EMBEDDED SYSTEMS

SUBJECT CODE: 18EC62

BY:

DILEEP J

ASST. PROF., DEPT. OF ECE, KSSEM

MODULE-2:

Module-2
ARM Cortex M3 Instruction Sets and Programming: Assembly basics, Instruction list and description, Useful instructions, Memory mapping, Bit-band operations and CMSIS, Assembly and C language Programming (Text 1: Ch-4, Ch-5, Ch-10 (10.1, 10.2, 10.3, 10.5 only) **L1, L2, L3**

Text Books:

1. Joseph Yiu, "The Definitive Guide to the ARM Cortex-M3", 2nd Edition, Newnes, (Elsevier), 2010.

Assembly Basics

- **Assembler Language: Basic Syntax**
- Label opcode operand1, operand2, … ; Comments
- The label is optional. Some of the instructions might have a label in front of them so that the address of the instructions can be determined using the label.
- Then, you will find the op-code (the instruction) followed by a number of operands. Normally, the first operand is the destination of the operation.
- For example, immediate data are usually in the form #number, as shown
- MOV R0, #0Xff ; Set R0 = 0xFF (hexadecimal)
- MOV R1, #'S' ; Set R1 = ASCII character S

- The text after each semicolon (;) is a comment. These comments do not affect the program operation,
- but they can make programs easier for humans to understand.
- You can define constants using EQU, and then use them inside your program code. For example,

```
NVIC_IRQ0_ENABLE   EQU    0x1
.....
MOV  R1,#NVIC_IRQ0_ENABLE ; Move immediate data to register
STR  R1,[R0] ; Enable IRQ 0 by writing R1 to address in R0
```

- For the Cortex-M3, the conditional execution suffixes are usually used for branch instructions.

Suffix	Description
S	Update Application Program Status register (APSR) (flags); for example: `ADDS R0, R1 ; this will update APSR`
EQ, NE, LT, GT, and so on	Conditional execution; EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, and so forth. For example: `BEQ <Label> ; Branch if equal`

- **Unified Assembler Language:** UAL was developed to allow selection of 16-bit and 32-bit instructions and to make it easier to port applications between ARM code and Thumb code by using the same syntax for both.
- ADD R0, R1 ; R0 = R0 + R1, using Traditional Thumb syntax
- ADD R0, R0, R1 ; Equivalent instruction using UAL syntax

- **Assembler Language: Moving Data**
- **Syntax: MOV DEST , SRC**
- One of the most basic functions in a processor is transfer of data. In the Cortex-M3, data transfers can be of one of the following types:

• Moving data between register and register

• Moving data between memory and register

• Moving data between special register and register

• Moving an immediate data value into a register

- The command to move data between registers is MOV (move). For example, moving data from register R3 to register R8 looks like this:

MOV R8, R3

- Another instruction can generate the negative value of the original data; it is called MVN (move negative).
- Ex: MVN R0, R1; copies inverted version R1 data to R0

- The basic instructions for accessing memory are Load and Store.
- **Load (LDR) : transfers data from memory to registers, and Store transfers data from registers to memory.**
- The exclamation mark (!) in the instruction specifies whether the register *Rd should be updated after the instruction is completed*.
- For example, if R8 equals 0x8000:

1. **STMIA.W R8!, {R0-R3} ;** R8 changed to 0x8010 after store; (increment by 4 words)
2. **STMIA.W R8 , {R0-R3} ;** R8 unchanged after store ; w means "wide"

- **Ex 1: Address** 0x8000: 8-Bit Data
- 0x8001: 8-Bit Data
- 0x8002: 8-Bit Data
- 0x8003: 8-Bit Data ; till here R0= 32-bit data storage is done
- 0x8004: 8-Bit Data
- 0x8005: 8-Bit Data
- 0x8006: 8-Bit Data
- 0x8007: 8-Bit Data; till here R1 = 32 bit data storage is done
- 0x8008: 8-Bit Data
- 0x8009: 8-Bit Data
- 0x800A: 8-Bit Data
- 0x800B: 8-Bit Data; till here R2 = 32 bit data storage is done
- 0x800C: 8-Bit Data
- 0x800D: 8-Bit Data
- 0x800E: 8-Bit Data
- 0x800F: 8-Bit Data; till here R3 = 32 bit data storage is done

R8 changed to 0x8010 after store; (increment by 4 words)

- ARM processors also support memory accesses with pre-indexing and post-indexing. For pre-indexing, the register holding the memory address is adjusted. The memory transfer then takes place with the updated address. For example,
- **LDR.W R0,[R1, #offset]! ;** Read memory [R1+offset], with R1 and update to R1+offset
- Offset can be 16 bit data or 32 bit data
- R1 contents will be added to 16 or 32 bit data. This forms the memory address. Content which is present in that address will be copied to R0 register.
- Exclamatory mark indicates the update of R1 register
- Ex: before execution: R1=0x1000 offset=0x0050
- After execution: R1=0x1050

Table 4.14 Commonly Used Memory Access Instructions

Example	Description
`LDRB Rd, [Rn, #offset]`	Read byte from memory location Rn + offset
`LDRH Rd, [Rn, #offset]`	Read half word from memory location Rn + offset
`LDR Rd, [Rn, #offset]`	Read word from memory location Rn + offset
`LDRD Rd1,Rd2, [Rn, #offset]`	Read double word from memory location Rn + offset
`STRB Rd, [Rn, #offset]`	Store byte to memory location Rn + offset
`STRH Rd, [Rn, #offset]`	Store half word to memory location Rn + offset
`STR Rd, [Rn, #offset]`	Store word to memory location Rn + offset
`STRD Rd1,Rd2, [Rn, #offset]`	Store double word to memory location Rn + offset

B=Byte (8-bit), H=half word (16-bits), W= Word (32-bits), D= Double Word (64-bits)

- Two other types of memory operation are stack PUSH and stack POP. For example,
- PUSH {R0, R4-R7, R9} ; Push R0, R4, R5, R6, R7, R9 into stack memory
- POP {R2,R3} ; Pop R2 and R3 from stack
- Usually a PUSH instruction will have a corresponding POP with the same register list, but this is not always necessary. For example, a common exception is when POP is used as a function return:
- PUSH {R0-R3, LR} ; Save register contents at beginning of

; subroutine Processing

- POP {R0-R3, PC} ; restore registers and return
- In this case, instead of popping the LR register back and then branching to the address in LR, we POP the address value directly in the program counter.

- **Assembler Language: Processing Data**
- The Cortex-M3 provides many different instructions for data processing. A few basic ones are Introduced here. Many data operation instructions can have multiple instruction formats. For example, an ADD instruction can operate between two registers or between one register and an immediate data value:
- **ADD R0, R0, R1 ; R0 = R0 + R1**
- **ADDS R0, R0, #0x12 ; R0 = R0 + 0x12 and update APSR**
- **ADD.W R0, R1, R2 ; R0 = R1 + R2**
- **ADD.W R0, R1, R2 ; Flag unchanged**
- **ADDS.W R0, R1, R2 ; Flag change**

- **Assembler Language: Call and Unconditional Branch**
- B label ; Branch to a labeled address
- BX reg ; Branch to an address specified by a register
- In BX instructions, the LSB of the value contained in the register determines the next state (Thumb/ARM) of the processor.
- In the Cortex-M3, because it is always in Thumb state, this bit should be set to 1.
- If it is zero, the program will cause a usage fault exception because it is trying to switch the processor into ARM state (See Figure 4.2.).
- To call a function, the branch and link instructions should be used.
- BL label ; Branch to a labeled address and save return address in ;LR

- POP {R15} ; Do a stack pop operation, and change the
  ; program counter value to the result value.
- When using these methods to carry out branches, you also need to make sure that the LSB of the new program counter value is 0x1.
- Otherwise, a usage fault exception will be generated because it will try to switch the processor to ARM mode, which is not allowed in the Cortex-M3 redundancy.

- ***Assembler Language: Conditional Execution Using IT Instructions***
- The IT (IF-THEN) block is very useful for handling small conditional code. It avoids branch penalties
- because there is no change to program flow. It can provide a maximum of four conditionally executed instructions.
- In IT instruction blocks, the first line must be the IT instruction, detailing the choice of execution, followed by the condition it checks.

- TRUE-THEN-EXECUTE,
- which is always written as *IT xyz, where T means THEN and E means*
- ELSE. The second through fourth statements can be either THEN (true) or ELSE (false):

IT<x><y><z> <cond> ; IT instruction (<x>, <y>,

; <z> can be T or E)

instr1<cond> <operands> ; 1st instruction (<cond>

; must be same as IT)

instr2<cond or not cond> <operands> ; 2nd instruction (can be

; <cond> or <!cond>

instr3<cond or not cond> <operands> ; 3rd instruction (can be

; <cond> or <!cond>

instr4<cond or not cond> <operands> ; 4th instruction (can be

; <cond> or <!cond>

- You can have fewer than four conditionally executed instructions. The minimum is 1. You need to make sure the number of *T and E occurrences in the IT instruction matches the number of conditionally* executed instructions after the IT.
- If an exception occurs during the IT instruction block, the execution status of the block will be stored in the stacked PSR (in the IT/Interrupt-Continuable Instruction [ICI] bit field).
- So, when the exception handler completes and the IT block resumes, the rest of the instructions in the block can continue the execution correctly.
- In the case of using multicycle instructions (for example, multiple load and store) inside an IT block, if an exception takes place during the execution, the whole instruction is abandoned and restarted after the interrupt process is completed.

4.3.7 Assembler Language: Instruction Barrier and Memory Barrier Instructions

- The Cortex-M3 supports a number of barrier instructions.
- These instructions are needed as memory systems get more and more complex.
- In some cases, if memory barrier instructions are not used, race conditions could occur.

Table 4.27 Barrier Instructions

Instruction	Description
DMB	Data memory barrier; ensures that all memory accesses are completed before new memory access is committed
DSB	Data synchronization barrier; ensures that all memory accesses are completed before next instruction is executed
ISB	Instruction synchronization barrier; flushes the pipeline and ensures that all previous instructions are completed before executing new instructions

SDIV and UDIV

- The syntax for signed and unsigned divide instructions is as follows:

SDIV.W <Rd>, <Rn>, <Rm>

UDIV.W <Rd>, <Rn>, <Rm>

- The result is Rd = Rn/Rm. For example,

```
LDR  R0,=300 ; Decimal 300
MOV  R1,#5
UDIV.W   R2, R0, R1
```

- This will give you an R2 result of 60 (0x3C).

Web Links:

https://os.mbed.com/media/uploads/4180_1/cortexm3_instructions.htm

https://www.youtube.com/watch?v=topbkiRevWM