

# CSE 3113: Microprocessor and Assembly Lab

Dr. Upama Kabir  
Professor

Computer Science and Engineering, University of Dhaka

**Lab 2**

**February 06, 2024**

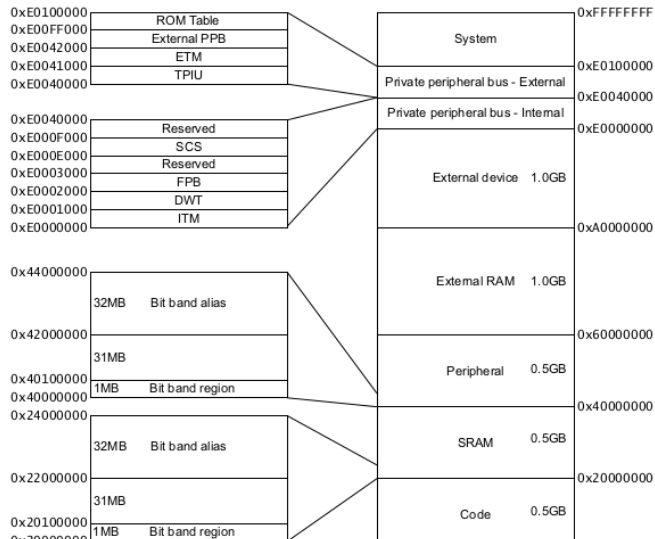
# Outline

- 1 Memory usage
- 2 System address map
- 3 Cortex-M4 Assembly Syntax
- 4 ARM Assembly Program Example
- 5 Directives
- 6 Directive: Data Allocation
- 7 Memory Access Instructions
- 8 Data Processing Instruction Set
- 9 General Data Processing Instructions
- 10 Your Task
- 11 Submission Guideline

# Memory usage

- Code memory (normally read-only memory)
  - Program instructions
  - Constant data
- Data memory (normally read/write memory – RAM)
  - Variable data/operands
- Stack (located in data memory)
  - Save information temporarily and retrieve it later
  - Return addresses for subroutines and interrupt/exception handlers
  - Data to be passed to/from a subroutine/function
- Peripheral addresses
  - Used to access registers in “peripheral functions” (timers, ADCs, communication modules, etc.) outside the CPU

# System address map



# Cortex-M4 Assembly Syntax

label

mnemonic operand1,operand2, ...; Comments

- Label is used as a reference to an address location;
- Mnemonic is the name of the instruction;
- Operand1 is the destination of the operation;
- Operand2 is normally the source of the operation;
- Comments are written after “;”, which does not affect the program;

- For example

MOVSR3, #0x11;Set register R3 to 0x11

- Note that the assembly code can be assembled by either ARM assembler (armasm) or assembly tools from a variety of vendors (e.g. GNU tool chain). When using GNU tool chain, the syntax for labels and comments is slightly different

# ARM Assembly Program Example

```
AREA test, CODE, READONLY
ENTRY ; starting point of the code execution
EXPORT main ; the declaration of identifier main
main ; address of the main function
    ; User code starts from the next line
    MOV r0, #4 ; store some arbitrary numbers
    MOV r1, #5
    ADD r2, r0, r1 ; add the values in r0 and r1 and store the result in r2
STOP B Stop ; Endless loop
END ; End of the program, matched with ENTRY keyword
```

Figure 3

# Assembler Directives

- Keil has an ARM assembler which can compile and build ARM assembly language programs.
- To drive the assembly and linking process, we need to use directives, which are interpreted by the assembler.
- Assembler directives are commands to the assembler that direct the assembly process.
- They are executed by the assembler at assembly time not by the processor at run time.
- Machine code is not generated for assembler directives as they are not directly translated to machine language.

# Directives: AREA

- The AREA directive indicates to the assembler the start of a new data or code section.
- Areas are the basic independent and indivisible unit processed by the linker.
- Each area is identified by a name and areas within the same source file cannot share the same name.
- An assembly program must have at least one code area.
- By default, a code area can only be read and a data area may be read from and written to.
- Example:  
**AREA myData, DATA, READWRITE ; Define a data section**



# Directives: END

- The END directive indicates the end of a source file.
- Each assembly program must end with this directive

# Directives: ENTRY

- The ENTRY directive marks the first instruction to be executed within an application.
- There must be one and only one entry directive in an application, no matter how many source files the application has.

# Directives: PROC and ENDP

- PROC and ENDP are to mark the start and end of a function (also called subroutine or procedure).
- A single source file can contain multiple subroutines, with each of them defined by a pair of PROC and ENDP.
- PROC and ENDP cannot be nested. We cannot define a subroutine within another subroutine

# Directives: EXPORT and IMPORT

- The EXPORT declares a symbol and makes this symbol visible to the linker.
- The IMPORT gives the assembler a symbol that is not defined locally in the current assembly file.
- The IMPORT is similar to the “extern” keyword in C.
- Example:  
**EXPORT** `_main` ; Make `_main` visible to the linker  
**IMPORT** `sinx` ; Function `sinx` defined in another file


# Directives: EQU

- The EQU directive associates a symbolic name to a numeric constant. Similar to the use of #define in a C program, the EQU can be used to define a constant in an assembly code.
- Example:  
`MyConstant EQU 1234 ; Constant 1234 to use later`  
`MOV R0, #MyConstant ; Constant 1234 placed in R0`

# Directives: ALIGN

```
AREA example, CODE, ALIGN = 3 ; Memory address begins at a multiple of 8
ADD r0, r1, r2                ; Instructions start at a multiple of 8

AREA myData, DATA, ALIGN = 2 ; Address starts at a multiple of four
a DCB 0xFF                    ; The first byte of a 4-byte word
  ALIGN 4, 3                   ; Align to the last byte of a word
b DCB 0x33                    ; Set the fourth byte of a 4-byte word
c DCB 0x44                    ; Add a byte to make next data misaligned
  ALIGN                        ; Force the next data to be aligned
d DCD 12345                    ; Skip three bytes and store the word
```



ALIGN generally used as in this example,  
to align a variable to its data type.

Figure 4

# Directives: INCLUDE

```
INCLUDE constants.s      ; Load Constant Definitions
AREA main, CODE, READONLY
EXPORT __main
ENTRY
__main PROC
...
ENDP
END
```

Figure 5

- The INCLUDE directive is to include an assembly source file within another source file.
- It is useful to include constant symbols defined by using EQU and stored in a separate source file

# Directive: Data Allocation

<b>Directive</b>	<b>Description</b>	<b>Memory Space</b>
DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values
DCQ	Define Constant	Reserve 64-bit values
SPACE	Defined Zeroed Bytes	Reserve a number of zeroed bytes
FILL	Defined Initialized Bytes	Reserve and fill each byte with a value

Figure 6

- DCx : reserve space and initialize value(s) for ROM (initial values ignored for RAM)
- SPACE : reserve space without assigning initial values (especially useful for RAM)



# Directive: Data Allocation

AREA	myData, DATA, READWRITE	
hello	<b>DCB</b> "Hello World!",0	; Allocate a string that is null-terminated
dollar	<b>DCB</b> 2,10,0,200	; Allocate integers ranging from -128 to 255
scores	<b>DCD</b> 2,3,-8,4	; Allocate 4 words containing decimal values
miles	<b>DCW</b> 100,200,50,0	; Allocate integers between -32768 and 65535
p	<b>SPACE</b> 255	; Allocate 255 bytes of zeroed memory space
f	<b>FILL</b> 20,0xFF,1	; Allocate 20 bytes and set each byte to 0xFF
binary	<b>DCB</b> 2_01010101	; Allocate a byte in binary
octal	<b>DCB</b> 8_73	; Allocate a byte in octal
char	<b>DCB</b> 'A'	; Allocate a byte initialized to ASCII of 'A'

Figure 7

# Memory Access Instructions

Mnemonic	Brief Description	See Page
ADR	Load PC-relative address	36
CLREX	Clear exclusive	52
LDM{mode}	Load multiple registers	46
LDR{type}	Load register using immediate offset	37
LDR{type}	Load register using register offset	40
LDR{type}T	Load register with unprivileged access	42
LDR{type}	Load register using PC-relative address	44
LDRD	Load register using PC-relative address (two words)	44
LDREX{type}	Load register exclusive	50
POP	Pop registers from stack	48
PUSH	Push registers onto stack	48
STM{mode}	Store multiple registers	46
STR{type}	Store register using immediate offset	37
STR{type}	Store register using register offset	40
STR{type}T	Store register with unprivileged access	42
STREX{type}	Store register exclusive	50

# Arithmetic Data Operation

Table 1: Arithmetic Data Operation

Operation	Mnemonic	Index Mode
Add	ADD	ADD $R_d, R_n, R_m$
Add	ADD	ADD $R_d, R_n, \#immed$
Add with Carry	ADC	ADC $R_d, R_n, R_m$
Add with Carry	ADC	ADC $R_d, \#immed$
Subtract	SUB	SUB $R_d, R_m, R_n$
Subtract	SUB	SUB $R_d, R_n, \#immed$
Subtract with Borrow	SBC	SBC $R_d, R_n, \#immed$
Subtract with Borrow	SBC	SBC $R_d, R_n, R_m$
Reverse Subtract	RSB	RSB $R_d, R_n, R_m$

# Arithmetic Data Operation

Table 2: Arithmetic Data Operation

Operation	Mnemonic	Index Mode
Multiply	MUL	MUL $R_d, R_n, R_m$
Unsigned Division	UDIV	UDIV $R_d, R_n, \#immed$
Signed Division	ADC	SDIV $R_d, R_n, R_m$
Multiply Accumulate	MLA	MLA $R_d, R_n, R_m, R_a s$
Multiply Subtract	MLS	MLS $R_d, R_n, R_m, R_a s$
S Signed Multiply Long	SMULL	SMULL $Rd_{lo}, Rd_{hi}, R_d, R_n$
Unsigned Multiply Long	UMULL	UMULL $Rd_{lo}, Rd_{hi}, R_d, R_n$

# Logic Operation

Table 3: Logic Operation

Operation	Mnemonic	Index Mode
Bitwise AND	AND	AND $R_d, R_n$
Bitwise AND	AND	AND $R_d, R_n, R_m$
Bitwise OR	ORR	ORR $R_d, R_n$
Bitwise Bit Clear	BIC	BIC $R_d, R_n, R_m$
Bitwise OR NOT Clear	ORN	ORN $R_d, R_n, R_m$
Bitwise Exclusive OR	EOR	EOR $R_d, R_n, R_m, R_a s$
S Bitwise NOT	MVN	MVN $R_d$

# General Data Processing Instructions

Mnemonic	Brief Description	See Page
ADC	Add with carry	55
ADD	Add	55
ADDW	Add	55
AND	Logical AND	58
ASR	Arithmetic shift right	60
BIC	Bit clear	58
CLZ	Count leading zeros	63
CMN	Compare negative	64
CMP	Compare	64
EXOR	Exclusive OR	58
LSL	Logical shift left	60
LSR	Logical shift right	60
MOV	Move	65
MOVT	Move top	67
MOVW	Move 16-bit constant	65
MOVN	Move NOT	65
ORN	Logical OR NOT	58
ORR	Logical OR	58
RBIT	Reverse bits	68
REV	Reverse byte order in a word	68
REV16	Reverse byte order in each halfword	68
REVSH	Reverse byte order in bottom halfword and sign extend	68
ROR	Rotate right	60
RRX	Rotate right with extend	60
RSB	Reverse subtract	55
SADD16	Signed add 16	70
SADD8	Signed add 8	70
SASX	Signed add and subtract with exchange	81
SSAX	Signed subtract and add with exchange	81
SBC	Subtract with carry	55
SEL	Select bytes	98
SHADD16	Signed halving add 16	72
SHADD8	Signed halving add 8	72
SHASX	Signed halving add and subtract with exchange	74
SHSAX	Signed halving subtract and add with exchange	74
SRSUB16	Signed halving subtract 16	77
SRSUB8	Signed halving subtract 8	77

# General Data Processing Instructions

Mnemonic	Brief Description	See Page
SSUB16	Signed subtract 16	79
SSUB8	Signed subtract 8	79
SUB	Subtract	55
SUBW	Subtract	55
TEQ	Test equivalence	84
TST	Test	84
UADD16	Unsigned add 16	86
UADD8	Unsigned add 8	86
UASX	Unsigned add and subtract with exchange	88
USAX	Unsigned subtract and add with exchange	88
UHADD16	Unsigned halving add 16	91
UHADD8	Unsigned halving add 8	91
UHASX	Unsigned halving add and subtract with exchange	93
UHSAX	Unsigned halving subtract and add with exchange	93
UHSUB16	Unsigned halving subtract 16	96
UHSUB8	Unsigned halving subtract 8	96
USAD8	Unsigned sum of absolute differences	99
USADA8	Unsigned sum of absolute differences and accumulate	101
USUB16	Unsigned subtract 16	103
USUB8	Unsigned subtract 8	103

# Your Task

- Write an assembly language to perform all the arithmetic operations (Addition, Subtraction and Multiplication ) on two variables. You don't have to handle overflow. Two modes of operations:
  - ① In this case, you will put the data in memory in the form of constants before the program runs.
  - ② You first use the load register, **LDR r4,X** instruction to load register r4 with the contents of memory location.
- Find the smaller of two integer numbers.



# Submission Guideline

- ① Your Assembly code with proper comments. (\*.s file)
- ② A document (\*.tex file) that contains:
  - ③ Detail explanation of the code
  - ④ Screenshot that shows the state of the system after the code has been loaded.
  - ⑤ Screenshot that shows the situation after the code has been executed.
- ⑥ Submit as a .zip file. Example: your classroll\_lab#.zip (12\_lab2.zip)