #00Ah,HPIC        ; Send a host interrupt via the HPI
            .endm

;---------------------------------------------------------------------------
; Definitions of constants
;---------------------------------------------------------------------------
            .mmregs           ; to recognize pre-defined register names
                              ;

TIM1  .set  0x0030            ; to define registers not included by the
PRD1  .set  0x0031            ; ".mmregs" directive.
TCR1  .set  0x0032            ;

;---------------------------------------------------------------------------
; Static variables
;---------------------------------------------------------------------------
            .bss        counter,1   ; Static variable used as a tic counter
            .global     counter

;---------------------------------------------------------------------------
; Main loop
;---------------------------------------------------------------------------
            .global     main
            .text
main:
            Acknow                  ; Acknowledge for PC interface
            CALL  TimerInit         ; Setup of timer 1 and interrupts
Here:       B     Here              ; Stays here indefinitely...

;---------------------------------------------------------------------------
; Initialization code
;---------------------------------------------------------------------------
TimerInit:  STM   #0000000000011001b,TCR1      ; The timer is stopped,
                                               ; pre-division factor = 10
            STM   #9999,PRD1                   ; Division factor = 10000
                                               ; (period = 1ms)
```


© BRUNO PAILLARD ING.

```
            STM   #9999,TIM1                    ; Start counting at the
                                                ; beginning of a period
            ST    #0,*(#counter)                ; Reset the counter variable
                                                ;
            STM   #0000000010000000b,IFR        ; Reset any old interrupt
                                                ; that may be pending
            ORM   #0000000010000000b,*(#IMR)    ; Unmask the local mask.
            RSBX  INTM                          ; Unmask the global mask
            STM   #0000000000001001b,TCR1       ; Start the timer pre-division
                                                ; factor = 10
            RET

;-----------------------------------------------------------------------------
; Timer 1 interrupt service routine
;-----------------------------------------------------------------------------
      .global     TimerISR

TimerISR:   PSHM  AR5
            MVDM  counter,AR5
            NOP                     ; Use DAGEN instead of ALU to increment
            MAR   *AR5+             ; counter with 16-bit rollover
            MVMD  AR5,counter       ;
            POPM  AR5
            RETE
```

The code includes:

- The definitions of the addresses of the control registers of timer 1 in the *definitions of constants* section.
- The allocation of the static variable *counter* in the *Static variables* section. The variable is defined as a *global* so that its name can appear in the symbol table used by the mini-debugger.
- The main loop that performs the initialization of the timer and then waits indefinitely. No operation has to be executed in the foreground loop, since everything is done in the interrupt service routine.
- The timer initialization routine (*TimerInit* function). This code does the following:
  - Stops the timer.
  - Initializes the division factor.
  - Resets the static variable *counter*.
  - Unmasks the local interrupt mask of the timer 1. The ORM instruction is used to modify only the mask for timer 1.
  - Unmasks the global mask (INTM).
  - Starts the timer.
- The timer 1 interrupt service routine.
  This code increments the variable *counter* each time it is executed. It includes:

- The protection of AR5, which is used by the function. In the case of an interrupt service routine the responsibility of register protection is always on the function.
- The variable *counter* is incremented using the Data Address Generation Unit (instruction MAR). This is simpler than using the ALU because the increment must be done in 16 bits and roll over from $FFFF_H$ to 0000_H. Using the ALU would have required setting the overflow mode to no-saturation (OVM=0). The ST1 register that contains OVM would then have required protection. All this would have cost extra execution cycles.

The interrupt table section is given below.

***Note**: To be compatible with Signal Ranger's communication kernel, the vector table must be located at address 0080_H. It must define the HPIINT interrupt vector, which is used by the communication kernel (branching to address 0112_H). In this case it also defines the timer 1 interrupt vector, which is used by the user code.*

***Note**: The label "TimerISR" is defined as a global symbol in the main code, as well as in the interrupt vector table. This is required so that the linker may know that it is the same symbol.*

***Note**: A RETE instruction has been placed in all the unused vectors. This is a precaution designed to automatically return to the caller in case one of the unused interrupts would be mistakenly enabled and triggered.*

© BRUNO PAILLARD ING.

```
;---------------------------------------------------------------------------
;     vectors.asm
;
;     This section defines the interrupt vectors.
;
;
;     To be compatible with Signal Ranger's communication kernel, this section
;     must be loaded at address 0080H and contain a branch at address 0112H at
;     the location of the HPIINT vector.
;
;---------------------------------------------------------------------------
            .sect ".vectors"
            .global     TimerISR    ; Defines the global symbol TimerISR

kernel      .set    0112h     ; Entry point of the communication kernel

            .align  0x80      ; The table must always be aligned on a
                              ; multiple of 0x0080
;---------------------------------------------------------------------------
RESET :     B     kernel      ; This interrupt is never taken here since
            Nop               ; the vector table reverts to FF80H on a DSP
            nop               ; reset.
nmi         RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop
sint17      RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop

            .space  52*16     ; Jumps over the software interrupt vectors.

int0        RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop
int1        RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop
int2        RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop
tint0       RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop
brint0      RETE              ; This interrupt is not used.
            nop               ;
            nop               ;
            nop
bxint0      RETE              ; This interrupt is not used.
```

```
            nop                 ;
            nop                 ;
            nop
dmac0       RETE                ; This interrupt is not used.
            nop                 ;
            nop                 ;
            nop
tint        B      TimerISR     ; timer 1 vector
            nop
            nop
int3        RETE                ; This interrupt is not used.
            nop                 ;
            nop                 ;
            nop
HPIINT      B      kernel       ; Required for PC-DSP communications
            nop
            nop
brint1      RETE                ; This interrupt is not used.
            nop                 ;
            nop                 ;
            nop
bxint1      RETE                ; This interrupt is not used.
            nop                 ;
            nop                 ;
            nop
dmac4       RETE                ; This interrupt is not used.
            nop                 ;
            nop                 ;
            nop
dmac5       RETE                ; This interrupt is not used.
            nop                 ;
            nop                 ;
            nop

            .end
```

5.3. Interrupts and reentrancy

An interrupt service routine can (and often does) call a function. When it does it may cause reentrancy problems. More precisely, reentrancy happens when an interrupt is triggered while the CPU is executing a function that is also called within the interrupt. The function is interrupted in the foreground, and is called again (re-entered) through the triggering of the interrupt routine. Functions that can be re-entered without problems are called **reentrant**. Figure 9-14 describes the sequence of events leading to the reentrance of a function.

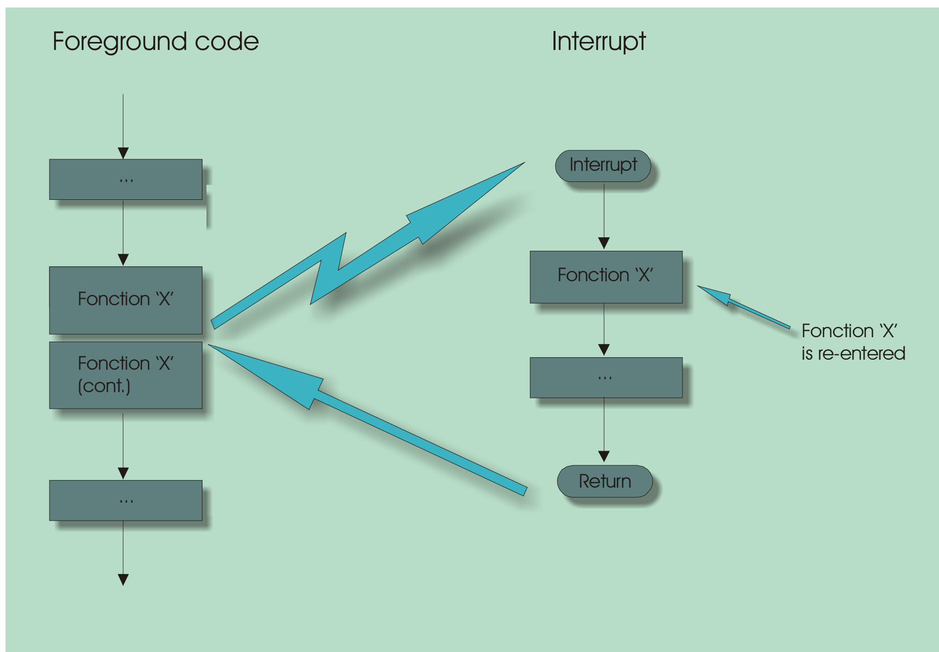


Figure 9-14

The reentrancy problem often appears when the function that is re-entered uses (and in particular stores data in) static variables. In this case, both calls of the re-entered function try to access the same variables, therefore causing an interference between the 2 calls.

For instance let's study the following ADD2 function, which adds 2 to an input argument.

The input argument is passed by value in the lower part of accumulator A. The output argument is also passed by value in accumulator A. The function is given below:

```
ADD2        STL         A,*(#0x2000)
            LD          #2,A
            ADD         *(#0x2000)
            RET
```

The function temporarily stores the lower part of accumulator A (the input argument) into address 2000_H. Accumulator A is then loaded with the immediate value 2. The function adds the content of the accumulator to the content of address 2000_H. The result of the addition is in accumulator A. The function finally returns to the caller.

Note: *Obviously this is not the most straightforward solution to perform this operation. However it provides us with a good example of reentrancy problem.*

Let's see what happens if an interrupt, also calling ADD2, interrupts the function in the middle of its execution:

- The ADD2 function is executing and has just stored the input argument (the lower part of accumulator A) into address 2000_H.
- The interrupt is triggered at precisely this instant.

- The ISR does some processing, loads a new input argument into the lower part of accumulator A and calls ADD2 again.
- At the beginning of the execution of the second call of ADD2, the function stores the new input argument (the lower part of accumulator A) into address 2000_H. This operation corrupts the content of address 2000_H that was used by the first call of ADD2.
- The second call of ADD2 completes its execution with the correct result and returns to the interrupt routine.
- The interrupt completes its execution and returns to the ADD2 function that had been interrupted initially.
- The initial ADD2 function resumes its execution. However, the value stored at address 2000_H is no longer the initial input argument. The function completes its execution with an incorrect result.

The second call of ADD2 corrupts the execution of the initial one. The function is clearly non-reentrant.

When a non-reentrant function is used within an interrupt, this interrupt should always be disabled during the portions of foreground code that call the function. Otherwise, there is always a small chance that the function may be re-entered, leading to a flawed execution. The bug can be especially difficult to correct since it happens very infrequently. However, if the possibility exists it is not a question of “if” it will happen, but rather “when” it will happen.

One obvious solution to the reentrancy problem is to avoid as much as possible the use of static variables, especially if they are used to store temporary results. Variables that are dynamically allocated on the stack are a much better choice for temporary storage. For instance, a reentrant version of the ADD2 function is given below:

```
ADD2        FRAME       -1
            NOP
            STL         A,*SP(0)
            LD          #2,A
            ADD         *SP(0)
            FRAME       1
            RET
```

This new function uses a variable that has been dynamically allocated on the stack to store temporarily the input argument. In this case, a reentrant call of the function would allocate a new variable on the stack. The 2 variables would reside at separate addresses and would not interfere.

***Note**:* *A NOP instruction is used between the instructions FRAME –1 and STL A,*SP(0) to resolve an unprotected pipeline conflict.*

***Note**:* *For this code to be functional, the CPL bit of the St1 register must be set. Otherwise the instructions that use direct addressing will work in the DP-based variant rather than the SP-based variant, and obviously will not give the expected result! We did not show the instructions that set the CPL bit. We assume that this was done prior to the execution of the code presented in the example.*

© BRUNO PAILLARD ING.

6. HIGH-LEVEL LANGUAGE DEVELOPMENT

Even though it usually produces a less efficient code, many non-critical portions of the software may be developed in high-level language. High-level languages offer several advantages:

- They help the developer or developers achieve structure and consistency.
- Complex operations are often easier to describe in a high-level language.
- High-level language development does not require very specialized skills or in-depth knowledge of the target processor. From an employer's point of view it facilitates the management of human resources. Engineers who are not specialists can readily undertake the development of the high-level language modules, sharing the workload of the few specialized engineers who are able to develop in assembly language on a particular platform.

In the field of embedded systems the high-level language of choice is C, and very recently C++. It is usually advantageous to write the higher hierarchical levels of the software application in C or C++ because it is often the place where the most complexity is found, and at the same time it often represents the less critical portions of the code.

In embedded applications it is almost always necessary to develop some modules of code directly in assembly language. Generally the following factors contribute to the development of code in assembly language:

- The C standard is not well defined for many functions that are important in numerical computation and signal processing applications. For instance the overflow behavior during an integer addition (saturation or rollover) is not defined in the C standard. For certain combinations of compiler and CPU, the behavior may even change with the context of execution during the same application.
- A code developed in C or C++ is almost always less efficient than the same code [well] developed in assembly language. At least 2 factors contribute to the loss of efficiency.
  - A C or C++ compiler does not have as much information as the developer about the process that has to be implemented. Many details of the process simply cannot be specified in C or C++. For instance a compiler would not be able to know that a filtering operation uses a symmetric impulse response. In this case, the compiler would not use the highly specialized instructions that the processor provides for this type of operation. As another example, when performing a floating-point division, the C standard requires that the divisor be tested for zero. In a particular application, the developer may know that a certain divisor can never be zero and be able to skip the test. Usually a C or C++ compiler chooses general solutions that are applicable in a wide range of conditions, rather than specialized solutions that are often the most efficient ones.
  - The developer normally chooses to write code in a high-level language to avoid being concerned with the low-level implementation of the code. High-level languages therefore do not give the developer access to low-level functions of the CPU or its ALU. However, it is precisely by getting access to these low-level functions that the developer is usually able to optimize the efficiency of the code.

The C compiler of the Code Composer Studio development environment is an **optimizing C compiler** that has been specifically designed for the TMS320VC5402 DSP. This very sophisticated compiler is capable of optimizing the code to a large

extent. In sections of code where complex logic is implemented the efficiency of the code provided by the compiler is excellent and we recommend its use. However, in sections of code where simple but repetitive numerical computations or signal processing functions are implemented, the efficiency of the compiler is poor. For instance a digital filter may be 10 to 40 times less efficient when written in C than when it is written in assembly language.

For these reasons, the developer is often placed in a situation where the shell, and sometime even a large portion of the code is written in C or C++, but where some critical functions must be written in assembly language. Writing assembly code that must be interfaced to C code requires a perfect knowledge of the target CPU and its assembly language, but also requires a detailed working knowledge of the C calling conventions, and more generally of the way the C compiler synthesizes the assembly code.

The C calling conventions include:

- **Argument-passing conventions**: These conventions specify which arguments are passed on the stack and which are passed in registers. When they are passed in register, which registers are used.
- **Register protection conventions**: For each CPU register, these conventions specify which calling level (caller or function) has the responsibility of protecting the register.
- **Additional conventions**: Additional conventions specify various aspects of the C implementation. For example they specify how the modes of operation of the CPU and the ALU are used by the C compiler, how static variables are initialized, how automatic variables are allocated, how various data types are represented in memory, how dynamic allocation is performed… etc.

© BRUNO PAILLARD ING.

Signal Ranger's Development Tools

- 1 Introduction
- 2 Mini-debugger
- 3 The communication kernel

1. INTRODUCTION

Signal Ranger is an evaluation board. Developers use evaluation boards to get familiar with a microprocessor and its development tools. In addition, development boards may be used to develop and test code for a particular embedded application. In many situations the target hardware does not provide comfortable resources and development support to facilitate the development of code. In many cases, the software development phase must begin while the target hardware is still under development. In these cases the use of an evaluation board may facilitate the developer's work.

Signal Ranger has been designed with a focus on DSP education. However, its very complete hardware resources and software support tools make it well suited to use in industrial and commercial instrumentation applications. For instance, the company Mecanum inc. uses Signal Ranger as an embedded spectral analyzer designed for the full characterization of open-cell poroelastic materials.

Evaluation boards differ slightly from usual embedded systems, in that they do not execute a unique application code at power-up. Rather, the developer uses the board to design and test various software modules and applications.

Evaluation boards provide special software tools that allow the downloading of user code in memory, its controlled execution, and some form of interaction with the tested code. For Signal Ranger, these tools are supported on the DSP side by a special **communication kernel** that provides memory and registers read and write capabilities. On the host side, a **mini-debugger** provides a comfortable user-interface to control and visualize the execution of the user code. In practice, the communication kernel that is provided with Signal Ranger can be used to support specialized host applications, other than the mini-debugger.

2. MINI-DEBUGGER

The mini debugger allows the developer to interactively:

- Reset the Signal Ranger board.
- Download a COFF executable file to DSP memory.
- Launch the execution of code from a specified address.
- Read and write CPU registers.

- Read and write DSP memory.

This mini debugger can be used to explore the features of the DSP, or to test DSP code during development.

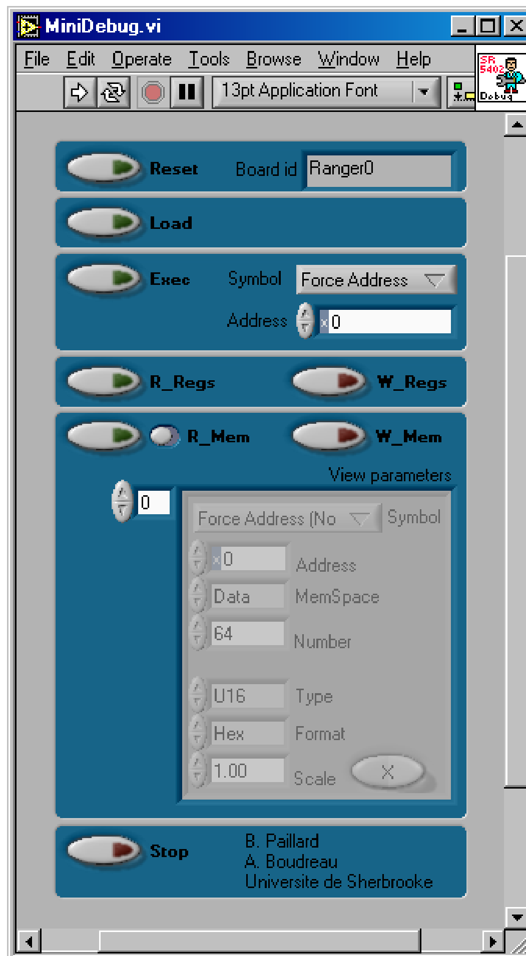


Figure 10-1 User interface of the mini-debugger

2.1. User interface

- **Board id** Selects which Signal Ranger board to use for debugging. If only one is connected to the USB port, "Ranger0" should be used. If not, change the index to match the desired board. Boards are assigned a number from 0 to N at connexion time, in the order of connexion.
- **Reset** Resets the board and reloads the kernel. All previously executing DSP code is aborted.
- **Load** Loads a DSP COFF file. The application presents a file browser to allow the selection of the file. The file must be a legitimate C5402 COFF1 or COFF2 file.

 © BRUNO PAILLARD ING.

Note: It is recommended to always reset the board before proceeding to a load. Otherwise, the execution of the user DSP code can interfere with the load operation.

- **Exec** Forces execution to branch to the specified label or address. The DSP code that is activated this way should contain an Acknowledge in the form of a Host Interrupt Request (HINT). Otherwise the USB controller will time-out, and an error will be detected by the PC after 5s. The simplest way to do this is to include the acknowledge macro at the beginning of the selected code. This macro is described in the two demo applications.

  - **Symbol or Address** is used to specify the entry point of the DSP code to execute. Symbol can be used if a COFF file containing the symbol has been loaded previously. Otherwise, Address allows the specification of an absolute branch address. Address is used only if Symbol is set to the “Force Address (No Symbol)” position.
    When a new COFF file is loaded, the mini-debugger tries to find the _c_int00 symbol in the symbol table. If it is found, and its value is valid (different from 0) Symbol points to its address. If it is not found, Symbol is set to the “Force Address (No Symbol)” position.

- **R_Regs** Reads the CPU registers mapped in RAM at address 0000H, and presents the data in an easy to read format.

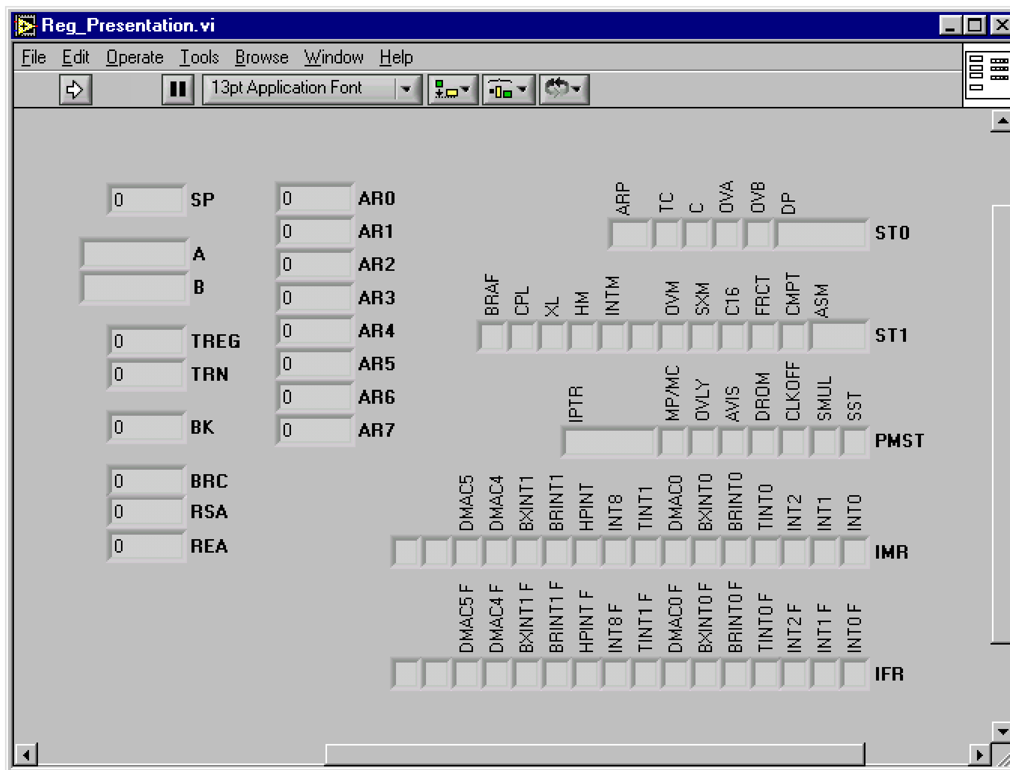


Figure 10-2 Register presentation panel

- **W_regs** Allows to write most of the CPU registers mapped in RAM at address 0000_H. Some fields are grayed out and cannot be written, either

because the kernel uses them and would restore them after modification (ST0, AR1, AR2, AR3), or because their modification would compromise its operation (SP, BRAF, INTM, IPTR, OVLY, HPINT).

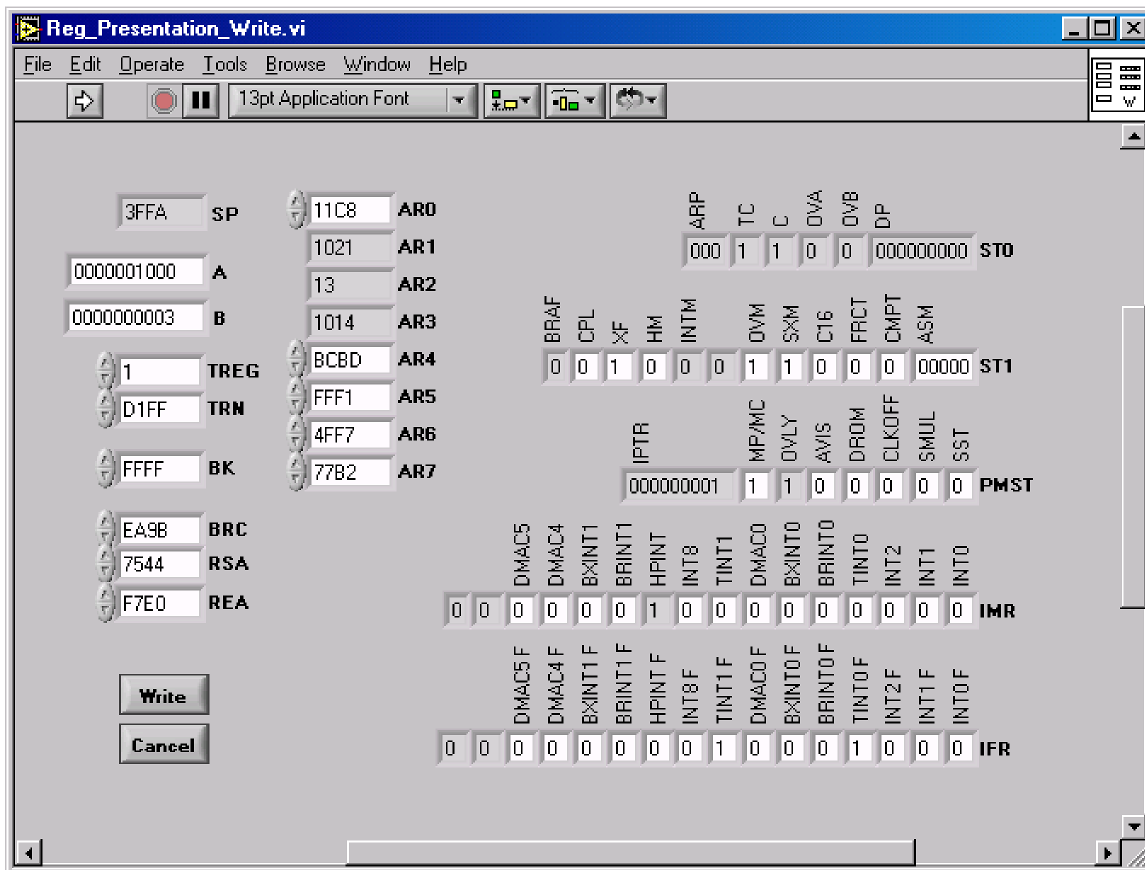


Figure 10-3 Register write presentation panel

- **R_Mem** Reads DSP memory and presents the data to the user.
  The small slide beside the button allows a continuous read. To stop the continuous read, simply replace the slide to its default position.
  The View parameter array is used to select one or several memory blocks to display. Each index of the array selects a separate memory block.
  To add a new memory block, simply advance the index to the next value, and adjust the parameters for the block to display. To completely empty the array, right-click on the index and choose the "Empty Array" menu. To insert or remove a block in the array, advance the index to the correct position, right-click on the Symbol field, and choose the "Insert Item Before" or "Delete Item" menu.

  For each block:

  - ***Symbol*** or ***Address*** is used to specify the beginning of the memory block to display. ***Symbol*** can be used if a COFF file containing the symbol has been loaded previously. If ***Symbol*** is set to a position other than "Force Address (No Symbol)", ***Address*** is forced to the value specified in the COFF file for this symbol.
    The list of symbols is cleared when a new COFF file is loaded, or when the Mini-Debugger is stopped and run again. It is not cleared when the DSP

board is reset.
By right-clicking on the Symbol field, it is possible to remove or insert an individual element.

- ***MemSpace*** indicates the memory space used for the access. The position "???" (Unknown) defaults to an access in the Data space. If ***Symbol*** is set to a position other than "Force Address (No Symbol)", ***MemSpace*** is forced to the value specified in the COFF file for this symbol.
- ***Number*** specifies the number of bytes or words to display.
- ***Type*** specifies the data type to display. Three basic widths can be used: 8 bits, 16 bits, and 32 bits. All widths can be interpreted as signed (*I8, I16, I32*), unsigned (*U8, U16, U32*), or floating-point data. The native DSP representation is 16 bits wide. When presenting 8-bit data, the bytes represent the high and low parts of 16-bit memory locations. They are presented MSB first and LSB next. When presenting 32-bit data (I32, U32 or Float), the beginning address is automatically aligned to the next even address. The upper 16 bits are taken as the first (even) address and the lower 16 bits are taken as the next (odd) address. This follows the standard alignment and bit ordering for 32-bit data, as explained in Texas Instruments document *SPRU131f "TMS320C54x DSP reference set – Volume 1 CPU and peripherals"*.
- ***Format*** specifies the data presentation format (Hexadecimal, Decimal or Binary).
- ***Scale*** specifies a scaling factor for the graph representation.
- ***X or 1/X*** specifies if the data is to be multiplied or divided by the scaling factor.

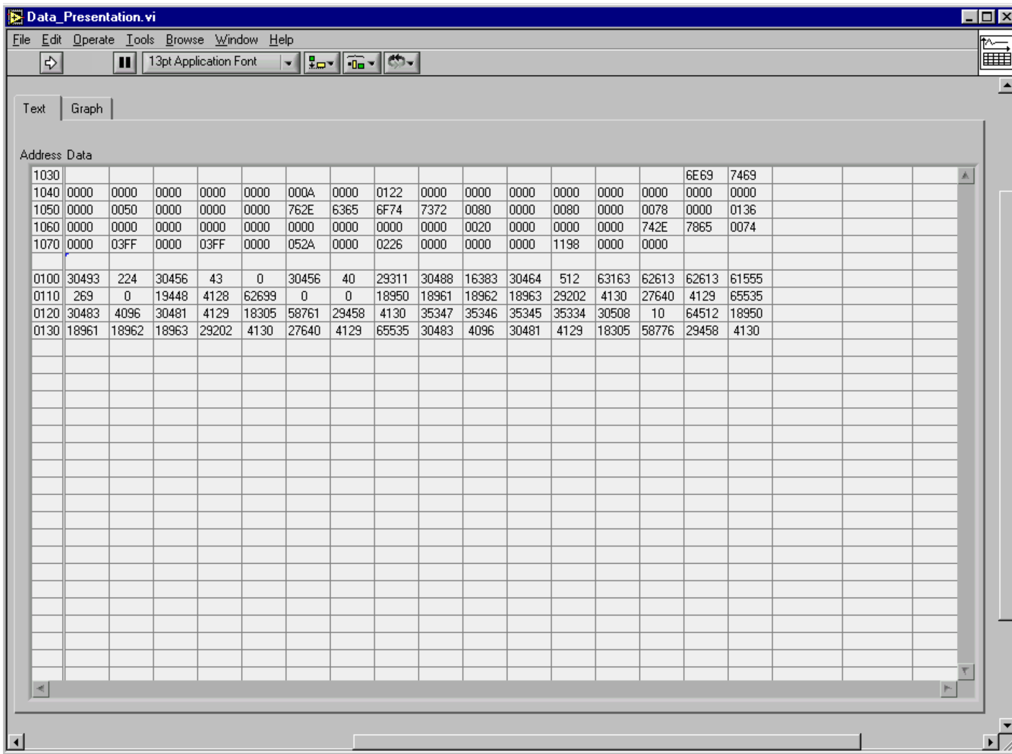


Figure 10-4 Data presentation (text mode)

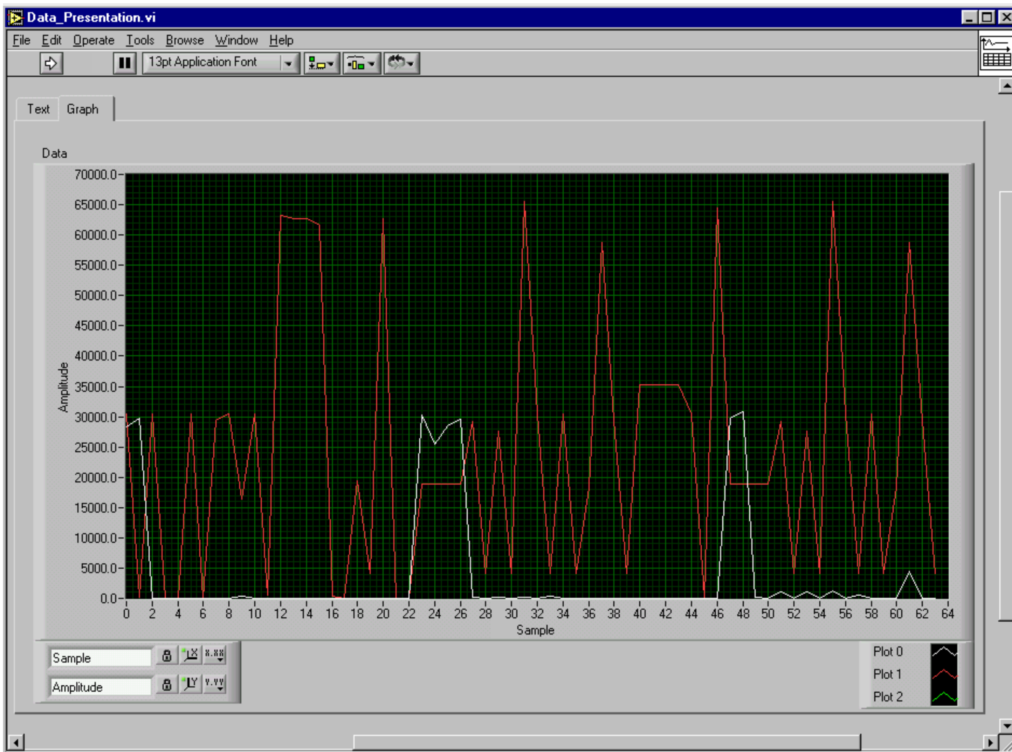


Figure 10-5 Data presentation (graph mode)

The user can choose between *Text* mode (figure 3), and *Graph* mode (figure 4) for the presentation of memory data. In *Text* mode, each requested memory

block is presented in sequence. The addresses are indicated in the first column. In *Graph* mode, each memory block is scaled, and represented by a separate line of a graph.

- **W_Mem** Allows the memory contents to be read and modified. The function first reads the memory, using the View_parameters, and presents a text panel similar to the one presented for the R_Mem function. The user can then modify any value in the panel, and press the Write button to write the data back to memory.
  Several points should be observed:
  - Even though data entry is permitted in any of the cells of the panel, only those cells that were displayed during the read phase (those that are not empty) are considered during the write.
  - When writing 8-bit data, one should only attempt to write an even number of bytes. This is because physical accesses occur on 16-bits at a time. If a write is attempted with an odd number of bytes, the last one (which is not paired) will not be written.
  - Data must be entered using the same type and format as was used during the read phase.
- **Stop** Stops the execution of the mini debugger.

2.2. Wide accesses

When presenting, or writing 32 bit data words (I32, U32 or Float), the PC performs 2 separate accesses (at 2 successive memory locations) for every transferred 32-bit word. In principle, the potential exists for the DSP or the PC to access one word in the middle of the exchange, thereby corrupting the data.
For instance, during a read, the PC could upload a floating-point value just after the DSP has updated one 16-bit word composing the float, but before it has updated the other one. Obviously the value read by the PC would be completely erroneous.
Symmetrically, during a write, the PC could modify both 16-bit words composing a float in DSP memory, just after the DSP has read the first one, but before it has read the second one. In this situation The DSP is working with an “old” version of half of the float, and a new version of the other half.
These problems can be avoided if the following precautions are observed:

- When the PC accesses a group of values in the data or program space (this is not true for the I/O space), it does so in blocks of up to 32 16-bit words at a time. Each of these 32-word block accesses is atomic (the DSP cannot do any operation in the middle of the PC access). Therefore the DSP cannot interfere in the middle of any single 32-bit word access.
- This alone does not guarantee the integrity of the transferred values, because the PC can still transfer a complete block of data in the middle of a DSP operation on this data. To avoid this situation, it is sufficient to also render the DSP operation on any 32-bit data atomic (by disabling interrupts for the length of the operation), then the accesses are atomic on both sides, and data can safely be transferred 32 bits at a time.

3. COMMUNICATION KERNEL

3.1. Overview

On the DSP side, the kernel used to support PC communications is extremely versatile, yet uses minimal DSP resources in terms of memory and processing time. Accesses from the PC will wait for the completion of time critical processes running on the DSP, therefore minimizing interference between PC accesses and real-time DSP processes.

Two USB vendor requests (K_Data_Move and K_Exec) are used to send commands to the DSP.

The exchange of data and commands between the PC and the DSP is done through a 35 words mailbox area at address 1000_H in the HPI-accessible RAM of the DSP.

The DSP interface works on 3 separate levels:

- **Level 1**: At level 1, the kernel has not yet been loaded onto the DSP. The PC relies on the hardware of the HPI and DMA, as well as the USB controller to exchange code and/or data with DSP RAM. At this level, the PC only has access to the DSP RAM between addresses $007F_H$ and $3FFF_H$. This level is used, among other things, to download the kernel into DSP RAM, and launch its execution.

At level 1, the PC has access to low-level functions to perform the following operations:

- Set the color of the LED.
- Read or write the HPI control register.
- Read or write values to/from the DSP addresses $007F_H$ to $3FFF_H$.
- Assert or release the DSP reset.
- Assert the DSPInt interrupt (DSP interrupt No25).

- **Level 2**: At level 2, the kernel is loaded and running on the DSP. Through intrinsic functions of the kernel, the PC can access any location in any memory space of the DSP, and can launch DSP code from an entry point anywhere in memory. This level is used to load user code in DSP memory, and launch it. Level 1 functions are still functional at level 2.

At level 2, the PC has access to low-level functions to perform the following operations:

- Read or write values to/from anywhere in the memory space of the DSP.
- Launch execution of code from any entry point in the program memory of the DSP.

- **Level 3:** Level 3 is defined when user code is loaded and running on the DSP. There is no functional difference between levels 2 and 3. The level 1 and 2 kernel functions are still available to support the exchange of data between the PC and the DSP, and to direct execution of the user DSP code. The main difference is that at level 3, user functions are available too, in addition to intrinsic functions of the kernel.

 © BRUNO PAILLARD ING.

3.2. Resources used by the kernel on the DSP side

To function properly, the kernel uses the following resources on the DSP. After the user code is launched, those resources should not be used or modified, in order to avoid interfering with the operation of the kernel, and retain its full functionality.

- The kernel resides from address 0100_H to address $02FF_H$ in the on-chip RAM of the DSP. The user should avoid loading code into, or modifying memory space below address 0300_h.
- A “Mail Box Area” is used from address 1000_H to address 1022_H in the on-chip RAM of the DSP, to support communications with the PC.
- The PC (via the USB controller) uses the DSPInt interrupt (DSP interrupt No 25) from the HPI to request an access to the DSP. If necessary, the user code can temporarily disable this interrupt through its mask in the IMR register, or the global interrupt mask, in register ST1. During the time this interrupt is disabled, all PC access requests are latched, but are held until the interrupt is re-enabled. Once the interrupt is re-enabled, the access request resumes normally.
- The kernel locates the interrupt vector table at address 0080_H. If the user code needs to relocate the interrupt vector table, the vector for interrupt No 25 should be initialized (with a jump to address 0112_H), prior to the relocation. This insures that interrupt No 25 (DSPInt from the HPI) will continue to be functional during and after the relocation.
- The on-chip dual access RAM is the only space where code can be executed from on the DSP board. For the kernel to function properly, and to be able to download and execute user code from the on-chip RAM of the ‘C5402, it must be defined as Program and Data space (the bit OVLY of the PMST register must be set to 1).
- The communication kernel initializes the stack pointer at the end of the DSP on-chip RAM (address $3FFF_H$), and places the return address to the kernel on top of the stack when the first user function is launched. The stack pointer is therefore at address $3FFE_H$ when the user code starts executing. The user code can relocate the stack pointer temporarily, but should replace it, or copy the return address found in $3FFF_H$ on top of the new relocated stack, before the last return to the kernel (if this last return is intended). Examples where a last return to the kernel is not intended include situations where the user code is a never-ending loop that will be terminated by a board reset or a power shutdown. In this case, the stack can be relocated without concern.

3.3. Functional description of the kernel

After taking the DSP out of reset, the user-mode PC application loads the kernel into the DSP on-chip RAM at address 1000_H, together with a small relocation seed. After loading the kernel and relocation seed, the starting address of the relocation seed (1000_H) is written to address $007F_H$, which signals the DSP to start executing from this location. The kernel immediately begins relocating itself between addresses 0100_H and $02FF_H$. Once the relocation is completed the kernel starts executing, and performs the following initializations:

- Locates the interrupt vector table (IPTR register) at address 0080_H.
- Initializes the stack pointer at address $3FFF_H$.

- Configures the on-chip RAM as Program and Data (sets the OVLY bit of the PMST register).
- Sets the number of wait-states for the external RAM to 1.
- Unmasks interrupt No 25 (DSPInt from the HPI).
- Unmasks interrupts globally.

The relocation of the kernel and its execution after address $007F_H$ is written, occur automatically and take 5µs. The LabVIEW VI that loads the kernel waits for at least this amount of time, so that when the first command is sent from the LabVIEW environment after loading the kernel, the kernel is already ready to respond.

After all the initializations, the kernel is simply a never-ending loop which waits for the next access request from the PC, triggered by the DSPInt interrupt from the HPI.

3.3.1. Launching a DSP function

At levels 2 and 3, the kernel protocol defines only one type of action, which is used to read and write DSP memory, as well as to launch a simple function (a function which includes a return to the kernel or to the previously running code) or a complete program (a never-ending function which is not intended to return to the kernel or to the previously running code). In fact, a data space memory read or write is carried out by launching a ReadMem or WriteMem function, which belongs to the kernel (intrinsic function) and is resident in DSP memory between addresses 0120_H and 0150_H. Launching a user function uses the same basic principle, but requires that the user function be loaded in DSP memory prior to the execution.

The mailbox is an area in the HPI RAM (addresses 1000_H to 1022_H). The function of each address in the mailbox is described below.

Address	Function
1000_h to $101F_h$	32 data words used for transfers between the PC and the DSP.
1020_h	Branch address (intrinsic read and write functions, or user function)
1021_h	Number of words to transfer
1022_h	Transfer address

Table 10-1: Mail box

To launch a DSP function (intrinsic or user code), the PC (via the USB controller) does the following operations:

Initiates a K_Data_Move or K_Exec vendor request. This request contains information about the DSP address of the function to execute (user or intrinsic). For K_Data_Move, it also contains information about the direction of the transfer (Read or Write); and in the case of a Write transfer, it contains the data words to be written to the DSP in the data stage of the request.

- If data words are to be written <u>to</u> the DSP (K_Data_Move in the Write direction), the USB controller places these words in addresses 1000_H to $101F_H$ of the mailbox.
- If words are to be transferred <u>to or from</u> the DSP (K_Data_Move in the Read or Write direction), places the number of words to be transferred (must be between 1 and 32) in address 1021_H of the mailbox.

- If words are to be transferred to or from the DSP (K_Data_Move in the Read or Write direction), places the DSP transfer address at address 1022_H of the mailbox.
- Clears the HINT (host interrupt) signal, which serves as the DSP function acknowledge.
- Sends a DSPInt interrupt to the DSP, which forces the DSP to branch to the intrinsic or user function.

When the DSP receives the DSPInt interrupt from the HPI, the kernel does the following:

If the DSP is not interruptible (because interrupt No25 is temporarily masked, or because the DSP is already serving another interrupt), interrupt No 25 is latched until the DSP becomes interruptible again, at which time it will serve the PC access request.

If - or when - the DSP is interruptible, it:

- Fetches the branch address at address 1020_H in the mailbox, and pushes it on top of the stack.
- Executes a return from interrupt (RETE instruction), which causes execution to jump to the desired branch address and re-enabling of the interrupts at the same time.
- From this instant, the DSP becomes interruptible again, and can serve a critical user Interrupt Service Routine (ISR). If no user ISR is pending, the DSP starts executing the function requested by the PC (intrinsic or user function).
- If data words are to be transferred from the PC (K_Data_Move in the Write direction), the intrinsic function reads those words from mailbox addresses 1000_H to $101F_H$ and places them at the required DSP address.
- If data words are to be transferred to the PC (K_Data_Move in the Read direction), these words are written by the DSP to addresses 1000_H to $101F_H$ in the mailbox.
- After the execution of the requested function, the DSP asserts the HINT signal, to signal the USB controller that the operation has completed. This operation has been conveniently defined in a macro "**Acknowledge**" in the example codes, and can be inserted at the end of any user function.
- If data words are to be transferred to the PC, upon reception of the HINT signal, the USB controller fetches those words from addresses 1000_H to $101F_H$ in the mailbox area and sends them to the PC in the data stage of the request (K_Data_Move in the Read direction).
- The assertion of the HINT signal by the DSP also causes the status phase of the vendor request that initiated the access (K_Data_Move or K_Exec) to be ACKed by the USB controller, therefore signaling completion to the PC. If the DSP fails to assert the HINT signal within 5s, the USB controller times-out, and stalls the vendor request that initiated the operation. In this case, the request completes with an error on the PC side. Note that the time-out can be disabled.
- Once the DSP function has completed its execution, a return (or delayed return) instruction (RET or RETD) can be used to return control to the kernel, or the previously executing user code, after the acknowledge. This is not necessary however, and user code can keep executing without any return to the kernel if

this is what the user intends. In this case, subsequent PC accesses are still allowed, which means the kernel is re-entrant.

Since a PC access is requested through the use of the DSPInt interrupt from the HPI, it can be obtained even while user code is already executing. Therefore it is not necessary to return to the kernel to be able to launch a new DSP function. This way, user functions can be re-entered. Usually, the kernel will be used at level 2 to download and launch a user main program, which may or may not return to the kernel in the end. While this program is being executed, the same process described above can be used at level 3 to read or write DSP memory locations, or to force the execution of other user DSP functions, which themselves may, or may not return to the main user code, and so on… It is entirely the decision of the developer. If a return instruction (RET or RETD) is used at the end of the user function, execution is returned to the code that was executing prior to the request. This code may be the kernel at level 2, or user code at level 3.

The acknowledgment of the completion of the DSP function (intrinsic or user code) is done through the assertion of the HINT signal. This operation is encapsulated into a macro for the benefit of the developer. This acknowledge operation is done for the sole purpose of signaling to the initiating vendor request that the requested DSP function has been completed, and that execution can resume on the PC side. Reception by the USB controller of the HINT signal directs it to ACK the request. Normally, for a simple user function, which includes a return (to the main user code or to the kernel), this acknowledge is placed at the end of the user function (just before the return) to indicate that the function has completed. As a matter of "good programming", the developer should not implement DSP functions that take a long time before returning an acknowledge. In the case of a function that may not return to the kernel or to previously executing user code, or in any case when the user does not want the vendor request that initiated the access to wait until the end of the DSP function, the acknowledge can be placed at the beginning of the user function. In this case it signals only that the branch has been taken. Another method of signaling the completion of the DSP function must then be used. For instance the PC can poll a completion flag in DSP memory.

During the PC access request, the DSP is only un-interruptible during a very short period of time (between the taking of the DSPInt interrupt and the branch to the beginning of the user or intrinsic function – the equivalent of 10 cycles). Therefore, the PC access process does not block critical tasks that might be executing under interrupts on the DSP (managing analog I/Os for instance).

At the beginning of a DSP user function, it may be wise to protect (store) all the DSP registers that are used within the function, and to restore them just before the return. Since the function is launched through the DSPInt interrupt, it is done asynchronously from the main executing code (kernel or user code). Responsibility of register protection must then be assumed by the called function (same as with any Interrupt Service Routine, which it actually is!). All the intrinsic functions of the kernel perform this protection and will not interfere with user code.

3.3.2. Memory read and write

The kernel includes 6 intrinsic functions (ReadMem, WriteMem, ReadProg, WriteProg, ReadIO and WriteIO), which are part of it, and are resident in memory at the time the kernel is executing. These 6 functions allow the PC to read or write the DSP memory.

 © BRUNO PAILLARD ING.

3.3.2.1.ReadMem, ReadProg, ReadIO :

These functions read *nn* successive words from the DSP memory at address *aaaa* (1<=*nn*<=32).

To launch the execution of any of these functions, the PC (via the USB controller and the invocation of K_Data_Move in the Read direction) does the following:

- Writes the entry point of the function (0120_H, 0180_H or $01E0_H$ for ReadMem, ReadProg or ReadIO resp.) into address 1020_H in the mailbox.
- Writes the number *nn* of words to transfer, into address 1021_H in the mailbox.
- Writes the DSP transfer address *aaaa*, into address 1022_H in the mailbox.
- Clears the *HINT* signal.
- Sends a *DSPInt* interrupt to the DSP.

In response to these actions, the kernel does the following:

- Branches to the beginning of the function and becomes interruptible again.
- Pushes all the registers used within the function on the top of the stack (context protection).
- Reads the beginning transfer address *aaaa* in location 1022_H
- Reads *nn* words in the DSP memory and writes them into addresses 1000_H to $101F_H$ of the mailbox.
- Places the address directly following the block that has just been transferred into address 1022_H in the mailbox. This way subsequent accesses do not have to reinitialize the address, to begin transferring the following words.
- Restores the context.
- Asserts the *HINT* signal, which signals the completion of the operation.
- Returns to the caller.

At this point, the USB controller does the following:

- Reads the *nn* words from address 1000_H in the HPI RAM and sends them to the PC as part of the data block of the request.
- If it is the last block, ACKs the status stage of the request, which completes the transfer on the PC.

The default control pipe 0 that supports the exchange has a size of 64 bytes (32 words). For transfers larger than 32 words, the data stage of the request is broken down into blocks of 64 bytes. The USB controller invokes the function and carries out the above operations for each of the 64-byte blocks that form the data stage. Assertion of the *HINT* signal at the end of each block triggers the transfer of the block to the PC. The status stage of the request is ACKed at the end of the very last block.

Because of current limitations in Windows 98, the size of the transfer cannot be larger than 4096 bytes (2048 words). Transfers larger than this limit have to be broken down into blocks of 2048 words, with one K_Data_Move vendor request for each of these blocks.

© BRUNO PAILLARD

***Note**: The actual size of the transfer requested by K_Data_Move is expressed in number of bytes in the set-up stage of the request. Since it is 16-bit words that are actually exchanged with the PC, the number of bytes indicated in the set-up stage must always be even.*

***Note**: The number of words to read must never be zero.*

3.3.2.2.WriteMem, WriteProg, WriteIO

These functions write *nn* successive words into the DSP memory, from address *aaaa* (1<=*nn*<=32).

To launch the execution of any of these functions, the PC (via the USB controller and the invocation of K_Data_Move in the Write direction) does the following:

- Places the *nn* words to write to DSP memory at addresses 1000_H to $101F_H$ of the mailbox.
- Writes the entry point of the function (0150_H, $01B0_H$ or 0210_H for WriteMem, WriteProg or WriteIO resp.) into address 1020_H in the mailbox.
- Writes the number *nn* of words to transfer, into address 1021_H in the mailbox.
- Writes the DSP transfer address (*aaaa*), at address 1022_H in the mailbox.
- Clears the *HINT* signal.
- Sends a *DSPInt* interrupt to the DSP.

In response to these actions, the kernel does the following:

- Branches to the beginning of the function and becomes interruptible again.
- Pushes all the registers used within the function on the top of the stack (context protection).
- Reads the transfer address at address 1022_H
- Reads the *nn* words from the addresses 1000_H to $101F_H$ of the mailbox and writes them back to the DSP memory at the *aaaa* transfer address.
- Places the address directly following the block that has just been transferred into address 1022_H in the mailbox. This way subsequent accesses do not have to reinitialize the address, to begin transferring the following words.
- Restores the context.
- Asserts the *HINT* signal, which signals the completion of the operation.
- Returns to the caller.

At this point, the USB controller does the following:

- Acks the block so the PC can send the next one.
- If it is the last block, ACKs the status stage of the request, which completes the transfer on the PC.

© BRUNO PAILLARD ING.

The default control pipe 0 that supports the exchange has a size of 64 bytes (32 16-bit words). For transfers larger than 32 words, the data stage of the request is broken down into blocks of 64 bytes. The USB controller invokes the function and carries out the above operations for each of the 64-byte blocks that form the data stage. Assertion of the *HINT* signal at the end of each block triggers the transfer of the next block from the PC. The status stage of the request is ACKed at the end of the very last block.

Because of current limitations in Windows 98, the size of the transfer cannot be larger than 4096 bytes (2048 words). Transfers larger than this limit have to be broken down into blocks of 2048 words, with one K_Data_Move vendor request for each of these blocks.

***Note**: The actual size of the transfer requested by K_Data_Move is expressed in number of bytes in the set-up stage of the request. Since it is 16-bit words that are actually exchanged with the PC, the number of bytes indicated in the set-up stage must always be even.*

***Note**: The number of words to read must never be zero.*

3.3.2.3.Use of the K_Data_Move request for user functions

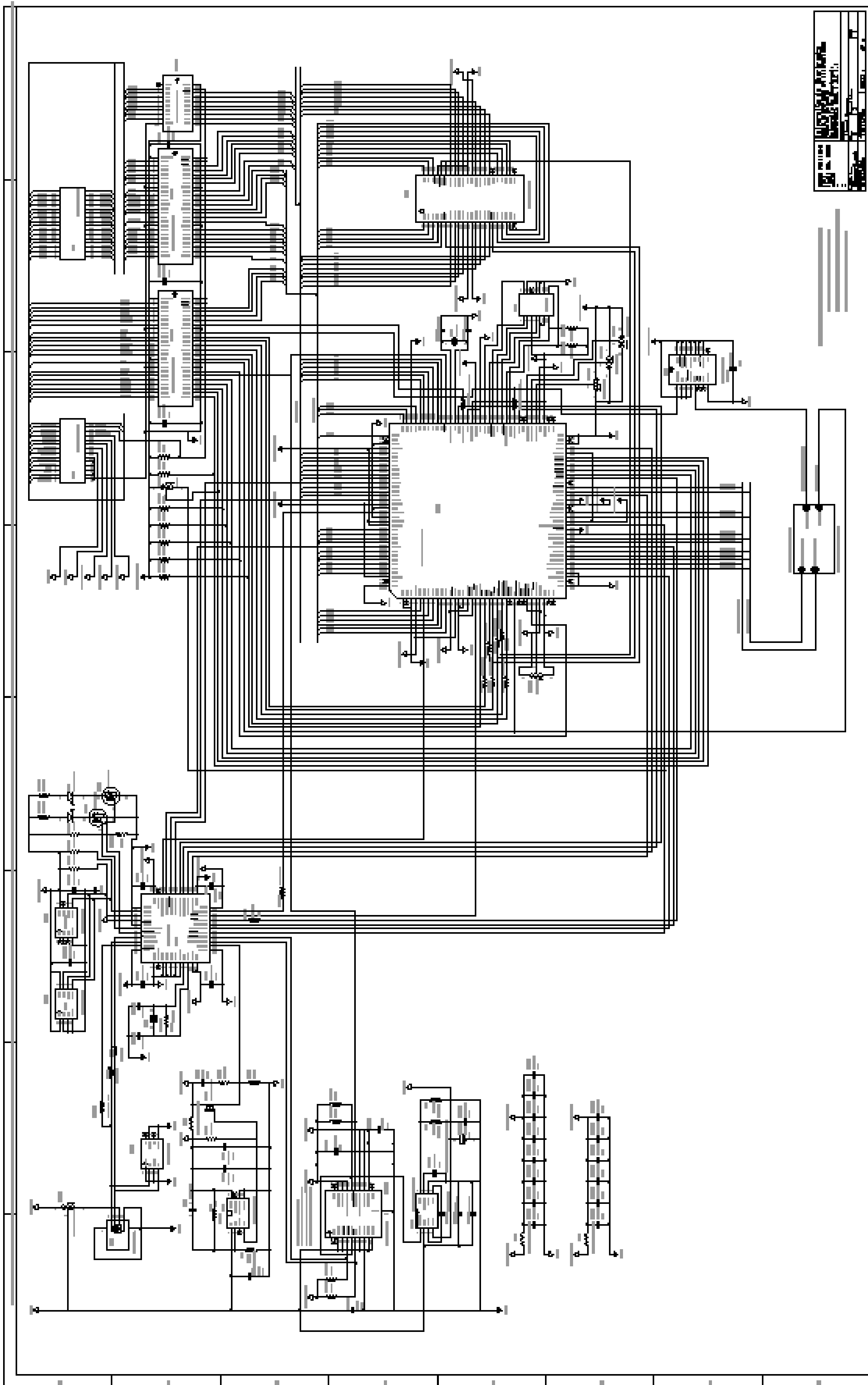
In principle, the K_Data_Move request is used only to invoke the 6 intrinsic kernel functions. However, nothing bars the developer from using this request to invoke a user function. This may be advantageous for functions that have to exchange data (arguments) with the PC. To achieve this, the user function should behave in exactly the same manner as the *intrinsic* functions do for Read resp. Write transfers. The function address specified in the set-up stage of the request should then be the user function entry point, instead of the entry point of an intrinsic function.

It should be noted that arguments, data, and parameters can alternately be passed to/from the PC by the K_Data_Move request after or before the invocation of any user function. With the use of synchronizing structures (semaphores), such exchanges can easily be organized at a higher level so that they represent arguments being passed to/from the user function.

The absence of protocol at a higher level gives the developer freedom in defining ways to pass arguments to user functions.

Appendix A – Schematics Of The Signal Ranger DSP Board

© BRUNO PAILLARD ING.



© BRUNO PAILLARD ING.