

MARIE: An Introduction to a Simple Computer

1- Register Transfer Notation

- We have seen that digital systems consist of many components, including ALUs, registers, memory, decoders, and control units.
- These units are interconnected by buses to allow information to flow through the system.
- The instruction set presented for MARIE in the preceding sections constitutes a set of machine level instructions used by these
- components to execute a program.
- Each instruction appears to be very simplistic; however, if you examine what actually happens at the component level, each instruction involves **multiple operations**.
- For example, the Load instruction loads the contents of the given memory location into the AC register. But, if we observe what is happening at the component level, we see that multiple "mini-instructions" are being executed.
- First, the address from the instruction must be loaded into the MAR.
- Then the data in memory at this location must be loaded into the MBR.
- Then the MBR must be loaded into the AC.
- These mini-instructions are called **micro operations** and specify the elementary operations that can be performed on data stored in registers.
- The symbolic notation used to describe the behavior of micro operations is called **register transfer notation (RTN) or register transfer language (RTL)**.
- We use the notation $M[X]$ to indicate the actual data stored at location X in memory, and \leftarrow to indicate a transfer of information.

- In reality, a transfer from one register to another always involves a transfer onto the bus from the source register, and then a transfer off the bus into the destination register.
- However, for the sake of clarity, we do not include these bus transfers, assuming that you understand that the bus must be used for data transfer.
- We now present the register transfer notation for each of the instructions in the ISA for MARIE.
- **Load X**
This instruction loads the contents of memory location X into the **AC**. However, the address X must first be placed into the **MAR**. Then the data at location $M[\mathbf{MAR}]$ (or address X) is moved into the **MBR**. Finally, this data is placed in the **AC**.

$$\mathbf{MAR} \leftarrow X$$

$$\mathbf{MBR} \leftarrow M[\mathbf{MAR}]$$

$$\mathbf{AC} \leftarrow \mathbf{MBR}$$

- Because the IR must use the bus to copy the value of X into the **MAR**, before the data at location X can be placed into the **MBR**, this operation requires two bus cycles.
- Therefore, these two operations are on separate lines to indicate they cannot occur during the same cycle.
- Because we have a special connection between the **MBR** and the **AC**, the transfer of the data from the **MBR** to the **AC** can occur immediately after the data is put into the **MBR**, without waiting for the bus.
- **Store X**
This instruction stores the contents of the **AC** in memory location X :

$$\mathbf{MAR} \leftarrow \mathbf{X}$$

$$\mathbf{MBR} \leftarrow \mathbf{AC}$$

$$\mathbf{M [MAR]} \leftarrow \mathbf{MBR}$$

- **Add X**

The data value stored at address X is added to the AC. This can be accomplished as follows:

$$\mathbf{MAR} \leftarrow \mathbf{X}$$

$$\mathbf{MBR} \leftarrow \mathbf{M [MAR]}$$

$$\mathbf{AC} \leftarrow \mathbf{AC + MBR}$$

- **Subt. X**

Similar to Add, this instruction subtracts the value stored at address X from the accumulator and places the result back in the AC:

$$\mathbf{MAR} \leftarrow \mathbf{X}$$

$$\mathbf{MBR} \leftarrow \mathbf{M [MAR]}$$

$$\mathbf{AC} \leftarrow \mathbf{AC - MBR}$$

- **Input**

Any input from the input device is first routed into the InREG. Then the data is transferred into the AC.

$$\mathbf{AC} \leftarrow \mathbf{InREG}$$

- **Output**

This instruction causes the contents of the AC to be placed into the OutREG, where it is eventually sent to the output device.

$$\mathbf{OutREG} \leftarrow \mathbf{AC}$$

- **Halt**

No operations are performed on registers; the machine simply ceases execution.

- **Skipcond**

Recall that this instruction uses the bits in positions 10 and 11 in the address field to determine what comparison to perform on the AC.

Depending on this bit combination, the AC is checked to see whether it is negative, equal to zero, or greater than zero. If the given condition is true, then the next instruction is skipped. This is performed by incrementing the PC register by 1.

if IR[11–10] = 00 then {if bits 10 and 11 in the IR are both 0}

If AC < 0 then PC ← PC+1

else If IR[11–10] = 01 then {if bit 11 = 0 and bit 10 = 1}

If AC = 0 then PC ← PC + 1

else If IR[11–10] = 10 then {if bit 11 = 1 and bit 10 = 0}

If AC > 0 then PC ← PC + 1

- **If the bits in positions ten and eleven are both ones, an error condition results.**

- However, an additional condition could also be defined using these bit values.

- **Jump X**

This instruction causes an unconditional branch to the given address X. Therefore to execute this instruction, X must be loaded into the PC.

$$\mathbf{PC \leftarrow X}$$

In reality the lower or least significant 12 bits of the instruction register (or IR[11–0]) reflect the value of X. So this transfer is more accurately depicted as:

$$\mathbf{PC \leftarrow IR [11-0]}$$

- However, we feel that the notation $PC \leftarrow X$ is easier to understand and relate to the actual instructions, so we use this instead.
- Register transfer notation is a symbolic means of expressing what is happening in the system when a given instruction is executing.
- RTN is sensitive to the data path, in that if multiple micro operations must share the bus, they must be executed in a sequential fashion, one following the other.

2- INSTRUCTION PROCESSING

- All computers follow a basic machine cycle: the fetch, decode, and execute cycle.

2.1 The Fetch–Decode–Execute Cycle

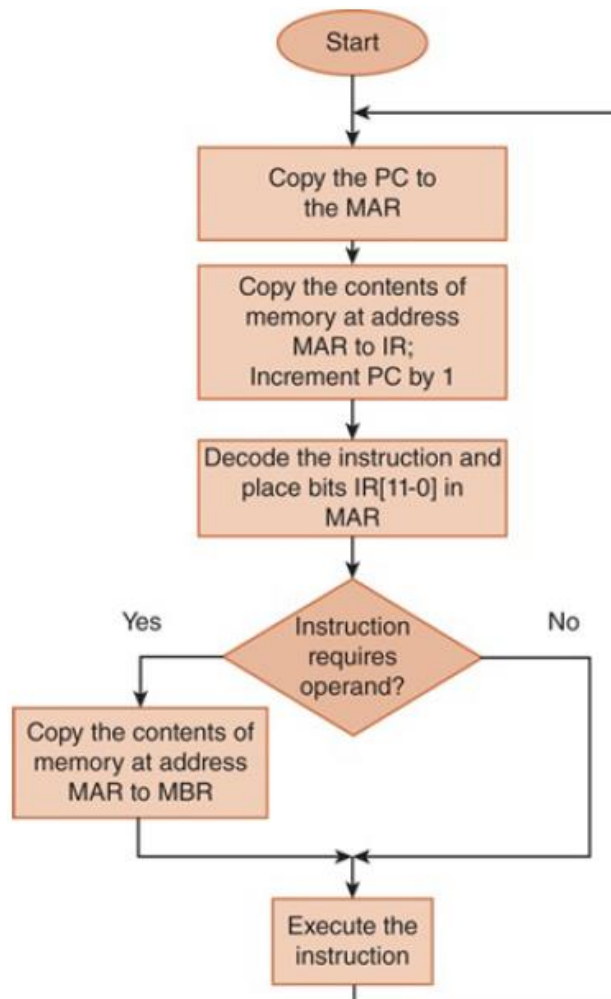
- The fetch–decode–execute cycle represents the steps that a computer follows to run a program.
- The CPU fetches an instruction (transfers it from main memory to the instruction register), decodes it (determines the opcode and fetches any data necessary to carry out the instruction), and executes it (performs the operation[s] indicated by the instruction).
- Notice that a large part of this cycle is spent copying data from one location to another.
- When a program is initially loaded, the address of the first instruction must be placed in the PC.
- The steps in **this cycle**, which take place in specific clock cycles, are listed below.
- Note that Steps 1 and 2 make up the fetch phase, Step 3 makes up the decode phase, and Step 4 is the execute phase.
 1. Copy the contents of the PC to the MAR: $MAR \leftarrow PC$.
 2. Go to main memory and fetch the instruction found at the address in the MAR, placing this instruction in the IR; increment PC by 1 (PC now

points to the next instruction in the program): $IR \leftarrow M[MAR]$ and then $PC \leftarrow PC + 1$. (Note: Because MARIE is word addressable, the PC is incremented by 1, which results in the next word's address occupying the PC. If MARIE were byte addressable, the PC would need to be incremented by 2 to point to the address of the next instruction, because each instruction would require 2 bytes.)

3. Copy the rightmost 12 bits of the IR into the MAR; decode the leftmost 4 bits to determine the opcode, $MAR \leftarrow IR[11-0]$, and decode $IR[15-12]$.

4. If necessary, use the address in the MAR to go to memory to get data, placing the data in the MBR (and possibly the AC), and then execute the instruction $MBR \leftarrow M[MAR]$ and execute the actual instruction.

- This cycle is illustrated in the flowchart in Figure 4.11.



- Note that computers today, even with large instruction sets, long instructions, and huge memories, can execute millions of these fetch–decode–execute cycles in the blink of an eye.

2.2 Interrupts and the Instruction Cycle

- All computers provide a means for the normal fetch–decode–execute cycle to be interrupted.
- These interruptions may be necessary for many reasons, including a program error (such as division by 0, arithmetic overflow, stack overflow, or attempting to access a protected area of memory); a hardware error (such as a memory parity error or power failure); an I/O completion (which happens when a disk read is requested and the data transfer is complete); a user interrupt (such as hitting Ctrl-C or Ctrl-Break to stop a program); or an interrupt from a timer set by the operating system (such as is necessary when allocating virtual memory or performing certain bookkeeping functions).
- All of these have something in common: They interrupt the normal flow of the fetch–decode–execute cycle and tell the computer to stop what it is currently doing and go do something else. They are, naturally, called interrupts.

3- A SIMPLE PROGRAM

- Consider the simple MARIE program given below. We show a set of mnemonic instructions stored at addresses 100 - 106 (hex):

Hex Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0010000100000110	2106
103	Halt	0111000000000000	7000
104	0023	000000000100011	0023
105	FFE9	111111111101001	FFE9
106	0000	0000000000000000	0000

- Let's look at what happens inside the computer when our program runs.
- This is the **LOAD 104** instruction:

a) Load 104

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	-----	-----	-----	-----
Fetch	MAR ← PC	100	-----	100	-----	-----
	IR ← M[MAR]	100	1104	100	-----	-----
	PC ← PC + 1	101	1104	100	-----	-----
Decode	MAR ← IR[11-0]	101	1104	104	-----	-----
	(Decode IR[15-12])	101	1104	104	-----	-----
Get operand	MBR ← M[MAR]	101	1104	104	0023	-----
Execute	AC ← MBR	101	1104	104	0023	0023

- Our second instruction is **ADD 105**.

b) Add 105

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Decode	MAR ← IR[11-0]	102	3105	105	0023	0023
	(Decode IR[15-12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	AC ← AC + MBR	102	3105	105	FFE9	000C

c) Store 106

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	000C
Fetch	MAR ← PC	102	3105	102	FFE9	000C
	IR ← M[MAR]	102	2106	102	FFE9	000C
	PC ← PC + 1	103	2106	102	FFE9	000C
Decode	MAR ← IR[11-0]	103	2106	106	FFE9	000C
	(Decode IR[15-12])	103	2106	106	FFE9	000C
Get operand	(not necessary)	103	2106	106	FFE9	000C
Execute	MBR ← AC	103	2106	106	000C	000C
	M[MAR] ← MBR	103	2106	106	000C	000C

4- A DISCUSSION ON ASSEMBLERS

- Mnemonic instructions, such as LOAD 104, are easy for humans to write and understand.
- They are impossible for computers to understand.
- Assemblers translate instructions that are comprehensible to humans into the machine language that is comprehensible to computers – We note the distinction between an assembler and a compiler:
- In assembly language, there is a one-to-one correspondence between a mnemonic instruction and its machine code.
- With compilers, this is not usually the case.
- Assemblers create an object program file from mnemonic source code in two passes.
- **During the first pass**, the assembler assembles as much of the program as it can, while it builds a symbol table that contains memory references for all symbols in the program.

- **During the second pass**, the instructions are completed using the values from the symbol table.
- Consider our example program (top).
- Note that we have included two directives **HEX** and **DEC** that specify the radix of the constants.

Address		Instruction	
	100	Load	X
	101	Add	Y
	102	Store	Z
	103	Halt	
X.	104	0023	
Y.	105	FFE9	
Z.	106	0000	

- During the first pass, we have a symbol table and the partial instructions shown at the bottom.
- After the second pass, the assembly is complete.

1 1 0 4
3 1 0 5
2 1 0 6
7 0 0 0
0 0 2 3
F F E 9
0 0 0 0

Address		Instruction	
	100	Load	X
	101	Add	Y
	102	Store	Z
	103	Halt	
X.	104	DEC	35
Y.	105	DEC	-23
Z.	106	HEX	0000

5- A DISCUSSION ON DECODING: HARDWIRED VERSUS MICROPROGRAMMED CONTROL

- The control unit, driven by the processor's clock, is **responsible for decoding the binary value in the instruction register and creating all necessary control signals; essentially, the control unit takes the CPU through a sequence of "control" steps for each program instruction.**
- More simply put, there must be control signals to assert the control lines on the various digital components in the system, such as the ALU and memory, to make anything happen.
- Each control step results in the control unit creating a set of signals (called a control word) that executes the appropriate microoperation.
- We can opt for one of two methods to ensure that all control lines are set properly.
- **The first approach, hardwired control**, physically connects the control lines to the actual machine instructions.
- The second approach, **microprogrammed control**, employs software consisting of microinstructions that carry out an instruction's microoperations.
- **Hardwired control is very fast, but the circuits required to do this are often very complex and difficult to design, and changes to the instruction set can result in costly updates to hardwired control.**
- **Microprogrammed control is much more flexible and allows easier and less costly updates to hardware, because the program simply needs to be updated; however, it is typically slower than hardwired control.**

5.1 Hardwired Control

- Hardwired control uses the bits in the instruction register to generate control signals by feeding these bits into fixed combinational logic gates.

- There are three essential components common to all hardwired control units:
- the **instruction decoder**, the **cycle counter**, and the **control matrix**.
- Depending on the complexity of the system, specialized registers and sets of status flags may be provided as well.
- Figure below illustrates a simplified control unit.

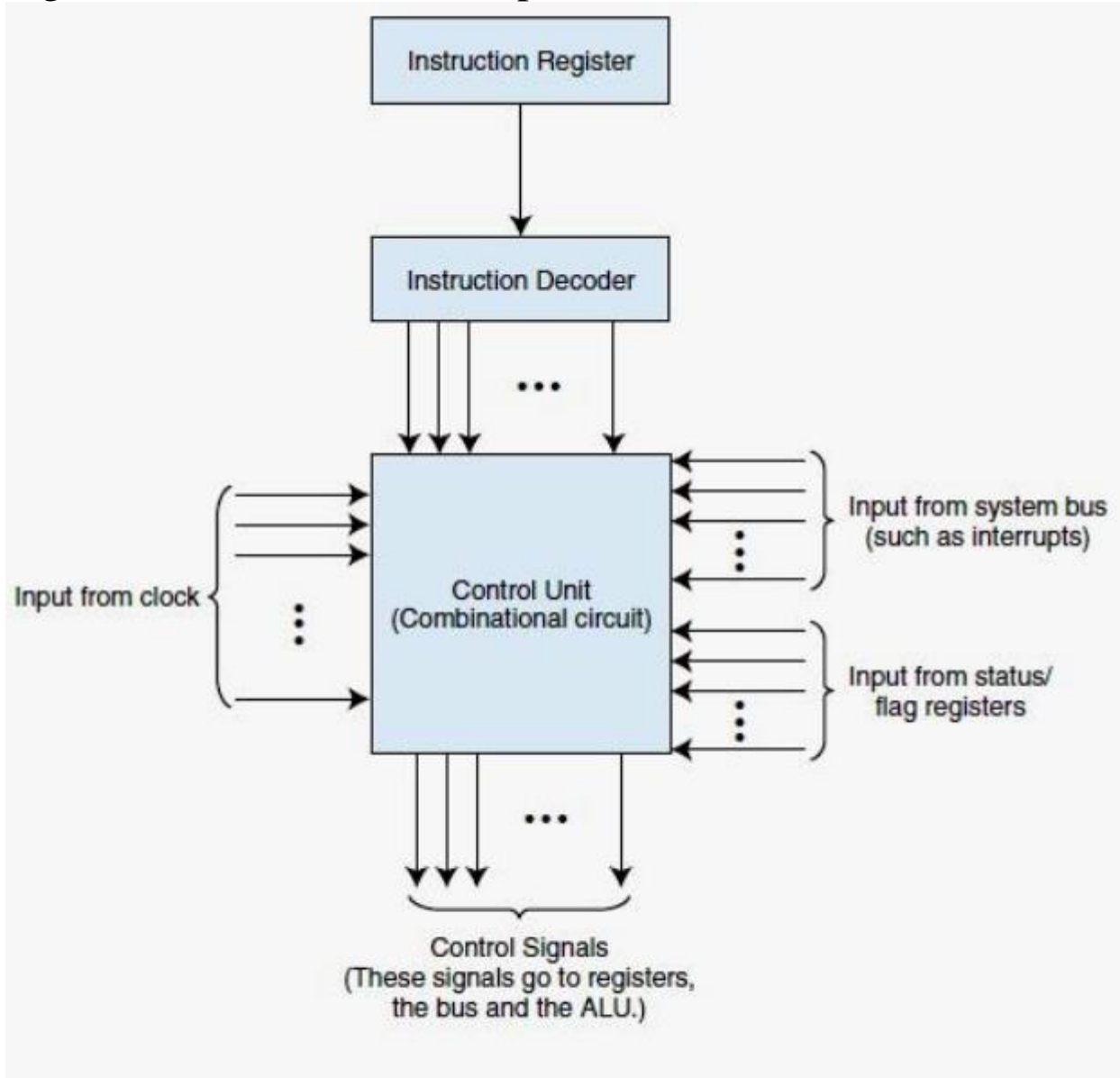


Figure 3.5 The hardwired control unit.

- The first essential component is the instruction decoder. Its job is to raise the unique output signal corresponding to the opcode in the instruction register. If we have a 4-bit opcode, the instruction decoder could have as many as 16 output signal lines.
- The advantage of hardwired control is that it is very fast. The disadvantage is that the instruction set and the control logic are tied together directly by complex circuits that are difficult to design and modify.
- If someone designs a hardwired computer and later decides to extend the instruction set, the physical components in the computer must be changed. This is prohibitively expensive, because not only must new chips be fabricated, but the old ones must be located and replaced.

5.2 Microprogrammed Control

- For a computer with a large instruction set, it might be virtually impossible to implement hardwired control.
- In microprogrammed control, instruction microcode produces the necessary control signals.
- A generic block diagram of a microprogrammed control unit is shown in Figure below.

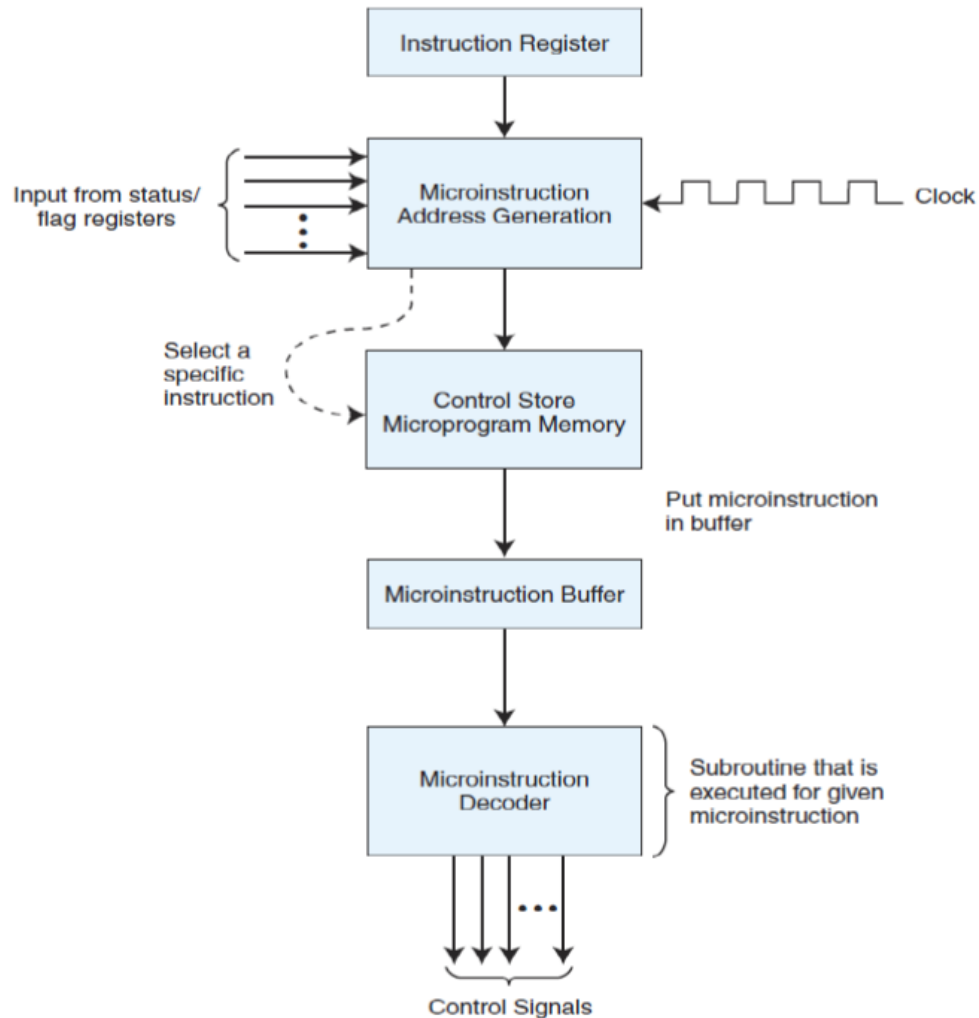


Figure 3.6 The microprogrammed control unit.

- All machine instructions are input into a special program, the microprogram, that converts machine instructions of 0s and 1s into control signals.
- The microprogram is essentially an interpreter, written in microcode, that is stored in firmware (ROM, PROM, or EPROM), which is often referred to as the control store.
- The great advantage of microprogrammed control is that if the instruction set requires modification, only the microprogram needs to be updated to match: No changes to the hardware are required.

- Thus, microprogrammed control units are less costly to manufacture and maintain.
- Because cost is critical in consumer products, microprogrammed control dominates the personal computer market.

6 REAL-WORLD EXAMPLES OF COMPUTER ARCHITECTURES

- MARIE shares many features with modern architectures, but it is not an accurate depiction of them.
- Two contemporary computer architectures better illustrate the features of modern architectures.
- The Intel architecture (the x86 and the Pentium families) and then follow with the MIPS architecture.
- Each member of the x86 family of Intel architectures is known as a CISC (complex instruction set computer) machine, whereas the Pentium family and the MIPS architectures are examples of RISC (reduced instruction set computer) machines.
- CISC machines have a large number of instructions, of variable length, with complex layouts.
- Many of these instructions are quite complicated, performing multiple operations when a single instruction is executed (e.g., it is possible to do loops using a single assembly language instruction).
- The basic problem with CISC machines is that a small subset of complex CISC instructions slows the systems down considerably.
- Designers decided to return to a less complicated architecture and to hardwire a small (but complete) instruction set that would execute extremely quickly.
- This meant it would be the compiler's responsibility to produce efficient code for the ISA.
- Machines utilizing this philosophy are called RISC machines.
- In RISC the number of instructions is reduced.

- However, the main objective of RISC machines is to simplify instructions so they can execute more quickly.
- Each instruction performs only one operation, they are all the same size, they have only a few different layouts, and all arithmetic operations must be performed between registers (data in memory cannot be used as operands).
- Virtually all new instruction sets (for any architectures) since 1982 have been RISC, or some sort of combination of CISC and RISC.