Instruction Set Architecture

Processor design involves the instruction set design and the organization of the processor.

Organization is concerned with the internal design of the processor, the design of the bus system and its interfaces, the design of memory and so on. Two machines may have the same ISA, but different organizations.

The organization is implemented in hardware and in turn, two machines with the same organization may have different hardware implementations, for example, a faster form of silicon technology may be used in the fabrication of the processor.

The **first point** that must be made about computer architecture is that there is no standard computer architecture, in the same way as there is no such thing as a standard house architecture or standard motor car design

Instruction Set

One of the crucial features of any processor is its **instruction set**, i.e. the set of machine code instructions that the processor can carry out. Each processor has its own unique instruction set specifically designed to make best use of the capabilities of that processor. The actual number of instructions provided ranges from a few dozens for a simple 8-bit microprocessor to several hundred for a 32-bit processor. However, it should be pointed out that a large instruction set does not necessarily imply a more powerful processor.

Many modern processor designs are so called **RISC** (Reduced Instruction Set Computer) designs which use relatively small instruction sets, in contrast to so called

CISC (Complex Instruction Set Computer) designs such as machines based on the Intel 8086 and Motorola 68000 microprocessor families.

Classification of Instructions

The actual instructions provided by any processor can be broadly classified into the following groups:

• **Data movement instructions**: These allow the processor move data between registers and between memory and registers (e.g. 8086 mov, push, pop instructions).

• **Transfer of control instructions:** These are concerned with branching for loops and conditional control structures as well as for handling subprograms (e.g. 8086 je , jg , jmp, call, ret instructions).

• **Arithmetic/logical instructions:** These carry out the usual arithmetic and logical operations (e.g. 8086 cmp, add, sub, inc, and, or, xor instructions).

ADD AX, BX ; AX=AX+BX

SUB AX,BX ; AX=AX-BX

INC AX ; AX=AX+1

DEC AX; AX=AX-1

CMP AX,BA ; COMPARE AX with BX

AND AX, BX ; AX=AX AND BX

OR AX, BX ; AX = AX OR BX

XOR AX,BX ; AX=AX XOR BX

• **Input/output instructions:** These are used for carrying out I/O (e.g. 8086 in, out instructions)

IN AL,DX; Read byte from port in DX into AL

OUT DX,AL ; sent byte in AL to DX

Fixed and Variable Length Instructions

Instructions are translated to machine code. In some architectures all machine code instructions are the same length i.e. **fixed length**. In other architectures, different instructions may be translated into **variable** lengths in machine code.

This is the situation with 8086 instructions which range from one byte to a maximum of 6 bytes in length. Such instructions are called **variable length** instructions and are commonly used on **CISC** machines.

The advantage of using such instructions, is that each instruction can use exactly the amount of space it requires, so that variable length instructions reduce the amount of memory space required for a program.

On the other hand, it is possible to have **fixed length** instructions, each instruction has the same length. Fixed length instructions are commonly used with **RISC** processors such as the PowerPC and Alpha processors.

Thus the comparison between fixed and variable length instructions **is memory usage versus execution time**.

Fetch-Execute Cycle

The **fetch-execute cycle** is the core operational process of a CPU, outlining how instructions are fetched from memory, decoded, and executed continuously until the CPU is halted. Here's an explanation in the context of your **Simple.**

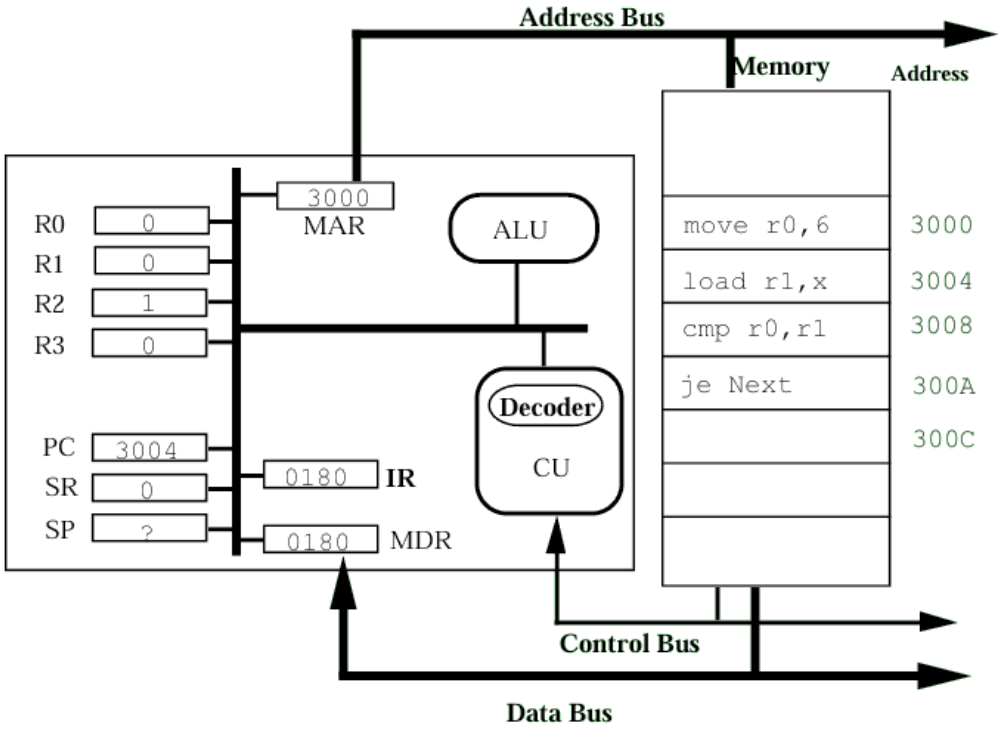

Fig. 1: SAM state after fetching move r0,6 (0180H in machine code)

Architecture Machine (SAM) example:

1. Fetch Phase:

The cycle starts with fetching the next instruction to be executed. The **program counter (PC)** holds the address of this instruction in memory.

- In your example, the instructions are stored starting at **3000H**.

- The current instruction being fetched is the move r0, 6 instruction, stored at **3000H** with machine code **0180H**.

2. Decode Phase:

Once the instruction is fetched, the **control unit** decodes it to determine what operation is required.

- For the move r0, 6 instruction, the CPU understands it needs to move the value **6** into **register r0**.

3. Execute Phase:

After decoding, the CPU executes the instruction. Here, it performs the operation specified by the instruction.

- The value **6** is loaded into **register r0**.

4. Update PC and Repeat:

After executing the instruction, the CPU updates the **program counter (PC)** to point to the address of the next instruction. The value of the **PC** is incremented based on the size of the current instruction.

- In this case, the PC increments by 4 (size of the move instruction), so it changes from **3004H** to **3008H**, pointing to the next instruction (cmp).

Example Instructions:

Address	Instruction	Machine Code
3000H	move r0, 6	0180H
3004H	load r1, x	2180H
3008H	cmp r0, r1	3180H

Current SAM State (After Fetching move r0, 6):

- **Registers:**
  - r0 = 0 (will be updated to 6 after execution)
  - r1 = 0
  - r2 = 1
  - r3 = 0
  - PC = 3004H (next instruction's address)
  - SP = ? (not relevant here)
  - SR = 0 (flags are reset)

Overall Process:

The fetch-execute cycle repeats for each instruction. After executing the move instruction, the next cycle will fetch the load r1, x instruction, update the PC to **3008H**, and continue executing the program.

Accessing Memory

In order to execute programs, a microprocessor fetches instructions from memory and executes them, fetching data from memory if it is required.

In Figure 1 we introduced two registers that we have not mentioned before namely the **memory address register, MAR and the memory data register, MDR**.

The MAR and MDR registers are used to communicate with memory.

The **MAR** register is used to store the address of the location in memory that is to be accessed for reading or writing.

When we retrieve information from memory we refer to the process **as reading** from memory.

When we store an item in memory, we refer to the process as **writing** to memory.

In either case, before we can access memory, we must specify the location we wish to access, i.e. the address of the location in memory. This address must be stored in the MAR register.

The **MDR** register is used either to store information that is to be written to memory or to store information that has been read from memory. The MDR register is connected to memory via the data bus whose function is to transfer information, to or from memory and other devices.

Reading from Memory

The following steps are carried out by the SAM microprocessor to read an item from memory. The item may be an instruction or a data operand.

1. The address of the item in memory is **stored** in the MAR register.

2. This address is **transferred** to the address bus.

3. The VMA line and R/W line of the control bus are used to indicate to memory that there is a **valid address** on the address bus and that a **read** operation is to be carried out.

4. Memory responds by **placing** the contents of the desired address on the data bus.

5. Memory **enables** the MOC line to indicate that the memory operation is complete, i.e. the data bus contains the required data.

6. The information on the data bus is **transferred** to the MDR register.

7. The information is **transferred** from the MDR register to the specified CPU register.

Writing to Memory

This procedure is similar to that for reading from memory:

1. The address of the item in memory is **stored** in the MAR register.
2. This address is **transferred** to the address bus.
3. The item to be written to memory is **transferred** to the MDR register.
4. This information is **transferred** to the data bus.
5. The VMA line and R/W line of the control bus are used to indicate to memory that there is a **valid address** on the address bus and that a **write** operation is to be carried out.
6. Memory responds by **placing** the contents of the data bus in the desired memory location.
7. memory uses the **MOC** line to indicate that the memory operation is complete, i.e. the data has been written to memory.