Computer Architecture

Computer architecture involves the design of computers.

Processor design involves the **instruction set** design and the **organisation** of the processor.

Instruction set architecture (ISA) describes the processor in terms of what the assembly language programmer sees, i.e. the instructions and registers.

Organisation 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 organisations.

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

Introduction

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

However, just as all cars have some basic features in common, so too do computers. In this section, we take a high level look at the components of computer architecture that are common to all computers, noting that any particular computer will differ in various details from the general model presented.

As we have seen earlier, computer programs are translated to machine code for execution by the CPU. Once a program has been loaded into the computer's memory (carried out by the operating system on our behalf), the program may then be executed.

This means that the CPU obeys the instructions making up the program and carries them out one at a time.

It is worth noting, at this point, the **primitive** nature of the CPU. The CPU does not *understand* programs, rather it *obeys* individual instructions.

6.2 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 dozen for a simple 8-bit microprocessor to several hundred for a 32-bit VAX 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 the VAX and 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). A '`move`' instruction and its variants is among the most frequently used instructions in an instruction set.

• **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). These are also commonly used instructions.

• **Arithmetic/logical** instructions: These carry out the usual arithmetic and logical operations (e.g. 8086 `cmp`, `add`, `sub`, `inc`, `and`, `or`, `xor` instructions). Surprisingly, these are not frequently used instructions, and when used, it is often in conjunction with a conditional jump instruction rather than for general arithmetic purposes. Note that we have included the `cmp` instruction with the arithmetic/logical instructions because it actually behaves like a `sub` instruction except it does not modify its destination register.

• **Input/output** instructions: These are used for carrying out I/O (e.g. 8086 `in`, `out` instructions) but a very common form of I/O called memory mapped I/O uses '`move`' instructions for I/O.

• **Miscellaneous** instructions (e.g. 8086 `int`, `sti`, `cti, hlt, nop`) for handling interrupts and such activities. The `hlt` instruction halts the processor and the `nop`[1] instruction does nothing at all! These instructions are again not that frequently used relative to data movement and transfer of control instructions. The `int` instruction could also be classified as a transfer of control instruction and interrupts are described in more detail below.

This is not the only way to classify instructions. For example, the arithmetic/logical instructions mentioned above may be classified as **operate** instructions. Operate instructions also include instructions that move data between registers and manipulate stacks. **Memory-access** instructions refer to those that transfer data between registers and memory.

[1] Most processors provide a non-operation (nop) instruction. Its execution takes one clock cycle, the purpose of which is simply to use time. It is used (frequently in a loop) to delay some time while the processor waits for an event to happen such as an I/O device to respond to an I/O request.

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, where as the name suggests, each instruction has the same length. Fixed length instructions are commonly used with **RISC** processors such as the PowerPC and Alpha processors.

Since each instruction occupies the same amount of space, every instruction must be long enough to specify a memory operand, even if the instruction does not use one. Hence, memory space is wasted by this form of instruction. The advantage of fixed length instructions, it is argued, is that they make the job of fetching and decoding instructions easier and more efficient, which means that they can be executed in less time than the corresponding variable length instructions.

Thus the comparison between fixed and variable length instructions comes down to the **classic computing trade off of memory usage versus execution time.**

In general, computer programs that execute very quickly tend to use larger amounts of storage, while programs to carry out the same tasks, that do not use so much storage, tend to take longer to execute.

Fetch-Execute Cycle

This is the fundamental operation of the processor. The CPU executes the instructions that it finds in the computers memory. In order to execute an instruction, the CPU must first **fetch** (transfer) the instruction *from memory* into one of its registers.

This is a non-trivial task requiring several steps and is described later.

The CPU then **decodes** the instruction, i.e. it decides which instruction has been fetched and finally it **executes** (carries out) the instruction.

The CPU then repeats this procedure, i.e. it fetches an instruction, decodes and executes it. This process is repeated continuously and is known as the **fetch-execute** cycle.

This cycle begins when the processor is switched on and continues until the CPU is halted (via a halt instruction, e.g. 8086 `hlt` instruction or the machine is switched off).

Instead of looking at the details of a particular microprocessor's architecture at this point, we will use a simple hypothetical microprocessor to explain the basic concepts of computer architecture.

We call the machine SAM (Simple Architecture Machine). Figure 1 illustrates the major components of SAM. It is a 16-bit microprocessor with 4 general purpose registers `r0` to `r3`, a program counter register `PC`, a stack pointer register `SP` and status register `SR`. The status register is made up of similar flags to the 8086 flags register, e.g. a zero flag, an overflow flag, a carry flag and so on.

The fetch-execute cycle operates by first fetching an instruction. The program counter register `PC` always contains the address of the next instruction to be executed.

Let us assume that a particular program has been loaded into memory and is currently being executed. Program execution has reached a certain point (the `move` instruction is being executed) and the three instructions of the program are listed in Example 1.

To illustrate how the fetch execute cycle operates, we will trace the execution of these instructions.

We assume that these instructions are stored in memory beginning at location `3000H`.

The instructions and their machine code equivalents (in hexadecimal) are listed below.

We use hexadecimal instead of binary as it is easier to work with but you must remember **that it is the binary form of the instructions that are actually stored in memory**.

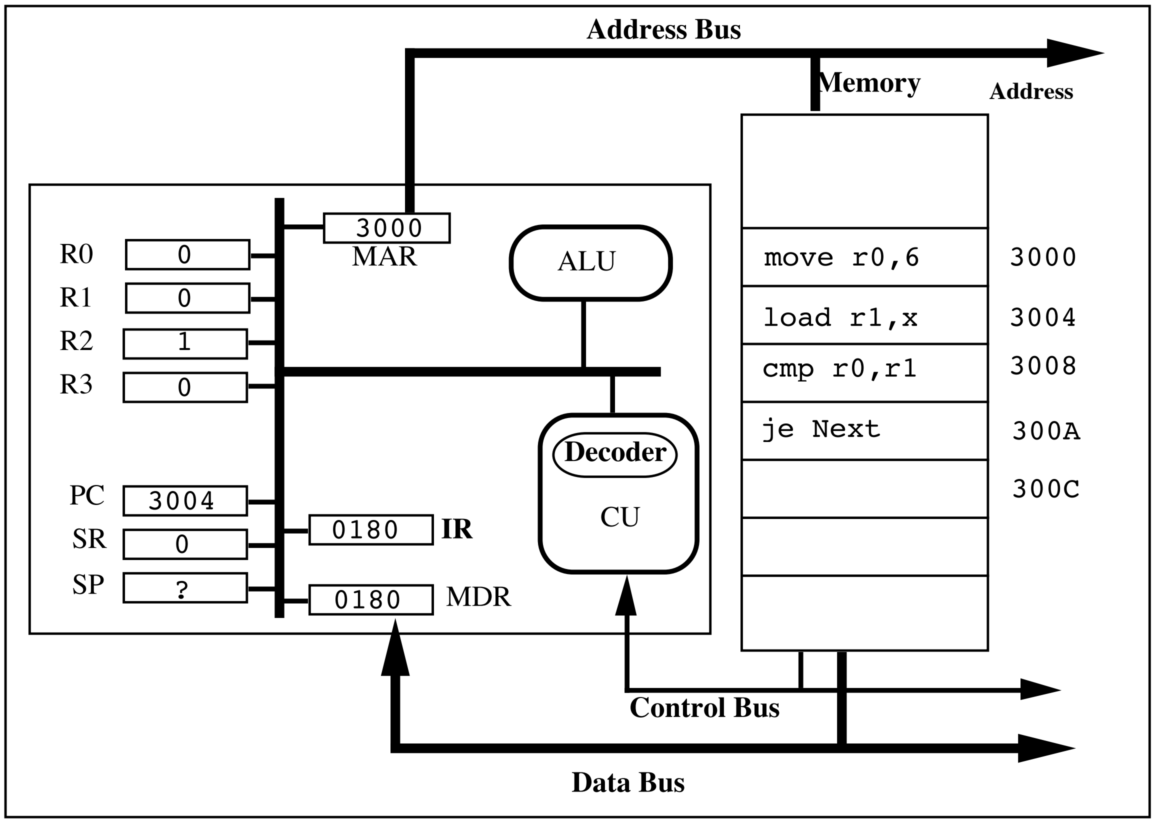

Fig. 1: SAM state after fetching `move r0,6` (`0180H` in machine code)

Example 1: The following is a SAL program fragment and its machine code version. It also shows the addresses of where the instructions are stored in memory.

The memory variable `x` is stored at location `0100H` in memory which contains the value `6`.

The label `Next` is `16` (`10H`) bytes forward from the `je` instruction.

SAL Code	**SAM Code**	**Memory Address**	
		Hex	**Decimal**
`move  r0, 6`	`0180H`	`3000H`	`4096`
	`0006H`	`3002H`	`4098`
`load  r1, x`	`0284H`	`3004H`	`4100`
	`0100H`	`3006H`	`4102`
`cmp   r0, r1`	`0F01H`	`3008H`	`4104`
`je    Next`	`2F10H`	`300AH`	`4106`
`.....`		`300CH`	`4108`
`.....`			

This code fragment assigns register `r0` the value `6`, assigns register `r1` the value of `x` (also `6`) and compares the values of registers `r0` and `r1`.

If the registers are equal, which they are in this case, control transfers to the instruction stored at `Next` not shown in this example.

The fetch-execute cycle operates by first fetching an instruction. The program counter register `PC` always contains the address **of the next instruction to be executed.**

For the code fragment under consideration, the `move` instruction is currently being executed. Thus, the program counter contains the value 3004H at the point where we begin tracing execution of the program i.e. the address of the next instruction, in this case the `load` instruction.

Having **executed** the move instruction, the `load` instruction is fetched.,

After fetching the load instruction, the control unit updates the program counter to point to the next instruction to be fetched. The control unit increments the program counter by the size of the current instruction, in this case the `PC` register is incremented by `4`, giving it the value `3008H`.

The program counter now points to the `cmp` instruction.

The `MDR` register (see later) contains the instruction just fetched from memory and this is transferred to the **instruction register**, **`IR`**.

The instruction register is another of the CPU's hidden registers which we have not encountered to date. It is logically part of the control unit and its function is to store an instruction so that it can be **decoded** for execution.

The load instruction is now executed and the fetch-execute cycle begins again.

Figure 1 illustrates the state of the SAM registers **after** the `move` instruction has been fetched. Only the relevant portions of memory are shown.

We assume that the general purpose registers have the value 0 except for `r2` which has the value 1. The `SP` register is shown to have "?" as its value to indicate that we are not interested in its contents, in this example. The status register, `SR`, has value 0, indicating that the flags are all set to 0.

Note: We show memory to contain text for illustrative purposes. memory will **always contain binary** and in this example, it will contain the machine code of the given instructions.

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`**. There are a number of such CPU registers that do not appear in the **programming model** of a CPU. We shall refer to these registers as **hidden** CPU registers to distinguish them from the programming model registers `R0` to `R4`, `PC`, `SR` and `SP` registers.

The `MAR` and `MDR` registers are used to communicate with memory (and other devices attached to the system bus).

In addition, Figure 1 shows the buses that allow the devices making up a SAM computer system communicate with each other.

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 `MAR` register is connected to memory via the **address bus** whose function is to transfer the address in the `MAR` register to memory. In this way the memory unit is informed as to which location is to be accessed.

The **address bus is a uni-directional bus**, i.e. information can only travel along it in a single direction, from the CPU to memory and other devices.

The `MAR` register is a 16-bit register like all the other SAM registers. This means that the maximum address it can contain is $2^{16} - 1$ (65,535) bytes, i.e. it can address up to 64Kb of memory.

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.
The **data bus is a bi-directional bus**, i.e. information can travel along it, both, to and from the CPU.

The control bus plays a crucial role in I/O. It carries control signals specifying what operation is to be carried out and to synchronise the transfer of information.

For example, one line of the control bus is the **read/write** (**R/W**) line which used to specify whether a read or write operation is to be carried out.
Another line is the **valid memory address** (**VMA**) line which indicates that the address bus now carries a valid memory address. This tells the memory unit when to look at the address bus to find the address of the location to be accessed. A third line is the **memory operation complete** (**MOC**) line which signals that the read/write operation has now completed. We should note at this point, that the other devices attached to the computer, such as I/O and storage devices, usually communicate with the CPU in a similar fashion to that described for communicating with memory

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.

We can see from the above descriptions (which have been simplified!) that accessing memory or any device is quite complicated from an implementation viewpoint. So, when an instruction such as

```
load   r1, i
```

to load a register with the contents of a memory variable is to be executed, a lot of work has to be carried out.

Firstly the instruction must be fetched from RAM, then the value of `i` must be fetched from RAM and finally the transfer of the value of `i` to register `r1` is carried out.

It is important to realise that every operation concerning memory involves either reading or writing memory.

Memory is a **passive** device. It can only store information. No processing can be carried out on information in memory. The information, stored in memory, must be transferred to a CPU register for processing and the result written back to memory.

So, for example, when an instruction such as the 8086 `inc` instruction is carried out to increment a memory variable (as in `inc memvar`), its execution involves **both** a memory read operation and a memory write operation.

Firstly, the value of `memvar` must be transferred to the CPU where it can be incremented by the ALU. This transfer is carried out via a memory read operation. Then, once this value has been incremented by the ALU, the new value of `memvar` must be written out to `memvar`'s address in memory, via a memory write operation.

Encoding Instructions in Machine Code

Instructions are represented in **machine code** as binary numbers in same way as all other information is represented in a computer system. We noted earlier that assembly language instructions for most processors are broadly similar and have the form:

```
    [label] operation   [operand ..]  [;comment]
```

The general form of a machine code instruction is illustrated in Figure 2 with the bits making up the instruction being grouped into **opcode** and **operand** fields.

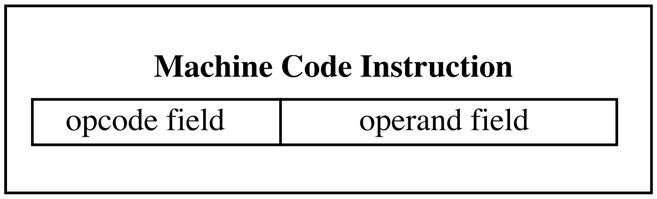

Figure 2: Machine code instruction format

The opcode field contains a binary code that specifies the operation to be carried out (e.g. add, jmp). Each operation has its own unique opcode. The operand field specifies the operand or operands that the operation is to be carried out on..

It should be emphasised that the instruction encoding for SAM is designed for illustration purposes. The aim is to keep it as simple as possible while remaining basically similar to the encoding of instructions on real processors. The reader is encouraged to look at ways the instructions could be more efficiently encoded.

Table 1 lists the opcodes of the commonly used SAM instructions in binary and hexadecimal.

Instruction	Binary Code	Hex Code
`move`	`0000 0001`	`01`
`load`	`0000 0010`	`02`
`store`	`0000 0011`	`03`
`add`	`0000 1000`	`08`
`sub`	`0000 1001`	`09`
`cmp`	`0000 1111`	`0F`
`jmp`	`0001 1111`	`1F`
`je`	`0010 1111`	`2F`

Table 1: SAM opcodes

The operand field of an instruction must be able to specify the registers, memory addresses or constants that the instruction is to operate on. SAM instructions have at most two operands. If there are two operands then one is always a register.

If a memory address is specified (e.g. in the case of a memory variable or label) then the instruction is encoded using 32-bits.

Since SAM has four general purpose registers we can represent them using 2-bit codes as follows:

`00` for `r0`
`01` for `r1`
`10` for `r2`
`11` for `r3`.

Thus, 4 bits are required to represent the two registers that may be used in an instruction. Bit numbers `0` and `1`, represent the source register and bit numbers `2` and `3` represent the destination register.

Example: Encoding of `load r1, X` where `X` refers to a memory variable stored at location `00FFH (255D)` in memory,

Instruction and binary encoding **Hex encoding**

(1)

```
     load     r1, X                  ; r1 = X

0000 0010 1000 01 00  0000 0000 1111 1111    0284 00ffH
                          (address of val)

          ←B-field>  ←    W-field     →
```

Explanation

This instruction is encoded using 32-bits. The opcode for `load` is `0000 0010` (`02H`), the destination register `r1` is encoded as `01`, while the source register is encoded as `00`, but is not used because the `load` instruction looks for its source operand in memory, hence bit `7` of the B-field is set to `1`, indicating a memory operand. Finally, the address of the memory variable `X` (`00ffH`) is stored in the W-field.