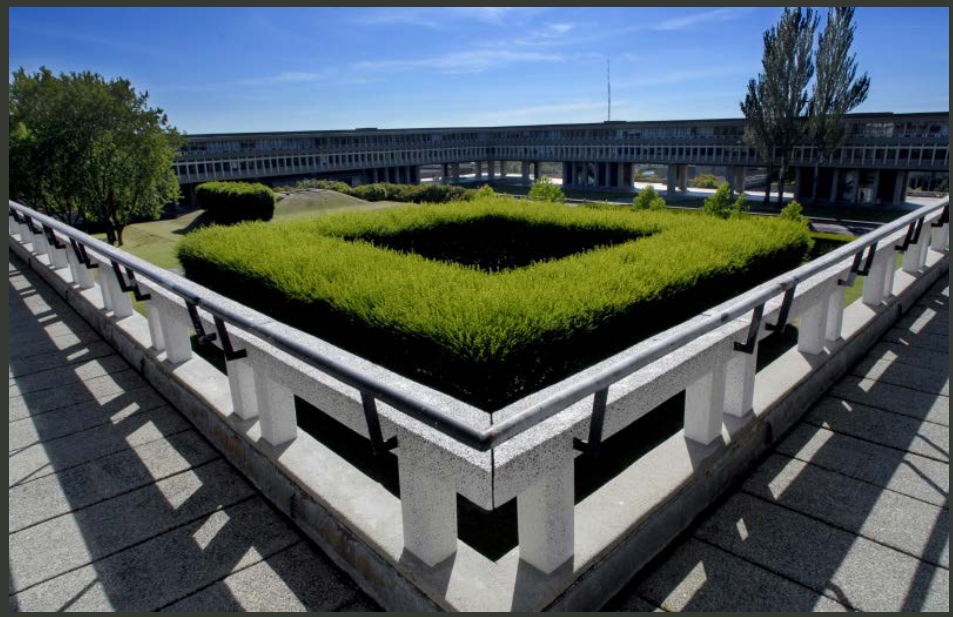
SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Assembly Rules and Directives

June 2020

© 2017 -- Fabio Campi and Craig Scratchley
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada



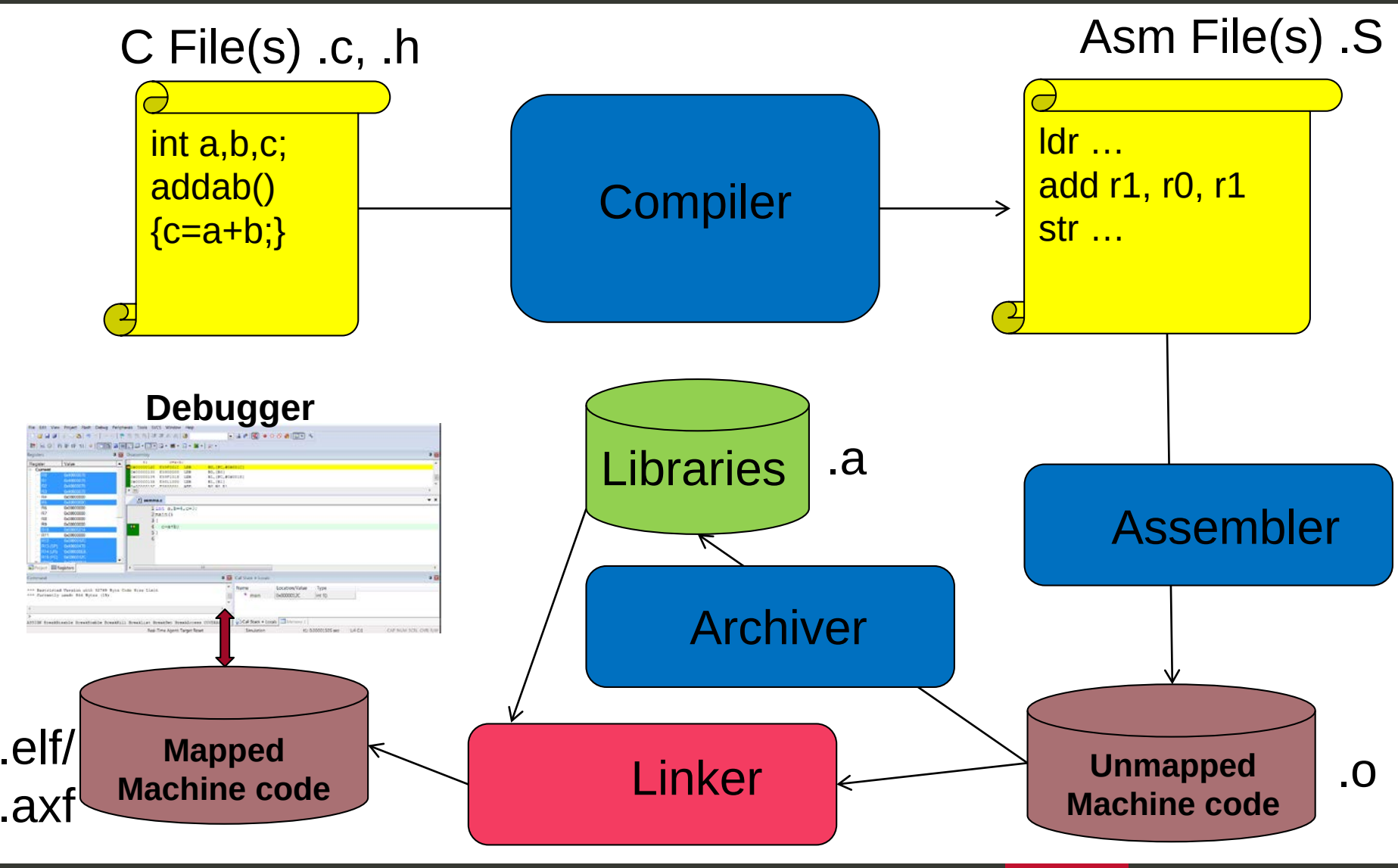
Last Lecture

What you need to remember:

- Cross-Compilation ToolFlow for embedded applications
- Role of the Linker, role of the assembler

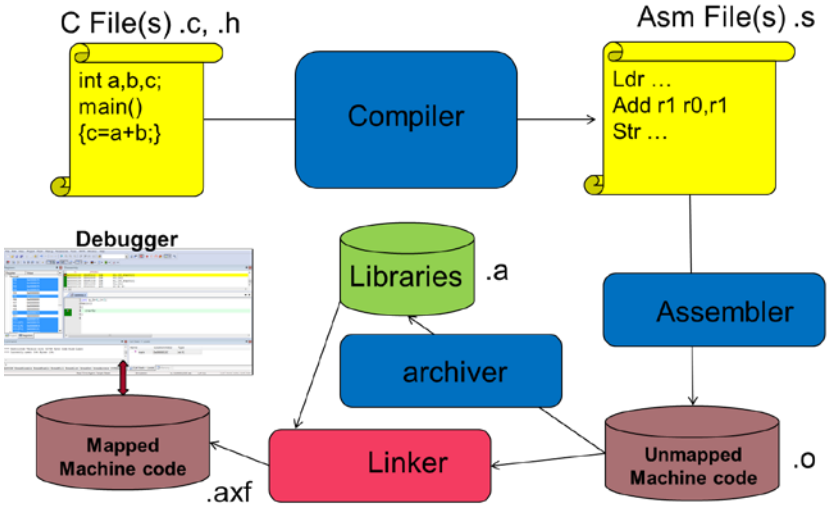
- What is the ARM Current Program Status Register (CPSR) and in particular Status Flags
- What is ARM Program Counter (R15)
- What is ARM Link Register (R14)

Compilation Flow



The role of the Compiler and of the Linker

In a program compilation flow, the most critical task is that of the ***COMPILER***, that will break our high-level C description into low-level assembly instructions. There are many possible ways to render the same C code in assembly, and a good compiler can make an enormous difference in terms of optimization, both for code size and computation speed.

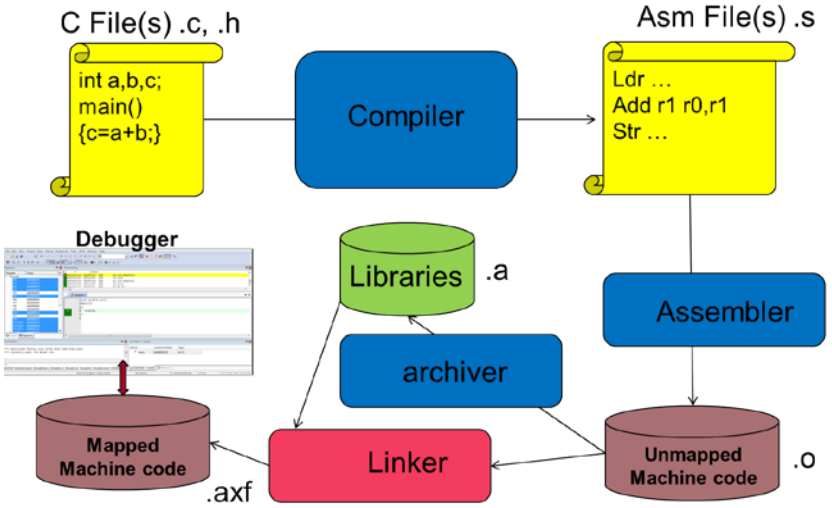


The task of the ***LINKER***, too, is very critical. That is, mapping all labels and addresses to actual memory locations, based on the specifications of the user. But this task is mechanical, there is no space for optimization: the linker simply does what it needs to do.

The role of the Assembler

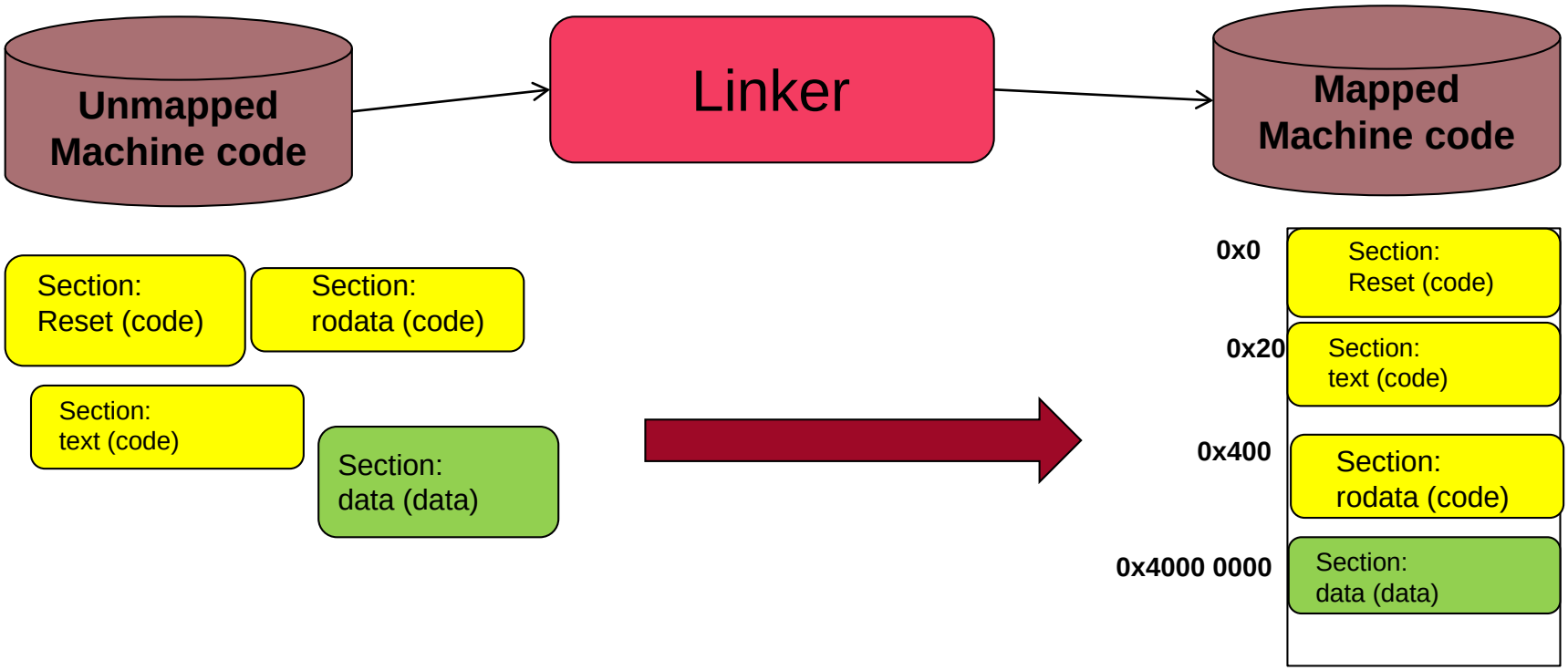
The role of the Assembler is partly that of *translating the assembly symbolic instructions into their equivalent machine language.*
The task is mostly mechanical and there is very little space for interpretation or optimization.

The most challenging part of using an ASSEMBLER is not learning machine-level instructions, but understanding specific DIRECTIVES that can be used to instruct the Assembler and the Linker on how to handle specific situations, mostly in the mapping of your code/data to memory.



Memory Sections

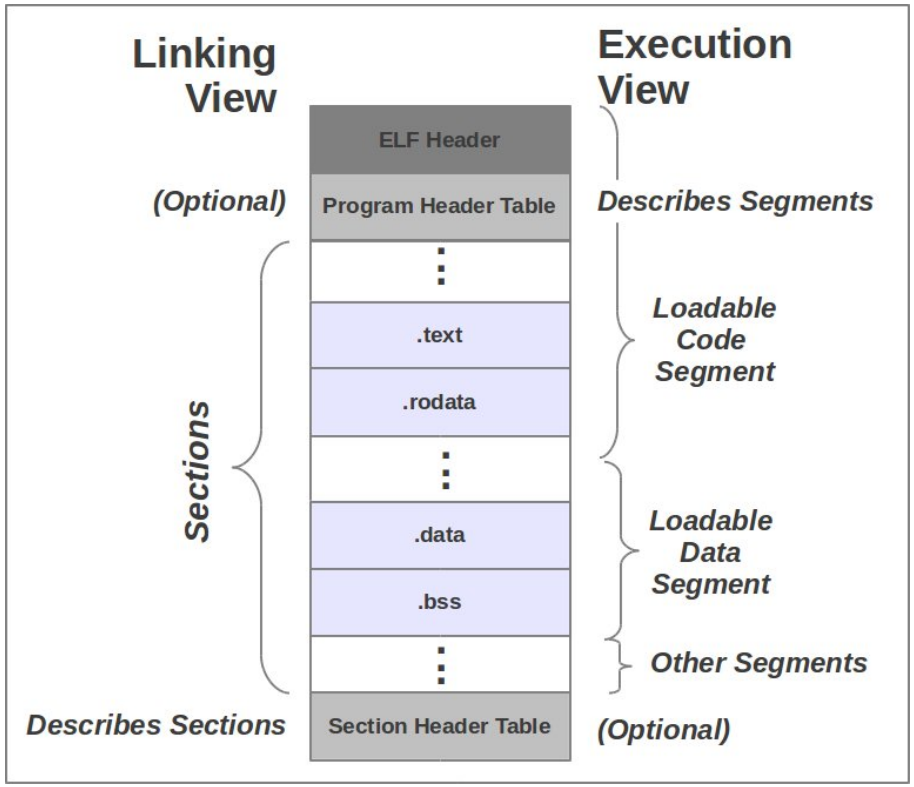
- The *LINKER* translates symbolic names (Labels) into PHYSICAL ADDRESSES, based on information contained in a MAPPING file (sometimes called a SCATTER file on ARM).



- In your assembly language files you define *MEMORY SECTIONS,* that is blocks of information that can be put in Data or Code segments.
- *The mapping or scatter file specifies the location for each SECTION*

Memory Sections and Segments

- The *LINKER* translates symbolic names (Labels) into PHYSICAL ADDRESSES, based on information contained in a LINKER SCRIPT (sometimes called a SCATTER file on ARM).



https://web.archive.org/web/20171129031316/http://nairobi-embedded.org/040_elf_sec_seg_vma_mappings.html

http://nairobi-embedded.org/040_elf_sec_seg_vma_mappings.html

- In your assembly language files you define *MEMORY SECTIONS,* that is blocks of information that can be put in Data or Code segments.
- *The linker-script / scatter file specifies the location for each SEGMENT / REGION*

An Example with GNU assembler:

```
.text                           ;@ a directive – switch to text section
;@ a comment
            .global Start       ;@ make Start visible to the linker
Start:
            MOV r0, #10         ;@ an instruction
            ADRL r1, Start      ;@ a pseudo-instruction
            ADD r0, r0, r1      ;@ another comment
Stop:       B stop
            .END
```

The structure of a line of symbolic assembly code for us is always
{label:} {instruction | pseudo-instruction | directive} {;@comment}

START and STOP are **LABELS**: labels are ASCII-string symbols that represent locations in memory. Assembler and linker will translate them into real memory addresses.
Instructions, pseudo-instructions, and sometimes directives are usually preceded by whitespace

Note: with the ArmAsm assembler from ARM Inc., a colon (:) is not placed after each label.

Instructions

- Instructions start with text mnemonics that represent the operations that we want our program to execute.
- In some cases, an assembly mnemonic can represent a set of 2 or more instructions

- In our uVision projects we will use Version 4 of the ARM Instruction Set (IS)
- There are different conventions to describe the ARM IS mnemonics, the most common is UAL (Unified Assembler Language). Differences are mostly “cosmetic” …..
- The following table describes all possible instructions in the ARM 4T IS

ADC	ADD	AND	B	BL
BX	CDP	CMN	CMP	EOR
LDC	LDM	LDR	LDRB	LDRBT
LDRH	LDRSB	LDRSH	LDRT	MCR
MLA	MOV	MRC	MRS	MSR
MUL	MVN	ORR	RSB	RSC
SBC	SMLAL	SMULL	STC	STM
STR	STRB	STRBT	STRH	STRT
SUB	SVC/SWI	SWP	SWPB	TEQ
TST	UMLAL	UMULL		

Instructions (2)

Operands of symbolic instructions are typically

- Registers
- Labels
- Immediates

```
add r1,r2,r3          ;@ Computing r1=r2+r3
B STOP                ;@ branching to label stop
mov r0, #5            ;@ moving immediate data 5 to r0
```

“GP” registers are numbered from r0 to r15 (Can be R0..R15 too)

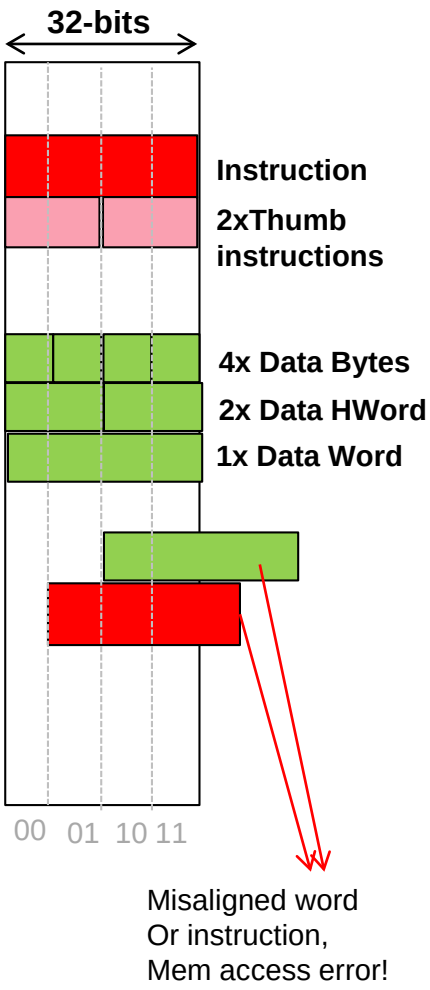
At times, to make code more readable, we can use predefined register aliases:

- SP is the Stack Pointer (r13)
- LR is the Link Register (r14)
- PC is the Program Counter (r15)

- CPSR is the current program status register
- SPSR is the saved program status register (we’ll learn about it later)

Memory Organization: Addressing

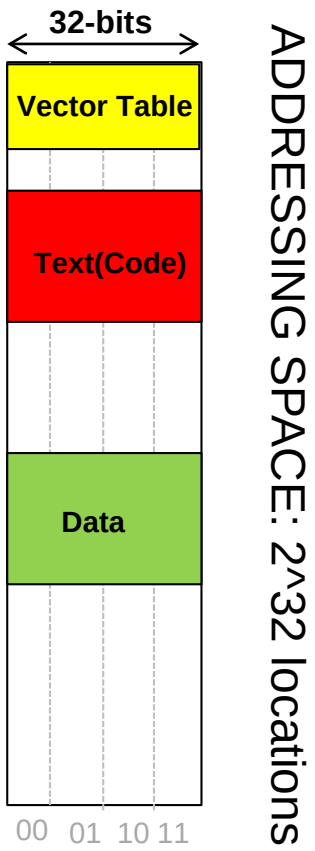
- ARMv4 is a 32-bit processor, featuring instructions and directly handling data of up to 32-bits
- For this reason, the ARM memory is organized in groups of 32-bits
- But, since some instructions (Thumb) can also be 16-bit, and data can be 8- or 16-bit, memory is byte-addressable: any specific address will point to a byte location.
    - 32-bit data and instructions will be word aligned, that is they will have addresses ending with bits “00”
    - 16-bit data and instructions will be half-word aligned, that is they will have addresses ending with bit “0”
    - 8-bit data can be placed anywhere in the addressing space
    - Addresses are composed of 32-bits, so that ARMv4 can address at most 2^32 8-bit locations, from 0x0 to 0xffff ffff
    - ARMv4 can address at most 2^30 32-bit words



Memory Organization: Sections

- The memory addressing space, that is the set of all 2^32 available addresses is organized in SECTIONS;
- A SECTION is a homogeneous, unique and contiguous portion of the addressing space.
- We call *mapping* a task of the linker. It translates all labels into physical addresses. The linker will perform that task based on Sections: each section has a reserved region and will be placed in a specified sub-space of the addressing space

- Sections normally have a default placement but we can alter that by specifying alternative «Mappings» to the assembler and linker using the Linker Script or “Scatter” file
- Examples of sections are: TEXT, DATA, VECTOR TABLE
- Sections are normally aligned to words, or even larger blocks of memory



Memory Organization: Altering default organization

- When we write assembler code, instructions and data we specify will be placed incrementally by the assembler in the specified section (ex. text for instructions, data for writeable data, etc.)

- In case we want to specifically alter their placement in memory we use specific Assembler Directives.

Assembler Directives:

- Directives are NOT mnemonics that need to be translated to processor instructions, but commands we give to the assembler itself in order to build the final code and data in the way we want. They are largely related to the way our instructions and data should be organized in the processor memory (i.e. what goes where)

  - Please note that Assembler instructions are related to the ISA, so they mostly depend on the processor we are using.
  - On the other hand, Assembler Directives depend on the ASSEMBLER, or more generally, TOOLCHAIN we are using. We can have several different toolchains for ARM, ***and their directives differ substantially***
  - GNU directives are shown on the next pages – details at: https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_7.html
  - ArmAsm directives can be found at: https://developer.arm.com/documentation/dui0473/k/directives-reference

.REQ , .EQU

Name* .REQ *definition

.REQ can be used to associate user-friendly mnemonics to a given register. It can be used to help the code become more readable by associating a helpful name to a register.

Examples:
```
vect1_addr    .REQ   r4
a0            .REQ   r0 ;@ argument 0
```

.EQU *name* ,*constant*

.EQU is used to give a symbolic name to a numeric constant, similarly to #define in C

Example:
```
.EQU SRAM_BASE, 0x40000
```

.ALIGN

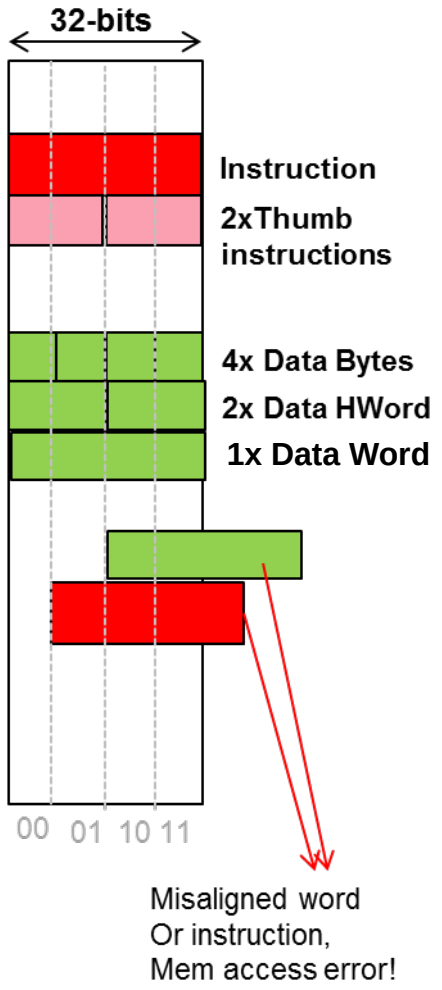
.ALIGN <expr>

Will align the current location to a specified boundary, padding the empty space with zeros by default.

It will position the next address at the nearest ALIGNED address.

*If expr is not specified, next address will be 2^**2**-bytes-aligned (expr=**2**)*

- In some cases, it is essential that we use ALIGN to ensure that our code and instruction is aligned to the appropriate boundaries



.SPACE, .WORD, .BYTE .END

.SPACE

Reserves a block of memory, zeroed by default

If you are now not aligned, you may need to use .align before you add further code/data to your assembly file

Example: `.SPACE 3 ;@ space for 3 bytes`

.WORD

Defines one or more 32-bit words in memory

Example: `.WORD 0x40000, -1234`

.BYTE

Defines one or more 8-bit bytes in memory

Example: `.BYTE 0x40, -123`

.END

Is used to specify the end of a given source assembler file

USER DEFINED MACROS

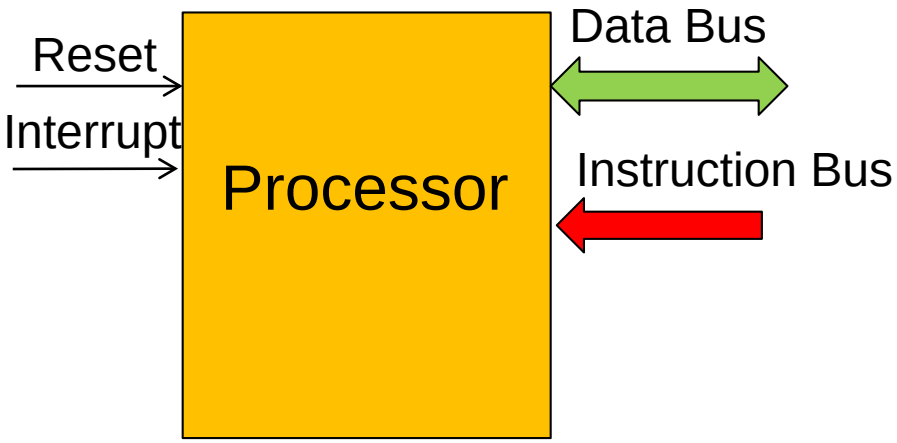
- We call User defined MACRO a piece of assembler code that is written once but can be reused many times across your source file
    - It is very similar to the C preprocessing directive «#define» and the function-like macros it can define.

- It should be noted that from the processor point of view a macro does not exist, it is just a trick that we can use to INSTANTIATE MULTIPLE TIMES THE SAME CODE, and it is resolved by the assembler. Subroutines, on the other hand, affect the link register for example.

- MACROS are very different from branching to a subroutine:
    - a subroutine allows the user to execute multiple times THE SAME PORTION OF CODE in memory
    - A MACRO is defined once, but then copied by the assembler at every instantiation, so that multiple instances of a macro with instructions will lead to different code instances in memory

Question: Is there a sprintf equivalent in assembly

- There is no native “sprintf” in assembly, because assembly is a representation of machine code, and machine code is the language of a processor.
- Think Hardware! A processor is a piece of doped silicon with a set of metal pins:



- The only interaction between a processor and the rest of the world can be:
  1. We give instructions via the instruction bus
  2. We call the processor to our attention via reset/interrupt
  3. We exchange data with the processor via data bus

Is there an sprintf in assembler?

So, sprintf is not an instruction, it's a complex function that performs the following tasks:

```
sprintf("My Assignment avg in this course is %d so I expect a grade of
   %s",100,''A+'');
```

1. Resolve all arguments and translate them into strings depending on the format
2. Produce an array of chars in a given location in data memory corresponding to the string that needs to be output

- Can we write sprintf in assembly? We can more easily write putchar in assembly. We could write sprintf in assembly too, but being a complex function(~100 lines of C, 1000 to ~10000 lines of ASM) we are better off using C

Tutorial: Subroutine Versus MACRO

- Suppose we want to write a routine that computes the following calculation

$$Y = a \; + \; b \ll 3 \; + \; c \gg 1$$

multiple times in ***different*** regions of our code:

We can render the calculation using different strategies:

a) MACRO INSTANTIATIONS

b) SUBROUTINE CALLS

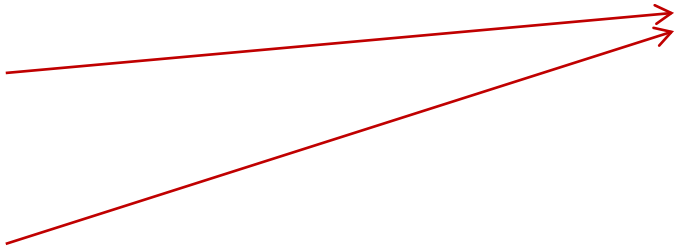
Tutorial: Subroutine call example

- Suppose we want to write a routine that computes the following calculation

$$Y = a + b \ll 3 + c \gg 1$$

multiple times in different regions of our code: in the following we are defining a sort of ***SUBROUTINE CALL***

```
            .TEXT
            .GLOBAL   Reset_Handler
Reset_Handler:
            mov r0, #2      ;@   r0=a,r1=b,r2=c
            mov r1, #1
            mov r2, #4
            bl  shiftadd_f
            mov r0, #3
            mov r1, #8
            mov r2, #3
            bl  shiftadd_f
stop:       b   stop
```



```
            .TEXT
            .GLOBAL   shiftadd_f
shiftadd_f:

            mov r1, r1,LSL #3
            mov r2, r2,LSR #1
            add r0,r0,r1
            add r0,r0,r2
            mov PC,LR
            .size shiftadd_f, .  -   shiftadd_f
```

Tutorial: Macro example

- Suppose we want to write a routine that computes the following calculation

$$Y = a + \ll 3 + c \gg 1$$

multiple times in different regions of our code: in the following we are defining a sort of ***MACRO:***

```
            .TEXT
            .GLOBAL Reset_Handler
Reset_Handler:
             mov r0, #2      ;@   r0=a,r1=b,r2=c
            mov r1, #1
            mov r2, #4
            ShiftAdd r0,r1,r2
            mov r0, #3
            mov r1, #8
            mov r2, #3
            ShiftAdd r0,r1,r2
stop:       b   stop
```

```
.MACRO ShiftAdd a, b, c
mov \b, \b, LSL #3
mov \c, \c, LSR #1
add \a,\a,\b
add \a,\a,\c
.ENDM
```

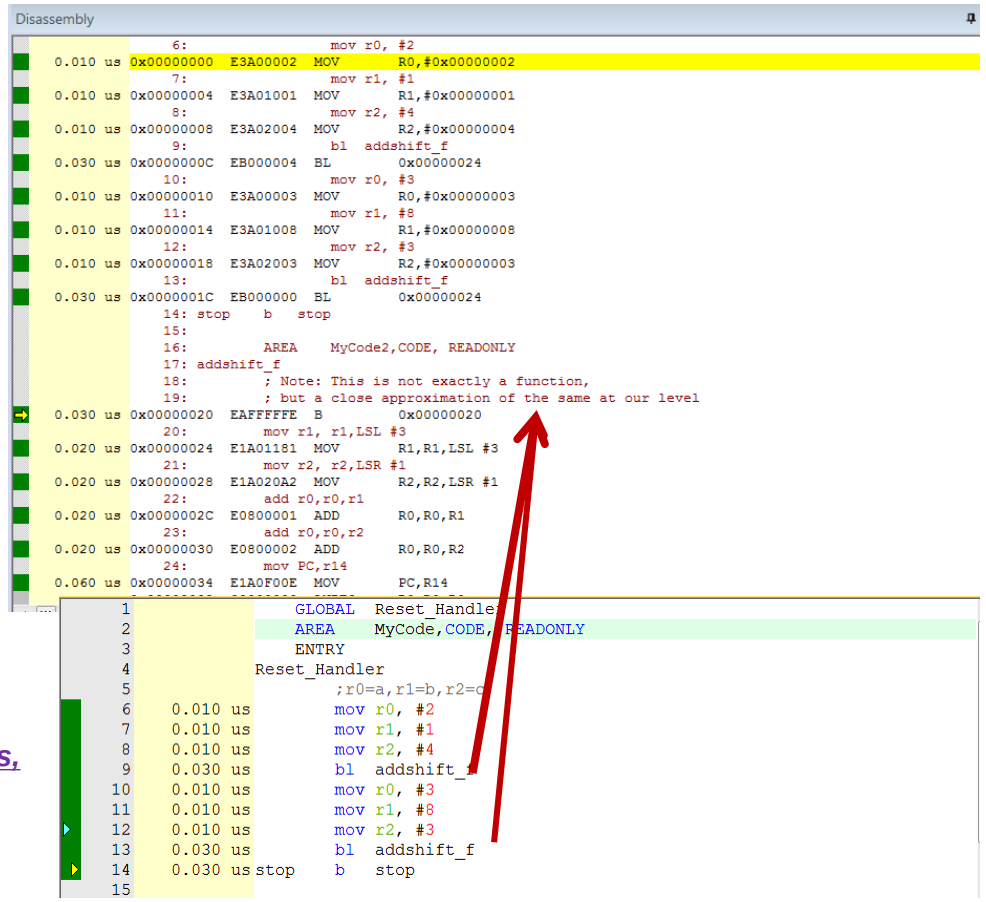
subroutine-based Computation

- ☺ There is one same area of code we branch to every time the functionality is required
- ☹ We need to Branch there and back every time, Computation is slower

Total Time @100MHz:
260 ns (26 Cycles)

Total #instructions in IMEM:
14 (56 bytes)

Note: Branches take 3 times as much as other instructions, We'll explore why in a future unit



MACRO-based Computation

- ☹ The macro code is Copy-Pasted into Instruction memory every time the Macro is called -> can occupy a lot of IMEM area
- ☺ There is no Branch -> Computation is faster

Total Time @100MHz:
140 ns (14 Cycles)

Total #instructions in IMEM:
15 (60 bytes)