CSC258 Notes

https://github.com/ICPRplshelp

December 14, 2022

Contents

1 Electrical components **1**
- 1.1 Electricity . . . 2
- 1.2 The Flow of Electrons . . . 2
  - 1.2.1 What happens when a battery runs out of power? . . . 2
- 1.3 The Idea of Static Electricity . . . 2
  - 1.3.1 Why do Electrons orbit? . . . 3
  - 1.3.2 Plug or Battery . . . 3
  - 1.3.3 The Ground . . . 3
- 1.4 Semiconductors . . . 3
  - 1.4.1 Impurities . . . 3
- 1.5 PN-Junctions . . . 4
- 1.6 Fields, Diffusion, and Drift . . . 4
- 1.7 Making It Conduct Again – Forward Bias . . . 5
- 1.8 Reverse Bias . . . 5
- 1.9 MOSFETs . . . 6
- 1.10 Power consumption . . . 7
- 1.11 Capacitors . . . 7
- 1.12 pMOS and nMOS Conventions . . . 7

2 Binary and hexadecimal **7**
- 2.1 Conversions between . . . 7
- 2.2 Two's complement . . . 8

3 Demystifying logic gates **8**
- 3.1 Axioms and Theorems of Boolean algebra . . . 8
- 3.2 K-Maps . . . 11
- 3.3 Don't care (X) . . . 11

4 X and Zs **11**
4.1 The illegal value X . . . 11
4.2 The floating value Z . . . 12
4.3 Demystifying Zs . . . 12
4.4 The tristate buffer . . . 13

5 Multiplexers **13**
5.1 The objective of a multiplexer . . . 14
5.2 More complicated multiplexers . . . 15
5.3 Compacting multiplexers . . . 15
5.4 Demultiplexers . . . 16
5.5 Encoding and decoding . . . 16

6 Timing specifications and delays **16**

7 Arithmetic building blocks **17**
7.1 Half adder . . . 17
7.2 Full adder . . . 18
7.3 Ripple-carry adder and its delay . . . 19
7.4 Permitting subtraction . . . 20
7.5 Option to add or subtract based on OP . . . 20
7.6 The carry-lookahead adder . . . 21
7.7 Equality Comparators . . . 22
7.8 Inequalities . . . 23

8 Saving state – Bistable elements **23**
8.1 The inverter . . . 23
8.2 The unamended SR Latch . . . 24
8.3 The clock . . . 25
8.4 The D-Latch . . . 25
8.5 The D Flip-Flop . . . 27
8.6 Other flip-flops . . . 28
8.7 Input waveforms: D-latch vs. D flip-flops . . . 29
8.8 Synchronous Logic Design . . . 29
8.9 Timing constraints . . . 30
8.10 Shift registers . . . 30

9 Finite state machines **30**
9.1 State transition diagrams . . . 30
9.2 State transition tables . . . 31
9.3 The looks of a Moore FSM . . . 32
9.4 To make a Moore FSM . . . 32
9.5 The Mealy FSM . . . 33

9.6 Factoring state machines 35

10 Digital Building Blocks **35**
10.1 Operations done by ALUs 35
10.2 Representing Shifts 36
10.3 The Datapath vs. The Control Unit 36
10.4 Complex Operations With Only One ALU 37

11 Memory **37**
11.1 Interface to Memory Array 37
11.2 Multi-Ported Memory And Register Files 39
11.3 Registers vs. Memory 40
11.4 Tristate Buffers 40
11.5 The Processor-Memory Interface 41
11.6 Comparing Between Different Types of Memory 41
11.7 Estimating Costs 42
11.8 General Purpose Processors and Optimized Processors 43
11.9 In Summary 44

12 How to Assembly **44**
12.1 Programming the processor 44
12.2 Adding two numbers together 44
12.3 Register-type operation 45
12.4 What's up with the register locations? 46
12.5 Label names for registers 46
12.6 I-type operations 47
12.7 J-type instructions 48
12.8 How are the operations broken down? 48
12.9 Assembly language 49
12.10Divide and multiply? 50

13 Assembly Language Instructions **50**
13.1 R-type vs. I-type arithmetic 50
13.2 Logical operations 51
13.3 Shift instructions 51
13.3.1 Shift instructions 51
13.4 Data movement instructions 51
13.5 ALUs and Control Flow 51
13.6 Example program 52
13.7 To make an assembly program 52
13.8 To Simulate MIPS A.K.A. MARS. 53
13.9 Jumps 53

13.9.1 Limits of `j` . . . 53

14 Control flow instructions **53**
14.1 Jump to register . . . 54
14.2 Branch instructions . . . 54
14.3 Multiple if conditions . . . 55
14.4 Interacting with memory . . . 56

15 Loading and storing instructions **56**
15.1 Memory instructions in MIPS assembly . . . 56
15.2 Alignment requirements . . . 56
15.3 Little endian vs. Big endian . . . 57
15.3.1 Big endian . . . 57
15.3.2 Little endian . . . 57
15.4 Reading from devices . . . 57
15.5 Trap instructions . . . 57
15.6 Arrays . . . 57

16 The call stack **58**
16.0.1 When do you store values on the stack? . . . 59
16.1 Passing function parameters / levels . . . 59
16.2 Calling convention . . . 60
16.3 During a function call . . . 60
16.4 The recursive factorial function . . . 60
16.4.1 Steps to perform: . . . 61
16.5 Okay, so how do I actually call and how do I return? . . . 61

17 Pseudo-Instructions **61**

18 Interrupts **62**
18.1 Polled Interrupting . . . 63
18.2 Vector . . . 63
18.3 Interrupt handling . . . 63
18.4 Shutting down the computer . . . 64

19 Microarchitecture **64**
19.1 The Single Cycle Processor . . . 65
19.2 What does a processor do? . . . 65
19.3 What state do we need in the processor? . . . 66
19.4 The Processor's Data And Control Unit . . . 66
19.5 Outputs of the Control Units . . . 67
19.6 Outputs of Anything Else . . . 68
19.7 Analyzing The Single cycle Processor . . . 68

19.8 Supporting J-type Instructions 70
19.9 Determining The Critical Path 70

20 Processors **71**
20.1 How Can ARM Run Intel Apps? 71
20.2 Takeaways 71
20.3 The actual memory map 71
20.4 The Dynamic Data Segment 72
20.5 Translating and Starting a Program 72
20.6 Compile vs. Run-Time 73
20.7 Compiled vs. Interpreted 73
20.8 Sharing Memory 74
20.9 Sharing The Processor 74
20.10 Exceptions in MIPS 75
20.11 Causing Exceptions 75
20.12 Is This Section on the Exam 76

21 Exam Tips **76**
21.1 Disambiguating Assembly To Machine Code 76
21.1.1 The Confusing Shift Function 77
21.2 What Is The Difference Between RS, RT, and RD? 78
21.3 How to Annotate the Datapath 79
21.4 Do You Want To Divide, or Do You Want To Shift? 79
21.5 Set Less Than: The Most Unclear instruction 79

1 Electrical components

Transistors are the basic building blocks of hardware. We can see them as gates that could pass a value. A transistor acts as a faucet.

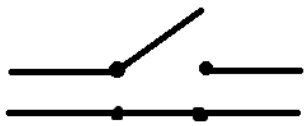


Figure 1: What transistors are – gates.

When looking at logic gates, we need to see where they come from:

- Logic gates are made from
- Transistors, which are based on

- PN-junctions, which are made from
- Semiconductors, that conduct
- Electricity

1.1 Electricity

> Electricity is the flow of electrons through a material.

Protons have positive (+) charge. Electrons have negative (-) charge. **In a regular atom, the proton count is $=$ to the electron count.** Hence, if you remove an electron from an atom, the atom becomes positively charged as it has more protons than electrons. The same argument can be made the other way around.

1.2 The Flow of Electrons

Electrons want to flow from regions of **high electrical potential** to regions of **low electrical potential.** Electrical potential is called **voltage**. If something has high electrical potential, we can say that it has a lot of electrons (similar to high concentration).

In a battery, the volts is the **difference in potential** between the negative side and the positive side. Electrons flow from the negative to the positive, but out from the battery first then back into the positive side.

Potential difference pushes a current, which goes down the higher the resistance. We can generalize this through an equation:

$$V = IR$$

The current flows depending on how smooth the path is. A path has high resistance if the path isn't smooth.

1.2.1 What happens when a battery runs out of power?

No, the potential difference does not go down the more you use a battery. It suddenly drops down when the battery runs out. **It behaves more like an hourglass – the sand dispensed downwards per unit time is the same until the top part of the hourglass runs out.**

1.3 The Idea of Static Electricity

When you shuffle your feet on a carpet, you are collecting electrons. Then when you touch someone else, a zap occurs.

That is because electrons from you transfer to the other person, as the other person has less electrons than you. Recall that electrons go from a place of high concentration to a place of low concentration.

1.3.1 Why do Electrons orbit?

Electrons are attracted to protons, as unlike charges attract. Circular motion is caused by the acceleration being parallel to the velocity, so a similar argument can be made with satellites orbiting earth.

1.3.2 Plug or Battery

Batteries are limited because you're going to have to replace them. On the other hand, plugs are connected to a power source that always replenishes itself, so you don't need to worry about it running out (at least not you).

1.3.3 The Ground

The ground is always treated as zero volts. Everything else has a voltage, either positive or negative – even if insignificant.

1.4 Semiconductors

Semiconductors are

- Conductors in some cases
- Insulators in other cases

Silicon has 4 (out of the 8 max) valence electrons. Hence, its outer shell is filled halfway. However, each atom wants a full outer layer. That is why you would want to make a silicon lattice by attaching silicon atoms to each other. Covalent bonds are formed, electrons are shared, and each silicon gets to have a full outer layer.

This essentially makes a silicon lattice an insulator. How do we make it a conductor? We **introduce impurities.** We toss in something.

1.4.1 Impurities

We can either toss in a Boron (3 valence electrons) or a Phosphorous (5 valence electrons) into the lattice.

- Tossing in Boron results in “empty seats”, caused by the need for electrons, as now not all valence shells are filled.
  - This creates a p-type (positive)
  - Holes bump into each other, so they move
- Tossing in Phosphorous results in an “extra person”, caused by more electrons being present than the number needed for full valence shells across the lattice. Those free electrons wander around the lattice.
  - This creates an n-type (negative)
  - Free electrons bump into each other, so they move

When impurities are added, **materials can conduct electricity.**

1.5 PN-Junctions

What happens when P and N-types come in contact with each other? That creates a PN junction, which is located at where the P and N-types meet.

The p-type material has holes, and the n-type materials have free electrons. The free electrons closest to the holes (the part closest to the p-type) will occupy these holes, but note that they are still being brought back as they still have the protons back in the n-type to be attracted to.

The depletion region represents the area where the holes have been filled and where the electrons have left. The region with the filled holes is negative, as there are more electrons than protons there. The region where the electrons just left is positive, as when the protons still remain, thus there being more protons than electrons.

This is known as diffusion.

The depletion region will continue to grow... until something stops it.

1.6 Fields, Diffusion, and Drift

When enough electrons fill the empty seats, the positive region (the area the electrons left) becomes positive enough to the point it starts attracting the electrons that just left back. This is **drift.**

Initially, drift isn't strong enough to attract all the electrons that leave back. But drift will strengthen as more electrons leave, until equilibrium is reached: the diffusion and drift effect are the same.

By that point, no electrons will move anymore, and we see no more conduction. Hence, we did all of this to get something else that doesn't conduct anymore.

- Pure silicon (does not conduct)
- Impurities added (conducts again)
- P and N put together (no conduction anymore)

1.7 Making It Conduct Again – Forward Bias

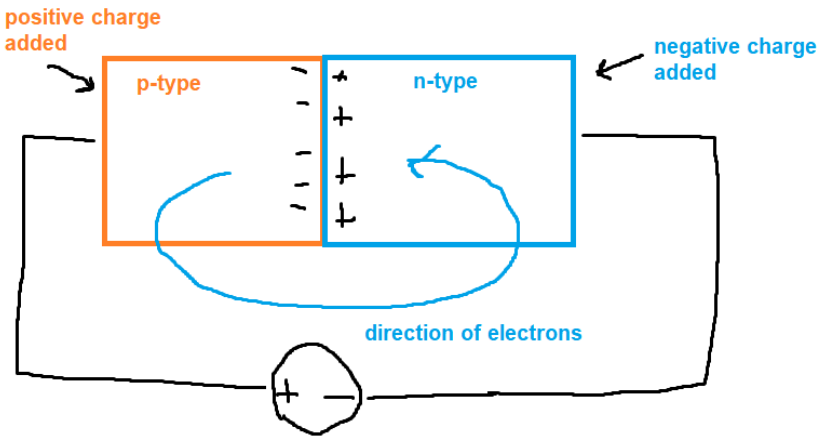


Figure 2: Causing a forward bias.

If we drive negative charge into the n-type and positive charge into the p-type, the following occurs:

- More electrons enter the n-type region, which means more electrons to move to the holes in the p-type region
  - This shrinks the depletion region
- Electrons in the p-type region move back to the n-type through the wire, as it is attracted to the positive charge applied into the p-type (maybe through the wire). This creates more holes (remember that the depletion region is caused by holes being filled).

- **A current is created** in the opposite direction of electron flow (that is by convention).

This is forward bias and **permits a current. We observe a conductor.**

1.8 Reverse Bias

If we apply positive voltage to the n-type, and negative voltage is applied to the p-type, this causes more electrons to go to the p-type region. A large depletion region prevents current (see: equilibrium).

1.9 MOSFETs

Metal oxide semiconductor field effect transistors. Where I want current to flow in some cases and not in other cases.

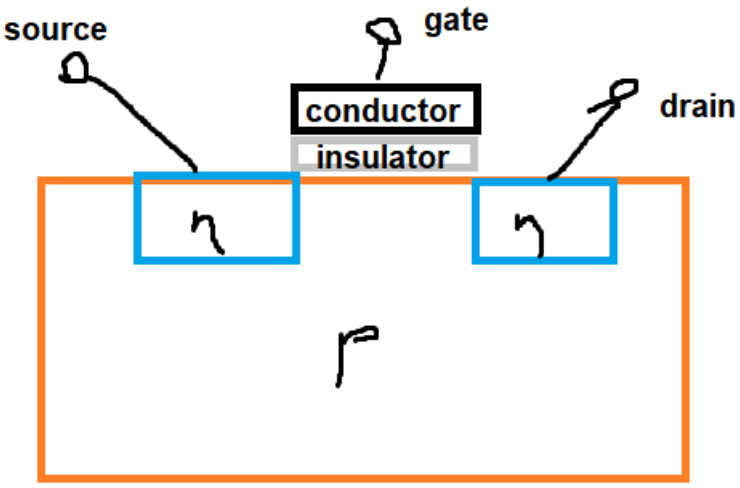


Figure 3: nMOS

If we apply a positive current to the gate (top middle of the diagram), free electrons from the top left region will be attracted to the conductor on the top (which is positively charged by the gate). It can never pass the insulation, so it simply forms a bridge to the top right n region. This allows current to flow from the source to the drain.

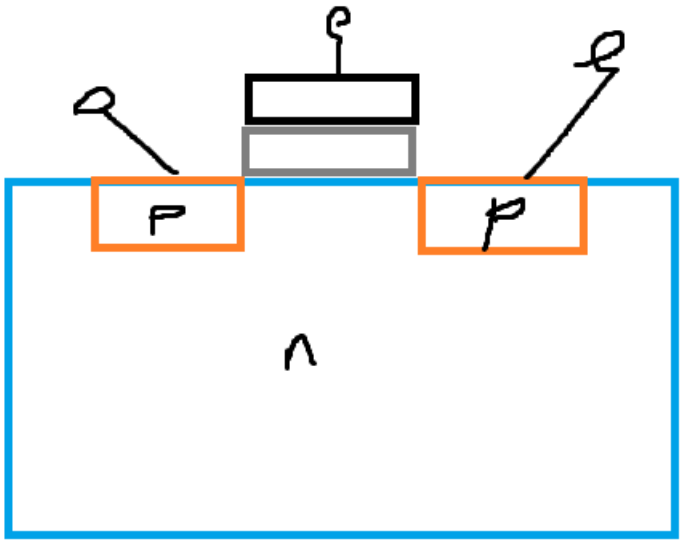


Figure 4: pMOS

The pMOS does the opposite of what an nMOS does. When the gate is 0 (no positive current), then it is *slightly negative,* and will attract the p region, forming a channel that allows current to be passed to the p region on the right.

1.10 Power consumption

Turning transistors on and off dissipates power and takes time.

- Dynamic power is the power used to charge capacitance (ability of a system to store electric charges) when signals change between 0 to 1.
- Static power is power that is used even if the system is idle (not changing between 0s and 1s).

1.11 Capacitors

Imagine two massive metal plates. They are next to each other, but are not touching, and maybe an insulator in between. Charge one of them positively, band equally charge the other metal plate negatively. There's no path between the two, but there's a strong electric field between the two plates. In a way, the capacitor is holding charge, such that if I create a path from the positive side to the negative side, and there is a lightbulb in between, the lightbulb will light up.

Capacitors are not battery. It just holds charge, and batteries use different material to hold charge.

Charging and discharging takes time and energy.

1.12 pMOS and nMOS Conventions

pMOS passes 1s well and nMOS passes 0s well.

2 Binary and hexadecimal

Binary is base 2, and hexadecimal is base 16.

2.1 Conversions between

Binary to hex: from right to left, group them in 4. Then, convert each one to hex. Because any 4-bit binary can be represented as a unique hex value, this is easy to do.

Hex to binary: same logic as above, but reverse.

Binary to decimal:

$$\begin{aligned} 101101_2 = &\left(1 \times 2^0\right) + \left(0 \times 2^1\right) \\ &+ \left(1 \times 2^2\right) + \left(1 \times 2^3\right) \\ &+ \left(0 \times 2^4\right) + \left(1 \times 2^5\right) \end{aligned}$$

Hexadecimal to decimal: Similar to binary.

Decimal to binary: Repeatedly divide the number by 2. Append to the left of your binary number the remainder. Repeat until your quotient ends up being 0.

Decimal to hexadecimal: Repeatedly divide the number by 16. Append to the left of your hexadecimal number the remainder, in the form of a hexadecimal digit. Repeat until your quotient ends up being 0.

2.2 Two's complement

To represent a negative number: Take the positive, invert all the bits, and add 1.

You can use two's complements in arithmetic without any problems. Note that a positive plus a positive is a positive, and a negative plus a negative is a negative, so if you see these rules being broken you know you have an overflow.

To subtract, add the negative.

To take the absolute value of a two's complement negative number: Subtract 1 and invert the bits.

3 Demystifying logic gates

Minterms are based on the inputs and have no idea on what makes Y true.

Minterms are like Π and the way you translate ones and zeroes to As and $\overline{A}$s is one-to-one. If the minterm statement returns true, then you've got a 1.

Maxterms are like $\sum$, and the way you translate ones and zeros to As and $\overline{A}$s is flipped. If the maxterm statement returns true, then you've got a 0.

Figure out a way to combine them.

3.1 Axioms and Theorems of Boolean algebra

Table 1: Axioms of Boolean algebra

Axiom Number	Axiom	Dual	Name
$A1$	$B \neq 1 \Rightarrow B = 0$	$B \neq 0 \Rightarrow B = 1$	**Binary** field – true isn't false
$A2$	$\overline{0} = 1$	$\overline{1} = 0$	**NOT** – we can invert them
$A3$	$0 \cdot 0 = 0$	$1 + 1 = 1$	**AND**/**OR** – with itself
$A4$	$1 \cdot 1 = 1$	$0 + 0 = 0$	**AND**/**OR** – with itself, again
$A5$	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$	**AND**/**OR** – commutativity and propagation

These axioms follow the principle of duality – if we swap 0s and 1s and ANDs and ORs, the statement is still correct. Use the prime symbol on the axiom number to indicate its dual.

Table 2: Theorems of one variable

Axiom Number	Axiom	Dual	Name
$T1$	$B \cdot 1 = B$	$B + 0 = B$	**Identity** – when the constant matters not!
$T2$	$B \cdot 0 = 0$	$B + 1 = 1$	**Null element** – when B matters not! As 0 nullifies B.
$T3$	$B \cdot B = B$	$B + B = B$	**Idempotency** – same gives itself. *Idem* = same; *potent* = power.
$T4$	$\overline{\overline{B}} = B$	see left	**Involution** – two wrongs make a right.
$T5$	$B \cdot \overline{B} = 0$	$B + \overline{B} = 1$	**Complement** – a zero times a one makes a zero.

Table 3: Theorems of several variables

Axiom Number	Axiom	Dual	Name
$T6$	$B \cdot C = C \cdot B$	$B + C = C + B$	**Commutativity**
$T7$	$(BC)D = B(CD)$	$(B + C) + D = B + (C + D)$	**Associativity** – brackets are free to move… if the operators are identical.
$T8$	$BC + BD = B(C + D)$	$(B + C)(B + D) = B + (CD)$	**Distributivity** – its dual looks confusing. Get used to it.
$T9$	$B(B + C) = B$	$B + BC = B$	**Covering** – couldn't have it in any other way
$T10$	$BC + B\overline{C} = B$	$(B + C)\left(B + \overline{C}\right) = B$	**Combining** – exhausting all combinations.
$T11$	$BC + \overline{B}D + CD = BC + \overline{B}D$	$(B + C)\left(\overline{B} + D\right)(C + D) = (B + C)\left(\overline{B}D\right)$	**Consensus** – the other one doesn't matter.

Table 3: Theorems of several variables

Axiom Number	Axiom	Dual	Name
$T12$	$\overline{B_0 B_1 B_2 \ldots} =$ $\overline{B_0} + \overline{B_1} + \overline{B_2} \ldots$	$\overline{B_0 + B_1 + B_2 \ldots} =$ $\overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2} \ldots$	**De Morgan's Theorem**
Custom	$\overline{A}B + A = B + A$	$\left(\overline{A} + B\right) A + BA$	**Absorption**

The theorem of why factoring like this is acceptable: $B + BC = B(1 + C)$:

Proof. Of the above:

- $B + BC = B \cdot 1 + B \cdot C$ (Identity, opposite direction $B = B \cdot 1$)
- $B \cdot 1 + B \cdot C = B(1 + C)$ (Distributivity – factoring)
- $B(1 + C) = B(1)$ (Null element – $1 + C = 1$ as 1 nullifies C)
- $B(1) = B$ (Identity, regular direction $B \cdot 1 = B$)

■

Simplifying Boolean equations require trial and error, and it is pretty hard without Karnaugh maps.

3.2 K-Maps

Order of placement in a 4×4 k-map:

$$\begin{matrix} 0 & 1 & 3 & 2 \\ 4 & 5 & 7 & 6 \\ 12 & 13 & 15 & 14 \\ 8 & 9 & 11 & 10 \end{matrix}$$

Circles must cover all 1s but no zeros.

Tips for circling:

- Use the fewest circles.
- Circles must look like a rectangle and cannot be rotated. They must span a block with a size that's a power of 2. Meaning 1, 2, 4, 8, or 16.

- Make circles as large as possible. The larger the circle, the less literals in the equation. If that means that you have to overlap, then overlap. The only times where large circles become redundant is if they don't cover any new 1s.
- Circles may wrap around edges.

3.3 Don't care (X)

If these values appear in K-maps, they are wildcards and can be left circled or left alone.

4 X and Zs

Boolean algebra is limited to 0s and 1s, but real circuits can have illegal and floating values, represented symbolically by X and Z.

4.1 The illegal value X

Unknown or illegal value. Occurs if it is being driven to both 0 and 1 at the same time. This occurs when two nodes are combined without a gate. This situation is called **contention**, is an error, and thus must be avoided.

The voltage here is somewhere between 0 and V_{DD}, most likely in the forbidden zone. X may also represent a value that wasn't initialized.

In the context of circuits, this is **not** a "don't care" value, unlike their appearances in truth tables. If X appears in a circuit, the circuit has an unknown or illegal value.

4.2 The floating value Z

The symbol Z indicates the node is neither HIGH nor LOW. We can say it's *floating, high impedance, or high Z.* It may be 0, 1, or in between. It doesn't mean there's an error. If another circuit element drives the node back to a valid logic value, then the Z indicates no error.

Causes include:

- Forgot to connect a voltage to circuit input
- Assume that an unconnected input is the same as an input with a value of 0

4.3 Demystifying Zs

Neither high nor low. A wire is at Z if it isn't connected to power or ground. Here's an example of a bus:

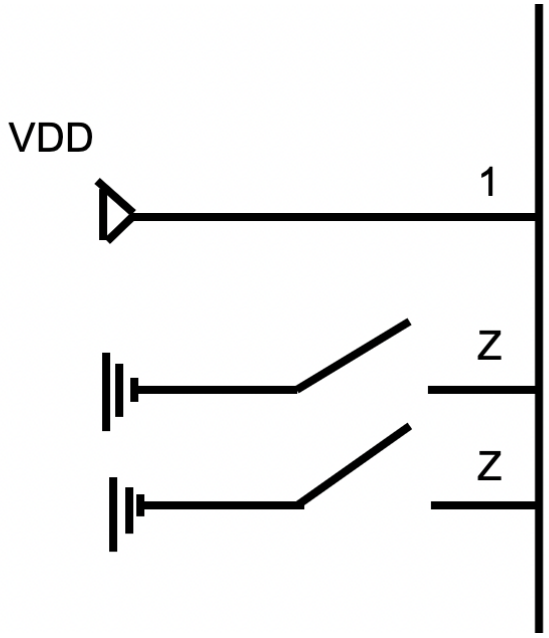


Figure 5: Bus

The one V_{DD} and the two GRDs on the left side are the only parts in the circuit that can write to the write on the right – everything else may only read from the wire at the right. The illegal value X will pop up if the wire on the right is connected to ground and V_{DD} at the same time. To prevent this from happening, we can see that the two gates in the middle are OPEN – and thus what goes from the two grounds to the wire on the right is a Z – because the actual information on whether the wire labelled Z is connected to V_{DD} or ground on the left is removed through the OPEN gate.

This allows the wire on the right to have a value of 1. Note that the vertical wire on the right doesn't have a specific direction of information flow (readers can either be on the top or the bottom), but the horizontal wires relay information from the left to the right.

4.4 The tristate buffer

Has three possible output states:

- HIGH (1)
- LOW (0)

- FLOATING (Z)

Inputs: A, output Y, and enable E.

- When enable is TRUE, the tristate buffer acts as a simple buffer (think Redstone repeater). When false, the output is allowed to float.

5 Multiplexers

A library of combinational circuits:

- Hierarchy
- Modularity
- Regularity

Common applications of them:

- Arithmetic
- If/else statements
- Translating data (encoding and decoding)

5.1 The objective of a multiplexer

Wiring.

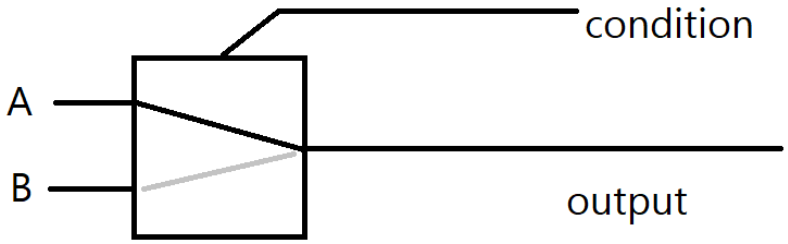


Figure 6: A 2:1 multiplexer.

A multiplexer equation may look like this:

$$Y = A\overline{R} + BR$$

In this example:

- A only matters if $\overline{R}$
- B only matters if R

As if controlling what wires to the output.

- **The nomenclature for a multiplexer.** An a:b multiplexer accepts a inputs and returns b outputs.
- **The relationship between number of selectors and number of inputs and outputs.** If we have M selectors, we will have $N = 2^M$ inputs and 1 output.

5.2 More complicated multiplexers

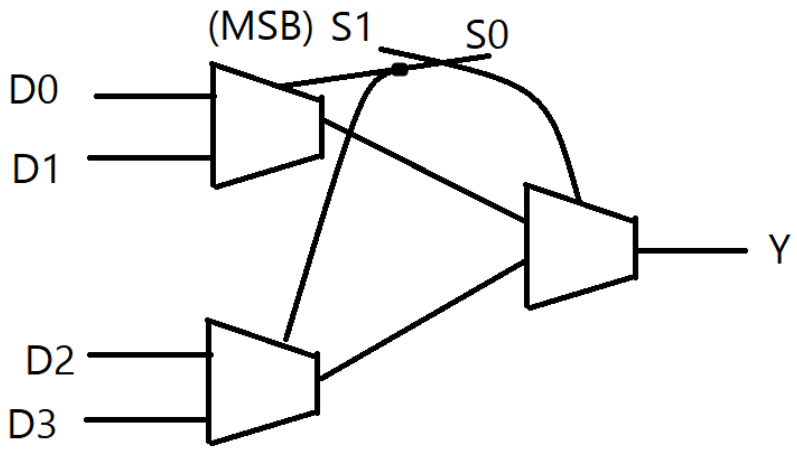


Figure 7: A 4:1 multiplexer.

The MSB connects to the multiplexer on the rightmost side of the diagram. The LSB connects to the multiplexers on the leftmost side of the diagram.

5.3 Compacting multiplexers

An 8:1 multiplexer can be seen having 8 inputs. The switch has 3 bits, which determines which of the 8 inputs should be reflected to the output. For example, if the switch's bits represent 010, the second input from the top should be reflected in the output.

5.4 Demultiplexers

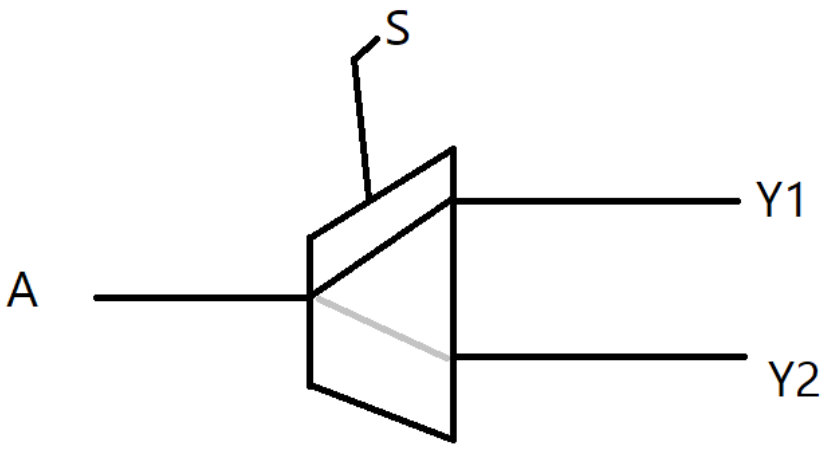


Figure 8: A 1:2 multiplexer.

This tells which output should listen to A.

5.5 Encoding and decoding

In the format no. inputs $\rightarrow$ no. outputs.

Encoding: $2^N \rightarrow N$.

Decoding: $N \rightarrow 2^N$.

A simple decoder will only assert one of its outputs. Encoders are the opposite; only one of its inputs are high. Encoders may have an extra output called “IS VALID,” which is only on if only one of its inputs are high to prevent the effects of precondition violation from propagating.

6 Timing specifications and delays

Definition 6.1 (Propagation delay)**.** Maximum time from when an input changes until **all** outputs reach their final value.

Definition 6.2 (Contamination delay)**.** Minimum time needed for any output value to change.

The delays for each circuit are given to you. Redstone repeaters have some sort of delays, and so will the gates that you will construct with them – and that also must

happen in the real world. At least the delay is way shorter. **The delays are typically given to you for a circuit and/or gate.**

Definition 6.3 (Critical path)**.** Slowest path through the circuit (feels like the longest but is not the longest). Note that wire length does not affect path speed.

Definition 6.4 (Short path)**.** The fastest path through the circuit; opposite of critical path.

To solve questions relating to delays:

- For the **propagation delay, use the critical path.**
- For the **contamination delay, use the short path.**

7 Arithmetic building blocks

7.1 Half adder

A **half adder** looks like this:

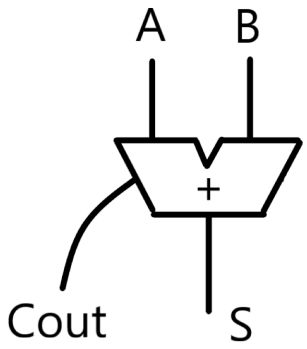


Figure 9: A half adder

It adds A and B together. S is the sum, and C_{out} stands for "carry out," which is 1 if there is any carrying over from addition. The truth table for this circuit goes as follows:

Table 4: Half adder truth table. Input variables are A and B

A	B	S	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This leads up to the Boolean equations of S and C_{out}:

$$S = A \oplus B$$
$$C_{\text{out}} = A \cdot B$$

7.2 Full adder

A **full adder** looks like this, taking in A, B, C_{in} and outputting S and C_{out}:

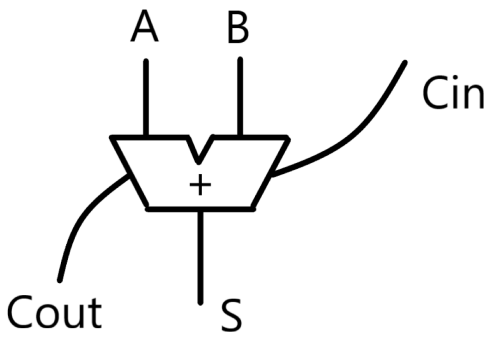


Figure 10: A full adder

It has this truth table:

Table 5: Full adder truth table. The input variables are A, B, and C_{in}

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

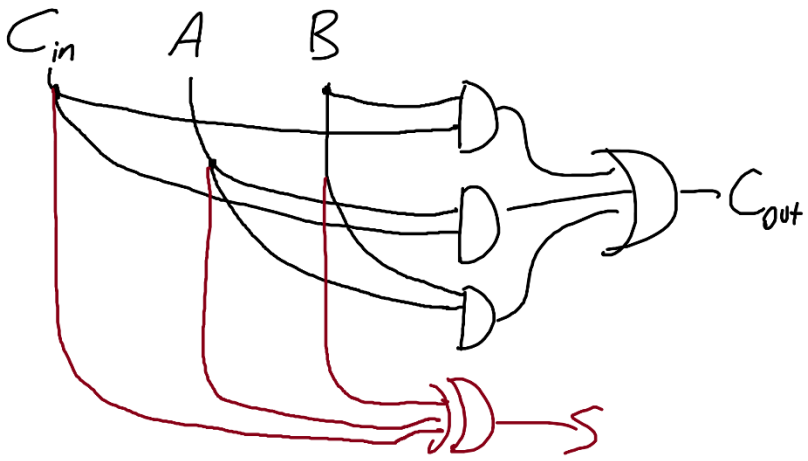
Table 5: Full adder truth table. The input variables are A, B, and C_{in}

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

We end up with the following equations:

$$S = A \oplus B \oplus C$$
$$C_{\text{out}} = BC + AC + AB$$

Here's how the adder looks like internally:



7.3 Ripple-carry adder and its delay

Full adder units chained together.

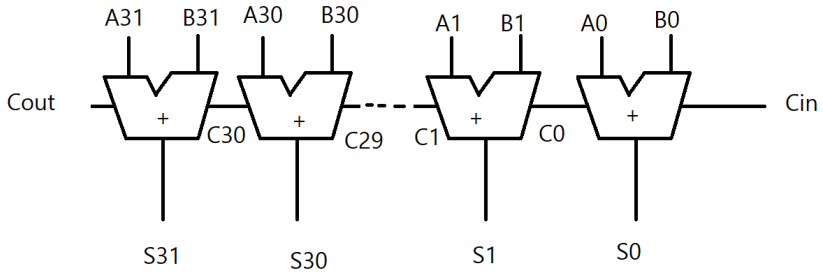


Figure 11: Chained full adder units to form a Ripple-carry adder

Let's say the delay of the full adder t_{FA} is, in picoseconds:

$$t_{FA} = 120$$

Insight – combo every input to output if it exists. Then determine the time it takes. For full adders, the paths are very similar. Choose the path that seems to take the most time to go through.

The time of delay for an N-bit ripple-carry adder is $t = N \cdot t_{FA}$. This means a 4-bit ripple carry adder would have a delay of $4 \cdot t_{FA}$.

7.4 Permitting subtraction

$$A - B = A + (-B)$$

To enable subtraction for a 32-bit adder, just negate all of $B_{0...31}$ and use $C_{in} = 1$.

7.5 Option to add or subtract based on OP

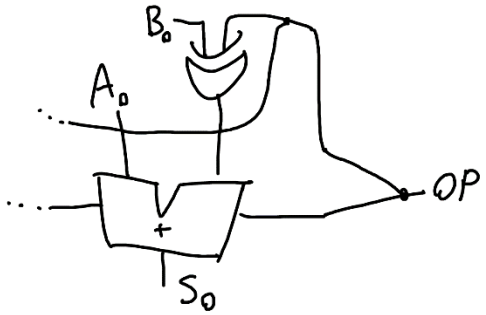


Figure 12: A ripple-carry adder with the option for B to subtract.

When OP is 0, result is sum of A and B. When OP is 1, result is difference between A and B ($A - B$). The XOR gate acts as a togglable negator.

7.6 The carry-lookahead adder

Source: `https://en.wikipedia.org/wiki/Carry-lookahead_adder`

An adder, but fast. It reduces the amount of time required to determine carry bits. The carry-lookahead adder calculates one or more carry bit before summing, which reduces the wait time to calculate the more significant bits of the adder.

This adder uses the concepts of **generating** and **propagating** carries.

- The addition of two bits $A,\ B$ is said to **generate** if the addition will always carry, regardless of C_{in}. In binary addition, $A+B$ generates if and only if $A = 1$ and $B = 1$.
  - Then, $G(A,\ B) = A \cdot B$. More clearly:
  - $(C_{\text{in}} = 0) \Rightarrow A + B$ results in a carry over
- The addition of two bits A, B is said to **propagate** if there is an input carry (when the next less significant digit in the sum carries) – meaning if $C_{\text{in}} = 1$.
  - Then, $P(A,\ B) = A + B$. More clearly:
  - $(C_{\text{in}} = 1) \Rightarrow A + B$ results in a carry over
  - Sometimes, a different definition of propagate is used: $P'(A,\ B) = A \oplus B$.

A digit of addition carries precisely when either

- the addition generates
- or the next LSB carries, **and** the addition propagates.

Thus, we can determine the carry bit of digit i through the following, where P_i and G_i's values are dependent on the bits of digit i (are an output):

$$C_i = A_i \cdot B_i + (A_i + B_i) \cdot C_{i-1} = G_i + (P_i \cdot C_{i-1})$$

These extend to multiple-bit blocks. A block generates a carry if it produces a carry out independent of the carry in, and it propagates a carry if it produces a carry if there's a carry into a block.

If an adder generates, or propagates AND the previous on generates, then it generates.

Let $G_{i:j}$ and $P_{i:j}$ be the generate and propagate signals for blocks with columns i to j inclusive.

Then, a block generates a carry if the MSB in the block generates a carry, or if the MSB in the block propagates a carry and the previous column generated a carry, and so on. This means:

$$\begin{aligned} G_{3:0} &= G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)) \\ &= A_3 B_3 + ((A_3 + B_3)(A_2 B_2 + ((A_2 + B_2)(A_1 B_1 + (A_1 + B_1) G_0)))) \\ G_{n:m} &= G_n + P_n \cdots \end{aligned}$$

A block propagates a carry if all the columns propagate.

$$P_{3:0} = P_3 P_2 P_1 P_0$$

This means we can quickly compute the carry out of the block using the carry in, where the block spans i to j, $i > j$.

$$C_{\text{out}} = G_{i:j} + P_{i:j} C_{\text{in}}$$

For $N > 16$, the carry-lookahead adder is much faster than the ripple-carry adder. However, the delay still increases linearly, so it is not making the runtime from linear to logarithmic, however.

7.7 Equality Comparators

For one-bit numbers, this is the Boolean equation for equality:

$$\overline{A \oplus B}$$

For N-bit numbers, it's this equation:

$$\prod_{i=0}^{N-1} \overline{A \oplus B}$$

7.8 Inequalities

$$a < b \Leftrightarrow a - b > 0$$
$$a > b \Leftrightarrow a - b < 0$$
$$a \leq b \Leftrightarrow a - b \leq 0$$
$$a \geq b \Leftrightarrow a - b \geq 0$$

Weak inequalities are strong inequalities which also return true if the two values are the same.

8 Saving state – Bistable elements

Starting from circuits that store 1 bit of data.

Definition 8.1 (Combinational circuits)**.** Combinational circuits do not depend on previous inputs (what the input "was" and are often seen as having no registers or cycles anywhere).

Definition 8.2 (Sequential circuits)**.** Sequential circuits are those that are dependent on clock cycles and depends on present and past inputs.

8.1 The inverter

An attempt at a bistable circuit using inverters. If $Q = 1$, then $\overline{Q} = 0$.

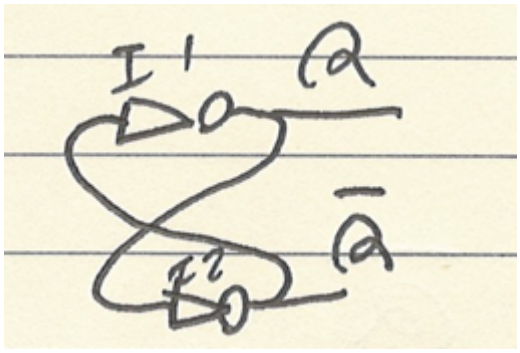


Figure 13: Inverter bistable circuit attempt

The problem is that this circuit has no inputs. It's on or off forever and can't change once set.

8.2 The unamended SR Latch

Two inputs, two outputs.

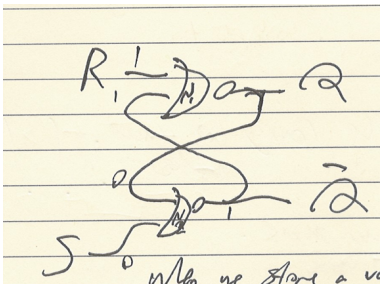


Figure 14: Standalone SR latch

The truth table for this is as follows:

Table 6: SR latch truth table

S "set"	R "rest"	Q	$\overline{Q}$	Comments
0	0	Q_{prev}	$\overline{Q}_{\text{prev}}$	Retain
0	1	0	1	Reset to 0
1	0	1	0	Set to 1
1	1	0	0	??? Avoid

Q_{prev} is the state of Q just before $S = R = 0$.

Pros:

- Can control state
- Can retain state

Cons:

- A contradiction arises when $S = R = 1$, so avoid S and R being 1 at the same time whenever possible
- Asserting S or R dictates both:
  - What state should be
  - When state should change
  - I probably don't want direct control to S and R

8.3 The clock

The heartbeat. It oscillates between 0 and 1.

A clock's value over time might look like this:

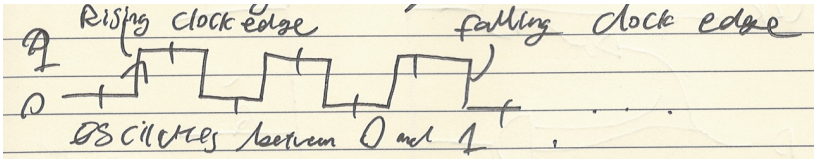


Figure 15: Plot of clock's level over time.

Period should remain constant to prevent issues. A clock has periodicity: it repeats itself with a period t, which is the time between each rising clock edge.

- The point where the clock rises from 0 to 1 is called the rising clock edge
- The point where the clock falls from 1 to 0 is called the falling clock edge

$$\text{Frequency} = \frac{1}{\text{periodicity}}$$

Frequency is measured in hertz.

8.4 The D-Latch

Motivation. We want to:

- Store state
- Control it
- Decouple it – state changes should be decoupled from what the state should be.

Start with the S-R latch (abstracted):

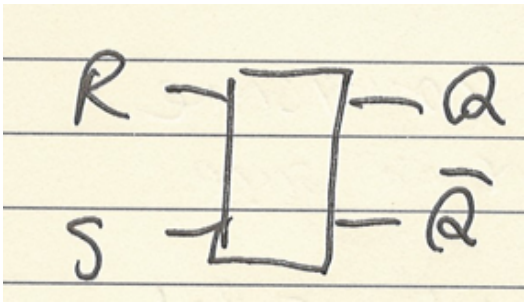


Figure 16: SR latch with inner circuitry hidden.

Connect D and CLK to some circuitry, which then connects to the S-R latch:

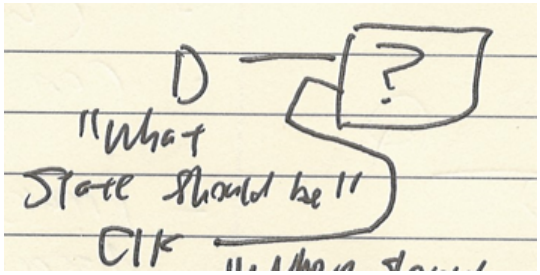


Figure 17: D and CLK connected to some circuitry, which connects to the SR latch.

How do we determine the circuitry? When should $Q = D$?

- If CLK is high, then $Q = D$ and the latch is **transparent**.
- If CLK is low, then retain Q's state: $Q_{\text{prev}} = Q$ and the latch is **opaque**.

We implement the circuitry. The D-latch is presented here:

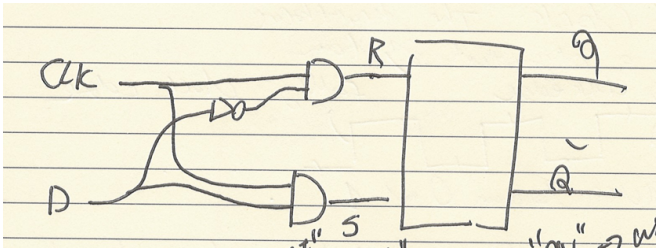


Figure 18: D-latch in full

Clock is ON $\Rightarrow$ window for D to change.

The truth table goes as follows:

Table 7: D-latch truth table. The input variables are CLK and D.

CLK	D	R "reset"	S "set"	Q	Comments
0	0	0	0	Q_{prev}	Q doesn't change
0	1	0	0	Q_{prev}	Q doesn't change
1	0	1	0	0	Reason for Q to change
1	1	0	1	1	Reason for Q to change

Unfortunately, the D-latch always updates while $CLK = 1$. What problems could this bring?

8.5 The D Flip-Flop

I want $Q = D$ for a split second.

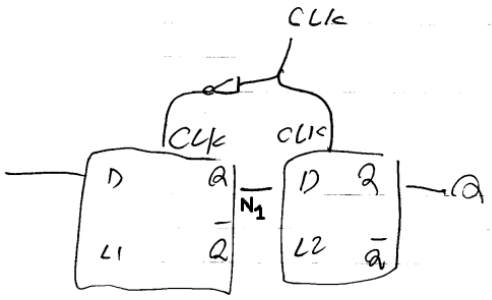


Figure 19: A D-flip flop. The master latch is on the left, and the slave latch is on the right.

The D flip-flop has two inputs and two outputs. It re-uses two D latches.

Table 8: D flip-flop table

CLK	Master (left)	Slave (right)
0	Transparent $N_1 = D$	Opaque $Q = Q_{\text{prev}}$

Table 8: D flip-flop table

CLK	Master (left)	Slave (right)
1	Opaque $N_1 = D_{\text{prev}}$	Transparent $Q = N_1 = D_{\text{prev}}$

Assuming that D and CLK never change at the same time, when CLK changes from 0 to 1, it "captures" what D is at, and sends it to Q. Q will remain the same until CLK goes from 0 to 1 again.

> **MEMORIZE ME:** A D flip-flop copies D to Q on the rising edge of the clock and remembers its state at all other times.

The condensed version of a D flip-flop looks like this:

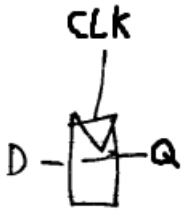


Figure 20: D flip-flop compacted

8.6 Other flip-flops

The enable input: A way to control whether the D flip-flop should ignore CLK; useful if you want to load a new value only sometimes. $CLK_{\text{filtered}} = CLK \cdot EN$

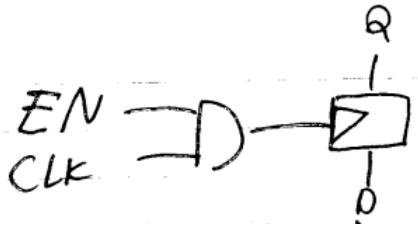


Figure 21: The enable input

The reset input: When(ever) reset is 1, the flip-flop resets to 0. Useful when you're turning on a system.

8.7 Input waveforms: D-latch vs. D flip-flops

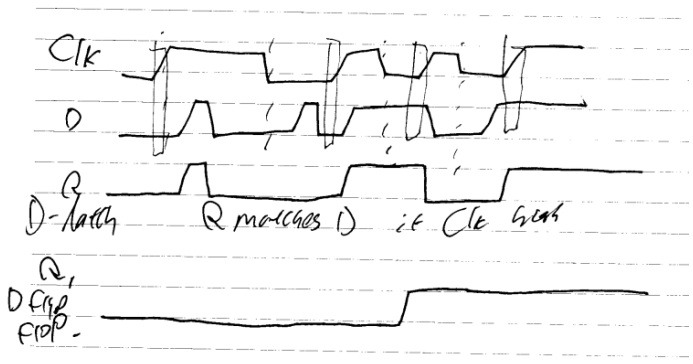


Figure 22: Waveforms for CLK, D, and Q if CLK and D connect to a D-latch, and Q if they are connected to a D flip-flop.

- For a D-latch, Q just copies D whenever CLK is high, and I extend the line the moment CLK goes low until it goes high again, then I copy D again, and so on (look at the dashed lines).
- For the D-flip flop, I look at all points where CLK goes from 0 to 1, and update Q to match where D is at (look for the outlined rectangles).

8.8 Synchronous Logic Design

Astable circuits are any circuit that can't be stable. For example, a three-inverter loop. There's no stability to this circuit, creating a ring oscillator. **Avoid this.**

To avoid this from happening, we **break cycles with register.** The simplest register is m a 1-bit flip-flop.

- Every circuit is either a register or a combinational circuit. For example, a D-flip flop is a register. There may be 64 bits coming in and out of the D-flip flop.
- At least one circuit element is a register
- All registers receive the same clock signal
- Every cyclic path contains at least one register. This means a cycle might look like a CL $\rightarrow$ D-latch and back again.

Synchronous circuits' state changes with the clock; async circuits state can change independent of the clock.

8.9 Timing constraints

Setup time: The input to D should be stable for some time before the clock edge.

Hold time: The input to D should be stable for some time immediately after the clock edge.

Inputs should not be changing at the same time as the rising clock edge.

This is necessary for circuits to work.

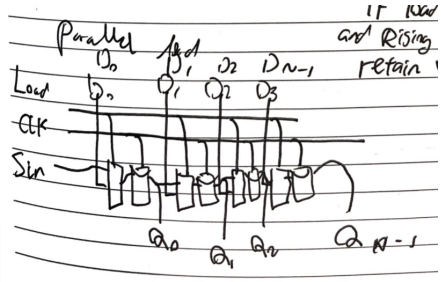
Lowest clock period: The clock period cannot be shorter than:

1. The longest propagation delay between any two flip-flops
2. The setup time of the flip-flop

8.10 Shift registers

Shift registers bit shift to one direction (depending on how the circuit is built and depending on which direction is specified) by one unit on a rising clock edge.

Parallel loads have two states: where `load` is 1 and `load` is 0. If `load` is 1, then you can assign values directly into the outputs Q_0 to Q_{N-1}, which save on the next rising clock edge. If `load` is 0, then on each rising clock edge, all bits shift: $Q_n = Q_{n-1}$.

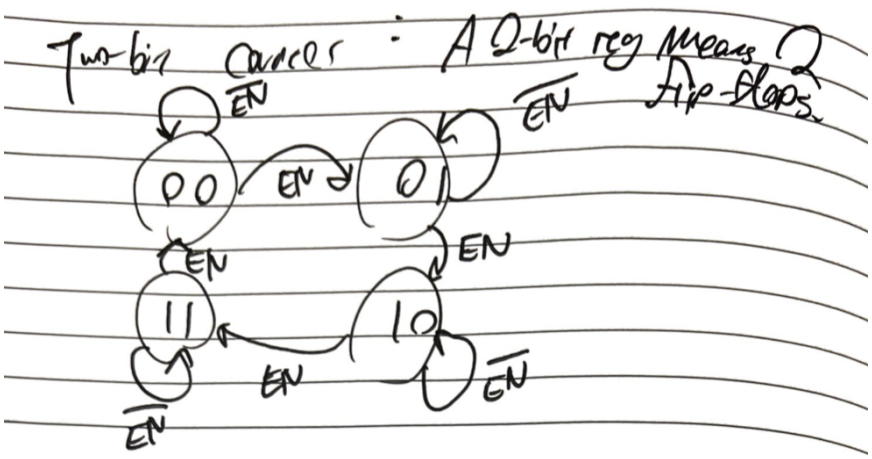


9 Finite state machines

A machine that can be in and transition to a finite number of states at a time.

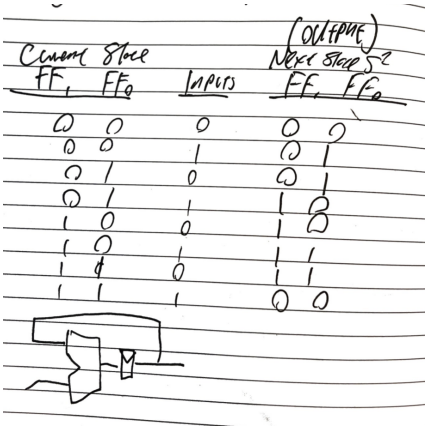
9.1 State transition diagrams

A state diagram looks like this:



This is a state diagram for a two-bit counter. On each rising clock edge, one state may transition to the next state depending on some other factor. For this specific diagram, EN must be on for the state to transition. This diagram also states the only possible states, given an existing state, that it can transform to.

9.2 State transition tables



The inputs are the current state and any input that may affect what the next state is. The output is the next state. The reason why we use state transition tables is that we can easily derive a sum of products table for each possible next state.

9.3 The looks of a Moore FSM

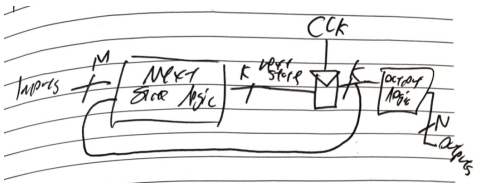


Figure 23: A Moore FSM schematic

The Moore FSM goes like this:

$$S \mapsto \text{output}$$
$$(S_{\text{cur}},\ \text{Inputs}) \mapsto (S_{\text{next}})$$

The **state register** contains the current state. After the next rising clock edge, it contains the next state.

In **combinational logic** inside FSMs:

- The **next state logic** computes the next state
- The **output logic** computes the output based on the current state, which happens to be its inputs. The output logic does not determine the next state. The output logic could be an encoder.

9.4 To make a Moore FSM

1. Determine inputs and outputs
2. Sketch the state transition diagram, using words as states
3. Derive state transition tables
4. Decide state encoding (how each state should be represented, most likely in binary). *Not dependent on step 3.*
5. Re-do the table done in step 3 with binary encoding
6. Make the output table (a map of each state to the outputs decided by the encoder). *Not dependent on step 5.*

7. Derive Boolean equations for the next state based on the current state and any extra inputs, and all outputs based on current state.
8. Sketch the circuit schematic.

The state encoding table maps a state “state 0”, “state 1”, “state 2” to binary, meaning:

$$\{\text{state } 0,\ \text{state } 1,\ \ldots\} \mapsto \{\ldots 00,\ \ldots 01,\ \ldots 10,\ \ldots 11,\ \ldots\}$$

Each state has a meaning. For example, for traffic lights, $S0$ may mean green light. That’s output encoding:

Output	Encoding
Green	00
Yellow	01
Red	10

Our state transition table might look like this:

Current state S	Inputs	Next state S' (output)
…	…	…

And our output table will look like:

Current state S	Outputs
…	…

9.5 The Mealy FSM

The inputs and the state directly determine the outputs. In other words:

$$(S_0,\ A) \mapsto \left(S_0',\ Y\right)$$

Which updates on each rising clock edge.

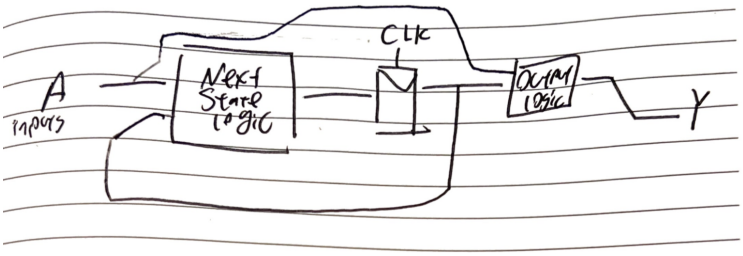


Figure 24: Mealy generic circuit

An example of an FSM constructed with the mealy method:

> A snail crawls over a tape with bits, and each bit may be 0 or 1. The snail smiles if the bit it is on is 1 and the bit behind it is 0.

The steps to designing a Mealy FSM is very similar to designing the Moore FSM, with one change step 5 and step 6 are combined. This means that the state transition and output tables with the encodings are joined.

Using a mealy FSM, this can be represented in two states:

- S_0: the snail is currently on a 1, or a reset was initiated
  - Remain on this state if input is 1. Outputs 0.
  - Move to S_1 if input is 0. Outputs 0.
- S_1: the snail is currently on a 0
  - Remain on this state if input is 0. Outputs 0.
  - Move to S_0 if input is 1. Outputs 1.

We can construct a truth table from this:

Table 12: Truth table for the electric snail

Current state S_x	Input A	Next state S'_x	Output Y
0	0	1	0
0	1	0	0
1	0	1	0
1	1	0	1

We can derive Boolean equations for S'_x and Y:

$$S'_x = \overline{A}$$
$$Y = S_x A$$

We can use this to construct the circuit that represents this.

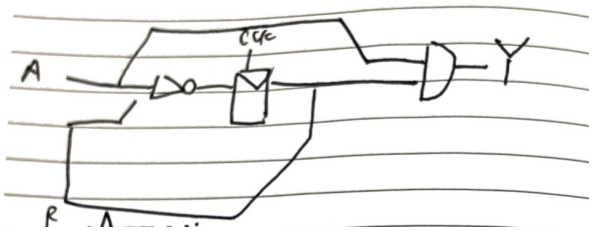


Figure 25: Electric snail circuit. Note that the wire at the bottom doesn't actually do anything

9.6 Factoring state machines

Complex FSMs can be broken down into multiple simpler FSMs where the outputs of some of the smaller FSMs are inputs to other FSMs. This is known as the factoring of state machines.

10 Digital Building Blocks

The ALU stands for the arithmetic/logical unit. It's a combinational circuit that combines mathematical and logical operations. It's like a wildcard – we can choose what operation we want it to do, and it'll follow it, given that it was implemented. That is, it takes A and B and does an operation determined by F.

We can implement an ALU by segregating all of the operations it could do. Or we could combine them, reducing the space it can take. An ALU that has 8 different operations may not always need a multiplexer that takes 2 bits.

10.1 Operations done by ALUs

- Shifting (logical: 0 push, arithmetic: preserve sign)
- Rotating (wrap-around)
- Multiplication

- Dividing

They may output:

- An overflow flag (V)
- A carry out flag
- A negative or zero flag

10.2 Representing Shifts

To represent a logical shift by k bits, we may say:

- X LSL $k = Y$

The same applies for all other shifts:

- LSR (logical right shift)
- ASL (arithmetic left shift)
- ASR (arithmetic right shift)
- ROR (rotate right)
- ROL (rotate left)

10.3 The Datapath vs. The Control Unit

Where are the inputs to the ALU (denoted as $A,\ B$) coming from? A register? Below the data on the clock edge? A circuit? A user? Memory? Who is controlling F?

The **Datapath** is where the computation happens. It operates on words of data, and mainly involves an ALU, register, and multiplexers. In particular, I may need to use multiplexers to control ALU's inputs and outputs.

The **Control Unit** dictates the actions that take place on the Datapath, and it is mainly done with an FSM.

I can use ALUs as alternatives to adders and multipliers if I want to build up complex operations. For example, attempting to replicate $x^2 + 2x$ without using adders and multipliers not inside an ALU.

10.4 Complex Operations With Only One ALU

You can always relay the output of the ALU back to the register that feeds into the ALU. Combine that with an FSM to do whatever you desire.

11 Memory

Problem. I want to retain data. The issue is that I don't have enough registers. They take up a lot of space, and moreover, they are volatile. Anyways, here's the ways memory can be stored:

- CPU
  - Registers, perhaps. The ALU can access this right away.
- Cache
  - To retain data in chip, as it takes way too long to search in the main memory. This is as if you're carrying a cart of books out of the library. Your cart can't store the entire library.
- Main memory
  - The library.
- Hard disk
  - What is larger than a library? Likely not volatile.

The lower we go in the list, the more each element can store (the hard disk stores the most), but the further it is from the CPU.

If I'm requesting data from the cache, I'm requesting 32 bits, **given** my OS is 32 bits. The further the data, the more time it takes.

11.1 Interface to Memory Array

- Memory is a row of bits. Each row has n bits, which we can see as a word.
- Each value has addresses
- The size is depth $\times$ width
- If I have 32 registers, I need 5 bits to distinguish between them.

For example:

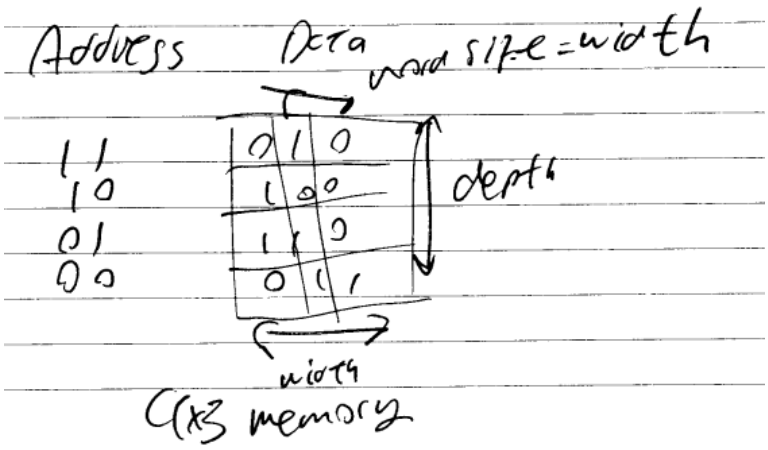


Figure 26: 4-by-3 memory.

I have the ability **read** and **write** to the memory array: here's the supposed interfaces for them:

- `read(address)`
- `write(address, info)`

Each **bit cell** stores 1 bit of data. They differ for SRAM, DRAM, and ROM. When we read from memory, we often read from a word. We select which word we want to read, and it gets output.

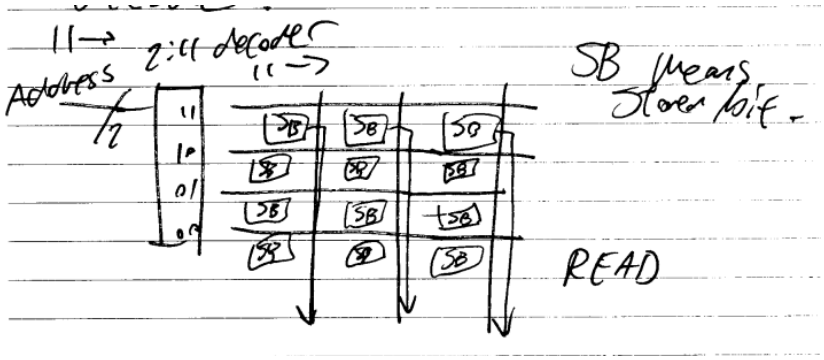


Figure 27: Read only memory. I choose which cell to access.

We can also write to them.

11.2 Multi-Ported Memory And Register Files

Read port count: How many different values I can read at a time. Inputs: Who to read (A_1, A_2). Outputs: $RD_1, RD_2, \ldots$

Write port: Changing data in memory. Typical of register file. Inputs: what to write WD_3, whether write is enabled (WE_3), and who gets written to (AD_3). Mutates, but outputs nothing.

Here's an example register file:

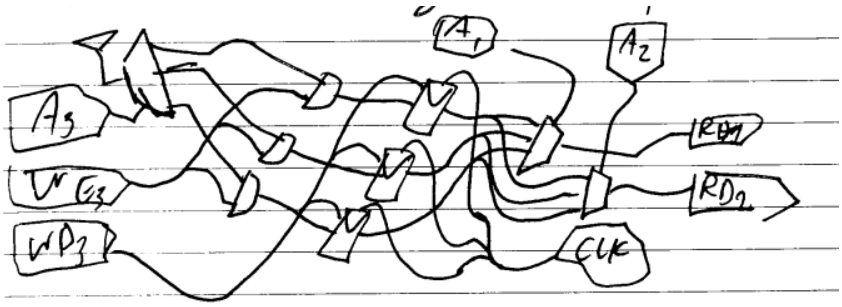


Figure 28: A register file.

- WD_3: what I want to write
- WE_3: acts as the write enable
- A_3: chooses what the write enable may target
- A_1: decides what register output the first output pin takes
- A_2: that, but targeting the second output pin

On the same rising clock edge, I can

- Read **two** (both) values, for the figure above
- Write one value (bit) at a time (there are three bits that can be stored)

When the clock rises, and write & read enable is on, the value is read *after* updates occur.

11.3 Registers vs. Memory

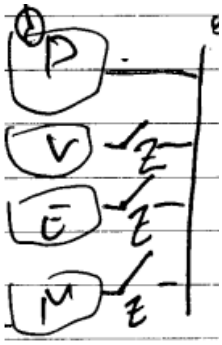
Registers and memory:

- Store values
- Use addresses... but the addresses are different!
- Registers have a small capacity, but memory can store a lot more.
- Registers are close to the ALU, and memory is far from the ALU.
- Registers have immediate access time, whilst memory requires very long access times, relatively.

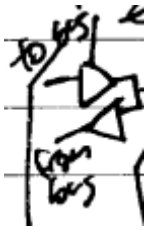
Caches are somewhere in between registers and memory.

11.4 Tristate Buffers

Recall Z s. Here's a shared bus:



Only one chip can drive the bus at a time.



To bus is used in a writing context. *From bus* is used in a reading context. This is useful for a processor to communicate with main memory.

- If a row is not being read, output Z.

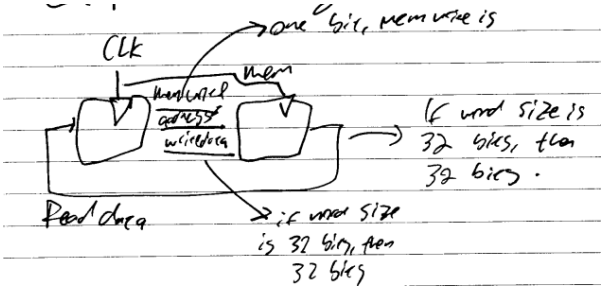
- If a row is being read, output the value.
- If a row is being written to, output Z.

11.5 The Processor-Memory Interface

Data comes from memory, and the processor must explicitly:

- Request data from the memory (load)
- Upload data in memory (store)

The load-store architecture looks like this:



To the memory, it gets

- Mem write, which is one bit representing the memory value
- Address. The larger the memory the larger the bit size of the address I need.
- Write data: same as the word size. In this example, it's 32 bits.

11.6 Comparing Between Different Types of Memory

DRAM. It takes, for one bit:

- 1 capacitor
- 1 transistor

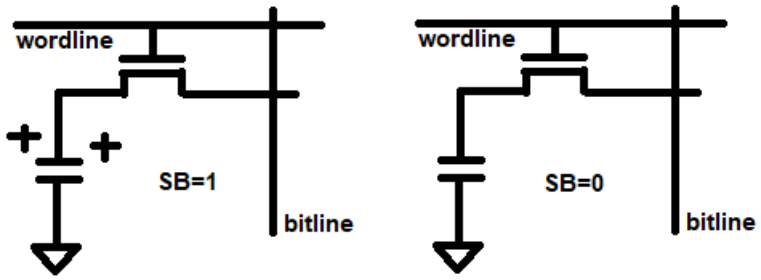


Figure 29: DRAM.

When DRAM is:

- 0, the capacitor is not charged.
- 1, the capacitor is charged.

DRAM contents must be refreshed every few milliseconds due to capacitor charge leaks. Regardless, it **is** cheaper. The capacitor node is dynamic, as it isn't being driven HIGH or LOW by V_{DD} or GND.

Data is written by data being transferred from the bit line to the capacitor. Every time DRAM is read, its bit value is destroyed, so it must be written again.

SRAM. More expensive, cross-coupled, and inverters are bistable.

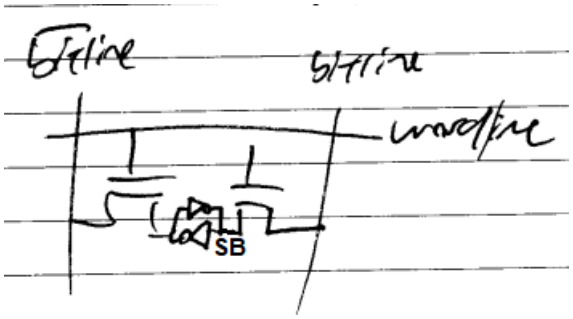


Figure 30: SRAM. SB is the stored bit, on the *right* side.

No refreshes are required, as inverters restore the stored value if noise degrades it.

This thing costs **6 transistors.**

11.7 Estimating Costs

Transistors can be used to estimate the cost of something. DRAM consumes less power and area, but it is **slow.**

The number of transistors is a proxy for:

- Area
- Power
- $$ Cost

Yet:

- Flip flops are fast but expensive (in terms of latency). I don't have a lot of it, so I should only use them when necessary.
- SRAMs are in the middle
- DRAM is slow but cheap

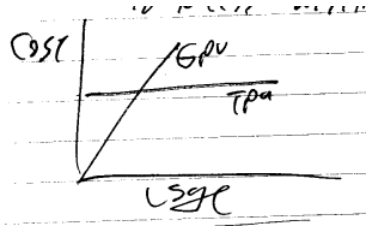
Rule of thumb: the larger the memory, the slower it is. I would put hard drives right on the bottom of this list, right below solid-state drives.

11.8 General Purpose Processors and Optimized Processors

May not be tested on.

A general-purpose processor can do whatever I want. The problem is that it is not optimized. If there's an operation I want to do very commonly, I can design hardware optimized for them. They're becoming more relevant today, and now, Google uses optimized processors, which they call TPU (tensor processor unit).

The GPU is great at calculating what colors pixels on a screen should be. GPGPUs stand for general purpose GPUs and are mainly used for mining bitcoin. CPUs are not that. Hence, Google designed TPU so it could do what they wanted, quickly, such as matrix multiplication.



11.9 In Summary

- ALUs differ on processor
- ALUs are the main part of the Datapath, controlled by an FSM
- There are different kinds of memory in different parts of a computer system. They may be close, or they may be far.
- Read-only memory is not volatile... and that's all for read-only memory.

12 How to Assembly

Creating an entire program is painful using machine code. There's a better way to do this, using the assembly language. It is a way of expressing the machine code instructions in a more human-readable way. All of these things we're doing is providing instructions to the processor so it knows what to do for every clock cycle.

12.1 Programming the processor

How do instructions work such that the processor knows what we want to do? Firstly, we need to understand how machine code works.

The ultimate goal: we need data path signals to go to the processor so that the processor knows what to do.

- The control unit
  - Produces datapath signals
  - Gets the op code
  - Which is extracted from the machine code
  - Which comes from the assembly language

But where do the instructions come from? How are they provided to the instruction memory? If I have an operation, could you figure out how the 32-bit machine code would look like?

12.2 Adding two numbers together

Example: replicate `C = A + B`. Assume that `A` is stored in `$t1`, `B` in `$t2`, and `C` in `$t3`. The assembly instructions go by:

```
add $t3, $t1, $t2
#   ^    ^    ^
# destination
```

Add `$t1` *and* `$t2` *and stores it in* `$t3` *– the locations, that is.* Registers are always referred starting with a $ – it indicates that "this is a register." Machine code instruction:

```
000000 01001 01010 01011 XXXXX 100000
```

This is parsable by the instruction unit.

Note: There's a one-to-one mapping for all assembly code and machine code instructions!

How do we understand the machine code to be able to know what kind of instructions assembly needs to express? What does the process need? It is made of many different parts.

- The control unit needs the op code, otherwise it won't know what FSM branch to go to.
- We need to know what registers are being used in the operations. It must provide instructions pertaining to registers.
- Other information that might be needed to make the operation to happen (immediate or shift values)

When we write or interpret a machine code instruction, we need to know how to encode or decode these details into 32 bits.

12.3 Register-type operation

An R-type operation involves

- an operation (formed by op code)
- the two registers going into the ALU, as input
- what register gets the output, which is overwritten

We have an op code for all R-type operations. What needs to be done is that the instruction set needs to tell the ALU what operation is being done.

Recall our machine code instruction:

1. `000000 01001 01010 01011 XXXXX 100000`
2. Descriptions, in order from left to right, for an R-type.
3. `000000` – always 0s, to represent that this is an **R-type** operation
4. **(sources go first)** `01001` – references `$t1`, source 2
5. `01011` – references `$t2`, source 2
6. `01001` – the address of the register the result is stored (Destination)
7. XXXXX – only relevant for shifts, so maybe make these zeroes
8. `100000` – I want to add (not subtract, multiply, or divide) – what operation should an ALU do?

12.4 What's up with the register locations?

When doing things in assembly, we're creating things that are more intuitive. It is better to type `add` than `000000 ... 100000`. We don't want to think about the binary values of what we are doing.

For those who made the assembly language, labels are used to indicate functions. The `add` instruction is one of the many operations that processes two register values and stores the result in the third.

12.5 Label names for registers

There are **32 registers**. We can refer to them using dollar-sign notation. When making our program, we refer to the labels.

- `$0` or `$zero` – value of 0, always. The first value is always going to have a register value of 0, and it will not be set to anything else. We need a source of 0 in all our programs.
- `$8` to `$15` and `$24` to `$25`, A.K.A. `$t0` to `$t9` – these are temporary values, there are 10. They are like scratch paper when performing calculations. These values are meant to be trashed.
- `$16` to `$23` A.K.A. `$s0` to `$s7`: these are saved temporaries that we may need later.
- Registers `$2` and `$3` are return values. A.K.A. `$v0` and `$v1`.

- Registers `$4` to `$7` A.K.A. `$a0` to `$a3` are function arguments. When the program is invoked it will look at these four registers.
- Register `$1` is reserved for the assembler.
- Registers `$gp`, `$sp`, `$fp` and `$ra` are really important. `$ra` is the return address. It is going to be the thing that stores the return address of whatever function called you.
- `$sp` is the stack pointer. It stores where the stack is. *Don't use this as a variable you're operating on.*
- Also, three special registers `PC`, `HI`, `LO` that are not directly accessible.
- `HI` and `LO` are used in multiplication and division and have special instructions for accessing them.

12.6 I-type operations

I-type instructions also operate on registers, but they involve a constant value as well. The constant is encoded in the last 16 bits of the instruction. For instance:

```
addi $t2, $t1, 42
```

We're taking one of the registers `$t1`, adding the number 42, and storing it in `$t2`. Here's the machine code:

```
001000 01001 01010 0000000000101010
```

Described:

- `001000` – this is an `addi` operation.
- `01001` – the source `$t1`
- `01010` – the destination `$t2`
- `0000000000101010` – encodes the number 42. 16 bit limit, so it's sign extended.

Anything where a constant is involved is an `i`-type, because `i`, an immediate value, is a way that instructions represent constants.

There's another type of instruction:

12.7 J-type instructions

Tells me I want to jump somewhere. J-type instructions jump to a location in memory encoded by the last 26 bits of the instruction, everything but the opcode.

The location is stored as a label, which is resolved when the assembly program is compiled. For instance:

```
j main
```

Translates to

```
000010 00000000000000011000101010
^ it's a jump | ^ where is main?
```

When the assembler works and converts everything to code, the assembler deals with converting `main` to the extremely long sequence of binary digits.

12.8 How are the operations broken down?

R-type

- `opcode` – 6 long, `000000`
- `rs` – source register 1, 5 long
- `rt` – source register 2, 5 long
- `rd` – destination register - 5 long
- `shamt` – shift - 5 long
- `funct` – am I adding? subtracting? multiplying? – 6 long

I-type:

- `opcode` – what is the type of operation being performed?
- `rs` – a source, 5 long
- `rt` – another source? a destination? that depends. 5 long.
- `immediate` – 16 long – the “constant”

J-type:

- `opcode` to tell that it's a jump operation, one of two types (6 long)
- `address` – 26 long, the address to jump to

These are the three types of operations that your processor can do. When you're doing these things:

- If it's an R-type, the first 6 bits are always 0
- You'll see the table with the 6-bit values that will tell you what they do if it's not an R-type. Otherwise, look at the end to find the opcode for R-type operations only.
- If you see "don't cares" as `X`s, the assembly language interpreted always assigns them to some value, like `0`.
- No one has time to program a processor with machine code.

12.9 Assembly language

AMD will have a different assembly language set than Intel, because their architecture is different.

MIPS is short for Microprocessor without Interlocked Pipeline stages. It provides a set of simple and fast instructions. Compiler translates instructions into 32-bit instructions for instruction memory. Complex instructions are built out of simple ones by the compiler and the assemble.

It is a type of RISC (reduced instruction set computer) architecture.

Why do we have unsigned operators? Because of overflow detection.

- Note that **SE** stands for "sign extend"

I-type and J-type use the first six bits to express the opcode; R-type makes the first six bits zero. The last six bits go to the ALU.

In **assembly**, in operations that have a destination, the **destination** goes first, then the sources. We may expect to see the following patterns:

- destination, source, source
- destination, source

- source, source (no destination. There may be a specific destination that is readable, but you shouldn't explicitly write to it)

In **machine code**, the sources go before the destination.

12.10 Divide and multiply?

You know multiply is very overflow prone. `hi` and `lo` refer to the high and low bits, which are registers. In multiply, the result is stored in `hi:low`. This is convoluted but there's no other way to handle multiplication. Do whatever you want afterwards.

For division, `lo = $s / $t` and `hi = $s % $t`. If you need a remainder, that's how you get it. Want a real number? Use floating point numbers. We won't be dealing with them.

13 Assembly Language Instructions

When you create an assembly language instruction:

- start with an operation `$t3 = $t1 + $t2`
- so I would like to `add $t3, $t1, $t2` in assembly (destination, source, source)
- in machine code, I need to figure out
  - opcode, which is `100000`
  - syntax is `$d, $s, $t`
  - The machine code will look like `opcode rs rt rd shamt funct`
  - So, I get the following as machine code: `000000 01001($t1) 01010($t2) 01011($t3) [000000] 100000`
  - Note that the destination will always be positioned more to the right.

This is the R-type way of breaking instructions down.

13.1 R-type vs. I-type arithmetic

I-type helps you deal with constants. The way they are translated into machine code is different. No, you cannot multiply and divide constants without being clever – this is by RISC architecture – I don't want to cram my processor with things that can easily be done by other things.

- I want to do the operation `$t2 = $t1 + 42`
- In assembly, that is `addi $t2, $t1, 42`

- The instruction looks like `opcode rs rt immediate`
- Which is `001000 01001($t1) 01010($t2) [42 sign extended]`
- A criss cross is needed because **the destination is always on the right of all arguments that references addresses.** That is, for machine code. Processors understand them better.

13.2 Logical operations

Literally the keywords: `and <..> <..>` or `andi` for the i-type counterpart.

When using the `i`-type counterpart, it **will** be zero-extended to match the length of the other thing to compare.

Note that ZE means zero extend (unlike sign extend). Note that if you are performing a logical operation on multiple bits, you will perform them on all corresponding bits. For example, `[32] AND [32] = [32]`. Sounds like a `zip()` in Python.

13.3 Shift instructions

`s_l` means logical, and `s_a` means arithmetic. The `v` denotes a variable number of bits, specified by `$s`. No `i` types, because that was a waste of space – `sll` and `sra` (the non-Vs) are the `i`-types. Replace the blanks `_` with `l` or `r` which means left and right respectively.

`a` is the shift amount and is stored in `shamt` when encoding the R-type machine code instructions.

13.3.1 Shift instructions

- `<<` shift left this much
- `>>` arithmetic shift right this much
- `>>>` logical shift right this much

13.4 Data movement instructions

Getters and setters for `hi` *and* `lo`*. Which is* `mfhi, mflo, mthi, mtlo`

`mf[...]` means move from (getter), and `mt[...]` means move to (setter).

13.5 ALUs and Control Flow

Most ALU instructions are R-type instructions. The only exceptions are the I-type instructions. Note that not all R-type instructions have an I-type equivalent, as RISC architectures dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations. For example: `addi + div -> divi`

13.6 Example program

Fibonacci sequence: how do we convert this into assembly?

```
int fib(void){
    int n = 10;
    int f1 = 1, f2 = -1;
    while (n!= 0){
        f1 = f1 + f2;
        f2 = f1 - f2;
        n = n - 1;
    }
    return f1;
}
```

In assembly:

`addi` is used to initialize values. We use `$zero` as the constant 0. That's how we initialize them.

Then, we enter the loop. We do `add $t4, $t4, $t5` and `sub $t5, $t4, $t5`, and we decrement `n`: `addi $t3, $t3, -1`.

There's not a lot of structure here. Now, what is `beq`? See the labels? Use labels to indicate that a line has a name. `FIB:` is the start of the function call, `LOOP:` is the start of the loop, and `END:` is the end of the loop.

`beq` means branch if equal. Test to see if the first two arguments are equal, and if so, jump to the label specified in the last argument. `beq $t3, $zero, END`

`j LOOP` means jump to the label `LOOP`.

`sb` means store byte. Store it somewhere in memory.

13.7 To make an assembly program

Assembly language programs typically have structure similar to simple Python or C programs:

- They set aside registers to store data
- They have sections of instructions that manipulate this data
- It is always good to decide at the start which registers will be used for what purpose!

13.8 To Simulate MIPS A.K.A. MARS.

CONTROL FLOW:

- Not all programs follow a linear set of instructions.
  - If/else, for/while
- We use labels to indicate the points that the program flow might need to jump no.

13.9 Jumps

`j`: the opcode is `000010`, and the syntax needs a label.

`jal`: jumps, executes, and comes back. The register `$31` stores the address that's used when returning from a subroutine (i.e. the next instruction to run). For example, `$31 = pc + 4; pc = ...` This stands for jump and link.

Note: `jr` and `jalr` are jumps, but are not J-type instructions.

13.9.1 Limits of `j`

For `j` and `jal` instructions, the address is supplied by the instruction. This is a potential problem: if the first six bits are occupied by the opcode, the remaining bits aren't enough for a full 32-bit address! How do we get around this?

14 Control flow instructions

I'm not just going to execute each instruction in sequence. Maybe our next instruction is further back in our code. Those are all control flow.

If I get to this point, I want to jump to a line. How do I do that?

We have labels on the left-hand side that indicate the points that the program might want to jump to.

`jal` is used for function calls. Once you finish executing, you use `jr` which is a return.

What you do: take the program counter + 4, and store it into `$31`. When done, restore the program counter back. `jr` means jump to a register. Takes the contents of a register and dumps it into the program counter. It taxes in the parameter `ra` (`$s`) the return address. Once you're done, say `Jr` to `RA`, and it'll say whatever was on `RA` was the previous program counter value – put it back to the program counter and take it back to the calling program so I can continue executing. (note that `$31` is `$ra`).

`jalr` means jump to some register location and store where I came from if I want to come back. It will never be used for this course.

Can I nest functions (can I jump more than once)? Is there only one register? The answer, is **yes.** Nested functions cause some issues. Every nested function call overwrites `RA` repeatedly, and not that when you come back it restores `RA` to its previous value. Whenever you're doing a nested function call, you put `RA` in somewhere safe. That is when we'll need a stack so we can store values in function calls.

What is the `+4`? When you call `jal`, your program call has a certain value. When you add `4`, when I jump back it goes to the line after.

14.1 Jump to register

For this instruction `jr $ra`, the processor moves the address stored in `$ra` into the program counter.

The next instruction to be fetched will be at this new address, and the program will continue from there.

What happens in other cases, when the destination address is stored in the instruction?

For `j` and `jal` instructions, the address is supplied by the instruction. The issue is that our address can only be 26 bits. We can't fit a 32-bit address!!! How do we solve this? We can only do messy solutions.

Trailing zeroes: since jump instructions load new addresses into the program counter, the values being loaded must be divisible by 4 (all addresses are multiples of 4). Use the 26 bits to store all the other bits except for the least two significant bits. Great. Now we have a 28-bit address. Use lading bits? What should we use for the first 4 bits?

- One that MIPS uses is to keep the first 4 bits of the previous PC value. This is where this formula comes in:
- `pc = (pc & 0xF0000000) | (i << 2)`

14.2 Branch instructions

How does a branch instruction work? If-else statements.

- `beq` if the two provided registers are equal, go there otherwise go to next.
- `bgtz` – branch if > 0
- `blez` – branch if ≤ 0
- `bne` – branch if $\neq$

Branch is an I-type. It stores an immediate value. It says how far from the current program counter value where you need to go. Note that if the PC is incremented first, `i = (label location - (current PC)) >> 2`

If the branch offset is calculated first, `i = (label location - (current PC + 4)) >> 2`

The branch is the opposite of the `if`: skip to this if condition met. Flip it if you want to view it as an `if`.

Maybe branch to the else branch first? `bne, __ ELSE __ j END ELSE: __ END: __`

14.3 Multiple if conditions

Branch statement for each condition.

```
main: beq ___, IF
      bne ___, ELSE
IF:   ___
      j END
ELSE: ___
END: ____
```

For ANDs:

```
main: bne COND1, ELSE
      bne COND2, ELSE
IF:   ____
...
```

This short circuits.

While loops look similar:

```
START: beq STOPCOND, END
       ____
       j START
END: ___
```

For loops / while loops are equivalent.

We can't tell the difference between a for and while loop.

14.4 Interacting with memory

All previous instruction performs operations on registers and immediate values. What about memory? We want to load or store into them,

`lw $t0, 12($s0)` ← register storing address of data value in memory

Storing: DEFAULT we're storing signed. `u` for unsigned. Because we may sign extend or just add zeros to the start.

15 Loading and storing instructions

Memory is a long stream of spaces. I can load a word, byte, or full word, In the cases where I'm loading a byte or full word, I need to state whether the byte / whatever will be used in a signed context. By default, we will think of a byte as a number, but if it is just a bunch of 1s and 0s, then we can just say unsigned.

15.1 Memory instructions in MIPS assembly

`[(l)OAD or (s)TORE] [(b)YTE, (h)ALF-WORD, (w)ORD], [DESTINATION OR SOURCE REGISTER] [LOCATION IN MEMORY]`

The location in memory is `i($s)`, as `Mem[$s + SE(i)]` (remember that `SE` stands for sign-extend)

15.2 Alignment requirements

When I am putting things into or from memory, the amount of space I'm loading and storing can determine what addresses you can load from and store.

Word addresses must be divisible by 4 and are typically addresses specified in a `lw` or `sw` instruction. This ensures **word-aligning.**

Half word addresses should only involve half-word aligned addresses, (i.e even addresses)

No constraints for byte access.

(A word is four bytes or 32 bits)

Failing to abide by memory alignment will result in runtime errors. (Address error exception, and could cause the computer to BSOD).

How are the bytes within a word or half-word stored?

15.3 Little endian vs. Big endian

Let's say we want to read a word (4 bytes) starting from address X. How do we assemble these multiple bytes into a larger data type?

Big endian: `byte A, byte B, byte C, byte D (least significant)`

Little endian: other way around.

15.3.1 Big endian

MSB is stored first

15.3.2 Little endian

LSB is stored first

MIPS processors are big-endian.

15.4 Reading from devices

If I have a processor that has the ability to connect to many different devices without having way too many wire coming from each processor?

Memory can be used to communicate with outside devices, such as keyboard and monitors. This is known as memory-mapped IO, invoked with a trap or `syscall` function.

When I display something on screen, there's an address on memory which corresponds to the display on the screen. It is one long column that gets folded automatically based on screen resolution.

15.5 Trap instructions

`trap` $\leftrightarrow$ `011010` (the syntax is `i`). Trap instructions send system calls to the operating system. It is similar but not quite the same as the `syscall` command.

15.6 Arrays

Arrays are stored in consecutive locations in memory. The address of the array is the address of the array's first element. To access element `i` of an array, use `i` to calculate the offset distance. Add that offset to the address of the first element to get the address of the `i`th element. The offset is $i\times$ the size of a single element.

```
.data
```

```
A:  .space 400  # array of 100 integer; 400 bytes
B:  .word 42:100  # array of 100 ints all =42
# note that each integer is 4 bytes long.

    .text
main: la $t8, A  # t8 holds address of array A
    la $t9, B  # $t9 holds address of array B
    addi $t0, $zero, $zero
    addi $t1, $zero, 100

LOOP: bge $t0, $t1, END  # exit loop when i >= 100

    sll $t2, $t0, 2  # $t2 = $t0 * 4 = i * 4 = offset
    # everytime the loop loops, this goes up by 4
    add $t3, $t8, $t4  # addr(A) + i*4 = addr(A[i])
    add $t4, $t9, $t2  # $t4 = addr(B) + i*4 = addr(B[i])
    addi $t5, $t5, 1  # $t5  = $t5 + 1 = B[i] + 1

UPDATE: addi $t0, $t0, 1  # i++
      j loop

END:  ...
```

Copies everything from `A` to `B`.

16 The call stack

`jal` is j-type and stores the address next to it in `$ra` a.k.a. `$31`. That will be where you would want to teleport to after the function call is complete.

`jr $ra` is equivalent to `return`. The `PC` will be set to the address in `RA`. How do we know what's in `$ra`? `$ra` was set by the most recent `jal` instruction (function call)!!

. . . but don't recklessly change it yourself. Now, how do we not lose `$ra`, like the case of a nested function?

The **stack** is a spot in memory used to store function values independent of the registers ,which can be overwritten easily. A special register stores the stack pointer which points to the last element pushed onto the top of the stack. For MIPS, the stack pointer is `$29` A.K.A. `$sp`. This holds the address of the last element pushed to the top of the stack.

We can push data like `$ra` onto the stack (which makes it grow) and pop data from the stack (which makes it shrink).

Stack overflow? That's the error that is thrown if a stack overflow occurs.

The stack pointer starts at the highest address (lower), and the stack pointer goes lower and lower as the stack grows. The stack uses LIFO order (last-in first-out) order, just like a physical stack.

16.0.1 When do you store values on the stack?

When I call a function and I want to preserve values from being overwritten `$ra`

Different `$ra` values will exist in layers on the stack for each function call.

When we pop an element from the stack, we move the stack pointer to the previous one.

We pop a word off the stack pointer: `lw $ra, 0($sp)` and move the stack pointer a word back `addi $sp, $sp, 4`.

To push values on the stack, we move the stack pointer a word `addi $sp, $sp, -4` and push a word onto the stack `sw $ra, 0($sp)`.

Pop means get the value from the top of the stack and reroute the stack pointer (obvious where).

Advice on stack? Any space you allocate to the stack, you should later deallocate. If you push items in a certain order, you should pop the items in the reverse order. When pushing more than one item onto the stack, allocate all the space at the start or allocate space as you go. The same applies for popping.

16.1 Passing function parameters / levels

Since functions have entry and exit points, they also need input and output parameters. In other languages, these parameters are assumed to be available at the start of the function. In assembly, you have to fetch those values from memory first before you can operate on them. Where do I look for these parameters?

Here are some calling conventions:

Registers 2 and 3 are return values: `$v0, $v1`

Registers 4-7 `$a0 to $a3` are function arguments

if your function has four arguments, you would use the `$a0` to `$a3` registers in that order. Any additional arguments would be pushed onto the stack. More common convention is to just push all arguments to the stack, but on an exam you'll be told what to do.

16.2 Calling convention

Caller is the function calling another function. The callee is the function being called.

We separate registers into the caller-saved registers `$t0 to $t9` and the callee saved registers `$s0 to $s7`. The general convention is that the `s` registers are still going to retain their values the moment the function returns (unlike `t`, which we can mess up however we like). This is convention and is not exhaustive.

Callers must save all temporaries before calling functions (to the stack).

As a callee, it is your responsibility to save the `s` registers and later restore time if it is going to modify any of them.

Functions necessitate the use of the stack to preserve variables and to communicate. One of the other extensions of the stack is not to only hold onto registers, we would see them as input or output parameters. Use the stack to pass variables into and out of a function.

16.3 During a function call

- Before the function is called:
  - Push registers onto the stack to preserve their values.
  - Pass our input parameters to the function I'm calling, push them on the stack.
- At the start of a subroutine:
  - Pop the input parameters from the stack and store it in the local registers.
  - Does its thing
  - Takes the return value and pushes that onto the stack before it calls jump return
- Coming back from a subroutine call:
  - Pop the return values from the stack
  - Pop the saved register values and restore them

Every time I call a function, these are the steps that are to be happening.

16.4 The recursive factorial function

$$n! = n \cdot (n-1)! \;\; 1 \text{ if } n = 0$$

Basic pseudocode:

- Base case `n == 0`
- Return 1
- Get factorial `n - 1`
- Store result in product
- Multiply product by `n`
- Store in result
- Return result

16.4.1 Steps to perform:

- Pop `x` of the stack
- Check if `x` is zero
  - If so, push `1` onto the stack and return to the calling program.
  - Otherwise, push `x-1` onto the stack and call factorial again
  - After recursive call, pop result off of stack and multiply that value by `x`
  - Push result onto the stack, and return to the calling program

16.5 Okay, so how do I actually call and how do I return?

No, the call stack is NOT a constant size per function. One function call might make me pile 1000 different things on the stack, and another one might only take 1.

I may put function calls on the stack. They may be in any order, but the most prominent (a.k.a. the ones I want to use first) ones should be placed on the top of the stack. I **could** use the argument registers if I wanted to.

Normally, the return value would replace what **was** on the top of the stack before the function was called. Never dig below where you are. This means if your function call starts with a stack pointer at `A`, **do not tamper anything below what the caller doesn't want you to mess up** (most of the time, nothing below if you only have one return value – who does multiple return values anyway).

17 Pseudo-Instructions

Things that are not part of the main assembly set. They're like macros and can be translated to series of instructions that are part of the main instruction set. For example:

- If `BGE0` exists, but `BLE0` does not in the instruction set, `BLE0` can be a pseudo instruction.
- The assembler often uses the special `$at` register (also as `$at`) when mapping pseudo-instructions to MIPS-instructions. Please don't use it in your programs.

Example: the `la` pseudo-instruction (**load address**) – loads an address from a label. For example:

- `la $d, label`
  - Loads a register `$d` with the memory address that `label` corresponds to.

These instructions expand to more lines, which are all in the MIPS architecture (has a machine code equivalence). Do I need to know them? No. But it's handy.

18 Interrupts

After I finish this course, I can

- try doing compilers
- operating systems

For that, you do need to know a bit about interrupts. When checking keyboard inputs – when checking to see that these happens, but the thing that they are updated in memory is something I may have not thought about. There seems to be things going in the background that aren't part of your program where memory is being updated without your specific involvement. What happens when someone presses the power key?

Things that are outside your program that are going to happen, your code or system has to respond to it.

Some of the system calls do this (sleep – turn things on, wait for timer to go, and resume)

These are thing that are going on underneath. Sometimes, they are useful, and sometimes they are going to disrupt the programs that you are trying to do.

Below: in both cases, something will happen. It'll tell you that an interrupt happens

18.1 Polled Interrupting

Goes to a section of memory (the interrupt-handling code: what interrupt has happened?) that will handle that part. For example, the keyboard handling code. I'll take a look at the key that was pressed, and I will branch to a different section. It has a section of code that gets invoked when an interrupt takes place, and it goes to somewhere to handle it (handler code sections).

MIPS uses this one.

18.2 Vector

Processor can branch to different address for each type of interrupt. Each exception address is separated by only one word. A jump instruction is placed at each of these addresses for the handler code for that exception.

18.3 Interrupt handling

In the case of polled interrupt handling, the processor jumps to the exception handler code, based on value in the cause register.

This will interrupt your final state machine. It will overwrite your states. The cause register tells you what level that is (you're in this state now, handle this interrupt, and decide if you want to come back). Like I'll jump and link to some section of memory and I'll decide if I want to link back.

If the original program can resume afterwards, this interrupt handler returns to the program by calling `rfe` instruction (return from exception). Otherwise, the stack contents are dumped, and execution will continue elsewhere.

Here are some types, the lower the number is the highest priority:

- 0 (INT): external interrupt
- 4 (ADDRL): address error exception; load or fetch
  - Happens when I specify an odd number address (You know this is a dumb idea, right?)
- 4 (ADDRS): address error exception; store
- 6 (IBUS): buss error on instruction fetch – a bad error
- 7 (DBUS): buss error on data fetch – a bad error
- 8 (`Syscall`): `Syscall` exception
- 9 (BKPT): breakpoint exceptions
- 12 (OVF): arithmetic overflow exception

Registers `$26` and `$27` are used by the exception handler, so don't use them.

18.4 Shutting down the computer

Pressing the power button does not cut of the electricity to your computer. It is like pressing a key that says, turn your computer off. It'll do it in a more controlled, meaningful way, so it goes to a section of code that will shut everything down. Only a hard reboot (press and hold the power button) cuts everything off.

At least, it will interrupt your program. Interrupt signals take priority of the code that is running right now.

19 Microarchitecture

Datapath:

- Does calculation

Interface to memory:

- Memory is a row of bits, each row has M bits.
- Address may not be memory address; address is used to identify a row

Register files:

- The one we've looked at has 2 read ports. The addresses are only 5 bits long, as our MIPS processor has 32 registers, so we only need 5 bits to distinguish them.
- We have a write port, the address, and the value to be written to.

The program counter:

- Not the program itself
- PC contains the address of the instruction
- In the beginning of time, PC contains the address of the first instruction
- As time continues, $PC = PC + 4$, or changes based on instruction

Data vs. text segment:

- The data segment is in a very far away region of memory.
- Unlike the text segment, but it may not start at the very start of memory.

Instruction formats:

- We have the R-type, I-type, and J-type instructions
- Most significant 6 bits tells us a lot
- For R-type and I-type instructions, the next 10 bits are `rs` and `rt`, each taking 5 bits.
  - This is useful when designing hardware, as I can split off the first six bits and use them; however, I need to.

19.1 The Single Cycle Processor

Note that for now, we can't support every instruction. Recall the ISA:

- IT tells programmers how to encode the instructions
  - Assembler or compiler (for a higher-level language) can produce machine code for an ISA.
  - The ISA tells hardware designers what they must support
    * A microarchitecture is one of many hardware implementations of a processor that supports an ISA

19.2 What does a processor do?

1. Fetch the next instruction from memory
2. Decode the instruction
3. Execute the instruction
4. Access memory (`sw`, `lw`)
5. Write back a result to the register file (write back)

19.3 What state do we need in the processor?

- A register
- A register file
- Memory to access instructions from the text segment (instruction memory). **Read only once loaded**
  - The OS is responsible for taking your program to put it into memory
- Memory to access data from the data segment. **Read and write is permitted,** as we do need to mutate what is in memory
- The program counter (a register that stores the program center). At the end of the cycle, I should be changing PC.

19.4 The Processor's Data And Control Unit

The image can be found in figure 7.11 of the textbook, which I will not be including here.

How does the processor, assuming no branches or jumps, access the next instruction?

- Program counter increments exactly by 4 if no jumps or branches occur, which is the next memory address for the next instruction
- Hence, when the instruction memory is read, it now reads the new PC location
- PC $\rightarrow$ Instr

What are the inputs to the control unit and why?

- Opcode and `funct`
- These inputs let the control unit know what we want to do

We may wish to annotate the Datapath to see what is being used, and why.

19.5 Outputs of the Control Units

The outputs and what they do:

- `MemToReg`
  - Tells me which value to write back (WB) to the `RegFile` from ALU or memory
- `MemWrite`
  - On if I want to write/store a register's value to the `.data` segment. When off, I can't write there.
- `Branch`
  - If this is on and the output of the ALU is 0, I'm likely branching. Otherwise, $PC = PC + 4$ and no branching would occur
  - The 0 flag is used for `beq` and its variants.
  - Explicitly, `Branch` indicates that it is a branch instruction.
- $\texttt{ALUControl}_{2:0}$
  - Tells me the operation the ALU should do
- `ALUSrc`
  - Most of the time, 1 if it's an immediate instruction, 0 otherwise (there are exceptions!)
  - Select register or immediate for R-type or I-type instructions
- `RegDst`
  - X if RegWrite is 0. Otherwise:
  - 1 if the third register in the machine code is being written to (destination is `rd`). 0 if the second register is being written to.
  - Is the destination register an R-type (1) or I-type (0)?
- `RegWrite`
  - Whether I should write to the register file. The program counter (PC) is not part of the register file.

- `PCSrc`
  - If I should jump. This has a double condition: whether my instruction could cause a branch, and if the condition for the branch is successful
  - Whether a branch has been taken

19.6 Outputs of Anything Else

The `ALUResult` is connected to data memory, as:

- Load instructions need `address = base address + offset` (such as `lw`, `sw`, and to accommodate its syntax). This does not apply to all instructions that start with an `l` or an `s`.

19.7 Analyzing The Single cycle Processor

Questions might look like:

- Highlight the Datapath being used
- What are the values of some control signals that are used when a specific given instruction is being executed?
  - Control signals are `MemToReg, MemWrite,` and so on.

For instance: what are the values of the control signals and the portions of the Datapath that are used when a(n) sw instruction is being executed?

- A(n) `sw` instruction looks like `sw $t0, 0($s0)`
  - It does not write to the register file
  - Hence, I do not need to use the write port of the register file. Hence, `RegWrite` is 0.
  - `RegDst` is X, as `RegWrite` is 0.
  - `MemtoReg` is X, as we are not fetching anything from the data memory
  - `Branch` is 0 because store word is not a branch instruction
  - `MemWrite` is 1 as we're writing to memory
  - $\texttt{ALUControl}_{2:0}$ should be adding (`010`)
  - `ALUSrc` is 1 so I select the immediate value as my second input to the ALU.

    * Note that the first input to the ALU has nothing to do with the control unit.

- A2 is the register where its value we want to store
- A1 is the register that holds the address
- The immediate in the instruction's 15:0 stores the offset of the address
- ALUSrc is set to 1, as it targets the immediate (signaling that I'm doing an I-type instruction)

Questions like these will end up with a table, one column for each output of the control unit.

We may also want to annotate the Datapath by highlighting it:

- My instruction comes in and something goes into A1, as I need the base address (a register) going into the ALU
- Something goes into A2, as I need the value to write to memory
- The sign extended immediate goes into the ALU, which is the offset located outside the brackets
- The `ALUResult` is the final address we wish to write to in memory
- Nothing else

Your annotations can clip through the components itself if that makes things easier to see. Just note the difference between an address and its value.

Table 13: The outputs of the control unit depending on operation

Ins	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUControl
R-type	1	1	0	Mostly 0	0	X	Decode `funct`
`lw`	1	0	1	0	0	1	Add (010)
`sw`	0	X	1	0	1	X	Add (010)
`beq`	0	X	0	1	0	X	Subtract (110)
`addi`	1	0	1	0	0	X	Add (010)

Here are some hints:

- If `RegWrite` is 0, `RegDst` is X
- Write enables should never be X

19.8 Supporting J-type Instructions

The most significant 6 bits are still the opcode, but the remaining 26 bits are the immediate path. There is no way to support a J-type instruction, so how do we change the Datapath?

Note that none of the splitters in the single-cycle MIPS processor does a 25:0 split. Because MIPS is word-addressable, we know that each instruction address is a multiple of 4. So:

- We're going to make the jump address 27 bits long by appending two zeros at the start of it, then shifting it left by two.
- Then, append the four most significant bits from the program counter plus four to it
- And that's our new address.
- I need a new control signal from our control unit to tell us if a jump address is being used. I'll put a new 2:1 multiplexer right next to the program counter multiplexer, which determines if I'm using the jump address or just $PC+4$.
- Now, I have a new row from the jump instruction, and I need to update the table.

Table 14: The control unit with the jump instruction. J is 0 for all non-j instructions.

Ins	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemToReg	ALUControl	J
j	0	X	X	X	0	X	X	1

19.9 Determining The Critical Path

The load word instruction takes the longest.

- It must go through the register file, quite a few multiplexers, the ALU, the data memory, and so on.

In terms of single cycle performance, everything is limited by the critical path.

What could be faster is a multi-cycle CPU. Instead of each cycle having to complete every instruction, can break up an instruction into different stages? Can we come up with a faster processor? The tradeoff is that the control unit is more complex.

Modern instructions do not implement a multi-cycle CPU, but instead a pipelined CPU. (ECE552)

20 Processors

20.1 How Can ARM Run Intel Apps?

(Will not appear on the exam)

By softening the transition, it is possible to run applications compiled for previous systems. They did it completely in software. It does dynamic binary translation (at runtime). In runtime, not only I am running the application on intel, but Rosetta also 2 is translating the machine code instructions to ARM ISA instructions.

There is a slowdown, and not everything is extremely fast. Always think critically if anyone is making claims about processors.

20.2 Takeaways

Control unit is simple – it extracts instructions and tells the Datapath what to do

The Datapath fetches, decodes, executes, and so on.

20.3 The actual memory map

MIPS address space is 4 GB, made up of four segments

- Text, 256MB
- Global data, 64KB
- Dynamic data, 2GB
- Reserved

The global data segment where variables are initialized at startup go there. That is done by the compiler. There is a register in MIPS called GP. It does not change. That address is different – between the start address and the end address. If we go halfway between the start address and the end address, that is the value contained by

`$gp`. Whenever I need to access a global variable, the global variable will always exist at a constant offset from `$gp`.

20.4 The Dynamic Data Segment

Data is dynamically allocated, during runtime.

- Stack: local variables and arrays, grows downwards
- Heap: Grows upwards
  - Grows too much and interferes with the stack, that is an out of memory error.

If your code is running on a processor and there's no OS to save you, you could ruin the program's memory and you'll have to reboot the entire system.

20.5 Translating and Starting a Program

Compiling is a multi-step process.

- The compiler takes some high-level code and translates it into assembly code (mainly C, C++)
  - There are many assembly languages (MIPS, Arm, x86)
- Once it's in assembly, the assembly code gets translated into machine code. It's 1s and 0s that can be interpreted by a microarchitecture.
- Usually, we program in more than one file. Each becomes its own translation unit. We don't bring that together; we get the linker to do that for me. The last steps for getting and generating an executable is the linker.
- The loader launches the code so the processor can run the instructions.

1. Compilation.
   a. Global variables are put in the `.data` segment. They may not have any initial values
   b. Values are assigned using `sw`
      i. Before we can store word, we need to put the value we want into a register
      ii. Labels are replaced with addresses as an offset from the global pointer

2. Linking

    a. Program can be multiple files
    b. Programs import other libraries
    c. Linker resolves symbols across files and deals with conflicts

20.6 Compile vs. Run-Time

Compile-time errors are caught without running the program. Best to optimize the code at compile time.

Runtime errors are only known while running the program and takes forever to resolve. You'll have to debug the code.

20.7 Compiled vs. Interpreted

- Compiled C code runs on the physical machine
    - Sometimes, called native execution and is called native execution
    - We get machine code. It's something a processor can interpret.
    - May not be cross-compatible with other operating systems
        * C program was compiled for a specific type of system, e.g., Windows 10 x86, Mac OS X x86, Mac OS X Arm
        * Can run program without anything installed if the program is compatible
- Compiled Java code runs on JVM.
    - Not native, so it can slow down
    - It's in binary, so it can be interpreted way faster
    - Java program is compiled for a versioned java virtual machine. It converts the byte code into machine code. But because it's in byte code, it is a lot easier to run and distribute to users.
    - But you need JVM installed. Your JVM may be outdated.
        * The actual implementation of the JDK depends on the distributor.

There is a large difference between C and byte code.

20.8 Sharing Memory

An embedded system can only run program. However, most systems run multiple programs at the same time.

How do we share memory without changing the ISA and microarchitecture? Right now, we have 4GB of MIPS, and the memory map is for a single program. But what do we do for two programs?

Virtual memory – we're going to swap the programs in different times. When one program is running on one system, the addresses are physical addresses. In virtual memory, we use virtual addresses. They're not actual physical address, but whenever you do a load, the OS will translate that into a physical address.

20.9 Sharing The Processor

Cooperative scheduling:

- Yielding: let another program run.
- How will I deal with the registers? This requires a context switch, so the architectural state needs to be saved somewhere.

If not:

- An OS must be able to interrupt a process.
- Interrupt and saving the architectural states need some hardware support. Hence, we use exceptions. In hardware, maybe a key is pressed (more like callbacks).
- An interrupt-based system is much more elegant – le me know when a user presses a key, and I will respond
- There are also software types of exception
  - Such as undefined instructions
  - Or division by 0
- Exceptions need to be handled. The operating system handles them.

20.10 Exceptions in MIPS

- The register file are general purpose registers
- In addition to those, there are special purpose registers
- One such is the cause register: it records the cause of an exception
- The EPC is for a PC at the time of an exception

Program is running and an exception occurs.

1. Finish current instruction
2. Save arch state
3. Jump to handler
4. Exception handler finished
5. Restore architectural state (all registers and the program counter, but not the memory)
6. Resume next instruction

20.11 Causing Exceptions

- Use `syscall` (or trap)
- Ask the OS to do something (I cause an exception and I switch to the operation system, such as generating a random number)

Why should we cause an exception instead of using a library?

- Usually, when you ask the operating system to do something, you're asking for something important – something you're not allowed to do. The OS makes sure that processes are independent of each other. For security purposes, my program can't do whatever it wants, so the OS will do that.
- The processor can run in user mode, or in kernel / privileged mode.
  - Kernel / privileged mode gives features that the OS can do that it normally cannot.

20.12 Is This Section on the Exam

Only if it has been covered before this. Most things here aren't. For example, trap instructions and the stack are covered.

21 Exam Tips

Here are a couple of tips I've compiled that should be some help on the exam.

21.1 Disambiguating Assembly To Machine Code

On the exam cheat sheet, you have a MIPS reference card. A row might look like this:

```
add rd, rs, rt . . .  R 0 / 20
```

Beware that the positioning of the three registers may not be in the same order as machine code. Moreover, **machine code tries to put any destination register to the left. This means `rd` for R-type instructions, and `rt` for I-type instructions.** Here's the diagram:

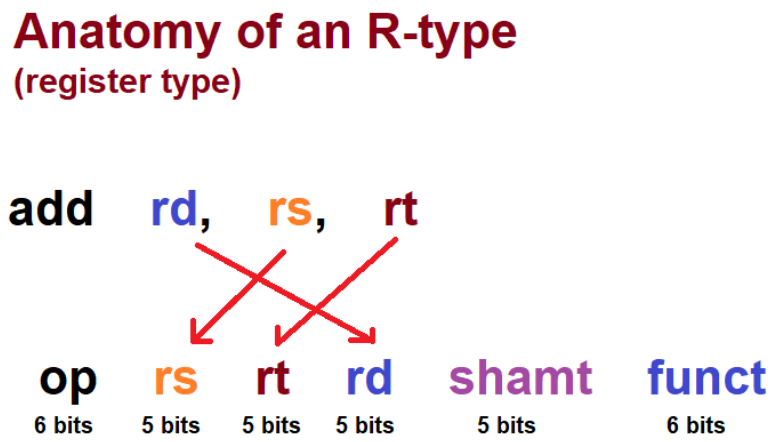


Figure 31: Anatomy of an R-type. Note that some R-type instructions break this format, so pay attention to the MIPS reference sheet. One example is the variable shift operation.

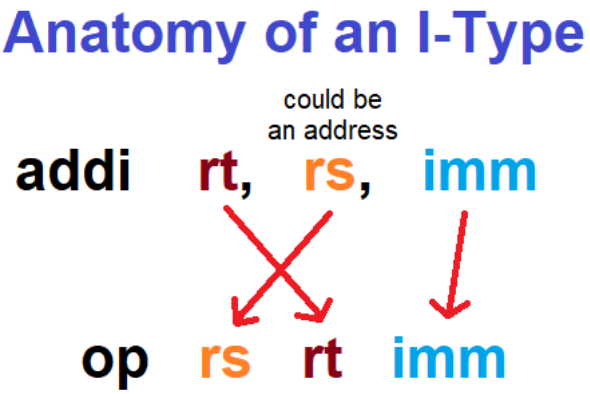


Figure 32: Anatomy of an I-type. Has the same warnings as the previous figure. **The immediate is 16 bits.**

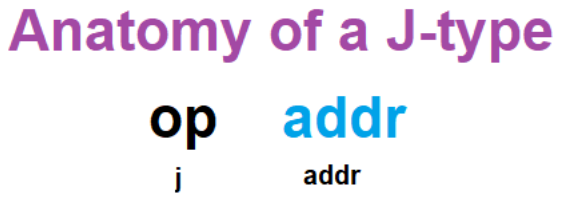


Figure 33: Anatomy of a J-type.

21.1.1 The Confusing Shift Function

The shift function has to be the most confusing, as it may involve a constant, yet all shift operations provided in the MIPS reference card are R-type operations.

There are **two categories** of shifts:

- Regular
  - Register format in ASM: `rd, rt, sh`
  - Output: `rd` = `rt << sh` for a left shift
  - `rs` is very likely a don't care or a zero. `sh` will likely take `rs`'s place in the ALU.
- Variable
  - Register format `rd, rt, rs`

    * Caution! This R-type instruction swaps the positioning of `rs` and `rt`.
    * Output: `rd = rt << rs`
    * Why is the order swapped? No idea.

Each of them having three possible directions:

- Left `<<`
- Right logical `>>>`
- Right arithmetic `>>`

21.2 What Is The Difference Between RS, RT, and RD?

I can't show the image of the complete single-cycle MIPS processor here, but here's what each register is supposed to do:

- `rs` is the source and takes up slices **25:21**. It will almost always be an input to the ALU. If a register is only targeted by `rs` and nothing else, it will never be written to in the cycle.
    - Note that this convention may break if we modify the single-cycle MIPS processor present in the textbook. For example, to support shift instructions, we may need to break this convention.
- `rt` is another source for R-types. It may be considered a destination for I-types. It takes up slices **20:16**.
    - **R-type:** When it is considered a source, in the control unit, `RegDst` is 1 (this lets `rd` be the destination).
    - **I-type:** When it is considered a destination, `RegDst` is 0 and is thus the register targeted for writing.
- `rd` can only be considered in R-type instructions. It takes up slices **15:11**. Note that slices are inclusive here.
    - If the simple-cycle MIPS processor is not modified, `rd` will never be an input to the ALU, and its value will never be read, as it is always used in the context of overwriting.

21.3 How to Annotate the Datapath

Trace through it. You're trying to communicate how the multiplexers are direction the data. Remember that you *should* label the values within the control unit that are not don't cares.

21.4 Do You Want To Divide, or Do You Want To Shift?

> Dividing or multiplying by a power of 2? This can be done in one instruction – USE SHIFTS!!!!!!!

21.5 Set Less Than: The Most Unclear instruction

You might think that `slt` meant to conditionally set a value to something. That is wrong. Here's what it does, in the MIPS cheat sheet:

```
rd = 1 if (rs < rt) else 0
```

Hence, this is a magnitude comparator, as if you're running `int(rs < rt)` in Python. Similar logic applies for the immediate version: just replace the value of `rt` with the immediate.