Lecture Two: Embedded ARM Microcontrollers

1. ARM Microcontroller – Overview

ARM *(Advanced RISC machine) is a family of Reduced Instruction Set Computing (RISC) microprocessors developed specifically for mobile and embedded computing environments.* Due to their small sizes and low power requirements, ARM processors have become the most widely used processors in mobile devices, e.g. smart phones, and embedded systems. Depending on their capabilities and intended applications, ARM Cortex cores can be classified into three categories.

- **The Cortex-M series**: these are microcontroller-oriented processors intended for a wide range of embedded applications.
- **The Cortex-R series**: these are embedded processors intended for real-time signal processing and control applications.
- **The Cortex-A series**: these are applications processors intended for general purpose applications, such as embedded systems.

2. Cortex -M Architecture

Figure 2.1 shows a simplified block diagram of a microcontroller based on the ARM ® Cortex™-M processor. It is a Harvard architecture because it has separate data and instruction buses. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface. There are many advanced debugging features using the DCode bus. Some internal peripherals, like the NVIC (nested vectored interrupt controller) communicate directly with the processor via the private peripheral bus (PPB). The tight integration of the processor and interrupt controller provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt response time.

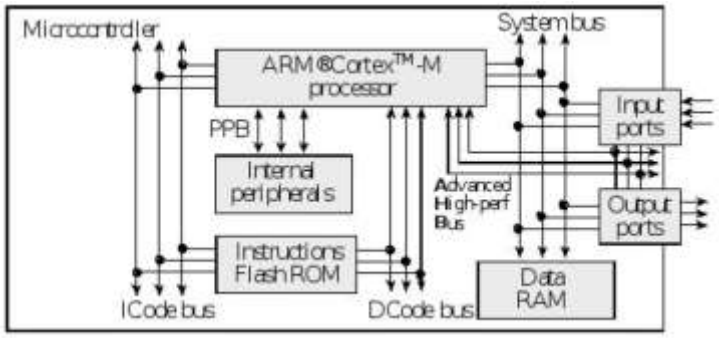

Figure 2.1. Harvard architecture of an ARM® Cortex -M-based microcontroller.

Even though data and instructions are fetched 32-bits at a time, each 8-bit byte has a unique address. This means memory and I/O ports are byte addressable. The processor can read or write 8-bit, 16- bit, or 32-bit data. Exactly how many bits are affected depends on the instruction.

2.1) Registers

Registers *are high-speed storage inside the processor*. The registers are shown in Figure 2.2. R0 to R12 are general purpose registers and contain either data or addresses. Register R13 (also called the stack pointer, SP) points to the top element of the stack. Register R14 (also called the link register, LR) is used to store the return location for functions. The LR is also used in a special way during exceptions, such as interrupts. Register R15 (also called the program counter, PC) points to the next instruction to be fetched from memory. The processor fetches an instruction using the PC and then increments the PC.

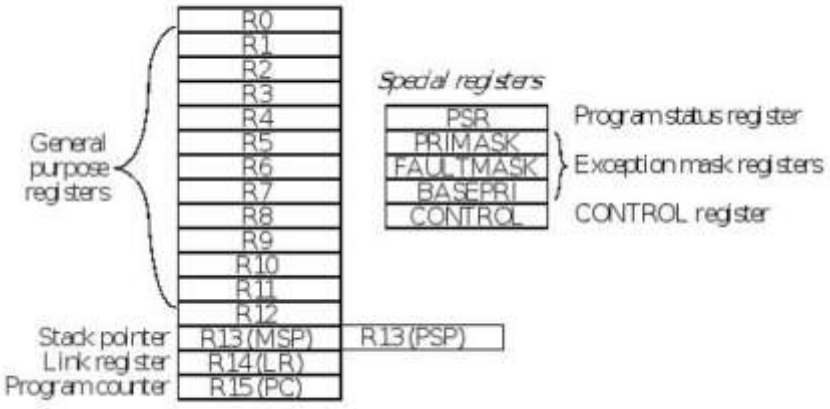

Figure 2.2. Registers on the ARM® Cortex -M processor.

There are three status registers named Application Program Status Register (APSR), the Interrupt Program Status Register (IPSR), and the Execution Program Status Register (EPSR) as shown in Figure 2.3. These registers can be accessed individually or in combination as the Program Status Register (PSR). The N, Z, V, C, and Q bits give information about the result of a previous ALU operation. In general, the N bit is set after an arithmetical or logical operation signifying whether or not the result is negative. Similarly, the Z bit is set if the result is zero. The C bit means carry and is set on an unsigned overflow, the V bit signifies signed overflow and The Q bit indicates that ″saturation″ has occurred (arithmetic saturation). As oxf000 +ox2000=oxffff

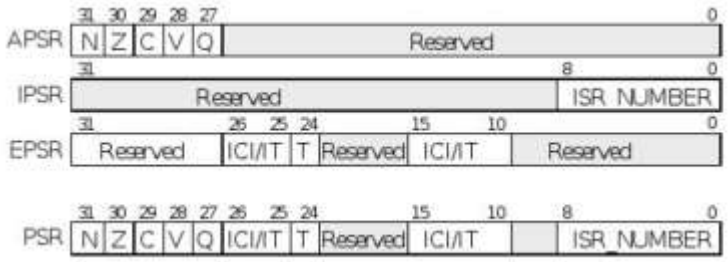

Figure 2.3. The program status register of the ARM® Cortex -M processor.

Bit 0 of the special register PRIMASK is the interrupt mask bit. If this bit is 1, most interrupts and exceptions are not allowed. If the bit is 0, then interrupts are allowed. Bit 0 of the special register FAULTMASK is the fault mask bit. If this bit is 1, all interrupts and faults are not allowed. If the bit is 0, then interrupts and faults are allowed. The non maskable interrupt (NMI) is not affected by these mask bits. The BASEPRI register defines the priority of the executing software. It prevents interrupts with lower or equal priority but allows higher priority interrupts. For example if BASEPRI equals 3, then requests with level 0, 1, and 2 can interrupt, while requests at levels 3 and higher will be postponed. A lower number means a higher priority interrupt.

2.2) Reset

A reset occurs immediately after power is applied and when the reset signal is assured – this can usually be triggered by pushing the reset button available on most boards. The 32-bit value at flash ROM location 0 is loaded into the SP. This value is called the reset vector. On reset, the processor initializes the LR to 0xFFFFFFFF.

2.3) Memory

Microcontrollers within the same family differ by the amount of memory and by the types of I/O modules. All LM3S and TM4C microcontrollers have a Cortex-M processor. There are hundreds of members in this family; some of them are listed in Table 2.1.

Part number	RAM	Flash	I/O	I/O modules
LM3S811	8	64	32	PWM
LM3S1968	64	256	52	PWM
LM3S6965	64	256	42	PWM, Ethernet
LM3S8962	64	256	42	PWM, CAN, Ethernet, IEEE1588
TM4C1231C3PM	32	12	43	floating point, CAN, DMA
TM4C1233H6PM*	32	256	43	floating point, CAN, DMA, USB
TM4C123GH6PM	32	256	43	
				floating point, CAN, DMA, USB, PWM
TM4C123GH6ZRB	32	256	120	floating point, CAN, DMA, USB, PWM
TM4C1294NCPDT	256	1024	90	floating point, CAN, DMA, USB, PWM, Ethernet
	KiB	KiB	pins	

Table 2.1. Memory and I/O modules

In general, Flash ROM begins at address 0x0000.0000, RAM begins at 0x2000.0000, the peripheral I/O space is from 0x4000.0000 to 0x5FFF.FFFF, and I/O modules on the private peripheral bus (PPB) exist from 0xE000.0000 to 0xE00F.FFFF. In particular, the only differences in the memory map for the various 180 members of the LM3S/TM4C family are the ending addresses of the flash and RAM. The following is some of the tasks that can occur in parallel

ICode bus Fetch opcodes from ROM

DCode bus Read constant data from ROM

System bus Read/write data from RAM or I/O, fetch opcode from RAM

PPB Read/write data from internal peripherals like the NVIC

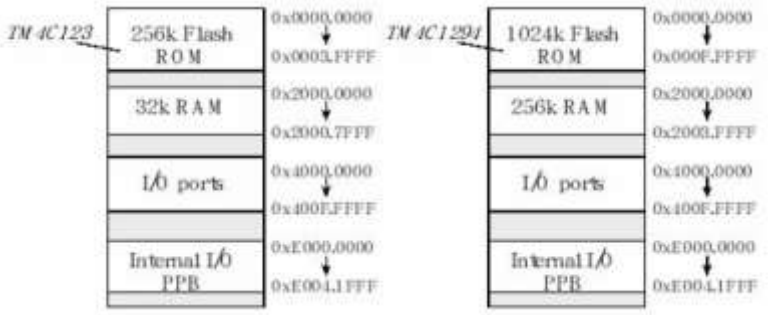

Figure 2.4. Memory maps of the TM4C123 and the TM4C1294

When we store 16-bit data into memory it requires two bytes. Since the memory systems on most computers are byte addressable (a unique address for each byte), there are two possible ways to store in memory the two bytes that constitute the 16-bit data. Freescale microcomputers implement the big endian approach that stores the most significant byte at the lower address. Intel microcomputers implement the little endian approach that stores the least significant byte at the lower address. Cortex M microcontrollers use the little endian format. Many ARM processors are bi-endian, because they can be configured to efficiently handle both big and little endian data. Instruction fetches on the ARM are always little endian. Figure 2.5 shows two ways to store the 16-bit number 1000 (0x03E8) at locations 0x2000.0850 and 0x2000.0851. Computers must choose to use either the big or little endian approach when storing 32-bit numbers into memory that is byte (8-bit) addressable. Figure 2.6 shows the big and little endian formats that could be used to store the 32-bit number 0x12345678 at locations 0x2000.0850 through 0x2000.0853. Again the Cortex M uses little endian for 32-bit numbers.

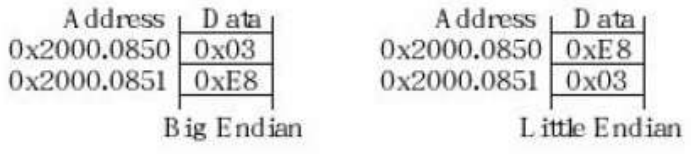

Figure 2.5. Example of big and little endian formats of a 16-bit number.

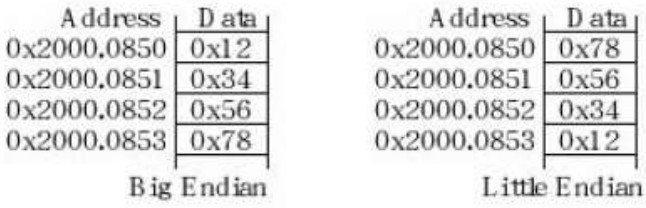

Figure 2.6. Example of big and little endian formats of a 32-bit number.

3. The Software Development Process

The process described in this section applies to both assembly and C. Either the ARM Keil™ uVision® or the Texas Instruments Code Composer Studio™ (CCStudio) integrated development environment (IDE) can be used to develop software for the Cortex M microcontrollers. Both include an editor, assembler, compiler, and simulator. Furthermore, both can be used to download and debug software on a real microcontroller. Either way, the entire development process is contained in one application, as shown in Figure 2.8.

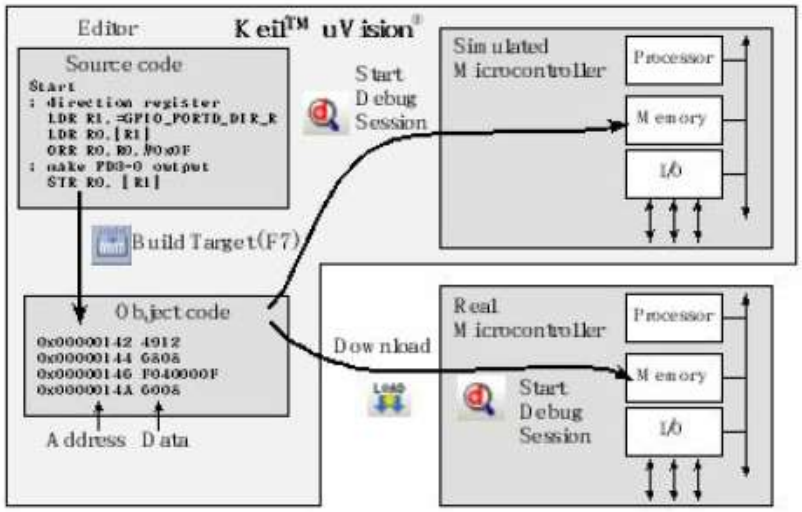

Figure 2.8. Assembly language or C development process.

To develop software, we first use an editor to create our source code. Source code contains specific set of sequential commands in human-readable-form. Next, we use an assembler or compiler to translate our source code into object code. On ARM Keil™uVision® we compile/assemble by executing the command Project->Build Target (short cut F7). Object code or machine instructions contains these same commands in machine-readable-form. Most assembly source code is one-to-one with the object code that is executed by the computer. For example, when programming in a high level language like C or Java, one line of a program can translate into several machine instructions. In contrast, one line of assembly code usually translates to exactly one machine instruction. The assembler/compiler may also produce a listing file, which is a human-readable output showing the addresses and object code that correspond to each line of the source program. The target specifies the platform on which we will be running the object code. When testing software with the simulator, we choose the Simulator as the target. When simulating, there is no need to download, we simply launch the simulator by executing the Debug->Start Debug Session command. The simulator is an easy and inexpensive way to get started on a project. However, its usefulness will reduce as the I/O becomes more complex.

In a real system, we choose the real microcontroller via its JTAG debugger as the target. In this way the object code is downloaded into the EEPROM of the microcontroller. Most microcontrollers contain built-in features that assist in programming their EEPROM. In particular, we will use the JTAG debugger connected via a USB cable to download and debug programs. The JTAG is both a loader and a debugger. We program the EEPROM by executing the Flash-

>Download command. After downloading we can start the system by hitting the reset button on the board or we can debug it by executing Debug->Start Debug Session command in the uVision® IDE. In contrast, the loader on a general purpose computer typically reads the object code from a file on a hard drive or CD and stores the code in RAM. When the program is run, instructions are fetched from RAM. Since RAM is volatile, the programs on a general purpose computer must be loaded each time the system is powered up.

For embedded systems, we typically perform initial testing on a simulator. The process for developing applications on real hardware is identical except the target is switched from a simulated microcontroller to the real microcontroller. It is best to have a programming reference manual handy when writing assembly language.

A description of each instruction can also be found by searching the Contents page of the help engine included with the ARM Keil™ uVision® or TI CCStudio applications. There are a lot of settings required to create a software project from scratch. I strongly suggest those new to the process first run lots of existing projects. Next, pick an existing project most like your intended solution, and then make a copy of that project. Finally, make modifications to the copy a little bit at a time as you morph the existing project into your solution. After each modification verify that it still runs. If you take a project that runs, make hundreds of changes to it, and then notice that it no longer runs, you will not know which of the many changes caused the failure.

The objective of software development is to translate a desired set of actions into a clear set of commands that the computer executes. It is simple when the desired actions occur as a sequence of commands. Most of the time software asks the machine to execute one command after another. We write software as an ordered list of instructions one after another from the top to the bottom of the page, and the machine executes them in this order. The complexity occurs when we need to make decisions (branch), execute subtasks (functions), and perform multiple tasks concurrently or in parallel (interrupts and distributed systems). To handle these complexities we use instructions that are exceptions to this ″one after another″ rule (Figure 2.9). Interrupt are triggered by hardware events, causing the software interrupt service routine (shown as Clock in Figure 2.9) to be executed. A bus fault occurs if the software accesses an unimplemented memory location or accesses an I/O that is not properly configured. Bus faults are usually caused by software bugs.

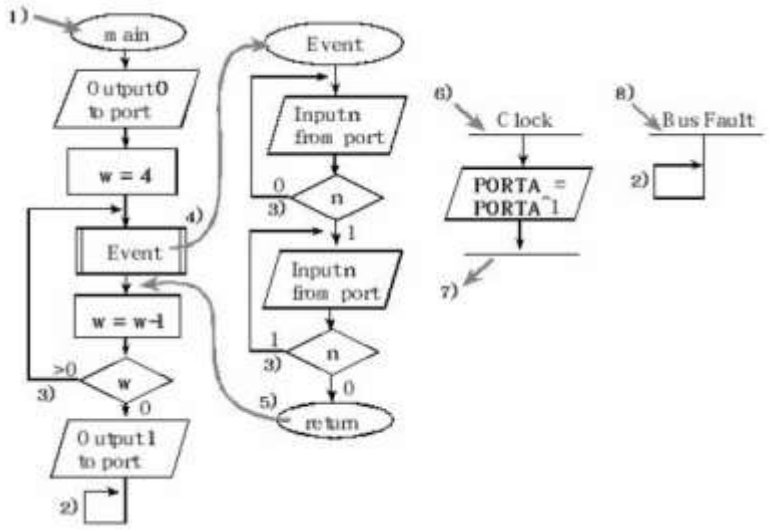

Figure 2.9. Flowchart showing examples of execution that breaks the ″one after another rule″.

These exceptions, as numbered in Figure 2.9, include

1) The computer uses the reset vector to decide where to start (reset or power on).

2) There can be an unconditional branch causing the software to always go to a spot.

3) A conditional branch will go to a spot if a certain condition is true.

4) A function call will cause the software to go execute the code for that function.

5) A return from function will return to the place that called the function.

6) An interrupt will interrupt execution and begin an interrupt service routine.

7) A return from interrupt will return to the place where it was before the interrupt.

8) A hardware or software mistake will cause a bus fault and stop execution.

4. ARM Cortex-M Assembly Language

This section focuses on the ARM ® Cortex™-M assembly language. There are many ARM ® processors, and this lecture focuses on Cortex -M microcontrollers, which executes Thumb ® instructions extended with Thumb-2 technology.

4.1) Syntax

Assembly language instructions have four fields separated by spaces or tabs. The **label field** is optional and starts in the first column and is used to identify the position in memory of the current

instruction. You must choose a unique name for each label. The **opcode field** specifies the processor command to execute. The **operand field** specifies where to find the data to execute the instruction. Thumb instructions have 0, 1, 2, 3, or 4 operands, separated by commas. The **comment field** is also optional and is ignored by the assembler, but it allows you to describe the software making it easier to understand. You can add optional spaces between operands in the operand field. However, a semicolon must separate the operand and comment fields. Good programmers add comments to explain the software.

```
Label   Opcode   Operands      Comment
Func    MOV      R0, #100      ; this sets R0 to 100
        BX LR                  ; this is a function return
```

When describing assembly instructions we will use the following list of symbols

Ra Rd Rm Rn Rt and **Rt2** represent registers

{Rd,} represents an optional destination register

#imm12 represents a 12-bit constant, 0 to 4095

#imm16 represents a 16-bit constant, 0 to 65535

operand2 represents the flexible second operand

{cond} represents an optional logical condition as listed in Table 2.2

{type} encloses an optional data type as listed in Table 2.3

{S} is an optional specification that this instruction sets the condition code bits

Rm {, shift} specifies an optional shift on **Rm**

Rn {, #offset} specifies an optional offset to **Rn**

For example, the general description of the addition instruction

ADD {cond} {Rd,} Rn, #imm12 could refer to either of the following examples.

ADD R0,#1 ; R0=R0+1 or ADD R0,R1,#10 ; R0=R1+10

Table 2.2 shows the conditions **{cond}** that we will use for conditional branching. **{cond}** used on other instructions must be part of an if-then (IT) block, explained in next Section.

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or **HS**	C = 1	Higher or same, unsigned ≥
CC or **LO**	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N ≠ V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 or N ≠ V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix specified

Table 2.2. Condition code suffixes used to optionally execution instruction.

It is much better to add comments to explain how or even better why we do the action. Good comments also describe how the code was tested and identify limitations. The assembly **source code** is a text file (with Windows file extension **.s**) containing a list of instructions.

If register R0 is an input parameter, the following is a function that will return in register R0 the value (100*input+10).

```
Func  MOV    R1,#100      ; R1=100
      MUL    R0,R0,R1     ; R0=100*input
      ADD    R0,#10       ; R0=100*input+10
      BX LR               ; return 100*input+10
```

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. This means instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0. The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

Address	Object code	Label	Opcode	Operand	comment
0x000005E2	**F04F0164**	**Func**	**MOV**	**R1,#0x64**	**; R1=100**
0x000005E6	**FB00F001**		**MUL**	**R0,R0,R1**	**; R0=100*input**
0x000005EA	**F100000A**		**ADD**	**R0,R0,#0x0A**	**; R0=100*input+10**
0x000005EE	**4770**		**BX LR**		**; return 100*input+10**