Section 17

ADSP-BF533 VisualDSP++
C/C++ Compiler

Strategic Objective:
Make C as fast as assembler!

Advantages:

C is much cheaper to develop.

C is much cheaper to maintain.

C is comparatively portable.

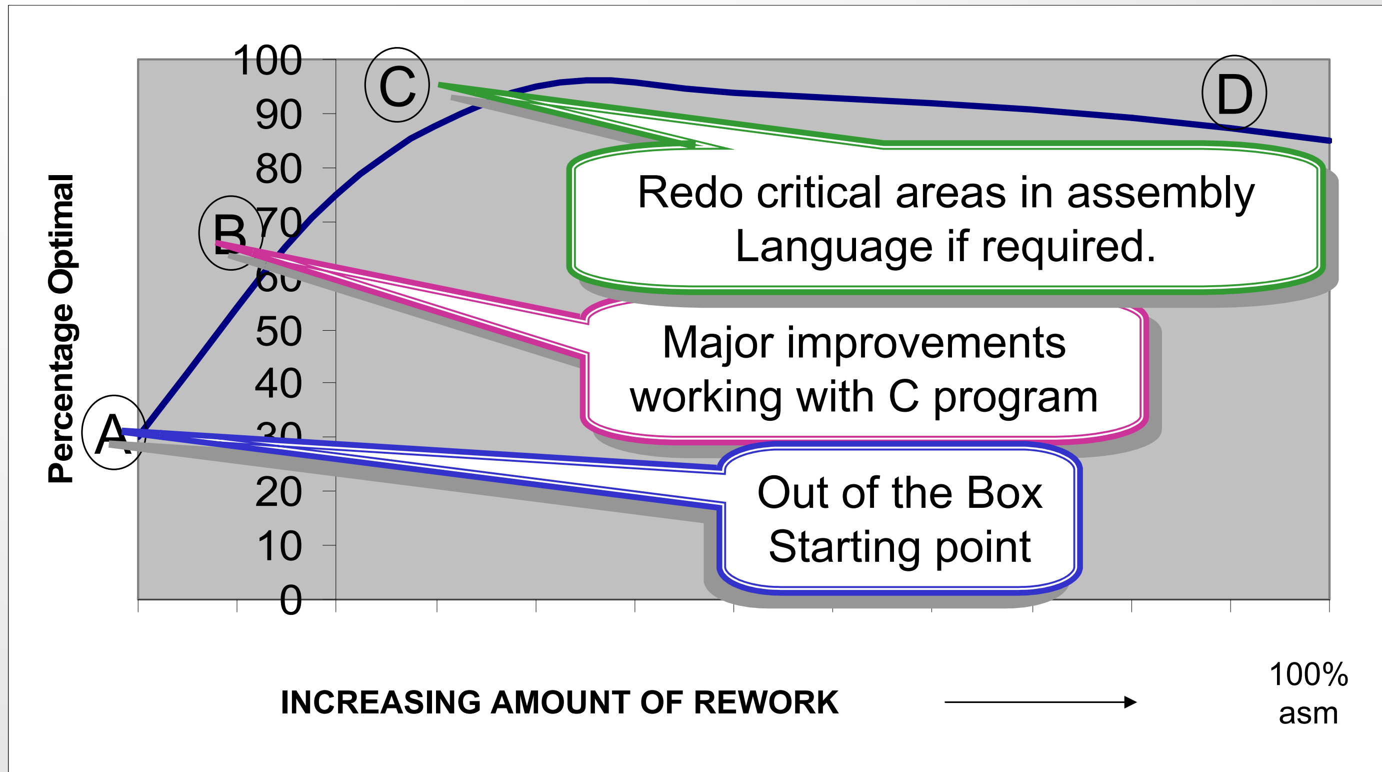
- **Disadvantages:**

ANSI C is not designed for DSP.

DSP processor designs usually expect assembly in key areas.

DSP applications continue to evolve.

The Performance Curve



KAZTEK ENGINEERING

ANALOG DEVICES

Pillars of Effective Programming

- Understand Underlying Hardware Capabilities
- Discover What Compiler Can Provide
- Design Program Effectively
  - general choice of algorithm
  - choice of data representation
  - finer low-level programming decisions

- Usually the process of performance tuning is a specialisation of the program for particular hardware.   It may grow larger or more complex and is less portable.

C Compiler (VDSP++ 4.0)

- State-of-the-art optimizer.
  - Provides flexibility
  - Ease of adding architecture-specific optimizations
- Exploitation of explicit parallelism in the architecture
  - Vectorization – exploiting wide load capabilities
  - Recognizing SIMD opportunities
  - Software pipelining
- Whole Program Analysis
  - A wider view enables the optimizer to be more aggressive.

Other features with VDSP 4.0

- long long support - 64-bit integer support
- Enhanced GNU compatibility features.
- compiler built-ins added for Blackfin video operations.
- ADSP-BF561 support
- multiple-heap support
- improved cache support
- C++ Exception Handling
- Profile-Guided Optimization
- Software emulated 64 bit integers.

- 64-bit IEEE floating-point support - long double

Emulated support with hand coded compiler support routines will be added in a future release

Understanding Underlying Hardware

- **Isn't C supposed to be portable & machine independent?**
  - **yes, but at a price!**
  - **Uniform computational model, BUT….**
    - **missing operations provided by software emulation (slow)**
    - **for example: C provides floating point arithmetic everywhere**
  - **C is more machine-dependent than you might think**
    - **for example: is a "short" 16 or 32 bits? (more later)**
- **Machine's Characteristics will determine your success.**

C programs can be ported with little difficulty.

But if you want high efficiency, you can't ignore the underlying hardware

KAZTEK ENGINEERING

ANALOG DEVICES

Evaluate Algorithm against Hardware.

- **What's the native arithmetic support?**
  - Can we use floating point hardware?
  - how wide is the integer arithmetic?
    - doing 64-bit arithmetic on a 32-bit unit is slow
    - doing 16-bit arithmetic on a 32 bit part is awkward
  - Can we use packed data operations?
    - 2x16 arithmetic might be ideal for your application (more computation per cycle, less memory usage)
    - implications for data types, memory layout, algorithms
- **What is the computational bandwidth and throughput?**
  - what are the key operations required by your algorithm?
  - (macs?, loads?, stores?….)
  - how fast can the computer perform them?

Signal Processing Unique Challenges

- **Special Aspects of Digital Signal Processors:**
  - Reduced memory
  - Extended precision accumulators
  - Specialized architectural features
    If not well modeled by C : lose portability and efficiency
    - Example: Zero overhead loop – good
    - Fractional arithmetic - problem.
  - mathematical focus (historically not C's orientation)

- **Features which compiler must exploit**
  - Efficient Load / Store Operations in Parallel
  - Utilize multiple Data-paths; SISD, SIMD, MIMD operations
  - minimize memory utilization

C and the Compiler

- C provides common computational model
  - portability
  - higher level
- Compiler's job: map this to a particular machine
  - tries for optimal use of instructions
  - supplement by instruction sequences or library calls
- Optimizer improves performance
  - do things less often, more cheaply
  - try to utilize resources fully
- Optimizing Compiler has Limited Scope
  - will not make global changes
  - will not substitute a different algorithm
  - will not significantly rearrange data or use different types
  - correctness as defined in the language is the priority

Example C Program

// Simple dot product example

```
extern short* x;
extern short* y;

short dot (void)
{
    short s = 0;
    int j;

    for (j=0; j<1024; j++)
    {
            s += x[j]*y[j];
    }
    return s;
}
```

KAZTEK ENGINEERING

ANALOG DEVICES

Compiler Produced Assembly Code (.s File)

```
        .section program;
        .align 2;
_dot:
.LN1:
                P0.L = _x;
                P1.L = _y;
                P0.H = _x;
                P1.H = _y;
                P0=[P0+ 0];
                P1=[P1+ 0];
                R2 =   3;
                link  0;
//              -- 3 bubbles --
                R0 = P0 ;
                R1 = P1 ;
                R0 = R0 | R1;
                R0 = R0 & R2;
                CC =  R0 ==  0;
                IF !CC JUMP  ._P1L2 ;
                ...............................................
                I0 = P0 ;
.LN2:
                P2 =  511 (X);
                A1=A0=0 || R1 = [P1++] || R0 = [I0++];
                LSETUP (._P1L4 , ._P1L5-8) LC0=P2;
.align 8;
._P1L4:
.LN3:
                A1+= R1.H*R0.H, A0+= R1.L*R0.L (IS) || R1 = [P1++] || R0 = [I0++];
.LN4:
                // end loop ._P1L4;
._P1L5:
.LN5:
                A1+= R1.H*R0.H, A0+= R1.L*R0.L (IS) || P0=[FP+ 4] || NOP;
```

Annotations:

- (P0.L … P1.H lines) Load address of x and y pointers into P1 and P0, respectively
- (P0=[P0+ 0]; P1=[P1+ 0];) Load pointers to x and y pointers into P1 and P0
- (R0 = P0 … CC = R0 == 0;) Check that pointers to x and y are on quad aligned boundaries
- (IF !CC JUMP ._P1L2 ;) → If not, jump to ._P1L1
- Otherwise, fetch and perform operations on 2x16 bit words at a time

KAZTEK ENGINEERING

ANALOG DEVICES

Compiler Produced Assembly Code (.s File)

```
.LN6:
                A0+=A1;
.LN7:
                R0 = A0.w;
.LN8:
                R0 = R0.L (X);
                unlink;
//              -- 2 bubbles --
                JUMP (P0);
```

Complete SIMD dot product and return

```
._P1L2:
                I0 = P0 ;
                P2 =  1023 (X);
                A0 = 0 || R0 = W[P1++] (X) || R1.L = W[I0++];
                LSETUP (._P1L8 , ._P1L9-8) LC0=P2;
.align 8;
._P1L8:
.LN9:
                A0 += R0.L*R1.L (IS) || R0 = W[P1++] (X) || R1.L = W[I0++];
.LN10:
                // end loop ._P1L8;

._P1L9:
.LN11:
                A0 += R0.L*R1.L (IS) || P0=[FP+ 4] || NOP;
                R0 = A0.w;
.LN12:
                R0 = R0.L (X);
                unlink;
//              -- 2 bubbles --
                JUMP (P0);
```

Perform non-SIMD fetch and operations on non-quad aligned data

KAZTEK ENGINEERING

ANALOG DEVICES

C++

- C++ Programs can have high efficiency
  - depends which features are used: pay as you go
- “Same as C” runs at same efficiency
- Overloaded functions, namespaces: no cost
- Classes for modularity / new data types:
  - no inherent cost
  - pointer-based data will be slower (also aliasing problems)
  - templates not inherently slower
- Inheritance: no cost
- Virtual functions: slight cost

- C++ capability is great for porting control code or expert programming,
- But the greater capability to abstract leads to programs are harder to tune and often have hidden or unexpected performance problems.

Summary:
How to go about increasing performance.

1. Work at high level first

   most effective -- maintains portability

   – improve algorithm
   – make sure it's suited to hardware architecture
   – check on generality and aliasing problems

2. Look at machine capabilities

   – may have specialized instructions (library/portable)
   – check handling of DSP-specific demands

3. Non-portable changes last

   – in C?
   – in assembly language?
   – always make sure simple C models exist for verification.

- Compiler will improve with each release

ADSP-BF533 C/C++ Compiler

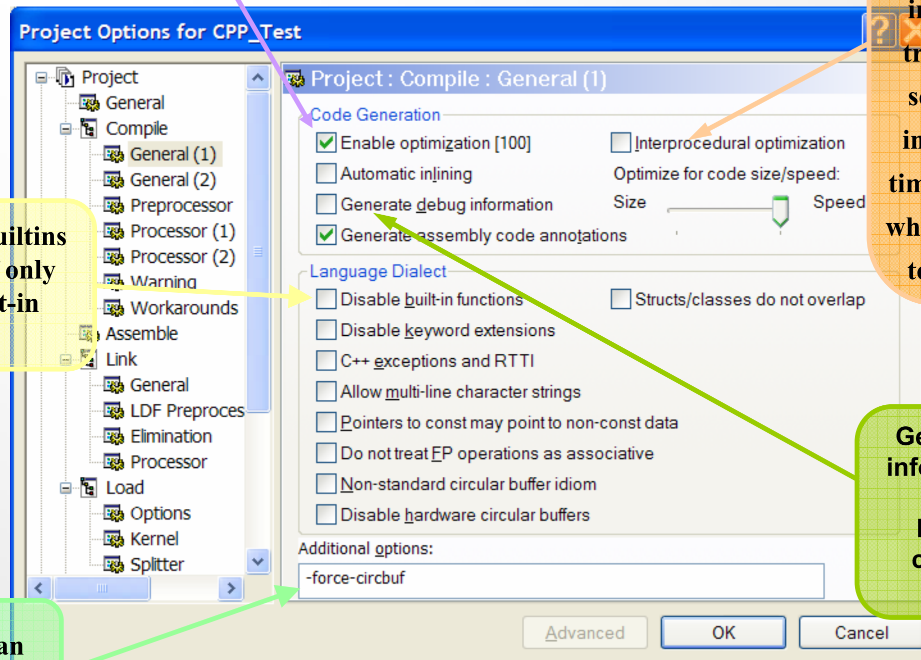
- Compiler
  - Invoked Via IDDE Using Settings from Compiler Property Page
  - Invoked from a DOS Command Line (ccblkfn.exe)
- Linker Description File (LDF)
  - Defines Segments in Memory for Code and Data
  - Defines Segment in Memory for the Stack
  - Defines Segment in Memory for the Heap
- Run Time Header
  - Run Time Header created by startup wizard when project is created
  - Linker Options Determine Which C Run-Time Libraries To Use
    - Size, File I/O, C++ Are All Selectable
  - Provides Interrupt Handling
  - Initializes C/C++ Run-Time Environment
  - Must Be Linked With C/C++ Code
    - Done by LDF

Compile / General Property Page

Corresponds to –O compiler switch*. Optimizes source code for better performance.

Allows compiler to optimize across translation units instead of within individual translation units. Compiler sees all the source files used in a final link at compilation time and uses that information while optimizing. Corresponds to the –ipa compiler switch.

Corresponds to –no-builtins switch. Allows use of only ANSI-standard built-in functions.



Generates DWARF-2 debug information. Allows users to debug projects and set breakpoints in C source code. Corresponds to –g switch*.

Any compiler switch can be specified here

** - Using '–O –g' gives preference to optimization. Using '-Og' gives preference to debug.*

KAZTEK ENGINEERING

ANALOG DEVICES

Supported Data Formats

Type	Bit Size	Number Representation	sizeof returns
char	8 bits signed	8-bit two's complement	1
unsigned char	8 bits unsigned	8-bit unsigned magnitude	1
short	16 bits signed	16-bit two's complement	2
unsigned short	16 bits unsigned	16-bit unsigned magnitude	2
int	32 bits signed	32-bit two's complement	4
unsigned int	32 bits unsigned	32-bit unsigned magnitude	4
long	32 bits signed	32-bit two's complement	4
unsigned long	32 bits unsigned	32-bit unsigned magnitude	4
long long	64 bits signed	64-bit two's complement	8
unsigned long long	64 bits unsigned	64-bit unsigned magnitude	8
pointer	32 bits	32-bit two's complement	4
function pointer	32 bits	32-bit two's complement	4
double	32 bits	32-bit IEEE single-precision	4
float	32 bits	32-bit IEEE single-precision	4
double	64 bits	64-bit IEEE double-precision	8
long double	64 bits	64-bit IEEE	8
fract16	16 bits signed	1.15 fract	2
fract32	32 bits signed	1.31 fract	4

Linker Description File for C/C++ Programming

- Memory Description
  - Define Memory Segments
  - Map Input Sections (Names Produced by Compiler) to Memory Segments
- Run Time Stack Supported
  - Stack Used for Branching, Local Variables, Arguments
  - LDF Defines Stack Size and Location
- Run Time Heap Supported
  - Used For Memory Management Protocols (malloc, free, etc)
  - LDF Defines Heap Size, Location, and Name (For Multiple Heap Support)

Compiler-Generated Memory Section Names

- **Compiler uses default section names that are mapped appropriately by the linker (through the LDF)**

  - **program** - contains all program instructions
  - **data1** - contains all global and "static" data
  - **constdata** - contains all data declared as "const"
  - **ctor** - C++ constructor initializations
  - **cplb_code** – code CPLB config tables
  - **cplb_data** – data CPLB config tables

Memory Descriptions

- Define Memory Segments In LDF For:
  - Code, Data, Stack*, Heap(s)
- Map Input Sections to Memory Segments
  (BF533 Default LDF Segment Names Used)

Segment Name	Use
– MEM_L1_CODE	code storage
– MEM_L1_CODE_CACHE	code storage, if not cache
– MEM_L1_DATA_A	used for default compiler data sections
– MEM_L1_DATA_A_CACHE	If not used as cache, it becomes heap space
– MEM_L1_DATA_B	used for default compiler data sections
– MEM_L1_DATA_B_CACHE	If not used as cache, it is used for data
– MEM_L1_DATA_B_STACK	dedicated stack space
– MEM_L1_SCRATCH	Dedicated 4 Kbyte Data Scratchpad
– MEM_ARGV	Optional Command Line Parsing (256 Bytes)
– MEM_SDRAM0_HEAP	If L1 Data A used as cache, heap is external
– MEM_SDRAM0	external SDRAM bank
– MEM_ASYNCx	(x=0,1,2,3) 1MB Async Banks

Software Build Process

Step 1 Example: C Source with Alternate Sections

foo.C

```
section ("extern") int array[256];

section ("foo") void bar(void)
{
   int foovar;
   foovar = 1;
   foovar++;
}
```

C-Compiler → foo.S → Assembler →

foo.DOJ

```
Object Section = extern
Type           = RAM
Width          = 8
.......................
_array [0]
_array [1]
  …
_array [255]
```

```
Object Section = foo
Type           = RAM
Width          = 8
.......................
 _bar : p0=_foovar;
 r0=w[p0]; r0=r0+1;
 w[p0] = r0;
```

```
Object Section = mem_stack
Type           = RAM
Width          = 8
.......................
 _foovar: 1
```

Note: The section( ) directive is used to place data or code into a section other than the default section used by the compiler.

Run Time Stack

- 32-Bit Wide Structure Growing in Memory from Higher to Lower Addresses

- Managed by a Frame Pointer, FP, and a Stack Pointer, SP
  - FP Points to Address of Beginning of Frame (Contains Previous Frame Address)
  - SP Points to Last Entry on Stack

- Stack Frame Contains:
  - Local Variables
  - Temporary Variables
  - Function Arguments

LDF and the Stack

- **C/C++ Runtime Environment Depends Upon the Initialization of FP and SP**

- **Variables Initialized by Constants Defined in the LDF**
  - **ldf_stack_space**
  - **ldf_stack_end**

- **Variables Used to Initialize FP and SP are Declared and Initialized in the Assembly File basiccrt.s**

LDF Stack Setup
(C/C++ Compiler Only)

- **Linker Calculates LDF Stack-Initializing Constants from the Stack Memory Segment Description**

```
stack
    {
        ldf_stack_space = .;
        ldf_stack_end = ldf_stack_space + MEMORY_SIZEOF(MEM_L1_DATA_B_STACK);
    } >MEM_L1_DATA_B_STACK
```

When Programming In C/C++, This Segment Must be Included in the SECTIONS() Portion of the LDF

LDF and the Heap

- **Four Library Functions Can Be Used to Allocate or Free Memory to/from the Heap**
  - **malloc, calloc, realloc, free**
- **Other C Library Functions Implicitly Use these Four Functions and ALSO Require the Heap**
  - **memmove, memcopy, etc.**
- **Initialized by Constants Defined in the LDF**
  - **ldf_heap_space**
  - **ldf_heap_length**
  - **ldf_heap_end**
- **Multiple Heaps are Possible**
  - **Can be defined at Link Time or at Run Time (see compiler manual)**

KAZTEK ENGINEERING

ANALOG DEVICES

LDF Heap Setup
(C Compiler Only)

- **Output Section 'heap' Calculates LDF Heap Initializers from Heap Memory Segment Description**

```
#ifdef USE_CACHE /* { */
	heap
	{
	   // Allocate a heap for the application
	   ldf_heap_space = .;
	   ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_SDRAM0_HEAP) - 1;
	   ldf_heap_length = ldf_heap_end - ldf_heap_space;
	} >MEM_SDRAM0_HEAP
#else
	heap
	{
	   // Allocate a heap for the application
	   ldf_heap_space = .;
	   ldf_heap_end = ldf_heap_space + MEMORY_SIZEOF(MEM_L1_DATA_A_CACHE) - 1;
	   ldf_heap_length = ldf_heap_end - ldf_heap_space;
	} >MEM_L1_DATA_A_CACHE
#endif /* USE_CACHE } */
```

- **When Programming In C, This Section Must be Included in the Sections Portion of the LDF**
- **Must Duplicate this Code for Each Defined Heap**

C Run Time Headers

- Sets Up the C Runtime Environment
  - Resets Registers and Initializes Global Data
  - Initializes Event Vector Table
    - Installs IVG15 vector (lowest priority)
  - Enables Interrupts
    - Only IVG15 is enabled
  - Sets up stack pointer, enables cycle counters
  - Allows processor to come up supervisor mode
  - Initializes File I/O support, if necessary
  - Configures Cache, if necessary
  - Initializes profiling support, if necessary
  - Initializes multi-thread support, if necessary
  - Initializes global C++ objects and sets up destructor calls for clean-up
  - Initializes argc/argv support, if necessary
  - Calls _main to start the actual program
  - Calls _exit when program terminates
- Configured by Startup Wizard with a new project
  - Can be modified later through project options window

Implementing Interrupts In C On BF533

- Use Direct Event Vector Table (EVT) Management Functions
  - EX_INTERRUPT_HANDLER (ISR_Name)
    - Inserts context save/restore code in ISR_Name's prologue/epilogue
    - Appends "RTI;" to return from interrupt
  - register_handler (sig_num, ISR_Name)
    - Maps ISR_Name's address into EVTx register indicated by sig_num
    - Sets appropriate IMASK bit (indicated by sig_num) and enables interrupts
- Use Interrupt Dispatcher
  - interrupt(sig_num, ISR_Name)
    - Places ISR_Name's address into internal look-up table using sig_num as the index into the table
    - Executes implicit call to register_handler(sig_num, _despint)
      - Maps Dispatcher's address to EVTx register associated with sig_num
      - Sets associated IVGx bit in IMASK
    - When Interrupt Occurs, Dispatcher
      - Does full context save/restore
      - Polls IPEND register to determine which interrupt occurred
      - Uses look-up table to determine ISR vector location

Direct EVT Management Functions

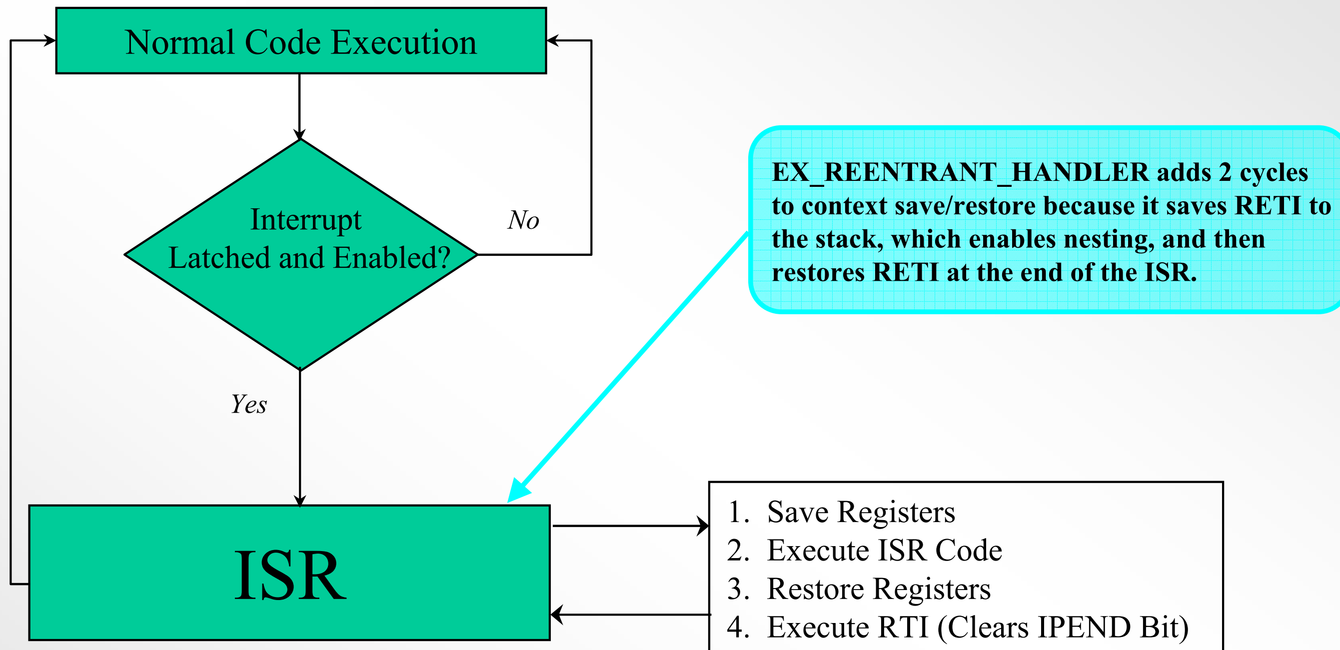
- **EX_INTERRUPT_HANDLER( ) and register_handler( ) Functions**

  Usage:

```
#include<sys\exception.h>
EX_INTERRUPT_HANDLER(ISR_Name);
register_handler (ik_ivg11, ISR_Name);
```

- **EX_INTERRUPT_HANDLER (ISR_Name);**
  - SAVES current processor state after entry into ISR_Name module
  - RESTORES former processor state before exit from ISR_Name module
    - 72 cycles to save/restore processor context and perform stack maintenance
      - All Data (R0-R7) and Pointer (P0-P5) Registers
      - Frame Pointer (FP) and Arithmetic Status Register (ASTAT)
      - RETI is NOT part of the context save so interrupt nesting is OFF!!!
        - To nest, use EX_REENTRANT_HANDLER (ISR_Name) instead
      - **Appends RTI Instruction At End Of “ISR_Name” Module**
- **register_handler(ik_ivg11, ISR_Name);**
  - Maps `ISR_Name`’s Address Into Event Vector Table Register (EVT11)
  - Sets IVG11 Bit in IMASK Register

Code Flow (Direct EVT Management Functions)



Refer to Application Note:

EE-192: Using C To Create Interrupt-Driven Systems On Blackfin Processors

KAZTEK ENGINEERING

ANALOG DEVICES

Analog Devices VisualDSP++ - [Target: BF533 EZ-KIT] - [Project: COMPILER_TEST.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

EVT

```
EVT0  2228A88C  EVT8  00000000
EVT1  1120207E  EVT9  00000000
EVT2  00000000  EVT10 00000000
EVT3  00000000  EVT11 00000000
EVT4  00000000  EVT12 00000000
EVT5  00000000  EVT13 FFA000D8
EVT6  FFA000B4  EVT14 00000000
EVT7  00000000  EVT15 FFA0008E
```

main.c

```c
#include <cdefBF533.h>      // BF533 Register Definitions
#include <cdef_LPBlackfin.h>
#include <sys\exception.h>

EX_INTERRUPT_HANDLER(CoreTimerISR)
{
    *pTCNTL |= TINT;            // W1C Core Timer Expired Bit
} // end core timer ISR

EX_REENTRANT_HANDLER(WDOG_ISR)
{
    *pWDOG_CTL = TMR_DIS;       // Disable Watchdog
    *pWDOG_CTL |= TRO;          // W1C Watchdog Timer Expired Bit
    *pWDOG_CTL = ENABLE_GPI;    // Re-Enable Watchdog For GP IRQ
} // end WDOG ISR

main()
{
    // assign ISRs to interrupts
    register_handler(ik_ivg13, WDOG_ISR);
    register_handler(ik_timer, CoreTimerISR);

    // SIC Watchdog Timer Interrupt Is Bit 23
    *pSIC_IMASK |= 0x00800000;

    while(1);   // loop endlessly waiting for interrupts
} // end main
```

Disassembly

```
_CoreTimerISR
[FFA000B4] [ -- SP ] = ASTAT ;
[FFA000B6] [ -- SP ] = FP ;
[FFA000B8] [ -- SP ] = ( R7:0 , P5:0 ) ;
[FFA000BA] SP += -64 ;
[FFA000BC] SP += -56 ;
[FFA000BE] P2.L = 0x3000 ;
[FFA000C2] P2.H = 0xffe0 ;
[FFA000C6] R7 = [ P2 + 0x0 ] ;
[FFA000C8] BITSET ( R7 , 0x3 ) ;
[FFA000CA] [ P2 + 0x0 ] = R7 ;
[FFA000CC] SP += 60 ;
[FFA000CE] SP += 60 ;
[FFA000D0] ( R7:0 , P5:0 ) = [ SP ++ ] ;
[FFA000D2] FP = [ SP ++ ] ;
[FFA000D4] ASTAT = [ SP ++ ] ;
[FFA000D6] RTI ;
_WDOG_ISR
[FFA000D8] [ -- SP ] = RETI ;
[FFA000DA] [ -- SP ] = ASTAT ;
[FFA000DC] [ -- SP ] = FP ;
[FFA000DE] [ -- SP ] = ( R7:0 , P5:0 ) ;
[FFA000E0] SP += -64 ;
[FFA000E2] SP += -56 ;
[FFA000E4] P1.L = 0x200 ;
[FFA000E8] P1.H = 0xffc0 ;
[FFA000EC] R7.L = 2768 ;
[FFA000F0] W [ P1 + 0x0 ] = R7 ;
```

Interrupt nesting gets enabled HERE

Ready | Halted

KAZTEK ENGINEERING

ANALOG DEVICES

Interrupt Dispatcher

- interrupt( ) function

  Usage:

  ```
  #include<sys\exception.h>
  interrupt(ik_ivg11, ISR_Name);
  ```

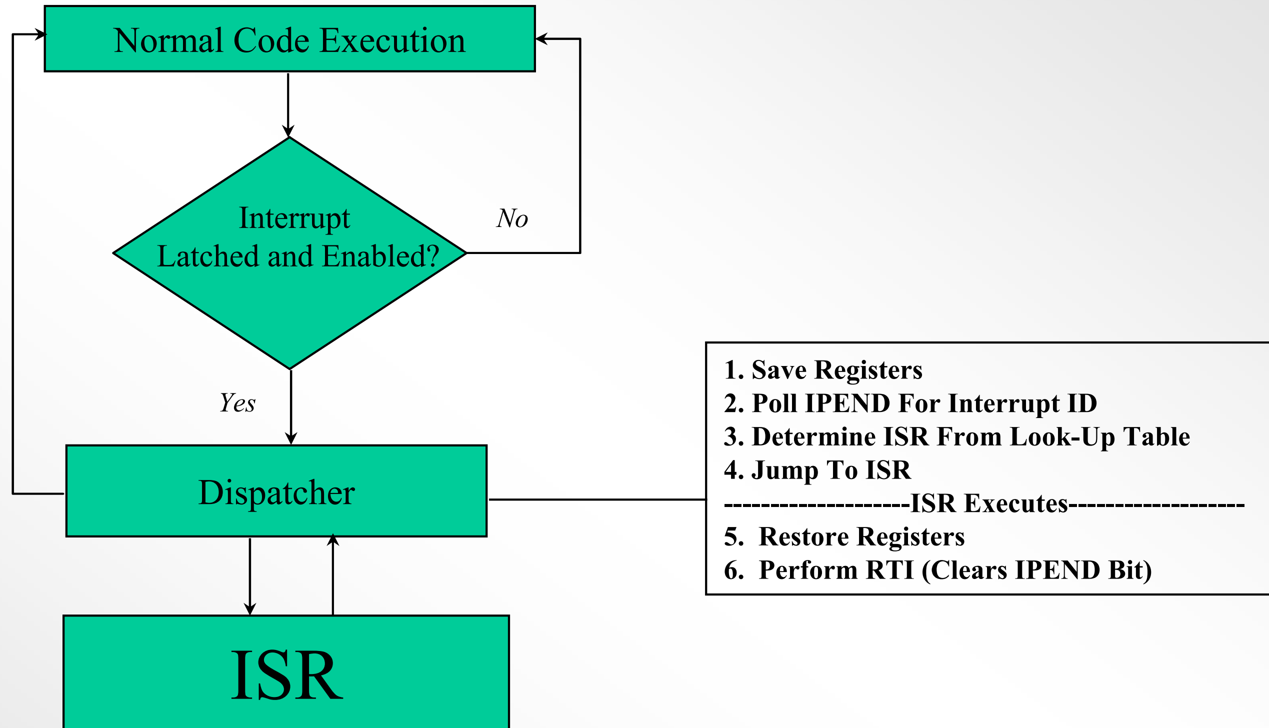
- interrupt (ik_ivg11, ISR_Name);
  - Places ISR_Name's address into internal look-up table (__vector_table)
  - Sets up implied call to register_handler (ik_ivg11, _despint);
    - Maps location of interrupt dispatcher (_despint) into EVT11
    - Sets IVG11 Bit In IMASK And Enables Interrupts
- Interrupt Dispatcher (_despint)
  - Saves processor context by pushing the following registers to the stack:
    - All Data (R0-R7), Pointer (P0-P5), and Accumulator (A0,A1) Registers
    - All DAG (I0-I3, M0-M3, L0-L3, B0-B3) Registers
    - All Loop (LB0-LB1, LT0-LT1, LC0-LC1) Registers
    - Arithmetic Status (ASTAT) and Sequencer Status (SEQSTAT) Registers
    - All Sequencer (RETS, RETI, RETX, RETN, RETE) Registers
      - Pushing of RETI enables interrupt nesting!!
    - System Configuration (SYSCFG) Register

Interrupt Dispatcher (cont.)

- Dispatcher (_despint) Also:
  - Polls IPEND To Determine Which Bit Is Set (Checks Highest Priority First)
  - When A Set IPEND Bit Is Found
    - Offset From Bit 0 Of IPEND Is Index Into Internal Look-Up Table
    - Fetches ISR_Name's Address From Look-Up Table
    - Vectors To and Executes ISR_Name Module
    - Restores Context
    - Executes RTI (Clears IPEND Bit)
  - If Multiple IPEND Bits Are Set, the Highest Priority Interrupt Is Serviced and _despint Gets Called Again Upon Execution of RTI

- The process of saving/restoring context, determining the interrupt source, and finding the vector to take as a result of the event takes ~400-450 cycles, depending on which IPEND bit is set

Code Flow (Dispatcher)



KAZTEK ENGINEERING

ANALOG DEVICES

Analog Devices VisualDSP++ - [Target: BF533 EZ-KIT] - [Project: COMPILER_TEST.dpj]

File Edit Session View Project Register Memory Debug Settings Tools Window Help

Debug

EVT

EVT0	3228A88C	EVT8	00000000
EVT1	1B20A87E	EVT9	00000000
EVT2	00000000	EVT10	00000000
EVT3	00000000	EVT11	00000000
EVT4	00000000	EVT12	00000000
EVT5	00000000	EVT13	FFA00208
EVT6	FFA00208	EVT14	00000000
EVT7	00000000	EVT15	FFA0008E

main.c

```c
#include <cdefBF533.h>      // BF533 Register Definitions
#include <cdef_LPBlackfin.h>
#include <sys\exception.h>

void CoreTimerISR (void)
{
    *pTCNTL |= TINT;            // W1C Core Timer Expired Bit
} // end core timer ISR

void WDOG_ISR (void)
{
    *pWDOG_CTL = TMR_DIS;        // Disable Watchdog
    *pWDOG_CTL |= TRO;           // W1C Watchdog Timer Expired Bit
    *pWDOG_CTL = ENABLE_GPI;     // Re-Enable Watchdog For GP IRQ
} // end WDOG ISR

main()
{
    // assign ISRs to interrupts
    interrupt(ik_ivg13, WDOG_ISR);
    interrupt(ik_timer, CoreTimerISR);

    // SIC Watchdog Timer Interrupt Is Bit 23
    *pSIC_IMASK |= 0x00800000;

    while(1);   // loop endlessly waiting for interrupts
} // end main
```

Disassembly

```
_despint
[FFA00208] [ -- SP ] = ( R7:0 , P5:0 ) ;
[FFA0020A] [ -- SP ] = I3 ;
[FFA0020C] [ -- SP ] = I2 ;
[FFA0020E] [ -- SP ] = I1 ;
[FFA00210] [ -- SP ] = I0 ;
[FFA00212] [ -- SP ] = M3 ;
[FFA00214] [ -- SP ] = M2 ;
[FFA00216] [ -- SP ] = M1 ;
[FFA00218] [ -- SP ] = M0 ;
[FFA0021A] [ -- SP ] = B3 ;
[FFA0021C] [ -- SP ] = B2 ;
[FFA0021E] [ -- SP ] = B1 ;
[FFA00220] [ -- SP ] = B0 ;
[FFA00222] [ -- SP ] = L3 ;
[FFA00224] [ -- SP ] = L2 ;
[FFA00226] [ -- SP ] = L1 ;
[FFA00228] [ -- SP ] = L0 ;
[FFA0022A] [ -- SP ] = A0.x ;
[FFA0022C] [ -- SP ] = A0.w ;
[FFA0022E] [ -- SP ] = A1.x ;
[FFA00230] [ -- SP ] = A1.w ;
[FFA00232] [ -- SP ] = LC1 ;
[FFA00234] [ -- SP ] = LC0 ;
[FFA00236] [ -- SP ] = LT1 ;
[FFA00238] [ -- SP ] = LT0 ;
[FFA0023A] [ -- SP ] = LB1 ;
[FFA0023C] [ -- SP ] = LB0 ;
[FFA0023E] [ -- SP ] = ASTAT ;
[FFA00240] [ -- SP ] = RETS ;
[FFA00242] [ -- SP ] = RETI ;
[FFA00244] [ -- SP ] = RETX ;
[FFA00246] [ -- SP ] = RETN ;
[FFA00248] [ -- SP ] = RETE ;
[FFA0024A] [ -- SP ] = SEQSTAT ;
```

Interrupt nesting gets enabled HERE

Ready | Halted

KAZTEK ENGINEERING

ANALOG DEVICES

Assembly Language Interface

- C-Callable Assembly Language Functions
- Assembly Language Statements Within a C Function (In-Line Assembly)
- Associate C Variables with Assembly Language Symbols

C-Callable Assembly Language Functions

- **Several Issues Involved When Writing C-Callable Assembly Language Functions**
  - **Register Usage**
    - *"Dedicated"* Registers
    - *"Call Preserved"* Registers
    - *"Scratch"* Registers
  - **Argument Passing**
    - First Three Arguments Passed in R0, R1 and R2, respectively
    - Arguments Four and Beyond Passed on Stack
      - 4th Parameter Is Closest to SP at [FP+20], 5th at [FP+24], etc.
    - Return Values of 32 Bits or Less Stored in R0
      - Overflows To R1 for Return Values of 33 to 64 Bits
      - Anything Over 64 Bits Is Allocated on Stack but Passed as Pointer in a Hidden Argument in P0

C/C++ Compiler Register Uses
Dedicated Registers

Registers that C/C++ Compiler Reserves for its Own Use

REGISTER	*VALUE*	*MODIFICATION RULES*
L0 – L3	0	See Note below
SP	Stack Pointer	Stack Management Only, Restore
FP	Frame Pointer	Stack Management Only, Restore

L0-L3 Rules:

The L0-L3 registers define the lengths of the DAG's circular buffers. The compiler makes use of the DAG registers, both in linear mode and in circular buffering mode. The compiler assumes that the Length registers are zero, both on entry to functions and on return from functions, and will ensure this is the case when it generates calls or returns. Your application may modify the Length registers and make use of circular buffers, but you must ensure that the Length registers are appropriately reset when calling compiled functions, or returning to compiled functions. Interrupt handlers must store and restore the Length registers, if making use of DAG registers.

C/C++ Compiler Register Uses

Call Preserved Registers

May be Used in an Assembly Function

Contents Should Be Saved and Restored

Values Assumed to be Preserved Across Function Calls

Call-Preserved Registers Are:

P3 - P5

R4 - R7

KAZTEK ENGINEERING

ANALOG DEVICES

C/C++ Compiler Register Uses

Scratch Registers

Contents DO NOT Need to Be Saved/Restored

Use Freely in Assembly Sub-Routines

P0	Used as the Aggregate Return Pointer.
P1—P2	
R0—R3	The first three words of the argument list are always passed in R0, R1 and R2 if present (R3 is not used for parameters)
LB0—LB1	
LC0—LC1	
LT0—LT1	
ASTAT	including CC
A0—A1	
I0—I3	
B0—B3	
M0—M3	

KAZTEK ENGINEERING

ANALOG DEVICES

C-Callable Assembly Language Functions

- **Macros in asm_sprt.h Provided to Make Function Calling Easier**
  - **Save/Restore Preserved Registers (pushs, pops)**
  - **Restore Frame and Stack Pointers (exit)**

```
pushs(x); // Save value in register onto stack
        pushs(R5);   ->   [- -SP] = R5;

pops(x);  // Read value off top of stack to a register
        pops(R5);  ->  R5 = [SP++];

exit;     // Restore stack/frame pointers and jump to return address
        exit;  ->  P0 = [FP + 0x4];
                   JUMP (P0);
```

KAZTEK ENGINEERING

ANALOG DEVICES

In-Line Assembly Language

- **In-Line Assembly Is Accomplished Using the asm( ) Construct**

Example:

```
asm("RO = w[p0];");
asm("BITSET(R0,7);");
asm("ssync;");
```

Note: Can Produce Less Efficient Compiled Code – Optimizer Might Re-Sequence Instructions for Optimal Performance

KAZTEK ENGINEERING

ANALOG DEVICES

Mixed C/Assembly Naming Conventions

C PROGRAM	*ASSEMBLY ROUTINE*
int c_var; */* global variable */*	.extern _c_var;
void c_func();	.extern _c_func;
extern int asm_var	.global _asm_var;
extern void asm_func();	.global _asm_func; …… _asm_func:

To name an assembly symbol that corresponds to a C symbol, add an underscore prefix to the C symbol. Declare as a global variable in C program and as EXTERN in assembly routine

To use an assembly function or variable in your C program, declare the symbol with .GLOBAL directive in assembly routine and as EXTERN in the C program

Example --
Add 5 Numbers in an Assembly Function

- **Example C Program That Calls an Assembly Function (add5)**
  - **Adds 5 Integers Passed From C Calling Routine As Arguments**

C code

```
extern int add5(int,int,int,int,int);           /* Function is located in assembly module */
volatile int sum;                                /* Variable only used in assembly sub-routine*/
                                                 /* volatile keeps sum from being optimized out */
main()
{
    int a=1; int b=2; int c=3; int d=4; int e=5;     /* Initialize parameters */
    int result=0;                                /* result and sum will have the same value */
    result = add5(a,b,c,d,e);                        /* Call to the ADD5 function */
    exit(0);
}
```

KAZTEK ENGINEERING

ANALOG DEVICES

Assembly Routine

/ Assembly Routines with Parameters Example - _add5 */*
/ int add5 (int a, int b, int c, int d, int e); */*
/ This is an assembly language routine that will add 5 numbers */*

```
#include <asm_sprt.h>          /* Header file that defines the stack manipulation macros */

.section program;
.global _add5;
.extern _sum;

_add5:
r0=r0+r1;          /* Add the first and second parameter */
r0=r0+r2;          /* Add the third parameter */
r1=[FP+20];        /* Put the fourth parameter in R1 */
r0=r0+r1;          /* Add the fourth parameter */
r1=[FP+24];        /* Put the fifth parameter in R1 */
r0=r0+r1;          /* R0 is always the return value, variable "result" from C will get r0 value */
p0.h = _sum;       /* we can also write directly to a globally defined variable as well */
p0.l =_sum;        /* could be used if this function was implemented with no return type */
w[p0] = r0;        /* Place the sum in the global variable (C is unaware of this assignment)*/
exit;              /* Restores frame and stack  pointers */
```

KAZTEK ENGINEERING

ANALOG DEVICES

Optimizing C Code

- Optimization Can Decrease Code Size or Lead to Faster Execution
  - Can Be Controlled by Optimization Switch
    - no switch — optimization disabled
    - -O — optimization for speed enabled
    - -Os — optimization for size enables
    - -ipa — inter-procedural optimization enabled
    - -Ov num — enable speed vs size optimization (sliding scale) (Automatically inlines small functions)
  - Can Be Further Controlled In C Source Code Using Pragmas
    - #pragma optimize_off — - Disables Optimizer
    - #pragma optimize_for_space — - Decreases Code Size
    - #pragma optimize_for_speed — - Increases Performance
    - #pragma optimize_as_cmd_line — - Restore optimization per command line options
- Other Optimization Ideas
  - PGO (Profile guided Optimization) used with IPA
  - Take Advantage of Existing Assembly Library Functions
  - Write Time-Critical Routines in Assembly as a C-Callable Subroutine
  - See App Note, “EE-149: Tuning C Source Code For The Blackfin DSP Compiler”

Profile Guided Optimization.

- Program is run with training data.
- Compiled Simulation produces execution trace.

  (Compiled simulation is hundreds of times faster than normal simulation.)
- Re-compile program using execution trace as guidance.
- Compiler now knows result of all conditional operations.
- Compiler also knows where execution hot spots are.
- Better code
- Could also be used to control space/speed trade-off.
- Problem: If what matters to you is worst case, not majority case, then choose training data appropriately.

Circular addressing

- **-force-circbuf**

The –force-circbuf switch treats array references of the form array[i%n] as circular buffer operations. (where n is a power of 2)

- Explicit circular addressing of an array index:

long circindex(long index, long incr, unsigned long nitems)

- Explicit circular addressing on a pointer:

*void * circptr(void *ptr, long incr, void *base, unsigned long buflen)*

KAZTEK ENGINEERING

ANALOG DEVICES

The Video Operations

- Align operations
- Packing operations
- Disaligned loads
- Unpacking
- Quad 8-bit add subtract
- Dual 16-bit Add/Clip
- Quad 8-bit average
- Accumulator extract with addition
- Subtract absolute accumulate

- Eg.

```
bytesI2 = loadbytes((int *)ptrI); ptrI += 4;
bytesB2 = loadbytes((int *)ptrB); ptrB += 4;
srcI = compose_i64(bytesI1, bytesI2);
srcB = compose_i64(bytesB1, bytesB2);
saar(srcI, ptrI, srcB, ptrB, sum1, sum2, sum1, sum2);
```

KAZTEK ENGINEERING

ANALOG DEVICES

Getting Started 80:20

Find out where program spends its time.

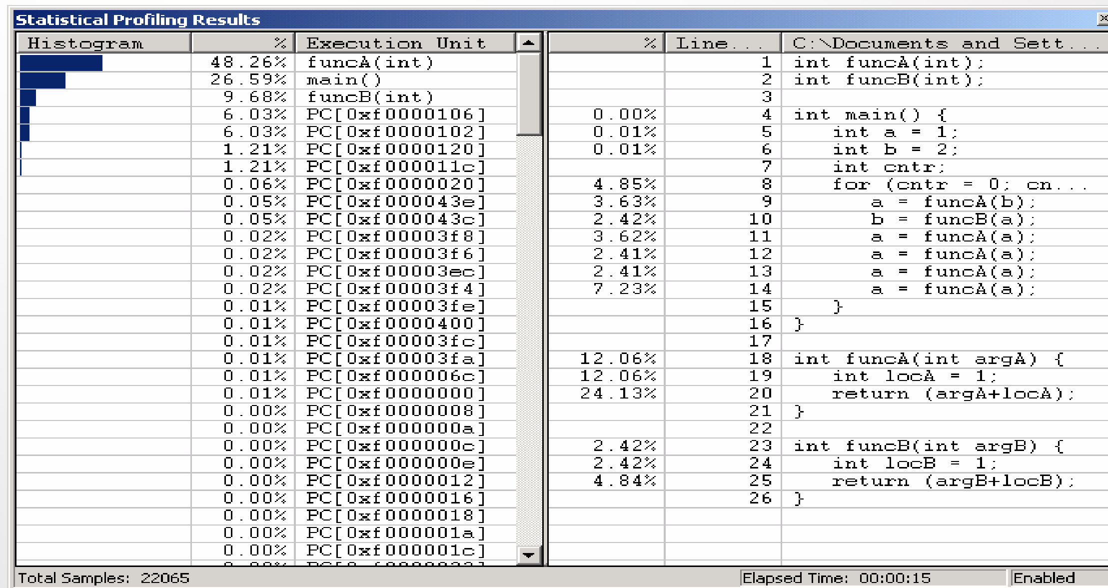
- **80 – 20 rule**
- Measure**: Intuition is notoriously bad here: instrument, use profiler and cycle accurate simulator.**
- Loops: **Are always a good place to look. Even a trivial operation can have a significant cost, if it is done often enough.**

KAZTEK ENGINEERING

ANALOG DEVICES

VDSP Statistical Profiler

- The profiler is very useful in C/C++ mode because it makes it easy to benchmark a system module-by-module (I.e. C/C++ function).
- Assembly or optimised code appears as individual instructions.



- Linear Profiler is also available for the simulator.

KAZTEK ENGINEERING

ANALOG DEVICES

Mixed Mode.
Statistical results at the instruction level.

Costly instructions are easy to spot.

[illegible]	308	cosine = (double)(2.0 * cos((2 * v_1 + 1) * cosMultB	
0.04%		[FFA00966] P0 = [FP + -72] ;	
0.17%		[FFA00968] R0 = [P0 + 0x0] ;	
0.04%		[FFA0096A] R0 <<= 0x1 ;	<- Pipeline stalls
0.04%		[FFA0096C] R0 += 1 ;	
0.21%		[FFA0096E] CALL ___int32_to_float32 ;	
0.04%		[FFA00972] R1 = R0 ;	<- Transfer of control
0.04%		[FFA00974] R0 = R6 ;	
0.21%		[FFA00976] CALL ___float32_mul ;	
0.21%		[FFA0098A] CALL __Sin ;	
0.04%		[FFA0098E] R1 = R0 ;	
0.21%		[FFA00990] CALL ___float32_add ;	
	309	}	
	310	else	