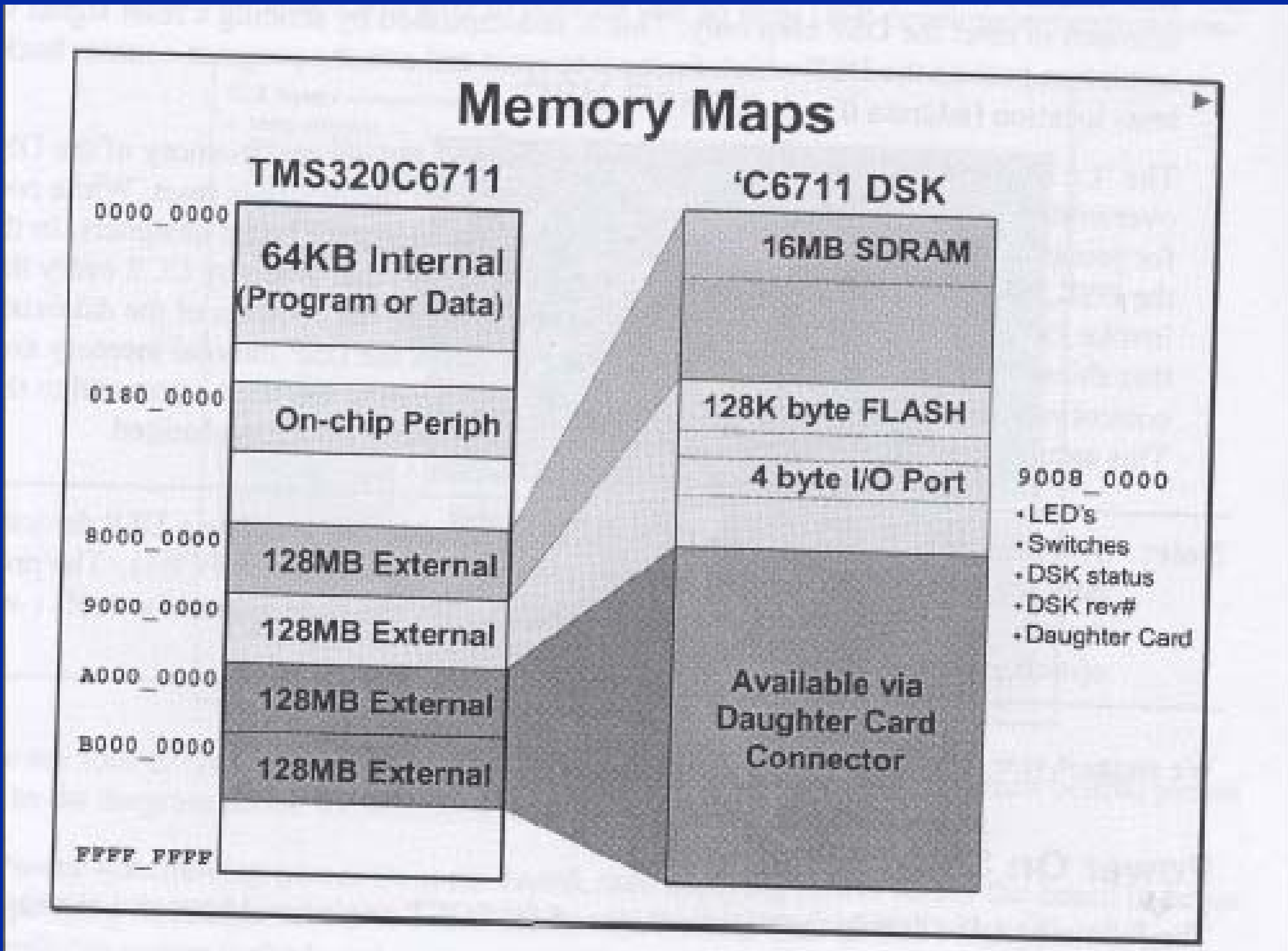
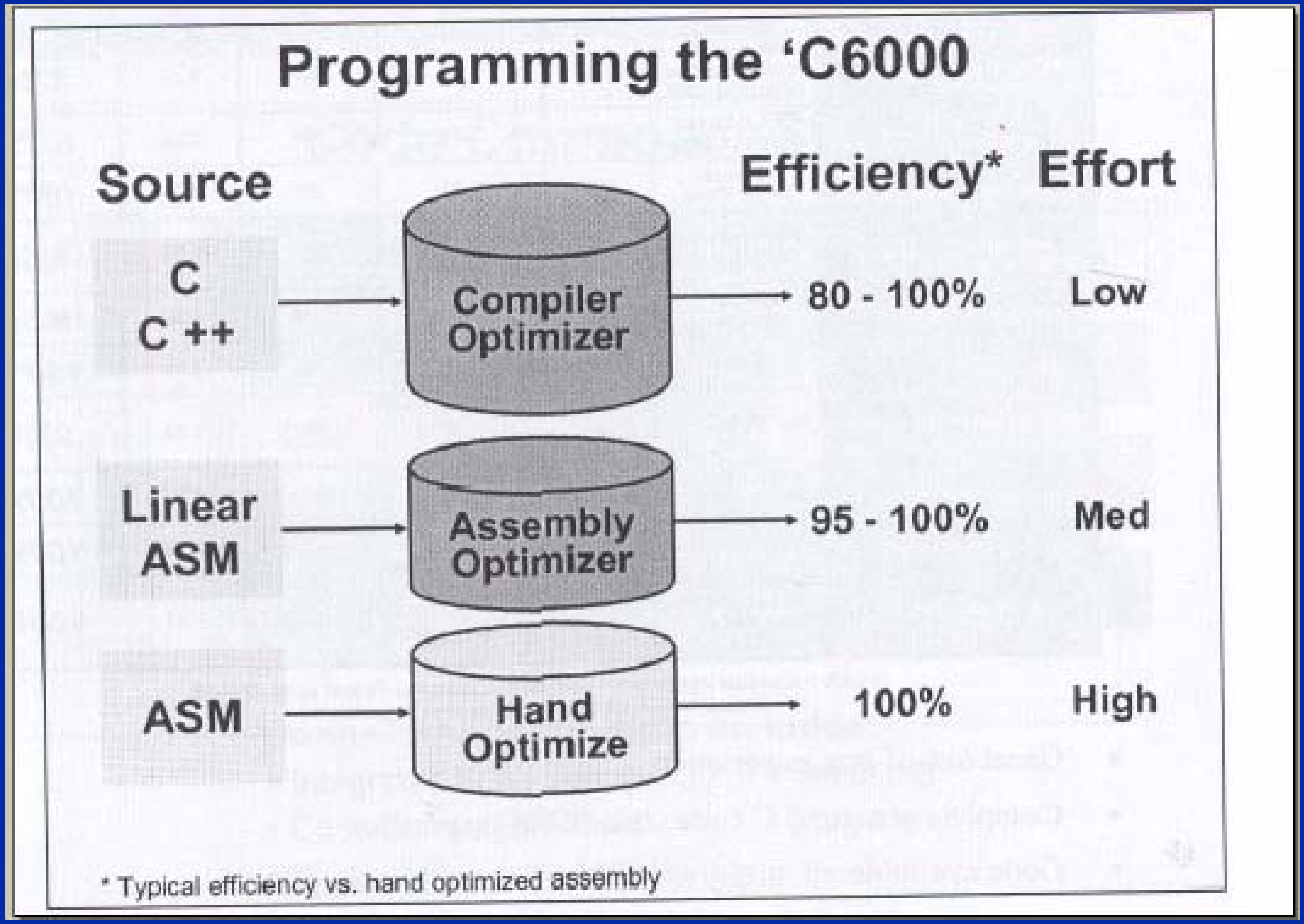
Processeurs de traitement du signal et de l'image

Chap 3: Programmation DSP TMSC6711: C et assembleur

Gestion Mémoire


Memory Maps
TMS320C6711
'C6711 DSK
0000_0000
64KB Internal
(Program or Data)
0180_0000
On-chip Periph
8000_0000
128MB External
9000_0000
128MB External
A000_0000
128MB External
B000_0000
128MB External
FFFF_FFFF
16MB SDRAM
128K byte FLASH
4 byte I/O Port
9008_0000
• LED's
• Switches
• DSK status
• DSK rev#
• Daughter Card
Available via
Daughter Card
Connector

```
/* Memory Map 1 , command file .cmd*/
MEMORY
{
   PMEM          : origin = 0x00000000,  len = 0x00010000
   DMEM          : origin = 0x80000000,  len = 0x01000000
}
SECTIONS
{
       .text          > PMEM
       .bss           > PMEM
       .cinit         > PMEM
       .const         > PMEM
       .stack         > PMEM
       .cio           > PMEM
       .sysmem        > PMEM
       .far           > PMEM
       .mydata  > DMEM
}
```


Programming the 'C6000
Source
Efficiency*
Effort
C
C ++
Compiler Optimizer
80 - 100%
Low
Linear ASM
Assembly Optimizer
95 - 100%
Med
ASM
Hand Optimize
100%
High
* Typical efficiency vs. hand optimized assembly

Sample Compiler Benchmarks

Algorithm	Used In	Asm Cycles	Assembly Time (µs)	C Cycles (Rel 4.0)	C Time (µs)	% Efficiency vs Hand Coded
Block Mean Square Error *MSE of a 20 column image matrix*	For motion compensation of image data	348	1.16	402	1.34	*87%*
Codebook Search	CELP based voice coders	977	3.26	961	3.20	*100%*
Vector Max *40 element input vector*	Search Algorithms	61	0.20	59	0.20	*100%*
All-zero FIR Filter *40 samples, 10 coefficients*	VSELP based voice coders	238	0.79	280	0.93	*85%*
Minimum Error Search *Table Size = 2304*	Search Algorithms	1185	3.95	1318	4.39	*90%*
IIR Filter *16 coefficients*	Filter	43	0.14	38	0.13	*100%*
IIR – cascaded biquads *10 Cascaded biquads (Direct Form II)*	Filter	70	0.23	75	0.25	*93%*
MAC *Two 40 sample vectors*	VSELP based voice coders	61	0.20	58	0.19	*100%*
Vector Sum *Two 44 sample vectors*		51	0.17	47	0.16	*100%*
MSE *MSE between two 256 element vectors*	Mean Sq. Error Computation in Vector Quantizer	279	0.93	274	0.91	*100%*

TI C62x™ Compiler Performance Release 4.0: Execution Time in µs @ 300 MHz
Versus hand-coded assembly based on cycle count

'C6000 C Data Types

Type	Size	Representation
char, signed char	8 bits	ASCII
unsigned char	8 bits	ASCII
short	**16 bits**	**2's complement**
unsigned short	16 bits	binary
int, signed int	**32 bits**	**2s complement**
unsigned int	32 bits	binary
long, signed long	**40 bits**	**2's complement**
unsigned long	40 bits	binary
enum	32 bits	2's complement
float	**32 bits**	**IEEE 32-bit**
double	64 bits	IEEE 64-bit
long double	64 bits	IEEE 64-bit
pointers	32 bits	binary

```
;Initialisation de la mémoire,file .asm

 .sect ".mydata"
          .short   0
          .short   7
          .short   10
          .short   7
          .short   0
          .short   -7
          .short   -10
          .short   -7
          .short   0
          .short   7
```

```
//ACCES LECTURE et affichage mémoire
#include <stdio.h>
void main()
{
    int i;
            short *point;
            point= (short *) 0x80000000;

            printf("BEGIN\n");

            for(i=0;i<10;i++)
            {
                        printf(“point[%d]=%d\n",i, point[i]);
            }

            printf("END\n");
}
```

Interfacing C and Assembly Code

Introduction

- This chapter shows how to interface C and assembly
- As a general rule the code written in C is used for initialisation and for non-critical (in terms of speed or size) code.
- Critical code (in terms of speed/size) can be written in assembly or linear assembly.
- There are three different ways to interface C and assembly code:

  (1) C code call to the assembly function.

  (2) An interrupt can call an assembly function.

  (3) Call an assembly instruction using intrinsics.

Calling Assembly from C

main_c.c

```
int c_Function (short a, short b);
Extern asm_Function ();

short x = 0x4000, y = 0x2000;
int z,v;

void main (void)
{
      z = c_Function (x, y);
      v = asm_Function (x, y);
}
```

- Use "_" underscore in assembly for all variables or functions declared in C.
- Labels also need to be global.

c_Function.c

```
int c_Function (short a, short b)
{
      int y;
      y = (a * b) << 1;
      return y;
}
```

asm_Function.asm

```
      .global _asm_Function

_asm_Function:

      MVK ....
```

Passing Arguments between C and Assembly

- **The following registers are used to pass and return variables when calling an assembly routine from C.**

A		B
	0	
	1	
	2	
	3	ret addr
arg1/r_val	4	arg2
	5	
arg3	6	arg4
	7	
arg5	8	arg6
	9	
arg7	10	arg8
	11	
arg9	12	arg10
	13	
	14	
	15	

Passing Arguments between C and Assembly

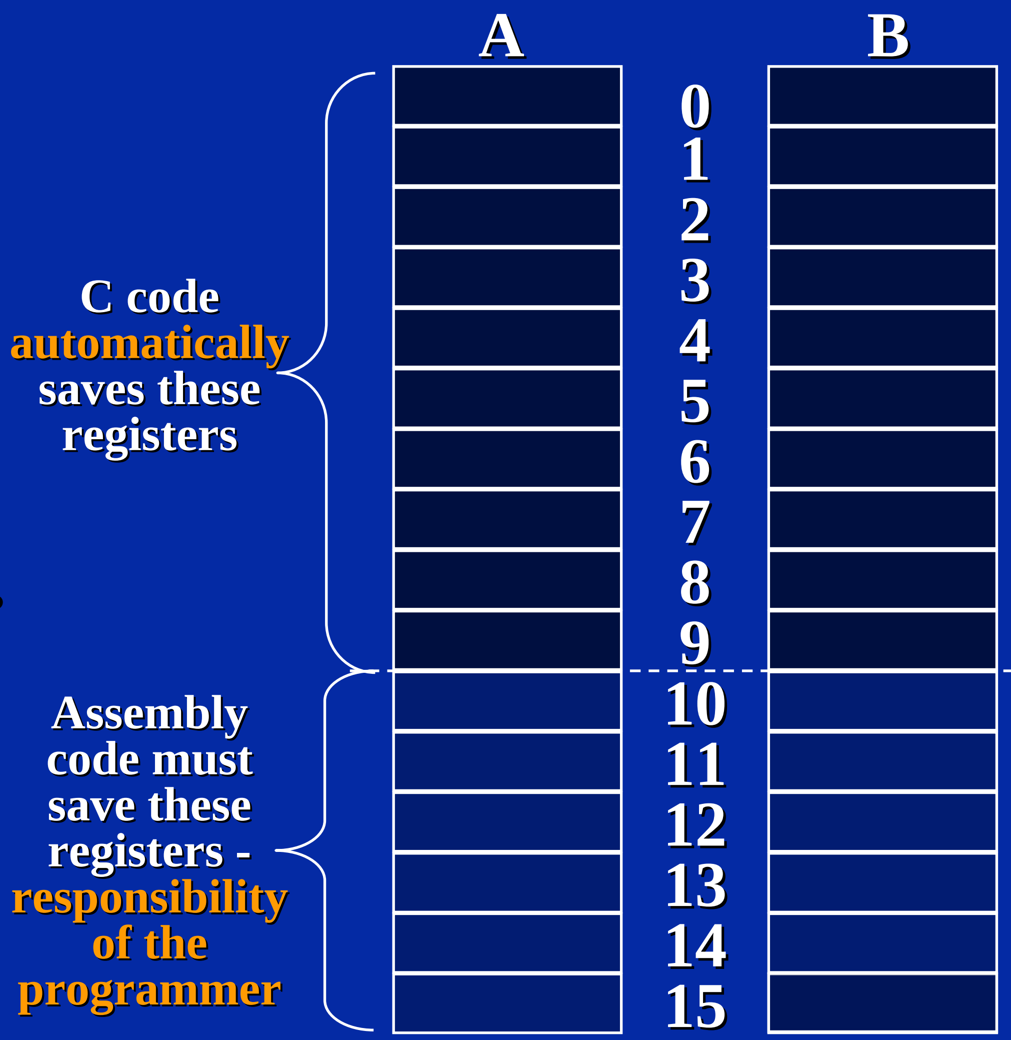
Problem:

- **The C code will use some or all of the registers.**
- **The assembly code may also require the use of some or all registers.**
- **If nothing is done then on return to the C code some of the values may have been destroyed by the assembly code.**

Passing Arguments between C and Assembly

Solution:

- Both the C code and assembly code are responsible for saving some registers if they need to use them.



```c
// Exemple Calcul somme par fonction C ret_sum
#include <stdio.h>
void main()
{
		int i,ret;
		short *point;
		point = (short *) 0x80000000;
		printf("BEGIN\n");
		for(i=0;i<10;i++)
		{
				printf(“point[%d]= %d\n",i, point[i]);
		}
		ret = ret_sum(point,10);
		printf("Sum = %d\n",ret);
		printf("END\n");
}
int ret_sum(const short* array, int N)
{
		int count,sum;

		sum=0;

		for(count=0 ; count < N ; count++)
				sum += array[count];

		return(sum);
}
```

A FAIRE EN TP

A FAIRE EN TP

```
// Exemple d'appel de fonction assembleur
#include <stdio.h>
extern sum();
void main()
{
        int i,ret;
        short *point;
        point = (short *) 0x80000000;
        printf("BEGIN: Call assembly sum function\n");
        for(i=0;i<10;i++)
        {
                printf("point[%d]= %d\n",i, point[i]);
        }
        ret = sum(point,10);
// arg1=point dans A4 et arg2=10 dans B4
        printf("Sum = %d\n",ret);
        printf("END\n");
}
```

;fonction sum dans file sum.asm

A FAIRE EN TP

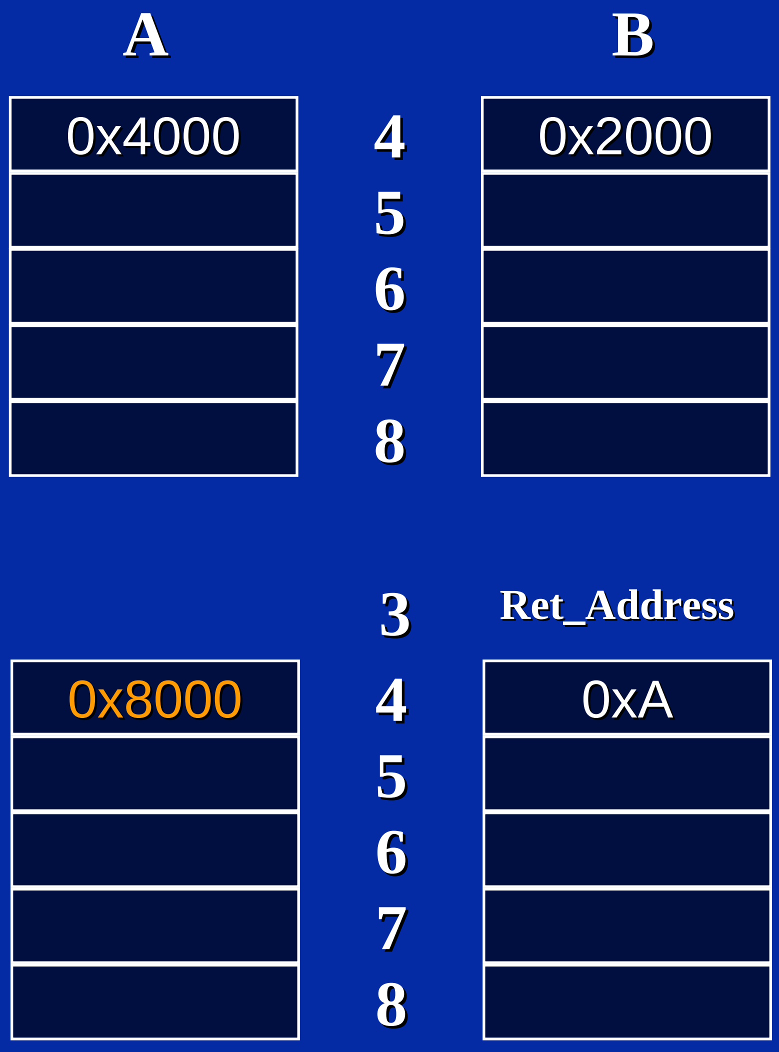
```
                    .global   _sum

_sum:
                    ZERO      .L1       A9              ;Sum register
                    MV        .L1       B4,A2           ;initialize counter A2 with passed argument
                                                         ; arg2=10 dans B4
loop:     LDH       .D1       *A4++, A7                 ;load value pointed by A4=arg1=address
                                                        ; into register A7
                    NOP       4
                    ADD       .L1       A7,A9,A9        ;A9 += A7
   [A2] SUB         .L1       A2,1,A2                    ;decrement counter
   [A2]      B      .S1       loop                       ;branch back to loop
                    NOP       5

                    MV        .L1       A9,A4            ;move result into return register A4
                    B         .S2       B3               ;branch back to address stored in B3
                    NOP       5
```

Passing Arguments between C and Assembly

- **Before assembly call.**
- **After return from assembly call.**



Benchmark / Profile Code

"Analyzing how long it takes code to execute" is defined as *profiling*. It is also called *benchmarking*. In our example, we are interested in benchmarking the dot-product function.

34. Before we begin, let's restart our program. Make sure the processor is halted, then:

```
Debug:Restart

Debug: Go Main
```

Note: If you choose *CPU Reset* rather than *Restart*, the DSP will be reset and your program will be erased. If this occurs, just reload the program again with **File:Reload Program.**

---

35. Also, let's close some windows to free up some extra room. Close the *Watch* and *Command* windows (**right-click** on each of them and select **close**.)

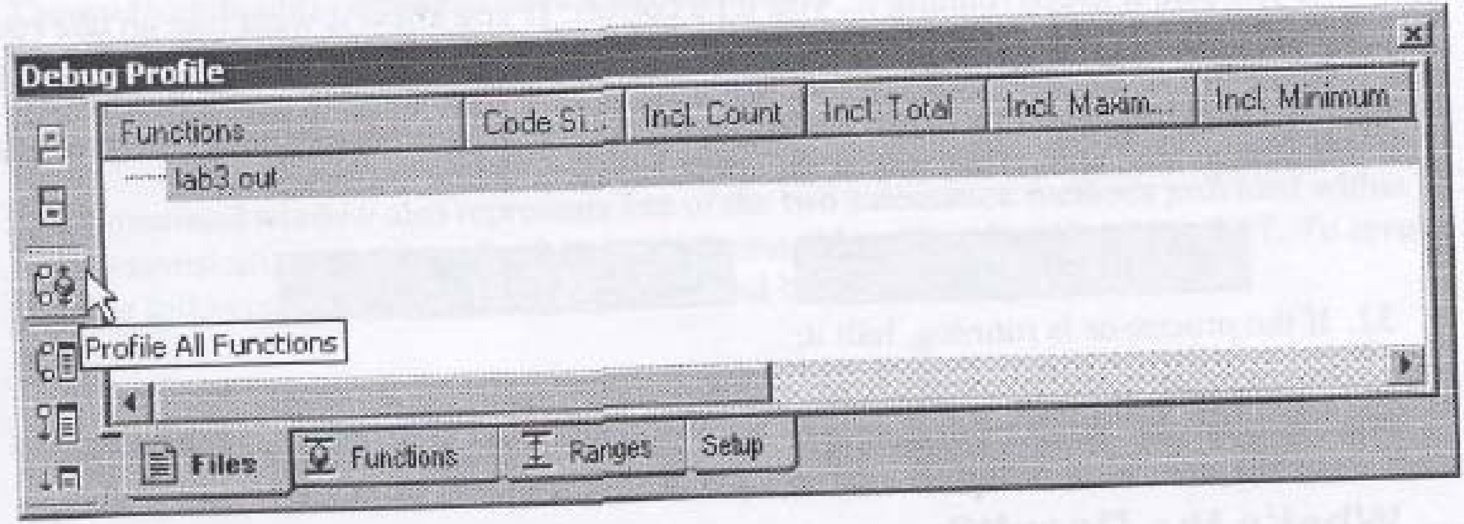
36. Open a new profiling session.

```
Profiler:Start New Session…
```

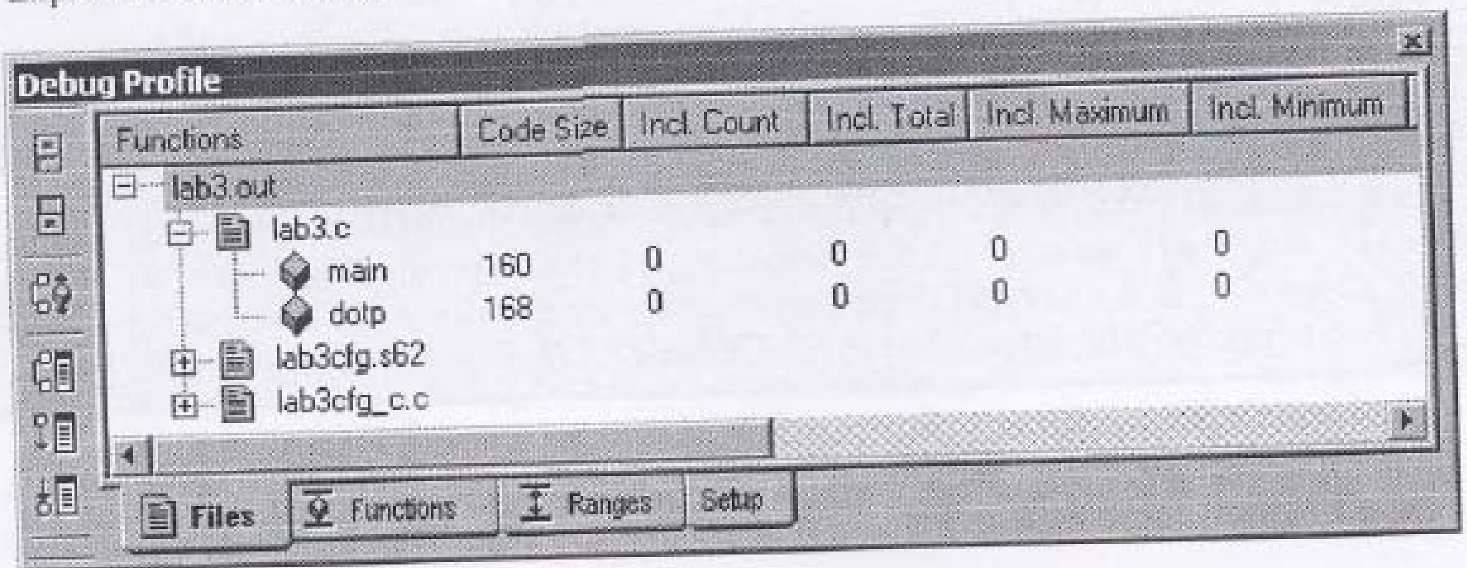
Go ahead and name the session anything you want. We called ours **Debug** profile.

37. Choose **Profile All Functions**.

Click the **Profile All Functions** toolbar button



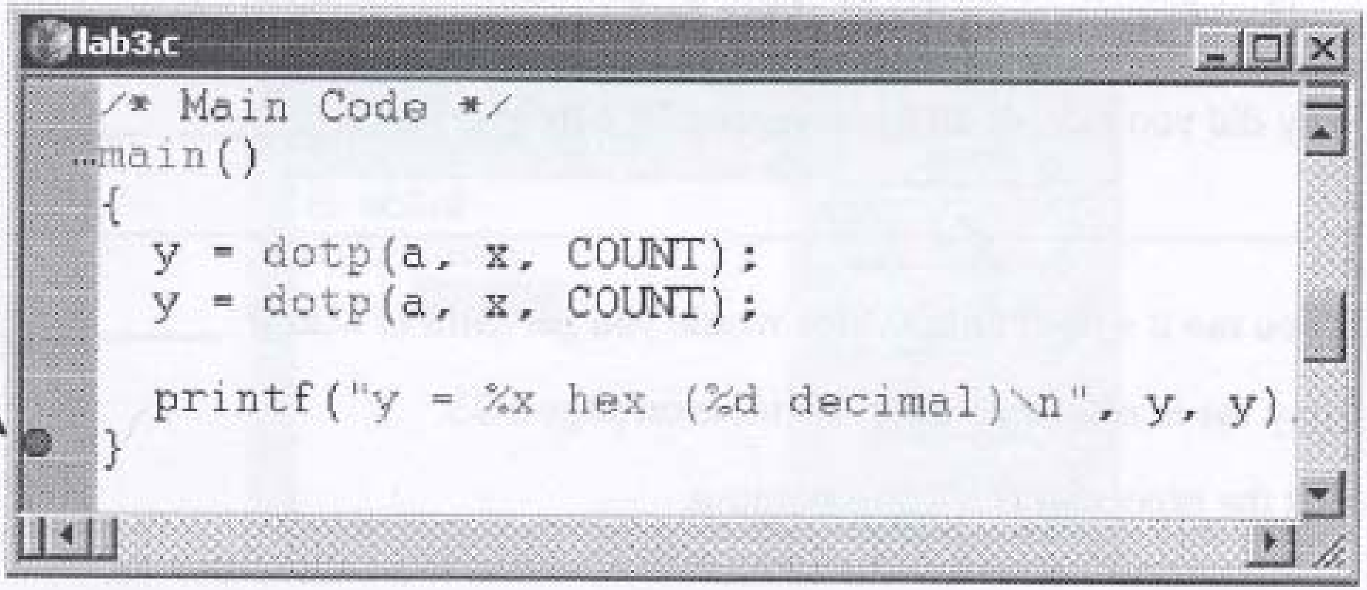
38. Expand the *lab3.c* entry.



39. Before running the code, set a breakpoint at the end of **main()** to stop the processor automatically.

```
Double-clicking in the left most column sets a breakpoint
```

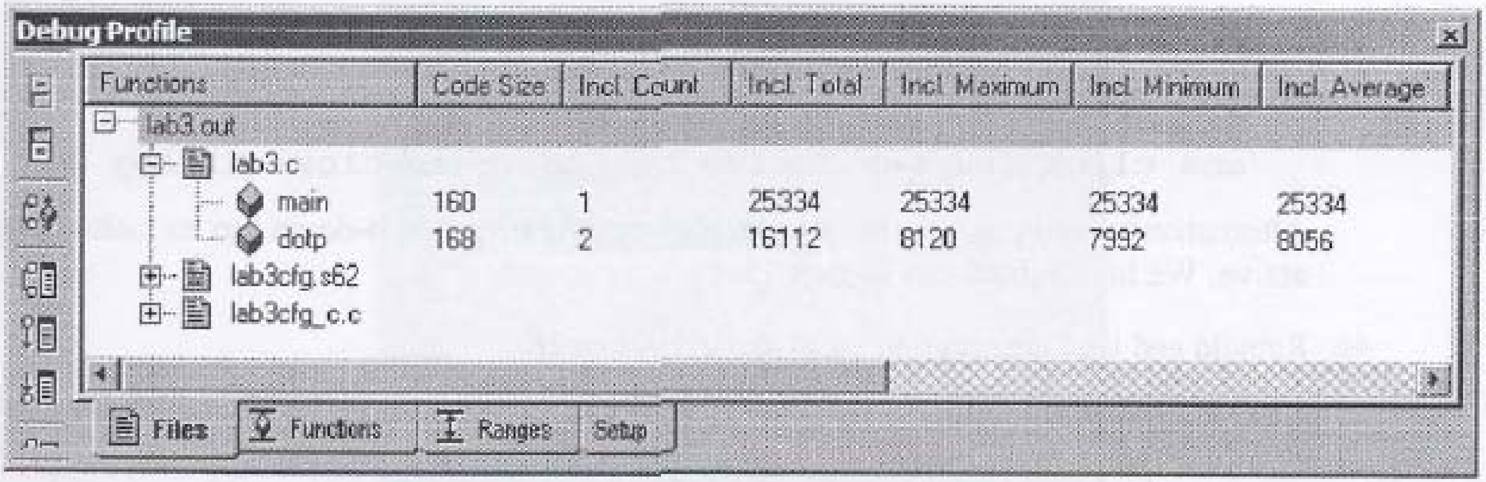
You should see a red dot appear, indicating a breakpoint.



40. Run the program and single-step once.

```
F5  then  F8
```

41. Examine the *Debug Profile* window.



So, what statistics do we care about?

We are interested in the minimum number of cycles for the dotp function: 7992

What are all the columns for?

There are so many statistics shown that we can't fit the screen capture on our printed page. Here are some points about the other columns:

- *Code Size* and *Count* should be self explanatory.
- *Exclusive* (Excl.) shows the cycles for the specified function only.
  (You may have to scroll right to see this in your profile window.)
- *Inclusive* (Incl.) is the cycles required for the specified function plus all sub-functions.
  For example:

main Incl Total =	main Excl Total +	dotp Excl Total +	printf (not shown)
25334	112	16112	9110

Chapter 9
Software Optimisation

Part 1:   Optimisation Methods.

Part 2:   Software Pipelining.

(Later)

Objectives

- Introduction to optimisation and optimisation procedure.
- Optimisation of C code using the code generation tools.
- Optimisation of assembly code.

Introduction

- Software optimisation is the process of manipulating software code to achieve two main goals:
  - Faster execution time.
  - Small code size.

Note: It will be shown that in general there is a trade off between faster execution type and smaller code size.

Introduction

- **To implement efficient software, the programmer must be familiar with:**
  - **Processor architecture.**
  - **Programming language (C, assembly or linear assembly).**
  - **The code generation tools (compiler, assembler and linker).**

Optimising C Compiler Options

- The ‘C6x optimising C compiler uses the ANSI C source code and can perform optimisation currently up-to about 80% compared with a hand-scheduled assembly.
- However, to achieve this level of optimisation, knowledge of different levels of optimisation is essential. Optimisation is performed at different stages and levels.

Assembly Optimisation

- To develop an appreciation of how to optimise code, let us optimise an FIR filter:

$$\boxed{y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]}$$

- For simplicity we write:

$$\boxed{y[n] = \sum_{i=0}^{N-1} h[i] \cdot x[i]} \qquad [1]$$

Assembly Optimisation

- To implement Equation 1, we need to perform the following steps:
  (1) Load the sample x[i].
  (2) Load the coefficients h[i].
  (3) Multiply x[i] and h[i].
  (4) Add (x[i] * h[i]) to the content of an accumulator.
  (5) Repeat steps 1 to 4 N-1 times.
  (6) Store the value in the accumulator to y.

Assembly Optimisation

- **Steps 1 to 6 can be translated into the following 'C6x assembly code:**

```
      MVK   .S2   0,B0        ; Initialise the loop counter
      MVK   .S1   0,A5        ; Initialise the accumulator
loop  LDH   .D1   *A8++,A2    ; Load the samples x[i]
      LDH   .D1   *A9++,A3    ; Load the coefficients h[i]
      NOP         4           ; Add "nop 4" because the LDH has a latency of 5.
      MPY   .M1   A2,A3,A4    ; Multiply x[i] and h[i]
      NOP                     ; Multiply has a latency of 2 cycles
      ADD   .L1   A4,A5,A5    ; Add "x [i]. h[i]" to the accumulator
[B0]  SUB   .L2   B0,1,B0     ; }
[B0]  B     .S1   loop        ; } loop overhead
      NOP         5           ; } The branch has a latency of 6 cycles
```

Assembly Optimisation

- In order to optimise the code, we need to:
  - (1) Use instructions in parallel.
  - (2) Remove the NOPs.
  - (3) Remove the loop overhead (remove SUB and B: loop unrolling).
  - (4) Use word access or double-word access instead of byte or half-word access.

Step 1 - Using Parallel Instructions

Cycle	.D1	.D2	.M1	.M2	.L1	.L2	.S1	.S2	NOP
1	ldh								
2		ldh							
3									nop
4									nop
5									nop
6									nop
7			mpy						
8									nop
9					add				
10						sub			
11							b		
12									nop
13									nop
14									nop
15									nop
16									nop

Step 1 - Using Parallel Instructions

Cycle	.D1	.D2	.M1	.M2	.L1	.L2	.S1	.S2	NOP
1	ldh	ldh							
2									nop
3									nop
4									nop
5									nop
6			mpy						
7									nop
8					add				
9						sub			
10							b		
11									nop
12									nop
13									nop
1[illegible]									nop
1[illegible]									nop
1[illegible]									

Note: Not all instructions can be put in parallel since the result of one unit is used as an input to the following unit.

Step 2 - Removing the NOPs

Cycle	.D1	.D2	.M1	.M2	.L1	.L2	.S1	.S2	NOP
1	ldh	ldh							
2						sub			
3							b		
4									nop
5									nop
6			mpy						
7									nop
8					add				
9									
10									
11									
12									
13									
14									
15									
16									

```
loop LDH   .D1  *A8++,A2
     LDH   .D1  *A9++,A3
[B0] SUB   .L2  B0,1,B0
[B0] B     .S1  loop
     NOP        2
     MPY   .M1  A2,B3,A4
     NOP
     ADD   .L1  A4,A5,A5
```

Step 3 - Loop Unrolling

- The SUB and B instructions consume at least two extra cycles per iteration (this is known as branch overhead).

```
loop    LDH     .D1     *A8++,A2
        LDH     .D1     *A9++,A3
[B0]    SUB     .L2     B0,1,B0
[B0]    B       .S1     loop
        NOP             2
        MPY     .M1     A2,B3,A4
        NOP
        ADD     .L1     A4,A5,A5
```

→

```
     LDH   .D1    *A8++,A2   ;Start of iteration 1
||   LDH   .D1    *B9++,B3
     NOP          4
     MPY   .M1X   A2,B3,A4   ;Use   of   cross   path

     NOP
     ADD   .L1    A4,A5,A5
     LDH   .D1    *A8++,A2   ;Start of iteration 2
||   LDH   .D1    *A9++,A3
     NOP          4
     MPY   .M1    A2,B3,A4
     NOP
     ADD   .L1    A4,A5,A5
;          :
;          :
;          :
     LDH   .D1    *A8++,A2   ; Start of iteration n
||   LDH   .D1    *A9++,A3
     NOP           4
     MPY   .M1    A2,B3,A4
     NOP
     ADD   .L1    A4,A5,A5
```

Step 4 - Word or Double Word Access

- The ‘C6711 has two 64-bit data buses for data memory access and therefore up to two 64-bit can be loaded into the registers at any time.
- In addition the ‘C6711 devices have variants of the multiplication instruction to support different operation.

Note: Store can only be up to 32-bit.

Step 4 - Word or Double Word Access

- **Using word access, MPYL and MPYH the previous code can be written as:**

```
loop
        LDW     .D1     *A9++,A3 ; 32-bit word is loaded in a single cycle
||      LDW     .D2     *B6++,B1
        NOP             4
[B0]    SUB     .L2
[B0]    B       .S1     loop
        NOP             2
        MPYL    .M1     A3,B1,A4 ;;;to be verified
||      MPYH    .M2     A3,B1,B3 ;;;to be verified
        NOP
        ADD     .L1     A4,B3,A5
```

- **Note: By loading words and using MPY and MPYH instructions the execution time has been halved since in each iteration two 16x16-bit multiplications are performed.**

Optimisation Summary

- **It has been shown that there are four complementary methods for code optimisation:**
  - **Using instructions in parallel.**
  - **Filling the delay slots with useful code.**
  - **Using word or double word load.**
  - **Loop unrolling.**

These increase performance and reduce code size.

Optimisation Summary

- It has been shown that there are four complementary methods for code optimisation:
  - Using instructions in parallel.
  - Filling the delay slots with useful code.
  - Using word or double word load.
  - Loop unrolling.

This increases performance but increases code size.