COMP12111 notes

Todd Davies

January 1, 2014

Note, extra space has been allocated for the right hand margin to allow for more extensive margin notes. Also, it gives you space to make your own annotations.

Contents

Introduction

Unlike many of the courses, the university supplied notes for this course are of a very high quality. This is especially true of the notes covering the first half of the course (weeks one through six). In light of this, I've decided not to write notes on the first half, but concentrate solely on the second half of the course. However, it is likely that I will produce other resources such as summary notes or flashcards for the whole of the course.

1 The three box model

The three box model describes the classic model of a computer. The three boxes consist of the CPU, the memory and I/O.

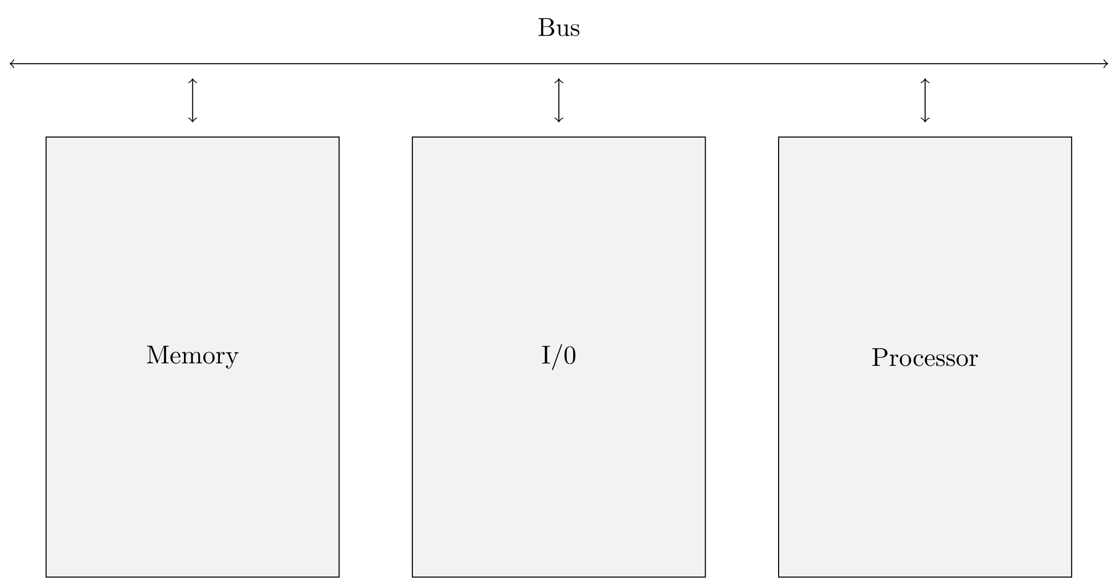

Figure 1: An example of the three box model

1.1 The Amdahl/Case rule

A computer that has one disproportionately powerful component is very wasteful since the other components will act as a limiting factor with regard to the speed of the computer. It's no good having a fantastically fast processor with a tiny amount of RAM.

The Amdahl/Case rule gives us guidelines that we can use to determine sensible specifications for components within a computer. Though there are many different versions of this rule, it is something along the lines of:

MIPS stands for million instructions per second

> A balanced computer system needs about one megabyte of main memory and about one megabit per second of I/O per MIPS of CPU performance.

1.2 CPU

1.2.1 The fetch, decode, execute cycle

The CPU is essentially a large FSM that loops over three operations; fetch, decode and execute in order to perform the instructions defined in a program.

FSM = Finite State Machine

Fetch

The processor first reads a word from an address that is pointed to in memory by a *pointer*. After the instruction has been read, the pointer is moved on to the next address in memory.

1 word = 4 bytes

Decode

The instructions that are to be loaded from memory are really just a really long list of numbers. However, each number is coded in such a way that it represents an action. A rudimentary way of encoding such a system would be to let the number 1 mean 'shift bits left' and the number 2 mean 'shift bits right'. An operation code is obtained from the instruction and the control signals specific to that code are then set for the FSM.

Execute

In this phase, the data is moved through the datapath and the instruction that was in memory is now performed. The processor then starts the cycle again, by fetching an instruction from memory.

A datapath is a collection of logic units to perform arithmetic or other functions. See course supplied notes for more information on datapaths.

1.2.2 Maintaining state

A CPU is required to maintain some form of state while processing instructions, since most instructions have interactions between one another. In order to keep track of what's going on in between instructions, the CPU uses both the registers that is has built in and the main memory.

It is important to realise that the whole system changes state at the same time, driven by a central *clock*. This means that all the parts of the system are in sync with each other. In fact, we can treat the system as a whole (including the memory and registers) as a finite state machine.

1.2.3 Address spaces

An address space is a number of memory locations that a system can address. Each location in memory has a unique address, which is a number.

Memory addresses are countable, i.e. you can increment one to get the next one and decrement one to get the previous one. However, they are sometimes not countable in the traditional sense. First of all, they are usually counted in hexadecimal in order to save characters and enable easy conversion to binary, with each digit converting to four bits. Second of all, the length of the word defines the gap between each countable memory location.

N.b. Most processors offer the capability to address bytes in between the words too

For example, in 32 bit processors, words are defined as 32 bits long. Henceforth, each memory location is contains 32 bits, and so the addresses go up in fours. In a 64 bit processor, the gap between adjacent addressable words in memory would be eight addresses.

The number of bits in a word is very important for a number of reasons. Longer words usually mean longer instructions, so more information can fit inside, meaning less instructions need to be executed to perform tasks. Also, longer words means more addressable memory locations; in a 32 bit system, there are 2^{32} memory locations, but in a 64 bit system, there are 2^{64} addressable memory locations. This is why 32 bit systems are limited to 4GB or RAM.

1.3 Memory

Memory allows the processor to write store and load data. It is often referred to as Random Access Memory. As opposed to hard drives, where in order to access different locations a physical component must be moved, RAM is able to access any location in any order with no time penalty, hence the usage of the term *random*.

We can work out how many bits are required to address a memory of a given size. In order to do this, we must find the power of 2 equal to or above the size of the memory (in bytes), and split it into common factors (which should also be powers of two) then we add up the powers, which will give us our number of bits. Here are some examples:

Find the number of bits required to address 1 Kbyte of memory

1. Find the powers of two that will go into 1 Kbyte: 1 Kybte $= 2^{10}$ bytes

Find the number of bits required to address 64M bytes of memory

1. Find the powers of two that will go into 64 Mbytes: 64 Mybtes $= 2^{26}$ bytes $= 2^6 + 2^{20}$ bytes

2. Add the powers together: $6 + 20 = 26$ bits

Find the amount of memory that can be addressed by 19 bits

1. $2^{19} = 2^{10} \times 2^9 = 1,024 \times 512 = 512$ Kbytes

1.3.1 Memory caching

A commonly used optimisation for memory is to use a cache. This is a small amount of extra memory that is very fast to access. The values stored in memory addresses that are being accessed frequently can be temporarily stored here instead to avoid the comparatively slow referencing of the main memory.

1.4 Input/Output

IO is concerned with interfacing with peripheral devices such as keyboards, monitors, networks etc. Each device will have an interface to a specific bus that can communicate with memory and the CPU.

A *port* is a form of I/O that is usually mapped to an area of memory. In the eyes of the CPU, a simple output port is just an area of memory to be read from and written to, however, it will also be mapped to some external connection such as lights, motor or even more complicated devices such as a printer.

An input port will also 'look' like an area of memory to the CPU, however, that area of memory will be connected to external signals.

Most ports are 8-bits wide, even in processors that use larger word lengths.

It is also possible to have bidirectional ports, however, this requires extra coordination to ensure that reading and writing doesn't take place at the same time.

Types of ports

There are two main types of ports, serial and parallel. Parallel ports are as described above; just a collection of wires that can be in the states 1 or 0. In order to send a 1 Mbyte file over a parallel port, could either have eight million wires or you can use only eight wires and splitting the file into one million parts.

Serial ports only deal with single bits, and so require one wire. This may seem very slow, but a lot of time is often spent optimising the transfer so its speeds are comparable with parallel ports. However, serial ports often need extra registers to signal other information such as transfer speed and the direction of transfer.

1.5 Buses

A bus is a collection of signals that act together.

There are three buses used by the CPU:

- **Address bus** This is an output for the processor, and is used to specify the location in Memory or I/O for data to be transferred. It is usually as wide as the word length of the processor.
- **Data bus** This is usually a bidirectional bus, usually as wide as a processor's registers (that in turn are usually as wide as a word). However, a smaller data bus will reduce the cost of the processor, but a larger bus will enable a higher bandwidth, which could let the processor fetch more than one instruction in one cycle!
- **Control bus** The main function of the control bus is to specify the direction of the flow of data. However, it also has a lot of other functions which aren't relevant here.

N.b. Another way to make a bus go faster is to increase the clock speed it is running at.

2 Processor design - the MU0

The MU0 is a very simple design of processor. So simple in fact, that it can be described within the scope of these notes. However, despite it's simplicity, it is a complete processor and is capable of running complete programs.

The MU0 is a 16 bit machine, that has a 12-bit address space. Instructions are coded like so for the MU0:

Since there are 4 bits for the operation inside an MU0 instruction, we have a capacity for 2^4 (16) different operations. Since the address space is comprised of 12 bits, the maximum amount of addressable memory locations is 2^{12}, which is equal to 4096 16 bit words, making for a total of 8 Kbytes of RAM.

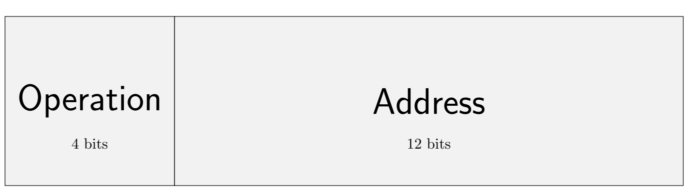

Figure 2: A sample MU0 instruction

2.1 The MU0 instruction set

Since the instructions are 16 bits wide, the memory and internal data paths inside the MU0 are also 16 bits wide. There are two registers available to programmers (though more will be used internally).

Each instruction can have either one or zero operands. Instructions that use more than one operand must implicitly use a register. Here is a table describing the instruction set:

You might notice that there are only eight operations here even though we have space for sixteen. The unmapped op codes are blank at the moment, but could be used to expand the capabilities of the processor.

Op Code	**Mnemonic**	**Description**
0	LDA [*op*]	$[op] \rightarrow Acc$
1	STO [*op*]	$Acc \rightarrow [op]$
2	ADD [*op*]	$Acc = Acc + [op]$
3	SUB [*op*]	$Acc = Acc - [op]$
4	JMP [*op*]	$PC = S$
5	JGE [*op*]	If $Acc >= 0$ then $PC = S$
6	JNE [*op*]	If $Acc \neq 0$ then $PC = S$
7	STP	Stop

2.2 Executing instructions

Like any CPU, the MU0 goes through the fetch execute cycle for every instruction it executes. However, since the MU0 is so simple, we can break the cycle into more steps:

1. Fetch the instruction from the memory address specified by the PC.
2. Increment the PC
3. Decode the instruction (i.e. read the first four bits)
4. Get the operand for the instruction from:
   - Memory for load or arithmetic instructions
   - The instruction register for jump instructions. We look inside the instruction register since it is where the instruction is held while being decoded. The address of the operand will be encoded in the instruction itself.
   - The accumulator register for store or arithmetic instructions.
5. Perform the operation
6. Write the result to the PC, accumulator or memory.

2.2.1 The program counter

The PC (Program Counter) is a register that contains the memory address of the next instruction to be executed. Every time an instruction is executed, the program counter is incremented by one.

2.3 The MU0 datapath

Instructions can be executed in two clock cycles of the MU0. The first cycle is used to fetch the instruction into the instruction register, and the second is used to decode the instruction, read the operand and store the result wherever it needs to be stored. Please refer to the course supplied notes for the full datapath design of the MU0.

2.4 Registers in the MU0

The MU0 contains three registers; ACC (Accumulator), PC (Program Counter) and IR (Instruction Register). Only the first two are visible to the programmer (since the job of the instruction register is to store the next instruction to be executed after a fetch has occurred).

Each register is comprised of 16 D-type flip flops and is therefore made up of sixteen bits. All of the flip flops are connected to the system clock, and are therefore synchronous in their operation.

The CE (Clock Enable) signal initiates the loading of information into the registers. If CE is high, then each flip flop will assume the input value that it is given when the system is clocked.

N.b. if there isn't a clock pulse when the CE is high, then nothing will happen, since the registers only change on each clock pulse.

The outputs of the three registers can be def into a shared bus. Only one register can feed into the bus at a time, and in order to coordinate this, the OE (Output Enable) signal must be high for a register to drive the output bus.

2.5 Register banks

In modern processors, there are usually far more registers than in the MU0, for example, ARM uses 16 registers and MIPS uses 32. Each register can be connected to any port at any time. It is common to be able to perform several read/write operations on registers in one instruction.

Like memory, registers need to be addressed using a series of bits. With 16 registers, 2^4 bits are needed to address all the registers. In the MU0, we don't need to bother with this, since it has only one programmer accessible register, the accumulator, and $2^0 = 1$.

2.6 The ALU

The MU0 doesn't really contain an ALU (Arithmetic Logic Unit), since the ALU contained within the MU0 doesn't have functions for logical operations such as `NOT`, `AND` and `XOR`. Thus it would be more accurate to simply call the ALU inside the MU0 an Arithmetic Unit!

A typical ALU will usually contain two input buses (lets say called `A` and `B`) and an output bus (called `Z` in our example). The ALU in the MU0 is capable of doing the following operations:

Instruction	**Description**
`ADD`	`Z = A + B`
`SUB`	`Z = A - B`
Fetch instruction	`PC = PC + 1`
`LDA`	`Z = A`

It is important to note that each of these operations can be expressed as an addition:

- Z = X + Y
- Z = X + (-Y)

- Z = X + 1
- Z = 0 + Y

This makes the implementation of the ALU easier.

Because all the MU0 ALU really does is add stuff together, its main component is an adder. Since the MU0 is a 16 bit machine, the adder must be 16 bits too. The most simple adder we can use for the MU0 is a 16 bit ripple-carry adder.

2.6.1 Critical paths

We can determine how fast a circuit can run by finding the critical path of the circuit. The critical path is defined as the path some logic can take from one end of the circuit to the other that will take the most time. If we are given the time it takes for each gate used in the circuit to change state, then we can simply add up all the times in each path, and see which one is longest.

When we have the time it takes to execute the critical path, we can use the formula $f = \frac{1}{T}$ to find the maximum clock frequency of the circuit

2.6.2 The structure of the ALU in the MU0

Though the ALU inside the MU0 is mostly comprised of just an Adder, there are a few more things going on too. There is some preconditioning that is applied to the input buses. For the following commands, the following preconditions are applied:

Function	First bus	Second bus
`ADD`	Normal	Normal
`SUB`	Normal	Inverted
`INC`	Normal	1
`output = second_bus`	0	Normal

'1' here means '0000000000000001' and '0' means '0000000000000000'

Selecting what logic to use in an ALU

An important feature of any ALU is the ability to select what Logic to employ on the input data. This is often accomplished by way of another bus going into the ALU carrying the code of the logic function to execute.

It is possible to use a multiplexer to execute certain logical tasks. Setting all bits to 0 in a bus for example could be done in verilog like so:

```
module setzero(fun, in, out);
        output [15:0] out;
        input [15:0] in;
        input fun;

        assign out = fun ? 0 : in
endmodule
```

The last line of the verilog block does the actual work here. It is a conditional operator that will set the output to 0 if the `fun` input is high, and set the output to the `in` bus if `fun` is low.

2.6.3 General ALU's

In general, ALU's have many more features than the one in the MU0. Such features may include:

- Do nothing to the data (also known as *true data*)
- Complement the input (invert all the bits)
- Zero the bits
- Make all the bits 1

2.6.4 Decoding the function code in an ALU

The required value of the control bits to activate any given function in the ALU are arbitrary, though choosing a good set of values is desirable since it may make the implementation easier.

2.7 Making decisions in the ALU

In any processor, when an `if` statement is executed then the processor will most likely (under many layers of syntax and abstraction) perform a branch.

In the MU0, the accumulator is used to evaluate conditions, however, in some other architectures, the result of a comparison is stored in a separate *condition code* register (these results are often referred to as *flags*).

In the MU0, there are only two conditional branches:

1. Jump if `Acc` is positive
2. Jump if `Acc` is not 0

This also allows programmers to test for a specific value, since you can use `SUB` to get 0 if two values are equal and then use the second jump condition.

2.8 How the MU0 executes operations

All instructions execute in the MU0 in two instruction cycles. The first cycle involves the instruction being fetched from memory and read into the instruction register, while the second cycle is when the instruction is decoded and executed.

2.8.1 Fetching an instruction

When the processor is in the fetch state, the value at the memory address specified by the program counter register is fetched into the instruction register. The PC is then incremented so that the next instruction is fetched when this one has finished executing.

Even if the instruction is a jump, the PC is still incremented after the fetch. This is because the instruction hasn't been decoded while it is being fetched so the processor has no way of knowing if it's a jump.

2.8.2 Executing an instruction

When the processor starts to decode the instruction, its behaviour is defined by the instruction that it is executing. It will first read the instruction code and then follow one of the eight possible paths of execution depending on what the instruction is to be executed.

The control signals that control all of the registers and the ALU depend on the value of F. The data in the instruction is then just piped through the processor, being mutated by the various components as it goes.

2.8.3 Implementing an instruction decoder in verilog

We can create a block of verilog code that will compile down to a FSM that will translate a three bit input from the instruction into the control lines to the various components inside the processor.

First, we need an always block to trigger whenever the `state` or the instruction register `IR` changes:

```
always @ (state, ir)
```

After that, we need to check the state of `state` to see if the processor is loading an instruction or decoding one:

```
        if(state==0)
                // We're fetching an instruction
                begin
                // Set the control lines to fetch
                // an instruction from memory
                Asel = 0;
                ...
                Ren = 1;
                Wen = 0;
                end
        else    // The state must be equal to 1
                        // therefore we're decoding
                begin
```

Now we need a case statement to decode the instruction code:

```
        case(ir[15:12])
                // LDA
                0:      begin
                        Ren = 1;
                        Wen = 0;
                        Asel = 0;
                        Xsel = 1;
                        M[1:0] = 2'b10;
                        ...
                        end
                1:      begin
                        ...
                        end
                ...
```

```
                    7:      begin
                            ...
                            end
            endcase
    end
```

2.9 Timing in the MU0

The MU0 employs a synchronous timing design. This means that all state changes happen at the same time, at the positive edge of the clock. This is handy, since when the control signals begin to be calculated at the start of decoding, the IR is latched and they have a whole cycle to settle.

In order to employ a synchronous design, we must assume that the clock signal arrives at all the flip flops at the same time, and that there is enough time between clock ticks so that the slowest possible set of logic changes have time to take place.

This would be the *worst case critical path.*

In order to calculate the fastest timing we can do, we must find the critical path of the processor (i.e. the slowest path data can take). We the have to use the formula:

$$Time\ period = \frac{1}{frequency}$$

to find the maximum clock frequency. So, if the critical path was $20ns$, then the fastest clock frequency we could set would be:

$$\frac{1}{20 \times 10^{-9}} = 50\,\mathrm{MHz}$$

3 Optimising an architecture

There are three main ways that we can optimise a processor so that it can run faster, they are:

- Improving the technology so that there are more transistors on the chip. This means that it can do more logic at any one time.
- Shrink the critical path so that the clock speed can be increased. This often has to balance power consumption and heat production with speed.
- Restructure the design so that each clock cycle can do more stuff. An example of this may be when an instruction is both fetched; the PC must be read and the value must be fetched from the memory. These operations could be done one after another, but it would mean that the fetch stage of an instruction would take twice as long, and the processor would take 1.5 times longer to execute.

A metric often used here is the number of clocks per instruction (CPI).

3.1 Carry look ahead

Adders are an important and oft used component in any computer architecture. Since there are so many of them, they are a good target for optimisation. It is possible to work out whether the adder will generate a carry or not almost as soon as the input bits arrive. Just by looking at the input bits, we can tell if `Cout` will be Zero, `Cin` or One.

A	B	Cout
0	0	0
0	1	`Cin`
1	0	`Cin`
1	1	1

Working out the carry out and propagating it onto the next adder is a technique called **Look Ahead Carry**.

3.2 Parallelism

One obvious way to improve the speed of any architecture is to make two things happen at once; i.e. make things happen in parallel. There are two different ways of achieving parallel execution in a computer:

- Start one instruction before another has finished executing (e.g. by decoding one instruction while fetching the next)
- Compute several instructions at the same time (e.g. by having two processing cores)

Using more than one processing core, or executing multiple instructions in parallel is also known as superscalar processing.

However, it is often the case that despite using parallel execution, gaining a speed increase of twice what was observed without using parallel execution is often not the case, since systems would be more complicated. However, the cost of the system would usually increase by at least two times, since it would be harder to make and would contain more components.

3.3 Adding bitwise operators to an ALU

If you wanted to add bitwise operators to an ALU, all you need to do is put the respective gates before the inputs into the ALU. The specific bitwise operator to be used can be selected by using a multiplexer.

An optimisation here, would be to recognise that it is possible to implement an XOR gate by just disabling the carry on the adders inside the ALU.

4 Bit shifting

In order to multiply or divide a number by two in binary, all you have to do is shift the numbers to the left or right by one place. This is called a bit shift.

You may notice, that when you shift to the right, data is lost. This problem can be mitigated by doing two things:

- A logical shift right (where zeros are shifted in to the MSB).
- An arithmetic shift right (where the existing MSB is copied to the new MSB). This will preserve the sign of the number if it is being interpreted as a signed integer.

However, since in most (if not all) computer systems, a number only has a finite set of binary digits, shifting to the left can also result in the loss of data. When doing a left shift, a zero is always shifted in to the right side.

4.1 Bit rotations

It is also possible to rotate bits, so when you do a shift, copy the bit that would be shifted out, and insert it onto the side of the number that is gaining a new bit. In this way, data is only never mutated, and never lost.

4.2 Implementing a single shift

The only components needed to implement a single place bit shift are multiplexers. See page 47 in the notes for a diagram.

4.3 Shift registers

A shift register is just like a normal register, but it inputs and outputs the data in parallel. You can also give them bits one at a time, so that all the bits in the register are shifted along to make room for the new one and the end bit 'falls off' and is lost.

Often, lots of extra components are required to implement a shift register, so it's not usually worth implementing inside a processor. However, they are often used for tasks involving I/O.

5 Multiplication

Simple multiplication would involve starting with a register set to `0`, and then adding `M` to the register `N` times. This however, is very slow for large numbers. This would be simple to implement though, since all you really need is an adder and some logic to count from `0` to `N`.

The time complexity would be $O(nm)$.

A faster way to multiply numbers would be using long division:

1. Start with zero in an accumulator
2. Get the least significant digit of `M`. We'll call this `X`.
3. Multiply `N` by `X`.
4. Add the result to the accumulator.
5. Multiply `N` by 10.
6. Make `X` the next least significant digit of `M`.
7. Repeat steps 3-6 until you run out of digits in N

Here's an example of applying the algorithm to the multiplication 453×643:

`Accumulator`	`X`	`M`
00000000	3	00000643
00001929	3	00000643
00001929	5	00006430
00034079	5	00006430
00034079	4	00064300
00291279	4	00064300

Here, we're using base 10 to do our calculations, but a computer would use base 2. This is no problem, since we'd only ever have to multiply by 1 or 0 (which is easy) or multiply by 2 (instead of 10, which is just a shift).

Here's an example in base 2. Multiply 0101 by 0010:

Accumulator	X	M
0000	1	00010
0010	1	00010
0010	0	00100
0010	0	00100
0010	1	01000
1010	1	01000
1010	0	10000
1010	0	10000

We can check that using base 10: $5 \times 2 = 10$.

It is possible to terminate this algorithm early if the remaining bits to be multiplied are all zeros. This is called **early termination**.

If the multiplication goes over the number of bits available inside the number, we can use **modulo arithmetic**. This is when we will take the answer and find the modulus of it with 2^n where n is the number of bits in the number.

If we use modulo arithmetic, then multiplication will work just fine on signed integers too!

6 Computer memory

Obviously, computers are able to temporarily store values inside registers, however, there are never enough registers in the processor to allow the execution of more than just the most basic of programs. This gives need for memory, which is a set of locations where data can be stored situated outside of the processor.

The program that is being executed on the computer is also situated inside the memory. As we saw on page 2, the CPU gets each instruction it's going to execute in a sequential manner from memory.

It is important to remember that memory treats programs and data the same. A single memory location could be interpreted as a number, character, pointer or CPU instruction, but all it is to the memory is a set of binary digits to be stored at a particular address. This is a vital part of the *Von Neumann* architecture.

There are other types of architecture that attempt to approach the problem of storing programs in different ways. For example; the Harvard architecture is where programs stored in memory are separated from the data they store.

6.1 Random Access Memory

Most computers will implement memory as Random Access Memory (RAM). This is where any location in the memory can be loaded from or stored to at any time.

The MU0 has a 12 bit address bus for it's random access memory, and therefore is able to address 2^12 different memory locations of one word each. This is equivalent to $4,000$ words or $8kB$.

Each bit of memory in RAM is usually implemented as a flip flop.

6.1.1 Tristate devices

A tristate device has three states; 1, 0 and off.

In	En	Out
0	0	0
0	1	0
1	0	0
1	1	1

6.1.2 Decoding an address

When the CPU wants to do something with an address in memory, the memory controller must convert the address supplied by the CPU into the correct selection of flip flops. The address is supplied as a binary number, so all we have to do is have a multiplexer with N address bits (where N is the width of the address), and then we can select 2^N addresses. The address produced by the multiplexer selects which output from memory should become active.

Often, it's impractical to decode an entire address bus in a single decoder, so they are nested, meaning that the first decoder might choose what memory device is active (from say two sticks of RAM) and then the next may decide what specific module on the RAM the address is on etc etc.

In verilog, it might look like this:

```
module address_dec(addr_in, enable, sel_out);
	input	[2:0]	addr_in;
	input			enable;
	output	[7:0]	sel_out;
	reg		[7:0]	sel_out;

	always @ (addr_in or enable)
		if(enable)
			case (addr_in)
			0: sel_out = 8'b00000001;
			1: sel_out = 8'b00000010;
			2: sel_out = 8'b00000100;
			3: sel_out = 8'b00001000;
			4: sel_out = 8'b00010000;
			5: sel_out = 8'b00100000;
			6: sel_out = 8'b01000000;
			7: sel_out = 8'b10000000;
			endcase
		else
			sel_out = 0;
endmodule
```

6.1.3 Implementing memory

Using a D-type flip flop is often to expensive for the millions of addresses needed in RAM. Consequently, the most desirable trait for RAM is to cram as many bits as possible into the smallest amount of space at low cost.

Static RAM (SRAM) is often used in RAM since it is small compared to D-type flip flops and other types of memory. Though SRAM (and DRAM) needs amplification before it can be used in a circuit, the amplifiers can be shared between many thousands of bits of SRAM, so the overhead is small. Having small amounts of space used up by RAM means that the cost is lower since less silicon is used.

Though DRAM takes up less space than SRAM, it is slower and requires more support logic. This means that for most applications, even though it takes up more space, SRAM is more suited.

The timing of the changes of the input signals to memory is very important. If the input address changes during a write operation for example, then two memory addresses could become corrupted! Also, since most memory has no clock input, it cannot rely on the processor's timing to ensure that things happen in sync.

A transparent latch is used to ensure that when writing starts, the write signal and address signals don't change for the duration of the write.

It is impractical to read and write individual bytes of memory when the address width of the memory is greater than one byte. Reading is slightly easier, since the whole word can be read, and then shifted so that the desired byte is in the first eight bits and the other bits are ignored. Writing is harder since you have to write a whole word, which requires you that the bits in the word that you *don't* want to set are the same as the ones currently in memory, otherwise they'll be overwritten.

6.2 Memory maps

A memory map is a diagram showing what different ranges of memory addresses map to. They usually start from the first address and finish at the last. Some areas of the memory map may not contain anything (if there aren't enough memory locations to use up all the bits in the bus). Reading from these undefined areas could give any value, but writing to them will have no effect.

Some architectures dedicate a separate area in memory just to IO. Since IO is often slower than normal memory, it might be possible to slow down bus cycles here automatically.

6.3 Endianness

Endianness is a property of a memory location that defines the order of the bits. There are two types of endianness, **little endian** and **big endian**.

In the word `0x12345678` there are four bytes:

- `0x12`
- `0x34`
- `0x56`

- 0x78

In little endian, the first byte would be 0x12 since bits are read from left to right in little endian.

In big endian, the first byte would by 0x78 since bits are read from right to left in big endian.

This is important when we deciding what the most and least significant bits in a word are. For example, in this instance the *lsb* is 0x12 in little endian, but 0x78 in big endian.

N.b. The least significant bit is the smallest address.

6.4 The memory hierarchy

Compared to the speed of a processor, memory is slow. This is a problem, since if fetching an instruction takes a long time, then the previous instruction will have already finished executing and the processor will be sitting idle.

A possible solution to this would be to increase the speed of the memory, but providing gigabytes of fast SRAM would cost a great deal of money. The solution used by most architectures is to have multiple layers of caching, where small, yet frequently accessed areas of memory can be replicated using very fast SRAM.

This allows us to create some sort of hierarchy of memory:

1. Registers inside the processor.
2. Small amounts of SRAM (static RAM) inside the processor (called the L1 cache). Typically around 1-5% of the size of the main memory.
3. The main memory. Usually made of DRAM (dynamic RAM).
4. Read Only Memory or Electrically Erasable Programmable Read Only Memory to store the information needed to boot.
5. The paging file (virtual memory) on the hard disk.

As we descend the memory hierarchy, the speed of the memory goes down, but so does the cost per bit. We may be able to have terabytes of virtual memory, but only a couple of kilobytes (probably less) of internal registers.

Having a cache is especially useful since most programs adhere to the *principle of locality*, which describes how areas of memory are likely to be used multiple times for a specific duration. There are many types of locality, as described below:

- **Temporal locality** - when a memory location is used, it is likely to be used again in the near future. This may be especially useful when the application is maintaining a stack, or is in a loop.
- **Spatial locality** - if a memory location is used, then locations around that memory location may also be used soon. This is especially true with arrays and other ordered data structures.

If the CPU uses a Harvard architecture rather than a Von Neumann architecture, then it may have two caches; one for the data memory and one for the instruction memory. Note that a Harvard architecture will require a higher memory bandwidth, which is more expensive, but faster.

Disadvantages of the Harvard architecture are that code can't be self modifying (since storing instructions won't be implemented for the instruction memory), more wiring it needed and the split between the amounts of data and instruction memory are hardwired, so a small program that needs a lot of data would be at a disadvantage for instance.

6.4.1 ROM

ROM is non-volatile, flash based memory. It cannot be written to (unless it's electrically erasable) and so is protected from accidental writing. It is very useful for finite state machines and bootstrap programs that start up a computer.

6.4.2 DRAM

Dynamic RAM stores bits as charged uncharged capacitors. This means that it is very dense, yet very cheap since capacitors can be made to be very small. However, capacitors (especially very small ones) tend to leak charge, and so the memory is volatile and needs refreshing every so often. This periodic refreshing is what gives rise to the name *Dynamic* RAM.

7 Input Output and Communications

There's no point having a blazingly fast CPU and memory unless you have methods of interfacing with it so that you can both give it data and receive it's output. This is the principle behind Input Output (I/O).

The maximum speed that information can travel at is equal to the speed at which the medium it is being carried by is able to propagate. This can be a limiting factor in very fast IO communications.

Digital signals have some characteristics that are important to remember:

- There are two widely separated, obvious levels of the signal with no intermediate state.
- This means that if noise is overlaid onto the signal, it is still discernible with little effort.

Different mediums are able to transmit signals at different speeds. This can be essential for achieving very fast speeds. Ultimately, it's a trade off between practicality and speed. Connecting a wire between two computers is easy and reading the voltage or current requires little effort, however, using a speaker to project audio requires next to no effort to set up, but requires lots of effort to decode and sound travels slowly through air. Light is a very fast medium for communication since it's the fastest thing in the universe, which is why optical fibre is very fast.

If bits are sent one after the other, then the communication is described as **serial**, but if more than one bit is sent at once (say via many wires), then the communication is said to be **parallel**. In both cases, timing is essential for the signals to be interpreted correctly.

7.0.3 Latency

Latency is the time it takes for a signal to get from it's start to it's destination. The longer a journey is, then the higher the latency is, and the slower the signal can travel through the medium, the higher the latency will get. Latency is measured in seconds.

7.0.4 Bandwidth

This is the amount of data that can go through a connection per unit time. Usually measured in bits per second. It's set by the width of the data channel (i.e. how many bits can be sent in parallel) and the transmission speed of the bits.

Bandwidth and latency aren't very related, you can increase the bandwidth by widening the data channel (i.e. adding more wires), which will increase the speed of the connection. You can also decrease the latency by reducing the distance the data has to travel (e.g. by moving the RAM nearer to the processor.)

7.1 Serial communications

As said before, a serial communication is (usually) where only one wire is used and different signals are distinguished by being sent at different times. This is good because it only needs one cable (and decoder) and is limited only by the data rate. However, since only one bit can be sent at a time, it may have a low bandwidth, and serial decoding logic will need to be implemented at the receiving end.

One extra wire is always needed for all (electrical) communications in order to establish a common ground level so that both communicating parties are in agreement as to what 'high' and 'low' are.

7.2 Parallel communications

Again, as described above, parallel communications have a width of at least two bits. This means that different elements of the signal can be distinguished by being sent down different paths as well as being delineated by time. An example of this could be a seven segment display.

Parallel connections often require more wires than serial connections, and so need bigger connectors and more interface circuits. They are also unsuitable for some mediums where only one method data path is applicable (e.g. single frequency radio). However, the main advantage is that they can have a far higher bandwidth.

Of course you can always have several serial bits operating in parallel and so get the best of both worlds!

7.3 Synchronous communications

Timing is very important in any serial communication (or parallel communications that are employing serial methods too). Using two different clocks, one at the transmitting end and one at the receiving end is hard, since no two clocks are ever exactly the same, meaning that they could become out of sync. The solution is to use a **common clock**. This requires an extra wire (which doubles the number of wires in a serial communication), but it is worth it since the chance of timing discrepancies drops vastly.

7.3.1 Clock skew

There is still a potential for a phenomenon called **clock skew** though. This is when there is a different latency in the connection for the clock and the connection for the data. This means that the clock will rise and fall just before or after the edge of the data signals. This can be mitigated though by doing a phase shift, so that the signals are mapped onto each other again.

7.3.2 Encoding clock transmission

It is possible to remove the need for an extra clock signal by encoding the clock signal into the data itself. The principle is that the receiving end will assume that there is always going to be a signal transition at a predictable time, and then it can use it's knowledge of the approximate bit rate to regenerate the clock. This is what happens in Ethernet.

Note that we'll still need a ground wire between the sender and receiver.

Manchester encoding is one way of encoding the clock into the signal. It codes a 0 as a falling edge and a 1 as a rising edge. Some other transitions (in between the clock ticks) may be inserted to make this work if there are consecutive $1's$ and $0's$.

7.4 The Universal Serial Bus (USB) protocol

USB is a packed based form of serial I/O communication. A USB cable has four cables; power, ground, data + and data -. The two data wires carry only one signal between them. To provide a better resistance to noise, there are read as carrying a 0 if the difference in voltage is less than 200 mV or a 1 if the difference is greater than 200 mV.

Information is encoded into packages as it is sent via USB, and communicating devices are synchronised by recovering the encoded clock.

7.4.1 Encoding information for USB

The encoding method used in the USB protocol is called *Non Return to Zero Inverted* (NRZI) encoding. This means that any change (i.e. $0 \rightarrow 1$ or $1 \rightarrow 0$) is interpreted as a 0, whereas no change is interpreted as a 1.

Bit stuffing is used to make sure there are no long sequences without any changes in the voltage. This means that a zero (i.e. a change in the signal) is inserted every six consecutive one's.

There are features built in to USB to assist clock synchronisation. A `SYNC` signal is a sequence of `00000001` which is used to facilitate clock synchronisation. There are also many error correction features such as a Cyclic Redundancy Check or the End of the Packet field.

7.5 Asynchronous Serial Communications

An Asynchronous communication is one where there is no transmitted clock signal. Instead, both parties know an approximate transmission frequency and all the symbols in the code have the same length, yet there can be an arbitrary time between symbols.

Having no clock means that Asynchronous communications are often more convenient than synchronous ones since less decoding may be required. In an **Asynchronous Serial Line**, though there is no clock, both communicating parties have decided on the period of transmission per data element. This means that though they will differ slightly, the transmitting and receiving ends of the line won't drift apart too far with their timing, though they must resynchronise every so often to ensure that their clocks do stay within acceptable times of each other. This will slow down communications since extra bits need to be inserted every couple of bytes.

7.5.1 Symbol transmission

In an asynchronous serial communication, there can be any length of gap between symbols. This means that the symbols must be coded so that they are easily distinguishable. Using the *NRZI* encoding, the following may be implemented:

- The symbol starts with a *start bit* which is always a 1
- The data is encoded as 8 bits.
- The symbol ends with an *end bit* which is always a 0.

This means that there is an overhead of two bits for every eight bits of data, and the symbol length is ten bits.

7.5.2 The RS232 serial interface

The RS232 interface is a very common asynchronous serial interface. The data line is idle when it is 'high', and transmission of a symbol is started by the line dropping to 'low' for one bit. This is called the *start bit*. Eight bits will then follow using standard $high = 1$, $low = 0$ encoding. The line will then go back to 'high' again to idle.

7.5.3 Strobing

A strobe line lies parallel to the data line in an interface. The idea is that the strobe going will high will signal that data is being sent over the data line. The sending device will place the data to be sent on a bus, and then wait for the bus to settle. Then it will turn on the strobe and the data will be read off the bus by the receiving device. Once the bus has been read, the strobe will be turned off indicating that transfer has finished.

7.5.4 Handshaking

Handshaking requires two lines that run parallel to the data line; a request line and acknowledge line. In order to set up a data transfer, the following takes place:

- The transmitter goes high on the `request` line.
- The receiver latches onto that and goes high on `acknowledge`.
- The data is sent along the `data` line.
- The transmitter goes low on `request`.
- This prompts the receiver to go low on `acknowledge`.

This is truly asynchronous, since the transmitter must wait for acknowledgement before it can send the data, so we know that it is ready to be received. However, it may require up to eleven wires (eight bits, a request, acknowledge and ground wire).