CSC258

Contents

1 - Transistors

To understand computer software, you also need to understand computer hardware (i.e., its limitations, operations, and behaviour). By building from the ground up, we can see how larger-scale programs are created!

- For example, we start with electricity and transistors to understand logic gates, and then combining them to get advanced logical operators.

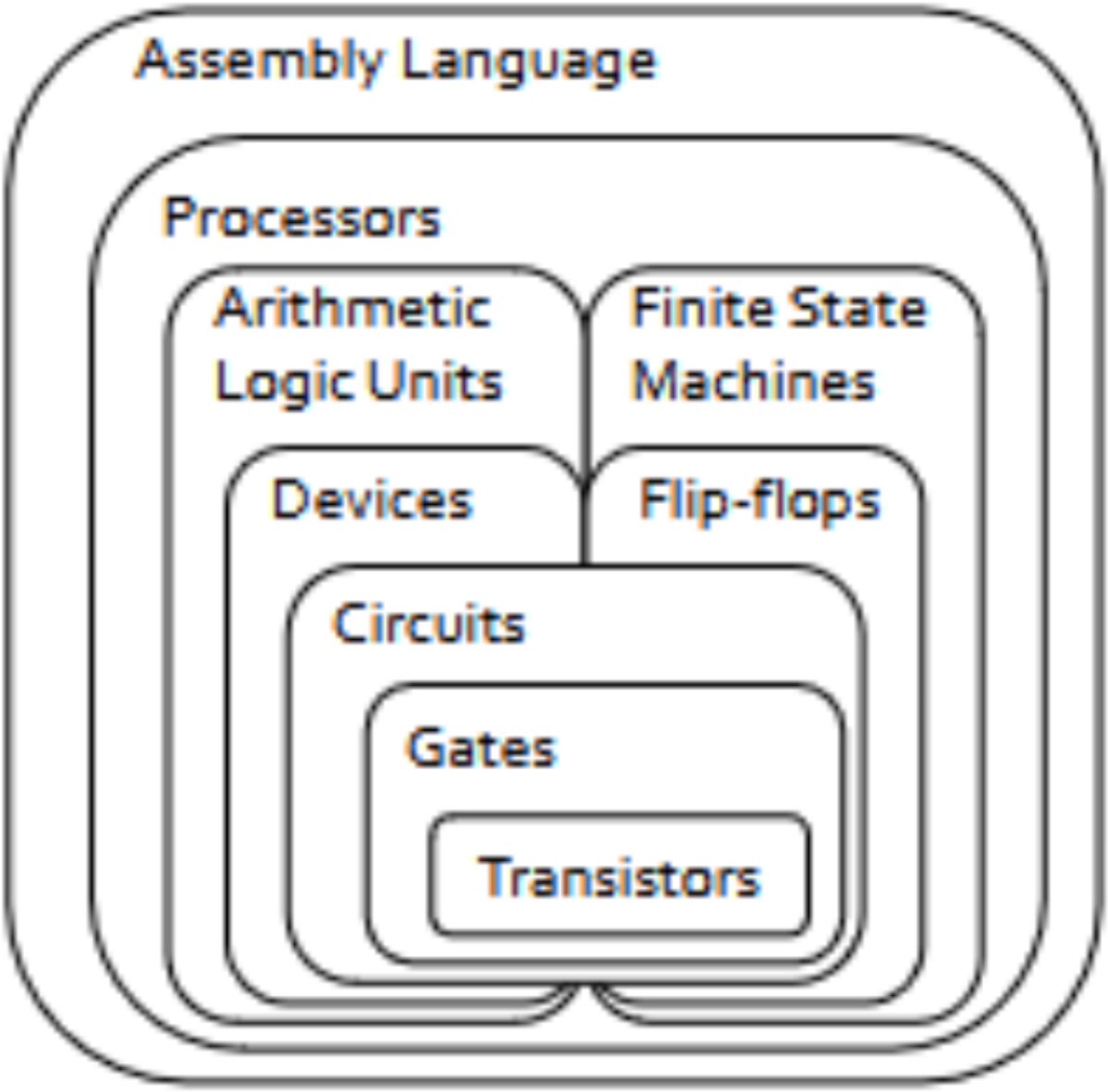


Figure 1: Pasted image 20250106112817.png

Transistors

Transistors - Basic building block of all computer hardware.

- Invented by William Shockley, John Bardeen and Walter in 1947, replacing previous vacuum-tube technology. Transistors connect point A to point B, based on the value at point C.

- If C is high, A and B are connected.
- If C is low, A and B are not.

Figure 2: Pasted image 20250108134359.png

Introduction to Electricity

Electricity - The flow of charged particles (usually electrons) through a material.

- Comes from atoms (protons, neutrons, and electrons). Electrical particles want to flow from regions of high electrical energy (many electrons) to low electrical potential (little electrons).
- This potential is referred to as **voltage** V.
- The rate of electron flow is the **current** I.
- The relationship between voltage and current is **resistance:** $R = \frac{V}{I}$.

Note About Current

Even though current is caused by electrons flowing through a material, we typically measure current as the movement of **positive charges**. - Protons don't actually move! When electrons move from A to B, B becomes more negative and A becomes more positive. - Scientists historically viewed current in terms of the creation of positive charge rather than negative.
There are two common sources of electricity (electrons):
1. Batteries - Has a concentration of particles stored inside them that eventually run out.
2. Electrical outlets - Constantly supplied with electric particles.
Electricity Pathing:
- Current always flows towards the zero voltage point of a circuit (**the ground**).
- Electricity will take the path of least resistance from source to ground. We can use this fact to drive circuits! Each circuit has a source of electrical particles - some path between this source and the ground, and some resistance along this path will dissipate electrons.
Resistance - The measure of how well a material allows electricity to flow through it.
- Measured in **ohms** Ω. Higher resistance means higher ohms. **Insulators -** Do not conduct electricity; high resistance. **Conductors -** Conduct electricity; low resistance. Generally used for wires. **Semiconductors -** In between conductors and insulators.

Using Transistors

Transistors - Devices made of semiconductor material, acting as a conductor and an insulator at some points.

Transistors are commonly implemented using **MOSFET**s (Metal Oxide Semiconductor Field Effect Transistor).

- Has three defining wires, the **source**, **drain**, and **gate**. There are two types of MOSFETs this course discusses:

1. **nMOS** Transistors - Electricity conducts between the source and drain when a **positive** voltage is applied to the gate.
2. **pMOS** Transistors - Electricity conducts between the source and drain when a **negative** voltage is applied to the gate. Gate voltage determines whether the source and drain are connected *only if* the source voltage is high.

We combine MOSFETs to create high and low voltage outputs – based on high and low voltage inputs.

Therefore, in a transistor, the output has to be supplied with high *or* low electrical values from one of the inputs.

Gates **Gate -** A combination of transistors that represent logic (e.g., AND, OR, NOT). Gates have the following physical data:

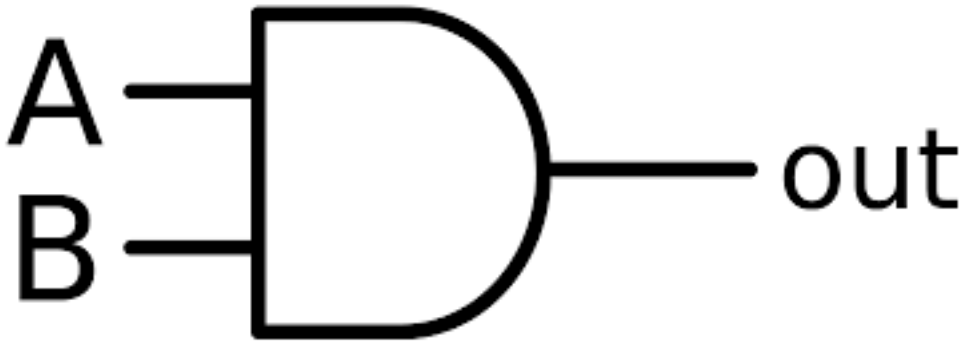
- High input ($V_{cc} = 5V$).
- Low input (ground $= 0V$).
- Switching time $\approx$ 120 picoseconds.
- Switching interval $\approx$ 10 nanoseconds.

[!NOTE] Example: NOT Gate

When designing/making gates from transistors, keep the following in mind:

- Each gate has one output and one or more inputs.
- Every combination of input values **must** connect to high voltage or low voltage.
- **Not connecting the output to high voltage isn't the same as connecting the output to low voltage!**

[!NOTE] Example: Two-Input AND Gate Here, we have two inputs and one output.



The output is high when both A and B are high. If A or B are low, the output connects to the ground.

Lab 1

Lab Equipment **Breadboard -** Standard working area for connecting digital components together.

- Red and blue horizontal rows at the top are connected.
- Columns in the middle sections are connected.

Wires - Connects different components together.

Gates - The logic used in digital components. We will be using IC chips, for example:

- NOT Gate: 74LS04

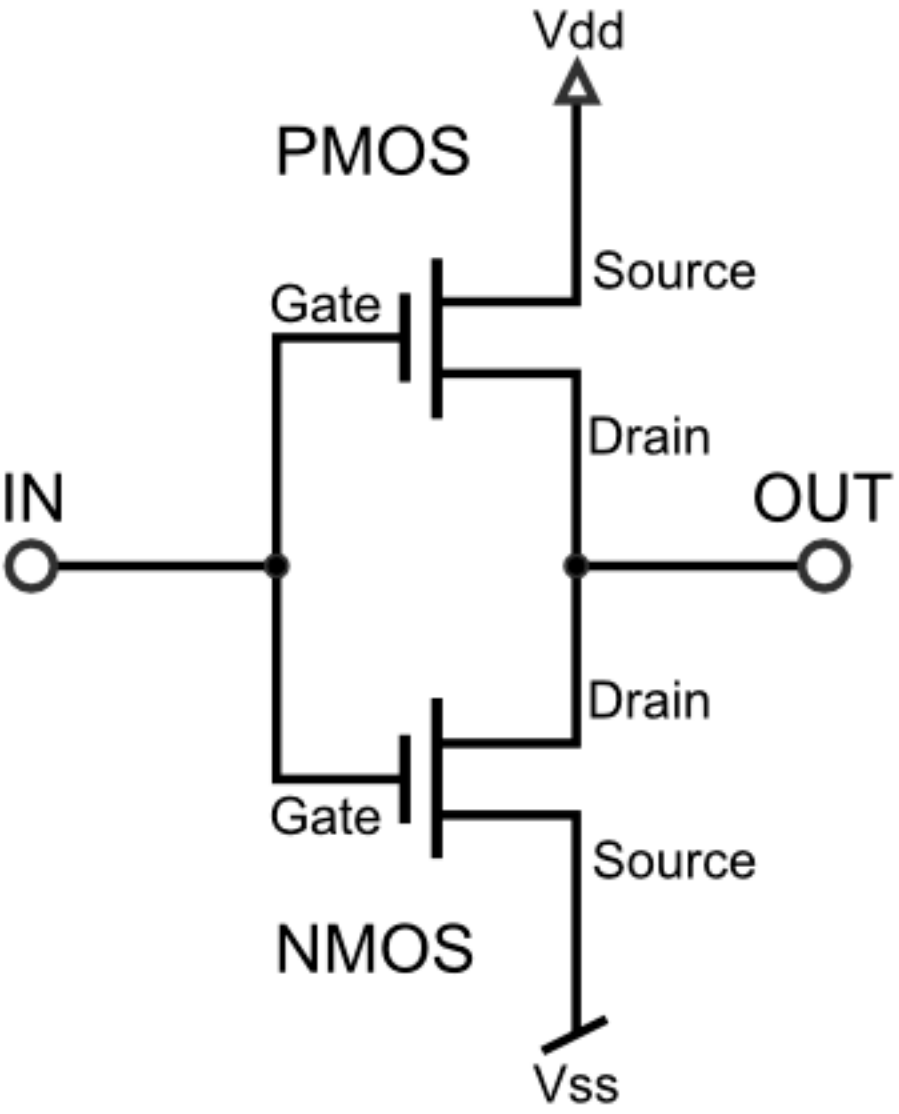


Figure 3: Typical-CMOS-Inverter-schematic.png

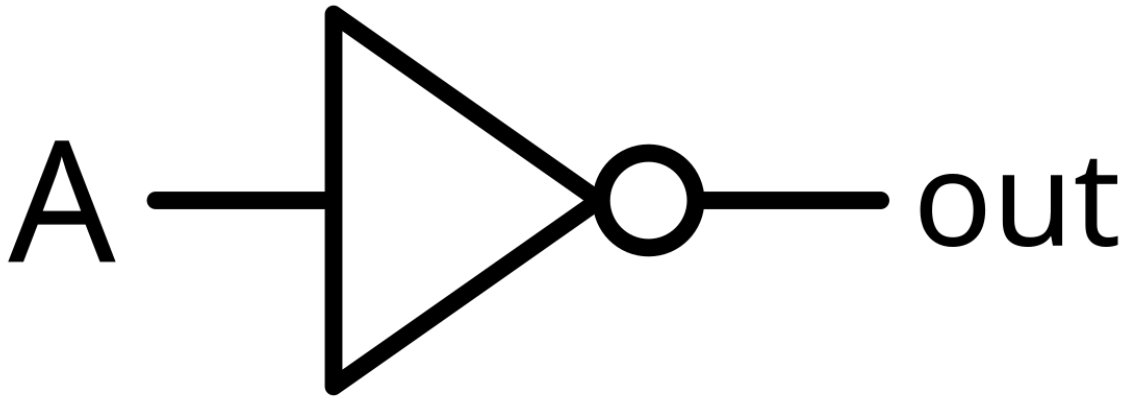


Figure 4: Not-gate-en.svg.png

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Figure 5: Pasted image 20250112201507.png

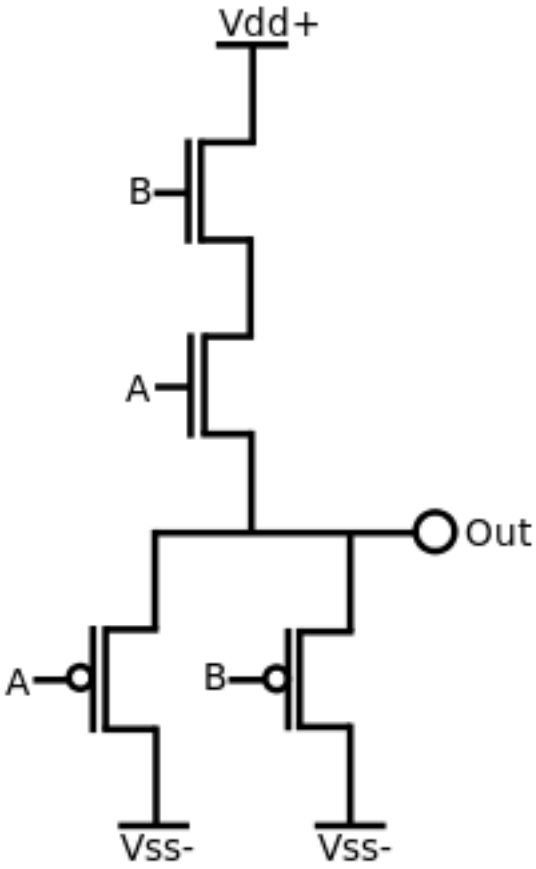


Figure 6: LOYtw.png

- AND Gate: 74LS08
- OR Gate: 74LS32

There is typically a notch at one end which helps with alignment, and a dot at the first pin. V_{cc} and GND always have to be connected to the power source and ground respectively.

2 - Circuit Design

Logic Gate Circuits

When designing a circuit, we must keep in mind two things:

1. Given a specified input and output, connect circuit components to produce given behaviour.
2. Create the circuit with the lowest possible cost (i.e., fewest components).

With more complex circuits, we need to follow a more methodical approach when designing them:

1. Create truth table(s).
2. Express truth table behaviour as a **boolean expression**.
    - This is the step where we make an efficient design!
3. Convert this expression into gates.

Truth Tables

Create a truth table that describes when a circuit's output is high based on the values of its inputs.

- There are 2^n total outputs, where n is the number of inputs.

To make conversions from truth tables to expressions easier, we express them as rows (minterms/maxterms) as shorthand.

Minterm - An AND expression with every input present in true or complemented form (e.g., $A \cdot \overline{B} \cdot C \cdot D$). Standard notation we use (easier to manipulate!) **Maxterm -** An OR expression with every input present in true or complemented form (e.g., $A + \overline{B} + C + D$).

Suppose we have an expression with 4 inputs A, B, C, D. Each row, beginning from 0 is also the binary representation of its inputs up to $2^n - 1$ (e.g., $A \cdot B \cdot \overline{C} \cdot D$ is 1101 in binary and 13 in decimal).

- A minterm is expressed with m_X.
- A maxterm is expressed with M_X.

Boolean Expression Notation

AND operations are denoted by multiplication: $A \cdot B \cdot C$ or $A * B * C$ or $A \wedge B \wedge C$ OR operations are denoted by addition: $A + B + C$ or $A \vee B \vee C$ NOT is denoted as $\not{A}$ or A' or $\overline{A}$ XOR rarely occurs but is denoted as $A \oplus B$

Minterms to Circuits A single minterm indicates a set of inputs that make the output go high. To combine minterms, just use an OR operation!

Sum-of-Minterms (SOM) - The combined high outputs are a sum/union of these minterm expressions.

- Which inputs cause the output to go high.
- Useful when there are few input combinations with high output. **Product-of-Maxterms (POM) -** The combined low outputs are an intersection of these maxterm expressions.

- Useful when there are few low output cases.

Once you get the SOM expression, you can easily convert it to the equivalent combination of gates.

Pasted image 20250119161948.png

Figure 7: Pasted image 20250119161948.png

Reducing Circuits To minimize the number of gates we use, we want to reduce the boolean expression as much as possible from a collection of minterms to something smaller.

Axioms:

- $0 \cdot 0 = 0$
- $1 \cdot 1 = 1$
- $0 \cdot 1 = 1 \cdot 0 = 0$
- If $x = 1$, $\overline{x} = 0$
- **Commutative Law:**
  - $x \cdot y = y \cdot x$
  - $x + y = y + x$
- **Associative Law:**
  - $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
  - $x + (y + z) = (x + y) + z$
- **Distributive Law:**
  - $x \cdot (y + z) = x \cdot y + x \cdot z$
  - $x + (y \cdot z) = (x + y) \cdot (x + z)$
- **Consensus Law:**
  - $x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z$
- **Absorption Law:**
  - $x \cdot (x + y) = x$
  - $x + (x \cdot y) = x$
- **De Morgan's Laws:**
  - $\overline{x} \cdot \overline{y} = \overline{x + y}$
  - $\overline{x} + \overline{y} = \overline{x \cdot y}$
  - De Morgan's is important since NAND gates are very cheap to fabricate!

Measuring Gate Cost The simplest gate is denoted with the lowest **gate cost** (G) or the lowest gate cost with NOTs (GN). To calculate cost, add all the gates together (as well as NOT gates in the case of GN cost).

Karnaugh Maps

Karnaugh Maps (K-Maps) represent the same information as a truth able, but in a format that helps us see what minterms can be combined.

- Adjacent minterms in the grid differ by a single literal.
- Values in the grid are the outputs for that minterm.

Pasted image 20250119165107.png Minterms can only be grouped into a single term that omits that value.

Once the K-map is created, draw boxes over groups of high output values.

- Boxes must be rectangular with size of power 2.
- Boxes may overlap with each other.
- Boxes may wrap across edges of the map.

Pasted image 20250119165356.png

Figure 8: Pasted image 20250119165356.png

- Once you find the minimal number of boxes that cover the high outputs, create boolean expressions from these inputs common to all elements in the box.
  - In the above example: $Y = B \cdot C + A \cdot \overline{C}$

We can also use this techniques for maxterms. Group zero entries together instead of ones.

3 - Logical Devices

Now that we know how to bring together gates to create circuits, we can create more complex structures.

Combinational Circuits

Logic Devices - A device where its particular structure is common to many circuits; has block elements of their own.

Combinational Circuits - A circuit where the outputs rely strictly on the inputs.

- Examples: Multiplexers ("mux"), decoders (e.g., seven-segment), adders (half/full), subtractors, comparators.

Another type of circuit that exists is a **sequential circuit**!

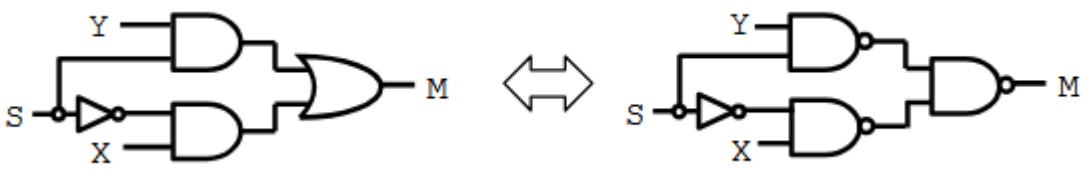
More on Karnaugh Maps

Recall that K-maps can be of any size and have any number of inputs. Since adjacent minterms only differ by a **single** value, they can be grouped into a single term that omits that value. - That is, it shows a guide on how neighbouring terms may be combined. There may be cases where **no** combinations are possible! K-maps will not help in these cases (e.g., multi-input XOR gates).
We also have **"Don't Care" values**, which are inputs that will either never happen or are not meaningful to a given design. They are denoted by an X.

Multiplexers

Multiplexers, or muxes, are a type of logic device, where, depending on the selector bits and inputs, will send to a single output.

- For example, a 2-to-1 mux: $M = (X * \overline{S}) + (Y * S)$, where S is the select input, and X, Y are the data inputs.



' Muxes

are commonly used when you need to select from multiple input values (e.g., surveillance monitors, routers).

Demultiplexers - The reverse of a multiplexer; related to decoders. Commonly used in something like for modems receiving internet data.

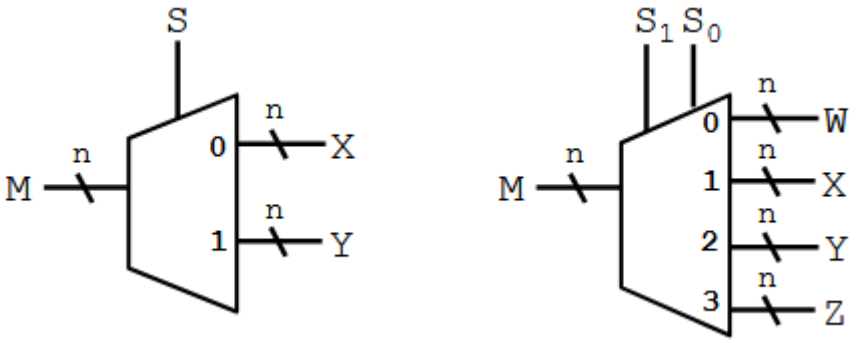


Figure 9: Pasted image 20250121232200.png

Decoders **Decoders -** A translator from the output of one circuit to the input of another; the mapping between two different encodings. For example:

- A **binary signal splitter** activates one of four output lines based on a 2 digit binary number.
- A **7-segment decoder** translates a binary number to the seven segments of a digital display. Each output segment has a particular logic defining it.
  - These segments are **"active-high"**, meaning that setting it to high turns it on.

Adder Circuits

Adders (Binary Adders) are small circuits that add two digits together. They are often combined together to create **iterative** combinational circuits. Types of adders:

- Half adders (HA)
- Full adders (FA)
- Ripple Carry Adder

[!NOTE] Binary Math Review **Decimal Number -** A power of 10, e.g., $258 = 2 \cdot 10^2 + 5 \cdot 10^1 + 8 \cdot 10^0$ **Binary Number -** A power of 2, e.g., $01101_2 = 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$ **Hexadecimal Number -** Base 16 numbers, going from $0 - 9$, then $A(10)$ to $F(15)$. Typically expressed as 0x____ **Conversions:** - Decimal $\rightarrow$ Binary - Continuously divide by 2 and write the remainders.

Unsigned Binary Addition: Suppose we want to add $27 + 53$ in binary.

$$27 = 00011011$$
$$53 = 00110101$$

Then:

$$\begin{array}{r} {\color{red}1\,1\,1\,1\,1\,1} \\ 00011011 \\ +00110101 \\ \hline 01010000 \end{array}$$

Note that bits can only represent a set of numbers. For example, $95 + 181$:

$$\begin{array}{r} 01011111 \\ +10110101 \\ \hline 100010100 \end{array}$$

With only 8 bits, we can only represent unsigned numbers 0 to 255. Therefore, $95 + 181$ in binary is 0010100.

Half Adders A 2-input, 1-bit wide binary adder performs the following computations:

- $X + Y = CS$
- $0 + 0 = 00$
- $0 + 1 = 01$
- $1 + 0 = 01$
- $1 + 1 = 10$ A half adder will add two bits to produce a two-bit sum, expressed as the sum bit S and carry bit C. Equations and circuits for half adders are easy to define!

$$C = X \cdot Y \qquad S = X \cdot \overline{Y} + \overline{X} \cdot Y = X \oplus Y$$

Full Adders Full adders are similar to half adders, but have *another* input Z, which is the **carry-in** bit.

- C and Z are labeled as C_{out} and C_{in}. When $Z = 0$, the unit behaves exactly like a half adder. When $Z = 1$:
- $X + Y + Z = CS$
- $0 + 0 + 1 = 01$
- $0 + 1 + 1 = 10$
- $1 + 0 + 1 = 10$
- $1 + 1 + 1 = 11$ Then, our full adder will have the following equation:

$$C = X \cdot Y + X \cdot Z + Y \cdot Z \qquad S = X \oplus Y \oplus Z$$

We can rewrite C as $C = X \cdot Y + (X \oplus Y) \cdot Z$, giving us two new terms:

- $X \cdot Y =$ **Carry generate** (G).
- $X \oplus Y =$ **Carry propagate** (P).

Ripple-Carry Binary Adder Full adder units can be chained together to perform operations on **signal vectors**.

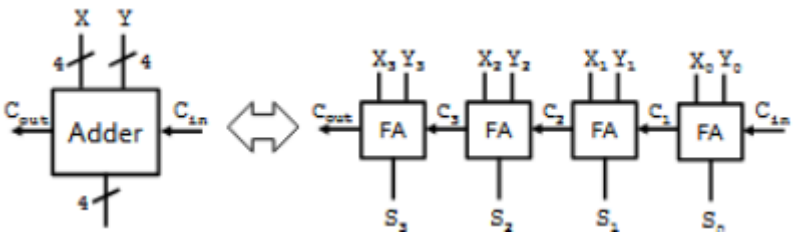


Figure 10: Pasted image 20250219150437.png

Subtractors Subtractors are an extension of adders (i.e., performing addition on a negative number).

[!NOTE] Binary Subtraction Assume you have an 8-bit binary number. To subtract (i.e., perform addition on a negative number), we use **signed numbers** (bits used to store a 2's complement negative number). For signed numbers, there are a few rules to remember: 1. The largest binary number is a **zero** followed by all ones. 2. The binary value for -1 in all the digits. 3. The most negative binary number is a **one** followed by all 0's.

There are 2^n values stored in a n-digit binary number; 2^{n-1} are negative, and $2^{n-1}-1$ are positive, and 1 is zero. For example, an 8-bit binary number has 256 values from -128 to 127.

Decimal	Unsigned	Signed
7	111	---
6	110	---
5	101	---
4	100	---
3	011	011
2	010	010
1	001	001
0	000	000
-1	---	111
-2	---	110
-3	---	101
-4	---	100

To get the 2's complement, flip all digits (the 1's complement), then add a 1.

Also note that if we add two signed positive numbers, we can get a negative number (e.g., 0110 + 0011). In these cases, we should use **overflow** flags (more on this in Processors).

For a 4-bit subtractor $(X-Y)$, we use the following operation:

- X plus the 2's complement of Y; or,
- X plus the 1's complement of Y plus 1.
  - Feed 1 as the carry-in at the least significant adder, and use NOT gates to get the 1's complement of Y.

Older processors, rather than using signed numbers, will use **sign and magnitude representation**, where the one bit is designated as the sign (+/-), 0 for positive, and 1 for negative; and the magnitude is the remaining bits of the number.

Comparators

A comparator is a circuit that takes in **two input vectors** and determines if the first is greater than, less than, or equal to the second.

Basic Comparators - For two binary numbers A, B longer than one bit (e.g., 2-bit):

- $A == B$: $\left(A_1 \cdot B_1 + \overline{A_1} \cdot \overline{B_1}\right) \cdot \left(A_0 \cdot B_0 + \overline{A_0} \cdot \overline{B_0}\right)$
  - The first checks the values of bit 1 are the same, and then the values of bit 0 are the same.

- $A > B$: $(A_1 \cdot \overline{B_1}) + \left(A_1 \cdot B_1 + \overline{A_1} \cdot \overline{B_1}\right) \cdot (A_0 \cdot \overline{B_0})$
- $A < B$: $\left(\overline{A_1} \cdot B_1\right) + \left(A_1 \cdot B_1 + \overline{A_1} \cdot \overline{B_1}\right) \cdot \left(\overline{A_0} \cdot B_0\right)$

The general circuit for comparators requires you to define equations for each case.

1. **Case #1 -** Equality
   - If A, B are equal, all bits are equal.
   - $A == B : X_0 \cdot \cdots \cdot X_n$ for all $X_i = A_i \cdot B_i + \overline{A_i} \cdot \overline{B_i}$.

A=B :

	$\overline{B}_0 \cdot \overline{B}_1$	$B_0 \cdot \overline{B}_1$	$B_0 \cdot B_1$	$\overline{B}_0 \cdot B_1$
$\overline{A}_0 \cdot \overline{A}_1$	1	0	0	0
$A_0 \cdot \overline{A}_1$	0	1	0	0
$A_0 \cdot A_1$	0	0	1	0
$\overline{A}_0 \cdot A_1$	0	0	0	1

$$EQ = \overline{B}_0 \cdot \overline{B}_1 \cdot \overline{A}_0 \cdot \overline{A}_1 + B_0 \cdot \overline{B}_1 \cdot A_0 \cdot \overline{A}_1 + B_0 \cdot B_1 \cdot A_0 \cdot A_1 + \overline{B}_0 \cdot B_1 \cdot \overline{A}_0 \cdot A_1$$

Figure 11: Pasted image 20250219152952.png

-
2. **Case #2** - $A > B$
   - The first **non-matching** bit occurs at bit i, where $A_i = 1$ and $B_i = 0$. All higher bits match.
   - $A > B : A_n \cdot \overline{B}_n + X_n \cdot A_{n-1} \cdot \overline{B}_{n-1} + \cdots + A_0 \cdot \overline{B}_0 \cdot \prod_{k=1}^{n} X_k$
   -
3. **Case #3 -** $A < B$
   - The first **non-matching** bit occurs at bit i, where $A_i = 0$ and $B_i = 1$. All higher bits match.
   - $A < B : \overline{A}_n \cdot B_n + X_n \cdot \overline{A}_{n-1} \cdot B_{n-1} + \cdots + \overline{A}_0 \cdot B_0 \cdot \prod_{k=1}^{n} X_k$
   -

As numbers get larger, the comparator circuit gets more complex. At a certain level, sometimes it is easier to just process the result of a subtraction operation instead (easier, less circuitry, but not faster).

4 - Latches & Flip-Flops

Sequential Circuits

Sequential Circuit - Circuits depending on the current input and the *previous* state of the circuit.

- Sequential circuits are valuable in memorizing values and reacting to inputs changing.

A>B:

	$\overline{B}_0 \cdot \overline{B}_1$	$B_0 \cdot \overline{B}_1$	$B_0 \cdot B_1$	$\overline{B}_0 \cdot B_1$
$\overline{A}_0 \cdot \overline{A}_1$	0	0	0	0
$A_0 \cdot \overline{A}_1$	1	0	0	0
$A_0 \cdot A_1$	1	1	0	1
$\overline{A}_0 \cdot A_1$	1	1	0	0

$$GT = \overline{B}_1 \cdot A_1 + \overline{B}_0 \cdot \overline{B}_1 \cdot A_0 + \overline{B}_0 \cdot A_0 \cdot A_1$$

Figure 12: Pasted image 20250219153003.png

A<B:

	$\overline{B}_0 \cdot \overline{B}_1$	$B_0 \cdot \overline{B}_1$	$B_0 \cdot B_1$	$\overline{B}_0 \cdot B_1$
$\overline{A}_0 \cdot \overline{A}_1$	0	1	1	1
$A_0 \cdot \overline{A}_1$	0	0	1	1
$A_0 \cdot A_1$	0	0	0	0
$\overline{A}_0 \cdot A_1$	0	0	1	0

$$LT = B_1 \cdot \overline{A}_1 + B_0 \cdot B_1 \cdot \overline{A}_0 + B_0 \cdot \overline{A}_0 \cdot \overline{A}_1$$

Figure 13: Pasted image 20250219152934.png

Understanding Time

Note that outputs of a circuit don't change instantaneously! Time is needed for the inputs to propagate through the circuit and cause a change. **Gate Delay -** The length of time it takes for an input change to result in the corresponding output change. This delay is denoted as T time.
![Pasted image 20250412235057.png](C:/Users/madeline/Documents/GitHub/Obsidian/Courses/4th Year/Winter/CSC258/Images/Pasted image 20250412235057.png) Q_T and Q_{T+1} represent values of Q at a time T, and a point in time immediately after $(T+1)$.
Timing signals (and when they are sampled) use **clock signals**, which are regular pulse signals where the high values indicates updates of the circuit. The number of pulses that occur per second are measured in **Hertz** (x hz is x cycles/second).
Feedback Circuits - Circuits where the outputs are fed back on inputs (i.e., Q_T and Q_{T+1} circuits). For example, a NAND gate causes **unsteady states**:
![Pasted image 20250412235837.png](C:/Users/madeline/Documents/GitHub/Obsidian/Courses/4th Year/Winter/CSC258/Images/Pasted image 20250412235837.png)
![Pasted image 20250412235849.png](C:/Users/madeline/Documents/GitHub/Obsidian/Courses/4th Year/Winter/CSC258/Images/Pasted image 20250412235849.png)

Latches

Latches - A circuit where multiple feedback circuits are combined; these create more steady behaviour. The output of a latch should not be applied directly or through combinational logic to the input of the same/another latch when they all have the same control/clock signal.

SR Latch $\overline{SR}$ **Latch -** A set/reset latch, where inputs of 11 maintain the previous output state.

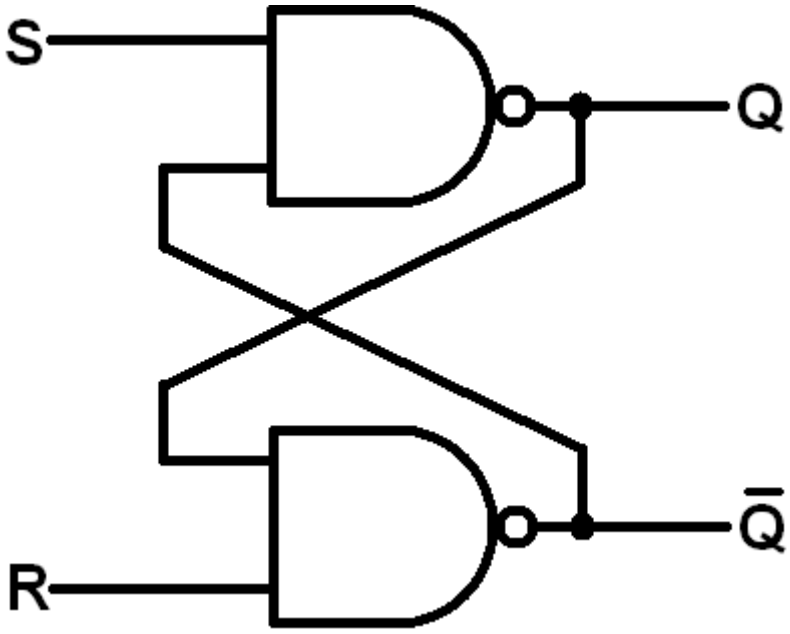
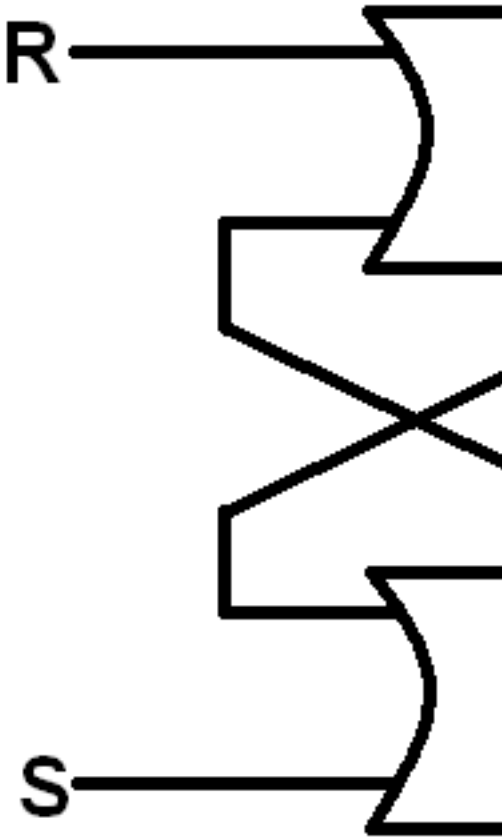


Figure 14: 34-sr-latch-nand.png

$\overline{S}$	$\overline{R}$	Q_T	$\overline{Q}_T$	Q_{T+1}	$\overline{Q}_{T+1}$
0	0	X	X	1	1
0	1	X	X	1	0
1	0	X	X	0	1
1	1	0	1	0	1
1	1	1	0	1	0

Note that going from 00 to 11 produces unstable behaviour; the order of which input changes matters! They are considered **forbidden states**.



There is also a SR **latch** where inputs of 00 maintain the previous output state:

Timing Diagram:

S	R	Q_T	$\overline{Q}_T$	Q_{T+1}	$\overline{Q}_{T+1}$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	0	0

Figure 15: Pasted image 20250413000932.png

Clocked SR Latch Sometimes an input C is connected to a pulse/clock signal so that the clock must be high for the inputs to have any effect.

D Latch SR latches have the issue of having a forbidden state, so we can instead make the inputs dependent on a single signal D.

D Latch (Gated D-Latch) - Feedback circuit acting similarly to an SR latch. It has no forbidden states but is **transparent** (any changes to its inputs are **visible** to the output when the clock is 1) which can create oscillating changes in output.

[!NOTE] Value- vs. Edge-Triggered Latches are **value** triggered since the output values will change for the duration of the clock being high. Instead, we want changes to happen one at a time and at the moment the clock changes (edge-triggered).

Flip Flops

To deal with the timing issues with latches, we can create a *disconnect* between the circuit output and input to prevent unwanted feedback and changes to output. To do this, we can connect two latches together.

Flip-Flop - A latched circuit whose output is triggered with the rising or falling edge of a clock pulse.

SR Master-Slave Flip-Flop:

Edge-Triggered D Flip-Flop: Instead of two SR latches, connect a D latch to the input of a SR latch. It is **negative-edge triggered**.

Positive-Edge Triggered Flip-Flops:

T Flip-Flop: *Toggles* its value whenever T is high.

JK Flip-Flop: Produces four behaviours;

1. If J, K are 0, **maintain** the output.

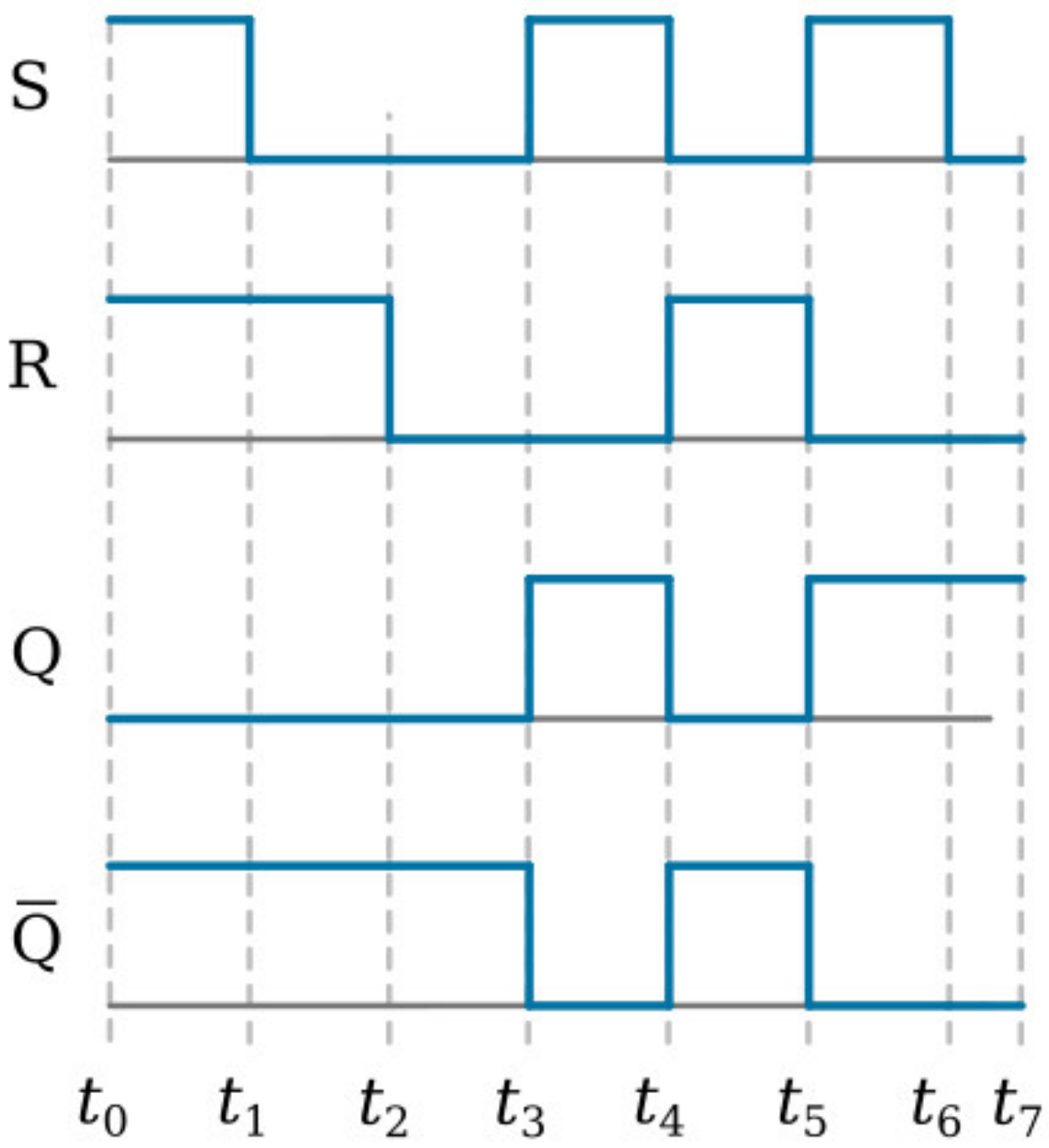


Figure 16: rs-flip-flop-signals.jpg

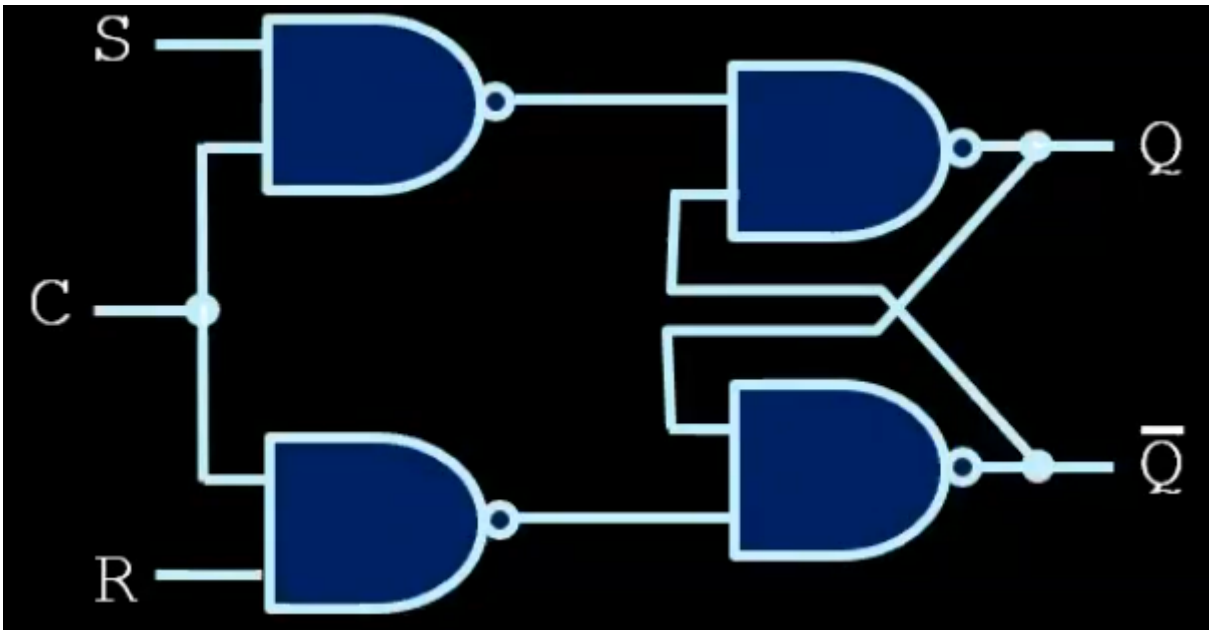


Figure 17: Pasted image 20250413001514.png

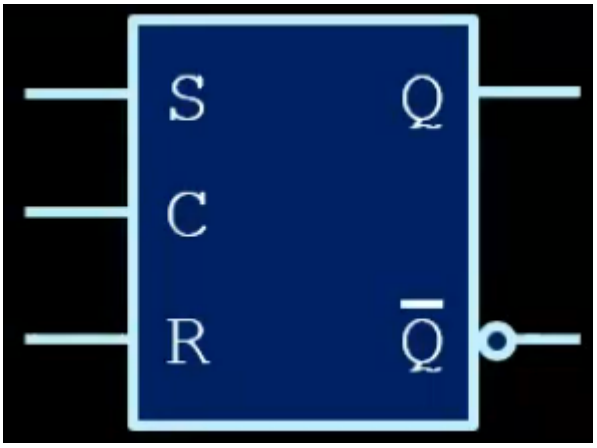
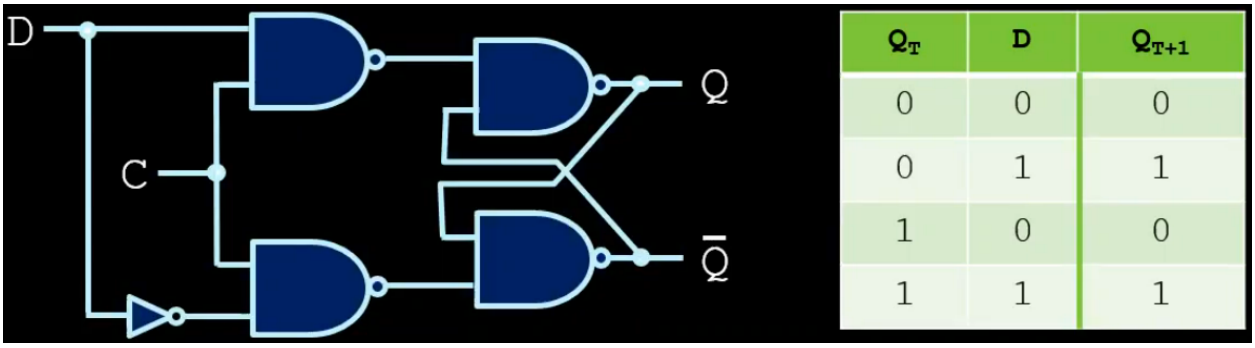


Figure 18: Pasted image 20250413001657.png



Q_T	D	Q_{T+1}
0	0	0
0	1	1
1	0	0
1	1	1

Figure 19: Pasted image 20250413001840.png

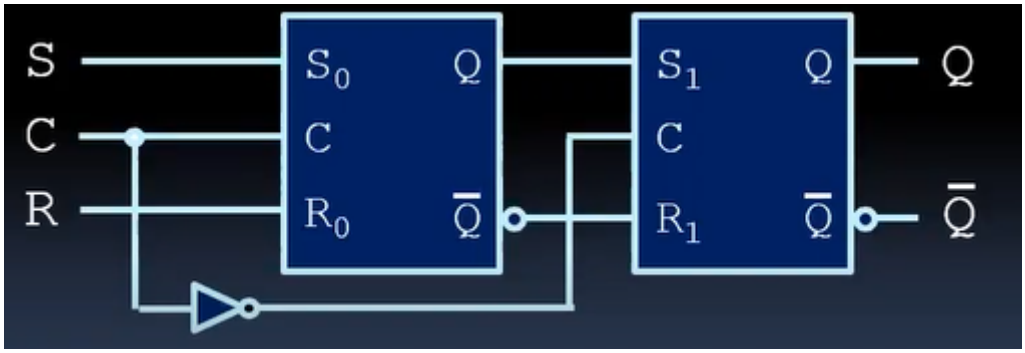


Figure 20: Pasted image 20250413094027.png

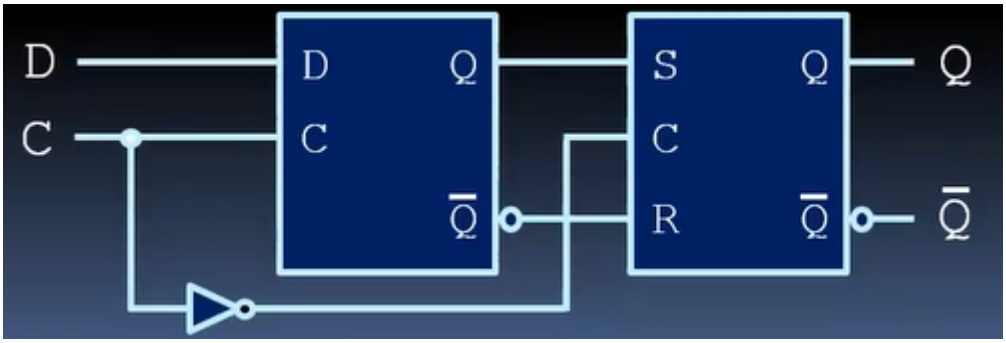


Figure 21: Pasted image 20250413094241.png

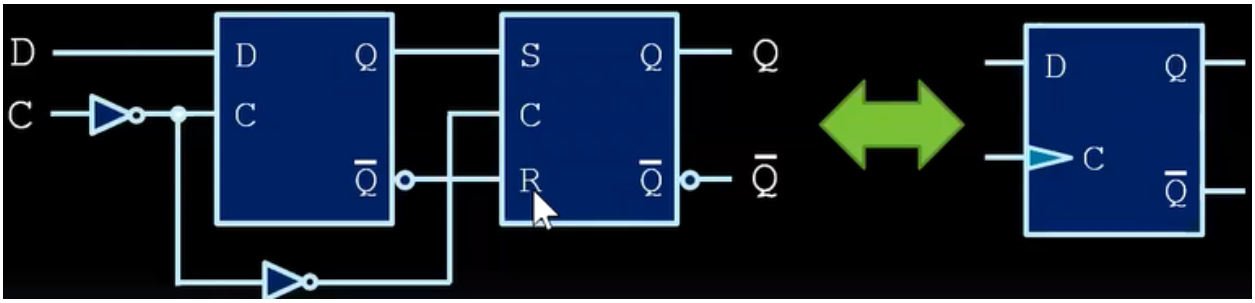


Figure 22: Pasted image 20250413094522.png

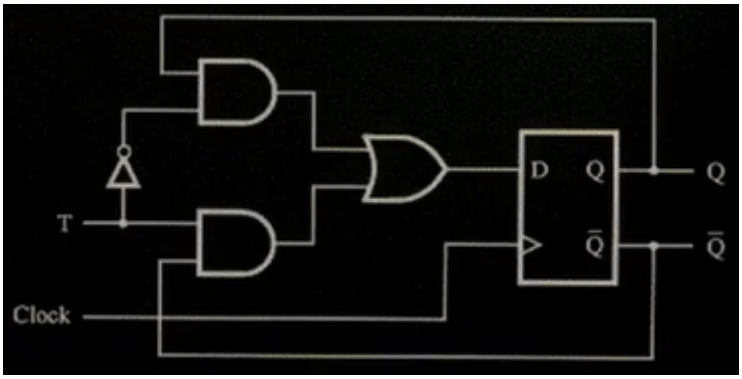


Figure 23: Pasted image 20250413094729.png

2. If J, K are 1, toggle output value.
3. If J is 1 and K is 0, **set** output to 1.
4. If J is 0 and K is 1, **reset** output to 0.

Sequential Devices

Now, with a better understanding of flip flops, we can use them in larger circuit designs as **sequential circuits**, which are the basis for memory, instruction processing, and any other operation that requires circuits to remember past data values (i.e., **states**) of the circuit.

Flip flops allow for better timing considerations, since input should not change at the same time as the active edge of the clock. Instead, we need:

- **Setup time -** input should be stable for some time before the active clock edge.
- **Hold time -** The input should be stable for some time immediately after the active clock edge. Note the **maximum clock frequency** (overclock): the time period between two active clock edges cannot be shorter than the **longest propagation delay** between any two flip flops and the *setup* time of the flip flop.

Since flip flops have unknown states when they are first switched on, we have to properly initialize them:

- Reset signal the flip flop output to 0 (unrelated to the R input!)
- Synchronous reset - Output is reset to 0 only on the active edge clock.
- Asynchronous reset - The output is reset to 0 immediately, *independent* of the clock signal.

Example: Registers

Shift Register - A series of D flip-flops that can store a multi-bit integer value. Data is shifted into the register one bit at a time.

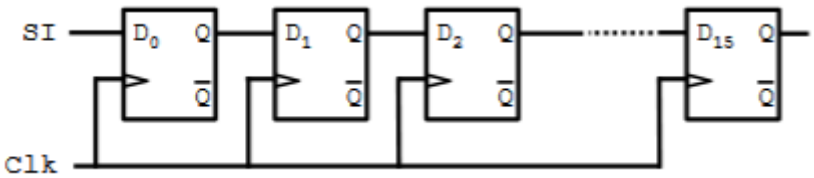


Figure 24: Pasted image 20250223230706.png

Load Register - A register that feeds values in all at once (i.e., to each flip-flop) via a splitter. Uses a "D flip-flop with enable" to allow/forbid loading of values.

Example: Counters

We can connect a series of T flip-flops together to create a counter. They are often implemented with a **parallel load** and **clear** inputs.

Ripple Counter - An asynchronous circuit where the **output** of the flip flop is connected to the clock input of the next.

- The timing is not synchronized with the rising clock pulse.
- Cheap to implement, but unreliable for timing; will cause propagation delays!

Synchronous Counters - Has a slight delay, but has a more reliable timing. These circuits can be even more synchronized by having AND gates for *all* previous flip-flops.

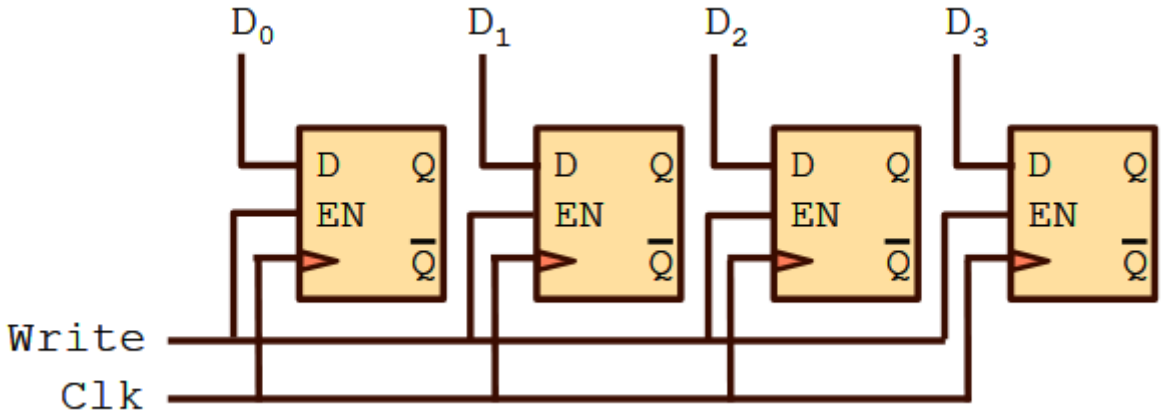


Figure 25: Pasted image 20250223230947.png

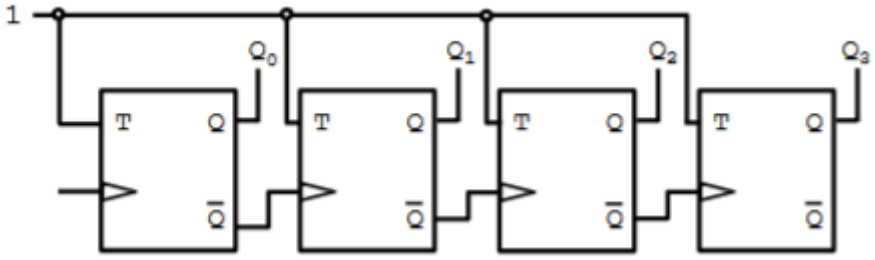


Figure 26: Pasted image 20250224001017.png

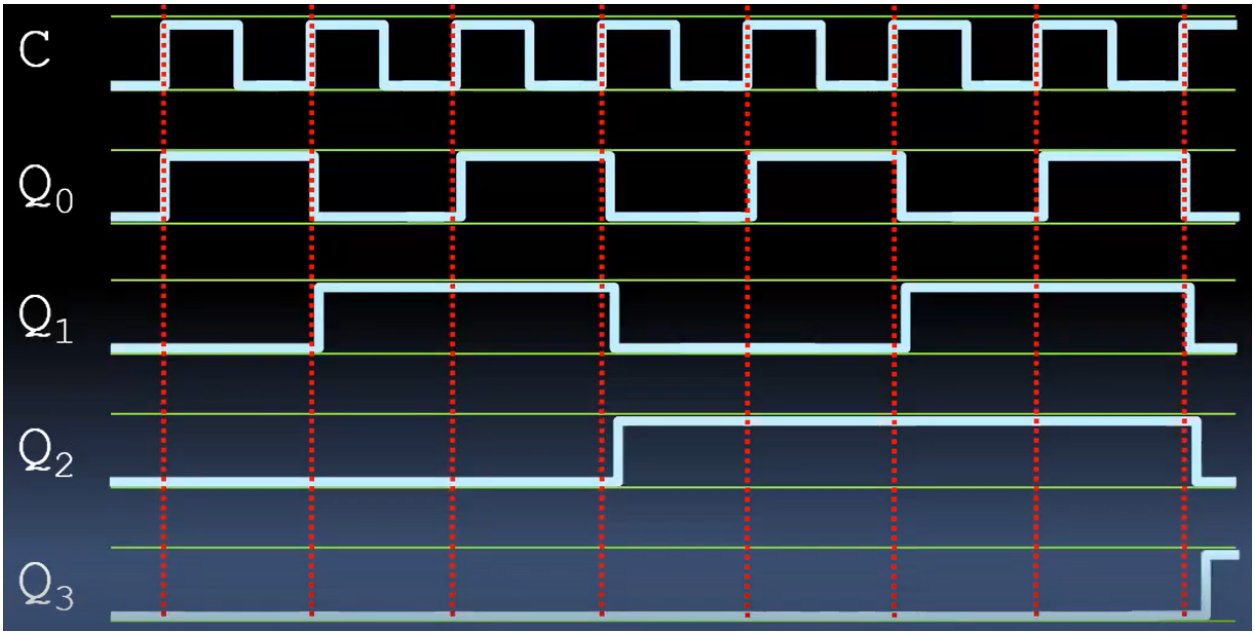


Figure 27: Pasted image 20250413124208.png

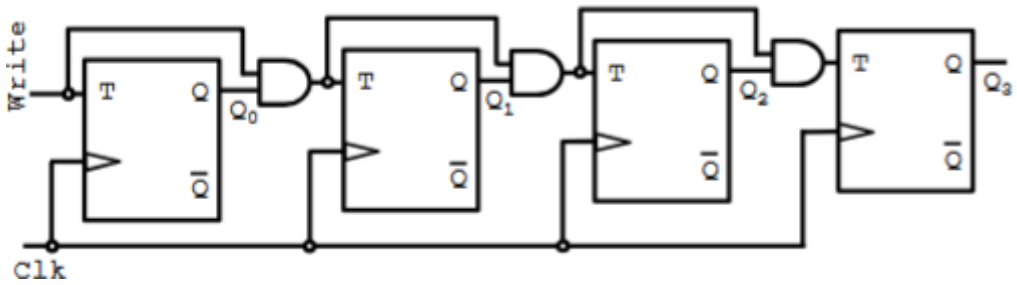
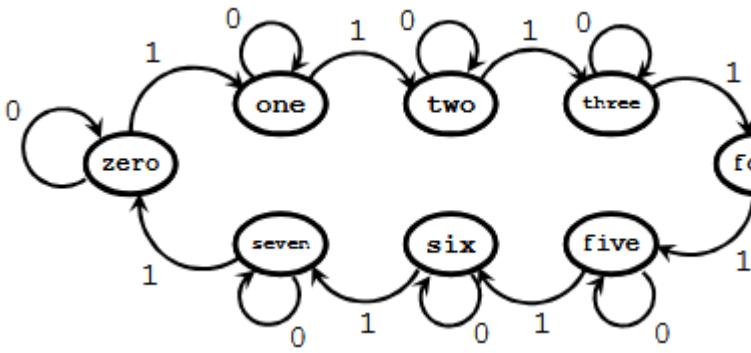


Figure 28: Pasted image 20250224001040.png

Finite State Machines

Counters and registers are examples of how flip-flops can implement circuits that store values as states.

- Each **state** in the counter is the current number being stored. On each clock tick, the circuit **transitions** from one state to the next based on the inputs.



State Table - An illustration of how the states of the circuit change with various input changes. Transitions are understood to take place on the clock ticks.

Finite State Machine - A model for a circuit design, representing internal states of the circuit stored in the flip-flop values.

- More generally, we can consider it the finite set of states, or, the set of transitions between states triggered by inputs to the state machine. They have start and end states, and output values associated with each state or transition. For example, a traffic light:

[!NOTE] FSM Example - Alarm Clock **Internal state description -** Starts in neutral state until the timer goes off. The clock moves to the alarm state and continues until: - Snooze button is pushed (move to snooze state) - Alarm is turned off (move to neutral) - Timer goes off again (move to neutral)

In the snooze state, the clock returns to the alarm state when the timer signal goes off again.

FSM Design Design steps:

1. Draw a state diagram.
2. Derive state table from state diagram.
3. Assign flip-flop configurations to each state.
    - Number of flip-flops needed: $\lceil \log_2(\#\text{ of states}) \rceil$. Each flip flop holds two values/states (0 and 1).
4. Redraw state table with the flip-flop values.
5. Derive the combinational circuit for the output and for each flip-flop input.
    - To derive the circuitry needed, use one of the two:

State	Write	State
zero	0	zero
zero	1	one
one	0	one
one	1	two
two	0	two
two	1	three
three	0	three
three	1	four
four	0	four
four	1	five
five	0	five
five	1	six
six	0	six
six	1	seven
seven	0	seven
seven	1	zero

Figure 29: Pasted image 20250224001708.png

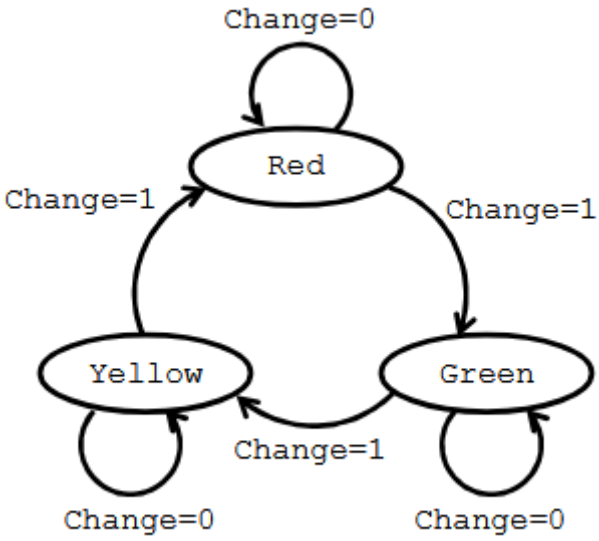
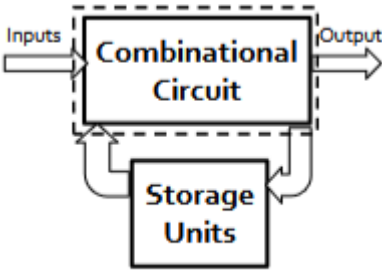
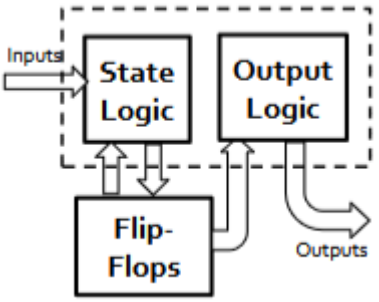
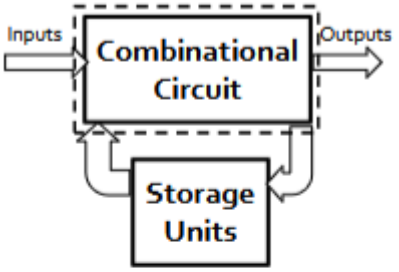


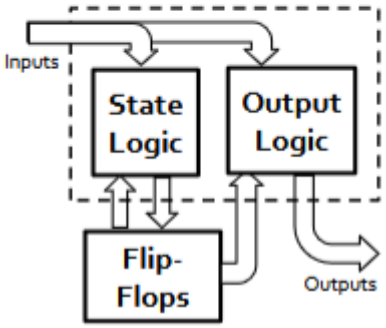
Figure 30: Pasted image 20250224002041.png





– **Moore Machine -** Output of the FSM depends solely on the current state.





– **Mealy Machine -** Output depends on the state and the current input.

When assigning states, be sure to consider the issue with *timing* of the states. For example, **race condition:** if two inputs change "at the same time". If the first flip-flop were to change first, the output would go high for an instant, causing unexpected behaviour. There are two possible solutions to solving this:

1. When possible, make flip-flop assignments differ by at most one flip-flop value.
2. If the intermediate states are unused in the state diagram, set the output for these states to provide the output you need. This may require more flip flops.

5 - ALUs, Registers, Memory

Now that we know how to make devices like adders and registers, we can now make a **microprocessor** - a digital device that processes input, can store values and produces output, according to a set of on-board instructions.

- Microprocessors are the basis of all computing (since the 1970s!).

Microprocessors

Processors can be deconstructed into three core units:

- **Controller Unit** - Orchestrates the actions that take place in the datapath; a big FSM instructing the datapath to perform the appropriate actions.
- **Datapath** - where all computations take place.
  - **Storage Unit**
  - **Arithmetic Unit**

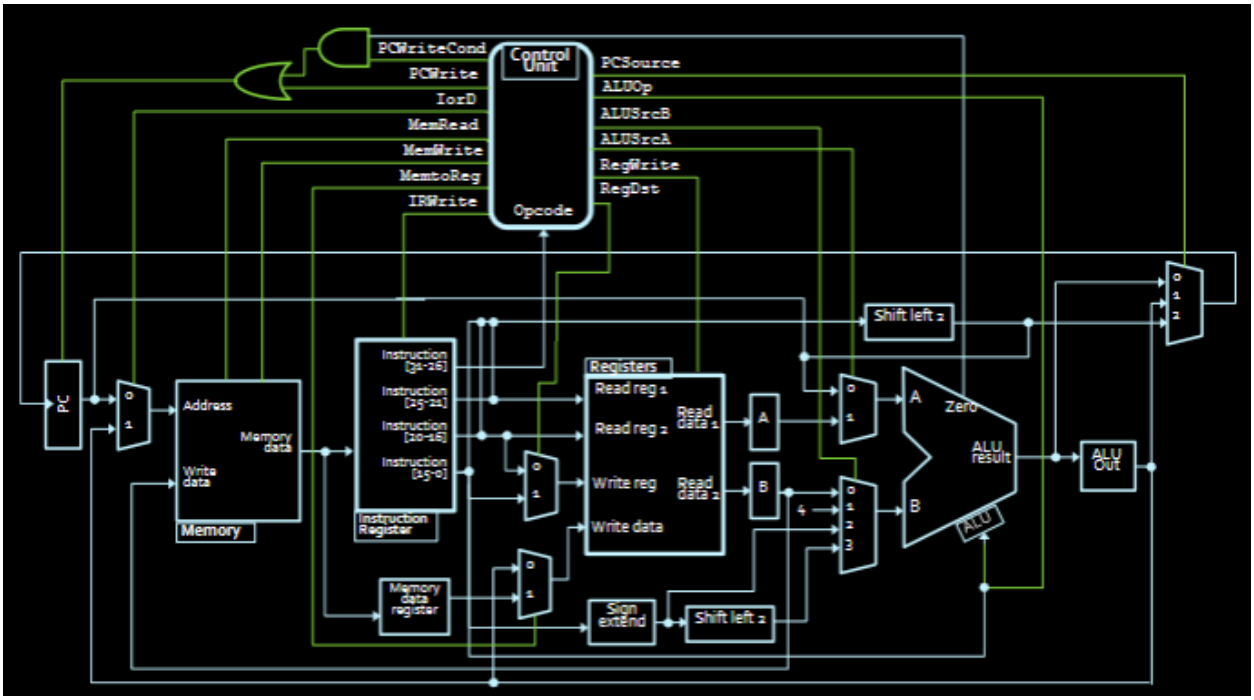


Figure 31: Pasted image 20250413171532.png

Computing Datapaths

Suppose we have an external value X given from outside the datapath, and we need to compute an equation. We must coordinate the datapath components:

- ALU (to add/subtract/multiply values).
- Multiplexers (to determine which inputs are sent to the ALU).
- Registers (to hold values in the calculation).

Example: Calculating $x^2 + 2x$.

1. Load X into registers RA and RB.
2. Multiply RA by RB, store result in RA.
3. Add X to RA, store result in RA.
4. Add X to RA, then output result. On a lower level, we need the control signals that are sent by a giant FSM.

- SelxA, SelAB - Control MUX outputs (ALU inputs)
- ALUop - Controls ALU operation
- LdRA, LdRB - Controls loading for registers RA, RB Then:

High-Level Step	Control Signals
Load X into RA and RB	SelxA = 0, ALUop = A, LdRA = 1, LdRB = 1
Multiply RA and RB	SelxA = 1, SelAB = 1, ALUop = Mult, LdRA = 1
Add X to RA	SelxA = 0, SelAB = 0, ALUopp = Add, LdRA = 1
Add X to RA again	SelxA = 0, SelAB = 0, ALUop = Add

Microprocessor Components

Arithmetic Unit The **arithmetic logic unit** (ALU) performs arithmetic operations and logical operations.

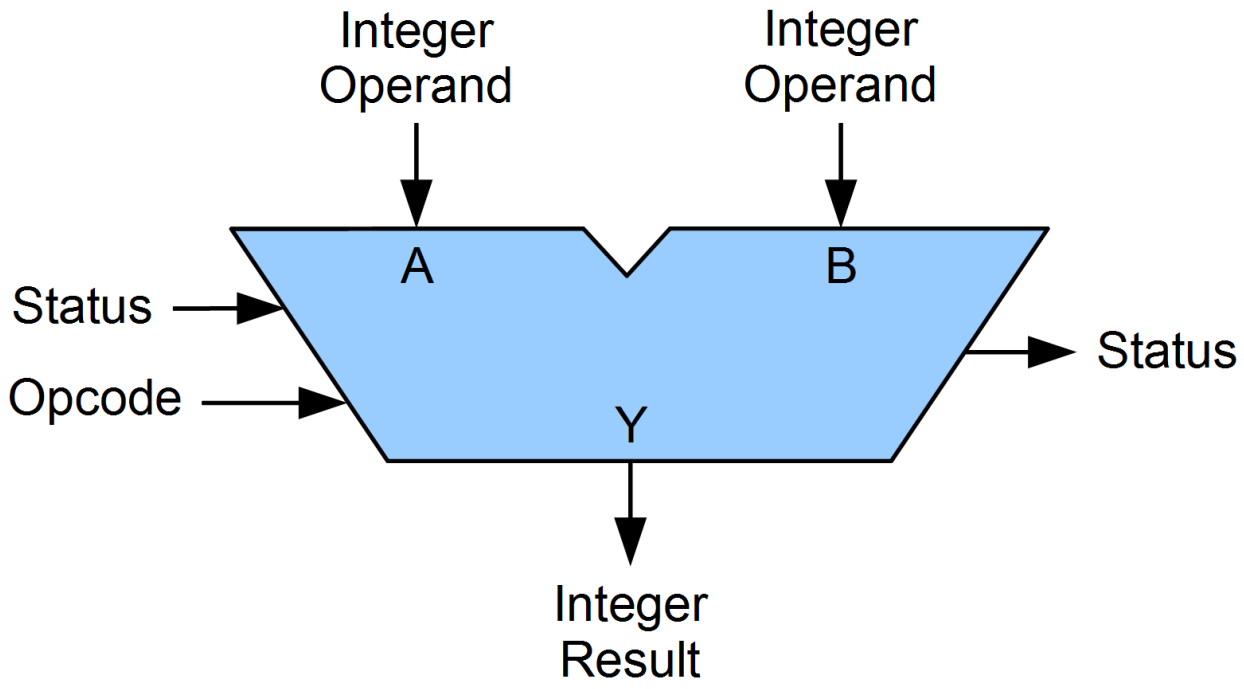


Figure 32: ALU_block.gif

- The input Status (or C_{in}) is is a carry bit for incrementing an input value or the overall result.
- The input Opcode (or S) is the select bit specifying the operation to perform.
- The output Status (or $VCNZ$) indicates special conditions in the arithmetic result.
  - V - Overflow indicator
  - C - Carry-out bit
  - N - Negative indicator
  - Z - Zero-condition indicator
- The output Y or G is the data output.

Operation Selection:

With just the arithmetic and logic units, our ALU looks like the following:

Select		Input	Operation	
S_1	S_0	Y	C_{in}=0	C_{in}=1
0	0	All 0s	G = A (transfer)	G = A+1 (increment)
0	1	B	G = A+B (add)	G = A+B+1
1	0	$\overline{B}$	G = A+$\overline{B}$	G = A+$\overline{B}$+1 (subtract)
1	1	All 1s	G = A-1 (decrement)	G = A (transfer)

Figure 33: Pasted image 20250413142931.png

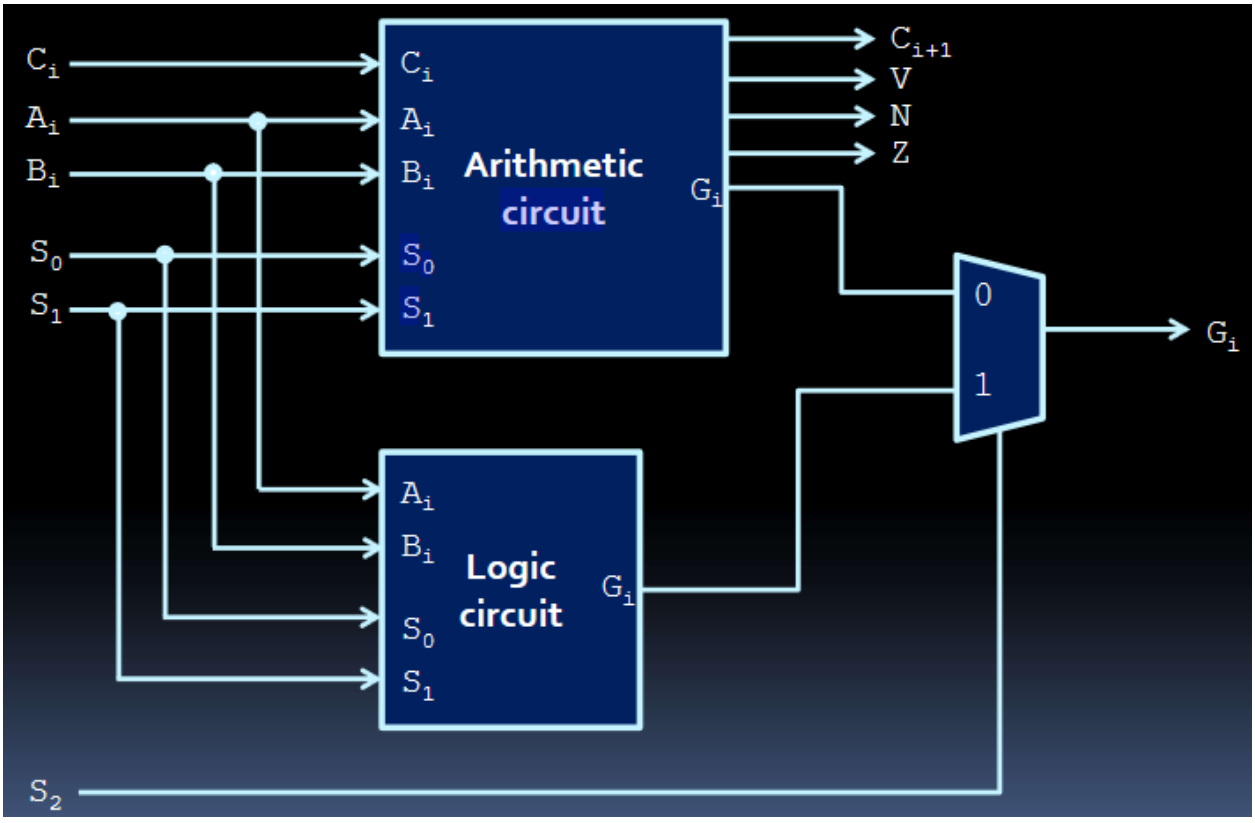


Figure 34: Pasted image 20250413143303.png

For multiplication/division, there are 3 major ways that multiplication can be implemented in circuitry:

- Layered rows of adder units - $O(N^2)$ size!
- An adder/shifter circuit - $O(N^2)$ time!
- Booth's Algorithm - The best option in terms of space and when shifting is cheaper than adding.

Binary Multiplication

We can write $5 \times 6 = 30$ in binary:

$$\begin{array}{rr} & 101 \\ \times & 110 \\ \hline & 000 \\ & 101 \\ & 101 \\ \hline & 11110 \end{array}$$

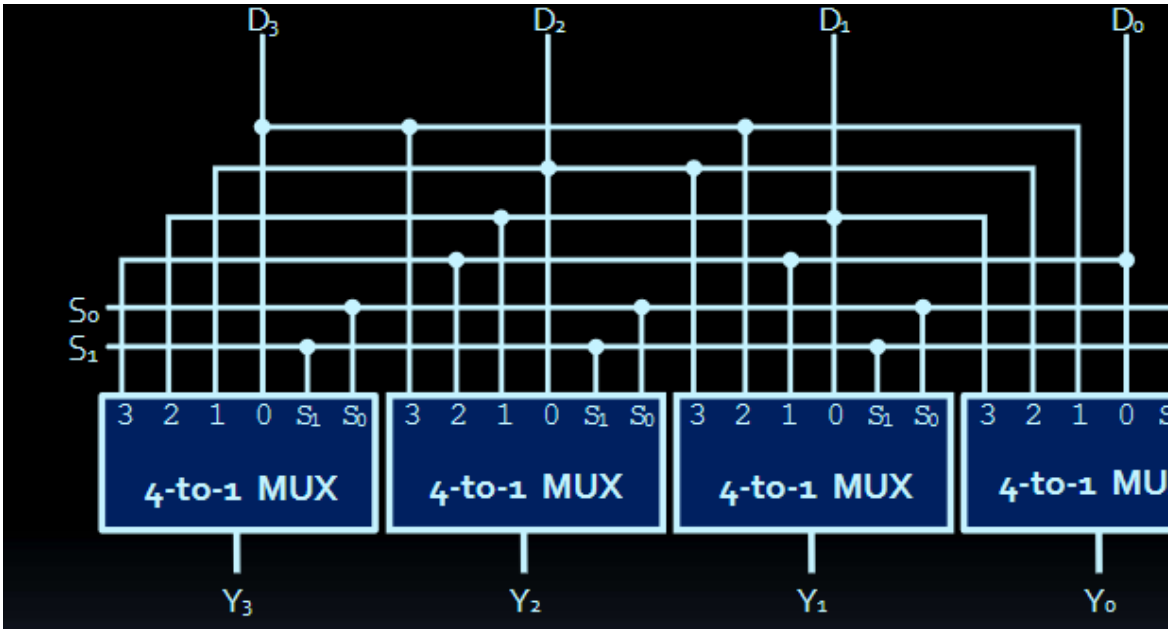
Booth's Algorithm - A cheap way of multiplying, used on cases where two neighbouring digits in an operand are different.
- If digits at i and $i-1$ are 0 and 1, the multiplicand is added to the result at position i.
- If digits at i and $i-1$ are 1 and 0, the multiplicand is subtracted from the result at position i. The result is always a value whose size is the sum of the sizes of the two multiplicands.

Example of Booth's:

$$\begin{array}{rrr} & 01010010 & \text{(B)} \\ \times & 00011110 & \text{(A)} \\ \hline & 01010010 & \text{(Add B)} \\ + & 1111\,10101110 & \text{(Subtract B, sign extend)} \\ \hline & 0100110011100 & \end{array}$$

To make Booth's work in hardware:
1. Have hardware set up to compare neighbouring bits at every position in A, with adders in place for where bits don't match (a lot of hardware).
2. Have hardware set up to compare two neighbouring bits, and have them move down through A, looking for mismatched pairs (but hardware doesn't actually do that).

Multiplication isn't as common of an operation, and most common multiplication/division operators are powers of 2. For this, use a shift register instead of the multiplier circuit.



Barrel Shifter Unit
A barrel shifter **shifts and rotates** D to the left by S bits.

- If S_1D_0 is 01, then $Y = D_2D_1D_0D_3$.
- If S_1D_0 is 11, then $Y = D_0D_3D_2D_1$. This is a purely combinational circuit.

Storage Unit The storage unit comprises of the register file and memory for use by the CPU, and registers that store single values.

Register Files **Register file -** Small number of fast memory units that allow multiple values to be read and written simultaneously.

- Typically used for quick calculations; contains 32 registers.
- All the registers share a single set of input/output wires.
- Write to register with the Write signal to the corresponding number (i.e., **address**).
  - Turn on the Write signal using a demux to specify a single register based on the address; a **one-hot decoder**.
  - One-hot decoders take in a m-bit binary address to activate one of the 2^m registers in the register file.
- Read from registers using a mux.

[!NOTE] Electronic Memory (RAM) Like register files, main memory is made up of a decoder and rows of memory units. - When the read/write signal is low, n data bits are written in parallel to the m-bit address provided. - When read/write signal is high, n data bits are read from memory at the m-bit address. Then: - There are 2^m rows (where m is the address width). - Each row contains n bits (where n is the data-width). - The size of the memory is $2^m \cdot n$ bits, or $\frac{2^m \cdot n}{8}$ bytes.

Connecting to Memory **Main Memory -** Larger grid of memory cells that are used to store the main information to be processed by the CPU.

Memory values are read to the registers and then processed by the ALU; results are eventually sent back to memory. Memory units use the same n-bit wires to both send and receive data, but we do not want to write to wires at the same time.

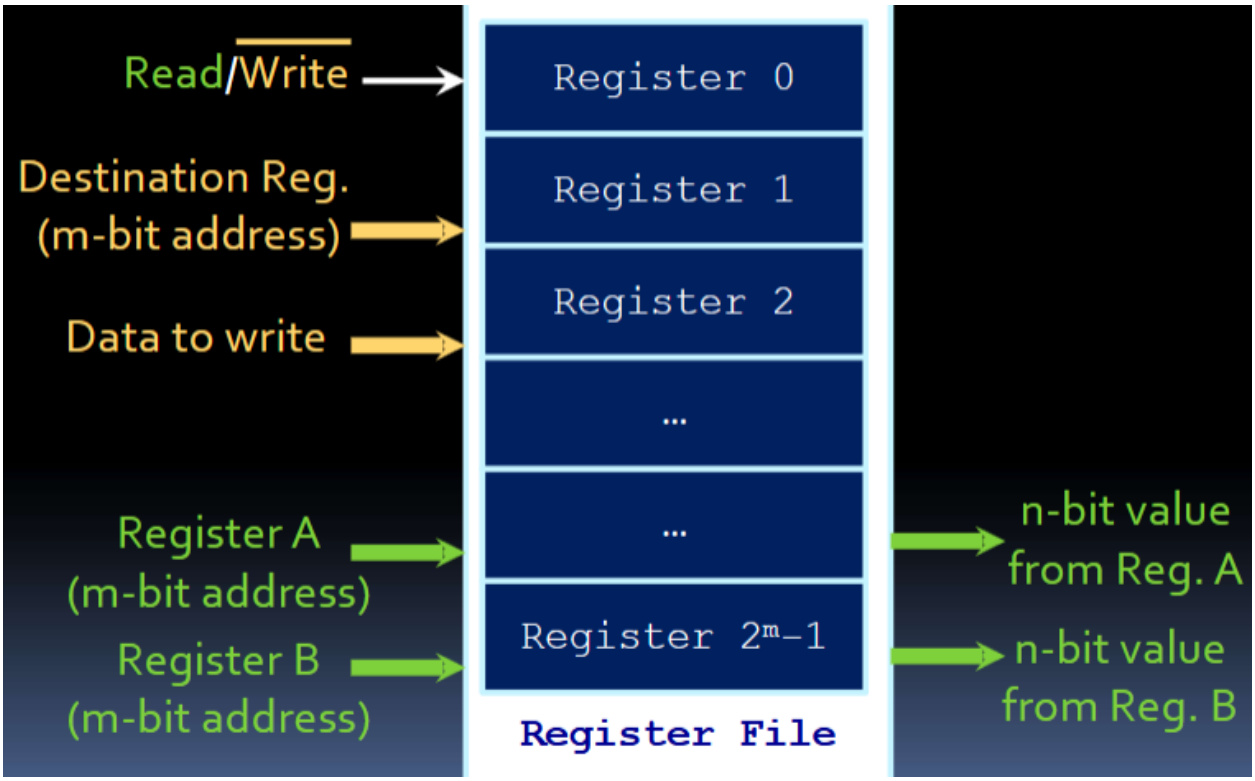


Figure 35: Pasted image 20250413162247.png

- Use **tri-state buffers** to control reading/writing memory locations; when WE (write enable) is low, the buffer output is a "high impedance" signal (neither high voltage or ground (Z)).

WE	A	Y
0	X	Z
1	0	0
1	1	1

Tri-state buffers allow us to use a **bus** (data bus) to communicate in both directions between memory and the processor. Has high impedance when:

1. The processor is writing to memory.
2. The memory location is not being accessed.

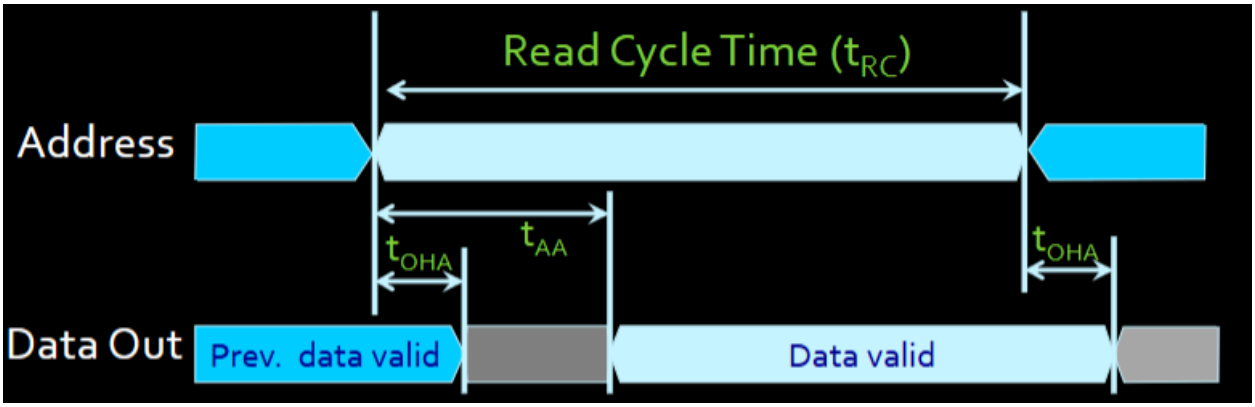
Memory vs. Registers:

- Similarities - Both store data values, both use addresses to specify which value to access.
- Differences -
    1. Memory is *much* bigger, houses most of the long-term data values being used by a program.

Registers are local data values, used internally by the processor to perform operations.
2. Register access is immediate. Memory units are separate from the main processor; requires more time for a single access.
3. Registers are stored apart from each other, memory is one large block of space where addresses are locations within the block.
4. Each register address refers to a single 32-bit register. Each memory address refers to one byte location (8-bits).

Memory Timing Issues Memory access can take multiple clock cycles to complete (e.g., because of delays from flip-flops). The "data bus" signal indicates when valid data values are available for reading/writing, since they account for delays:

- Time needed to send address/memory to memory.
- Time to turn Read/Write signal on.
- Time to keep data/address on after read or write.



The above is an **address-controlled read cycle**:

- t_{RC} - read cycle time (minimum time needed between two read cycles).
- t_{AA} - address access time (time between sending data address and receiving data values).
- t_{OHA} - output hold time.

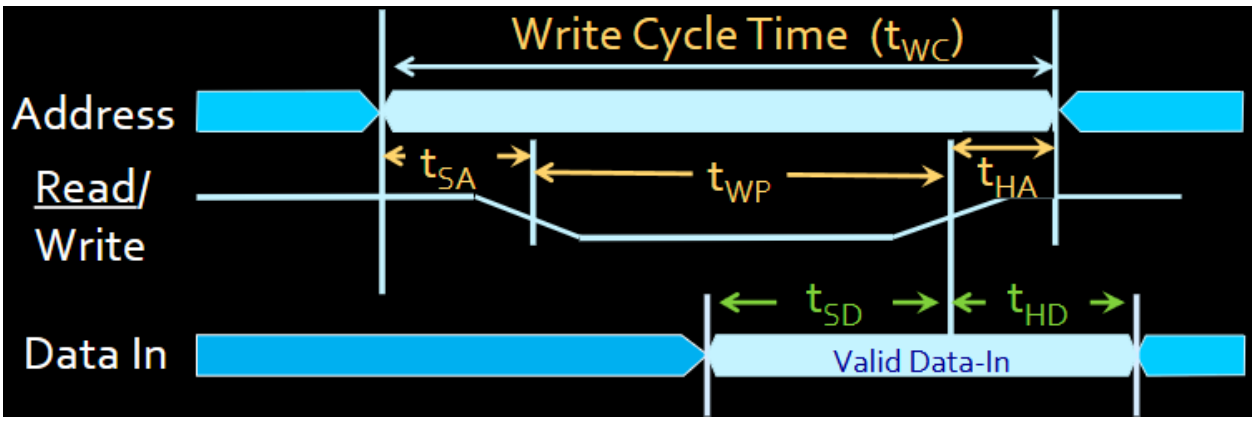


Figure 36: Pasted image 20250413164737.png

- t_{SA} - Address setup time (time for address to become stable before enabling write signal).
- t_{HA} - Address hold time (stability before enabling write signal).

- t_{WP} - Write pulse width
- t_{SD} - Data setup time (to write end); the time needed for data-in value to be written to memory.
- t_{HD} - Data hold time (from write end); the time the data-in value remains available after the write signal.

Load-store Architecture - When we fetch values from memory into the registers, process them using the ALU, and return values back to memory.

Control Unit Control units are full datapath FSMs that:

- Support hundreds of operations.
- Returns to the start state when an operation is complete.
- Has a 6-bit "operation code" input.
- Is responsible for setting up next opcode input.
  - Note the next opcode needs to be ready before the branch terminates and the FSM returns to the start state.

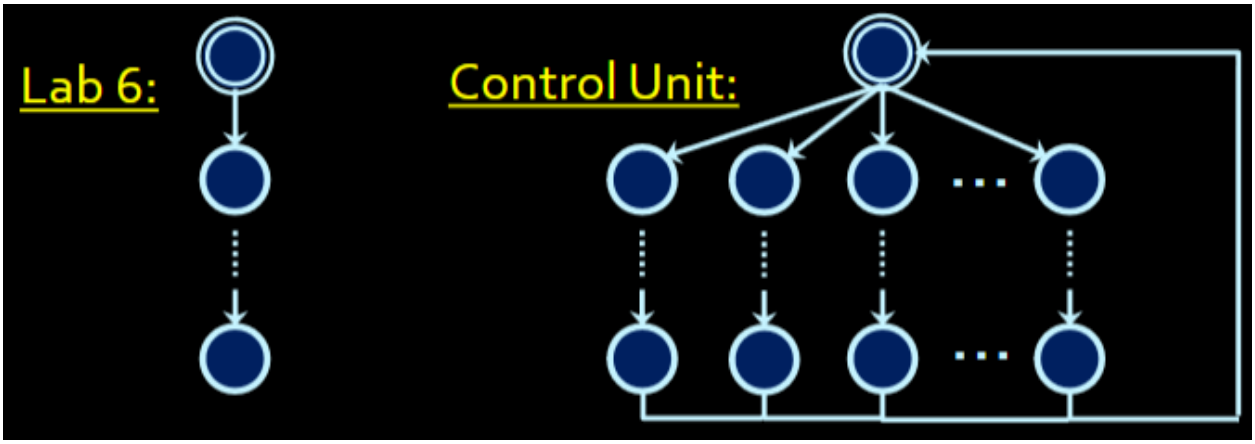


Figure 37: Pasted image 20250413165333.png

Signals - Control units have 13 signals:

- MemRead
- MemWrite
- MemToReg
- ALUOp
- ALUSrcA
- ALUSrcB
- RegWrite
- PCWrite
- PCWriteCond
- IorD (instruction or data)
- IRWrite

- PCSource

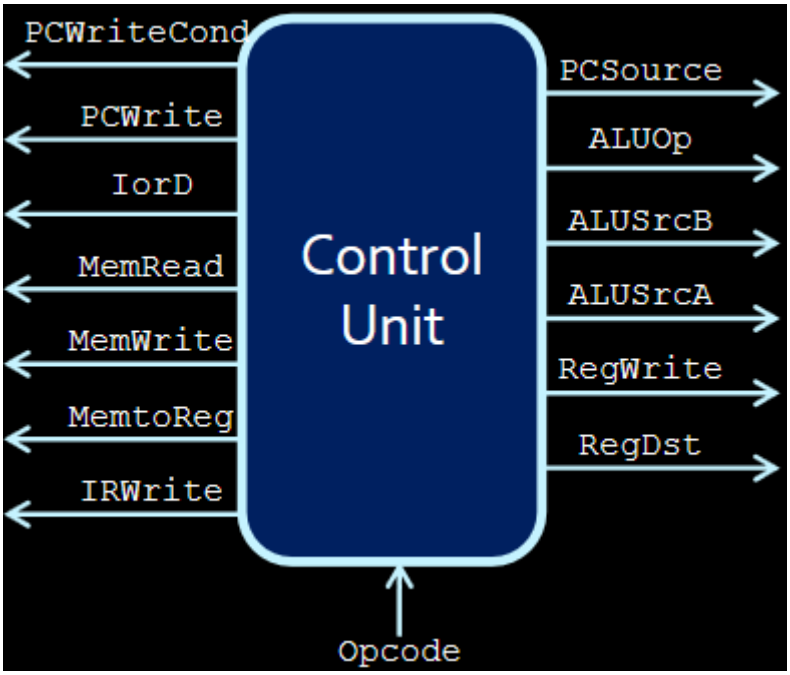


Figure 38: Pasted image 20250413170028.png

Instruction Architecture The control unit's opcode comes from a 32-bit instruction in the **instruction register**.

Instructions - 32-bit binary strings that encode:

- The operation to perform (first 6 bits; opcode).
- Other details needed to perform it (remaining 26 bits). Instructions and data values are both stored in main memory, and are stored separately from data values (`.text` segment of memory, whereas data values are the `.data` segment). The first instruction to be executed in a program is identified with a label `main`.

Instruction Execution: All programs are translated into a sequence of instructions, where the control unit continuously does the following:

1. Instruction fetch - Brings next instruction from memory and place into the instruction register.
2. Decode instruction - The operation to perform.
3. Execute instruction - Reads values of any registers, computes any computations needed in the ALU, accesses memory if we need to read/write, writes back any data that needs to be stored.
4. Move (or jump) to the next instruction in memory. Steps 1 and 4 assume that the control unit knows where to find the current instruction in memory, i.e., a special **program counter (PC)** register, which stores the memory address of the current instruction.

To update the program counter, increment by **4** each time it needs to fetch the next instruction.

- Memory locations are byte-addressable; each byte (8 bits) has its own unique address.
- Instructions are 32 bits long (4 bytes), instructions would be at 0, 4, 8, 12, 16, etc.

Example control unit signal: `addi $t7, $t0, 42` is equivalent to set register 15 equal to register 8 + 42. Then:

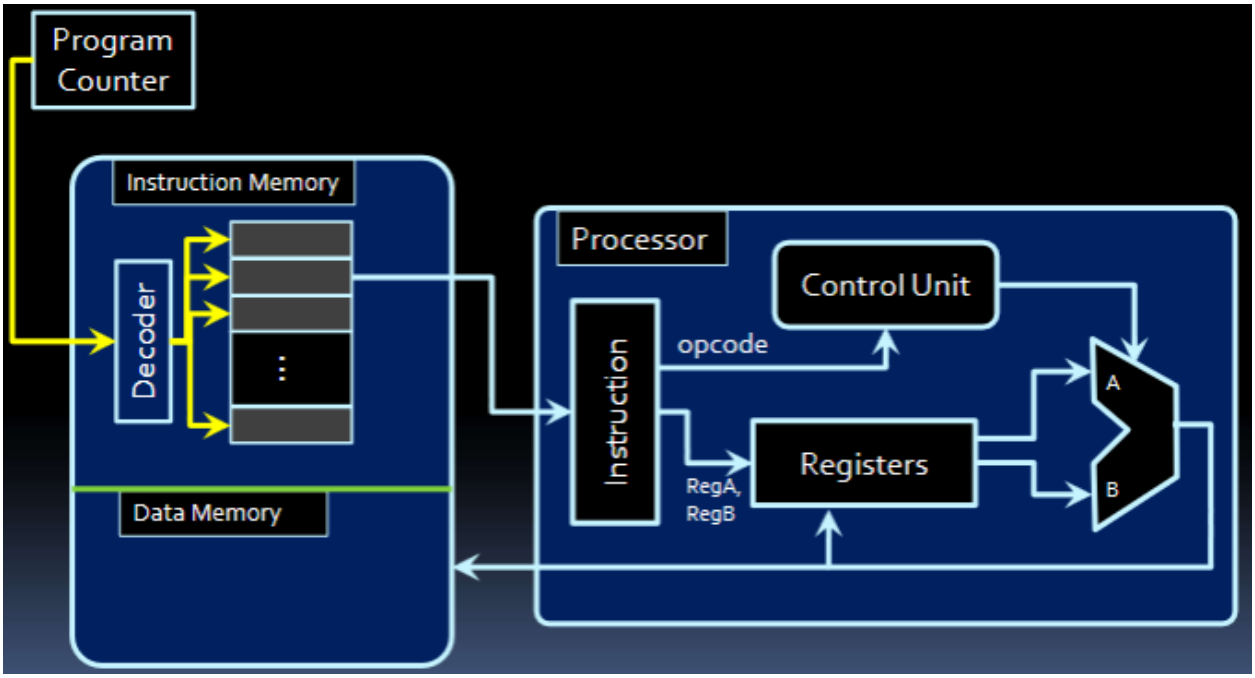


Figure 39: Pasted image 20250413170949.png

- PCWrite = 0
- PCWriteCond = 0
- IorD = X
- MemWrite = 0
- MemRead = 0
- MemToReg = 0
- IRWrite = 0
- PCSource = X
- ALUOp = Add
- ALUSrcA = 1
- ALUSrcB = 10
- RegWrite = 1
- RegDst = 0

6 - Assembly Language

Machine Code Instructions

Operations are performed by:

- The **instruction register** sending instruction components to the processor.
- The **control unit** based on the opcode value sending a sequence of signals to the rest of the processor.

Machine code instructions contain all details about a processor operation such as

- What operation is being performed (opcode),
- What registers are used in the operation
- What other information may be needed (e.g., immediate/shift values). The instruction is specified in the **Instruction Set Architecture (ISA)** implemented by a given processor. Each instruction (control word) is broken down into sections that contain information needed to execute the operation

Assembly Language - Each processor has its own language for representing instructions as user-level code words.

- `add $t3, $t1, $t2` in Assembly is equivalent to `000000 01001 01010 01011 XXXXX 100000` in machine code instruction.

MIPS - Microprocessor without Interlocked Pipeline Stages.

- A type of assembly language.
- A type of RISC (Reduced Instruction Set Computer) architecture.
- A simple and fast set of instructions in 32 bits.

MIPS is register-to-register; almost all operations rely on register data. It provides 32 registers with numerical and label names.

- Register `$0` (`$zero`) - Value 0.
- Register `$1` (`$at`) - Reserved for assembler.
- Registers `$28-$31` (`$gp, $sp, $fp, $ra`) - Memory and function support.
- Registers `$26-$27` - Reserved for OS kernel.
- Registers `$2-$3` (`$v0, $v1`) - Return values.
- Registers `$4-$7` (`$a0-$a3`) - Function arguments.

Instructions

R-Type Instruction - An operation where all inputs and outputs are all register. R-type instructions begin with an opcode of `000000`.

- Example: `add $t3, $t1, $t2`
- 9th register = `$t1` = 01001; 10th register = `$t2` = 01010, 11th register = `$t3` = 01011. `add` = 000000 100000.
- Machine code: `000000 01001 01010 01011 100000`

I-Type Instruction - An operation that involves constants. The constant is encoded in the *last 16 bits* of the instruction.

- Example: `addi $t2, $t1, 42`
- `addi` is `001000`, 42 in binary is `101010`.
- Machine code: `001000 01001 01010 0000000000101010`

J-Type Instruction - Jump to a location in memory encoded by the last 26 bits of the instruction. Stored as a label, which is resolved when the assembly program is compiled.

- Example: `j main`
- `j` is `000010`. Address takes up the remaining 26 bits.

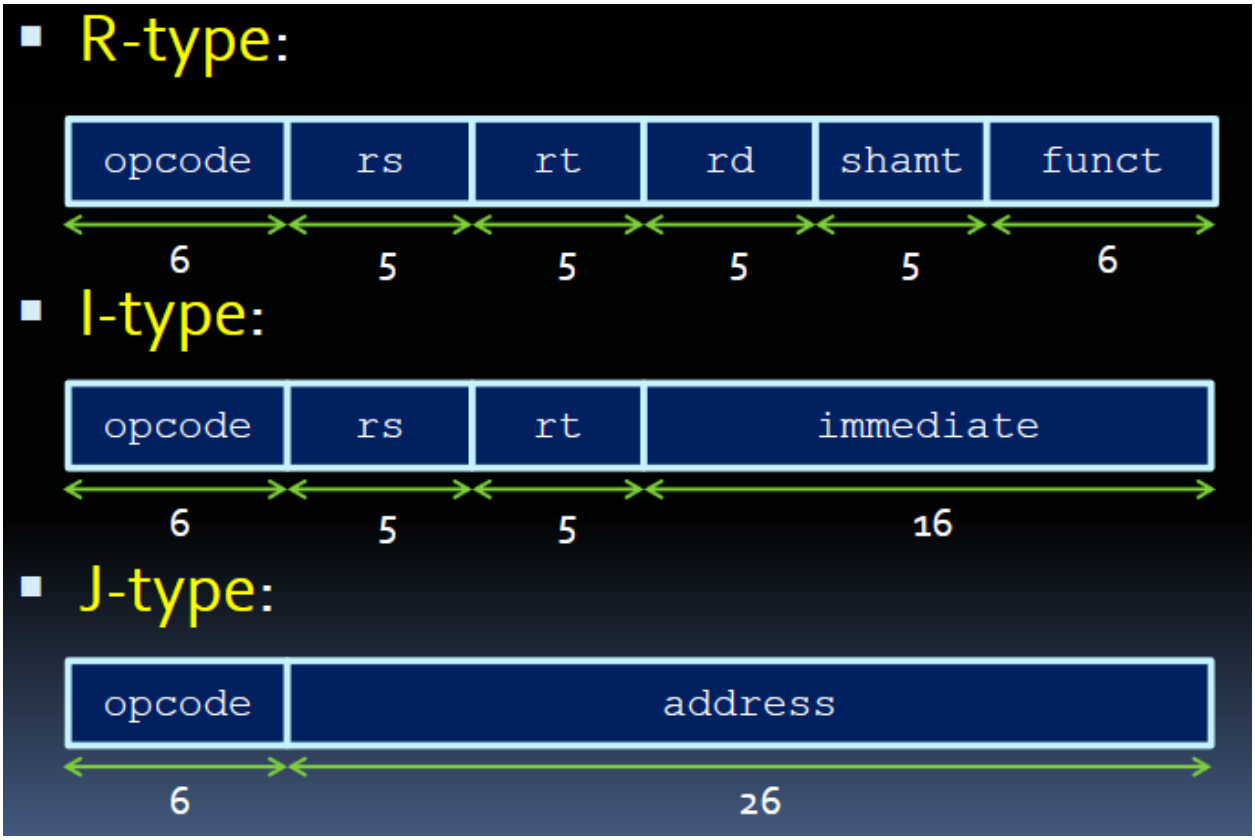


Figure 40: Pasted image 20250413183028.png

Assembly Language Instructions

Assembly - Lowest-level programming language. Compilers will translate high-level program commands into assembly commands, which is then converted into machine code.

Arithmetic Instructions:

Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t
addu	100001	$d, $s, $t	$d = $s + $t
addi	001000	$t, $s, i	$t = $s + SE(i)
addiu	001001	$t, $s, i	$t = $s + SE(i)
div	011010	$s, $t	lo = $s / $t; hi = $s % $t
divu	011011	$s, $t	lo = $s / $t; hi = $s % $t
mult	011000	$s, $t	hi:lo = $s * $t
multu	011001	$s, $t	hi:lo = $s * $t
sub	100010	$d, $s, $t	$d = $s - $t
subu	100011	$d, $s, $t	$d = $s - $t

Note: "hi" and "lo" refer to the high and low bits referred to in the register slide.
"SE" = "sign extend".

Figure 41: Pasted image 20250413183430.png

R-Type Arithmetic	I-Type Arithmetic
add, addu	addi
div, divu	addiu
mult, multu	
sub, subu	

Logical Instructions:

Shift Instructions:

Data Movement Instructions: The R-type instructions for operating on the HI and LO registers.

ALU Instructions: Most are R-type instructions; except for I-type instructions like `addi` and `ori`.

Jump and Branch Instructions **Jump Instructions:** If/else, for/while loops, etc.

- `jal` stands for "jump and link"; the register `$31` (`$ra`) stores the address used when returning from a subroutine (versus the next instruction to run).
- `jr` and `jalr` move the address stored in `$ra` and `$t0` into the program counter. The next instruction will be at this new address, and the program will continue from there.

Logical instructions

Instruction	Opcode/Function	Syntax	Operation
and	100100	$d, $s, $t	$d = $s & $t
andi	001100	$t, $s, i	$t = $s & ZE(i)
nor	100111	$d, $s, $t	$d = ~($s | $t)
or	100101	$d, $s, $t	$d = $s | $t
ori	001101	$t, $s, i	$t = $s | ZE(i)
xor	100110	$d, $s, $t	$d = $s ^ $t
xori	001110	$t, $s, i	$t = $s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

Figure 42: Pasted image 20250413183915.png

Shift instructions

Instruction	Opcode/Function	Syntax	Operation
sll	000000	$d, $t, a	$d = $t << a
sllv	000100	$d, $t, $s	$d = $t << $s
sra	000011	$d, $t, a	$d = $t >> a
srav	000111	$d, $t, $s	$d = $t >> $s
srl	000010	$d, $t, a	$d = $t >>> a
srlv	000110	$d, $t, $s	$d = $t >>> $s

Note: srl = "shift right logical", and sra = "shift right arithmetic".
The "v" denotes a variable number of bits, specified by $s.
a is a shift amount, and is stored in shamt when encoding the R-type machine code instructions.

Figure 43: Pasted image 20250413183930.png

Instruction	Opcode/Function	Syntax	Operation
mfhi	010000	$d	$d = hi
mflo	010010	$d	$d = lo
mthi	010001	$s	hi = $s
mtlo	010011	$s	lo = $s

Figure 44: Pasted image 20250413184010.png

Instruction	Opcode/Function	Syntax	Operation
j	000010	label	pc = (pc & 0xF0000000) | (i<<2)
jal	000011	label	$31 = pc+4; pc = (pc & 0xF0000000) | (i<<2)
jalr	001001	$s	$31 = pc+4; pc = $s
jr	001000	$s	pc = $s

Figure 45: Pasted image 20250413184945.png

- `j` and `jal` get the address from the instruction. To get the full 32-bit address:
  1. Use trailing 0's; the 26 bits in the instructions will store the new PC address (minus the last two zeros at the end).
  2. Use leading bits: Use the first 4 bits of the previous PC value (hence the formula).

Branch Instructions: Used for implementing `if` statements and `while` loops.

Instruction	Opcode/Function	Syntax	Operation
beq	000100	$s, $t, label	if ($s == $t) pc += i << 2
bgtz	000111	$s, label	if ($s > 0) pc += i << 2
blez	000110	$s, label	if ($s <= 0) pc += i << 2
bne	000101	$s, $t, label	if ($s != $t) pc += i << 2

Figure 46: Pasted image 20250413185402.png

```
main:   beq $t0, $t1, end      # check if $t0 == $t1
        ...                    # if $t0 != $t1, then
        ...                    # execute these lines

 end:   ...                    # if $t0 == $t1, then
        ...                    # execute these lines
```

Use `bne` to mimic if statement behaviour:

```
main:   bne $t0, $t1, end      # check if $t0 == $t1
        ...                    # if $t0 == $t1, then
        ...                    # execute these lines

 end:   ...                    # if $t0 != $t1, then
        ...                    # execute these lines
```

Branch statements are I-type instructions; the immediate value (`i`) is a 16-bit offset (relative address) to add to the current instruction if the branch is satisfied.

- Stored as the number of instructions (not bytes).
- `i` can be positive (forward) or negative (backward). Can be calculated as `i = (label location - (current PC)) >> 2`. When the branch condition is met, the branch is **taken**.

Comparison Instructions:

Instruction	Opcode/Function	Syntax	Operation
slt	101010	$d, $s, $t	$d = ($s < $t)
sltu	101001	$d, $s, $t	$d = ($s < $t)
slti	001010	$t, $s, i	$t = ($s < SE(i))
sltiu	001011	$t, $s, i	$t = ($s < ZE(i))

Figure 47: Pasted image 20250413190500.png

If Statements: Done using a `beq` or `bne` instruction. “‘MIPS Assembly

Example: if (i == j) i++ else j+=i

$t1 = i, $t2 = j

main: bne $t1, $t2, END # branch if (i != j) addi $t1, $t1, 1 # i++ END: add $t2, $t2, $t1 # j = j + i

If/Else Statements:
```MIPS Assembly

## Example: If (i==j) i++ else i--; then j+=i

## $t1 = i, $t2 = j
main:
    bne $t1, $t2, ELSE
    addi $t1, $t1, 1
    j END
ELSE:
    addi $t1, $t1, -1
END:
    add $t2, $t2, $t1
```

Multiple If Conditions: “‘MIPS Assembly

Example: if (i==j || i==k) { ... }

$t1 = i, $t2 = j, $t3 = k

main: beq $t1, $t2, IF (branch if i == j) bne $t1, $t3, ELSE (branch if i != k) IF: addi $t1, $t1, 1 j END ELSE: addi $t1, $ti, -1 END: add $t2, $t1, $t3

If we used AND statements:
```MIPS Assembly

## Example: if (i==j && i==k) { ... }

## $t1 = i, $t2 = j, $t3 = k
main:
    bne $t1, $t2, ELSE   (branch if i != j)
    bne $t1, $t3, ELSE   (branch if i != k)
IF:
    addi $t1, $t1, 1
    j END
ELSE:
    addi $t1, $ti, -1
END:
    add $t2, $t1, $t3
```

While Loops: Look similar to `if` statements. "'MIPS Assembly

Example: i=0; while (i<100) i++

main: add $t0, $zero, $zero addi $t1, $zero, 100 START: beq $t0, $t1, END addi $t0, $t0, 1 j START END:

For Loops:
```MIPS

## Example: for (i=0, j=0; i<00; i++) j=j+i

## $t0 = i, $t1 = j
main:
    add $t0, $zero, $zero   # set $t0 to 0
    add $t1, $zero, $zero   # set $t1 to 0
    addi $t9, $zero, 100    # set $t9 to 100
START:
    beq $t0, $t9, EXIT      # branch if i=100
    add $t1, $t1, $t0       # j = j + i
UPDATE:
    addi $t0, $t0, 1        # i++
    j START
EXIT:
```

Memory Instructions Memory Operations: I-type instructions.

- **Load -** Read operations.
- **Store -** Write operations.

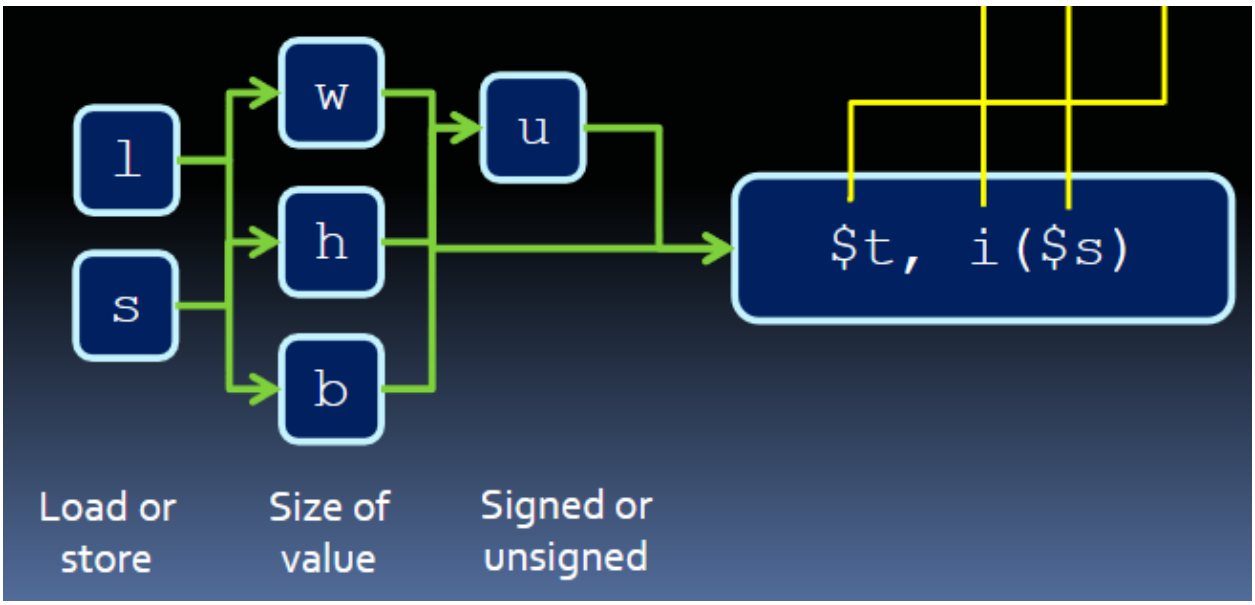


Figure 48: Pasted image 20250413192304.png

Instruction	Opcode/Function	Syntax	Operation
lb	100000	$t, i ($s)	$t = SE (MEM [$s + i]:1)
lbu	100100	$t, i ($s)	$t = ZE (MEM [$s + i]:1)
lh	100001	$t, i ($s)	$t = SE (MEM [$s + i]:2)
lhu	100101	$t, i ($s)	$t = ZE (MEM [$s + i]:2)
lw	100011	$t, i ($s)	$t = MEM [$s + i]:4
sb	101000	$t, i ($s)	MEM [$s + i]:1 = LB ($t)
sh	101001	$t, i ($s)	MEM [$s + i]:2 = LH ($t)
sw	101011	$t, i ($s)	MEM [$s + i]:4 = $t

"b", "h" and "w" correspond to "byte", "half word" and "word", indicating the length of the data.
"SE" stands for "sign extend", "ZE" stands for "zero extend".

Ensure that instructions are properly aligned.

- Word accesses (`lw` or `sw`) should be **word-aligned** (divisible by 4).
- **Half-word** accesses should only involve half-word aligned addresses (i.e., even addresses).
- No constraints for byte accesses.

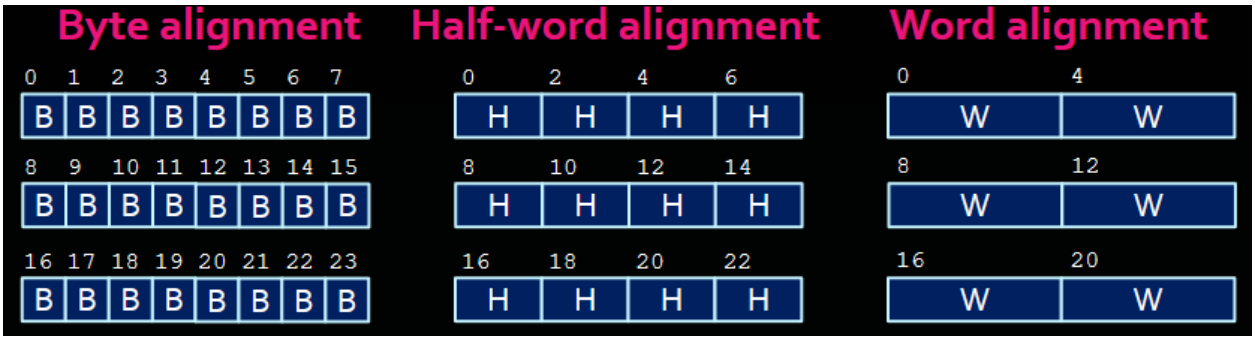


Figure 49: Pasted image 20250413192458.png

Aside About Memory

Big Endian - The most significant byte of the word is stored first (i.e., at address X). Second most is $X+1$, etc. **Little Endian -** The least significant byte is stored first at X, the least second at $X+1$, etc.
MIPS processors are **bi-endian** and operate with either byte order. MARS simulator uses the same endianness as the machine it runs on (e.g., x86 CPUs use little-endian).
Memory-Mapped IO: Used to read from devices, invoked with a **trap** or function.
- Trap instructions (‘011010‘) send system calls to the operating system. It is similar (but not the same) as ‘syscall‘.
![Pasted image 20250413193035.png](C:/Users/madeline/Documents/GitHub/Obsidian/Courses/4th Year/Winter/CSC258/Images/Pasted image 20250413193035.png)
Memory Segments and Syntax: How programs are divided in memory.
- ‘.data‘ - Start of the data values section.
- ‘.text‘ - Indicates the start of the program instruction section.
- ‘main:‘ - The initial line to run when executing the program.
- Other labels and functions after ‘main‘ (based on function names in the program).

Pseudo Instructions **Pseudo-instructions** exist for the convenience of the programmer; the assembler will translate them into 1 ore more real MIPS assembly instructions.

Examples: `la` (load address): `lad $d label`. Loads a register `$d` with the memory address that `label` corresponds to. Translated into the two following instructions:

- `lui $at, immediate1` (load upper immediate). immediate1 is the upper 16 bits of the memory address that label corresponds to.
- `ori $d, $at, immediate2`. immediate2 is the lower 16 bits of the memory address that label corresponds to.

`bge $s, $t, label` - Branch to label IFF `$s >= $t`. Translated using either `beq` or `bne`:

- `slt $at, $s, $t` (set `$at` to 1 if `$s < $t`).
- `beq $at, $zero, label` (branch if `$at==0`).

Arrays and Structs **Arrays -** Stored in continuous locations in memory. The address of the array is the address of the array's first element.

- To access element i, use i to calculate an offset distance. Add that offset to the address of the first element to get the address of the i'th element (i.e., `offset = i * size of single element`).

- To operate on arrays, load the array values into registers; operate on them; then store them back into memory.

```
## Example int A[100], B[100];

## for (i=0; i<100; i++) A[i] = B[i] + 1;
.data
    A: .space 400
    B: .word  21:100
.text
main:
    la $t8, A               # $t8 holds A
    la $t9, B               # $t9 holds B
    add $t0, $zero, $zero   # $t0 holds 4*i
    addi $t1, $zero, 400    # $t1 holds 100*sizeof(int)
LOOP:
    bge $t0, $t1, END   # branch if $t0 >= 400
    add $t3, $t8, $t0   # $t3 holds addr(A[i])
    add $t4, $t9, $t0   # $t4 holds addr(B[i])
    lw $t5, 0($t4)      # $t5 = B[i]
    addi $t5, $t5, 1    # $t5 = B[i] + 1
    sw $t5, 0($t3)      # A[i] = $t5
    addi $t0, $t0, 4    # update offset in $t0
    j LOOP
END:
```

Structs:

```
## Example: Store 5, 13, -7 at a1.
.data
a1: .space 12
.text
main:
    addi $t0, $zero, a1

    addi $t1, $zero, 5
    sw $t1, 0($t0)

    addi $t1, $zero, 13
    sw $t1, 4($t0)

    addi $t1, $zero, -7
    sw $t1, 8($t0)
```

Functions in Assembly

Functions create an interface to pieces of code by defining an entry and exit point (input and output parameters).

1. Define the start of a function with a label.
2. Take in function arguments and return values.
3. Store variables local to the function; also ensure functions do not clobber useful data on registers.
4. Return to the calling site.

Define a function using `jal FUNCTION_LABEL`.

- `jal` is a J-type instruction.
- Updates register `$31` (`$ra`), the **return address register**, and the program counter.
- Once executed, `$ra` contains the address of the instruction *after* the line that called `jal`.

Return from a function using `jr $ra`.

- The PC is set into the address in `$ra` (its value was set by the most recent `jal` function call).

To ensure that we do not overwrite `$ra` and lose where we came from, use the **stack** and **stack pointer**.

The Stack **Stack -** Spot in memory used to store values independent of the registers. **Stack Pointer -** What points to the *last* element pushed onto the top of the stack (`$sp`).

We can **push** data onto the stack (which makes it grow) and **pop** data from the stack (which makes it shrink).

- The stack uses LIFO (last-in first-out order).
- Different `$ra` values will exist in layers on the stack over time (useful for nested function calls)! Use the stack to also store function arguments and return values.

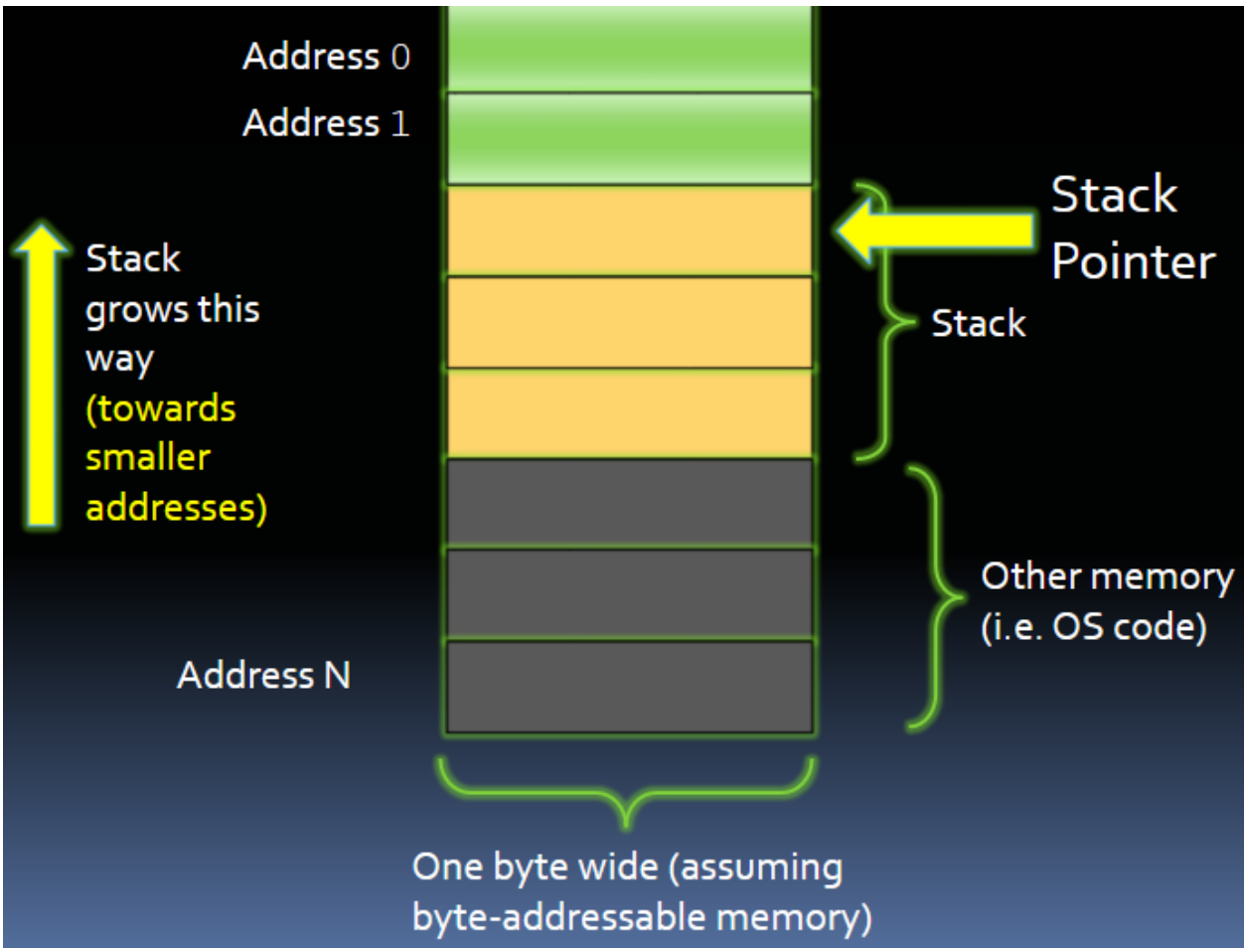


Figure 50: Pasted image 20250413200927.png

Pushing onto the stack:

- Allocate space by **decrementing** the stack pointer by the number of bytes.
- Do a store.

Popping from the stack:

- Do a load.
- De-allocate space by incrementing the stack pointer by the number of bytes.

Function Parameters There are common calling conventions to pass values to/from programs:

- Registers 2-3 (`$v0, $v1`) = Return values
- Registers 4-7 (`$a0-$a3`) = Function arguments For any additional arguments, they would be pushed onto the stack (or just all pushed onto the stack and not use `$a0-$a3`).

We also have calling conventions for the registers used.

- Caller vs. Callee calling conventions:
  - Caller-saved (user) registers - `$t0 - $t9`.
    - Registers that should be saved to the stack, since they may be erased.
  - Callee-saved (program) registers - `$s0 - $s7`.
    - It is the responsibility of the callee to save these registers and restore them.

Recursion in Assembly

Before the recursive call in the program, store the register values that you use onto the stack, and restore them when you come back to that point.

Example: Factorial of x

```
main:
    addi $t0, $zero, 10     # call fact(10)
    addi $sp, $sp,   -4     # stack
    sw   $t0, 0($sp)        # store
    jal  factorial
    ...

factorial:
    lw  $t0, 0($p)          # get x from stack
    bne $t0, $zero, rec     # base case
base:
    addi $t1, $zero, 1      # return value
    sw   $t1, 0($sp)
    jr   $ra                # return to caller
rec:
    addi $t1, $t0, -1       # x--
    addi $sp, $sp, -4       # put $ra val on stack
    sw   $ra  0($sp)        # put x-1 on stack
    addi $sp, $sp, -4
    sw   $t1, 0($sp)        # recursive call
```

Remember that `jal` always stores the next address located into `$ra`, and `jr` returns to that address!

Interrupts and Exceptions

Interrupts - Take place when an **external** event requires a change in execution.

- **Exceptions:** arithmetic overflow, undefined instructions, division by 0, etc.

  - All internal to the processor.
- **Interrupts**: Key pressed, printout finished, network packet received, etc.
  - Interrupts are signalled by an external input wire, usually checked at the end of each instruction.

Handling Interrupts/Exceptions:

- Jump to the **ISR** (Interrupt service routine).
  - The ISR is a piece of code in the OS that handles the interrupt and serves the interrupting device/exception event.
- Disable the interrupts.
- Save the return address (and some other registers).

Handle interrupts using either **single handlers** (MIPS uses single handlers) or **vectored handling**.

- For single-handler interrupt handling, the handler (SW) checks the value in the **cause register** (image below), and jumps to the corresponding exception handler code.

0 (INT)	external interrupt.
4 (ADDRL)	address error exception (load or fetch)
5 (ADDRS)	address error exception (store).
6 (IBUS)	bus error on instruction fetch.
7 (DBUS)	bus error on data fetch
8 (Syscall)	Syscall exception
9 (BKPT)	Breakpoint exception
10 (RI)	Reserved Instruction exception
12 (OVF)	Arithmetic overflow exception

Figure 51: Pasted image 20250413202908.png

- If the original program can resume afterwards, the interrupt handler returns to the program via `rfe`.
- Otherwise, the stack contents are **dumped** and execution continues elsewhere (i.e., back to OS).