Device architecture

Table of Contents

Outline architecture....1
The stored program concept....1
Harvard and von Neumann architectures....2
Storage....2
Memory....3
Processors and memory....4
Processor components....4
Instruction execution....6
Machine code....7
Some processors....7
Machine code and assembler....8
Opcodes and operands....9
Common instruction types....9
Addressing mode....11
Instruction formats....11
Optimisations....12
Cache memory....12
Pipelining....13
Multiple cores....14
Clock speed....14
Word length and data bus....14
Instruction length....15
Address bus width....15

This section is about how computer hardware is organised and how it functions. This is at a level above that of logic gates.

This should be studied while doing practical work on assembly language programming.

Outline architecture

The stored program concept

The most basic idea is that a program, as a sequence of instructions, is held in memory.

How else could a computer be programmed? By having a set of 0 or 1 switches which would correspond to a program instruction. The user would set the switches for an instruction and press the Execute button. Then set switches for the next instruction, press Execute, and so on.

Clearly this would not work very well, and execution could be much faster if instructions were first loaded into memory, then executed at full speed.

Harvard and von Neumann architectures

Harvard architecture has separate memory units for programs and data.

von Neumann architecture has a single memory unit, used for both program and data.

Desktops, laptops, tablets and smart phones use von Neumann architecture.

Micro-controllers, used in medical devices, weapons, microwave ovens and dishwashers, often use Harvard architecture.

Storage

Devices store data in processor registers, in main memory and a file store.

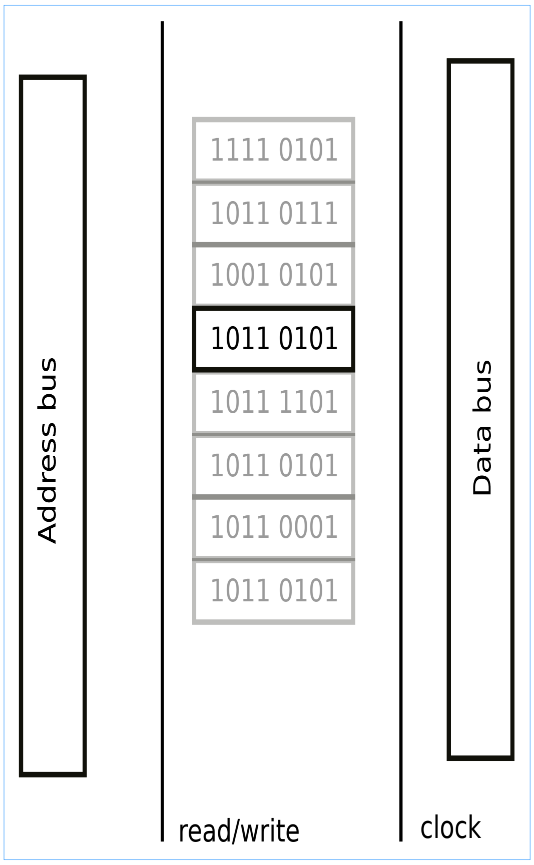
Registers are very fast, but there are usually less than 50 - maybe just 4. They are used by an instruction as it executes. A register might hold 64 bits = 8 bytes, so there might be a total of up to 400 bytes of register storage available.

Main memory (RAM and ROM) holds programs while they are executing, and associated data. This is medium fast - faster than the file store, but slower than processor registers. A device might have a few gigabytes of memory, so a few thousand million bytes.

File store might be on a magnetic disc (hard disc or disk), SSD, flash memory sticks, optical storage such as DVD, and it might be held in the cloud and accessed over the Internet. File store is used to hold files containing programs, data files of word-processed documents, spreadsheets, music mp3s and movie films. A file store would have storage measured in terabytes, so a thousand times the size of main memory, but be slower. File stores are non-volatile - they retain data when switched off.

Memory

Memory consists of a lot of cells or locations, each of which can hold 1 byte. Memory can do just two things. We can *write* data into memory (store it), and *read* data back out (fetch it back).



The cells are connected to a *data bus*. This is a set of parallel wires each carrying a 0 or 1. The data bus carries bytes to be written into memory, or bytes read out of memory and copied elsewhere (such as to a processor).

Each cell has a unique *address*. This is a binary pattern - a set of 1s and 0s. The address bus carries these binary patterns.

One wire carries a flag - a yes/no signal, which means do a read, or do a write.

Another wire carries a clock signal - a stream of 1 and 0 pulses, at around 1 GHz.

So if the read signal is set, on a clock pulse, memory decodes the address to select one cell, and copies the byte in that cell onto the data bus.

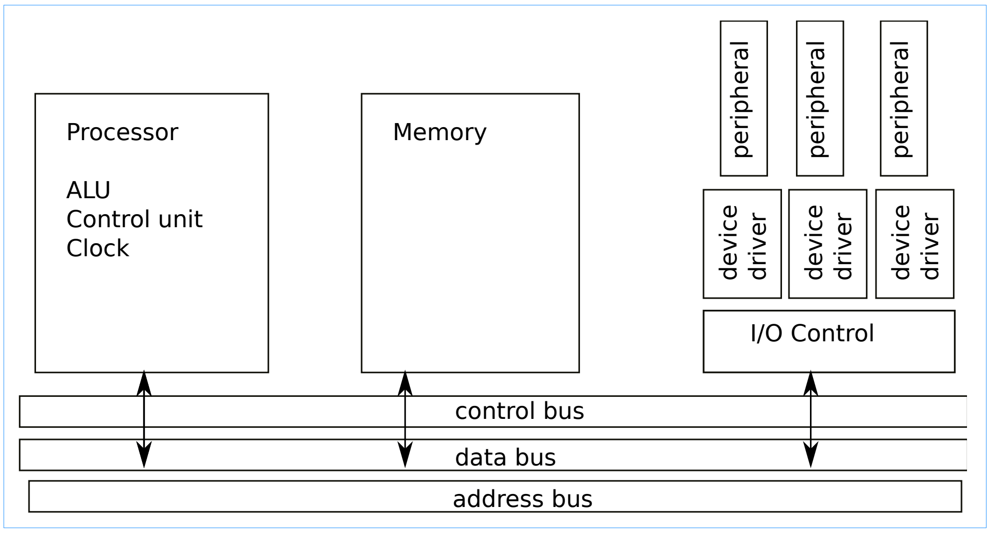
If write is set, on a clock pulse the address is decode and the data is copied into the cell selected.

This viewpoint is very simplified compared to real systems. Memory takes a while to respond, so a processor might wait a few clock pulses until the data is valid. The data bus is typically 64 bits wide (so can carry 8 bytes) so up to 8 bytes can be written or read by one instruction.

The binary patterns held in memory might be data, or program instructions.

Processors and memory

There is a processor and memory, connected by buses:



Also an input/output (I/O) control is connected to these buses, and in turn to peripheral devices such as mouse, keyboard, screen and file storage.

In outline, what happens is

1. Executing programs are held in memory, usually first having been loaded in from a filestore.
2. The processor *fetches* the next instruction from memory
3. *Decodes* it, meaning using digital logic to identify which instruction it is
4. *Executes* the instruction
5. *Repeats* by fetching the next instruction

This is usually called the *fetch-execute cycle*.

The instructions are *native code*, also called *machine code*.

Processor components

A processor contains registers. These store bit patterns and have logic gates around them to do various actions. Early processors had 4 bit registers. Current ones are 64 bit. They are faster than main memory. Registers could be thought of as sets of D-type latches.

Some registers have special purposes:

Current instruction register. This holds the current instruction while it is decoded and executed.

Program counter. This contains the address of the next instruction to be executed in memory. In other words it is a pointer to the next instruction. Usually this is incremented ready to fetch the next instruction. But for a *jump instruction*, it is loaded with a different address, to jump to that location.

Status register. This is also called the program status word or flags register. It contains a set of flags relating to the results of the last instruction. For example one flag would be the Plus flag, which is set if the last instruction gave a result greater than zero. This is used for what corresponds to an if statement in a high level language. For example

```
cmp x,y         ; compare values x and y - set
                ; flags as if doing x - y
jg newLocation  ; jump on greater than (zero).
                ; If the plus flag is set,go to
                ; newLocation
```

This is like

```
if x > y goto newLocation
```

in a high level language.

Other flags might be zero, parity, overflow, carry and others.

Memory buffer register. Values being stored into memory are first placed into this, then copied onto the data bus, over which it is actually sent to memory. When memory is being read, the data bus is first copied into this, then moved to wherever it is needed.

For example, the instruction

```
mov   num1, %rax
```

could move the value from address num1 in memory to register rax. This would go over the data bus to the memory buffer register, then through the processor into the rax register.

Memory address register. This would be a buffer onto the address bus. So when a value was being read from memory, the address to read would be put into the memory address register, then copied onto the address bus.

The processor *clock* puts a stream of 1 and 0 signals across the processor and down the control bus. Some instructions execute in one clock peried, but most take several. The clock will go at around 1 GHz.

The arithmetic and logic unit (ALU) can execute arithmetic and logic instructions.

The control unit decodes instructions and makes them execute.

Instruction execution

As an example, we look at one instruction:

```
add   num2, %rax
```

which adds the number from address num2 into the register rax, and puts the result into rax. The following happens:

1. The instruction has been fetched from memory, into the instruction register, and decoded.
2. num2 is an address. It is copied from the instruction onto the memory address register, then onto the address bus, and a read signalled on the control bus.
3. Memory responds and puts the value in address num2 onto the data bus.
4. This is copied onto the memory buffer register.
5. Then routed to the rax register, where it is added to the contents. The register is surrounded by full adders which can do addition.
6. The program counter is incremented, and we go back to step 1.

This will take several clock periods. The control unit will make it happen. In effect each machine code program is a small program in itself. This is sometimes called *micro-code*.

Test

1. What is Harvard architecture?

2. Explain the differences between processor registers, main memory and file store.

3. What is the purpose of the program counter?

4. In what ways is native code different from high level program code?

Machine code

A processor has a set of instructions it can execute. This is called the *instruction set*. These are instruction like move data, add and subtract, and compare values. A processor might have around 1 or 2 hundred instructions.

Some processors

Famous for their roles in computing history:

4004 - Intel. 1971. The first 'mircoprocessor'. 2250 transistors. Clock speed 750 kHz

8080 - Intel, 1974. 8 bits. 6000 transistors. 3 MHz

Z80 - Zilog 1976. Used in place of the 8080, in CP/M operating system.

8086 - Intel 1978. 16 bit. 10 MHz

AMD 1982 pin-compatible with the 8086, used in the IBM PC desktop

80186 Intel 16 bit 1982 25 MHz 55,000 transistors.

That began a sequence of Intel chips - the 286, 386, 486 and Pentium (586), with in 2021 the x86 architecture.

AMD have continued to produce their versions.

Motorola produced the 6800 and the 68000, used in Apples and Macs.

Qualcomm make the Snapdragon processors used in mobile devices.

Different processors have different instruction sets.

They are not compatible. Machine code for one processor will not run on a different processor.

Machine code and assembler

Machine code is how program instructions are represented in binary. So for example

0100 1000 0000 0001 1100 0011

is the instruction to add the number in register RBX into register RAX

Programming in actual machine is impractical and pointless - if you tried to remember hundreds of patterns of 64 bits, there would be countless errors.

An example of an *assembly language* instruction is

```
add %rbx, %rax
```

This means to add register RBX into RAX - same as above, but it makes much more sense.

Assembly language instructions are one-to-one with machine code

In other words, each assembly language instruction corresponds with just one machine code instruction.

An *assembler* is a type of compiler, which translates assembler code into machine code. Because they are one-to-one, this is pretty simple (compared to most compilers).

We write here assembler instructions, but they correspond to machine code instructions.

The details of assembly language instructions depends on the assembler used. That is, what % and $ and # mean depend on which assembler it is. This is different from Java or C or Javascript, which all have agreed standards. 'Assembler' is not a language. It is a type of language. The details depend on which assembler it is, as well as which processor it is for.

Some assemblers for x64 processors are

- GAS, the GNU project assembler,
- MASM from Microsoft, Windows only
- NASM, a free open-source widely used assembler

Opcodes and operands

Most machine code instructions contain *op-codes* (operation codes) and *operands*. The op-code is what to do. The operand is what to do it with.

For example

```
  add $5, %rax    # add 5 into rax
```

has op-code add, and operands 5 and the rax register

A few instructions have no operands. For example

```
  ret
```

which means return from a sub-routine.

Common instruction types

load, move, store - transfers data from one place to another . This might be to or from memory, and to or from registers.

arithmetic - add, subtract, multiply, divide. With early processors this was integer arithmetic only. Modern processors usually have floating point arithmetic instructions as well.

compare - this sets flags, comparing two operands as if they had been subtracted. For example the zero flag would be set if the operands are equal.

branching or jumps - the next instruction is not the next one. For example

jmp newLoc

would jump to newLoc, which would be a label, corresponding to some address. A conditional branch or jump is combined with a compare instruction. For example, this high level language code:

```
if (x>5)
     y=2;
else
     y=3;
```

in assembler would be :

```
      mov x, %bx    # copy x into bx register
      cmp 5, %bx    # compare 5 and x
      jg loc1       # jump on greater -
                    # if 5>x goto loc1
      mov 3, %ax    # put 3 in ax
      jmp loc2      # jump over
loc1: mov 2, %ax    # put 2 in ax
      jmp next
loc2: mov %ax, y    # store in y
next:   ..
```

logical bitwise operators AND, OR, NOT,

XOR. For example if register ax contains 1100 0101, then

```
      and %ax, 1011 1100b
```

woud do this:

1	1	0	0	0	1	0	1	ax before
1	0	1	1	1	1	0	0	operand
1	0	0	0	0	1	0	0	ax after

Bitwise means the AND is applied to every pair of corresponding bits

Shifts and rotates

Processors typically have shift and rotate instructions. Shifts move bits along a register, left or right. Bits that move out the end go into the carry flag (in the flags register).
For example, initially

CF		Register							
?		1	0	1	1	0	0	1	1

after a shift *left*

CF		Register							
1		0	1	1	0	0	1	1	0

and again:

CF		Register							
0		1	1	0	0	1	1	0	0

This has the same effect as multiplying the contents by 2 - but it is faster than a multiplication
A rotation is similar, but the bit that goes out one end goes back to the other. For example a rotate right:
Initially:

Register							
1	0	1	1	0	0	1	1

after a rotate right:

Register							
1	1	0	1	1	0	0	1

Addressing mode

This means how the operands are specified. One method is *immediate addressing*. In this, the operand is actually in the instruction. So for example

```
mov ax, #23
```

simply means move the number 23 into the ax register.

Direct addressing means we give the address of the operand. So for example

```
mov ax, $1234
```

means copy the value from address 1234 to the ax register. More usually we would use a symbolic address, so say

```
mov ax, $num1
```

where num1 would be a symbol, whose value is some address in memory. In the same program we would say

```
num1: db 48
```

so num1: labels a memory location. db means define bytes. This would store a number (48) in this location, and we can refer to it as num1. We do not need to know the actual address.

The assembler would have conventions for how the addressing mode was specified - like # for immediate addressing and $ for direct addressing.

Instruction formats

Some instructions are 1 byte long. Others might be 2 bytes or 3 or longer.

The instructions are formatted, with certain bits used for certain meanings. For example a 2 byte instruction might be

1	1	1	1	0	1	0	1	1	1	0	0	1	0	1	1
opcode						addressing mode		operand							

Two bits are used for the addressing mode, giving a choice of 4 modes. 00 might be immediate, 01 direct, 10 indirect and 11 register.

Six bits are used for the opcode, allowing for 26 - 64 different instructions. And 8 bits for the operand.

This is just an example. Different processors use different instrcution formats.

Test

1. What is assembler?

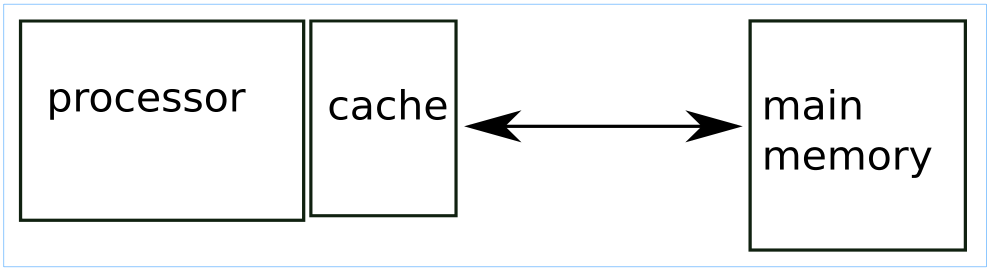
2. What is immediate addressing?

3. What is an op-code?

Optimisations

There are a set of ideas relating to improving processor performance.

Cache memory



This is a small amount of fast memory which is between the processor and main memory.

Code is often like this:

x=3;

x=x+5;

if (x>7) ..

high level langauge code	compiled into..
x=3	1. load 3 into register 2. *store* into address x
x=x+5	3. *load* address x into register 4. add 3 to register 5. *store* into address x
if (x>7)..	6. *load* address x into register 7. compare 7 8. ..

This involves reading and writing a small number of memory locations several times. Memory is slow compared with typical register processing, so this slows down the processor. Using a cache improves this:

At step 2, the x value is written into the cache.

At step 3, it is read back from the cache (fast) not from memory (slow).

At step 5 the value is stored into the cache

At step 6, re-loaded from the cache

We only need to read from memory when a new value is required which is not already in the cache.

We only write to memory when we must. This is when the cache is full, or if the program ends and the operating system takes the process out of memory.

Real processors currently often have 3 levels of cache memory, up to 1 MB, on the same chip as the processor.

Pipelining

Standard execution involves the 4 stages:

1. Fetch the next instruction
2. Decode it
3. Execute it
4. Increment the program counter

but because different parts of the processor are used by these, they can be overlapped:

fetch instruction 1	fetch instruction 2	fetch instruction 3	fetch instruction 4		
	decode 1	decode 2	decode 3	decode 4	
		execute 1	execute 2	execute 3	execute 4
			inc PC	inc PC	inc PC
time >>					

Multiple cores

This means having several processors in the same package, and probably on the same chip.

These can be used in *concurrent* or *multi-threaded* programming. As an example, a program might be a chat server, sending and receiving messages between several sets of users at the same time. This would probably be done by running each conversation in its own *thread*. One thread might run on one processor code.

Another use might be for operating system processes. At any one time an operating system is running many processes - applications programs such as web browsers and spreadsheets, GUIs, schedulers, IO processes and so on. These might be run on their own cores.

Clock speed

Obviously a higher clock speed means a faster operation. But the electronics used in hardware components has a limited speed, so a clock speed higher than memory speed, for example, is pointless.

Word length and data bus

A *word* is a general term meaning how much data including instructions) a processor moves at a time. The first micro-processor chips were 4-bit, and their word size was 4 bits. Many, like the Z80 and the 6502, were 8 bit word size. Many current chips are 64-bit.

The data bus width means how many bits the data bus can carry at one time.

Different situations are:

- Word size and data bus width are the same - say, 64 bit. This is simple - a memory read or write of 64 bits happens in one operation.
- Word size greater than data bus width. Then a read or write must be done in several steps, and will be slower.
- Word size less than data bus width. Then we either waste some bus width, or fetch several words at once

Instruction length

Native code instructions are also fetched over the data bus. An op-code with n bits can code for 2^n instructions. So if we want more instructions, we must have longer instructions.

Two design options are CISC and RISC.

RISC means Reduced Instruction Set Computer. Here we have a small number of simple, small instructions. The idea is that several instructions are fetched in 1 read.

CISC means Complex Instruction Set Computer. The instructions are larger. Only one is fetched in one read. But being more complex, one instruction can do more that one RISC instrcution.

Address bus width

The width of the address bus limits how much many memory can be used. If we had a 4 bit address bus, addresses could range from 0000 to 1111 - just 16 distinct addresses.

Typical current address bus widths are 48 bits, meaning 2^{48} = about 2.8 X 10^{14} bytes. This is 100 000 GB - far larger than actual installed main memory

Test

1. What is processor cache memory?

2. What are the advantages and disadvantages of having a wider data bus?

3. Explain how pipelining is used to increase processing speed.