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 4 inputs; A, B, C, and D, where $R_n = f_n(A, B, C, D)$. 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;

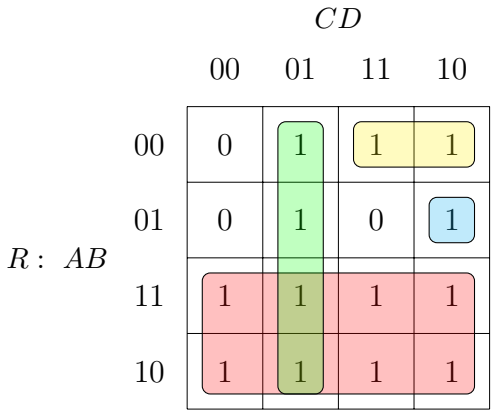
A	B	C	R	
0	0	0	0	maxterm $A + B + C$
0	0	1	0	maxterm $A + B + C'$
0	1	0	0	maxterm $A + B' + C$
0	1	1	1	minterm $A \cdot B \cdot C$
1	0	0	0	maxterm $A' + B + C$
1	0	1	1	minterm $A \cdot B' \cdot C$
1	1	0	1	minterm $A \cdot B \cdot C'$
1	1	1	1	minterm $A \cdot B \cdot C$

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 device 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 components 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 $V(0) = 0$)

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

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

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

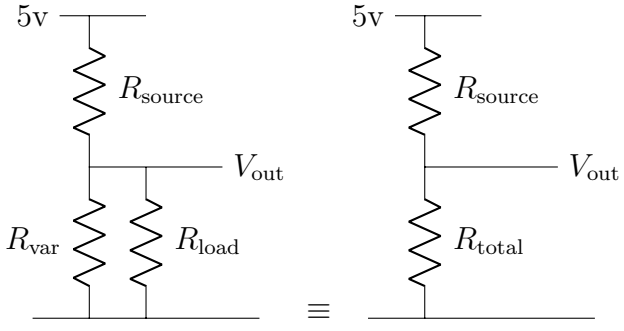
- **Resistance and Capacitance**

  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.

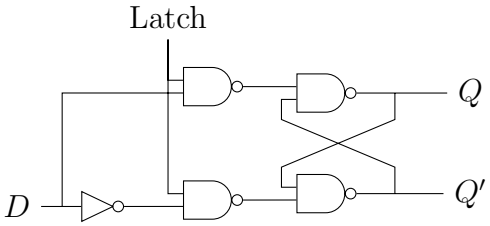
Lecture 7

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 $R = 0$.

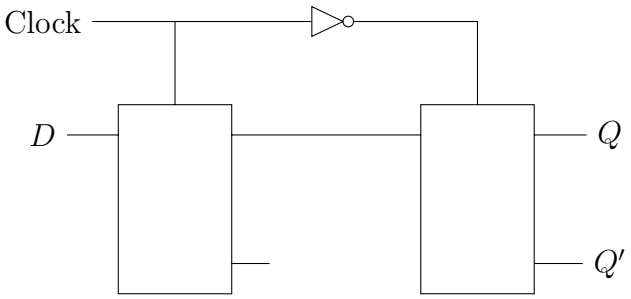
Looking at the states for the **R-S flip flop**, we can get the following states;

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 bistable, 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 $RS(0, 1) \to Q = 1$, and $RS(1, 0) \to Q = 0$. 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 $Q = D$.

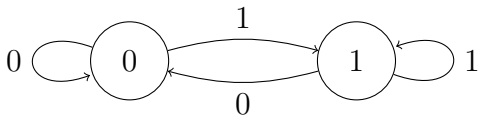


However; there would be a time delay, as the NOT gate would introduce a small delay. We can get around this by using a **Master-Slave Flip-Flop**, which combines two D Flip-Flops, thus allowing for Q to only be set on the **falling edge** of the clock signal.

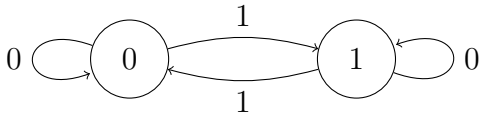


While we don't have to memorise the circuit diagrams of all the flip-flops for the exam, it's important to remember the state diagrams;

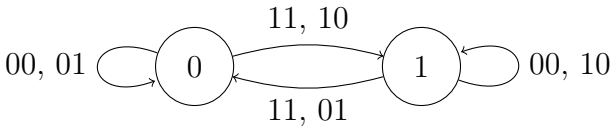
- D-Type Flip-Flop



- 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 synchronise 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;

- Mealy Machine

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

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

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

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

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.

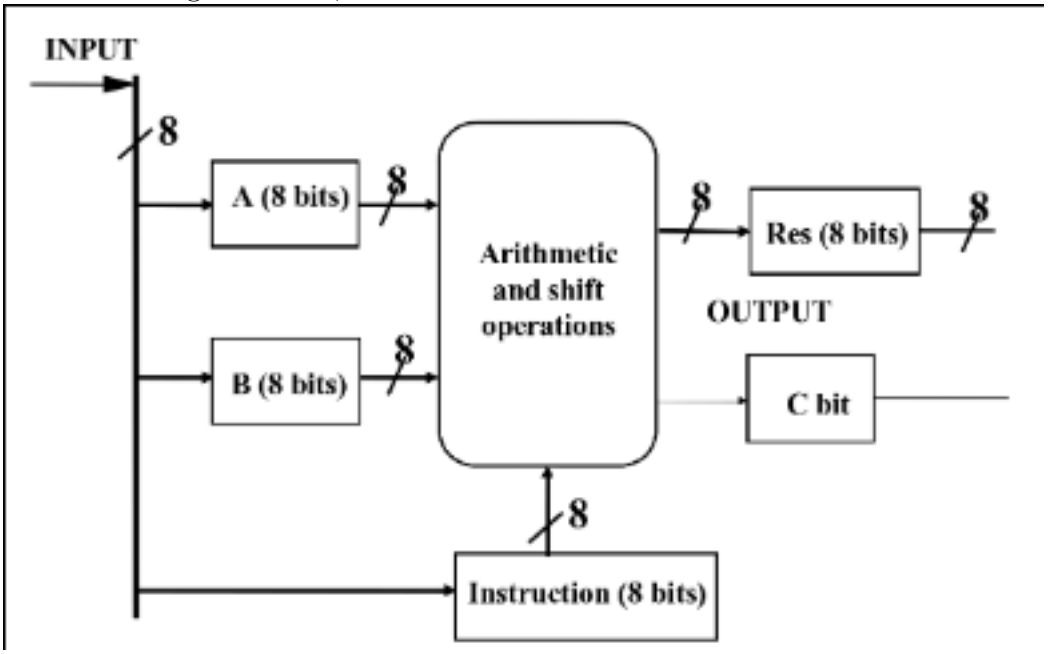
Our example action is to find the average of two numbers, A, and B, such that $R = \frac{A+B}{2}$. Since we are dealing with 8 bits, $A + B < 256$.

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 shift (A) one bit to the right, which is integer division by 2
6. Result is loaded into (Res).

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

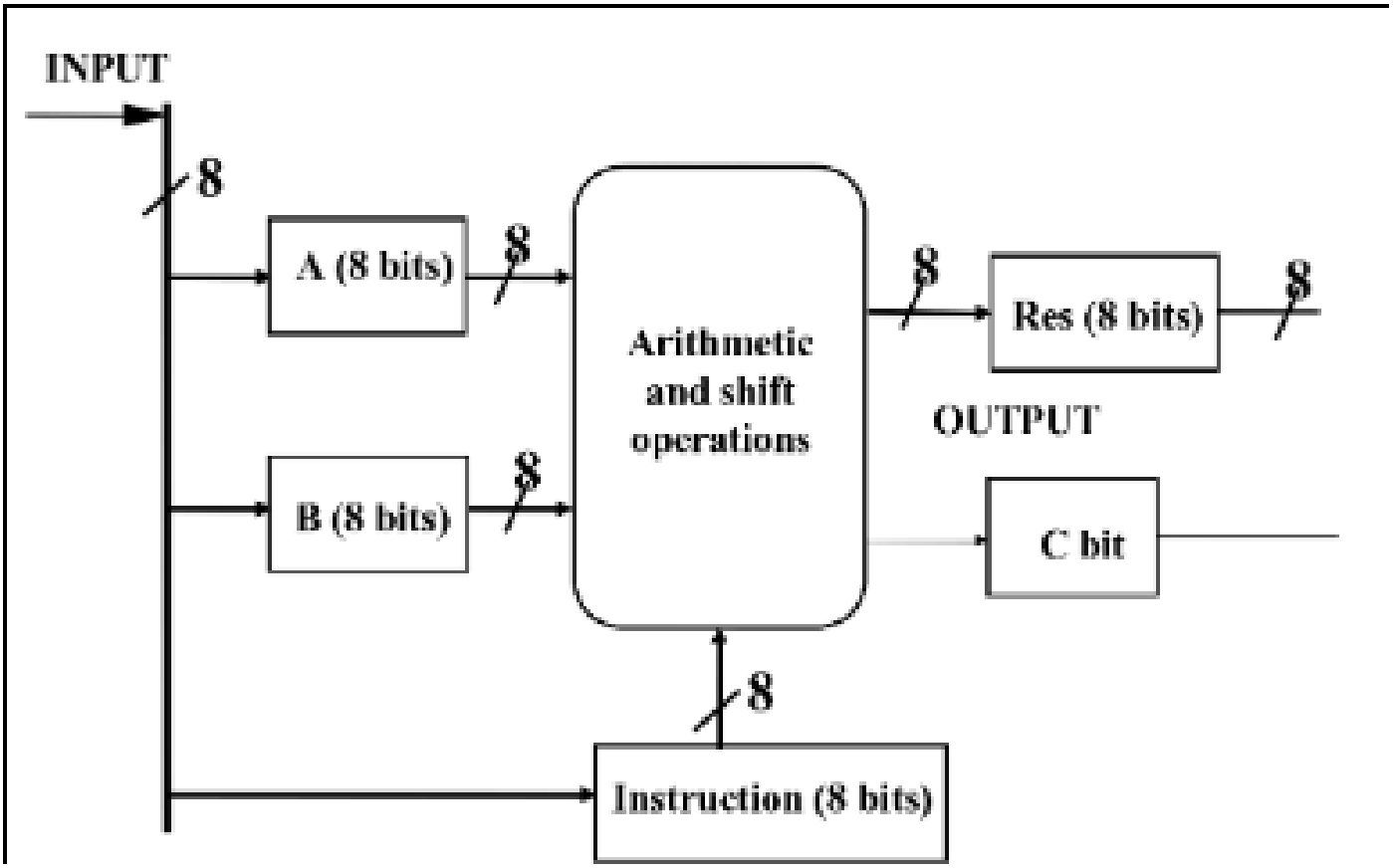
- Registers
  - store data (A), and (B)
  - store result (Res)
  - one bit for carry (C)
  - store instruction (IR)
- Arithmetic circuits
  - 8-bit adder
  - 8-bit shifter

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



PANOPTO 1:21:56

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.


INPUT
8
A (8 bits)
8
B (8 bits)
8
Arithmetic
and shift
operations
8
Res (8 bits)
8
OUTPUT
C bit
8
Instruction (8 bits)