Chapter 12 – Processor Structure and Function

12.1 Processor Organization (pp. 416-418)
12.2 Register Organization (pp. 418-423)

Processor Requirements

- Fetch Instruction
- Decode Instruction
- Fetch Data
- Process Data
- Write Data

Remember, a simplified view of the processor looks like the following:

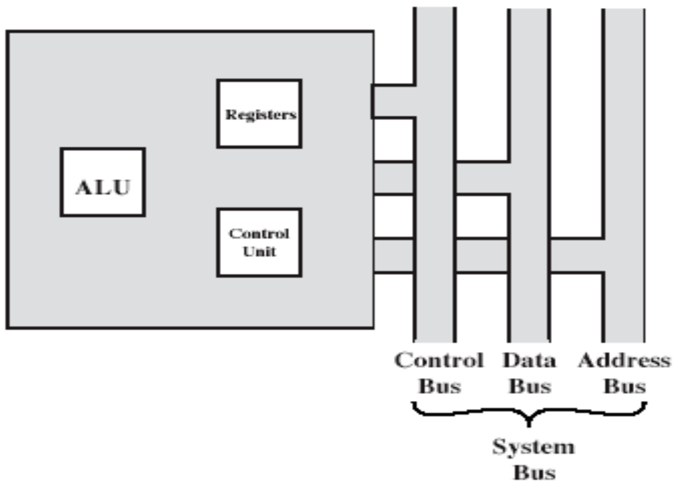


Figure 12.1 The CPU with the System Bus

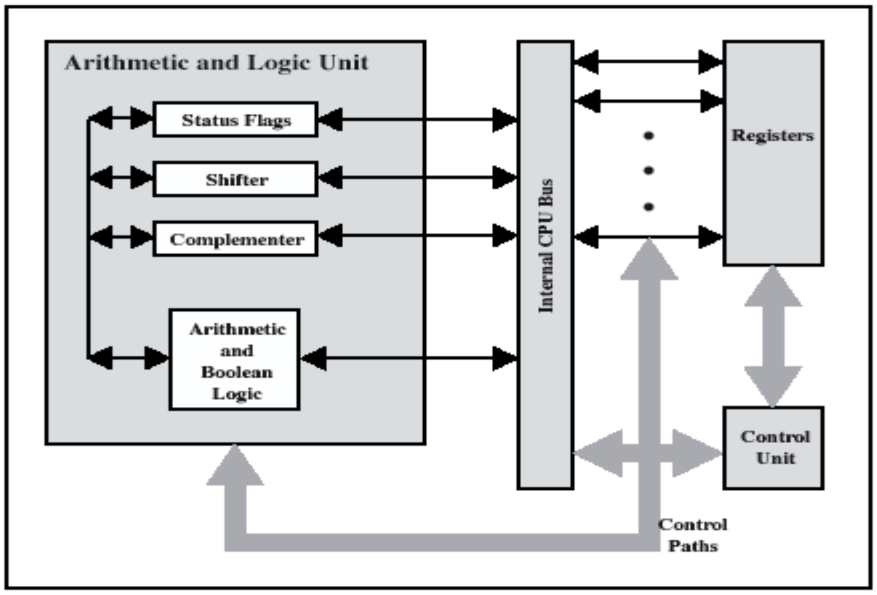


Figure 12.2 Internal Structure of the CPU

Registers fall into two categories:

User-visible – available to the programmer to minimize memory references

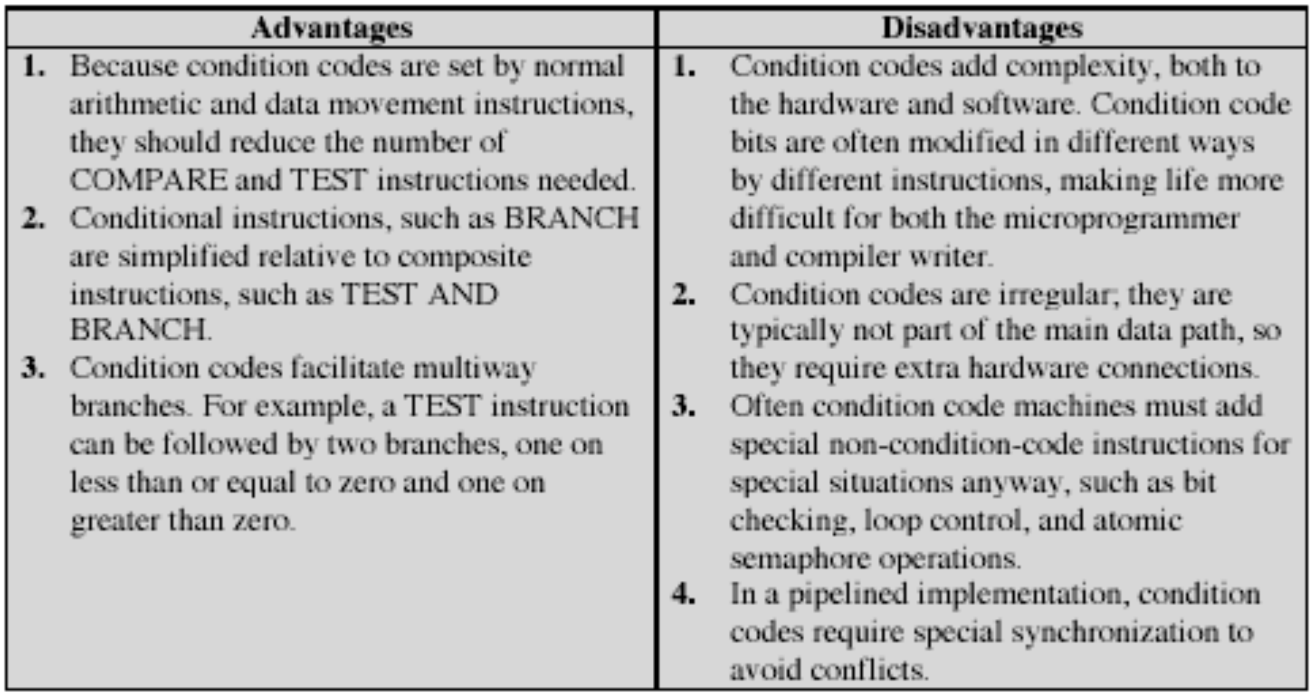
1. General Purpose – can be used for a variety of purposes
2. Data – used to hold data but cannot be used in the calculation of an operand address
3. Address – can be somewhat general purpose or used for a particular addressing mode (e.g. segment registers, index registers, stack pointer)
4. Condition Codes – bits set by the processor as the result of a particular operation

Some processors use condition codes and some do not.

Q1: The x86 uses condition codes. Give an example of several condition codes.

Q2: Show how a condition code can be used.

Table 12.1 Condition Codes



Advantages	Disadvantages
1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed. **2.** Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH. **3.** Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.	**1.** Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer. **2.** Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. **3.** Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations. **4.** In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

Control and Status Registers
Used by the processor's control unit and by privileged OS programs to control program execution.

We've already discussed uses for the:

- Program Counter (PC)
- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Buffer Register (MBR)

Remember, the processor updates the PC after each instruction fetch to point to the next instruction to execute.

Consider the following x86 assembly language program:

```
13CF:0100 B80000        MOV     AX,0000
13CF:0103 BB0000        MOV     BX,0000
13CF:0106 40            INC     AX
13CF:0107 01C3          ADD     BX,AX
13CF:0109 3D0A00        CMP     AX,000A
13CF:010C 75F8          JNZ     0106
13CF:010E 90            NOP
```

Q3: What does the above program do?

Q4: What is the initial value of the PC?

Q5: What is the PC called in the x86 world?

Q6: What is the value of the PC after the first assembly language statement is executed?

Q7: Which of the above statements affect the condition codes?

Q8: Why is the machine language for JNZ 0106 equal to 75F8?

Q9: How is the PC updated after execution of the statement JNZ 0106?

As for register organization, consider the following processors:

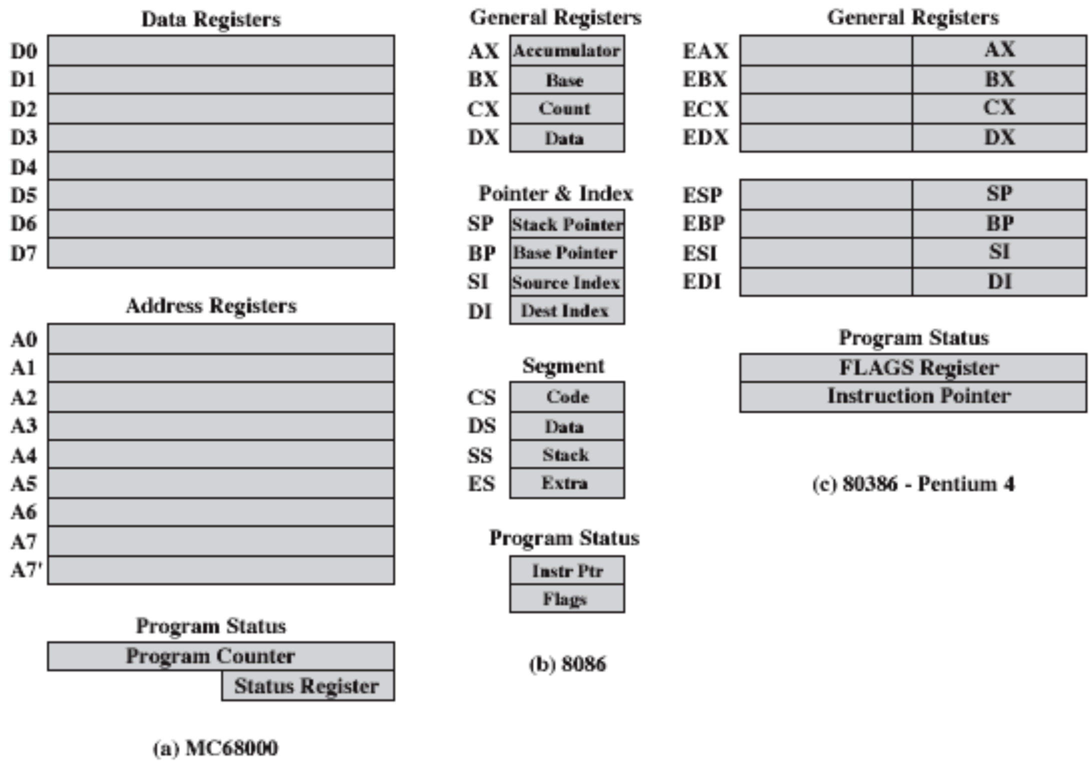


Figure 12.3 Example Microprocessor Register Organizations

Problem

Fill in the blanks in the machine language code below. Check your results using debug. There are no labels using debug but an examination of some sample debug code will show you how to jump to a location.

```
01ff: 0123  90        top:      nop
01ff: 0124  41                  inc cx
01ff: 0125  03 d1               add dx,cx
01ff: 0127  83 fa 14            cmp dx,20
01ff: 012a  74 __               je out1
01ff: 012c  41                  inc cx
01ff: 012d  48                  dec ax
01ff: 012e  75 __               jne top
01ff: 0130  90        out1:     nop
```

MIPS

In the early 1980s, a new trend in the design of processors began with the RISC (Reduced Instruction Set Computer) machines. The central idea was that by speeding up the commonest simple instructions, one could afford to pay a penalty in the unusual case and make a large net gain in performance. In contrast CISC (Complex Instruction Set Computer) chips can execute many complicated instructions, at the expense of slowing down the simplest ones.

In 1980, a group at Berkeley, led by David Patterson and Carlo Sequin, began designing RISC chips. They coined the term RISC and named their processor RISC1. Slightly later, in 1981, across the San Francisco Bay at Stanford, John Hennessy designed and fabricated a somewhat different RISC chip which he called the MIPS (Microprocessor without Interlocking Pipeline Stages), a play on the MIPS performance measurement.

MIPS processors are quite powerful, and are the heart of the capabilities of SGI's graphics servers and workstations, which were used to produce the special effects in many Hollywood movies (for example the new version of Star Wars, Jurassic Park and Toy Story). MIPS processors are also used in the Nintendo 64 game machine. Because of its use in high-performance embedded systems, it is estimated that MIPS currently sells more microprocessors than Intel.

The organization of memory in MIPS systems is conventional. A program's address space is composed of three parts. At the bottom of the user address space (0x400000) is the text segment, which holds the instructions for a program. Above the text segment is the data segment, starting at 0x10000000. The stack is a last in, first out data structure which is needed to implement procedures, allowing programmers to structure software to make it easier to understand and reuse. The program stack resides at the top of the address space (0x7fffffff). It grows down, toward the data segment.

MIPS assembly language is a 3-address assembly language. Operands are either immediates or in registers.

There are 32 registers that we commonly use. Each is 32 bits wide. The registers are identified by an integer, numbered 0 - 31.

To reference a register as an operand, use the syntax $x, where x is the number of the register you want. Examples: $12, $15

There are some limitations on the use of the 32 32-bit registers. Due to conventions set by the simulator, and by the architecture, certain registers are used for special purposes. It is wise to avoid the use of those registers, until you understand how to use them properly.

- $0 is 0 (use as needed)
- $1 is used by the assembler -- do **not** use it in your programs.
- $2-7 are used by the simulator -- do not use them unless you know what they are for and how they are used.

- $26-27 Used to implement the mechanism for calling special procedures that do I/O and take care of other error conditions (like overflow)
- $29 is a stack pointer -- you are automatically allocated a stack (of words), and $29 is initialized to contain the address of the empty word at the top of the stack at the start of any program.

The MIPS processor also has 16 floating-point registers $f0 . . . $f15 to hold numbers in floating point form.

We will be using WinMips64 – an instruction set simulator.
http://www.computing.dcu.ie/~mike/winmips64.html