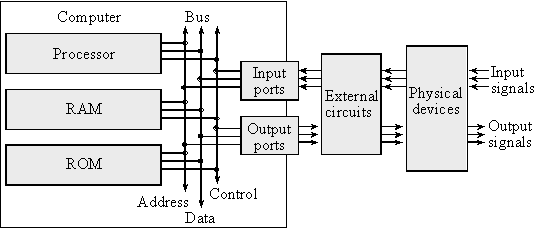
The common bus in Figure 2.7 defines the von Neumann architecture. Computers are not intelligent. Rather, you are the true genius. Computers are electronic idiots. They can store a lot of data, but they will only do exactly what we tell them to do. Fortunately, however, they can execute our programs quite quickly, and they don’t get bored doing the same tasks over and over again. **Software** is an ordered sequence of very specific instructions that are stored in memory, defining exactly what and when certain tasks are to be performed. It is a set of instructions, stored in memory, that are executed in a complicated but well-defined manner. The**processor** executes the software by retrieving and interpreting these instructions one at a time. A**microprocessor** is a small processor, where small refers to size (i.e., it fits in your hand) and not computational ability. For example, Intel Xeon E7, AMD Fusion, and Sun SPARC are microprocessors. AnARM®Cortex™-M microcontroller includes a processor together with the bus and some peripherals.



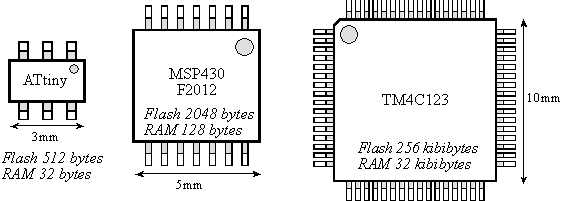
*Figure 2.7. The basic components of a von Neumann computer include processor, memory and I/O.*

A **microcomputer** is a small computer, where again small refers to size (i.e., you can carry it) and not computational ability. For example, a desktop PC is a microcomputer. Small in this context describes its size not its computing power. Consequently, there can be great confusion over the term microcomputer, because it can refer to a very wide range of devices from a PIC12C508, which is an 8-pin chip with 512 words of ROM and 25 bytes RAM, to the most powerful I7-based personal computer.

A **port** is a physical connection between the computer and its outside world. Ports allow information to enter and exit the system. Information enters via the input ports and exits via the output ports. Other names used to describe ports are I/O ports, I/O devices, interfaces, or sometimes just devices. A **bus** is a collection of wires used to pass information between modules.

A very small microcomputer, called a **microcontroller**, contains all the components of a computer (processor, memory, I/O) on a single chip. As shown in Figure 2.8, the Atmel ATtiny, the Texas Instruments MSP430, and the Texas Instruments TM4C123 are examples of microcontrollers. Because a microcomputer is a small computer, this term can be confusing because it is used to describe a wide range of systems from a 6-pin ATtiny4 running at 1 MHz with 512 bytes of program memory to a personal computer with state-of-the-art 64-bit multi-core processor running at multi-GHz speeds having terabytes of storage.

The computer can store information in **RAM** by writing to it, or it can retrieve previously stored data by reading from it. RAMs are **volatile**; meaning if power is interrupted and restored the information in the RAM is lost. Most microcontrollers have **static RAM** (SRAM) using six metal-oxide-semiconductor field-effect transistors (MOS or MOSFET) to create each memory bit. Four transistors are used to create two cross-coupled inverters that store the binary information, and the other two are used to read and write the bit.



*Figure 2.8. A microcontroller is a complete computer on a single chip.*

Information is programmed into **ROM** using techniques more complicated than writing to RAM. From a programming viewpoint, retrieving data from a ROM is identical to retrieving data from RAM. ROMs are nonvolatile; meaning if power is interrupted and restored the information in the ROM is retained. Some ROMs are programmed at the factory and can never be changed. A Programmable ROM (PROM) can be erased and reprogrammed by the user, but the erase/program sequence is typically 10000 times slower than the time to write data into a RAM. Some PROMs are erased with ultraviolet light and programmed with voltages, while electrically erasable PROM (EEPROM) are both erased and programmed with voltages. We cannot program ones into the ROM. We first erase the ROM, which puts ones into the entire memory, and then we program the zeros as needed. **Flash ROM** is a popular type of EEPROM. Each flash bit requires only two MOSFET transistors. The input (gate) of one transistor is electrically isolated, so if we trap charge on this input, it will remain there for years. The other transistor is used to read the bit by sensing whether or not the other transistor has trapped charge. In regular EEPROM, you can erase and program individual bytes. Flash ROM must be erased in large blocks. On many of Stellaris family of microcontrollers, we can erase the entire ROM or just a 1024-byte block. Because flash is smaller than regular EEPROM, most microcontrollers have a large flash into which we store the software. For all the systems in this class, we will store instructions and constants in flash ROM and place variables and temporary data in static RAM.

Figure 2.9 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. The Cortex-M instruction set combines the high performance typical of a 32-bit processor with high code density typical of 8-bit and 16-bit microcontrollers. Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface. On the Cortex-M4 there is a second I/O bus for high-speed devices like USB. There are many sophisticated debugging features utilizing the DCode bus. The nested vectored interrupt controller (NVIC) manages **interrupts**, which are hardware-triggered software functions. Some internal peripherals, like the NVIC 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 latency.

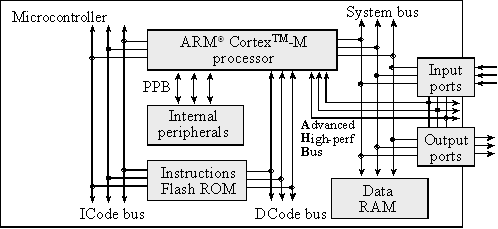


Figure 2.9. Harvard architecture of an ARM®Cortex-M-based microcontroller.

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, which we will see later in this chapter.

The registers are depicted in Figure 2.12. 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. Interrupts are covered in Chapter 12. 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.

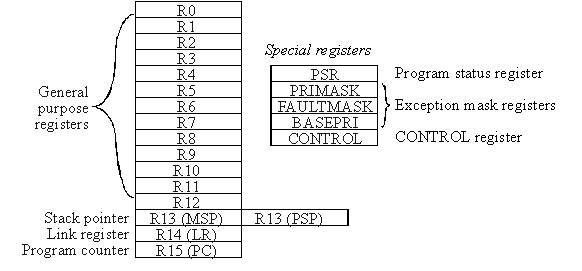


Figure 2.12. Registers on the ARM®Cortex-M processor.

The ARM Architecture **Procedure Call Standard**, AAPCS, part of the ARM **Application Binary Interface** (ABI), uses registers R0, R1, R2, and R3 to pass input parameters into a C function. Also according to AAPCS we place the return parameter in Register R0. In this class, the SP will always be the main stack pointer (MSP), not the Process Stack Pointer (PSP).

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 2.13. 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, and the **V bit** signifies signed overflow.  The **Q bit** indicates that “saturation” has occurred – while you might want to look it up, saturated arithmetic is beyond the scope of this class.

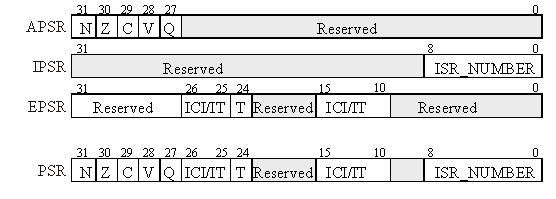


Figure 2.13. The program status register of the ARM®Cortex-M processor.

The **T bit** will always be 1, indicating the ARM®Cortex™-M processor is executing Thumb® instructions. The ISR\_NUMBER indicates which interrupt if any the processor is handling. 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 nonmaskable 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. The details of interrupt processing will be presented in subsequent chapters.

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**

**Observation:** A good comment explains why an operation is being performed, how it is used, how it can be changed, or how it was debugged. A bad comment explains what the operation does. The comments in the above two assembly lines are examples of bad comments.

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 as described in Section 3.4.2

**{cond}**represents an optional logical condition as listed in Table 2.10

**{type}** encloses an optional data type

**{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**

**ADD   R0,R1,#10    ; R0=R1+10**

Table 2.4 shows the conditions **{cond}** that we will use for conditional branching.

|  |  |  |
| --- | --- | --- |
| *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 and N ≠  V | Less than or equal, signed ≤ |
| **AL** | Can have any value | Always. This is the default when no suffix specified |

Table 2.4. 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. But for now we are learning what the instruction is doing, so in this chapter comments will describe what the instruction does. 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**

When we **build** a project all files are assembled or compiled then linked together. The address values shown in the listing are relative to the particular file being assembled. When the entire project is built, the files are linked together, and the **linker** decides exactly where in memory everything will be. After building the project, it can be downloaded, which programs the object code into flash ROM. You are allowed to load and execute software out of RAM. But for an embedded system, we typically place executable instructions into nonvolatile flash ROM. The listing you see in the debugger will specify the absolute address showing you exactly where in memory your variables and instructions exist.