
CS132

Computer Organisation and Architecture



Contents

1	C Programming	4
1.1	First programme	4
1.2	Data types	4
1.3	Operators	4
1.4	Loops	5
1.4.1	If statement	5
1.4.2	FOR loop (Bounded Iteration)	5
1.4.3	WHILE and DO-WHILE	5
2	Number Conversions	5
2.1	Binary	5
2.1.1	Binary to Octal	5
2.1.2	Binary to Hex	6
2.1.3	Signed Magnitude of Binary	6
2.1.4	Two's Complement Representation of Binary	6
2.1.5	Fixed Point Representation of Binary	7
2.1.6	Floating Point Representation	7
2.1.7	IEEE Floating Point	8
3	Logic and circuits	8
3.1	Simplifying Circuits	8
3.2	Karnaugh Maps (K-maps)	8
3.3	Bit Adder	9
3.3.1	N-bit adder	10
3.3.2	Representing negatives	10
3.4	Active and Inactive states	11
3.4.1	Active High	11
3.4.2	Active low	11
3.5	Decoder	11
3.6	Encoder	12
3.7	Multiplexer	12
3.8	De-Multiplexer	12
3.9	Sequential Logic	12
3.9.1	Flip-Flops	13
3.10	D-Type Latch	13
3.11	Clocked Flip-Flops	14
3.12	N-bit Register	14
3.13	N-bit Shift Register	15
3.14	N-bit Counter	15
3.14.1	First run	15
3.14.2	Second run	15
3.14.3	Third Run	15
3.15	Three-state Logic	15
3.16	Three-state Buses	16
3.17	N-bit Three-state Buses	16
3.18	Properties of Logic Gates	16
3.19	Propagation Delay	17
4	Assembler	17
4.1	Microprocessor Fundamentals	17
4.1.1	Central Processing Unit (CPU)	17
4.1.2	Fetch-Decode-Execute Cycle	18
4.2	68008 Architecture	18
4.2.1	Data Register	18
4.2.2	Status Register	18
4.2.3	Address Register	18
4.2.4	Stack Pointer	18

4.2.5	Program Counter	19
4.3	Register Transfer Language	19
4.4	Instruction Cycle	19
4.4.1	Fetching	19
4.4.2	Decode	20
4.5	Assembly Language	20
4.5.1	Assembler format	20
4.6	68008 Instructions	20
4.6.1	Data Movement	21
4.6.2	Arithmetic Instructions	21
4.6.3	Logical Instructions	22
4.6.4	Subroutines	23
4.6.5	Stacks	23
4.6.6	Addressing Modes	23
5	Memory Systems	23
5.1	Memory Hierarchy	23
5.1.1	Designer's Dilemma	23
5.2	Semiconductor Memory Types	24
5.3	Cache Memory	24
5.3.1	Concepts	25
5.3.2	Types of Cache Miss	25
5.3.3	Measuring Cache performance	25
5.3.4	Hierarchy in Cache	25
5.4	Moore's Law	25
5.5	Memory Cell Organisation	26
5.5.1	SRAM	26
5.5.2	DRAM	26
5.5.3	Comparing DRAM and SRAM	27
5.5.4	Organisation of a Memory Chip	27
5.6	Error Detection and Correction	27
5.6.1	Detecting single and isolated errors	28
5.6.2	Parity	28
5.6.3	Burst Error	28
5.7	Common Memory Components	28
5.7.1	Hard Disks	28
5.7.2	Optical Disks	28
6	I/O Systems	28
6.1	Memory Mapped I/O	28
6.2	Synchronising with I/O devices - Polling	29
6.2.1	Polling and Busy-wait Polling	29
6.2.2	Advantages	29
6.2.3	Disadvantaging	29
6.3	Synchronising with I/O devices - Handshaking	29
6.3.1	Unsynchronised	29
6.3.2	Open-ended handshaking	30
6.3.3	Closed-loop handshaking	30
6.3.4	Timing Diagram	30
6.4	Synchronising with I/O devices - Interrupts	30
6.4.1	Effect of an interrupt input	30
6.4.2	Interrupt Handling Sequence	30
6.4.3	Nested Interrupts	31
6.4.4	Interrupts for I/O Examples	31
6.4.5	Advantages of Interrupts	31
6.4.6	Disadvantages of Interrupts	31
6.5	Direct Memory Access (DMA)	31
6.5.1	DMA Operation	31

6.5.2	Modes of Operation	32
7	Processor Architecture	32
7.1	Microprocessor Organisation	32
7.1.1	Mainstore and Instructions	32
7.1.2	Register	32
7.1.3	Arithmetic Logic Unit	33
7.1.4	Bus	33
7.1.5	Control Unit	33
7.1.6	Internal Organisation	34
7.1.7	Instructions and Control Signals	35
7.2	Macro and Micro Instructions	36
7.2.1	Fetch Macro Instructions	36
7.2.2	Assumptions	37
7.3	Control Unit Design	37
7.3.1	Control Unit Tasks	37
7.3.2	Hardwired CU	37
7.3.3	Advantages of Hardwired CU	38
7.3.4	Disadvantages of Hardwired CU	38
7.3.5	Microprogrammed CU	38
7.3.6	Advantages of Microprogrammed CU	39
7.3.7	Disadvantages of Microprogrammed CU	39
7.4	How could we change our PATP?	39
7.5	RISC and CISC	39
A	A detailed explanation of the recursive relationship	39

1 C Programming

1.1 First programme

Let us create a first programme that prints Hello world!

```
#include <stdio.h>
int main()
{
    printf("Hello_world!\n")
    return 0;
}
```

In C, in order to execute libraries, we must use include <name>. The standard I/O library is stdio.h and gives us access to printf function.

We also indicate that this is a main block of code by declaring our main to be int. Main function is always int.

We include /n to denote a newline, and we have to use return 0 in order to exit our function main.

In order to compile, we must find the location of our .c programme extension. After finding, type gcc -o name name.c in order to compile it. After, just type ./name into console. The -o creates an executable file of the name "name" using name.c.

1.2 Data types

There are 4 primitive data types in C:

```
char // 1 byte , 0 to 255 unsigned , -128 to 127 signed
int  // 2 or 4 bytes , 0 to 6535 unsigned , -32768 to 32766 signed
float // 4 bytes , 32-bit IEEE single precision
double // 8 bytes , 64-bit IEEE double precision
```

Furthermore, you can use the % operator in order to specify what the print should be, e.g. integer, float etc.

```
%c // Character
%d // Signed integer
%e // Scientific notation of floats
%f // float value
%hi // Signed integer but short
%hu // Unsigned integer but short
%i // Unsigned integer
%l // long
%lf // Double
%Lf // long double
%s // string
%lu // unsigned long or int
%lli // long long
%llu // unsigned long long
%u // unsigned int
```

1.3 Operators

Binary arithmetic operators

```
+ // Addition
- // Subtraction
* // Multiplication
/ // Division
% // Remainder
```

Unary arithmetic operators

```
++ // adds +1 before or after depending on its location e.g. ++a
-- // subtracts 1 before or after depending on its location
```

Comparison operators

```
== // Is it equal to
< // less than
> // greater than
!= // not equal to
<= // less than or equal to
>= // greater than or equal to
```

Logical operators

```
& // bitwise and
&& // logical and
| // bitwise or
|| // logical or
! // not
```

The difference between logical is that logical only takes in and returns boolean data type, whereas bitwise can take in integers as well and return the same data type.

1.4 Loops

1.4.1 If statement

The if statement in C is really similar to that in Java. Consider the following example:

```
#include <stdio.h>
```

```
int main() {
    int age = 20;

    if (age < 20)
    {
        printf("You're still a teenager because you are %d\n", age);
    }
    else
    {
        printf("You're no longer a teenager because you are %d\n", age);
    }
}
```

1.4.2 FOR loop (Bounded Iteration)

A for loop is a loop which is done a fixed number of times.

1.4.3 WHILE and DO-WHILE

2 Number Conversions

2.1 Binary

2.1.1 Binary to Octal

We know that the binary system is represented by

$$\sum_{n=0}^k a_n 2^n$$

where $a_n \in \{0, 1\}$ However, notice that, for the octal system represented by

$$\sum_{n=0}^k b_n 8^n = \sum_{n=0}^k b_n 2^{3n}$$

This has several implications. What this means is that, for the coefficient b_n in the octal system, which can go from 0 to 7, can be added from 3 numbers in the binary. If we consider that $1111_2 = 15$, $1000_2 = 8$ and that $111_2 = 7$, we can see that we can get coefficients for our base 8 number system we split up our number by 3 digits. This can be seen more clearly in the following example:

Example 2.1. Conversion 1

Convert 11101001101_2 to base 8.

$$\begin{array}{c|c|c|c} 011 & 101 & 001 & 101 \\ \hline 3 & 5 & 1 & 5 \end{array}$$

Therefore, it is 3515_8

2.1.2 Binary to Hex

With a similar principle,

$$\sum_{n=0}^k c_n 16^n = \sum_{n=0}^k c_n 2^{4n}$$

Similarly $11111_2 = 31$, $10000_2 = 16$ and $1111_2 = 15$, we can see how we can split up our binaries into 4 digits each to calculate the coefficients. We can again clearly see this in an example:

Example 2.2. Conversion 2

Convert 110001110011011101_2 to base 16.

$$\begin{array}{c|c|c|c|c} 0011 & 0001 & 1100 & 1101 & 1101 \\ \hline 3 & 1 & 12 & 13 & 13 \end{array}$$

Therefore, it is $31CDD$

2.1.3 Signed Magnitude of Binary

The signed magnitude of 2 bit is defined by the fact that the most left number, if 1, will become negative. However, the rest of the coefficients will be negative.

Example 2.3. Signed Magnitude

Find 1100101_{2sm} in base 10.

We notice that the number on the most left is 1, Therefore our number must be negative. We now just have to compute 100101 which is $2^6 + 2^2 + 1 =$

$$1100101_{2sm} = -37_{10} \quad (1)$$

The range of numbers for this number system can be found by $Numb \in [-2^{k-1} - 1, 2^{k-1} - 1]$

2.1.4 Two's Complement Representation of Binary

The two's complement will also declare negative with the left most number, however, it does addition for each extra number. That is, the smallest number possible (the biggest negative) for a byte, is for example, $10000000_{2TC} = -128_{10}$ and $10000010_{2TC} = (-128 + 2)_{10} = -126_{10}$

Example 2.4. Finding Two Complement Numbers

Find -9 in a byte of a system in two complement.

$$9_{10} = 1001_2 \quad (2)$$

$$\text{Ensure enough bits: } 00001001_2 \quad (3)$$

$$\text{Invert: } 11110110_2 \quad (4)$$

$$\text{Add 1: } 11110111_2 \quad (5)$$

$$\Rightarrow -9_{10} = 11110111_{2TC} \quad (6)$$

Note that there is also a second method: subtract 1 and then invert. Both of these methods work for any number.

Furthermore, the range of two complements for some base b is $b \in [b^{k-1}, b^{k-1} - 1]$

Example 2.5. Find the range

Find the range of a number represented by $abcd, efgh, ijkl_{2TC}$.

$$Numb \in [-2^{11}, 2^{11} - 1] \quad (7)$$

2.1.5 Fixed Point Representation of Binary

We know in the number system we go in powers of 2. Therefore, 0.11_2 would represent $2^{-1} + 2^{-2}$ in base 10. This, however, it requires a lot of bits because you need a lot of numbers to represent specific decimals.

2.1.6 Floating Point Representation

The floating point representation is similar to the scientific notation, it used the same principles. All numbers are expressed in the form

$$sign \cdot mantissa \cdot b^{\text{biased exponent}}$$

Where mantissa represents the detail of the value to a certain precision;

The exponent represents the magnitude of the value. Note that the biased component is calculated by $\text{biased exponent} = \text{exponent} - (2^{k-1} - 1)$ where k is the amount of digits. This allows for negative numbers. Therefore, it follows that the biased exponent $\in [-b^{k-1} + 1, b^{k-1}] \cap \mathbb{Z}$. and b represents the base. In computers it's almost always 2.

Example 2.6. Base 10

Given that IEEE 754 is expressed 1 bit for sign, 23 bits for mantissa and 8 bits for exponent, express -12345_{10} in IEEE 754 base 2. \Rightarrow Our sign is 1 as it is negative.

$$12345_{10} = 11000000111001_2 \quad (8)$$

$$\Rightarrow 1.1000000111001 \cdot 2^{13} \quad (9)$$

$$\Rightarrow \text{Mantissa} = 10000001110010000000000 \quad (10)$$

$$\text{Note that we add more 0s to the mantissa for 23 bits} \quad (11)$$

$$\Rightarrow 13 + 127 = 140 \quad (12)$$

$$\Rightarrow 140_{10} = 10001100_2 \quad (13)$$

$$\text{Ensure that the exp is 8 bits.} \quad (14)$$

Therefore our final answer is

$$1100011001000000111001 \quad (15)$$

2.1.7 IEEE Floating Point

IEEE Standard 754 is widely used and specifies levels of binary precision.

Single precision (uses 32 bits)

Double precision (using 64 bits)

Quad precision (using 128 bits)

For example, single precision uses 1 bit for the sign, 8 bits for the exponent and 23 bits for the mantissa.

3 Logic and circuits




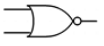

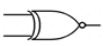

Logic Gate	Symbol	Description	Boolean
AND		Output is at logic 1 when, and only when all its inputs are at logic 1, otherwise the output is at logic 0.	$X = A \cdot B$
OR		Output is at logic 1 when one or more are at logic 1. If all inputs are at logic 0, output is at logic 0.	$X = A + B$
NAND		Output is at logic 0 when, and only when all its inputs are at logic 1, otherwise the output is at logic 1	$X = \overline{A \cdot B}$
NOR		Output is at logic 0 when one or more of its inputs are at logic 1. If all the inputs are at logic 0, the output is at logic 1.	$X = \overline{A + B}$
XOR		Output is at logic 1 when one and Only one of its inputs is at logic 1. Otherwise is it logic 0.	$X = A \oplus B$
XNOR		Output is at logic 0 when one and only one of its inputs is at logic 1. Otherwise it is logic 1. Similar to XOR but inverted.	$X = \overline{A \oplus B}$
NOT		Output is at logic 0 when its only input is at logic 1, and at logic 1 when its only input is at logic 0. That's why it is called and INVERTER	$X = \overline{A}$

Figure 1: Logic gates

It is sufficient to learn 4 gates: OR, AND, NOT and XOR. Rest are intuitive in symbol. You can also simplify boolean expressions using boolean logic or using karnaugh maps.

3.1 Simplifying Circuits

When given a function we need to be able to create equivalent circuits to meet several design criteria. These are:

1. Perform the designated function
2. Use the types of gates available
Minimise the number of gates used and hence cost

3.2 Karnaugh Maps (K-maps)

Karnaugh maps are tables which consider 2D truth tables. You would first construct a truth table for all the variables, and then map them onto the K-map. For example,

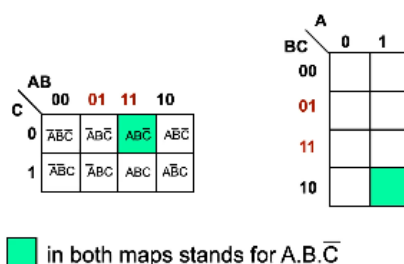


Figure 2: Karnaugh Maps for triple variables

Notice that these are "gray coded", which means that adjacent variables only differ by 1 boolean expression. That is,

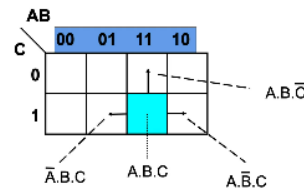


Figure 3: Gray coded K-map

The first thing to also notice is the wrap around of Karnaugh maps. That is, in the figure below

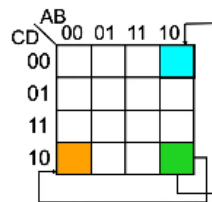


Figure 4: K-map Wrapping

The bottom right green shaded square is actually adjacent to the other shaded corner squares. The top right is only adjacent to the bottom right, whereas the bottom left is only adjacent to also the bottom right. The second thing to take note of is that K-maps are minimised when the number of groups (note that groupings must contain 2^n elements) that are identified are also minimal.

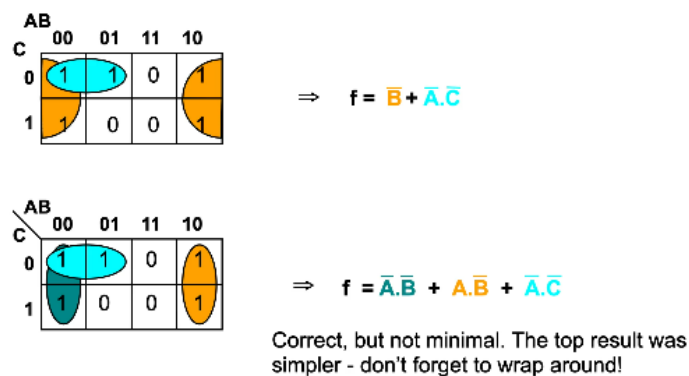


Figure 5: Grouping

In the first K-map, we notice in that in the orange circle that \overline{B} does not change, therefore we can write it. Similarly, in the teal colour, we notice that $\overline{A}C$ does not change, therefore we can also write it. In the second example, we have 3 examples, which work, but it is not minimal! You can also have "Don't care conditions" which are denoted by red x , where we do not care about a specific square's value.

3.3 Bit Adder

A bit adder is a logical circuit which has a carry on and a sum of 3 inputs. The sum 'S' functions by checking the number

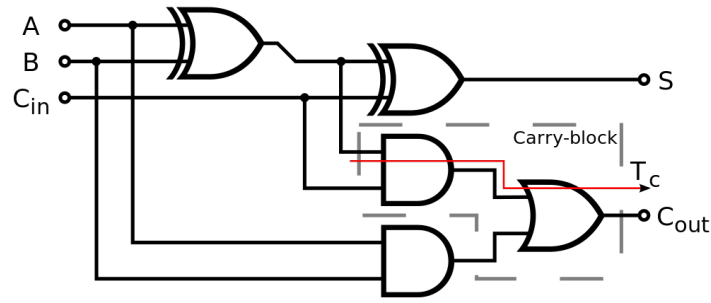


Figure 6: Bit adder circuit

Table 1: Full Adder Truth Table

Inputs			Outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.3.1 N -bit adder

Notice that the above only supports 2 bits of entry. We can, however, support multiple bits by combining 2 bits. That is, by combining $\times 2$ 2 bits. By having an N number of bit adders combined, we achieve N bit adders.

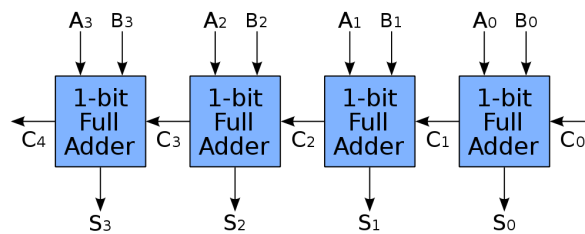


Figure 7: 4 Bit adder

3.3.2 Representing negatives

We can do this by introducing two-complement circuit Z .

$$Z = 0 \implies S = A + B$$

$$Z = 1 \implies S = A - B$$

We can convert the bit adder to also do negatives by introducing two's complement. As we know,

1. Invert the N -bit number B by finding $Z \oplus B$.
2. Add 1 (Carry in)

Table 2: Z switch

Z	B	Result
0	0	0 = B
0	1	1 = B
1	0	1 = \overline{B}
1	1	0 = \overline{B}

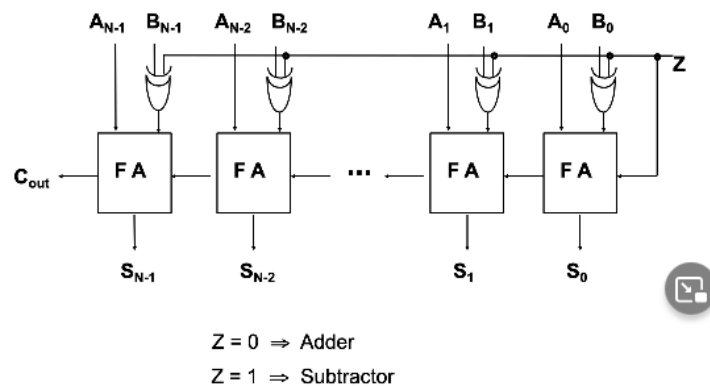


Figure 8: Subtractor

As from the previous thing we've discussed, the $+1$ is from the fact that if $Z = 1$, the input is already added 1 ! Note that this does not directly add to B , but this is fine as the $+1$'s position does not matter.

3.4 Active and Inactive states

Circuits can be defined using "active" and "inactive" states, which can change the meaning of the values 0 and 1.

3.4.1 Active High

In active high, it follows that:

0 = inactive
1 = active

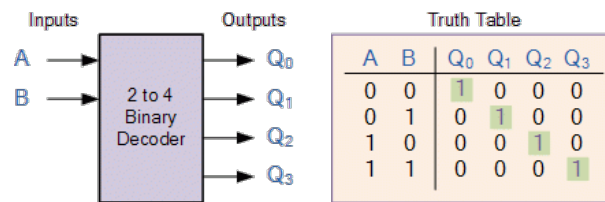
3.4.2 Active low

In active low, it follows that:

0 = active
1 = inactive

3.5 Decoder

A decoder is a circuit such that for n amount of entries, it will have 2^n amount of outputs to count in for each possibility. It follows that, the addition of the entries will in fact determine in which row the 0 will take place in. The description is more clear when noticed in the figure below:

Figure 9: 2×4 Decoder

Note that this is a high active binary decoder.

3.6 Encoder

Encoder does the exact opposite of a decoder. In some sense, it takes 2^n inputs, and has n amount outputs.

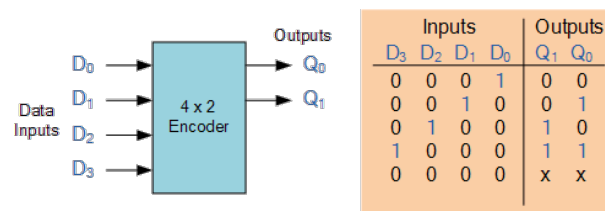
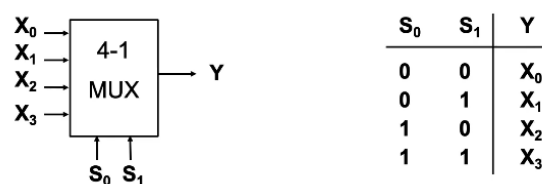


Figure 10: Encoder

3.7 Multiplexer

In multiplexers, output is a selected input. The sum of the gates determine the value of this customised input.



$$Y = X_0 \cdot \bar{S}_0 \cdot \bar{S}_1 + X_1 \cdot \bar{S}_0 \cdot S_1 + X_2 \cdot S_0 \cdot \bar{S}_1 + X_3 \cdot S_0 \cdot S_1$$

Figure 11: 4 to 1 Multiplexer

3.8 De-Multiplexer

In de-multiplexers, you allow one of the outputs to be any one of the outputs.

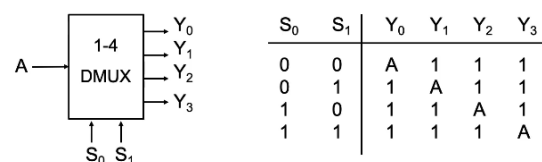


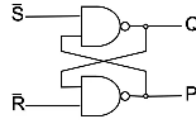
Figure 12: De-multiplexer

3.9 Sequential Logic

A logic circuit whose outputs are logical functions of its inputs and its current state.

3.9.1 Flip-Flops

Flip flops are a circuit whose outputs are fed back to inputs. In an active low flip flop, the set and reset have a bar over them. Furthermore, when set bar is set to 0, Q is set to 1. When reset is set to 0, Q is set back to 0. Note that $\neg P = Q$.



\bar{S} and \bar{R} are active low **SET** and **RESET** inputs

Consider: $\bar{S} = 1, \bar{R} = 0 \Rightarrow P = 1, Q = 0$
 then if $\bar{S} = 1, \bar{R} = 1 \Rightarrow P = 1, Q = 0$
 and then $\bar{S} = 0, \bar{R} = 1 \Rightarrow P = 0, Q = 1$
 then $\bar{S} = 1, \bar{R} = 1 \Rightarrow P = 0, Q = 1$

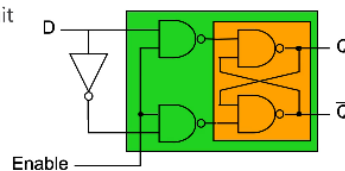
Figure 13: Active Low Flip-Flop

Essentially, Q is set to one when $\bar{S} = 0$ (and $\bar{R} = 1$)
 And Q is Reset (to zero) when $\bar{R} = 0$ (and $\bar{S} = 1$).
 If $\bar{S} = \bar{R} = 1$, then Q does no change.
 However, when $\bar{S} = \bar{R} = 0$, we get a hazard condition.

3.10 D-Type Latch

Using a modified form of the flip-flop circuit we've seen it is possible to design the D-type latch, which is essentially a 1-bit memory circuit.

Essentially a 1-bit memory circuit



Enable	D	Q	\bar{Q}
0	0	Q	\bar{Q}
0	1	Q	\bar{Q}
1	0	0	1
1	1	1	0

Output can change only when the Enable line is high, i.e., enable triggers circuit operation.

D-type latch is a 1-bit memory with the Enable terminal as the 'WRITE' command.

Figure 14: D-Type Latch

In the figure above, D is a 1-bit data that is either 0 or 1. The value of D is stored into Q if and only if Enable is 1.

Table 3: D-Type Latch Truth Table 1

Enable	D	Q	$\neg Q$
0	x	Q	$\neg Q$
1	0	0	1
1	1	1	0

Table 4: D-Type Latch Truth Table 2

Enable	D	Q	$\neg Q$
0	0	Q	$\neg Q$
0	1	Q	$\neg Q$
1	0	0	1
1	1	1	0

Table 5: D-Type Latch Truth Table 3

Enable	Q	$\neg Q$
0	Q	$\neg Q$
1	D	$\neg D$

3.11 Clocked Flip-Flops

In some cases, D -type is a special case of a flip-flop in which the enabler is a clock. That is, the clock only outputs true when the clock is 'rising', i.e. transitioning from low to high. There are many circuits in which a clocked flip-flop is used, most of which that are relevant for the module can be found in the figure below:

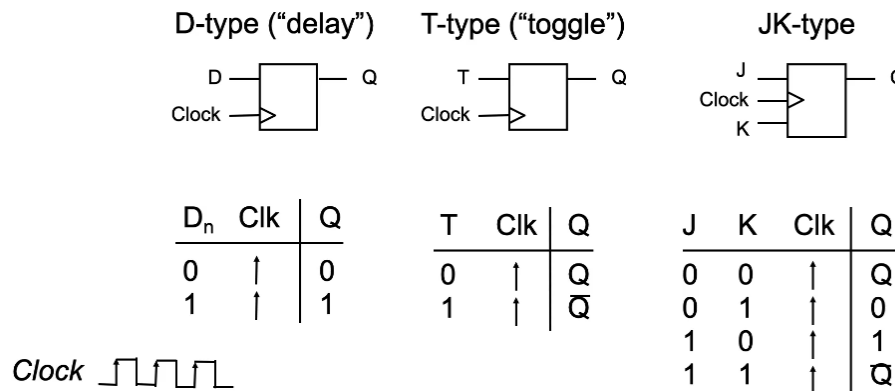
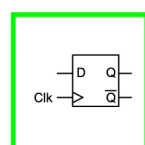


Figure 15: Examples of Clocked Flip-Flops

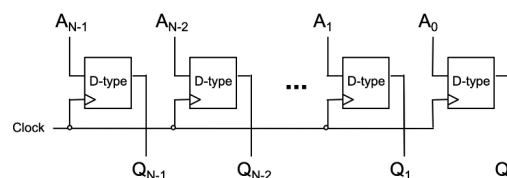
However, the module will primarily be focusing on D -type.

3.12 N-bit Register

An N-bit register is a multi-bit memory that allows us to store n bits using a D -type latch.



A multi-bit memory



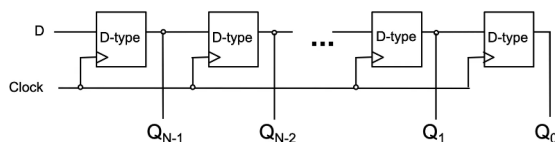
N-bit number $A_{N-1} A_{N-2} \dots A_1 A_0$ is stored in the register on a low to high transition of the clock

Stored number appears on the outputs $Q_{N-1} \dots Q_0$

Figure 16: Parallel Load N-bit Register

3.13 N-bit Shift Register

The important part of this register is that once the clock is ran once, the input of D will go to Q_{N-1} and will be stored there. Then, we can change the input D again and store that in Q_{N-1} and store the old Q_{N-1} in Q_{N-2} once we run the clock again.



When a clock transition occurs, each bit in the register will be shifted one place to the right

Figure 17: N-bit Shift Register

3.14 N-bit Counter

The N-bit Counter is a complex circuit.

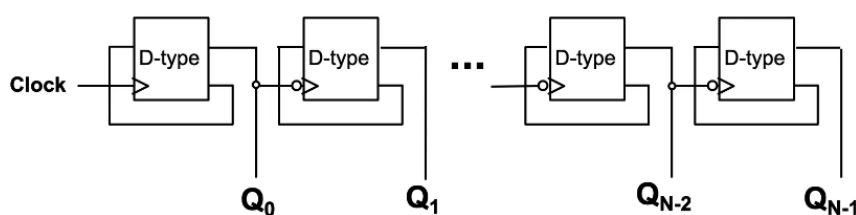


Figure 18: N-bit Counter

One thing to note is that the triangle with the circle denotes an inverter, that is, it is a clock that only registers when a value drops. Let us run this circuit 3 times to see the logic behind it.

3.14.1 First run

We first begin with the fact that we are at 000. That is, each $Q = 0$. Now, let us switch the clock and we get a 1 in the input for the enabler. Because $Q_0 = 0$, $\neg Q_0 = 1$, therefore our inputs into our first D-type are 1 and 1. Then, the D-type stores the 1 into Q_0 . Then, our new $\neg Q_0 = 0$, which is fed back into the D-type. Note that it stops here, because we have increased our value from 0 to 1 for Q_0 , therefore the inverter does not function.

3.14.2 Second run

In the second run, we insert a 1 again. Our D -type has the inputs 0 and 1, which means that Q_0 is now a 0. Since Q_0 has dropped in value, the inverter clock on the second D-type was triggered, which now has inputs of 1 and 1, setting $Q_1 = 1$.

3.14.3 Third Run

The third run is similar to the first. We run the clock and we get the inputs 1 and 1, which sets Q_0 to 1, and does not proceed.

3.15 Three-state Logic

The logical components we've considered so far had only 0 and 1. There is another gate, called the three-state buffer, whose output can be placed in a third state defined as 'UNCONNECTED'

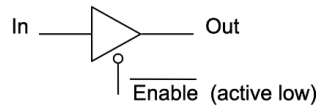


Figure 19: Three-state Logic

When \overline{Enable} is high, the input is disconnected from the output
 When \overline{Enable} is low, the input is connected to the output

3.16 Three-state Buses

Consider a bus with an output that we want to share on multiple D-types. However, we cannot have multiple outputs at the same time. Thus, we use a three-state logic connected to our buses to give control lines.

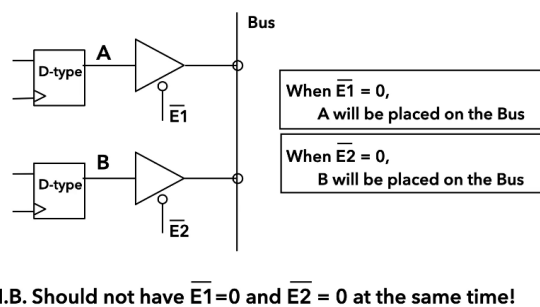


Figure 20: 1 Bit Three-state Bus

3.17 N-bit Three-state Buses

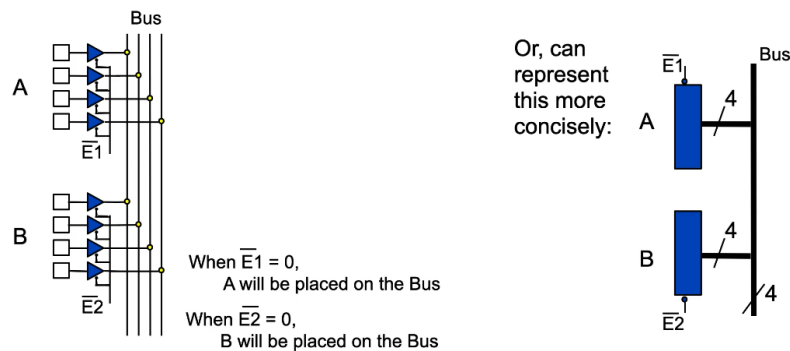
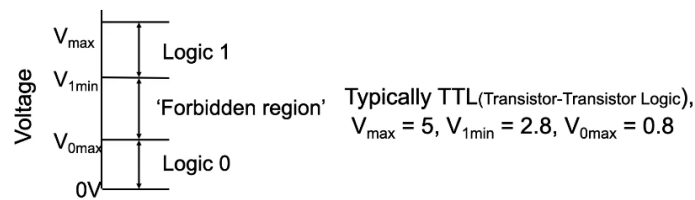


Figure 21: 4-Bit Three-state Bus

3.18 Properties of Logic Gates

There are several things to remember about how particular a 0 or a 1 is determined in a circuit, specifically that

Logic values by Voltage level:



Do NOT connect outputs together except three-state buffers

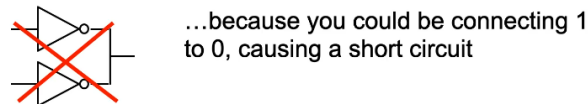


Figure 22: Properties of Logic Gates

3.19 Propagation Delay

Each logic has a propagation delay, typically in nano-seconds, (1×10^{-9} s or less, limiting the speeds of logic circuits. However, this can be reduced by ensuring logic gates are close together and are on the same integrated circuit.

Table 6: Propagation Delay

Gate	Propagation Delay (ns)
NOT	1.0
NAND	1.2
AND	1.7
OR	1.2
NOR	1.5

4 Assembler

4.1 Microprocessor Fundamentals

4.1.1 Central Processing Unit (CPU)

The CPUs control and perform instructions. They use two units that are the following:

1. Arithmetic Logic Unit (ALU) - Performs mathematical and logical operations
2. Control Unit (CU) - Decodes program instructions and handles logistics for the execution of decoded instructions.
3. Program Counter - Tracks the memory address of the next instruction to be executed
4. Instruction Register (IR) - Contains most recent instruction fetched
5. Memory Address Register (MAR) - Contains address of the region of memory to be read or written, i.e. location of data to be accessed
6. Memory Data Register (MDR) - Contains data fetched from memory or data to be written to memory. Also known as Memory Buffer Register.

The CPU continuously performs instruction cycle. These are retrieved from memory, decoded to form recognisable operations and executed to impact the current state of CPU.

4.1.2 Fetch-Decode-Execute Cycle

It consists of the following: Fetch

1. Instruction retrieved from memory held by program counter
2. Retrieved instruction stored in instruction register
3. Program Counter incremented to point to next instruction in memory

Decode

1. Retrieved instruction / operation code / opcode decoded
2. Read effective address to establish opcode type

Execute

1. Control unit signals functional central processing unit components
2. May result in changes to data registers, that is, the program counter, arithmetic logic unit, input/output etc.

4.2 68008 Architecture

4.2.1 Data Register

D0-D7, 32 bit registers which can partition themselves to 8,16 as well, store frequently used values / intermediate results. Strictly speaking we would only require one register on a chip. The advantage of many data registers is that fewer references to external memory are required. Registers can be treated as long, word or byte, that is, 32 bits, 16 bits or 8 bits respectively.

4.2.2 Status Register

The status register is 16 bits, and consists of two 8-bit registers. It explains the current status of the processor, that is, where it is currently. Its status bits are set or reset upon certain conditions or commands that happen in the ALU.

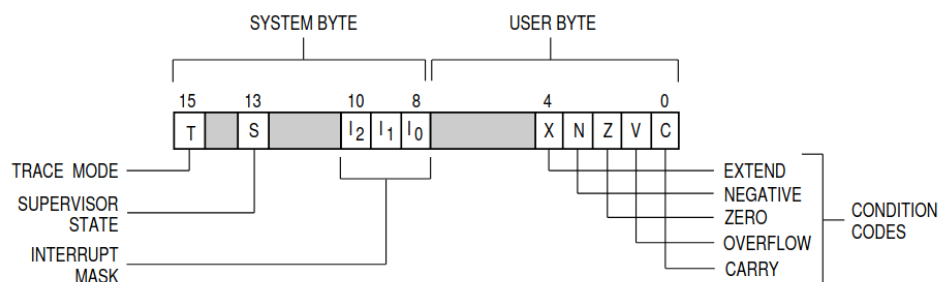


Figure 23: Status Register

4.2.3 Address Register

The Address Registers use A0-A6, 32 bit registers. These store addresses that we may use for future operands. That is, we are able to return to a memory address that we previously were at. The A7 is additionally used by the processor as a system slack pointer to hold subroutine routine addresses etc. It is important to note that operations on addresses do not alter the CCR.

4.2.4 Stack Pointer

Stack is a type of architecture that stores the latest operations.¹. The stack pointer is essentially used in A7 of the Address Register. It is also possible to use A0-A6 as stack pointers if needed.

¹Tenenbaum p.250-268 is worthy of read to understand fully

4.2.5 Program Counter

A 32 bit register that keeps track of the address at which the next instruction will be found. In simple terms, it points to the next instruction in memory.

4.3 Register Transfer Language

It is used to describe the set of operations of a microprocessor as it is executing instructions. E.g. $[MAR] \leftarrow [PC]$ means transfer of Program Counter to the Memory Address Register. The computer's main memory is called Main Store. The contents of memory location 12345 is written $[MS(12345)]$. It is important not to confuse register transfer language with assembler instructions.

4.4 Instruction Cycle

The RTL representation we will see makes no attempt to account for the pipelining of instructions. Pipelining is a common and a simple method of speeding up the fetch-execute cycle.

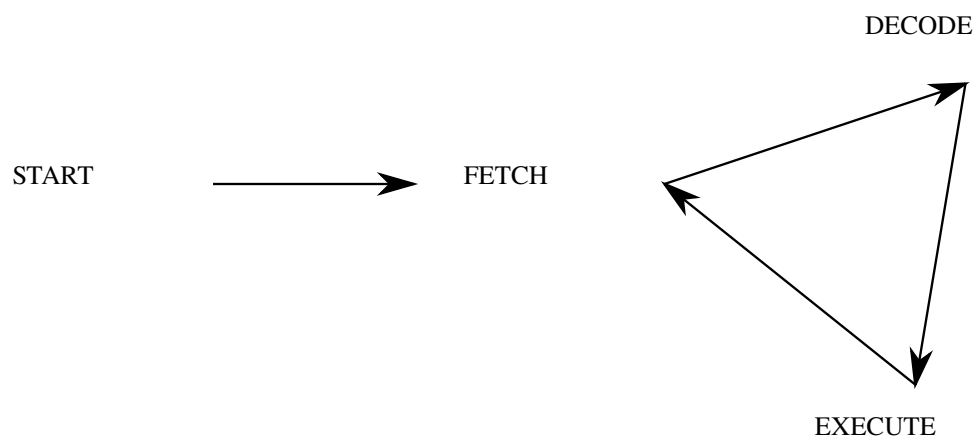


Figure 24: LFT Cycle

4.4.1 Fetching

1. Contents of Program Counter transferred from MAR address buffers and the Program Counter is incremented
2. MBR loaded from external memory (R/\overline{W} line set to Read)
3. Opcode transferred to Instruction Register from MBR
4. Instruction is decoded

In RLT, this is

1. $[MAR] \leftarrow [PC]$
2. $[PC] \leftarrow [PC] + 1$
3. $[MBR] \leftarrow [MS([MAR])]$ (R/\overline{W} set to Read)
4. $[IR] \leftarrow [MBR]$
5. $CU \leftarrow [IR(opcode)]$

And it is the same for every instruction set.

4.4.2 Decode

4.5 Assembly Language

In coding, the hierarchy Follows. Assembly language can give you the control over instructions you give to your

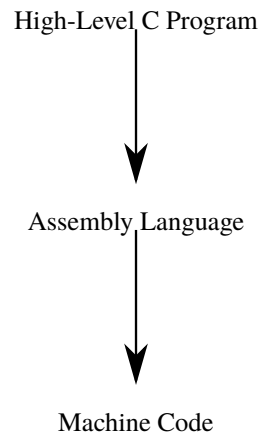


Figure 25: hierarchy

microprocessor.

4.5.1 Assembler format

Assembly language vary but typically have similar format:

`< Label > : < OPCODE > < OPERAND(S) > | COMMENT`

e.g.

`START : move.b 5, D0` | Load D0 with 5

ORG	\$4B0	this program starts at hex 4B0
move.b #5, D0		load D0 with 5
add.b	#\$A, D0	add 10 to D0
move.b D0, ANS		store result in ANS
ANS: DS.B	1	leave 1 byte of memory empty
		and give it the name ANS

Numbers:

: indicates a constant. A number without # prefix is an address

Default number base is DECIMAL

\$: means in HEX

% : means in BINARY

Assembler Directives:

ANS: A label, i.e., a symbolic name, that must be terminated by a colon

DS (Define Storage): instructs the assembler to reserve some memory

ORG (Origin): tells the assembler where in memory to start putting the instructions or data

Figure 26: Assembly Language

In other words, in 68008 it is *operation.datatype source, destination* for movement.

4.6 68008 Instructions

The 68008 instruction set is made up five categories of instructions:

1. Data movement
2. Arithmetic
3. Logical
4. Branch
5. System Control

For this course, the first four are sufficient.

4.6.1 Data Movement

Data instructions are written in the form

operation.datatype source, destination

The operation can be on one of the three data types:

byte .b(8 bits)
word .w(2 bytes)
longword .l(4 bytes)

In default, the data type is word.

move.b D0, D1	[D1(0:7)] ← [D0(0:7)]
moveb D0, D1	the same
move.w D0, D1	[D1(0:15)] ← [D0(0:15)]
move D0, D1	the same
move.l \$F20, D3	[D3(24:31)] ← [MS(\$F20)]
	[D3(16:23)] ← [MS(\$F21)]
	[D3(8:15)] ← [MS(\$F22)]
	[D3(0:7)] ← [MS(\$F23)]
	(Big-Endian mean this way around)
exg.b D4, D5	exchange
swap D2	swap lower and upper words
lea \$F20, A3	load effective address [A3] ← [\$F20]

Figure 27: I have 0 clue what the fuck this says

4.6.2 Arithmetic Instructions

add.l Di, Dj	[Dj] ← [Di] + [Dj]
addx.w Di, Dj	also add in x bit from CCR
sub.b Di, Dj	[Dj] ← [Dj] - [Di]
subx.b Di, Dj	also subtract x bit from CCR
mulu.w Di, Dj	unsigned multiplication:
	[Dj(0:31)] ← [Di(0:15)] * [Dj(0:15)]
muls.w Di, Dj	signed multiplication
divu.b Di, Dj	
divs.l Di, Dj	

Figure 28: Arithmetic Instructions

4.6.3 Logical Instructions

Logical AND:

AND.B #\$7F, D0

D0

1	1	0	1	1	0	1	0
---	---	---	---	---	---	---	---

7F 0 1 1 1 1 1 1 1

D0	0	1	0	1	1	0	1	0
----	---	---	---	---	---	---	---	---

(keep just the lower 7 bits and ignore the MSB)

Logical OR:

OR.B D1, D0

D1

1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

D0

0	0	0	1	0	1	1	0
---	---	---	---	---	---	---	---

D0	1	0	0	1	0	1	1	0
----	---	---	---	---	---	---	---	---

Figure 29: Logical Instructions

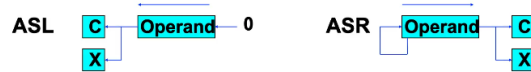
Similarly, shifts work by transferring the bits to the left or right. We can do this by multiplying by 2.

- Logical Shift (L - left, R - right)



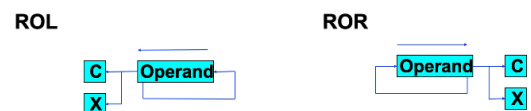
(C and X are in Condition Code Register)

| Arithmetic Shift



(ASR preserves sign bit)

Figure 30: Shifts left and right

| **RO**tate

- ROtate through eXtend bit**



Figure 31: No fucking clue

4.6.4 Subroutines

Subroutines are useful for frequently used sections of a program. Write and debug a subroutine once, and use that program code whenever it is needed. It reduces program size and improves readability. The commands are *JSR < label >* which is jump to subroutine and *RTS* return from subroutine.

4.6.5 Stacks

A stack can be used to capture the last in first out (LIFO) aspect of the assumption we were making. The use of stacks for subroutine calls is common. The JSR pushes the contents of the PC on the stack. Puts start address of subroutine in PC. The return from Subroutine pops the return address from the stack and puts it in PC, so the stack where to from a subroutine.

4.6.6 Addressing Modes

How we tell the computer where to find data it needs. It requires to organise application data. Some data never changes, some is variable and some needs to be located within a data structure, e.g. list, table or array. In 68008 systems data can be located in data register, within the instructor itself or in external memory.

Example 4.1. Example

"Heres 100\$" This is a literal value.
 "Get the cash from Rootes room 19" This is absolute address.
 "Go to Rootes room 23 and they'll tell you where to get the cash" This is indirect address
 "Go to rootes room 42 and get the cash from the fifth room to the right" This is relative address.

1. Data or Address Register Direct - the address of an operand is specified by either a data register or an address register e.g. `move D3, D2`
2. Immediate Addressing - The operand forms part of the instruction and remains constant throughout the execution of a programme. E.g. `move.b $42, D5`
3. Absolute Addressing - The operand specified the location in memory explicitly, meaning no further processing required. E.g. `move.l D2, $7FFF0`
4. Relative Addressing - Code like this contains no absolute addresses, i.e. it uses only addresses relative to the current program counter, and therefore can be placed anywhere in memory.

5 Memory Systems

5.1 Memory Hierarchy

To construct a memory system, we must think carefully about the properties of each storage approach. Many factors include the choice of memory technology such as:

- Frequency of access
- Access time
- Capacity required
- Cost e.g. cost per bit

5.1.1 Designer's Dilemma

The desire for low cost and high capacity - leads to choice of cheap per bit, high capacity and slow access time
 Desire high performance - leads to choice of expensive per bit, low capacity and fast access time
 In order to solve this problem, we have to choose both by organising memory into a hierarchy. This leads to the memory hierarchy:

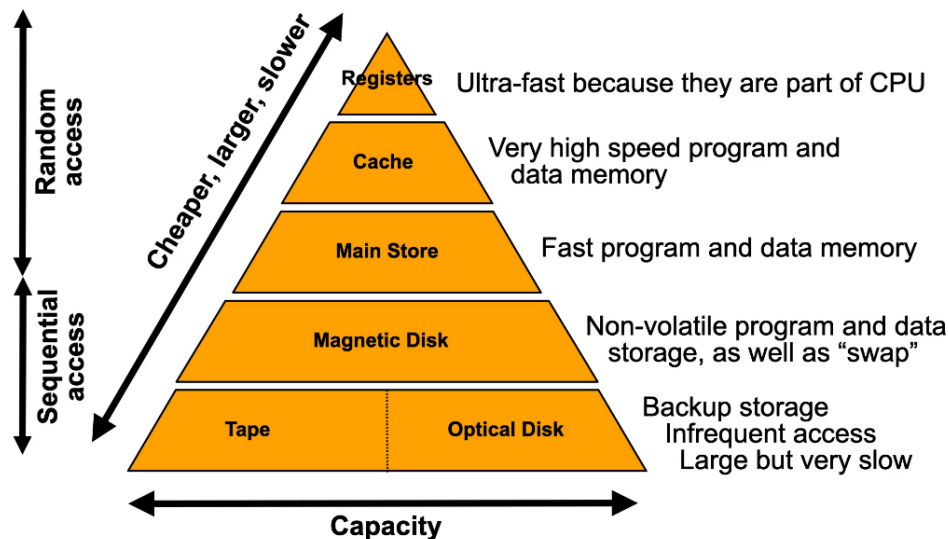


Figure 32: The Memory Hierarchy

The reason we have such a hierarchy is indeed because of economics.

Definition 5.1. Temporal Locality

If a particular memory location is referenced, it is likely that the same location will be referenced again in the near future

Definition 5.2. Spatial Locality

If a particular memory location is referenced, it is likely that nearby memory locations will be referenced in the near future

A reasonable assumption is that it has been shown that 90% of memory accesses are within ± 2 kilobytes of previous PC position.

5.2 Semiconductor Memory Types

Table 7: Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random Access Memory (RAM)	Read-write	Electrically at byte-level	Electrically written	Volatile
Read-only Memory (ROM)	Read-only	Not possible	Mask written	Non-volatile
Programmable ROM (PROM)	Read-only	Not Possible	Electrically written	Non-volatile
Erasable PROM (EPROM)	Read-mostly	UV-light at chip-level	Electrically written	Non-volatile
Electrically Erasable PROM (EEPROM)	Read-mostly	Electrically at byte-level	Electrically written	Non-volatile
Flash Memory	Read-mostly	Electrically at block-level	Electrically written	Non-volatile

We are particularly interested in random access. Furthermore, RAM is most common type of semiconductor memory, but its title is misleading since all things in the table allow random access.

5.3 Cache Memory

Cache is using the term "90% of memory accesses within only 2 KBytes" So store those 2 KBytes in a small, fast "cache" memory. If data required by CPU is in the cache, it is a huge speed improvement. It is also small to limit cost. It is also possible to have multilevel caches, which there are levels of cache depending on the range of KBytes.

5.3.1 Concepts

Cache read-only data is relatively straightforward. We don't need to consider the possibility that items will change, hence copies across memory hierarchy remain consistent. Two general strategies are adopted when caching writes, both of which can employ a buffer to enhance performance

- Write Through - update the item in cache and writes through to update lower levels of memory hierarchy
- Write Back - only updates copy in cache, ensuring that blocks are copied back to memory when they are replaced

5.3.2 Types of Cache Miss

Measures of cache performance include hit rate (h) and miss rate ($1 - h$).

$$h = \frac{\text{Number of times the words are in cache}}{\text{Total number of memory references}}$$

We can understand cache misses by sorting all misses into categories:

- Compulsory - misses that would occur regardless of cache size e.g. the first access to a block cannot be in cache (regardless of size) so the block must be retrieved
- Capacity - misses the occurring because a cache is not large enough to contain all blocks needed during the execution of program
- Conflict - misses occurring as a result of the placement strategy for blocks not being fully associative, which means that a block may be discarded or retrieved
- Coherency - misses occurring due to cache flushes in multiprocessor systems

5.3.3 Measuring Cache performance

Performance measures based solely on hit and miss rates don't factor in the cost of a cache miss. Average memory access time is an alternative measure that accounts for the cost of a cache miss

$$\text{Average memory access time} = \text{Hit time} + (\text{Miss rate} \times \text{Miss Penalty})$$

This is where hit time is the time to hit in the cache and miss penalty is the time taken to replace the block from the memory. Sometimes, however, this isn't even good enough and execution time can often be a better measure in real-world situations.

5.3.4 Hierarchy in Cache

Data storage is a balance of size and speed. Memory is closer to the CPU are the Cache with levels 1, 2 and 3. These are fast but have smaller capacity. Main memory is further and slower but has a far greater capacity and hard disks have the greatest capacity and do not need continuous power, but have the slowest access times. The optimisation then would consist through the use of fastest memory possible for highest performance, but problems may large datasets. The solution, is then, to reuse data already in cache as much as possible and prefetch data from main memory into cache before it is needed, masking load times.

5.4 Moore's Law

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly of the short term this rate can be expected to continue, if not increase. Over the long term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.". This is not commonly expressed as "The number of transistors on a memory chip doubles every 18 months". The effects of this are:

- The cost of computer logic and memory circuitry has fallen at a dramatic rate
- As logic and memory is placed closer together on more densely packed chips, the electrical path is shortened, increasing operating speed

- Computers have become smaller, making them more convenient for use in a variety of environments
- Reduction in power and cooling requirements
- Interconnections on an IC are more reliable than solder connections, thus having off-chip connection increases reliability.

Unfortunately, as clock speeds and everything else has improved, the memory access speed is improving much more slowly. As such, cache is the core to addressing this challenge.

5.5 Memory Cell Organisation

Most common form of main story is the RAM, which break down into two main technologies which are:

- Static RAM (SRAM)
- Dynamic RAM (DRAM)

SRAM uses flip-flop storage element for each bit.

DRAM for each bit, uses the presence or absence of charge in a capacitor to denote a 1 or 0. Capacitor charge unfortunately leaks away over time, requires period refreshing, but it is cheaper than SRAM.

5.5.1 SRAM

SRAM stores data using configurations of flip-flops and logic gates, and the memory cells hold data as long as there is power supplied. It uses four cross-coupled transistors to establish a stable logic state.

5.5.2 DRAM

DRAM memory cells store data as charge on capacitors. Unfortunately, it leads to these issues:

- Capacitors naturally discharge, hence DRAM cells require periodic charge to maintain data storage
- The term 'dynamic' refers to the natural leakage of charge that takes place even when power is supplied
- Presence or absence of charge is interpreted as logical 0 or 1

We can consider the appreciated the operation of DRAM by considering a 1 bit DRAM memory cell. Address line is activated when the bit value from the cell to be read or written A transistor acts as a switch that is closed (allowing current flow) if a voltage is applied to the address line and open if a voltage is not applied to the address line (no flow)

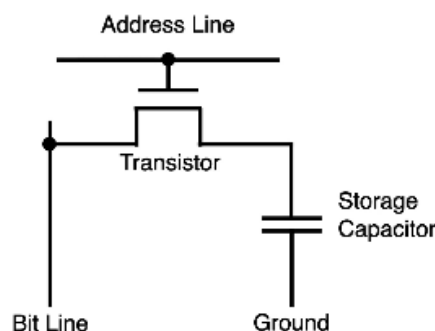


Figure 33: DRAM

The write operations are as follows:

- The voltage applied to the bit line which is high for a logical 1, and low for a logical 0.
- A signal is then applied to the address line, allowing a charge to be transferred to the capacitor (data store)

The read operations are as follows:

- Transistor turns on when address line is selected, allowing the stored charge to be fed out to the bit line and a sense amplifier
- Sense amplifier compared the capacitor voltage to a predetermined threshold value to determine if it is a 0 or a 1
- The read process discharges the capacitor, which means that it must be restored as well as refreshed following the read operation

5.5.3 Comparing DRAM and SRAM

Both are volatile - power must be continuously supplied Dynamic memory cells are generally simpler and more compact which allow for greater memory cell density and cheaper to produce than equivalent SRAM memory. Refresh circuitry incurs one-off cost that is only compensated for by larger memory capacities
SRAM are typically provide better read and write times than DRAM, meaning that Cache uses mainly SRAM and main memory uses DRAM.

5.5.4 Organisation of a Memory Chip

It is critical to maintain a structured approach to the organisation of on-chip memory.

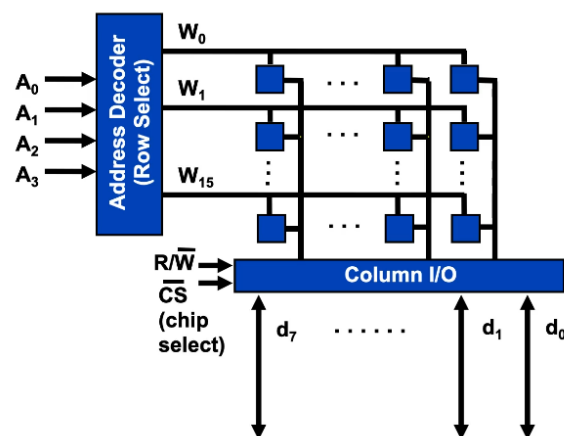


Figure 34: 16×8 memory integrated circuit (16×8 denotes $W \times B$ i.e. words \times bits, each of 8 bits)

IC design for $16 \times 8 = 128$ cells required 4 address inputs $= \log_2 W$, where W = no of words and 8 data lines B , where B = word size. Consider a 1 KBit = 1024 cells device. It is possible to organise as a 128×8 memory cell array, requiring 7 address pins $\log_2 128$ and 8 data pins, which is a total of 15 input output pins, plus power etc. Thus, we could take different approaches, but what would be the best way? A 1024×1 would require 10 address pins and 1 data pin, a total 11 IOs etc. isn't as efficient. The goal is to minimise decoding space. This could be done by dividing the address inputs into 2 parts - row address and column address. It is then most efficient to have it as a square.

5.6 Error Detection and Correction

Error occurs within a computer system, e.g. in system memory, and in the communication between systems, e.g. in the transmission of messages.

Noise is unwanted information that can cause these errors. It comes in various forms, but is always present and is one of the limiting factors in computer systems. It can occur from stuff such as thermal noise, noise of electronic components, noise of transmission etc.

5.6.1 Detecting single and isolated errors

These are considered to occur at random, usually due to noise. We could send the message three times and take a vote, but this is very expensive. However, if the probability of an error in the channel is low, the probability of two errors close together is also lower. Thus, we can add a "parity" bit to the message every so often, which "summarises" a property of the message that we can check to determine whether a message has been altered. It is much cheaper and adequate in many situations.

5.6.2 Parity

Parity adds an extra bit. There are two types of parity system:

- Even parity system - the value of the extra bit is chosen to make the total of logic 1 an even number
- Odd parity system - Make the total number of logic 1 s odd

And it is very popular in hardware. In fact, hardware implementations generally work better than software, despite being a relatively simple digital logic circuit.

To detect error using parity, consider our transmitter encoding a 7 bit message, then, we compute parity P and append parity to message, and send modified message across channel. Now that we receive the message, an 8 bit is received including parity is decoded. We compute parity Q from data bits of received message and compare our parity Q to parity P in message and flag *ERROR* if they are not consistent. However, if we have say, an odd or an even number of errors depending on the parity, the system does not work very well. These are called Burst Errors.

5.6.3 Burst Error

When the parity is the same but it is still not the same message that was sent, we can create burst error checkers as well. One way to do so is using checksum. For example, we send the word "MESSAGE", which is 1001101, 1100101, 1110011, 1110011, 1100001, 1100111, 1100101. So we calculate the bit-column parity "checksum" values, e.g. using even parity to get

$$1001011 = \text{ASCII "K"}$$

Then we send "MessageK" instead of "Message". This detects all burst errors < 14 bits. However, it does fail if an even number of errors occur in a bit-column. What we can do instead, is not just also calculate the bit-column parity, but also calculate the bit-row parity. This is called the ECC. This way, we can identify if a column is wrong when received. Furthermore, we can also detect if a row is wrong, and as such, we can detect which character is exactly wrong and correct it.

5.7 Common Memory Components

5.7.1 Hard Disks

Record data by magnetising a thin film of ferromagnetic material on a disk. As well as its operations, the general structure of a hard disk can give us insight into performance issues. In terms of data organisation, one track contains many sectors, which each sector being separated by an inter-sector gap. Sector contains preamble to allow head to be synchronised before read/write, data and ECC. The performance of hard disks isn't that great.

5.7.2 Optical Disks

Originally designed to hold music, spiral is 3.5 miles long if unwound and data encoded as "pits" and "lands". Errors in an optical disk are usually local because of scratches, if they're less than 2 mm, it can be corrected. The performance is very good, seek times averages are typically 80 ms.

6 I/O Systems

6.1 Memory Mapped I/O

The same address bus is used to address both memory and I/O devices. Memory, including register, on I/O devices are mapped to address values. An I/O device can then operate as designed on the data given. When a CPU

accesses a memory address, that memory address may be in physical memory (typically RAM) or be associated with memory of an I/O device. The advantages of a memory-mapped I/O are following:

- Simpler than many alternatives (compared to port I/O where a dedicated class of instructions are set)
- CPU requires less internal logic as a result, making it cheaper
- All addressing modes supported by a CPU are available to I/O

And the disadvantages:

- Portions of memory address space must be reserved
- Less of a concern as 64 bit processors, hence address spaces, have come to market
- Still relevant where 16 bit and sometimes 32 bit processors are used e.g. embedded and legacy systems

6.2 Synchronising with I/O devices - Polling

Most I/O devices are much slower than the CPU. Several issues to consider:

- Read - is there data to be read from the device?
- Write - is the device ready to accept data?

Consider a simple printer that takes a finite amount of time to print a character. It is much faster than processor instruction time

6.2.1 Polling and Busy-wait Polling

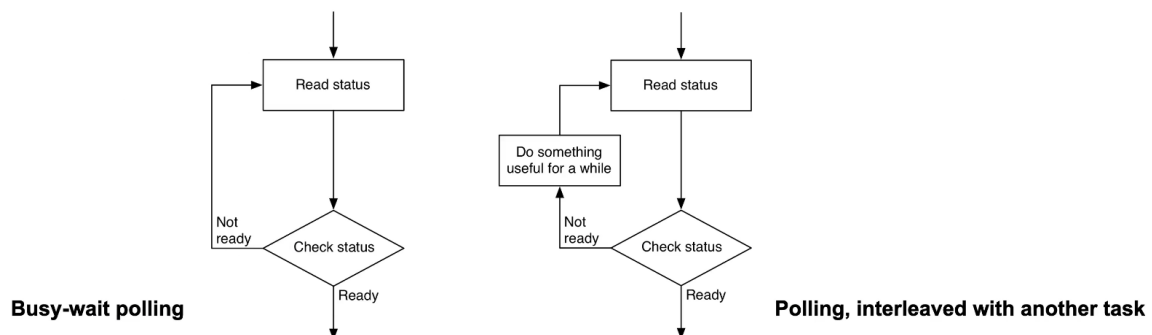


Figure 35: Polling

6.2.2 Advantages

- Simple software - a looping construct paired with some known checks
- Simple hardware - support for notion of "ready" is all that is required

6.2.3 Disadvantaging

- Busy-wait polling wastes CPU time and consumes power
- Polling when interleaved with other tasks can lead to significantly delayed response to device

6.3 Synchronising with I/O devices - Handshaking

6.3.1 Unsynchronised

Computer system transfers the data to the output device

6.3.2 Open-ended handshaking

The computer system provides data and then asserts its validity to the IO device. It is then the device's responsibility to use the data as it fits

6.3.3 Closed-loop handshaking

The recipient asserts readiness to receive. The computer system gives and validates the data. The point is that it establishes a period of time that is known to both parties.

6.3.4 Timing Diagram

The computer system responds to the printer being ready by placing a new character on the data bus and signalling DATA_VALID. We would write code to get the CPU to do this, or alternatively, we could use specialised hardware, which often requires fewer CPU instructions.

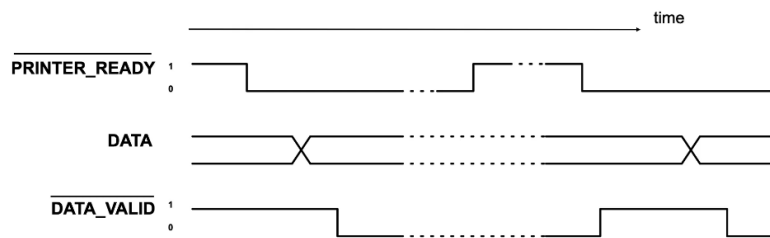


Figure 36: Timing Diagram

6.4 Synchronising with I/O devices - Interrupts

Consider the 6502 processor, designed in 1975, used in the BBC micro and others. IRQ (Interrupt requested) and NMI (Non-maskable interrupt) inputs. Code can disable response to IRQ, as it is only a request, but NMI cannot be disabled or "masked out".

6.4.1 Effect of an interrupt input

CPU normally executes instructions sequentially, unless a jump or branch is made. An interrupt input can force CPU to jump to a service routine. It can therefore make it appear that the CPU is performing two or more tasks simultaneously"

6.4.2 Interrupt Handling Sequence

1. External device signals interrupt

INTERRUPT RESPONSE

1. CPU completes current instruction
2. Push PC onto Stack
3. Push status register(s) onto stack
4. Load PC with address of Interrupt Handler

RETURN FROM INTERRUPT

1. Pop PC from Stack
2. Pop Status Registers from Stack
3. Load PC with popped return address

6.4.3 Nested Interrupts

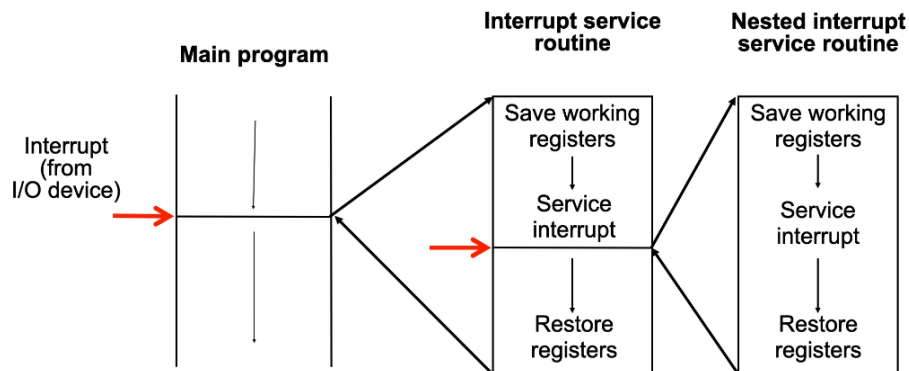


Figure 37: Nested Interrupt

6.4.4 Interrupts for I/O Examples

Switches can be connected to IRQ, if using switches we can OR them all together to form IRQ and then check the individual switch states within the service routine

A hard drive can generate an interrupt when data, requested some time earlier, is ready to read

A timer can generate an interrupt every 100ms and the service routine can then read a sensor input

A printer can generate an interrupt when it is ready to receive the next character to print, as we will see

6.4.5 Advantages of Interrupts

- Fast response
- No wasted CPU time / battery power

6.4.6 Disadvantages of Interrupts

- All data transfers still controlled by CPU
- More complex hardware and software

6.5 Direct Memory Access (DMA)

The CPU is a bottleneck for the I/O. Programmed I/O techniques, as in all the examples, we've seen so far, are slowed down by CPU.

DMA is used where large amounts of data must be transferred at high speed. Control of the system buses is surrendered by the CPU to a DMA Controller (DMAC).

- The DMAC is a dedicated device that controls the three system buses during a data transfer
- The DMAC is optimised for one operation i.e. data transfer
- The CPU is more general purpose, it both transfers data and is a processor of information.

DMA-based I/O can be more than 10 times faster than CPU-driven I/O

6.5.1 DMA Operation

1. DMA transfer requested by I/O
2. DMAC passes request to CPU
3. CPU initialises DMAC and then requires the following information:

- (a) Input or Output
 - (b) Start Address → DMAC Address Reg.
 - (c) Number of words transfer → Count Reg.
 - (d) CPU enables DMAC
4. DMAC requests use of system buses
 5. CPU responds with DMA Ack when it's ready to surrender buses

6.5.2 Modes of Operation

- Cycle stealing - the DMAC uses the system buses when they are not being used by the CPU - usually by "grabbing" available memory access cycles not used by the CPU
- Burst mode - DMAC requires system buses for extended transfer of large amounts of data at high speed and "locks" the slower CPU out of using the system buses for a fixed time or until the transfer is complete or the CPU receives an interrupt from device of greater priority

7 Processor Architecture

7.1 Microprocessor Organisation

7.1.1 Mainstore and Instructions

PATP – architecture (programmer's model)

CS132 T6L1H1

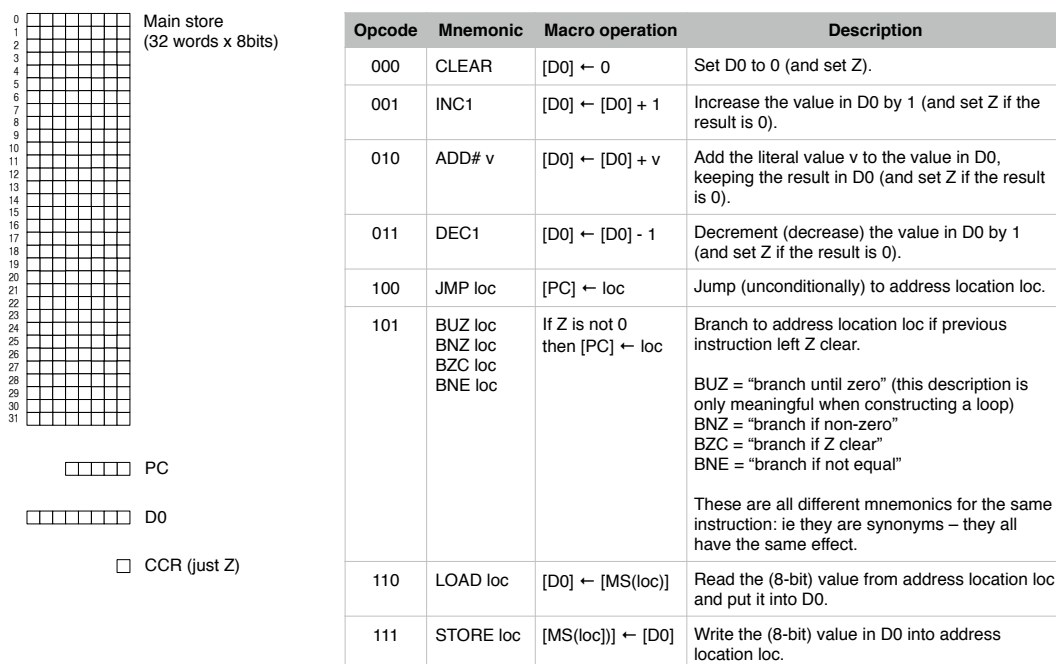


Figure 38: Handout 1

7.1.2 Register

The register is shown as

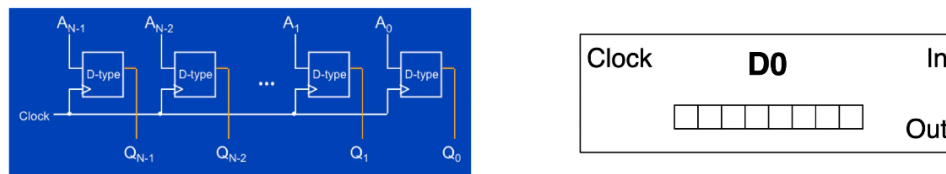


Figure 39: Register

And will be used for $D0$, the only data register in our processor

7.1.3 Arithmetic Logic Unit

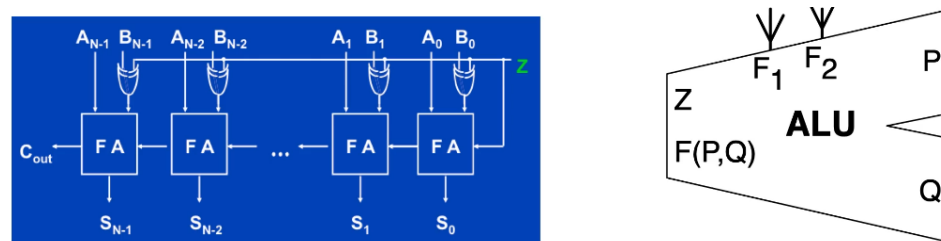


Figure 40: Arithmetic Logic Unit

The arithmetic logic unit ALU is shown above. The inputs are P and Q . Sometimes, we want the condition register Z . The F_1 and F_2 are function select and show us what functions we will be computing given P and Q .

7.1.4 Bus

We will have a common bus for different subsystems to communicate with each other. Three way logic will enable this. This will be done using the enable signals that will be under control of the control unit.

7.1.5 Control Unit

Control unit has been largely a black box until now. However, this is the part that regulates the Fetch Decode Execute system and requires to approach the clock speed of the microprocessor. Furthermore, it requires to know the state of the microprocessor and thus requires to know the Z flag. It also takes opcode as input and asserts the set of signals e.g. R/W.

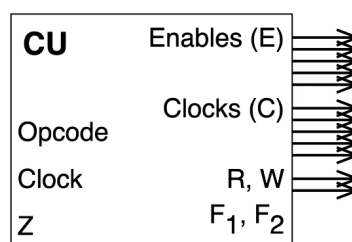


Figure 41: Control Unit

7.1.6 Internal Organisation

PATP – internal organisation

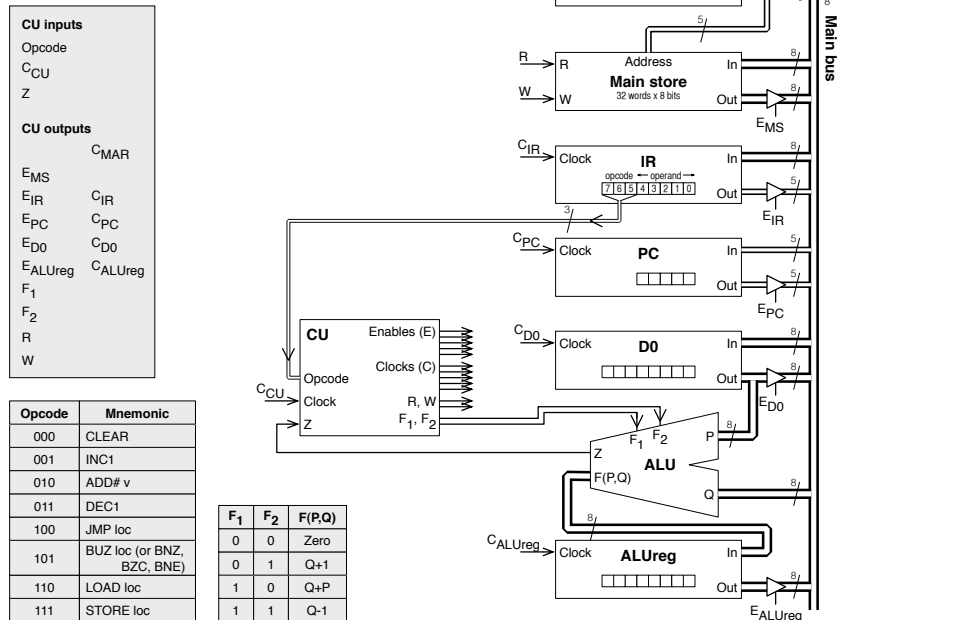


Figure 42: Handout 3

In handout 3, all these parts come in together to form the PATP, our processor. One thing to note, is that the enable lines, Read, Write etc. are actually connected to the CU, but these connections are not shown because it gets messy quickly otherwise.

7.1.7 Instructions and Control Signals

PATP – operations for example 0

CS132 T6L1H4

Macro Step	Mnemonic	Macro Operation	Micro Step	Micro Operations	Control Step	Control Actions	Comments
1	CLEAR	$[D0] \leftarrow 0$	1	$[ALU(Q)] \leftarrow [D0]$	1	E_{D0}	(Not strictly required)
			2	$[ALU(F)] \leftarrow 00$ ("Zero")	1	$F_1=0$ & $F_2=0$	
			3	$[ALUreg] \leftarrow [ALU]$	1	C_{ALUR}	
			4	$[D0] \leftarrow [ALUreg]$	2	E_{ALUR}, C_{D0}	
2	INC1	$[D0] \leftarrow [D0] + 1$	5	$[ALU(Q)] \leftarrow [D0]$	3	E_{D0}	
			6	$[ALU(F)] \leftarrow 01$ ("Q+1")	3	$F_1=0$ & $F_2=1$	
			7	$[ALUreg] \leftarrow [ALU]$	3	C_{ALUR}	
			8	$[D0] \leftarrow [ALUreg]$	4	E_{ALUR}, C_{D0}	
3	INC1	$[D0] \leftarrow [D0] + 1$	9	$[ALU(Q)] \leftarrow [D0]$	5	E_{D0}	
			10	$[ALU(F)] \leftarrow 01$ ("Q+1")	5	$F_1=0$ & $F_2=1$	
			11	$[ALUreg] \leftarrow [ALU]$	5	C_{ALUR}	
			12	$[D0] \leftarrow [ALUreg]$	6	E_{ALUR}, C_{D0}	
4	DEC1	$[D0] \leftarrow [D0] - 1$	13	$[ALU(Q)] \leftarrow [D0]$	7	E_{D0}	
			14	$[ALU(F)] \leftarrow 11$ ("Q-1")	7	$F_1=1$ & $F_2=1$	
			15	$[ALUreg] \leftarrow [ALU]$	7	C_{ALUR}	
			16	$[D0] \leftarrow [ALUreg]$	8	E_{ALUR}, C_{D0}	

Figure 43: Handout 4

For example, if we were to have 42 in binary in $D0$, then it would first be outputted into the bus and into the ALU, then have the correspond F for $Q + 1$, and then would be stored in $ALUreg$. Then, it would be outputted into the bus and back into $D0$.

PATP – operations for instructions and fetch

CS132 T6H5

Opcode	Mnemonic	Macro operation	Micro step	Micro operations	Control step	Control actions
	(fetch step A)	$[IR] \leftarrow [MS(PC)]$	1	$[MAR] \leftarrow [PC]$	1	E_{PC}, C_{MAR}
			2	$[IR(0:7)] \leftarrow [MS(MAR)]$	2	R, E_{MS}, C_{IR}
	(fetch step B)	$[PC] \leftarrow [PC] + 1$	3	$[ALU(Q)] \leftarrow [PC]$	3	E_{PC}
			4	$[ALU(F)] \leftarrow 01 ("Q+1")$	3	$F_1=0 \text{ \& } F_2=1$
			5	$[ALUreg] \leftarrow [ALU]$	3	C_{ALUR}
			6	$[PC] \leftarrow [ALUreg]$	4	E_{ALUR}, C_{PC}
000	CLEAR	$[D0] \leftarrow 0$	1	$[ALU(F)] \leftarrow 00 ("Zero")$	1	$F_1=0 \text{ \& } F_2=0$
			2	$[ALUreg] \leftarrow [ALU]$	1	C_{ALUR}
			3	$[D0] \leftarrow [ALUreg]$	2	E_{ALUR}, C_{D0}
001	INC1	$[D0] \leftarrow [D0] + 1$	1	$[ALU(Q)] \leftarrow [D0]$	1	E_{D0}
			2	$[ALU(F)] \leftarrow 01 ("Q+1")$	1	$F_1=0 \text{ \& } F_2=1$
			3	$[ALUreg] \leftarrow [ALU]$	1	C_{ALUR}
			4	$[D0] \leftarrow [ALUreg]$	2	E_{ALUR}, C_{D0}
010	ADD# n	$[D0] \leftarrow [D0] + v$	1	$[ALU(P)] \leftarrow [D0]$	1	No control actions required
			2	$[ALU(Q)] \leftarrow [IR(0:4)]$	1	E_{IR}
			3	$[ALU(F)] \leftarrow 10 ("Q+P")$	1	$F_1=1 \text{ \& } F_2=0$
			4	$[ALUreg] \leftarrow [ALU]$	1	C_{ALUR}
			5	$[D0] \leftarrow [ALUreg]$	2	E_{ALUR}, C_{D0}
011	DEC1	$[D0] \leftarrow [D0] - 1$	1	$[ALU(Q)] \leftarrow [D0]$	1	E_{D0}
			2	$[ALU(F)] \leftarrow 11 ("Q-1")$	1	$F_1=1 \text{ \& } F_2=1$
			3	$[ALUreg] \leftarrow [ALU]$	1	C_{ALUR}
			4	$[D0] \leftarrow [ALUreg]$	2	E_{ALUR}, C_{D0}
100	JMP loc	$[PC] \leftarrow loc$	1	$[PC] \leftarrow [IR(0:4)]$	1	E_{IR}, C_{PC}
101	BUZ loc / BNZ loc / BZC loc / BNE loc	If Z is not 0 then $[PC] \leftarrow loc$	1	If Z is not 0 then $[PC] \leftarrow [IR(0:4)]$	1	If Z is not 0 then E_{IR}, C_{PC}
110	LOAD loc	$[D0] \leftarrow [MS(loc)]$	1	$[MAR] \leftarrow [IR(0:4)]$	1	E_{IR}, C_{MAR}
			2	$[D0] \leftarrow [MS(MAR)]$	2	R, E_{MS}, C_{D0}
111	STORE loc	$[MS(loc)] \leftarrow [D0]$	1	$[MAR] \leftarrow [IR(0:4)]$	1	E_{IR}, C_{MAR}
			2	$[MS(MAR)] \leftarrow [D0]$	2	E_{D0}, W

Figure 44: Handout 5

7.2 Macro and Micro Instructions

7.2.1 Fetch Macro Instructions

The fetch cycle for the PATP would be defined as

$[IR] \leftarrow [MS(PC)]$ // Get the byte from memory that PC is pointing to, put it in IR
 $[PC] \leftarrow [PC+1]$ // Increment PC to point to the next byte

For example, for the program that uses *CLEAR*, *ADD#9* and *DEC1*:

PATP – operations for example 1

Initialisation: load example 1 program into memory, set PC to 00000, then start machine execution with the first fetch.

MS address	Mnemonic	Data
0 (=00000)	CLEAR	000 00000
1 (=00001)	ADD# 9	010 01001 (NB I've changed example 1: last time this was ADD# 3 but now it is ADD# 9)
2 (=00010)	DEC1	011 00000

CS138 T6H6

Macro step	Mnemonic	Macro operation	Micro step	Micro operations	MAR IN 0:4	IR IN 0:7, OUT 0:4	PC IN 0:4, OUT 0:4	D0 IN 0:7, OUT 0:7	ALU(P) IN 0:7	ALU(Q) IN 0:7	ALU(F) IN F ₁ , F ₂	ALUreg IN 0:7, OUT 0:7
		(initialisation)					00000					
fetch	(fetch step A)	$[IR] \leftarrow [MS(PC)]$	1	$[MAR] \leftarrow [PC]$	00000							
			2	$[IR(0:7)] \leftarrow [MS(MAR)]$		000 00000						
	(fetch step B)	$[PC] \leftarrow [PC] + 1$	3	$[ALU(Q)] \leftarrow [PC]$... 00000		
			4	$[ALU(F)] \leftarrow 01 ("Q+1")$							01 ("Q+1")	
			5	$[ALUreg] \leftarrow [ALU]$... 00001
			6	$[PC] \leftarrow [ALUreg]$			00001					
1	CLEAR	$[D0] \leftarrow 0$	7	$[ALU(F)] \leftarrow 00 ("Zero")$							00 ("Zero")	
			8	$[ALUreg] \leftarrow [ALU]$								0000 0000
			9	$[D0] \leftarrow [ALUreg]$				0000 0000				
fetch	(fetch step A)	$[IR] \leftarrow [MS(PC)]$	10	$[MAR] \leftarrow [PC]$	00001							
			11	$[IR(0:7)] \leftarrow [MS(MAR)]$		010 01001						
	(fetch step B)	$[PC] \leftarrow [PC] + 1$	12	$[ALU(Q)] \leftarrow [PC]$... 00001		
			13	$[ALU(F)] \leftarrow 01 ("Q+1")$							01 ("Q+1")	
			14	$[ALUreg] \leftarrow [ALU]$... 00010
			15	$[PC] \leftarrow [ALUreg]$			00010					
2	ADD# 9	$[D0] \leftarrow [D0] + v$	16	$[ALU(P)] \leftarrow [D0]$					0000 0000			
			17	$[ALU(Q)] \leftarrow [IR(0:4)]$						(000) 01001		
			18	$[ALU(F)] \leftarrow 10 ("Q+P")$							10 ("Q+P")	
			19	$[ALUreg] \leftarrow [ALU]$								0000 1001
			20	$[D0] \leftarrow [ALUreg]$				0000 1001				
fetch	(fetch step A)	$[IR] \leftarrow [MS(PC)]$	21	$[MAR] \leftarrow [PC]$	00010							
			22	$[IR(0:7)] \leftarrow [MS(MAR)]$		011 00000						
	(fetch step B)	$[PC] \leftarrow [PC] + 1$	23	$[ALU(Q)] \leftarrow [PC]$							01 ("Q+1")	
			24	$[ALU(F)] \leftarrow 01 ("Q+1")$... 00011
			25	$[ALUreg] \leftarrow [ALU]$								
			26	$[PC] \leftarrow [ALUreg]$			00011					
3	DEC1	$[D0] \leftarrow [D0] - 1$	27	$[ALU(Q)] \leftarrow [D0]$						0000 1001		
			28	$[ALU(F)] \leftarrow 11 ("Q-1")$							11 ("Q-1")	
			29	$[ALUreg] \leftarrow [ALU]$								0000 1000
			30	$[D0] \leftarrow [ALUreg]$				0000 1000				

Figure 45: Handout 6

7.2.2 Assumptions

- Enable signals are level triggered
- Clock signals are falling-edge triggered
- An output can be enabled onto the main bus and then clocked in elsewhere in one step

7.3 Control Unit Design

7.3.1 Control Unit Tasks

Our goal is to generate fetch control sequence. Observe opcode input and choose the right control signal, that is to decode. And depending upon opcode input, generate sequence of control signals, that is execute. The are two ways to design this:

- Hardwired - The CU is combinatorial logic circuit, transforming its input signals into a set of output signals
- Microprogrammed - each machine instruction is turned into a sequence of primitive microinstructions, which form a microprogram, stored in ROM called microprogram memory

7.3.2 Hardwired CU

WE employ the use of a sequencer, that is, we build a logic circuit that produces these outputs if we connect an input to it that repeatedly clocks up and down. In practice, we connect the clock input of the sequencer to CU clock input (CCU)

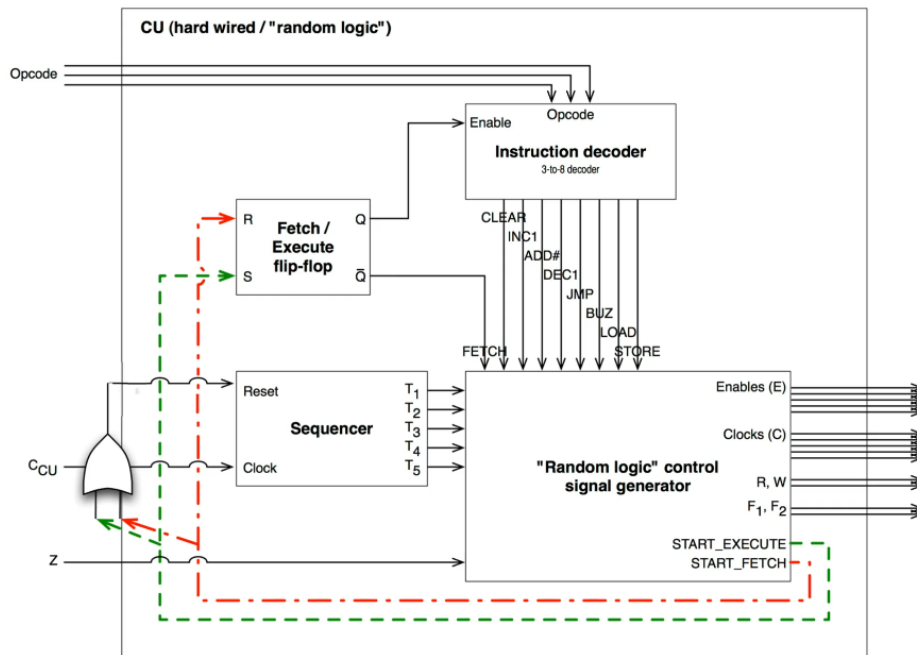


Figure 46: Hardwired CPU design

The macro instructions go to the random logic control signal generator that determines the outputs. For example, if it recognises if it is the increment operation, we know what enables and clocks require to be on each rounds. The round number is regulated by the sequencer. We will also require a mechanism to move onto the next round, and this is done by the fetch execute flip flop.

7.3.3 Advantages of Hardwired CU

- It is fast

7.3.4 Disadvantages of Hardwired CU

- Complex hardware makes it difficult to design and test - too many interconnections to get it right
- Inflexible as it is difficult to change the design if a new instruction must be added
- Long design time

7.3.5 Microprogrammed CU

This employs the use of a microprogram memory and requires to store the required control action in memory. What we do is we store the round of control signals in each round and look them up as we advance using the microprogram counter. The terminology is as follows:

- Microprogram routine - describes how to generate the CU outputs for one macroinstruction
- Microaddress - A location within microprogram memory
- MicroPC - The CU's internal program counter, i.e. points to microaddress
- MicroIR - The CU's internal microinstruction register is used to hold the current microinstruction
- Microinstruction - Holds the CU output values and other fields to control the microprogram flow

It is also the dominant technique for implementing CUs for CISC processors

7.3.6 Advantages of Microprogrammed CU

- Ease of design and implementation
- Flexibility of design allows families of processors to be built
- Simple hardware compared to hardwired implementation
- Microprogram memory can be reprogrammed for new instructions

7.3.7 Disadvantages of Microprogrammed CU

- Slower than hardwired implementations

7.4 How could we change our PATP?

We could add more memory for more complex applications, we would also need to increase the operand size for things such as branch operations. We could also add more instructions, e.g. subtraction, multiply, divide, and, or operators. This would require a higher opcode size. We could also add more registers and more instructions to these new registers. We could implement floating point, as we are only working with integers. However, the consequences of these improvements is increasing opcode, operand size, increasing complexity etc.

The PATP is turing complete, it can complete a universal turing machine and can evaluate every computable function. It can also perform any calculation that any other programmable computer is capable of, however, there are also caveats. These caveats are ignoring time that the machine takes to execute a program, pretending the machine has unlimited storage and addressing and ignoring effort to write the program.

7.5 RISC and CISC

This paper found out that data movement, program control and arithmetic dominated the instructions that people typically execute and they weren't being optimised at the time. Fairclough then did work that identified that certain groups of instructions were used more than others. For example, Intel has microprogrammed complex executions and hardwired simple ones. It is recommended that more reading is done outside these notes for this particular part of the module.

A A detailed explanation of the recursive relationship

Consider the first few digits of base 2 numbers:

$$\dots, 256, 128, 64, 32, 16, 8, 4, 2, 1$$

Writing each one in terms of 16 where possible:

$$\dots, 1 \times 16^2, 8 \times 16, 4 \times 16, 2 \times 16, 1 \times 16, 8 \times 16^0, 4 \times 16^0, 2 \times 16^0, 1 \times 16^0$$

We will now categories each one depending on how many 16s they have

$$\dots, 1 \times 16^2 \mid 8 \times 16, 4 \times 16, 2 \times 16, 1 \times 16 \mid 8 \times 16^0, 4 \times 16^0, 2 \times 16^0, 1 \times 16^0$$

Notice that, we can only obtain a maximum of number 15 if $8 + 4 + 2 + 1$. This means that for each category or split, we can only achieve a maximum of 15 as coefficient. If we were to factorised a 16 in the first split, we would get the same as split on the right. We can repeat this for all 16^n . The same logic applies to binary to base 8.