CO112 - Hardware

Prelude

The content discussed here is part of CO112 - Hardware (Computing MEng); taught by Bernhard Kainz, and Bjoern Schuller, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake). This should be used in conjunction with the notes, and lecture slides. This course starts off fairly slow, especially if you have an idea of how logic gates work, and therefore the first parts won't be covered in much detail.

Lecture 1

This section will be covered in less detail, as we've gone through the majority of this in much greater depth during logic. However, we will need to change the notation we use in this course from the one used in logic, from using $\wedge$ to $\cdot$, $\vee$ to $+$, and from $\neg$ to $'$.

A	B	$A \cdot B$ (AND)	$A + B$ (OR)	A' (NOT)
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

The same distributivity laws apply, just like in **CO140**, as well as the simplification laws. In general, the laws should be the same as propositional logic, with the notation being slightly changed. Use 1 for $\top$, and 0 for $\bot$. We will also be using de Morgan's theorem on any number of variables (this can be proven by induction), such that $(V_1 + V_2 + V3 + ... + V_n)' \equiv V_1' \cdot V_2' \cdot V_3' \cdot ... \cdot V_n'$, and the same the other way around. This can be very useful later on, as we will often use NAND / NOR gates to reduce silicon area.

Lecture 2

The three operators covered in the first lecture can be represented by three logic gates; AND, OR, and NOT. The inverter (NOT), is represented by the circle at the end of the triangle. We can also create operations such as NAND, and NOR. Any of the first three gates can be built with just NAND gates, or just NOR gates. Let us represent A NAND B, with $A \uparrow B$.

- A' — $A \uparrow A$
- $A \cdot B$ — $(A \uparrow B)'$, $(A \uparrow B) \uparrow (A \uparrow B)$
- $A + B$ — $(A' \cdot B')'$ (de Morgan's), $A' \uparrow B'$, $(A \uparrow A) \uparrow (B \uparrow B)$

We also need to introduce two new gates, which are commonly used in digital logic; XOR, and XNOR. Roughly, you can use the same rules for $\neg(A \leftrightarrow B)$, and $A \leftrightarrow B$ respectively. XOR is commonly represented by $A \oplus B$ (which is much shorter than $A \cdot B' + A' \cdot B$), and XNOR represented by $(A \oplus B)'$, instead of $A' \cdot B' + A \cdot B$. It has the following truth table;

A	B	$A \oplus B$	$(A \oplus B)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

In general, with n inputs, we can have 2^n unique gates. With this information, we can build a single control block. For example, let there be a block with an input A, a control C, and output R. It follows that if C is 0, then the output is 0, and A, if C is 1. Hence $R = A \cdot C$.

Now, if we had something with two inputs, A, and B, and a control C, we could use C to determine whether $R = A$, or $R = B$. This is done with the boolean equation $R = A \cdot C' + B \cdot C$. This is a **multiplexer**, which will be used very often in circuit design for other components.

Lecture 3

Consider an more complex circuit, where we have 3 outputs; R_1, R_2, and R_3, and 3 inputs; A, B, and C, where $R_n = f_n(A, B, C)$. For now, we ignore sequential circuits, where the output can be on either side of the equation. The first step of creating a circuit is to construct a truth table as a starting point. We also need to define a few terms;

- **minterm** - a boolean product where each input, or its complement, appears exactly once

  hence $A \cdot B' \cdot C$ is a minterm, but $A \cdot B$ isn't.

  also known as sum-of-products, or disjunction-of-conjunctions
- **maxterm** - a boolean sum where each input, or its complement, appears exactly once

  hence $A + B' + C$ is a maxterm, but $A + B$ isn't.

  also known as product-of-sums, or conjunction-of-disjunctions

For example, let's work on the following truth table;

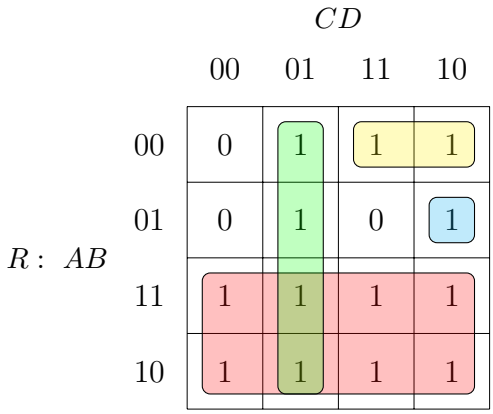
A	B	C	R	
0	0	0	0	maxterm $A + B + C$
0	0	1	0	maxterm $A + B + C'$
0	1	0	0	maxterm $A + B' + C$
0	1	1	1	minterm $A' \cdot B \cdot C$
1	0	0	0	maxterm $A' + B + C$
1	0	1	1	minterm $A \cdot B' \cdot C$
1	1	0	1	minterm $A \cdot B \cdot C'$
1	1	1	1	minterm $A \cdot B \cdot C$

Now, we can evaluate R in two ways;

- $R = (A' \cdot B \cdot C) + (A \cdot B' \cdot C) + (A \cdot B \cdot C') + (A \cdot B \cdot C)$ sum of minterms
- $R = (A + B + C) \cdot (A + B + C') \cdot (A + B' + C) \cdot (A' + B + C)$ product of maxterms

Knowing this, we can convert any truth table into a **Karnaugh map**

A	B	C	D	R	A	B	C	D	R
0	0	0	0	0	1	0	0	0	1
0	0	0	1	1	1	0	0	1	1
0	0	1	0	1	1	0	1	0	1
0	0	1	1	1	1	0	1	1	1
0	1	0	0	0	1	1	0	0	1
0	1	0	1	1	1	1	0	1	1
0	1	1	0	1	1	1	1	0	1
0	1	1	1	0	1	1	1	1	1



Hence we can use minterms, to get $R = \underbrace{A}_{\text{red}} + \underbrace{C' \cdot D}_{\text{green}} + \underbrace{A' \cdot B' \cdot C}_{\text{yellow}} + \underbrace{A' \cdot B \cdot C \cdot D'}_{\text{blue}}$.

Remember that the regions in the map can wrap around, and that don't cares can count as either 0, or 1.

Lecture 4

A general circuit can be used to generate all possible n-input digital circuits. If we were to have inputs $V_1, V_2, ..., V_n$, and have each one come out as two lines, so V_i would come out with V_i, and its complement V_i'. We can have 2^n n-input AND gates, which correspond to each possible combination (00...00), (00...01), (00...11), all the way to (11...11). This is hard to visualise (see *Notes04 - Description to Circuit.pdf*), but the general idea is that each AND gate corresponds to a possible minterm, which is joined to a 2^n-input OR gate, if it's a 1 in the truth table. This is a **Programmable Array Logic (PAL)** device, and the device can be programmed by sending a current through specific links to connect them to the OR gate.

The general steps for creating a device from a specified design are as follows;

1. Construct a truth table

   translate the natural language description of what the device should do into a truth table.

2. Generate a Karnaugh map

   using the techniques mentioned in the previous lecture, create the map, and find the resulting sum of minterms (or product of maxterms)

3. Minimise the boolean expression

   using the resulting sum or product, we can then use boolean algebra to simplify the expression

4. Design the circuit

   using the minimised boolean expression, we can finally construct a circuit

5. Minimise the circuit

   this is different from minimising the equation, as we're now trying to minimise the silicon area used

   in general, this is to replace ANDs, and ORs, with NANDs, and NORs

   a method of doing this is to replace expressions such as $(A \cdot B)$ with $(A' + B')'$, by using de Morgan's law

6. Test the circuit

   the usual process is to simulate the circuit, to validate it against the original specifications

   finding bugs before production is important (and expensive, if not spotted); see the *Floating Point Division Bug* in *Intel Pentium P5*

Lecture 5

While boolean algebra is a good abstraction for the behaviour of logic gates, it has some subtle differences, which can be problematic, and cause bugs. Practically, voltages aren't exact values, and therefore thresholds have to be arbitrarily determined - leading to more issues (such as what happens when the voltage is between the lower, and upper threshold). The main difference is that boolean algebra doesn't consider time delays, which exist despite electrons moving at light-speed. This failure to synchronise events is a common error in hardware design, and therefore we will require a more accurate physical model (note that all models are simply approximations, but we should choose one of a sensible degree of accuracy).

To define the physical representation of a logic gate, we'll need to reuse some content from A Level Physics.

- components

  resistor

  capacitor

  transistor

- equations

  $V = I \cdot R$ Ohm's law

  $I = C\frac{\mathrm{d}V}{\mathrm{d}t}$ Capacitance

Pure silicon is an extremely good insulator, however if we were to infuse (**dope**) it with impurities to give it surplus electrons, it would then be able to conduct electricity; this type of semiconductor is known as **n-type**. On the other hand, if we were to infuse it with positive charge carriers (which would just be holes, with missing electrons), we'd have **p-type**. A transistor is made of three adjacent pieces of these semiconductors; and can either be **n-p-n**, or **p-n-p**. While Ohm's law is a simple mathematical method for resistors, and it's possible to derive a set of equations for more complex devices, we're engineers. We will consider the transistor as a switch (as a set of rules, called a procedural model).

Consider the transistor as a switch with three terminals; a source S, a drain D, and a gate G. There is no connection between G, and S, nor is there a connection between G, and D. If the voltage between G, and S (let it be V_{GS}) ≤ 0.5v, there is no connection between S, and D. On the other hand, if $V_{GS} \geq 1.7$v, then S is connected to D, and therefore current can flow through. In an ideal world, when the switch is closed, it has 0 resistance, and when it is open, the resistance is ∞ (this isn't correct, for reasons that will be discussed later).

In general, if there is a high resistance (the circuit is broken), then the output is high (since we have (almost) no current flowing, $I \approx 0$, therefore the P.D. across the resistor, $V_{\mathrm{R}} \approx 0$), and $V_{\mathrm{out}} \approx 5$. However, if it is connected, we will consider it to have a very low resistance, hence the larger P.D. would be across the resistor, thus having a lower V_{out}.

The physical representation of logic gates explains why NAND, and NOR gates are cheaper, as they each only require two transistors (in series, or in parallel, respectively).

There are two main reasons for why there are time delays in a circuit. The first is the state change of the transistor; the electrons will still take time to move through it. Second is the representation of the transistor; we need to consider the transistor as more of a variable resistor. By reducing the size of the capacitor, we have a lower capacitance, and hence a faster charge. We can calculate a formula for voltage over time; (note that $K = -\ln(5)$, since $V(0) = 0$)

$$\begin{aligned} 5 - V &= I \cdot R \\ I &= C\frac{\mathrm{d}V}{\mathrm{d}t} \\ 5 - V &= RC\frac{\mathrm{d}V}{\mathrm{d}t} \end{aligned}$$

$$\frac{5-V}{\mathrm{d}V} = \frac{RC}{\mathrm{d}t}$$
$$\int \frac{1}{5-V}\,\mathrm{d}V = \int \frac{1}{RC}\,\mathrm{d}t$$
$$-\ln(5-V) = \frac{t}{RC} + K$$
$$-\ln(5-V) = \frac{t}{RC} - \ln(5)$$
$$\ln(5-V) = \ln(5) - \frac{t}{RC}$$
$$5-V = e^{\ln(5)-\frac{t}{RC}}$$
$$5-V = e^{\ln(5)} \cdot e^{-\frac{t}{RC}}$$
$$5-V = 5e^{-\frac{t}{RC}}$$
$$V = 5 - 5e^{-\frac{t}{RC}}$$
$$V = 5(1 - e^{-\frac{t}{RC}})$$

Lecture 6

Consider the case where we have a NAND gate, with inputs A, and R (which is coming from the output R). When $A = 1$, we have $R = (A \cdot R)' = (1 \cdot R)' = R'$; which is clearly inconsistent. This is where the logic breaks down - however in real life, there's nothing stopping us from doing this. Continuing on from the previous lecture, we have two models;

- **Switch and Delay**

  only differs from boolean algebra due to the inclusion of a time delay between the change in the input, and the change in the output

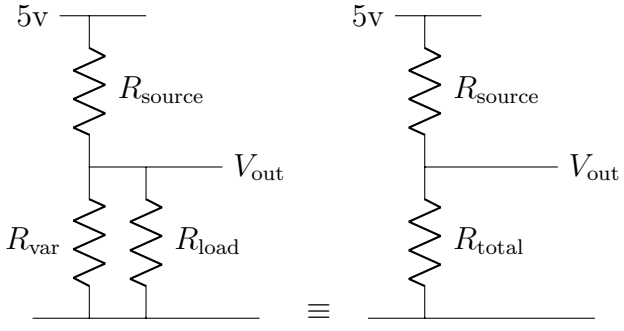
- **Resistance and Capacitance**

  this is a more accurate representation of real behaviour, and during the time delay period, it's no longer a valid boolean signal

  this is a analogue model, and not digital (digital electronics don't actually exist, due to noise, and therefore we use safety margins) - for example, a voltage above 1.7v would be logic 1, and a voltage between 0.5v would be logic 0

Despite defining the noise margins above, we should design our circuits so that they operate far from the thresholds, such that we can have boolean 1 around 3.5v, and boolean 0 at around 0.3v. Since we're considering the transistor as a variable resistor, it acts like a potential divider.

Remembering from physics, we can take $\frac{1}{R_{\text{total}}} = \frac{1}{R_{\text{var}}} + \frac{1}{R_{\text{load}}}$, and therefore calculate $V_{\text{out}} = \frac{5R_{\text{total}}}{R_{\text{source}}+R_{\text{total}}}$. Here, R_{load} is the combined resistance of all gates the transistor is connected to.



As we know the inverse law for parallel resistors, we can say the load resistance R_{load} becomes $\frac{1}{n}$ of a single gate, when there are n identical gates connected to a single gate output. Not only does this **fan-out** cause issues with the thresholds, we also need to consider the time delay. The time delay is

directly proportional to the size of the load capacitor (trivial to derive from the equations listed), and capacitors in parallel add up, hence a larger fan-out would have a larger time delay. For the time being, we will stick to the original **switch-and-delay** model.

Going back to the problem at hand, with the NAND gate, we would have an extremely high frequency oscillation. If we construct a table for the states, we can determine which states are stable. This effect can be harnessed to make memory.

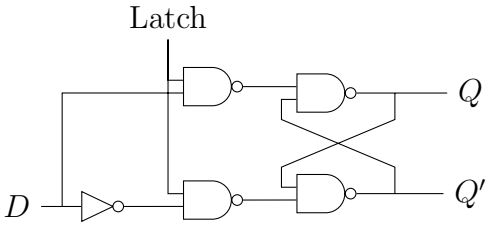
Lecture 7

I will use the following notation for stating the values of inputs; $I_1...I_n(v_1, ..., v_n)$ means that $I_i = v_i \ \forall i \in [1, n]$; so $SR(1, 0)$ would mean $S = 1$, and $R = 0$.

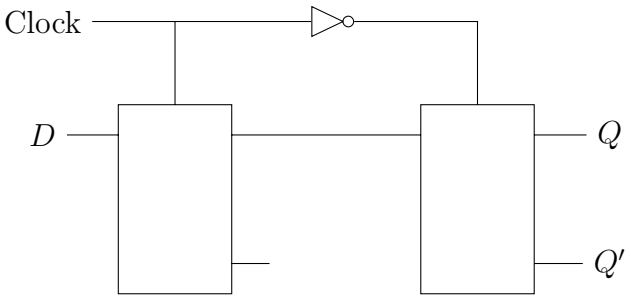
Looking at the states for the **R-S flip flop**, we can get the following states;

S	R	P_t	Q_t	P_{t+1}	Q_{t+1}
0	0	×	×	1	1
0	1	×	×	1	0
1	0	×	×	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	0	0

You might be able to notice that when $SR(1, 1)$, and $P = Q'$, we have a stable state. In general, we have $P = Q'$, as long as we reset the flip-flop, and avoid $RS(0, 0)$. Looking at the $RS(1, 1)$ state, where it's bi-stable, it can theoretically oscillate infinitely, but in practice the gates will likely have different time delays, and will therefore fall into a stable state. As we have this uncertainty, we send a reset signal $RS(0, 1)$, which sets the Q to 1, and after that point we have predictable behaviour (given we avoid $RS(0, 0)$). This way, we will be able to get $RS(0, 1) \rightarrow Q = 1$, and $RS(1, 0) \rightarrow Q = 0$. However, this mechanism isn't convenient for practical memory circuits, so we adapt it with a latch. This way, if the latch is engaged ($L = 1$), we set $Q = D$.

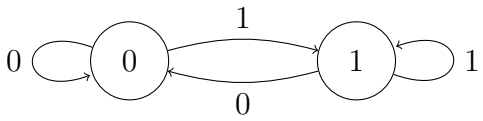


However; there would be a time delay, as the NOT gate would introduce a small delay. We can get around this by using a **Master-Slave Flip-Flop**, which combines two D Flip-Flops, thus allowing for Q to only be set on the **falling edge** of the clock signal.

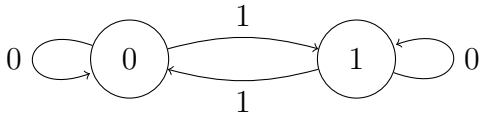


While we don't have to memorise the circuit diagrams of all the flip-flops for the exam, it's important to remember the state diagrams;

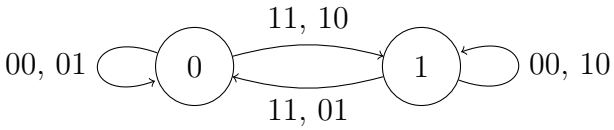
- D-Type Flip-Flop



- T-Type Flip-Flop



- J-K Flip-Flop



this is very useful, as it is able to store memory when you have $JK(0,0)$, since it doesn't change state on that input

Lecture 8

The content covered in this lecture, is done in Coursework 2, and as such, I will be skipping over a lot of things. In general, if we have k states, we will need to use $n = \lceil \log_2(k) \rceil$ D-type flip-flops, all connected to the same clock. A general synchronous sequential system would consist of a block of state sequencing logic, which takes in a set of m inputs, as well as the current states of the flip-flops, and has n outputs. The system would also need a block of output logic, which decodes the states into the appropriate outputs. In general, we can say that $Q_i = D_i(I_1, ..., I_m, Q_{1_{\text{old}}}, ..., Q_{n_{\text{old}}})\ \forall i \in [1..n]$, where Q_i the updated state of the i^{th} flip-flop.

The J-K flip-flop is a very simple example of such a synchronous circuit, which is detailed above.

In general, if we have "don't cares", we will consider them a logic 1 if they are inside any regions inside a Karnaugh map, and 0 otherwise. While components inside the Karnaugh map don't know about each other, we can reuse components when we implement our circuit in order to save silicon space.

Lecture 9

The general notation we will be using is $S(t)$, or S_t to represent S at time t. We cover two main types of finite state machines in this course;

- notation

  S = state of D-Q flip-flops

  O = output(s)

  I = input(s)

  f = input / sequencing logic

  g = output / decode logic

- Mealy Machine

  $S(t+1) = f(S(t), I(t))$

  $O(t+1) = g(S(t), I(t))$

- Moore Machine

  $S(t+1) = f(S(t), I(t))$

  $O(t) = g(S(t))$

A synchronous digital circuit avoids time issues by having a bank of D-Q flip-flops (with a common clock), acting as a barrier between the input, and output. However, spikes can still occur in a Mealy machine, as the output is also controlled by the input. In general, the method for designing a synchronous circuit is as follows;

1. determine the number of states
2. determine the state transitions (and draw FSM)
3. choose how the states are represented by the flip-flops
4. express the state sequencing logic, and minimise said logic with Karnaugh maps
5. express the output logic as a function of the states, minimising if possible

There's an example in the notes, and in the lecture, but once again this is covered mostly in Coursework 2, and therefore will be skipped here.

Lecture 10

Once again, the content from this lecture is covered in the coursework. However, it's important to note how to notice problems with the "don't care" states. In the slides, you'll see an example where the FSM gets stuck in a state, when we replace the "don't cares" with 1s or 0s, depending on whether they're in any Karnaugh map region, or not, respectively. To fix this, we find the offending line, and make a small change that will fix the bug, without adding too much additional bulk to the circuit. This is fixing by hacking.

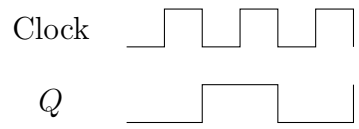
Lecture 11

Registers are fast, small bits of memory that are very accessible. An example of a register is the bank of D-Q flip-flops in the synchronous circuit., which can store an integer in the range $[-2^{n-1}, 2^{n-1} - 1]$, or $[0, 2^n - 1]$, depending on whether a sign is being used. However, we frequently use serial data, and not just parallel, in real life. For example, cables are often serial as it reduces size, and minimises the chance of cable failure. However in practice, data is often processed in parallel in a computer to increase speed, therefore conversion between the two types of data is required. The same circuit can be used for doing parallel-to-serial, and serial-to-parallel, being toggled by a multiplexer

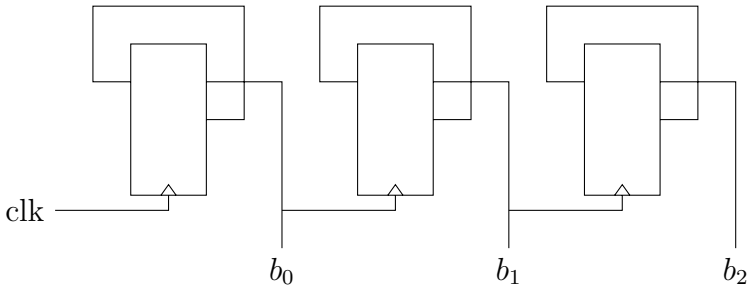
This method for loading serial data can be slow (due to the capacitance of the wires), especially with a larger set of data. Serial communication cannot be done as quickly as operations within the processor, which is why it is normally run on a separate clock, and utilises control lines to indicate when conversions are complete.

This register can also be used for division, and multiplication, by 2. By increasing the number of functions our shift register has, we need to implement a more complex multiplexer. First we have to create a binary to unary converter, which converts an n-bit input, to a 2^n separate lines. This can then easily be combined with n AND gates, and a single OR to select between 2^n inputs.

We can use this to half the frequency of a clock, by connecting Q' into D. This will create the following square wave;



This can then be easily chained to do divisions by powers of 2. However, only being able to divide by powers of 2 isn't exactly useful, and therefore division by other numbers is also important. By using a ripple through counter, we can count to a number, and then send a pulse on that tick, and then clear the memory, thus allowing us to divide by any integer. This counter however, is **not** synchronous. For example, if we wanted to divide by 5, we'd have $O = b_0 \cdot b_1' \cdot b_2$, which would also feed into the top, as a clear signal.



Lecture 12

There's a lot of diagrams in this, so just look at *Notes12 - Multiplexers.pdf* - TikZ is too painful.

While we can build more optimised circuits, calculating the optimal solution is often not feasible, especially as the number of inputs grow (since the possible situations grow exponentially). In order to do this, we will reuse components - this is known as functional design. Often, this will be done with the use of an Enable input, which in reality is just an additional AND gate right at the end.

Something that will be needed often in a computer is transferring data between two registers. This is done in the following steps;

1. select the input register (source)

   on a machine with n registers, this will be done on a n-1 multiplexer; the inputs of the multiplexer are the Qs of the registers (note that in the diagrams, a line with a slash running across it, and a number above, means that it has a certain number of wires - it's a bus)

   the output of the multiplexer goes into D for every register

2. select the output register (destination)

   you'll notice in the step above, it connects to every register - remember that a register only sets Q to D, on the falling / rising edge of a clock

   the clock is connected to the enable line of a demultiplexer (which is also known as a decoder), and the outputs of the demultiplexer are connected to the clock input for each corresponding register

3. the data is then transferred on the clock edge

   this is commonly written as $R_{\text{destination}} \leftarrow R_{\text{source}}$

One of the most important circuits is the comparator; which compares values in two registers (A, and B), and gives an output depending on; $A > B$, $A = B$, or $A < B$. In reality, only two of those are needed, as the missing one is the NOR of the other two, whichever one we decide to drop is arbitrary. See the diagrams in the notes, since I've had enough of drawing them in TikZ for today.

Lecture 13

Addition

First consider the half-adder; we have 3 inputs; A, B, and C_{in}, which is the carry bit. We also have 2 outputs; S, the sum, and C_{out}, which is the bit carried to the next most significant bit. This is trivial to represent as a truth table, so that will be skipped. We can say $C_{\text{out}} = A \cdot B$, and $S = A \oplus B$. We can also make a full-adder with two half-adders, this is done by having the $A_2 = S_1$, and $B_2 = C_{1_{\text{out}}}$.

From this, we can get S_2 (the final sum), to be $S = S_1 \oplus C_{\text{in}} = A \oplus B \oplus C_{\text{in}}$, and C_{out} (the final carry bit), to be $C_{\text{out}} = C_{1_{\text{out}}} + (S_1 \cdot C_{\text{in}}) = (A \cdot B) + C_{\text{in}} \cdot (A \oplus B)$.

An n-bit full adder chains together n full-adders, with a carry of 0 in the least significant bit, and the carries ripple through to the next most significant bit. A single full adder can also be used to sum up serial data; the C_{out} goes in to a D-Q flip-flop, and the Q of the flip-flop is used as the C_{in}. This way, a serial sum is produced.

Subtraction

Since subtraction is a bit less straightforward, as it needs to consider the borrowing, and payback between bits, I will write the truth table for it here, let P represent payback, from the previous bit, D represent the difference, and O represent the borrow, which is passed into the next bit;

A	B	P	D	O
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

As such, we end up with $D = A \oplus B \oplus P$, and $O = A' \cdot (B + P) + B \cdot P$. However, the case where B, and P are both 1 is covered by $B \cdot P$, hence we can change the $B + P$ to a $B \oplus P$, which reduces the number of gates, as we can reuse the $B \oplus P$ from the D circuit. This can then be chained in a similar fashion to the n-bit full adder. However, in practice, it's more common to negate every bit of B, and add it to A with an n-bit full adder, that has a carry of 1. This is a twos complement subtractor.

Multiplication

Multiplication in binary is done in a similar way to how multiplication is manually done in denary. For example, if we were to do $a_1a_0 \times b_1b_0$ (where a_i is a digit, hence $a_1a_0 = 10 \times a_1 + a_0$), we'd do $a_1 \times b_1 \times 10^2 + a_1 \times b_0 \times 10^1 + a_0 \times b_1 \times 10^1 + a_0 \times b_0 \times 10^0$. We do the same in binary, but instead of multiplying by 10^n, we multiply by 2^n, which is n shift to the left. Hence we'd do $(a_1 \cdot b_1) \Leftarrow 2 + (a_1 \cdot b_0) \Leftarrow 1 + (a_0 \cdot b_1) \Leftarrow 1 + (a_0 \cdot b_0) \Leftarrow 0$, where a_i, and b_i are binary digits. This is using ANDs, not multiplying, and the pluses are actual addition with adders. Additional circuits will be needed for signed integers; the sign bit of the output should be the XOR of the sign bits of the two inputs. The scaling of multiplication isn't as elegant as the scaling for addition, and subtraction, as you would need 4 2-bit multipliers, and the shifts would have to be done by powers of 4. This keeps scaling up, and the number of adders required also increases.

Lecture 14

Putting together a manual processor; generally the block diagram takes in a sequence of binary numbers, one sequence for data, and another for instructions. This will result in a binary number. Our design is based on the von Neumann architecture, which divides the processor into arithmetic units and registers, with a shared stream for data, and instructions. Our model will be based on 8 bits.

Our example action is to find the average of two numbers, A, and B, such that $R = \frac{A+B}{2}$. Since we are dealing with 8 bits, $A + B < 256$.

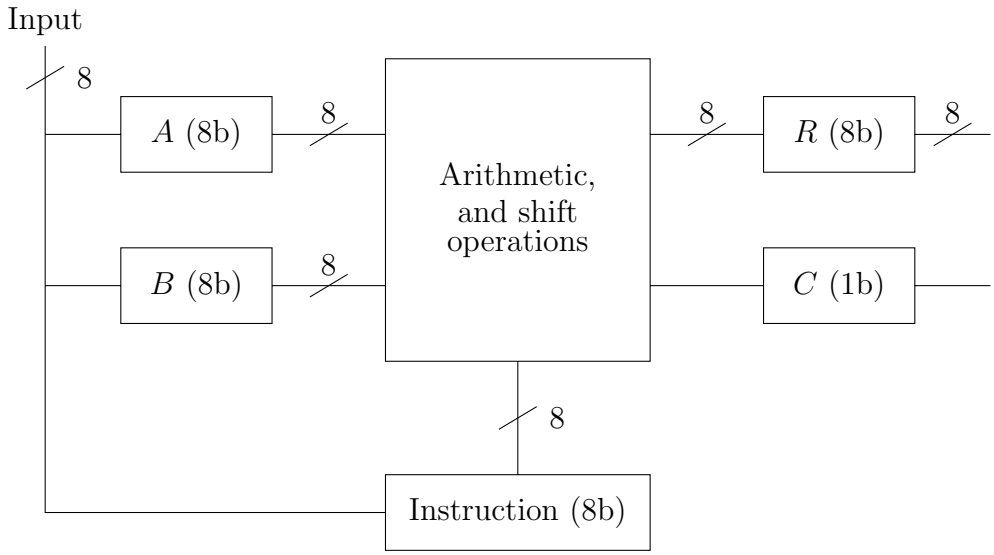
1. The first number is set up in input lines, and stored in register A
2. The same is done for the second number, and stored in register B

3. The arithmetic circuits are set to register $A + B$.
4. The resulting sum is put into A
5. The shift circuits shifts A one bit to the right, which is integer division by 2
6. Result is loaded into R.

In order to do this, we need a number of components;

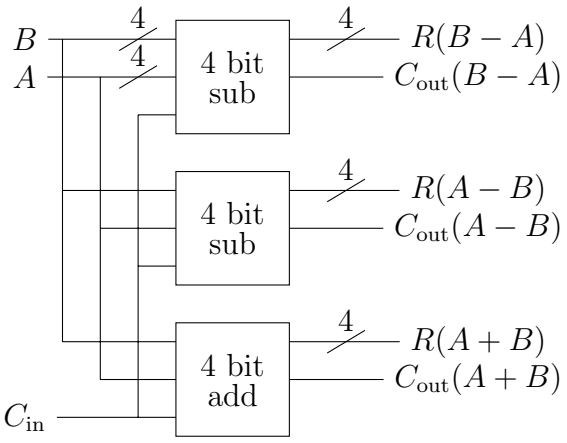
- Registers
  - store data A, and B
  - store result R (Res)
  - one bit for carry C
  - store instruction I (IR)
- Arithmetic circuits
  - 8-bit adder
  - 8-bit shifter

Note in the figure below, that the 8 means it is an 8-bit line.



However, the data path diagram above has no information about when the transfers occur, it it only shows the possible paths it can take. The arithmetic, and shift operations are done by combinational circuits, the function of which is determined by the bits in the instruction register. The processor also cannot operator on the results register. In our design, this is the specification for our ALU (in order to prevent ambiguity between the logical operators, and arithmetic, I will use symbols from **CO140**, and regular symbols for arithmetic);

S_2	S_1	S_0	Function
0	0	0	0
0	0	1	$B - A$
0	1	0	$A - B$
0	1	1	$A + B$
1	0	0	$A \oplus B$
1	0	1	$A \vee B$
1	1	0	$A \wedge B$
1	1	1	-1



For an n-bit ALU, there are $n+1$ multiplexers, as the last one handles the carry bit. The C_{in} bit for the n^{th} multiplexer is the C_{out} for the $(n-1)^{\text{th}}$ multiplexer. The C_{in} for the first multiplexer is 0.

In our 8 function shifter, we will be using the following rules;

	F_2	F_1	F_0	Shift	Carry	Function
A	0	0	0			unchanged (hold)
B	0	0	1	left		rotate left
C	0	1	0	left	0	arithmetic left shift
D	0	1	1	left	C_{in}	left shift with carry
E	1	0	0	right		rotate right
F	1	0	1	right	0	logical right shift
G	1	1	0	right	$I[7]$	arithmetic right shift
H	1	1	1	right	C_{in}	shift right with carry

In reality, for an n-bit shifter, it's just n multiplexers together, with carry cases for the first, and last, multiplexer. Given an n-bit shifter, these are the outputs for the multiplexers;

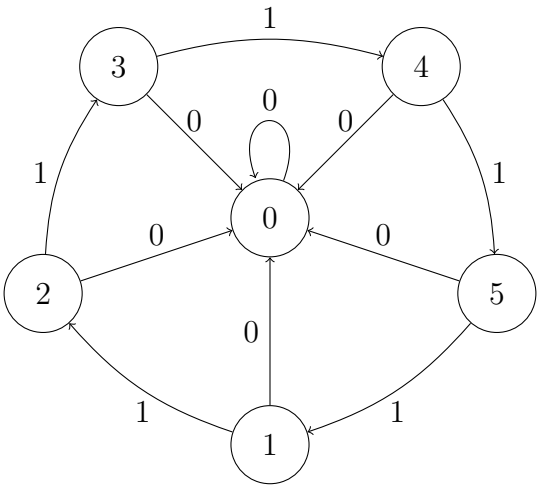
	$O[n-1]$	$O[i](i \neq n-1 \wedge i \neq 0)$	$O[0]$
A	$I[n-1]$	$I[i]$	$I[0]$
B	$I[n-2]$	$I[i-1]$	$I[n-1]$
C	$I[n-2]$	$I[i-1]$	0
D	$I[n-2]$	$I[i-1]$	C_{in}
E	$I[0]$	$I[i+1]$	$I[1]$
F	0	$I[i+1]$	$I[1]$
G	$I[n-1]$	$I[i+1]$	$I[1]$
H	C_{in}	$I[i+1]$	$I[1]$

Lecture 15

This builds on the manual processor we started in the previous lecture. The structure of our instructions that go into the 8-bit instruction register is as follows;

IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
×	ALU/SHIFT INS			×	S/R	S/C	S/A

Wherever there's a × - it's an unused bit. IR4-6 is used to select the function of the shifter or ALU; S/R (in IR2) decides which of the two outputs is loaded into R. S/C (in IR1) decides whether to use the C_{out} of the ALU (when S/$C = 1$), or to use a constant logic 1. Finally, S/A controls the multiplexer which decides whether A is loaded from R (S/$A = 0$), or from the data line (S/$A = 1$). Having unused bits has two benefits; there's redundancy built in, and also allows for the possibility of upgrading the instruction set in the future. We can represent the cycle in the following state transition diagram, where the input 0 means idle, and 1 means operate;



This has the corresponding output logic for each of the clock signals as follows;

State	ClkIR	ClkA	ClkB	ClkR	ClkC
0	0	0	0	0	0
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	1
4	1	0	0	0	0
5	0	0	0	1	1

The states are transitioned in the falling edge of the clock, and then pulsed out in the rising edge, by the NAND gate. In our execution cycle above, we can list the states as follow;

1. load the IR register from the Data In line

   this determines the source for A, and C
2. load in A

   source depends on IR0 - if it is 0, we have $A \leftarrow R$, otherwise we have $R \leftarrow$ Data
3. load in B, and C register

   C source depends on IR1 - if it is 0, we're using logic 1, otherwise we're loading from the C_{out} of the ALU

   $B \leftarrow$ Data
4. load IR again

   it's loaded again here, as now we're doing the actual processing, as we previously loaded the operands
5. load R, and C

   the source for R is loaded from the ALU, or the Shift register, depending on IR2

The structure of our instructions are in 5 bytes;

1. Opcode 1
2. Data 1 (A)
3. Data 2 (B)
4. Opcode 2
5. Unused

   since we have 5 states, not using a 5^{th} byte would mean that a second clock is needed, which would make the operation asynchronous

   we don't need the last byte, as the source for C_{in} is already specified in the other opcodes

These are the cycles for $\frac{A+B}{2}$ in our manual processor

Stage	Step	Operate	Data	Actions
0	0	0	`XXXXXXXX`	Set the processor to idle
1	1	1	`X000XXX1`	Loaded into IR, ALU is set to 0, C_{out} set to 0
1	2	1	`AAAAAAAA`	IR0 = 1, so $A \leftarrow$ Data
1	3	1	`BBBBBBBB`	$B \leftarrow$ Data, and $C \leftarrow 0$
1	4	1	`X011X11X`	ALU is set to $A + B$, $C_{\text{in}} = 0$, as IR1 = 1
1	5	1	`XXXXXXXX`	IR2 = 1, so $R \leftarrow$ ALU
2	1	1	`X011X110`	Loaded into IR
2	2	1	`XXXXXXXX`	IR0 = 0, so $A \leftarrow R$
2	3	1	`XXXXXXXX`	We don't care about B, and $C \leftarrow C_{\text{out}}$
2	4	1	`X110X0XX`	Shift is set to Arithmetic Right Shift
2	5	1	`XXXXXXXX`	IR2 = 1, so $R \leftarrow$ Shift
0	0	0	`XXXXXXXX`	Set the processor to idle

Lecture 16

When we move to larger systems, we will need to be able to address specific memory locations. This can be done with a decoder (which is also a demultiplexer). We will need to have a read line R, which is toggled based on whether memory is being read or not. The data in, and data out lines are connected to all of the flip-flops. The enable line of a D-Q flip-flop $M[i]$ has the logic $R' \cdot D_i(\text{ADDRESS}) \cdot \text{Clk}$. Note that $D_i(\text{ADDRESS})$ is the line corresponding to the address in the decoder, this then allows the specific bit to be written.

Note that reading is a combinatorial circuit, as it is simply $R' \cdot D_i(\text{ADDRESS}) \cdot Q_i$, but writing is a sequential circuit, as it requires the address, and data to be present on the clock edge pulse. This asymmetric design is known as static RAMs, which are used in special applications, due to their larger physical space, but are generally faster.

In the processor, we have collections of wires, known as busses; they are as follows;

- address bus

  since the same address lines go into each decoder, it is referred to as the address bus

- control bus

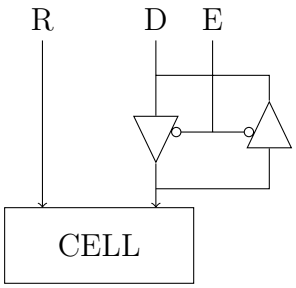
  the control bus consists of the system clock, as well as the read line

- data bus

  as we won't need to use both the data in, and data out lines at the same time, it makes sense for us to create a bi-directional bus, as it would reduce the size, and complexity of the memory circuit

  however, we can encounter some issues with this, as lines may short circuit, causing damage

In order to use a bi-directional data bus, a new gate called a **tri-state buffer** is required, which follows the input exactly if the input, C_i, of the buffer is 0. However, if C_i is set to 1, the output is neither 0, nor 1, and it is effectively disconnected from the data line. This way, many different sources can feed into a single line, as long as only of of the $C_i = 0$. This is yet another use of the demultiplexer. Physically, the RAM is organised in to a two-dimensional grid of units, where it is only active if both the decoded column, decoded row, are 1. Every cell is connected to the same bi-directional data line, and the same read/write line. The data line is connected through a two-way tri-state buffer, which prevents data from flowing in, or out, if the buffer isn't enabled. The following diagram is connected to each cell.



In order to connect RAM to the processor, we need a few more additional registers;

- Memory Address Register (MAR)

  stores the address in memory, which will be stored or read
- Memory Data / Buffer Register (MDR / MBR)

  stores the data read from the memory, or data to be written to memory
- Program Counter (PC)

  stores the address of the next program instruction
- Instruction Register (IR)

  already discussed, but this time it's connected to the MDR / MBR

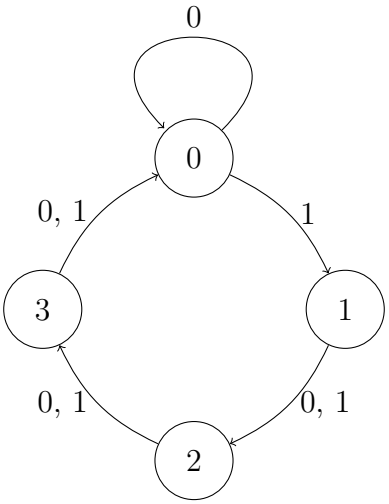
In order to retrieve data from memory, we need to use the fetch cycle; the following cycle would get the next program instruction, and load it into the instruction register.

1. $\text{MAR} \leftarrow \text{PC}$
2. $\text{MDR} \leftarrow \text{RAM}[\text{MAR}], \text{PC} \leftarrow \text{PC} + 1$

   note that this does two register transfers at the same time
3. $\text{IR} \leftarrow \text{MDR}$

We can model this as a sequential circuit;

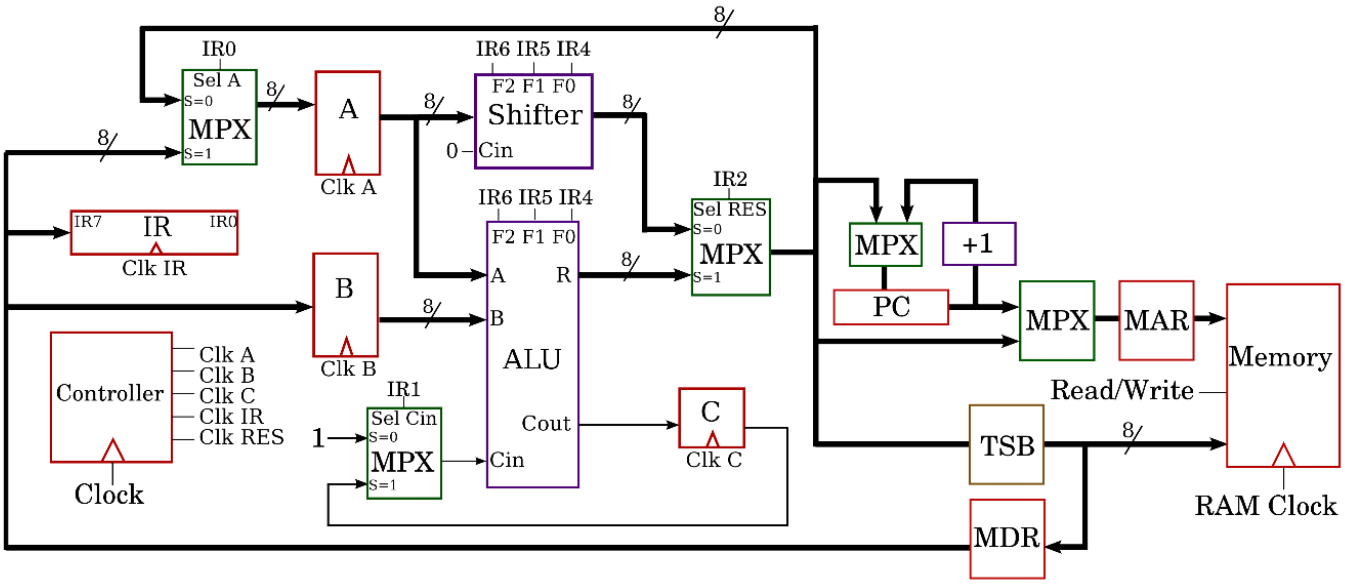
State	Action	Clock Control				Multiplexer Control	
		MAR	MDR	IR	PC	PC_{input}	$\text{MAR}_{\text{input}}$
0	executing	0	0	0	0	×	×
1	load MAR	1	0	0	0	×	0
2	load MDR/PC	0	1	0	1	0	×
3	load IR	0	0	1	0	×	×



For larger RAM, where the chip is greater than 1 Mbit, D-Q flip-flops are too big to be used. Each bit is instead a transistor and a capacitor, where the charge of the capacitor is the value of the bit. This is called dynamic RAM, as the store is not permanent, and the values drift to zero rapidly. Refreshing the charge is done when the computer is not accessing the memory, and a controller must periodically tell the DRAM to refresh itself - this has low overhead.

Lecture 17

The diagrams in these sections have too many parts for me to do in TikZ. I mean it this time, as such, they are pasted directly in from *Notes17 - 32-bit Processor.pdf*. Given that we have created a memory module, and the processor, we can then combine them to create the circuit below.



However, this design still has quite a few issues; you'll see how we fetch from memory 4 times during one instruction, which adds on 12 stages to our original sequential circuit, with a total of 17 steps (plus an idle state). We also don't have the result register, therefore we need to store that somewhere within memory. However, finding out where that will be would require another 3 steps, since that would have to be loaded as well, which means we'd need an extremely large multiplexer to replace the selection for R, or we'd need to read it through A, which would then overwrite it, thus causing us to lose our computed result. There are multiple ways to get around this;

1. scale the architecture up to 32-bits, or even higher

   everything will then use four times as many gates, but we aren't reading, or writing four times as fast

   we can also fetch all the bytes for our instructions at the same time

   we no longer need to consider carries, as we will have a much larger possible set of numbers to work on

2. most computations would need local storage, so we could provide a set of central processor registers, which would be much faster than saving in memory

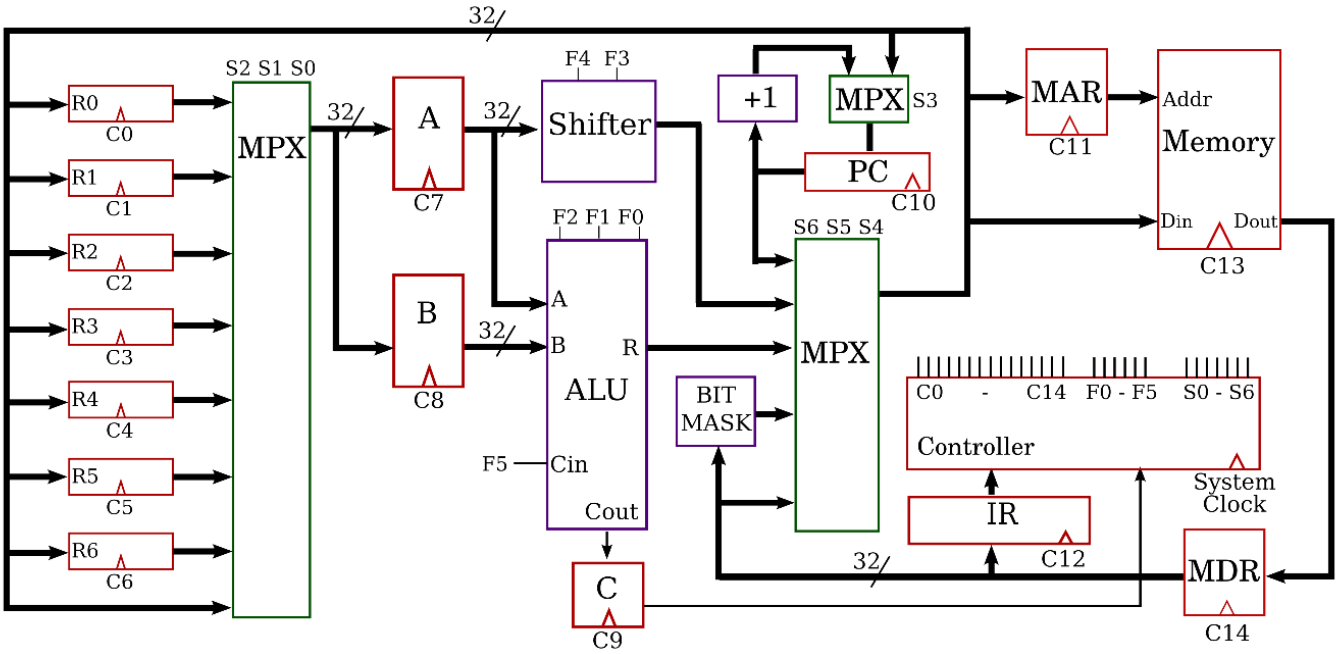
   this design will use 7 local registers, but can be any number, scaling up with cost

3. instead of using IR bits directly to set the ALU, use them as inputs to the controller to become more flexible

   this will a set of more useful instructions

4. rearrange the memory so that the data lines are separate, and therefore are always enabled

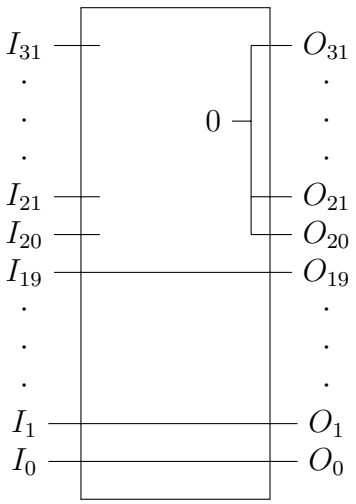
   while the bi-directional lines with the tri-state buffer saves space, optimising speed is more important in this case



You'll note that this design still retains registers A, and B. This is so we have registers that we can manipulate, but programmers cannot modify (or directly use). This way, we can modify the registers without crashing the program. Additionally, this removes the need for a large set of combinatorial circuits, which may have spike issues when the clock speed is increased. The ALU doesn't change, other than the size, and therefore the complex carry arrangements are no longer needed - the carry bit is now used as an input to the controller. If the controller is able to determine the carry bits, we can use that for instructions that allow for conditional branching.

The controller will exist as a FSM which will start by fetching program instructions from memory, and then providing operations to execute the instructions. However, we don't yet know what the steps required are, only that there will be fetch states F_i, followed by some execution states E_i.

Now, we can limit the number of operations to 255 - therefore we can reserve the top 8 bits of the word will define the instruction. This is called the **Opcode**. As the majority of the operations will involve the internal registers, the next 4 bits will store the destination register (note that it's not 3, as we want to be able to expand to more registers in the future). Any remaining data carried in the instruction will be stored in the remaining bits. This allows us to remove any operation data (the first 12 bits) with a simple mask pictured below (the first 12 outputs are connected to ground, and the remaining are connected directly through);



Memory Reference Instructions

In the table below, I'll list the instructions used. Note that for all of them, bits 31-24 (inclusive) are used for the Opcode, and therefore I won't include them. Note that to save space, LOADI means

LOADINDIRECT, and similar for STORE, JUMP, and CALL.

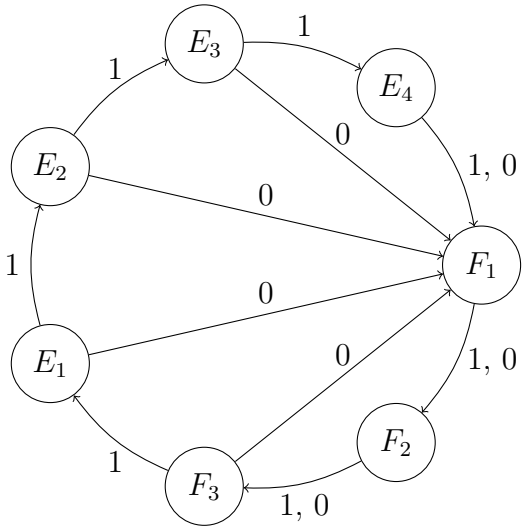
Instruction	I_{23-20}	I_{19-16}	I_{15-0}	Cycle	Transfers	Path
LOAD	R_{dest}	$\times$	Adr	E_1	MAR $\leftarrow$ MDR	bit mask
				E_2	MDR $\leftarrow$ Mem	
				E_3	$R_{\text{dest}} \leftarrow$ MDR	no mask
STORE	R_{dest}	$\times$	Adr	E_1	MAR $\leftarrow$ MDR; $A \leftarrow R_{\text{dest}}$	bit mask
				E_2	Mem $\leftarrow A$	via shifter (hold)
JUMP	$\times$	$\times$	Adr	E_1	PC $\leftarrow$ MDR	bit mask
CALL	R_{dest}	$\times$	Adr	E_1	PC $\leftarrow$ PC + 1	
				E_2	$R_{\text{dest}} \leftarrow$ PC	
				E_3	PC $\leftarrow$ MDR	bit mask
LOADI	R_{dest}	R_{src}	$\times$	E_1	$A \leftarrow R_{\text{src}}$	
				E_2	MAR $\leftarrow A$	via shifter (hold)
				E_3	MDR $\leftarrow$ Mem	
				E_4	$R_{\text{dest}} \leftarrow$ MDR	no mask
STOREI	R_{dest}	R_{src}	$\times$	E_1	$A \leftarrow R_{\text{src}}$	
				E_2	MAR $\leftarrow A$; $A \leftarrow R_{\text{dest}}$	via shifter (hold)
				E_3	Mem $\leftarrow A$	via shifter (hold)
JUMPI	$\times$	R_{src}	$\times$	E_1	$A \leftarrow R_{\text{src}}$	
				E_2	PC $\leftarrow A$	via shifter (hold)
CALLI	R_{dest}	R_{src}	$\times$	E_1	PC $\leftarrow$ PC + 1; $A \leftarrow R_{\text{src}}$	
				E_2	$R_{\text{dest}} \leftarrow$ PC	
				E_3	PC $\leftarrow A$	via shifter (hold)
MOVE	R_{dest}	R_{src}	$\times$	E_1	$A \leftarrow R_{\text{src}}$	
				E_2	$R_{\text{dest}} \leftarrow$ Shifter	via shifter (hold)
ADD	R_{dest}	R_{src}	$\times$	E_1	$A \leftarrow R_{\text{src}}$	
				E_2	$B \leftarrow R_{\text{dest}}$	
				E_3	$R_{\text{dest}} \leftarrow \text{ALU}_R$; $C \leftarrow \text{ALU}_{C_O}$	ALU=$A + B$, $C_I = 0$
COMPARE	R_{dest}	R_{src}	$\times$	E_1	$A \leftarrow R_{\text{src}}$	
				E_2	$B \leftarrow R_{\text{dest}}$	
				E_3	$C \leftarrow \text{ALU}_{C_O}$	ALU=$A - B$, $C_I = 0$
CLEAR	R_{dest}	$\times$	$\times$	E_1	$R_{\text{dest}} \leftarrow \text{ALU}_R$	ALU=0 out
INC	R_{dest}	$\times$	$\times$	E_1	$A \leftarrow R_{\text{dest}}$	
				E_2	$B \leftarrow \text{ALU}_R$	ALU=0 out
				E_3	$R_{\text{dest}} \leftarrow \text{ALU}_R$; $C \leftarrow \text{ALU}_{C_O}$	ALU=$A + B$, $C_I = 1$
DEC	R_{dest}	$\times$	$\times$	E_1	$A \leftarrow R_{\text{dest}}$	
				E_2	$B \leftarrow \text{ALU}_R$	ALU=-1 out
				E_3	$R_{\text{dest}} \leftarrow \text{ALU}_R$; $C \leftarrow \text{ALU}_{C_O}$	ALU=$A + B$, $C_I = 0$
COMP	R_{dest}	$\times$	$\times$	E_1	$A \leftarrow R_{\text{dest}}$	
				E_2	$B \leftarrow \text{ALU}_R$	ALU=-1 out
				E_3	$R_{\text{dest}} \leftarrow \text{ALU}_R$	ALU=$A \oplus B$
ASL	R_{dest}	$\times$	$\times$	E_1	$A \leftarrow R_{\text{dest}}$	
				E_2	$R_{\text{dest}} \leftarrow$ Shifter	via shifter (a. left)
RETURN	R_{dest}	$\times$	$\times$	E_1	$A \leftarrow R_{\text{dest}}$	
				E_2	PC $\leftarrow$ Shifter	via shifter (hold)
SKIP	$\times$	$\times$	$\times$	E_1	PC $\leftarrow$ PC + 1	

SUBTRACT, AND, OR, and XOR are done in the same way as ADD, with the appropriate ALU settings. COMPARE is just subtract with a zero check. Since we don't have any hardware for checking ALU, we can check the carry bit (0 if $R_{\text{src}} \leq R_{\text{dest}}$, 1 otherwise). Other shifts are done in the same way as ASL, but the settings, and carry will change. COMP would flip the bits for the register, and in order to get a number in twos complement, we'd do a COMP followed by INC.

There are some limitations to this design, for example - INC, DEC, and COMP, B is loaded from the main bus, but A is loaded from the registers the programmer specified; another multiplexer could be used to allow these to happen simultaneously.

Lecture 18

We can model the cycles as a FSM, but what we need to work out is the input to the FSM;



The input, C, is a combinatorial circuit of IR_{31-24}, Q_2, Q_1, and Q_0. In order to make it easier for us to deal with the opcode, we can use an 8-256 demultiplexer, which will give us 1 line corresponding to each possible instruction, only one of which will be 1, depending on the opcode. In order to make this easier for us, we can group together instructions that have the same state sequence; for example, we can say ADDS = ADD + SUBTRACT + AND + OR + XOR, and SHIFTS = ASL + ASR + ROR, and so on - this will allow us to save some time when we start the circuit. We will also need to know the states, however, this can be done with a 3-8 demultiplexer, in the same fashion as the demultiplexer used for the opcode.

Now, we can work the return condition from the state E_2 (I explicitly mention state here, as we will be using E_2 to refer to the 3-8 multiplexer output later on). The condition is $(E_2 \cdot (\mathrm{RETURN} + \mathrm{SHIFTS} + \mathrm{MOVE} + \mathrm{JUMPI}))'$. This is inverted, as if it is on E_2, and it's currently performing the last action (since these items all have 2 stages), it should output a 0, as it will need to return to the state F_1 (see the FSM). The final outcome will be $C = (F_3 \cdot \mathrm{NOP})' \cdot (E_1 \cdot (\mathrm{SKIPS} + \mathrm{CLEAR} + \mathrm{JUMP}))' \cdot (E_2 \cdot (\mathrm{RETURN} + \mathrm{SHIFTS} + \mathrm{MOVE} + \mathrm{JUMPI}))' \cdot (E_3 \cdot (\mathrm{COMP} + \mathrm{DEC} + \mathrm{INC} + \mathrm{COMPARE} + \mathrm{ADDS} + \mathrm{STOREI} + \mathrm{LOAD}))'$. This extremely long input, is then used in the state transition table, which follows the same methodology as we have used multiple times before. We can also reuse the outputs of the multiplexers, once we do our minterms.

Finally, we can start working on the output logic. However, there are 28 combinatorial circuits that we will need, each having 17 inputs. This is not feasible to do with Karnaugh maps. We can define the clock signals for when registers have to be updated by referencing the table. For example, we're updating MAR on the following stages; F_1, and certain cases for E_1, and E_2, hence we can say the gate logic for $C_{\mathrm{MAR}} = F_1 + E_1 \cdot (\mathrm{LOAD} + \mathrm{STORE}) + E_2 \cdot (\mathrm{LOADI} + \mathrm{STOREI})$ - the same is done for the other registers. Sometimes, we don't care about the content of the register, for example - because we only need the MAR to be accurate when we load the MDR, we can simplify the gate logic to $C_{\mathrm{MAR}} = F_1 + E_1 + E_2$. This is combined with a NAND gate with the system clock, so that the state changes happen on a different edge to the loading. However, because we care about the MDR, we will follow the proper procedure, and obtain $C_{\mathrm{MDR}} = F_2 + E_2 \cdot \mathrm{LOAD} + E_3 \cdot \mathrm{LOADI}$. The same is also done for the function bits, which are gated based on when they are used, depending on the configuration for the instruction.