COMP12111 notes

Todd Davies

December 19, 2013

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 and perhaps try some problems of your own.

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.

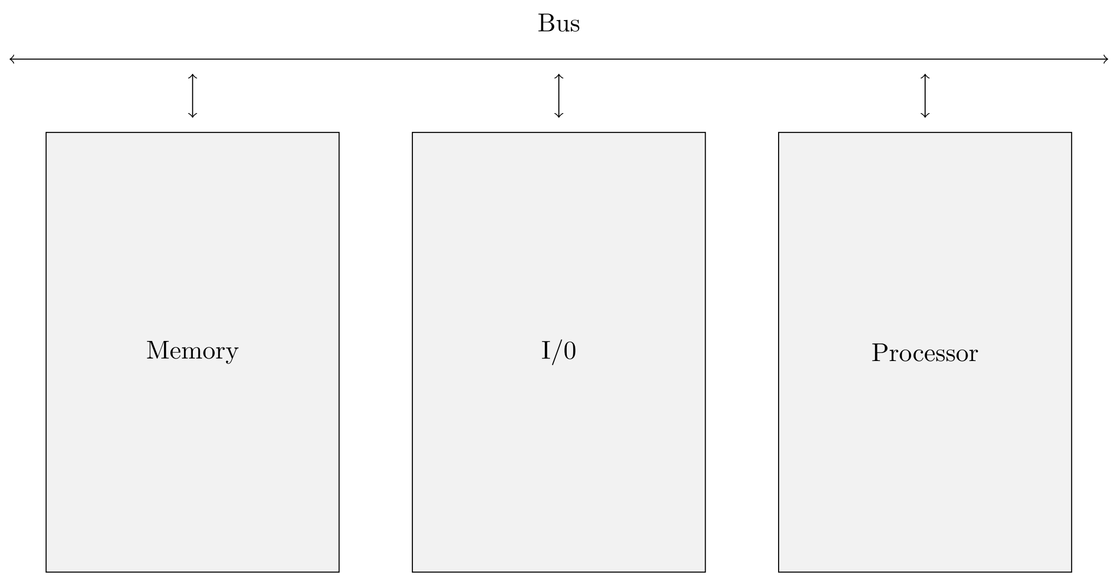

Figure 1: An example of the three box model

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 $2^3 2$ memory locations, but in a 64 bit system, there are $2^6 4$ 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.

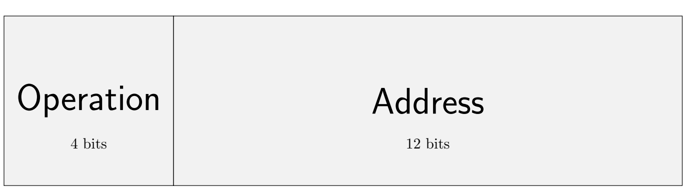

Figure 2: A sample MU0 instruction

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.

3 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.

3.1 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.

3.2 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.

3.2.1 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
```

4 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:

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

5 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).