Embedded System Design

ARM Programming

Building ARM code

- To use the ARM C compiler:

  armcc -c code.c tcc –c thumb.c

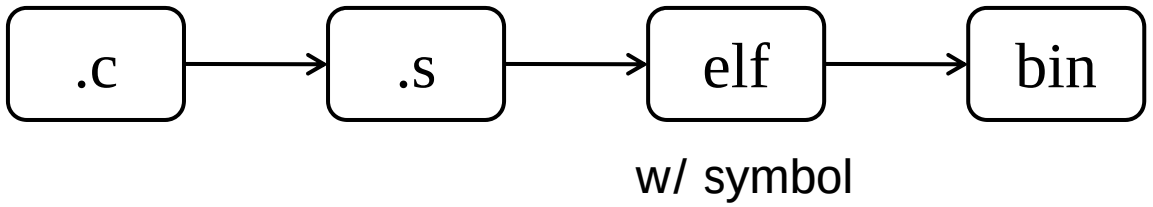
  armcc -S code.c
- ARM assembler:

  armasm -g code.s
- To build a library

  armar -r test.lib test1.o test2.o test3.o

  armar -t test.lib // display archived objects
- The object code can then be linked to produce an executable:

  armlink code.o -o code



Preprocess (-E option)

- arm –E test1.c > test1.i

- To prevent multiple #include

  ```
  #ifndef __TEST1_H__
  #define __TEST1_H__
  ...
  #endif
  ```

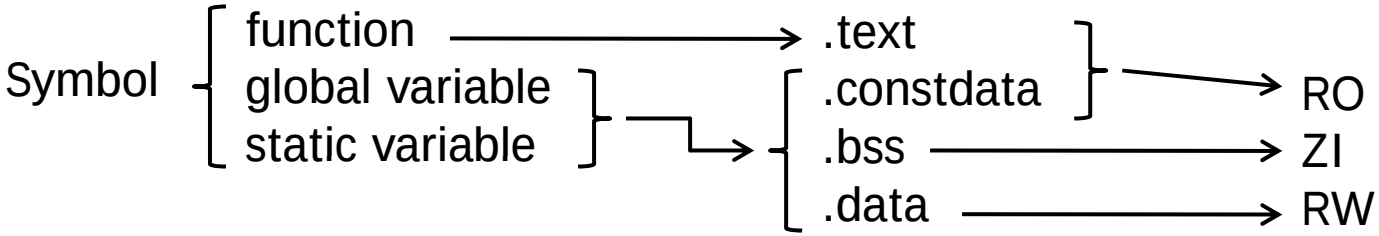
Variables

- local
  - valid within the function or block {}
- global
  - valid within a file
  - can be used by other files
- static
  - local static: retain its value
  - global static: protected from other files
- volatile

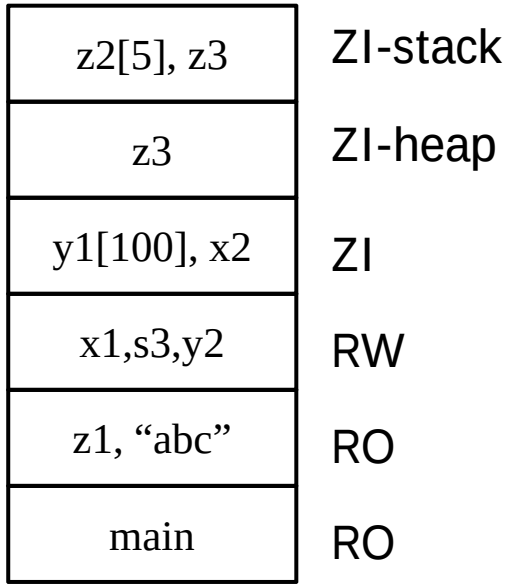
```
int a = 0x10;
void funcA (void)
{
  int a1 = 3;
  int b1 = 4;
  ......
  return;
}

char b = 0x20;
void funcB (void)
{
  .......
  return;
}
```

```
static int d = 0x10;

void func(void)
{
  static int a = 3;
  static int b = 4;

  a++;
  b++;

  return;
}
```

Symbol




```
int x1 = 5;                          // in .data
int y1[100];                         // in .bss
int const z1[3] = {1,2,3};           // in .constdata
char *s3 = "abc";                    // s3 in .data, "abc" in .constdata

int main(int x)                       // in .text
{
   static int x2;                    // in .bss
   static int y2 = 10;               // in .data
   char z2[5];                       // stack
   char z3;                          // stack

   z3 = (char *)malloc(suzeof(char)*200);       // heap
}
```



ELF

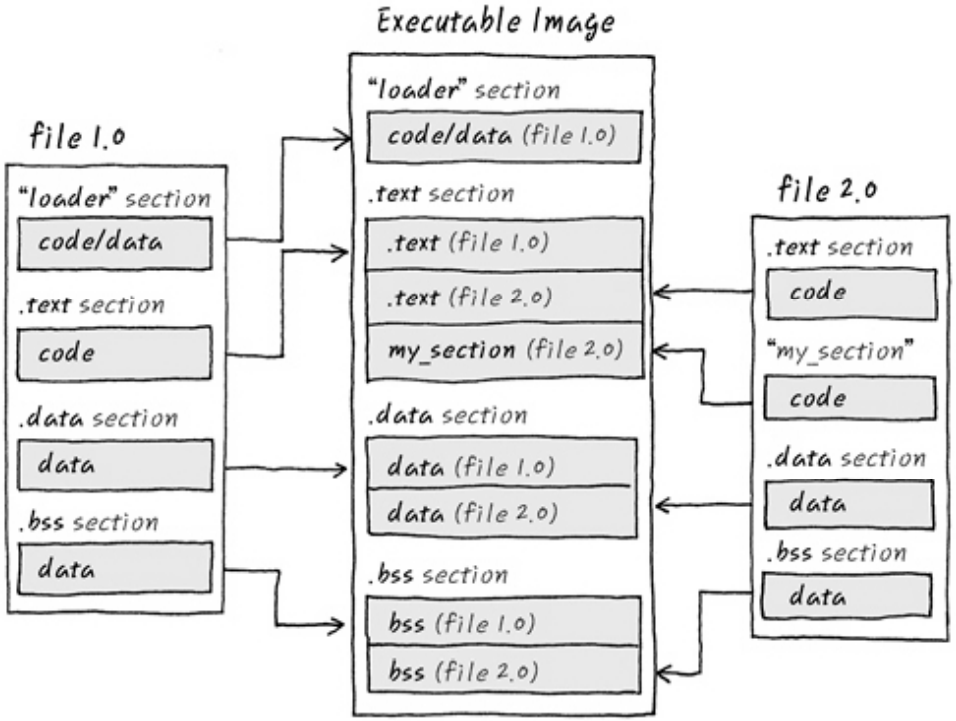
armcc –c test1.c test2.c
armlink –o final.elf test1.o test2.o
fromelf --bin -o outfile.bin infile.axf // convert elf to binary file (w/o symbol)
fromelf -c infile.axf // disassemble
fromelf -c -s -o outfile.lst infile.axf // disassemble & symbol table

axf: DWARF2.0 elf format

Linking View		Execution View
ELF Header		ELF Header
Program Header Table (Optional)		Program Header Table
Section 1		Segment 1
....		
Section n		Segment 2
....		
....		....
Section Header Table		Section Header Table (Optional)

ELF Header
Program Header Table
Text segment
Data segment
BSS segment
".symtab" section
".strtab" section
".shstrtab" section
Debug sections
Section Header Table

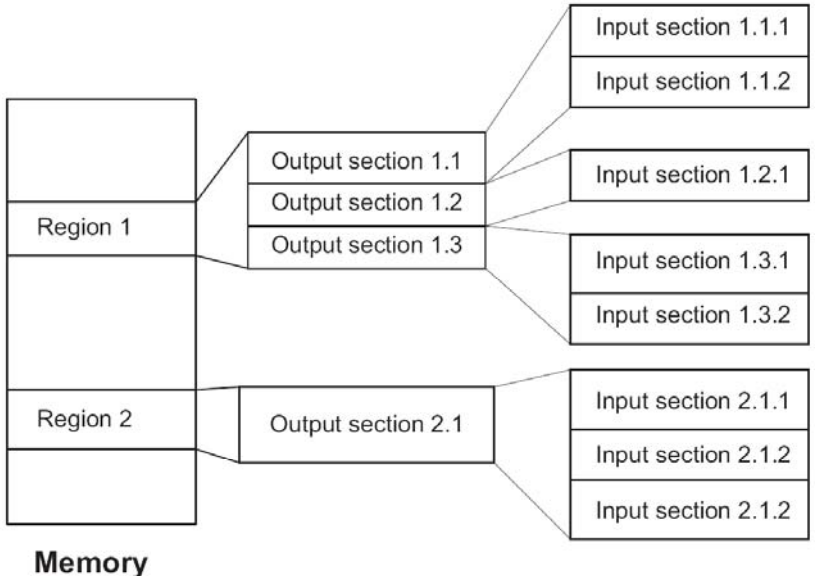
Executable Image



fromelf --bin -o outfile.bin file1.o file2.o

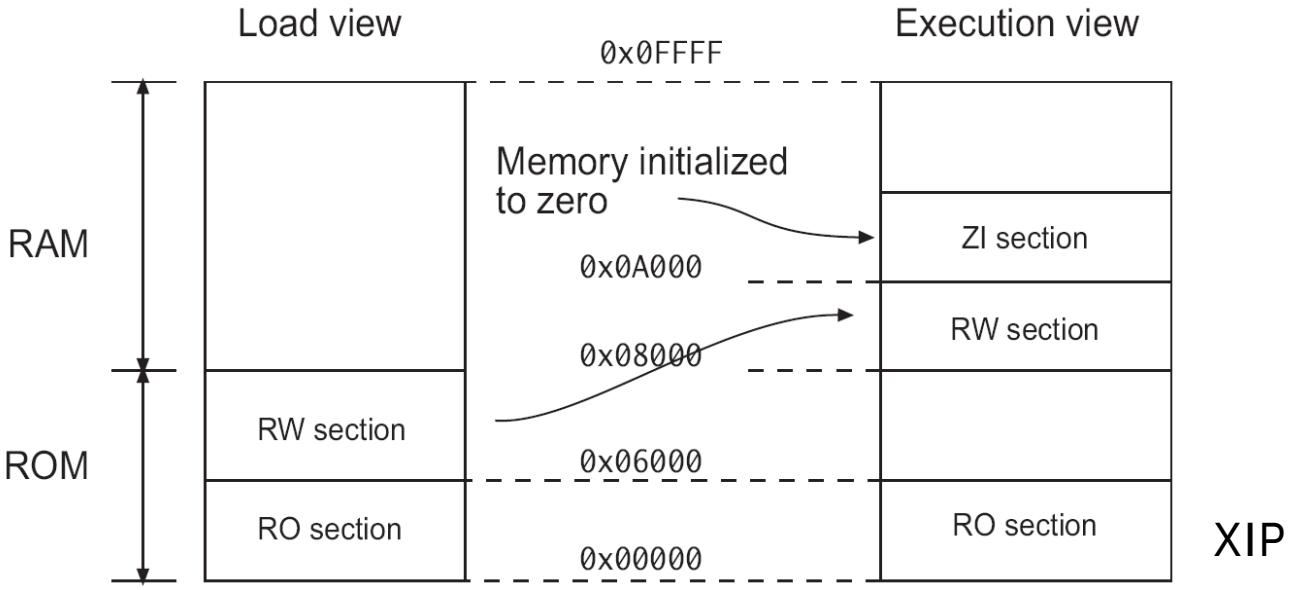
Linker

- Input section can have the attributes RO, RW, or ZI.
- armlink groups input sections into bigger building blocks called output sections and regions.
- An output section is a contiguous sequence of input sections that have the same RO, RW, or ZI attributes.
- A region is a contiguous sequence of one to three output sections.
- A region typically maps onto a physical memory device, such as ROM, RAM, or peripheral.



Load view and execution view

- **Load view**
  - Describes each image region and section in terms of the address it is located at when the image is loaded into memory, that is, the location before the image starts executing.
- **Execution view**
  - Describes each image region and section in terms of the address it is located at while the image is executing.

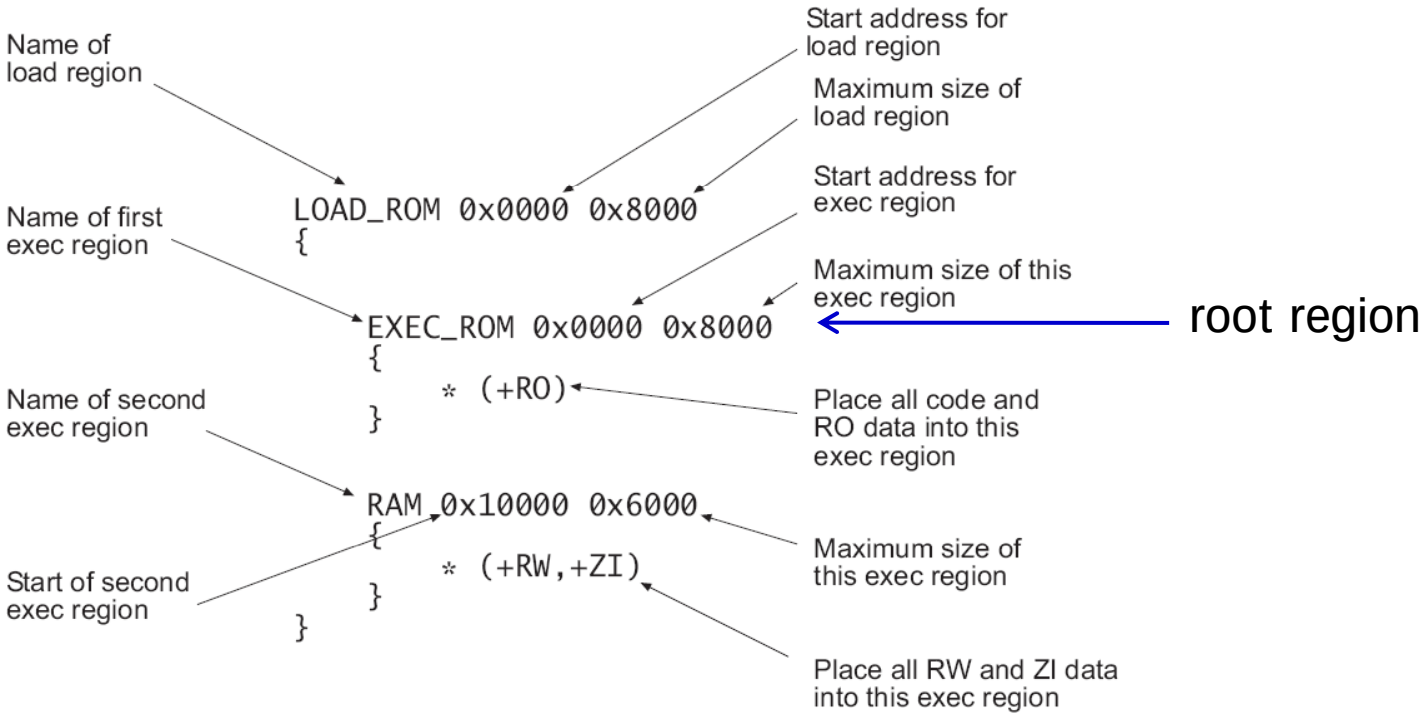


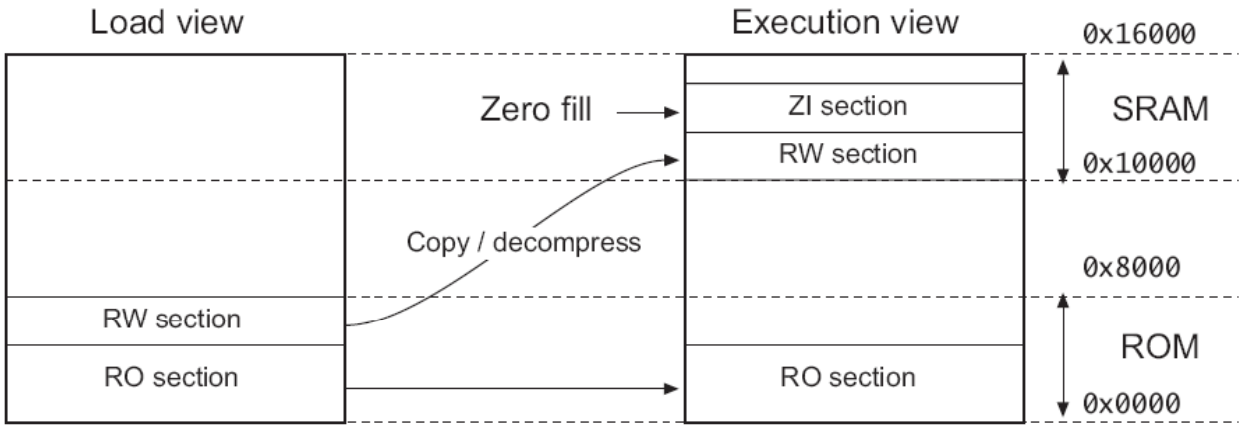
armlink --ro-base 0x0 --rw-base 0x8000

Scatter Loading

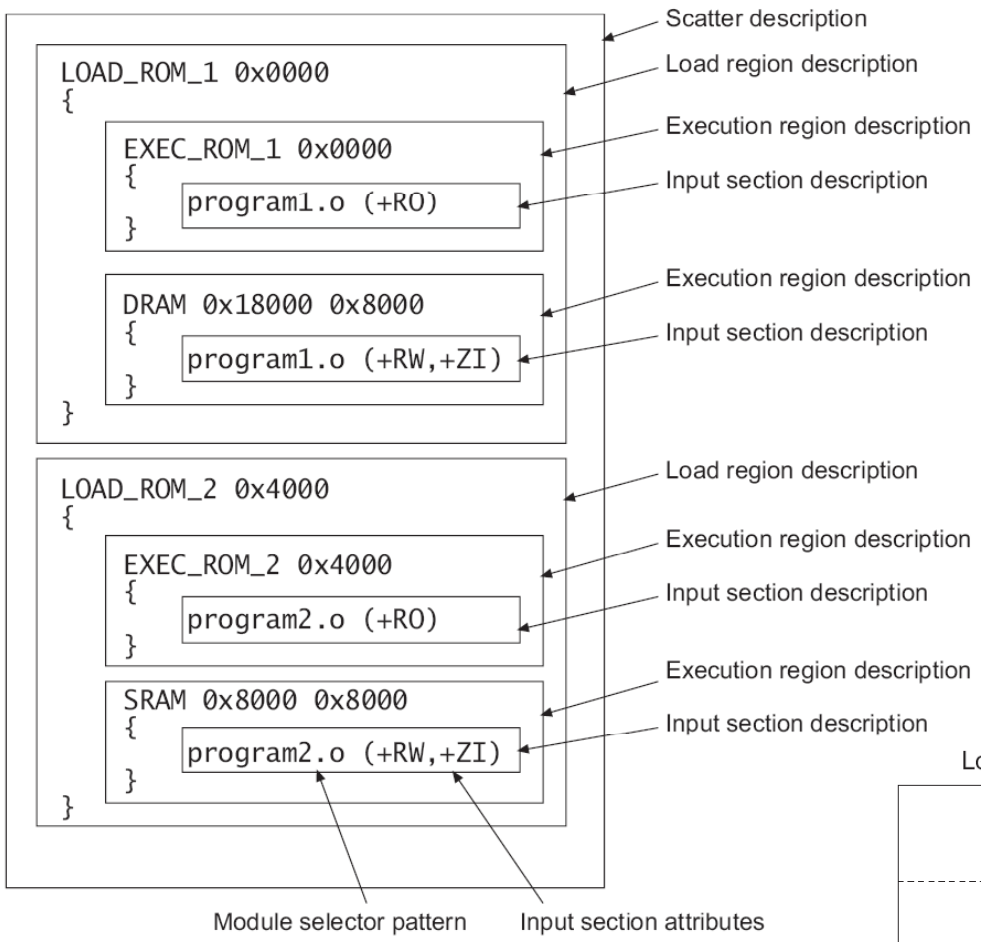
- enables to specify the memory map of an image to the linker using a description in a text file. (armlink --scatter *.scl)
- gives complete control over the grouping and placement of image components.
- When to use scatter-loading
  - Complex memory maps
    - Code and data that must be placed into many distinct areas of memory require detailed instructions on which section goes into which memory space.
  - Different types of memory
    - Many systems contain a variety of physical memory devices such as flash, ROM, SDRAM, and fast SRAM. A scatter-loading description can match the code and data with the most appropriate type of memory. For example, interrupt code might be placed into fast SRAM to improve interrupt response time but infrequently used configuration information might be placed into slower flash memory.
  - Memory-mapped I/O
    - The scatter-loading description can place a data section at a precise address in the memory map so that memory mapped peripherals can be accessed.

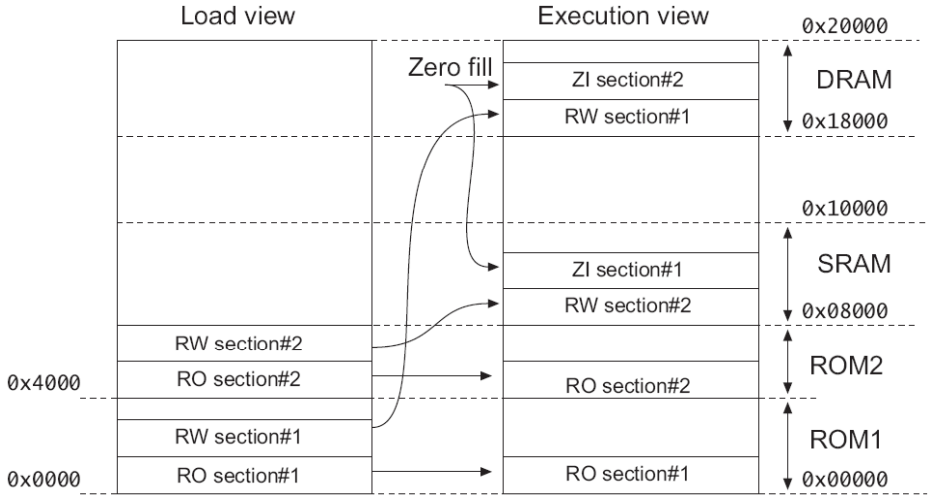
Scatter Loading





Scatter Loading





Overlay

- Use the OVERLAY attribute in a scatter-loading description file to place multiple execution regions at the same address.
- An *overlay manager* is required to make sure that only one execution region is instantiated at a time.
- A region marked as OVERLAY is not initialized by the C library at startup.
- The contents of the memory used by the overlay region are the responsibility of the overlay manager, which must copy any Code and Data, and initialize any ZI when it instantiates a region.

```
EMB_APP 0x8000
{
    .
    .
    STATIC_RAM 0x0                  ; contains most of the RW and ZI code/data
    {
            * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay...
    {
            module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
            module2.o (+RW,+ZI)
    }
    ...                             ; rest of scatter description...
}
```

Accessing linker-defined symbols

- **Region-related symbols**

Symbol	Description
Load$$region_name$$Base	Load address of the region
Image$$region_name$$Base	Execution address of the region
Image$$region_name$$Length	Execution region length in bytes (multiple of 4)
Image$$region_name$$Limit	Address of the byte beyond the end of the execution region
Image$$region_name$$ZI$$Base	Execution address of the ZI output section in this region
Image$$region_name$$ZI$$Length	Length of the ZI output section in bytes (multiple of 4)
Image$$region_name$$ZI$$Limit	Address of the byte beyond the end of the ZI output section in the execution region

```
LDR r0, =||Image$$region_name$$ZI$$Limit||
```

Accessing linker-defined symbols

- **Section-related symbols**

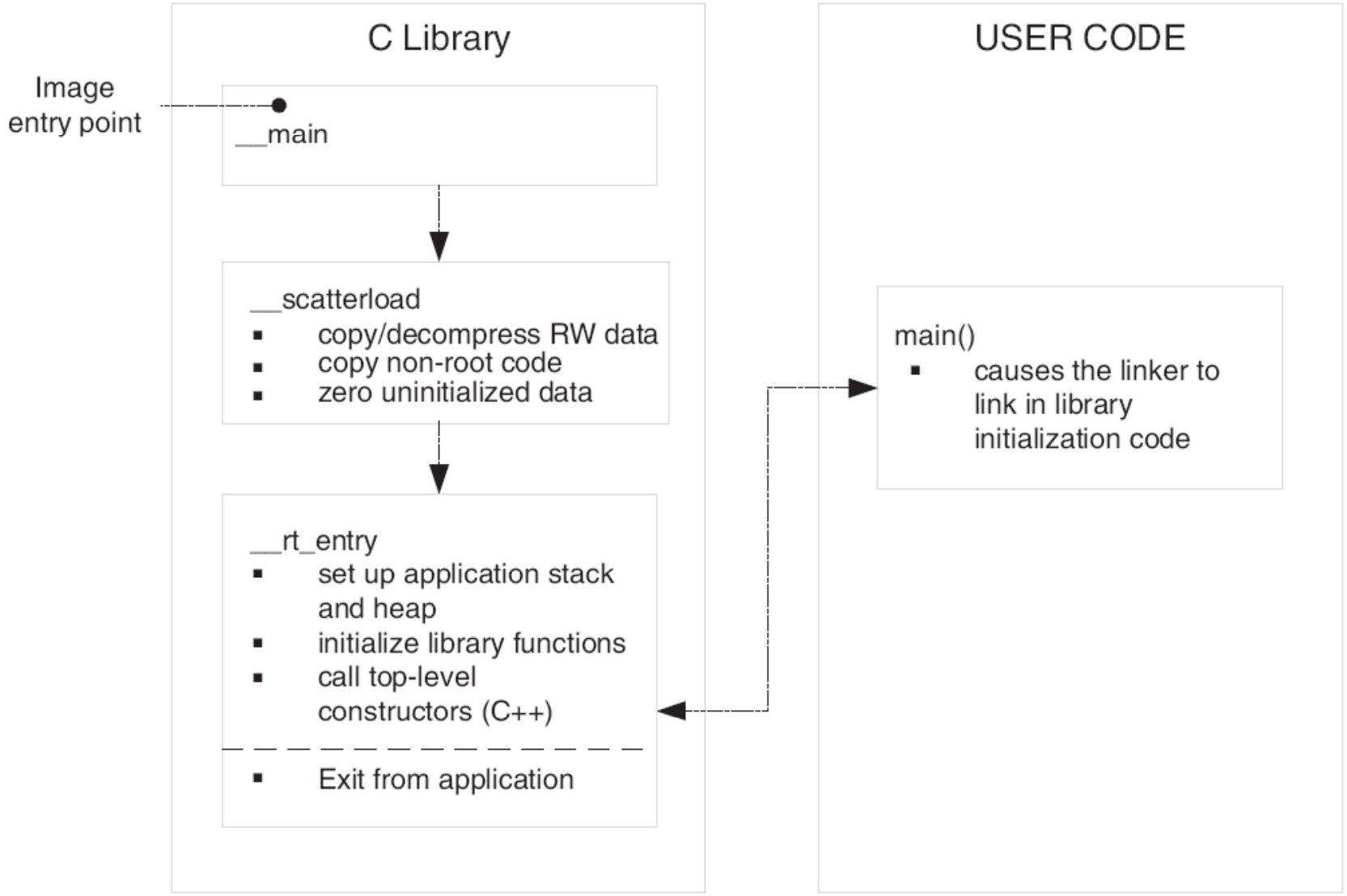
Symbol	Section type	Description
`Image$$RO$$Base`	Output	Address of the start of the RO output section.
`Image$$RO$$Limit`	Output	Address of the first byte beyond the end of the RO output section.
`Image$$RW$$Base`	Output	Address of the start of the RW output section.
`Image$$RW$$Limit`	Output	Address of the byte beyond the end of the ZI output section. (The choice of the end of the ZI region rather than the end of the RW region is to maintain compatibility with legacy code.)
`Image$$ZI$$Base`	Output	Address of the start of the ZI output section.
`Image$$ZI$$Limit`	Output	Address of the byte beyond the end of the ZI output section.
`SectionName$$Base`	Input	Address of the start of the consolidated section called `SectionName`.
`SectionName$$Limit`	Input	Address of the byte beyond the end of the consolidated section called `SectionName`.

Accessing linker-defined symbols

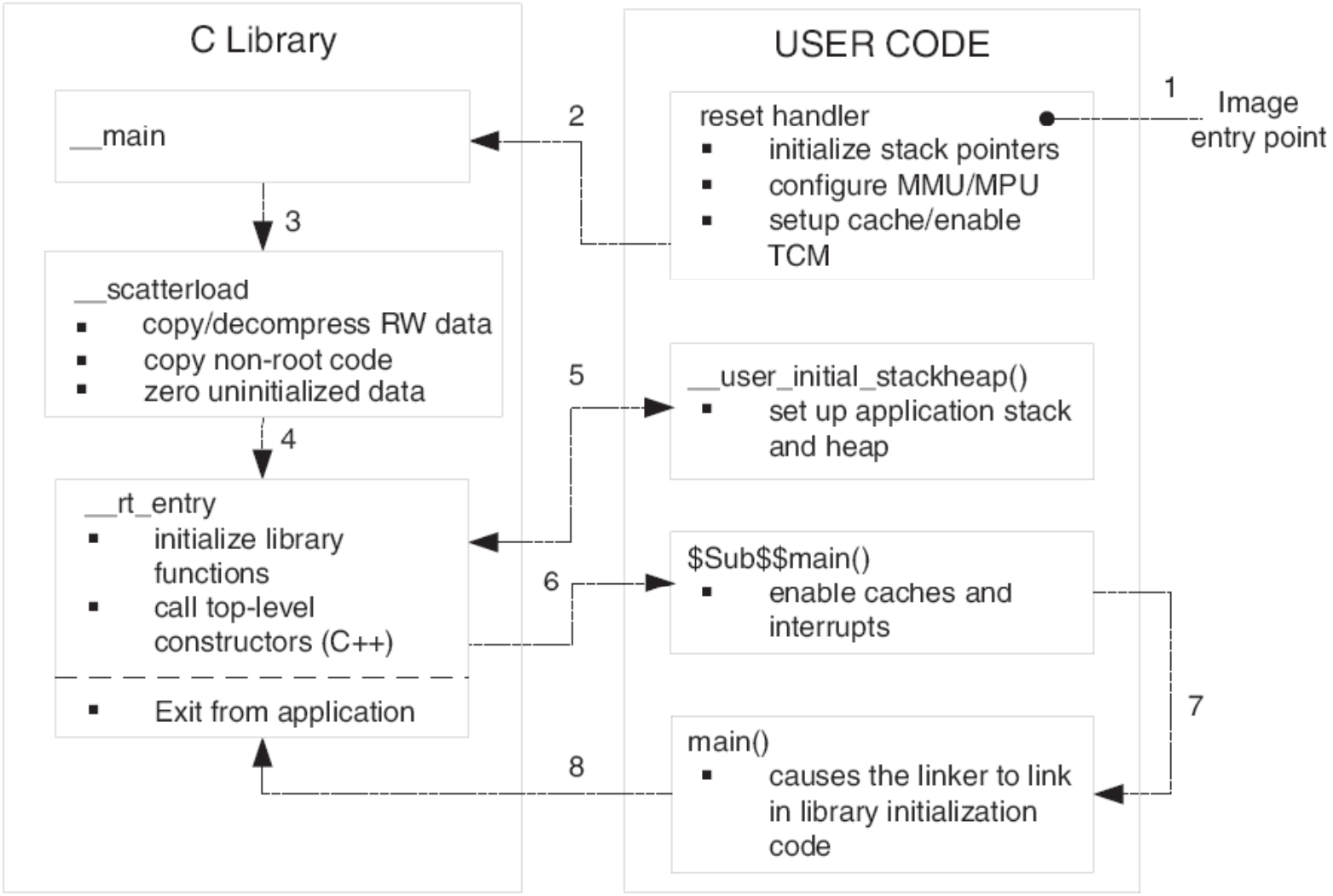
```
extern byte *Image__SRAM__Base;
extern byte *Image__SRAM__Length;
extern byte *Load__SRAM__Base;
extern byte *Image__SRAM__ZI__Base;
extern byte *Image__SRAM__ZI__Length;

end_point = (dword *) ( (dword) Image__SRAM__Base + (dword) Image__SRAM__Length);

for( src = (dword *) Load__SRAM__Base,
       dst = (dword *) Image__SRAM__Base;
     dst < end_point;
     src++, dst++ )
  {
    *dst = *src;
  }
```

```
Load__SRAM__Base
        DCD |Load$$SRAM$$Base|

Image__SRAM__Base
        DCD |Image$$SRAM$$Base|

Image__SRAM__Length
        DCD |Image$$SRAM$$Length|

Image   SRAM   ZI   Base
        DCD |Image$$SRAM$$ZI$$Base|

Image__SRAM__ZI__Length
        DCD |Image$$SRAM$$ZI$$Length|
```

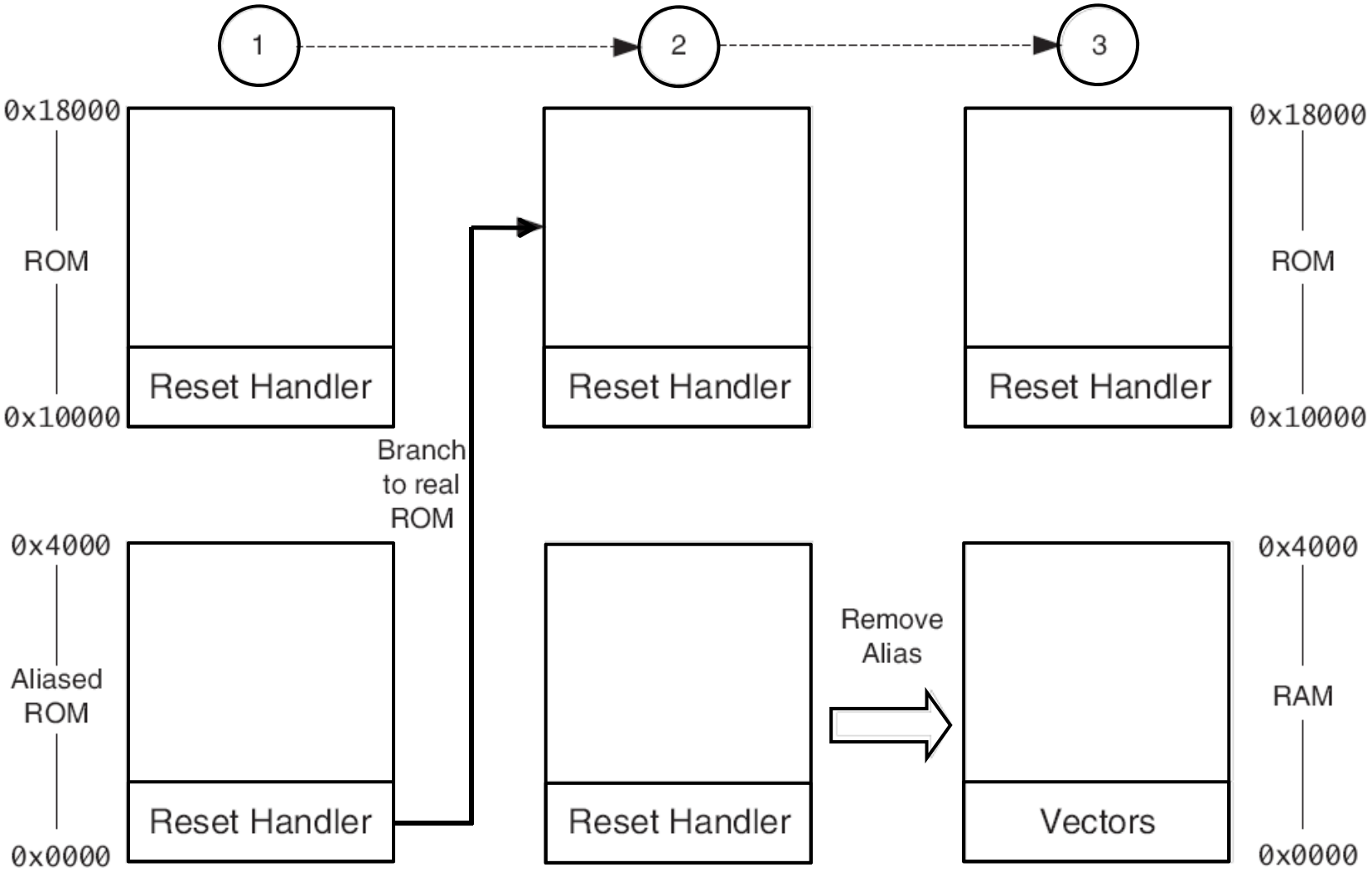
Application Start



Reset and Initialization Sequence



ROM/RAM Remapping



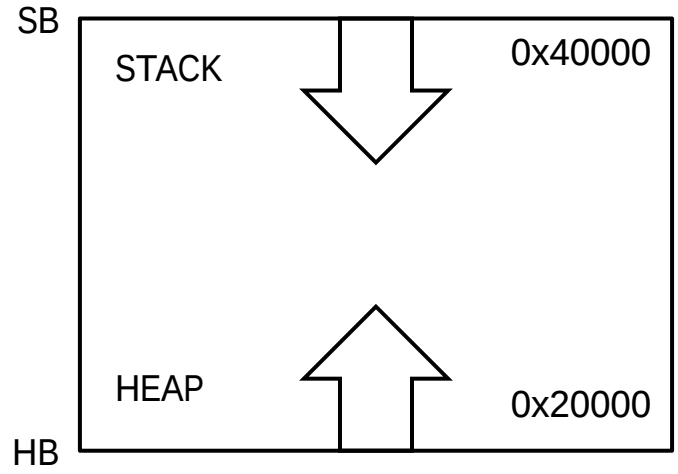
Stack Pointer Initialization

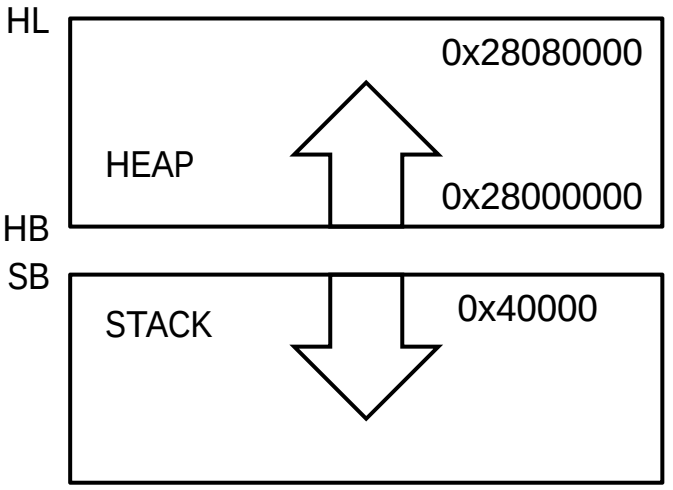
```
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
:
Reset_Handler
:
; stack_base could be defined above, or located in a scatter file
LDR     r0, stack_base ;

; Enter each mode in turn and set up the stack pointer
MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
MOV     sp, r0

MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
SUB     r0, r0, #Len_FIQ_Stack
MOV     sp, r0

MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
SUB     r0, r0, #Len_IRQ_Stack
MOV     sp, r0
; Leave core in SVC mode
```

```
    IMPORT heap_base
    EXPORT __user_initial_stackheap

__user_initial_stackheap

; heap base could be hard-coded, or placed by description file
    LDR   r0,=heap_base
    ; r1 contains SB value
    BX   lr
```

set by C lib init code

__user_initial_stackheap

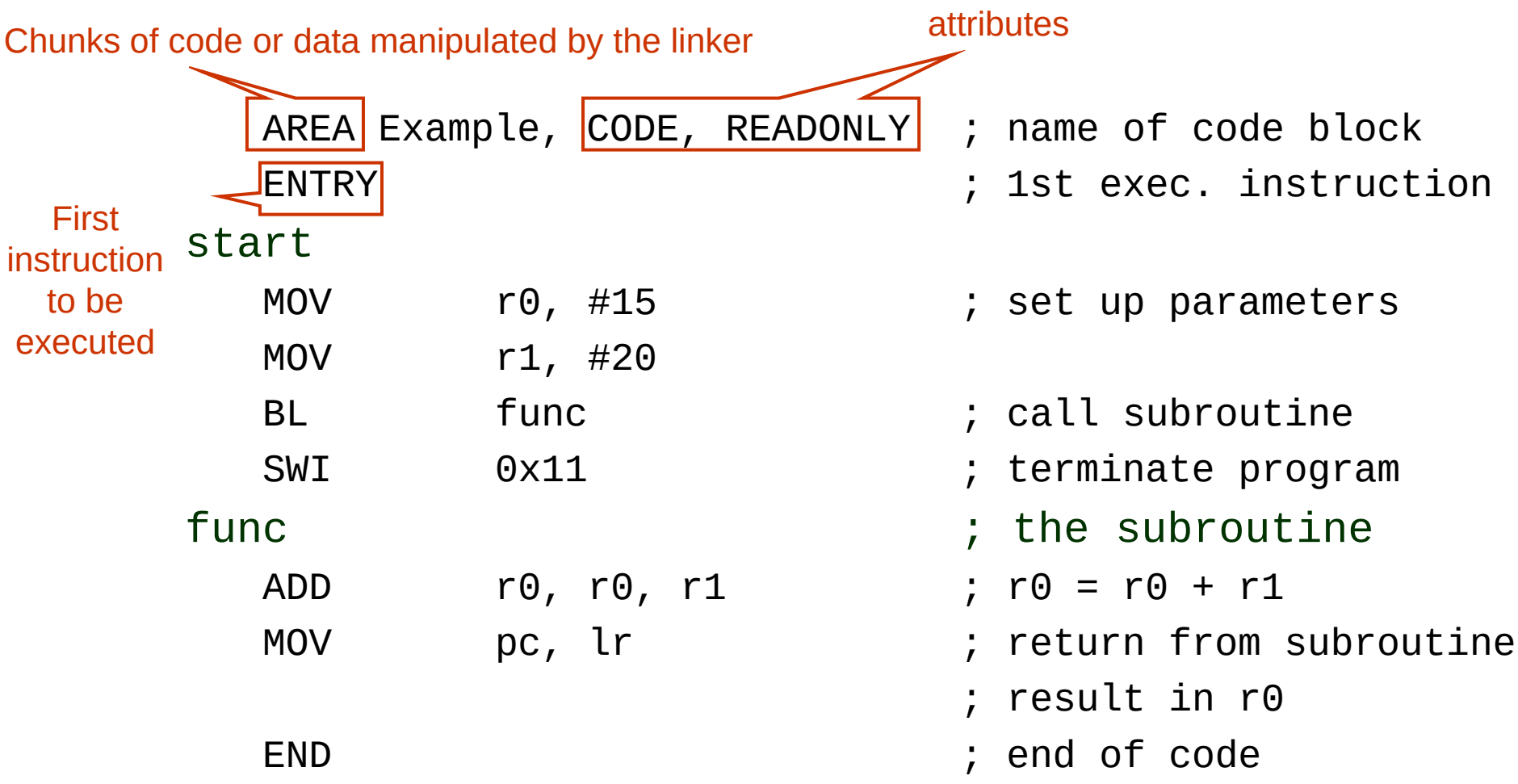





```
    EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR r0, =0x20000 ;HB
    LDR r1, =0x40000 ;SB
    ; r2 not used (HL)
    ; r3 not used
    MOV pc, lr
```

```
    IMPORT __use_two_region_memory
    EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR r0, =0x28000000 ;HB
    LDR r1, =0x40000 ;SB
    LDR r2, =0x28080000 ;HL
    ; r3 not used
    MOV pc, lr
```

The Structure of an Assembler Module




```
    AREA Example, CODE, READONLY  ; name of code block
    ENTRY                         ; 1st exec. instruction
start
    MOV     r0, #15               ; set up parameters
    MOV     r1, #20
    BL      func                  ; call subroutine
    SWI     0x11                  ; terminate program
func                              ; the subroutine
    ADD     r0, r0, r1            ; r0 = r0 + r1
    MOV     pc, lr                ; return from subroutine
                                  ; result in r0
    END                           ; end of code
```

A Simple Program

```
void hello ()
{
  char *data;
  const char text[]="Hello
     world“;
  data = (char *)text;
  printf (" %s ", data);
  return;
}
```

armcc -o helloworld.s -S helloworld.c

```
        CODE32
        AREA ||.text||, CODE, READONLY
hello PROC
|L1.0|
        STMFD    sp!,{r1-r3,lr}
        MOV      r0,sp
        MOV      r2,#0xc
        LDR      r1,|L1.36|
        BL       __rt_memcpy
        MOV      r1,sp
        ADR      r0,|L1.40|
        BL       _printf
        LDMFD    sp!,{r1-r3,pc}
|L1.36|
        DCD      ||.constdata$1||
|L1.40|
        DCB      " %s "
        DCB      "₩0₩0₩0₩0"
        ENDP

        AREA ||.constdata||, DATA, READONLY, ALIGN=0
||.constdata$1||
        DCB      0x48,0x65,0x6c,0x6c
        DCB      0x6f,0x20,0x77,0x6f
        DCB      0x72,0x6c,0x64,0x00

END
```

Inline Assembly

- For multiple instructions on the same line, separate them with a semicolon (;).
- Register names in the inline assembler are treated as C variables. They do not necessarily relate to the physical register of the same name. If you do not declare the register as a C variable, the compiler generates a warning.
- In Thumb mode, only r0-r7
- Do not save and restore registers in inline assembler. The compiler does this for you.
- If registers other than CPSR and SPSR are read without being written to, an error message is issued.
- BL, BLX cannot be used.
- cannot use pseudo instruction such as LDR

```
int f(int x)
{
   int r0;
   __asm
   {
                ADD r0, x, 1
                EOR x, r0, x
   }
   return x;
}
```

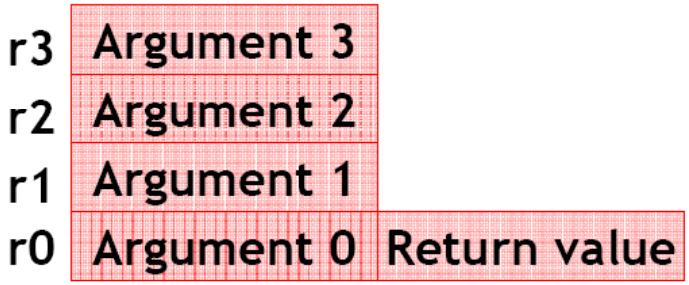
ARM Procedure Call Standard (APCS)

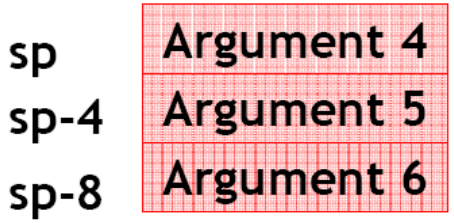
- defines:
  - restrictions on the use of registers
  - conventions for using the stack
  - passing/returning arguments between function calls
  - the format of a stack-based structure which may be 'backtraced' to provide a list of functions (and parameters given) from the failure point backwards to the program entry
  - Compiler option : -apcs

	APCS	APCS Role	reg	APCS	APCS Role
0	a1	argument 1 integer result	8	v5	register variable 5
1	a2	argument 2	9	sb/v6	Static base / register variable 6
2	a3	argument 3	10	sl/v7	stack limit / register variable 7
3	a4	argument 4	11	fp	frame pointer
4	v1	register variable 1	12	ip	scratch reg. / new sb in inter-link-unit calls
5	v2	register variable 2	13	sp	Lower end of current stack frame
6	v3	register variable 3	14	lr	link address
7	v4	register variable 4	15	pc	program counter

ARM Procedure Call Standard (APCS)

- Parameter Passing
  - Passing arguments: core registers (r0-r3) and on the stack
  - For subroutines that take a small number of parameters, only registers are used.
  - Passing argument for long long type: pair of consecutive argument registers (e.g., r0 and r1)
- Return value
  - Integer or pointer: r0
  - Two-word: r0 and r1





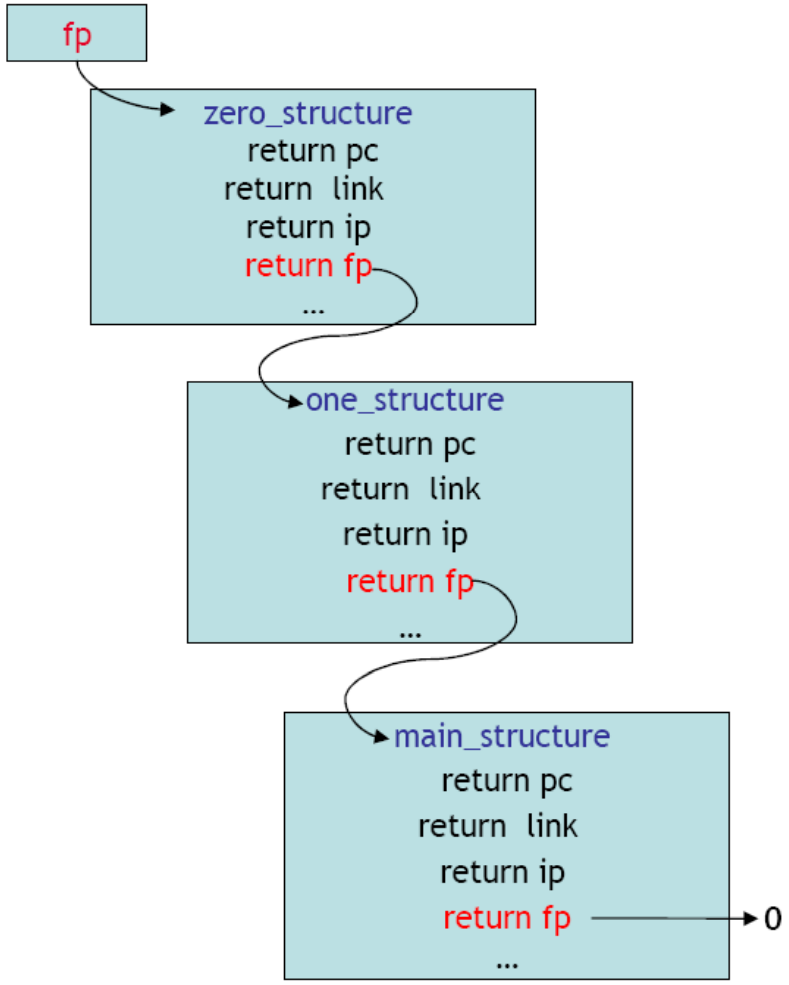
ARM Procedure Call Standard (APCS)

- Stack
  - Linked list of 'frames' which are linked through what is known as a 'backtrace structure‘
  - Stored at the high end of each frame
  - Allocated in descending address order
  - The register sp
    - Point to the lowest used address in the most recent frame.
- Backtrace
  - The register fp (frame pointer) should be zero, or it should point to the last in a list of stack backtrace structures which will provide a means of 'unwinding' the program to trace backwards through the functions called

save code pointer [fp] ◂ – – – – *fp points here*
return ip value [fp, #-4]
return link value [fp, #-8]
return pc value [fp, #-12]
return fp value [fp, #-16] *points to next structure*
[saved other registers]

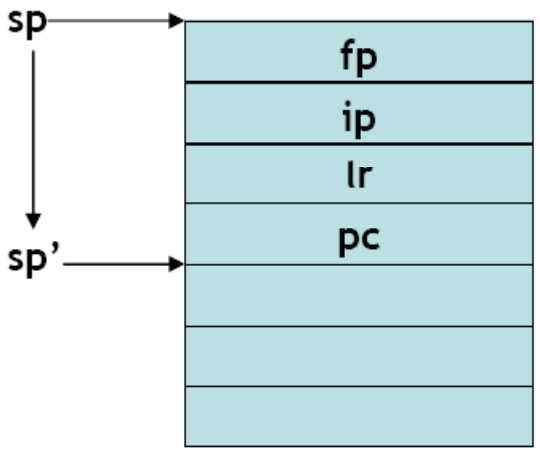
ARM Procedure Call Standard (APCS)

```
int main(void)
{
   one();
   return 0;
}

void one(void)
{
   zero();
   two();
   return;
}

void two(void)
{
   printf("main...one...two\n");
   return;
}

void zero(void)
{
   return;
}
```



ARM Procedure Call Standard (APCS)

```
main:
   mov      ip, sp
   stmfd    sp!, {fp, ip, lr, pc}
   sub      fp, ip, #4
   bl       one
   mov      r0, #0
   b        .L2

one:
  mov       ip, sp
  stmfd     sp!, {fp, ip, lr, pc}
  sub       fp, ip, #4
  bl        zero
  bl        two
  b         .L3

two:
  mov       ip, sp
  stmfd     sp!, {fp, ip, lr, pc}
  sub       fp, ip, #4
  ldr       r0, .L5
  bl        printf
  b         .L4
zero:
  mov       ip, sp
  stmfd     sp!, {fp, ip, lr, pc}
  sub       fp, ip, #4
  b         .L7
```



stmfd sp!, {fp, ip, lr, pc}

Outline of This Lecture

- Profiling
  - Amdahl's Law
  - The 80/20 rule
  - Profiling in the ARM environment

- Improving program performance
  - Standard compiler optimizations
  - Aggressive compiler optimizations
  - Architectural code optimizations

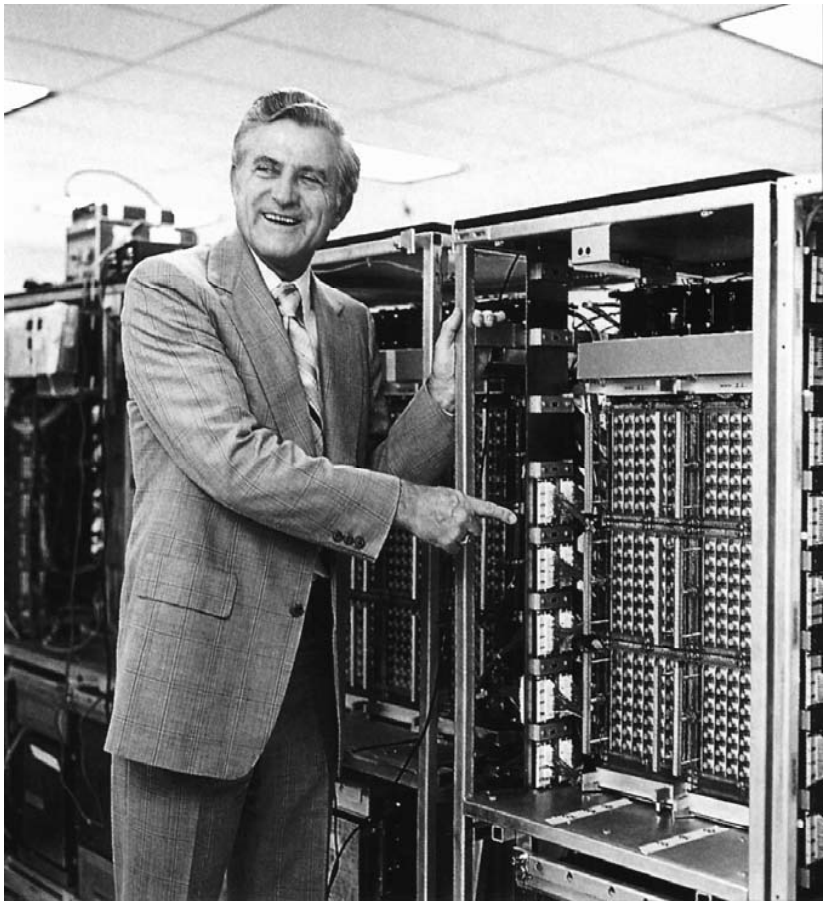
Profiling and Benchmark Analysis

- Problem: You're given a program's source code (which someone else wrote) and asked to improve its performance by at least 20%

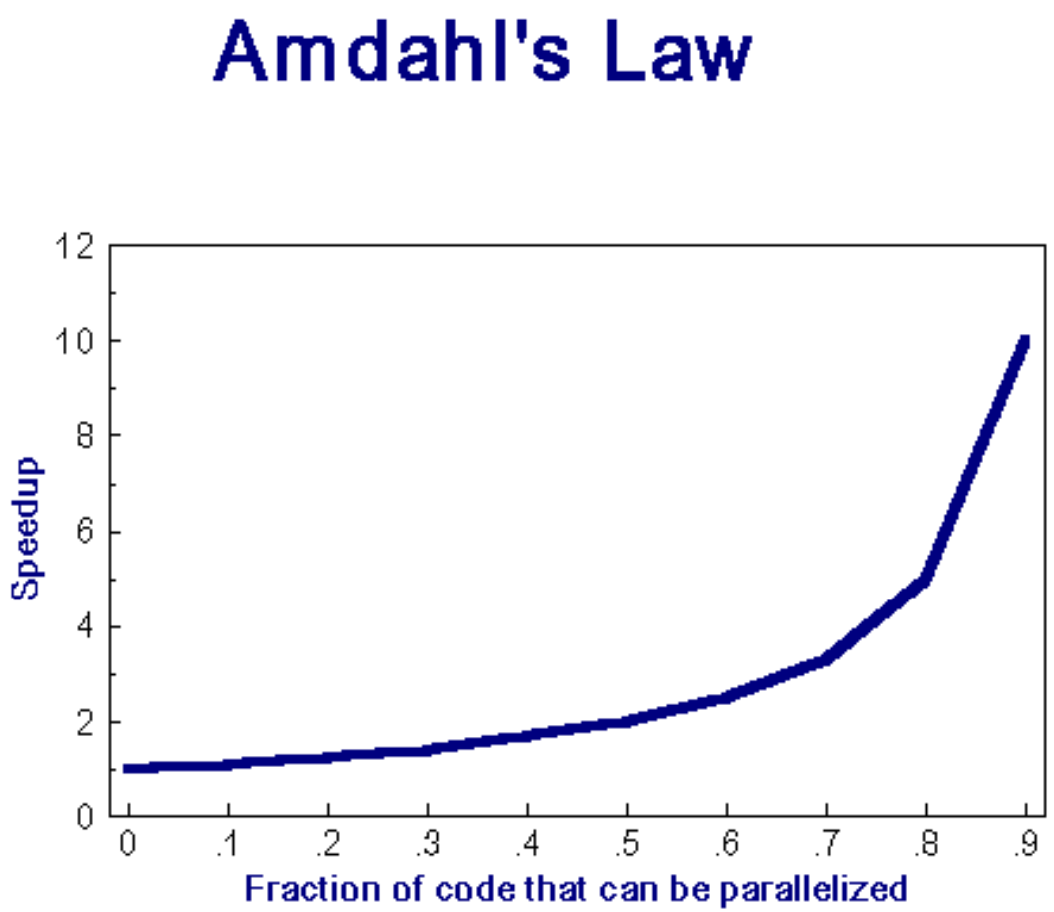
- Where do you begin?
  - Look at source code and try to find inefficient C code
  - Try rewriting some of it in assembly
  - Rewrite using a different algorithm
  - (Remove random portions of the code) ☺

Gene Amdahl

- One of the original architects of the IBM 360 mainframe series
- Founded four companies
  - Amdahl Corporation
  - Trilogy Systems (Part of Elxsi)
  - Andor Systems
  - Commercial Data Servers (CDS)
- A relatively few sequential instructions might have a limiting factor on program speedup such that adding more processors may not make the program run faster.



Amdahl’s Law

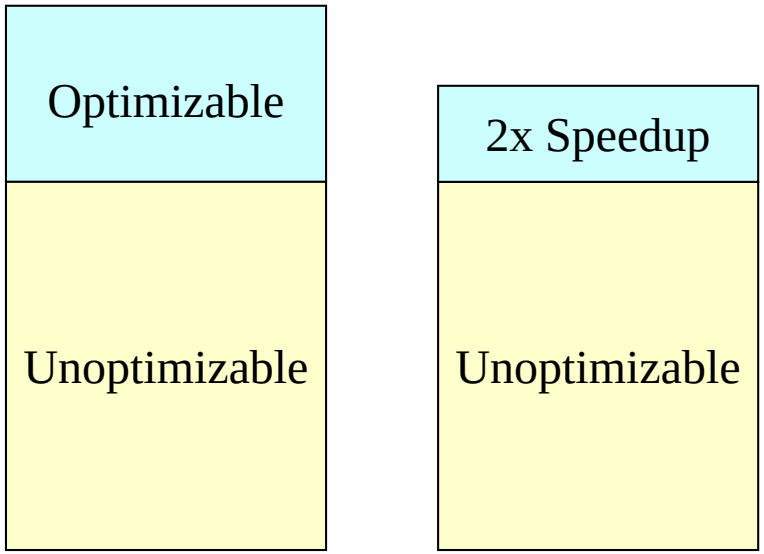

Amdahl's Law
Speedup
0
2
4
6
8
10
12
0
.1
.2
.3
.4
.5
.6
.7
.8
.9
Fraction of code that can be parallelized

Profiling and Benchmark Analysis (cont'd)

- Most important question ...
  - Where is the program spending most of its time?

- Amdahl's Law
  - The performance improvement gained from using some faster mode of execution is limited by the fraction of the total time the faster mode can be used

- Example:



Profiling and Benchmark Analysis (cont'd)

- How do we figure out where a program is spending its time?

  – If we could count *every* static instruction, we would know which routines (functions) were the biggest
    - Big deal, large functions that aren't executed often don't really matter

  – If we could count every dynamic instruction, we would know which routines executed the most instructions
    - Excellent! It tells us the "relative importance" of each function
    - But doesn't account for memory system (stalls)

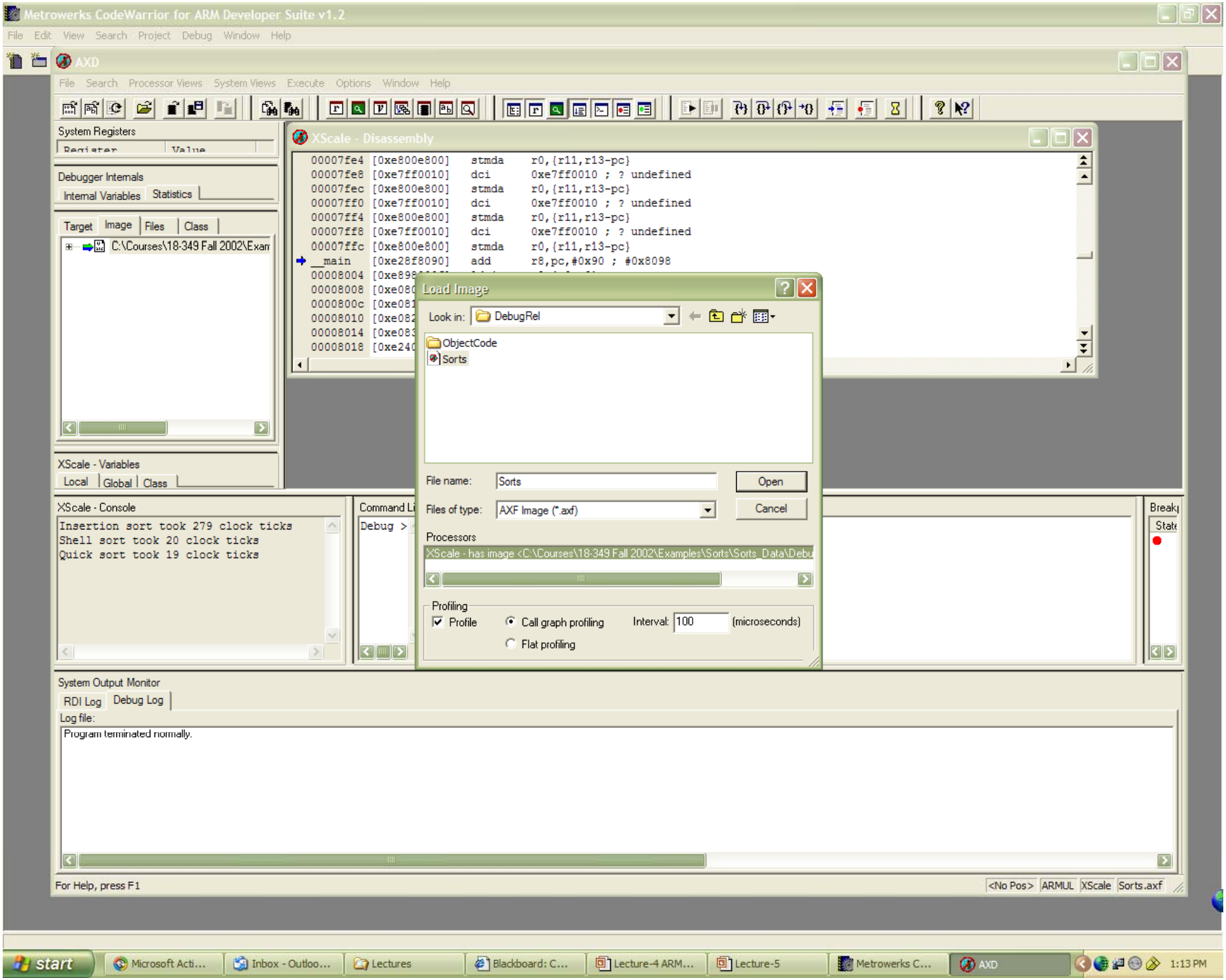
  – If we could count how many cycles were spent in each routine, we would know which routines took the most amount of time

Profiling

- *Profiling*: collecting statistics from example executions
  - Very useful for estimating importance of each routine
  - Common profiling approaches:
    - Instrument all procedure call/return points (expensive: e.g., 20% overhead)
    - Sampling PC every X milliseconds -- so long as program run is significantly longer than the sampling period, the accuracy of profiling is pretty good
  - Usually results in output such as

Routine	% of Execution Time
function_a	60%
function_b	27%
function_c	4%
...	
function_zzz	0.01%

  - Often over 80% of the time spent in less than 20% of the code (80/20 rule)
  - Can now do more accurate profiling with on-chip counters and analysis tools
    - Alpha, Pentium, Pentium Pro, PowerPC
    - DEC Atom analysis tool
    - Both are covered in Advanced Computer Architecture courses


Metrowerks CodeWarrior for ARM Developer Suite v1.2
File Edit View Search Project Debug Window Help
AXD
File Search Processor Views System Views Execute Options Window Help
System Registers
Register Value
Debugger Internals
Internal Variables Statistics
Target Image Files Class
C:\Courses\18-349 Fall 2002\Exam
XScale - Variables
Local Global Class
XScale - Disassembly
00007fe4 [0xe800e800] stmda r0,{r11,r13-pc}
00007fe8 [0xe7ff0010] dci 0xe7ff0010 ; ? undefined
00007fec [0xe800e800] stmda r0,{r11,r13-pc}
00007ff0 [0xe7ff0010] dci 0xe7ff0010 ; ? undefined
00007ff4 [0xe800e800] stmda r0,{r11,r13-pc}
00007ff8 [0xe7ff0010] dci 0xe7ff0010 ; ? undefined
00007ffc [0xe800e800] stmda r0,{r11,r13-pc}
__main [0xe28f8090] add r8,pc,#0x90 ; #0x8098
00008004
00008008
0000800c
00008010
00008014
00008018
Load Image
Look in: DebugRel
ObjectCode
Sorts
File name: Sorts
Open
Files of type: AXF Image (*.axf)
Cancel
Processors
XScale - has image <C:\Courses\18-349 Fall 2002\Examples\Sorts\Sorts_Data\Debu
Profiling
Profile
Call graph profiling
Interval: 100 (microseconds)
Flat profiling
XScale - Console
Insertion sort took 279 clock ticks
Shell sort took 20 clock ticks
Quick sort took 19 clock ticks
Command Li
Debug >
Break
State
System Output Monitor
RDI Log Debug Log
Log file:
Program terminated normally.
For Help, press F1
<No Pos> ARMUL XScale Sorts.axf
start
Microsoft Acti...
Inbox - Outloo...
Lectures
Blackboard: C...
Lecture-4 ARM...
Lecture-5
Metrowerks C...
AXD
1:13 PM

Timing execution with `armsd`

- The simulator simulates *every* cycle
  - Can gather *very* accurate timings for each function
- Run the simulator to determine total time
- Compiler can optimize for speed

```
prompt> armcc -Otime -o sort sorts.c
```

- Can also optimize for size

```
prompt> armcc -Ospace -o sort sorts.c
```

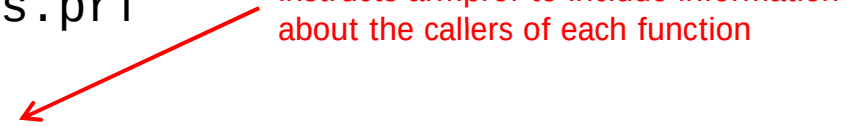
- Re-run the simulator to determine new total time
  - new time is 2,059,629 μsecs -- an improvement of 4.5% (compared to -g)

Profiling with `armsd`

- *No* compile-time options needed

- Run the simulator to profile, capturing callgraph data

```
prompt> armsd
armsd: load/callgraph sorts
armsd: ProfOn
armsd: go
armsd: ProfWrite sorts.prf
armsd: quit
prompt> armprof -Parent sorts.prf > profile
```



- To profile for only samples, skip the "`/callgraph`" portion
  - avoids the 20% overhead (in *this* example)

armprof output

time spent in this function

time spent in children of the current function

```
Name                    cum%     self%     desc%     calls
main                    96.4%    0.16%     95.88%    0
   qsort                         0.44%     0.75%     1
   _printf                       0.00%     0.00%     3
   clock                         0.00%     0.00%     6
   _sprintf                      0.34%     3.56%     1000
   randomise                     0.12%     0.69%     1
   hell_sort                     1.59%     3.43%     1
   insert_sort                   19.91%    59.44%    1
-------------------------------------------------------------
main                             19.91%    59.44%    1
insert_sort             79.35%   19.91%    59.44%    1
strcmp                           59.44%    0.00%     243432
-------------------------------------------------------------
qs_string_compare                3.17%     0.00%     13021
shell_sort                       3.43%     0.00%     14059
insert_sort                      59.44%    0.00%     243432
strcmp                  66.05%   66.05%    0.00%     270512
```

Optimizing "sorts"

- Almost 60% of time spent in `strcmp` called by `insert_sort`
- `strcmp` compares two strings and returns int
  - 0 if equal, negative if first is ``less than'' second, positive otherwise
- Replace "`strcmp(a,b)`" call with some *initial* compares

```
if (a[0] < b[0]) {
    result is neg
}
if (a[0] == b[0]) {
    if (a[1] < b[1]) {
        result is neg
    }
    if (a[1] == b[1]) {
        if (strcmp(a,b) <= 0) {
            result is neg or zero
        }
    }
}
```

- Result of this change is 20% reduction in execution time
  - Avoids some procedure call overheads (in-lining)
  - Avoids some loop control overheads (loop unrolling)
  - Handles common cases efficiently and other cases correctly

Improving Program Performance

- Compiler writers try to apply several standard optimizations
  - Do **not** always succeed
- Compiler writers sometimes apply aggressive optimizations
  - Often not “informed” enough to know that change will help rather than hurt
- Optimizations based on specific architecture/implementation characteristics can be very helpful
  - Much harder for compiler writers because it requires multiple, generally very different, “back-end” implementations
- How can one help?
  - Better code, algorithms and data structures (of course)
  - Re-organize code to help compiler find opportunities for improvement
  - Replace poorly optimized code with assembly code (i.e., bypass compiler)

Standard Compiler Optimizations

- Common Sub-expression Elimination
  - Formally, "An occurrence of an expression E is called a *common sub-expression* if E was previously computed, and the values of variables in E have not changed since the previous computation."
  - You can avoid re-computing the expression if we can use the previously computed one.
  - **Benefit**: less code to be executed

Before

```
b:
        t6 = 4 * i
        x  = a[t6]
        t7 = 4 * i
        t8 = 4 * j
        t9 = a[t8]
        a[t7]  = t9
        t10    = 4 * j
        a[t10] = x
        goto b
```

After

```
b:
        t6 = 4* i
        x  = a[t6]
        t8 = 4 * j
        t9 = a[t8]
        a[t6] = t9
        a[t8] = x
        goto b
```

Standard Compiler Optimizations

- Dead-Code Elimination
  - If code is definitely *not* going to be executed during *any* run of a program, then it is called dead code and can be removed.
  - Example:

```
debug = 0;
...
if (debug){
   print .....
}
```

  - **You** can help by using `ASSERTs` and `#ifdefs` to tell the compiler about dead code
    - It is often difficult for the compiler to identify dead code itself

Standard Compiler Optimizations (con't)

- Induction Variables and Strength Reduction
  - A variable X is called an induction variable of a loop L if every time the variable X changed value, it is incremented or decremented by some constant
  - When there are 2 or more induction variables in a loop, it may be possible to get rid of all but one
  - It is also frequently possible to perform strength reduction on induction variables
    - the strength of an instruction corresponds to its execution cost
  - Benefit: fewer and less expensive operations

```
    t4 = 0
label_XXX
    j  = j + 1
    t4 = 4 * j
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

Before

```
    t4 = 0
label_XXX
    t4 += 4
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

After

Aggressive Compiler Optimizations

- In-lining of functions
  - Replacing a call to a function with the function's code is called “in-lining”
  - Benefit: reduction in procedure call overheads and opportunity for additional code optimizations
  - Danger: code bloat and negative instruction cache effects
  - Appropriate when small and/or called from a small number of sites

```
    MOV     r0, r4          ; r4 --> r0 (param 1)
    MOV     r1, #4          ; 4  --> r1 (param 2)
    BL      c_add           ; call c_add
    MOV     r5, r0          ; r0 (result) --> r5
    SWI     0x11            ; terminate
c_add
    MOV     r12, r13        ; save sp
    STMDB   r13!, {r0,r1,r11,r12,r14,pc} ; save regs
    SUB     r11, r12, #4    ; (sp - 4) --> r11
    MOV     r2, r0          ; param 1 --> r2
    ADD     r3, r2, r1      ; param 1 + param 2 --> r3
    MOV     r0, r3          ; move result to r0
    LDMDB   r11, {r11, r13, pc} ; restore regs
```

Before

```
ADD  r5, r4, #4
SWI  0x11
```

After

Aggressive Compiler Optimizations (2)

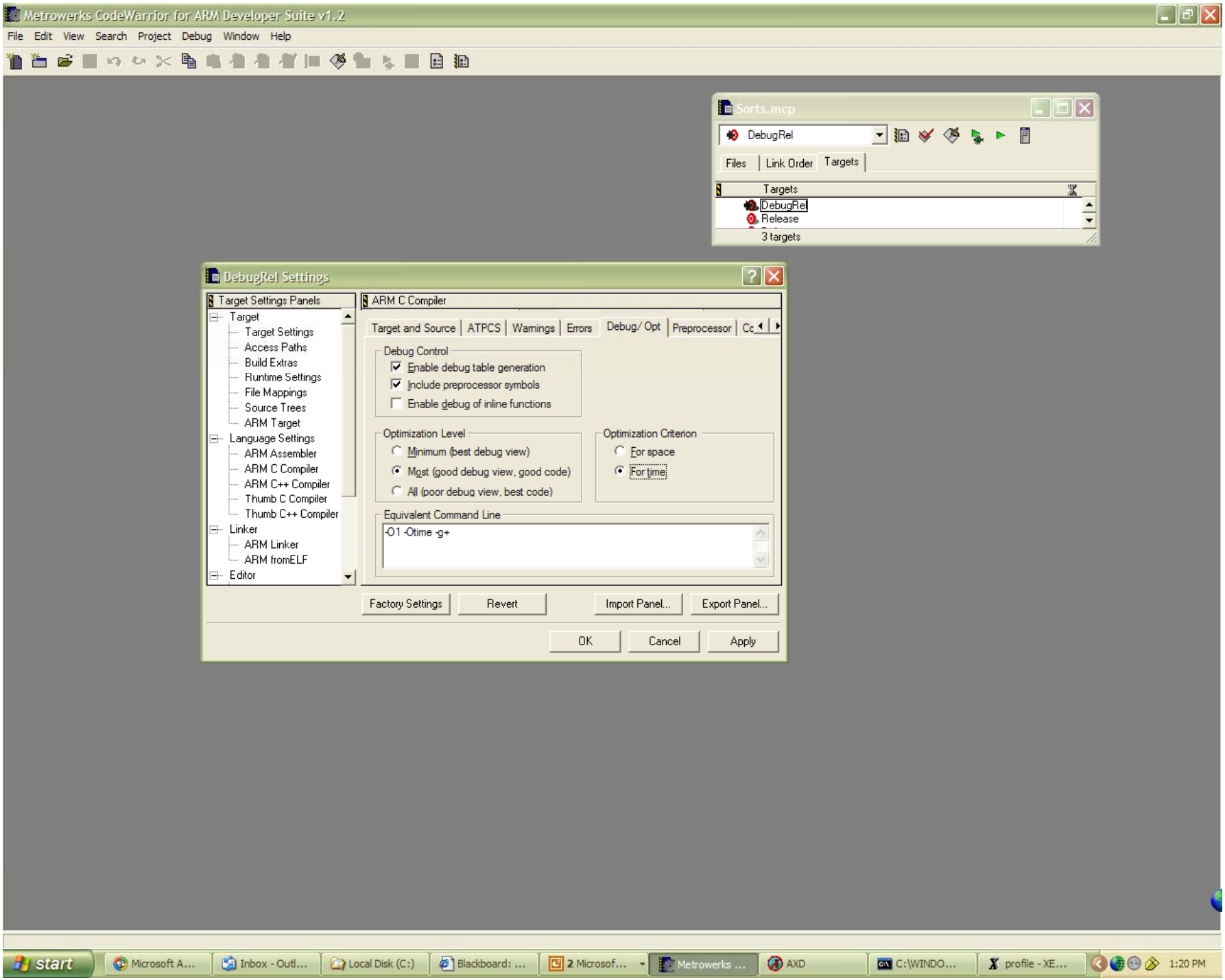
- Loop Unrolling
  - Doing multiple iterations of work in each iteration is called “loop unrolling”
  - Benefit: reduction in looping overheads and opportunity for more code opts.
  - **Danger**: code bloat, negative instruction cache effects, and *non-integral loop div*.
  - Appropriate when small and/or called from small number of sites

Bit-counting loop	Unrolled bit-counting loop

int countbit1(unsigned int n)
{
  int bits = 0;
  while (n != 0)
  {
  if (n & 1) bits++;
  n >>= 1;
  }
  return bits;
}
	int countbit2(unsigned int n)
	{
	  int bits = 0;
	  while (n != 0)
	  {
	  if (n & 1) bits++;
	  if (n & 2) bits++;
	  if (n & 4) bits++;
	  if (n & 8) bits++;
	  n >>= 4;
	  }
	  return bits;
	}

Bit-counting loop	Unrolled bit-counting loop

countbit1 PROC
  MOV r1, #0
  B |L1.20|
|L1.8|
  TST r0, #1
  ADDNE r1, r1, #1
  LSR r0, r0, #1
|L1.20|
  CMP r0, #0
  BNE |L1.8|
  MOV r0, r1
  BX lr
ENDP
	countbit2 PROC
	  MOV r1, r0
	  MOV r0, #0
	  B |L1.48|
	|L1.12|
	  TST r1, #1
	  ADDNE r0, r0, #1
	  TST r1, #2
	  ADDNE r0, r0, #1
	  TST r1, #4
	  ADDNE r0, r0, #1
	  TST r1, #8
	  ADDNE r0, r0, #1
	  LSR r1, r1, #4
	|L1.48|
	  CMP r1, #0
	  BNE |L1.12|
	  BX lr ENDP


Metrowerks CodeWarrior for ARM Developer Suite v1.2
File Edit View Search Project Debug Window Help
Sorts.mcp
DebugRel
Files Link Order Targets
Targets
DebugRel
Release
3 targets
DebugRel Settings
Target Settings Panels
Target
Target Settings
Access Paths
Build Extras
Runtime Settings
File Mappings
Source Trees
ARM Target
Language Settings
ARM Assembler
ARM C Compiler
ARM C++ Compiler
Thumb C Compiler
Thumb C++ Compiler
Linker
ARM Linker
ARM fromELF
Editor
ARM C Compiler
Target and Source ATPCS Warnings Errors Debug/Opt Preprocessor
Debug Control
Enable debug table generation
Include preprocessor symbols
Enable debug of inline functions
Optimization Level
Minimum (best debug view)
Most (good debug view, good code)
All (poor debug view, best code)
Optimization Criterion
For space
For time
Equivalent Command Line
-O1 -Otime -g+
Factory Settings
Revert
Import Panel...
Export Panel...
OK
Cancel
Apply
start
Microsoft A...
Inbox - Outl...
Local Disk (C:)
Blackboard: ...
2 Microsof...
Metrowerks ...
AXD
C:\WINDO...
profile - XE...
1:20 PM