COMP 273: Intro to Computer Systems Review

Julian Lore

Last updated: February 13, 2017

Adapted from Joseph Vybihal's Winter 2017 COMP273 slides.
Some things taken from Allan Wang's website.

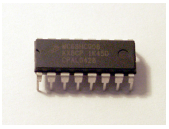
Contents

Pre-midterm Material

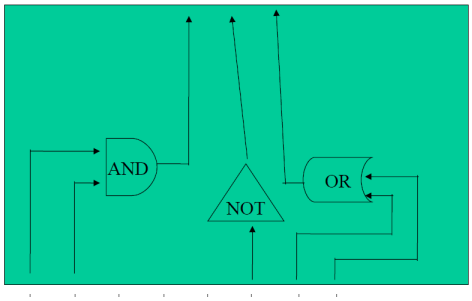
1 Basics

1.1 Bytes

A **byte** consists of 8 bits, 8 inputs and outputs. The main thing we will be working with in this course.



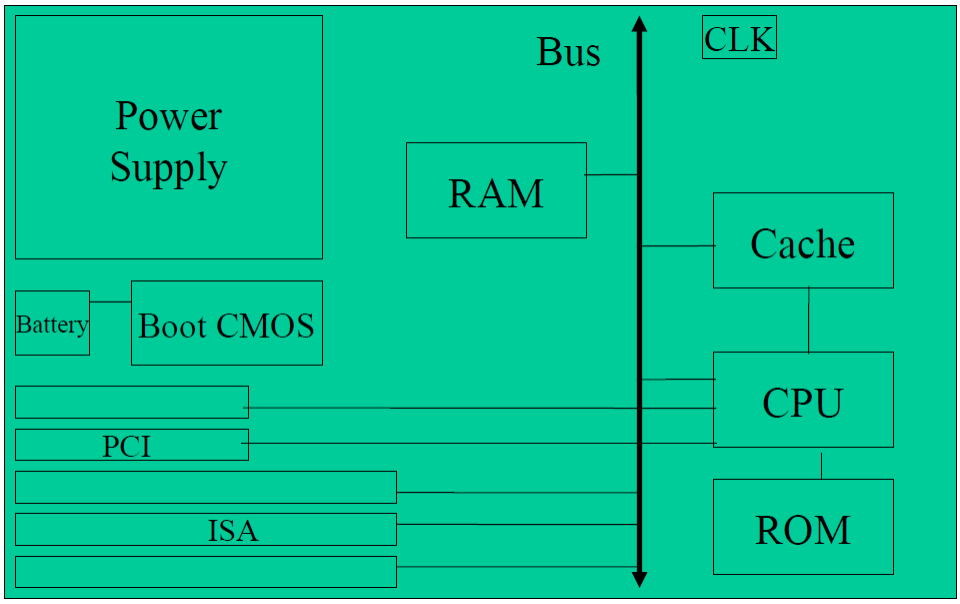
5v signifies T/1, whereas 2v signifies F/0.
Bytes consist solely of 3 gates (AND, OR, NOT), other gates can be made from these.



1.2 Electricity Flow

Electricity flows like water on a flat surface, will go in all directions. In order to control electricity flow, we will use **gates**.
If electricity flows the opposite way it's supposed to (in where it's out), the computer will freeze.

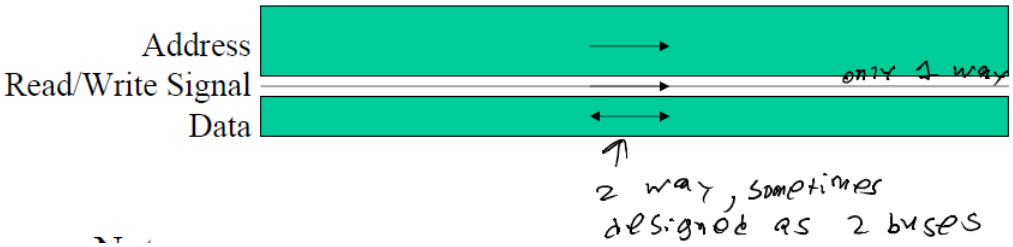
2 System Board



The system board consists of data paths, such that everything is connected.
Many definitions on Lecture 1, are they needed?

2.1 Bus

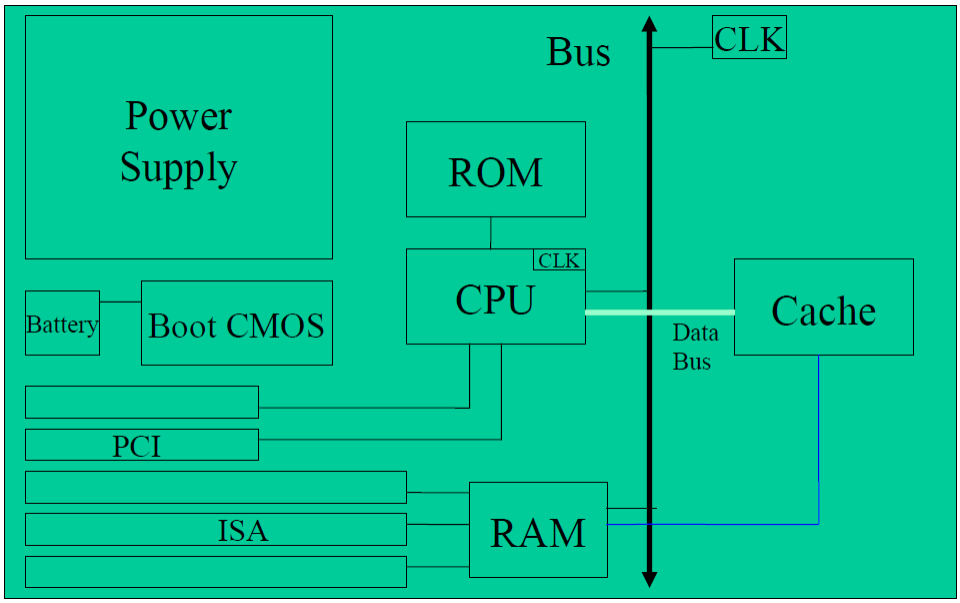
The **bus** links everything together on the system board. A conduit for bytes to travel from one location to another location (pathway). On the classical CPUs, there is only 1 bus, so only 1 thing can use the bus at once.



Here's an optimized schematic that uses the bus less. It provides several direct/private data lines:

- From the CPU to the cache (cache can operate at CPU clock speed)
- Cache to RAM
- CPU to PCI
- All of these make things faster

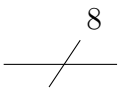
- Great use of speediness of CPU



Can multi-thread with multiple buses, but multiplies the cost of the computer

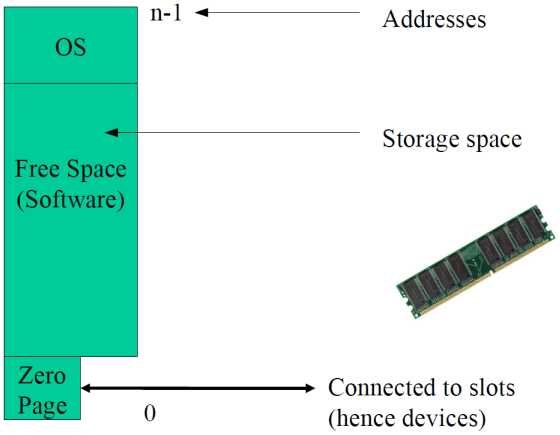
2.2 Pathways

There are many other **pathways**, such as system buses (ISA, PCI), data bus, CPU bus, wires. These pathways are to **interconnect** components. Pathways are composed of multiple parallel wires (to be executed independently), one wire per bit. One byte goes through a bus per tick.

Shorthand for multiple wires 8 wires 

2.3 RAM

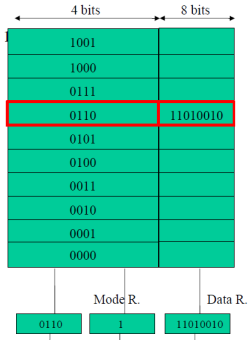
Volatile (live data) general purpose main memory bank, large & slow, **RAM** consists of addresses, storage space and the zero page, which connects to slots. Comparable to an array.



In comparison to cache, RAM costs pennies and cache costs dollars. Cache is much faster, but RAM is used as an illusion to make the computer work as if all the memory was cache.

RAM separated into 2 sections, addresses and data. RAM is actually mostly addresses. **Data** is typically 8 bits, whereas the size of addresses varies. Typically 32 or 64 bit right now, soon to be 128. Once RAM is built with some size, you can't change it (so you must decide wisely). Need 1 wire per bit (i.e. 8 wires going to data part and however many for addresses). In order to r/w to RAM, we need a few things. **Address register** to select address/row, mode register for r/w and data register. For large values/things, like strings, store in consecutive memory areas.

Video and RAM 1:1 correspondence with RAM and pixels. Pixels represented by integer numbers.



2.4 Clock

There are always at least 2 **clocks**. One for the bus (much slower) and one for the CPU (much faster). The bus clock is 1 or 2 orders of magnitude slower than the CPU clock.

Bus Clock

- Mainly for gating access to bus.

- Also regulates data movement on system board.

CPU Clock

- Responsible for instruction execution inside CPU.
- Moves data using CPU bus or moves code from IP $\rightarrow$ sequencer
- Doesn't affect things outside CPU.
- The clock will determine order of instruction execution (they have to take turns).

2.5 PCI & ISA

Connect outside of the computer, to other peripherals like monitors. PCI can run at higher clock speeds than ISA.

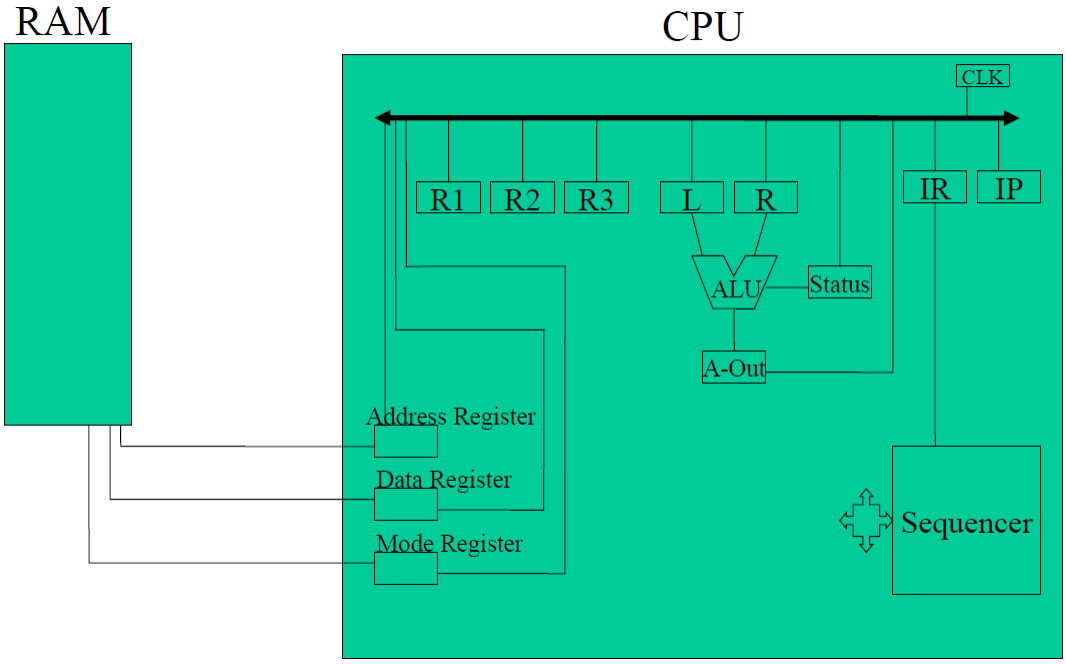
2.6 Addressing

All components of system board have a unique integer number identifying them. Needed, so that we can block electricity from going to certain peripherals.

2.7 CPU

Central Processing Unit, used for math, logic, data, movement & loops. The **CPU** is pretty simple/basic, even though it's the "brain" of the computer. It's main capabilities consist of adding, subtracting, multiplying, knows about the 3 gates.

Classical CPU Scheme Without Cache



Here, RAM reads the address from the address register, writes to the data register and the mode register differentiates read/write. The sequencer opens and closes gates.

Registers Like temp variables for the CPU. Store single value.

ALU: Arithmetic Logic Unit The ALU takes 2 inputs, L and R to get it's output, A-Out. The Status register takes input (for type of operation) and output to report errors, like overflow, dividing by zero.
ALU consists of a bunch of adders, 2s complement circuit for L and for R (need something that inverts bits and then use an adder to increment by 1).

IP(Instruction Pointer) or IC (Instruction Counter)

- Next instruction/address to execute
- Address Register points to it when needed

IR (Instruction Register)

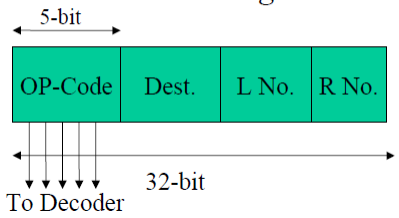
- Current instruction being executed
- From data register
- First part is op-code, goes to control unit

- Parts after are arguments
- Usually 3 args, 3^{rd} is usually optional?

Example Instructions

- lw $2, ($3) (indirect)
  - Loads word into reg 2 from address in reg 3
- lw $2, #1A2F
  - Goes directly to address, no need to check register (direct)

The Instruction Register



- L → L register
- R → R register
- A-Out → Dest
- OP-Code is used by sequencer to trigger circuit that will do this action.

We can't distinguish variables, addresses and operations.

RAM Access Register System

- Communicates CPU ↔ RAM
- Address register (r/w this address)
- Data register (data received or sending)
- Mode register (r/w)

CPU Boundary Register Keeps track of addresses used, addresses requested shouldn't pass boundary address.

Sequencer/Decoder

- Table, codes with circuits
- Circuits have gated triggers, allowing data to go in a specific order
- Sequencer and ROM have circuitry to react to machine lang, allow computer to execute instr. Sequencer: built-in instr. ROM: extended instr.

CPU Clock See 2.4.

CPU Loop (Execution)

1. Get instruction: IR ← RAM (or cache), slow bus
2. Sequencer ← IR[op-code]
3. Selected gates open
4. Clock ticks (if this is outside CPU, have to respect other clocks too)
5. All gates close
6. Increment → next instruction (fast). Back to step 1.

Shorthand:

1. AR ← PC
2. PC ← PC+1
3. DR ← RAM[AR]
4. IR ← DR
5. Re-loop

Constant switch from slow to fast clock means we are never really fast, which is why we need things like a cache.

Memory Different types of memory. See specific sections for more info.

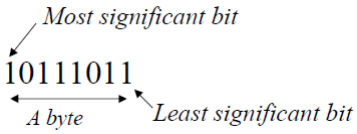
- RAM (primary storage). DRAM: Dynamic, must refresh. SRAM: Static, no refresh.
- ROM, read only memory (advanced instructions). ROM is hardwired. PROM programmable once (fuses). EPROM, erasable by heat. EEPROM electrically erasable. PAL, PLA.
- Cache. Very fast memory, usually on CPU. Store frequently accessed info. Doesn't use system bus. 1 direction, cache1 → CPU and CPU → cache2.
- Pipeline. Use assembly line to process instructions. Partially parallel. Series of instruction registers.

3 Data/Data Encoding

Data is information stored in RAM or secondary storage. Can be instructions or information. There are built-in types CPU can understand (int, real, char), since there are circuits made to interpret them. Language types must be simulated. The **bit** is the fundamental unit of the computer. Using bits and gates, we will group things in order to store data.

Bit Grouping Two forms: numerical binary representation and tabulated binary encoding (ASCII, UNICODE, instructions, etc.). Can count, add, subtract, multiply and divide with binary numbers.

Name	Bits per Symbol	Total Symbols	Comments
BCD	6	64	A-Z; 0-9, $ ... (only capitals)
ASCII	7	128	a-z; A-Z; 0-9; Bel, Tab, $, ...
USASCII	8	256	Even parity bit for transmit
EBCDIC	8	256	Odd parity bit (only IBM)
UNICODE	16	65,536	Many languages



How to represent data? How many bits do we need? What medium to use? (lights, sound, signals) Lightbulbs originally used, either on or off.

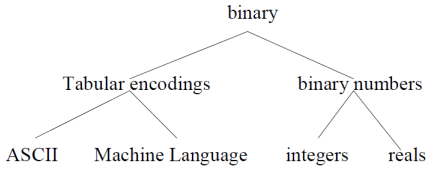
Nibble	4 bits
Byte	8 bits
Word	16 bits
Long Word/Word	32 bits
Quad Word/Word	64 bits

There are standards for data types/storage established by IEEE and ISO. 3 basic things to encode: chars, numbers and instructions. For **characters**, we use a predetermined table (ASCII, unicode), **numbers** use binary arithmetic and **instructions** use a table, gate sequence code-number.

Word Word is the common size of a register of CPU(best for design of CPU). CPUs often have multiple register sizes. Word sizes differ between CPUs. Nowadays, register size is usually the same as address size.

3.1 Data Types & Encoding

How does the machine know that a byte is a character? The ROM (Read Only Memory), has predefined instructions to deal with it. Compiler will look at ASCII table and display the respective character, etc.



You need the ROM for tabular encodings, but not for binary numbers.

3.1.1 Number Representations

Binary

- Used for flags, numbers, strings, encodings
- Longhand addition in binary is the same as elementary grade addition with decimals
  - But we have a fixed size
  - If we carry past size → **Arithmetic Overflow**
  - Signed overflow expected for 2's complement subtraction

Hexadecimal Easier to read for humans, better correspondence with binary. Base 16, with 10-15 being A, B, C, D, E, F. Decimal to Binary conversion is slow. Computers often convert binary to hex for errors to make it easier to read.

Octal

- Barely used anymore
- Except for Unix permissions and C, Perl escape codes

3.1.2 Conversion

Decimal → Binary Keep dividing by 2 until you get 0, read the remainders from bottom

$123_{10} = N_2 = 1111011_2$

123 /2 = 61 R 1
61/2 = 30 R 1
30/2 = 15 R 0
15/2 = 7 R 1
7/2 = 3 R 1
3/2 = 1 R 1
1/2 = 0 R 1

Read ↑

to top → left to right.

Decimal → Hex Divide by 16 until 0, read remainders in same way.

$53241_{10} = N_{16} = CFF9_{16}$

53241/16 = 3327 R 9
3327/16 = 207 R F
207/16 = 12 R F
12/16 = 0 R C

Binary → Decimal Depending on position of number, multiply value by $2^{n^{th}\text{position}}$

$$1011_2 = N_{10} = 11_{10} = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Hex → Decimal Same principle with base 16.

$$1AB_{16} = N_{10} = 427_{10} = 1 \times 16^2 + A \times 16^1 + B \times 16^0$$

Binary ↔ Hex 1 nibble = 1 hex digit

$$\underbrace{1111}_{F}\underbrace{0011}_{3}\underbrace{0001}_{1}\underbrace{0000}_{0} = F310_{16}$$

Binary ↔ Octal 3 bits = 1 octal, same as Hex strategy.

3.1.3 Binary Encodings

ASCII 2^7 values, but wasn't enough for special characters like accents.

0	0011 0000	O	0100 1111	m	0110 1101
1	0011 0001	P	0101 0000	n	0110 1110
2	0011 0010	Q	0101 0001	o	0110 1111
3	0011 0011	R	0101 0010	p	0111 0000
4	0011 0100	S	0101 0011	q	0111 0001
5	0011 0101	T	0101 0100	r	0111 0010
6	0011 0110	U	0101 0101	s	0111 0011
7	0011 0111	V	0101 0110	t	0111 0100
8	0011 1000	W	0101 0111	u	0111 0101
9	0011 1001	X	0101 1000	v	0111 0110
A	0100 0001	Y	0101 1001	w	0111 0111
B	0100 0010	Z	0101 1010	x	0111 1000
C	0100 0011	a	0110 0001	y	0111 1001
D	0100 0100	b	0110 0010	z	0111 1010
E	0100 0101	c	0110 0011	.	0010 1110
F	0100 0110	d	0110 0100	,	0010 0111
G	0100 0111	e	0110 0101	:	0011 1010
H	0100 1000	f	0110 0110	;	0011 1011
I	0100 1001	g	0110 0111	?	0011 1111
J	0100 1010	h	0110 1000	!	0010 0001
K	0100 1011	I	0110 1001	'	0010 1100
L	0100 1100	j	0110 1010	"	0010 0010
M	0100 1101	k	0110 1011	(	0010 1000
N	0100 1110	l	0110 1100	)	0010 1001
				space	0010 0000

3.1.4 Data Representation

Representing data consists of a choice, that comes at a cost. Pros/cons to each choice. Specify the base when writing numbers in this class. 3 representations:

- Logical description (how it truly looks & behaves)
- Physical construction
- Circuit

We need legal operators, abstraction of what is being recorded (chars don't exist) and want to know how we want to represent information in binary (size, addressing).

Integers Usually 16,32 or 64 bits. How do we differentiate sign? We can have a **signed** bit (use most significant bit as sign) or use "2's complement". With signed bits, there exists a negative 0, but with 2's comp, only one 0 and most significant bit is still a sign bit. We don't need a null at the end of an int, it's matched with x bit instructions made for its size.

Two's Complement Easier to build computers if they subtract using:

$$X - Y = X + (-Y) = X + (\text{2's complement of Y})$$

How to convert?

- Take Y
- Flip bits
- Add 1

Uses a signed overflow in order to get expected result.

Example

```
 00111
- 00101     Take value:  00101
----------  Flip bits:   11010
    ?       Add 1:       11011
```

```
   00111
  +11011
----------
00010₂
```

Overflow (Expected) (Ignored)

Base 10 Two's Complement

$$-N = Base^{size} - N$$

$$-12_{10} = 10^2 - 12 = 88_{10}$$

$$15 - 12 = 15 + 88 = 103 \implies \text{Drop carry, get 3}$$

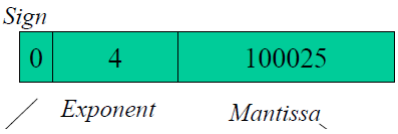
Signed vs 2's Comp

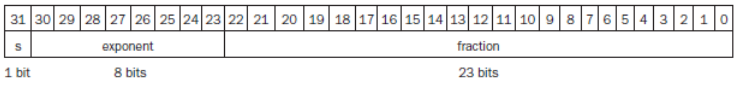
- Signed
  - Easy to read
  - No conversion
- 2's
  - Unique 0
  - Auto subtracts when adding

Characters Encoded using ASCII or UNICODE or something similar (standard). Type not stored in computer, there are no types.

Strings Each char stored in 1 byte in RAM (consecutive). Either a null at the end signifying the end of the String, or the most significant bit describes length.

Floating Point/Real Numbers Scientific notation approximation, store decimal and exponent.





In general, floating-point numbers are of the form

$$(-1)^S \times F \times 2^E$$

IEEE format Uses bias for exponent, i.e. for 32 bit, 127=0, 126=-1, 128=1. This way we don't need a signed bit.

		Single	Double	Quad
Number of bits taken by:				
	Sign	1	1	1
	Exponent	8	11	15
	Fractional mantissa	23	52	111
	Total	32	64	128
Exponent:				
	Bias	127	1023	16383
	Range of (biased) exponent:	0..255	0..2047	0..32767

Mantissa stores a fraction(raw digits after decimal) for approximation. Normalize your fraction such that you get 1.something and ignore the 1.

Example

$$\begin{aligned} &-0.75_{10} = -3/4_{10} = -3/2^2_{10} \\ \implies &-11_2/2^2_{10} = -0.11_2 = -0.11_2 \times 2^0 = -1.1_2 \times 2^{-1} \\ \implies &(-1)^1 \times (\underbrace{1}_{ignored} + .100\ldots00_2) \times 2^{126-217} \end{aligned}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1 bit				8 bits					23 bits																						

Notice that the leading 1 before the decimal is ignored. The 22^{nd} bit is 2^{-1}, the 23^{rd} is 2^{-2}, etc. Since we had 1.1_2, we look at the numbers after the decimal and just copy those.

Logical Single bit, 0=false, 1=true

Packed Decimal One nibble per digit of a number in base 10. Can be used to store decimal numbers. Addition is strange. Add 2 nibbles together, if you get over 10, subtract 10 and carry and add 1 over to next nibble. (i.e, if you do 3+9, you'll get 12 (5 bits), subtract 10 and represent 2 in binary)

$$5372_{10} = 0101\ 0011\ 0111\ 0010$$

Very user friendly, but wastes space and complicates hardware.

3.2 Mathematical Operation Algorithms for Floating Points

Addition

- Normalize (floating point)
- Round if required
- Put numbers to same exponent (shift smaller to right until it's the same as larger)
- Add significands
- Normalize (check for over/underflow)
- Round
- Sign?

Ex Add 0.5_{10} and -0.4375_{10}

$$\begin{aligned} 0.5_{ten} &= 1/2_{ten} &&= 1/2^1{}_{ten} \\ &= 0.1_{two} &&= 0.1_{two} \times 2^0 &&= 1.000_{two} \times 2^{-1} \\ -0.4375_{ten} &= -7/16_{ten} &&= -7/2^4{}_{ten} \\ &= -0.0111_{two} &&= -0.0111_{two} \times 2^0 = -1.110_{two} \times 2^{-2} \end{aligned}$$

Now we follow the algorithm:

Step 1. The significand of the number with the lesser exponent ($-1.11_{two} \times 2^{-2}$) is shifted right until its exponent matches the larger number:

$$-1.110_{two} \times 2^{-2} = -0.111_{two} \times 2^{-1}$$

Step 2. Add the significands:

$$1.000_{two} \times 2^{-1} + (-0.111_{two} \times 2^{-1}) = 0.001_{two} \times 2^{-1}$$

Step 3. Normalize the sum, checking for overflow or underflow:

$$\begin{aligned} 0.001_{two} \times 2^{-1} &= 0.010_{two} \times 2^{-2} = 0.100_{two} \times 2^{-3} \\ &= 1.000_{two} \times 2^{-4} \end{aligned}$$

Since $127 \geq +4 \geq -126$, there is no overflow or underflow. (The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.)

Step 4. Round the sum:

$$1.000_{two} \times 2^{-4}$$

The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.

This sum is then

$$\begin{aligned} 1.000_{two} \times 2^{-4} &= 0.0001000_{two} &&= 0.0001_{two} \\ &= 1/2^4{}_{ten} &&= 1/16_{ten} &&= 0.0625_{ten} \end{aligned}$$

This sum is what we would expect from adding 0.5_{ten} to -0.4375_{ten}.

Multiplication

- Remove exponent bias
- Match exp
- Multiply significands
- Normalize & round
- Sign

Ex Multiply 0.5_{10} and -0.4375_{10}

In binary, the task is multiplying $1.000_{two} \times 2^{-1}$ by $-1.110_{two} \times 2^{-2}$.

Step 1. Adding the exponents without bias:

$$-1 + (-2) = -3$$

or, using the biased representation:

$$(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127)$$
$$= -3 + 127 = 124$$

Step 2. Multiplying the significands:

$$\begin{array}{r} 1.000_{two} \\ \times \quad 1.110_{two} \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1110000_{two} \end{array}$$

The product is $1.110000_{two} \times 2^{-3}$, but we need to keep it to 4 bits, so it is $1.110_{two} \times 2^{-3}$.

Step 3. Now we check the product to make sure it is normalized, and then check the exponent for overflow or underflow. The product is already normalized and, since $127 \geq -3 \geq -126$, there is no overflow or underflow. (Using the biased representation, $254 \geq 124 \geq 1$, so the exponent fits.)

Step 4. Rounding the product makes no change:

$$1.110_{two} \times 2^{-3}$$

Step 5. Since the signs of the original operands differ, make the sign of the product negative. Hence, the product is

$$-1.110_{\text{two}} \times 2^{-3}$$

Converting to decimal to check our results:

$$\begin{aligned} -1.110_{\text{two}} \times 2^{-3} &= -0.001110_{\text{two}} = -0.00111_{\text{two}} \\ &= -7/2^{5}{}_{\text{ten}} = -7/32_{\text{ten}} = -0.21875_{\text{ten}} \end{aligned}$$

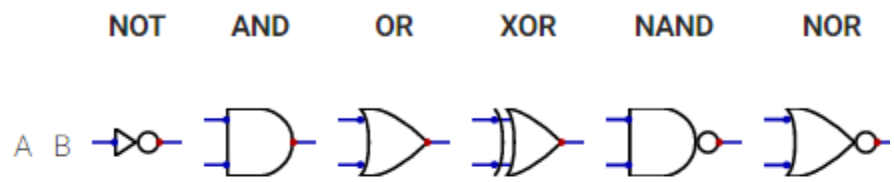
The product of 0.5_{ten} and -0.4375_{ten} is indeed -0.21875_{ten}.

4 Logic & Gates

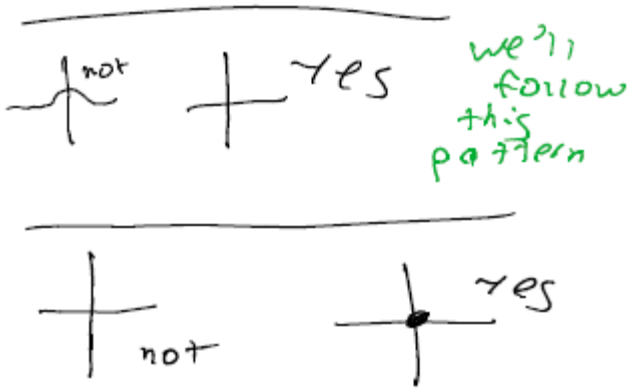
- Circle $\implies$ not
- Extra line $\implies$ exclusive

Figure 1: Taken from Allan Wang's website

Gates

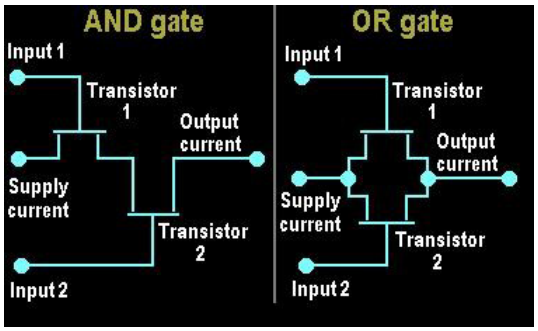


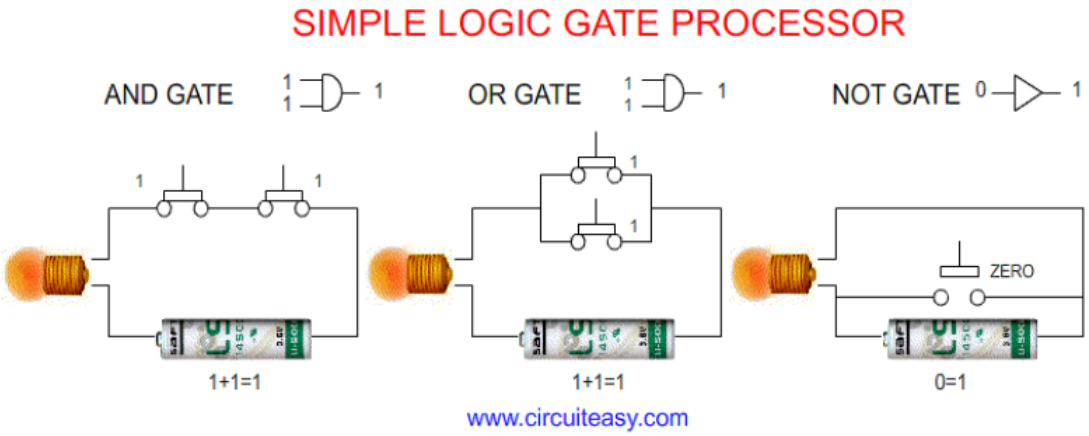
A	B	NOT	AND	OR	XOR	NAND	NOR
0	0	1	0	0	0	1	1
0	1	X	0	1	1	1	0
1	0	0	0	1	1	1	0
1	1	X	1	1	0	0	0



Wire crosses, i.e. when are two wires connected:

Low-level Construction

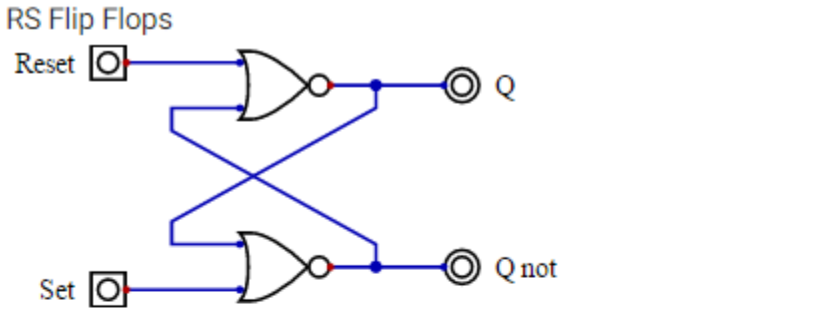




4.1 Basic Flip Flop Circuits

Flip Flops are bits!

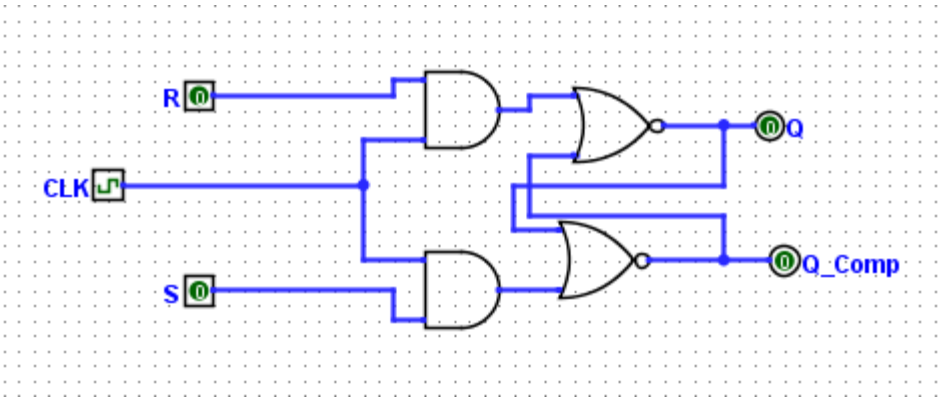
Figure 2: RS Flip Flops, Allan Wang



Basic reset set flip flop to save bit data; a clock can be connected to the inputs of both nor gates for synchronization

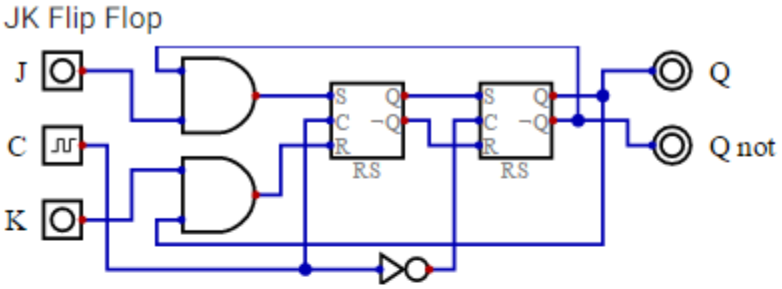
R	S	Q	Q'	Result
0	0	Q	Q'	No Change
0	1	1	0	Set
1	0	0	1	Reset
1	1	1	1	Avoid

Sending 1 to reset(R) and set(S) is an invalid input. They must be opposites, or both 0. Sending a 1 through set, sets the bit's output, Q to 1, and it's complement Q' to 0. Sending 1 through reset does the opposite. If both are 0, keep current value (clocked).
Can test **steady state** of the circuit. Give a 1 to R and follow the circuit, realize nothing conflicts. The flip flops don't actually store anything, they just have data going through them → live data, lost when unplugged.



Clocked RS: Like an RS Flip-Flop, except only let's things through when clock ticks.

Figure 3: JK Flip Flop, Allan Wang

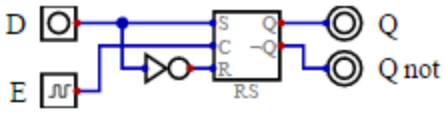


JK flip flops cycle at half the speed of its input, as only one SR flip flop is enabled at a time and it takes two clicks to pass data to the output.

J	K	Q	Q'	Result
0	0	Q	Q'	Unchanged
0	1	0	1	Reset
1	0	1	0	Set
1	1	Q'	Q	Toggle

2 SR flip flops, adds a delay.

Figure 4: D Flip Flop/D Latch, Allan Wang

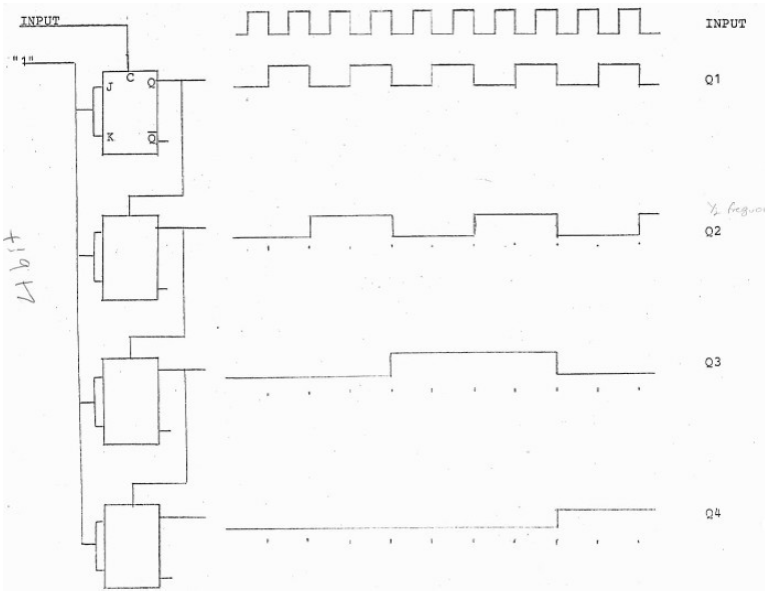


An addition to the RS flip flop to accept a single input. Together with the clock (E), the input (D) directly changes the output of the latch. This also eliminates the Reset = 1 & Set = 1 issue.

E	D	Q	Q'	Result
0	0	Q	Q'	Latch
0	1	Q	Q'	Latch
1	0	0	1	Reset
1	1	1	1	Set

Used for registers/data.

Chaining flip flops and their clock input stretches out inputs even more, how we use counters.

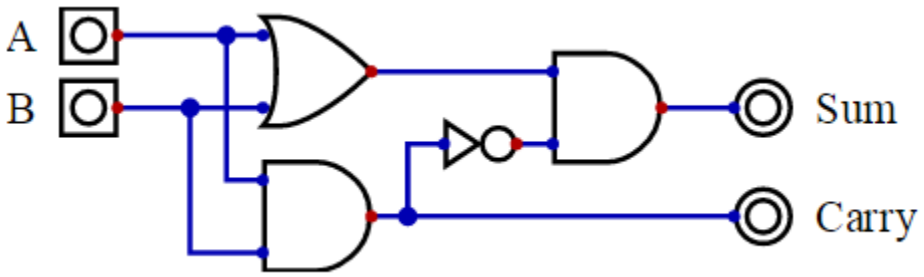


4.2 Adders

How do we do grade school addition in binary at the hardware level? When adding 2 numbers together, we need a half adder for least significant bit (no carry) and full adders for all the other bits. Be careful of overflows and signed bits!

Half-Adder 2 bits in, sum & carry out

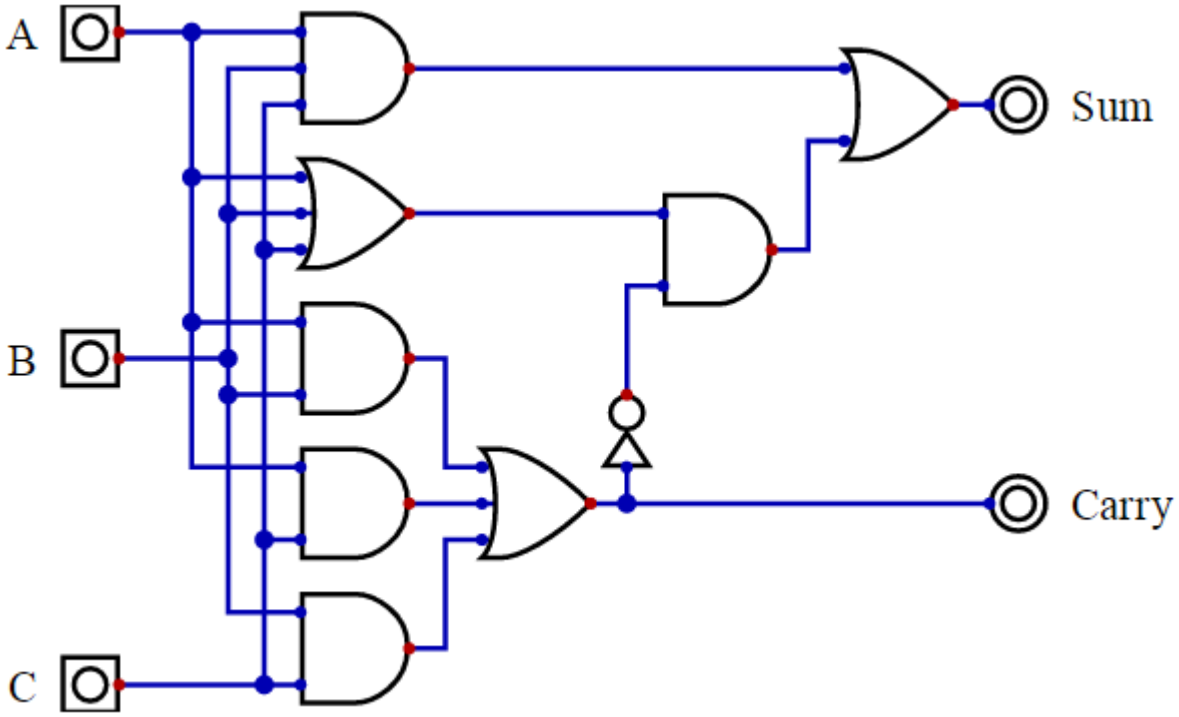
Figure 5: Half Adder, Allan Wang



If A & B are both 1, then we will give 1 to carry and negate adding to sum.

Full Adder 3 bits in (3^{rd} is carry), sum & carry out

Figure 6: Full Adder, Allan Wang

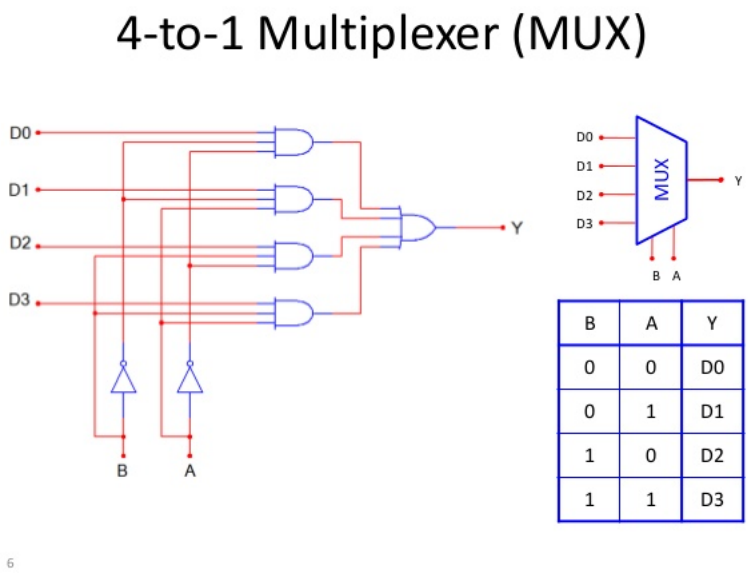


Above is the more optimized version. Can just have 2 half adders and an or. Include simpler version with half-adders?

4.3 Combinatorial Networks

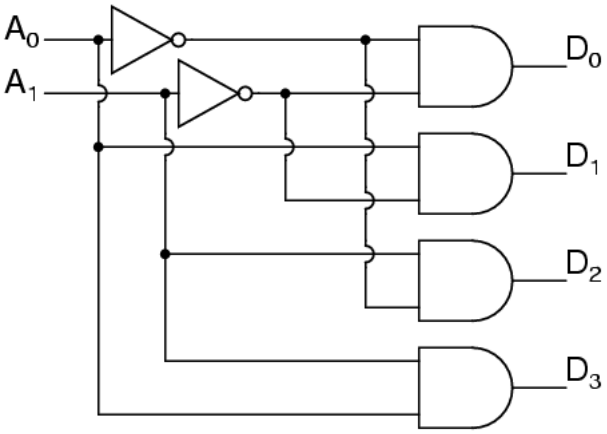
Premade circuits for specific purposes, readily available.

Multiplexers $2^a \to 1$ mux, 2^a inputs $x_0, \ldots, x_{n-1}$ and a single output z. Selector address a, with input signals $y_0, \ldots, y_{a-1}$, selects which x_i to pass through z. Useful for when you have multiple signals but can only let one through at a time. Addressing, but with 1 output, z.



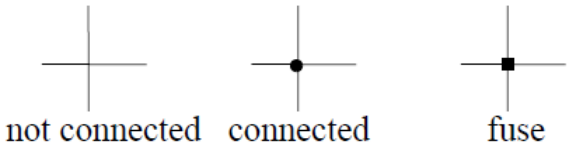
B	A	Y
0	0	D0
0	1	D1
1	0	D2
1	1	D3

Decoders $a \to 2^a$ decoder, gives out one of its 2^a outputs. Once again, like addressing, but selecting which you want to turn on. Like controlling switches. CPU sequencer is a decoder.



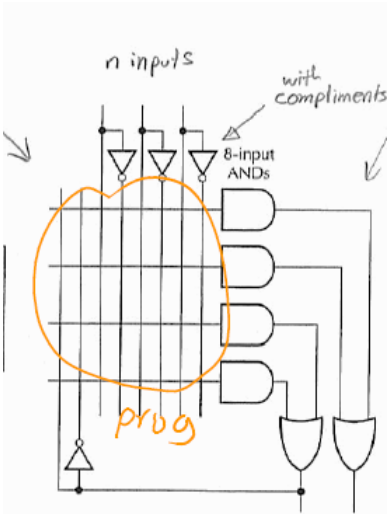
Encoders Outputs a-bit binary number, y_i equal to index of single 1 signal on among 2^a inputs $x_0, \ldots, x_{a-1}$. All of the xs are connected to an or gate, so if they're all off but x_0 is on, will show 0.

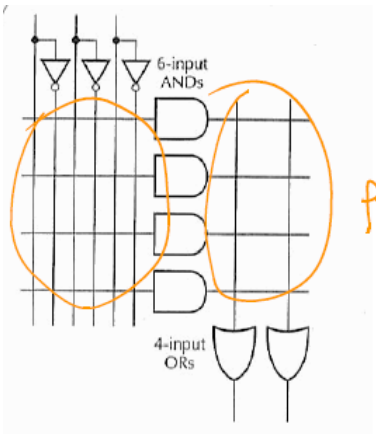
Programmable Combinatorial Parts VIA Fuses that can be blown (disconnected) from enough current or VIA anti-fuses that are set(connected) from enough current.



PROM Programmable read only memory, has anti-fuses that can be set, linked to decoder.

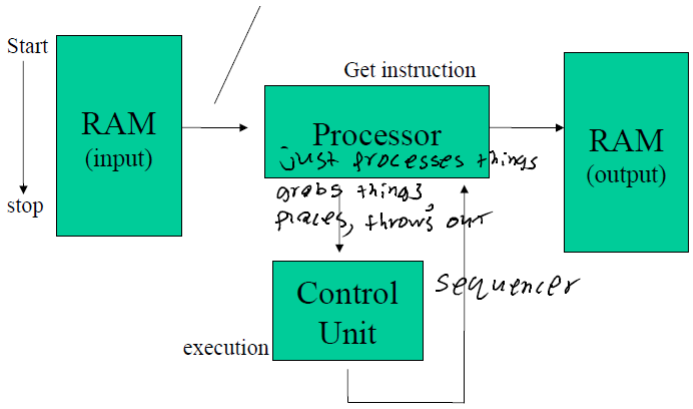
PAL Programmable Array Logic, input wires are programmable, but output isn't.



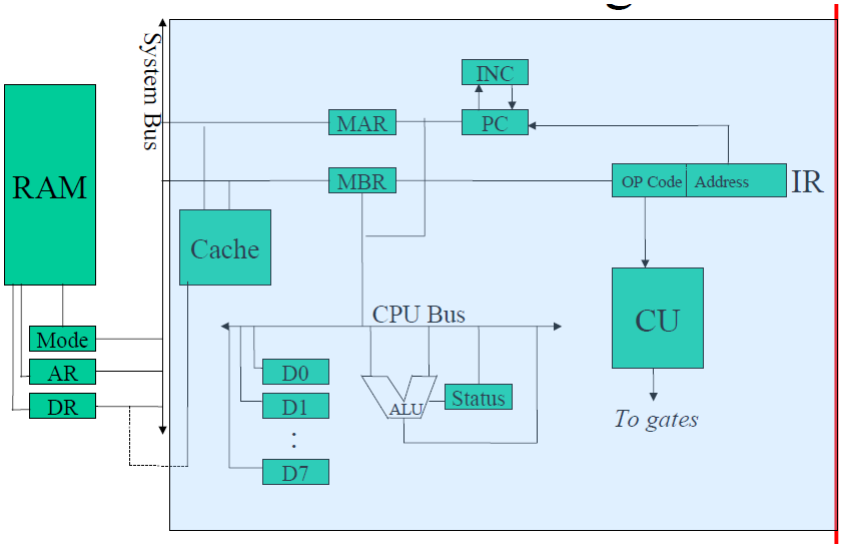


PLA Programmable Logic Array, input & output programmable.

5 In-Depth Classical CPU Analysis



Von Neumann Machine



Classical CPU Design

PC = Program Counter or Instruction Pointer. **MAR**, memory address register, downloads instructions from PC, sends to AR. AR and MAR work a lot together, both address registers. **MBR**, memory buffer register, can contain many things (reason why it's called a buffer), like address, data from data registers. AR, DR and MAR, MBR work well together. INC is incrementer (override data in address), CU is control unit. Can add onto this diagram by adding peripherals on bus.

CPU does one instruction at a time in however x ticks it takes, then increments to next address to do next operation. Making things parallel & local to avoid ticks and make it faster.

5.1 Instructions

Group of assembler instructions CPU supports. Standard size is usually the word size. Command that is formatted string of binary. Stored in RAM. Makes programs/algorithms.

Program running (Classical CPU)

- OS sleeps (no multithreading)
- 1. MAR ← PC
- PC ← PC+1
- 2. AR ← MAR
- DR ← RAM[AR]
- 3. MBR ← DR
- IR ← MBR
- CU ← IR(OP-CODE)

See instruction register for more info.

Register vs RAM Can give type to instructions, such that it goes through CPU registers or RAM. Through IR it does addition in 1 tick, but with RAM it has to go through several things, AR, MBR, IR in order to get what's at the RAM.
When program is done running, it tells the OS it can wake up now and stops program.

5.2 CPU Execution Cycle

Fetch

1. PC → MAR, PC++
2. MAR → RAM AR
3. Read signal
4. RAM DR → MBR → IR (or other register)

Decode

1. Do we need operands?
   (a) Get operands, using address in instruction to fetch each parameter, but last step goes to CPU register instead of IR and no incrementing of PC++
2. OP-CODE → CU

Execute CU triggers gates to perform instruction

Store

1. Register with address → MAR
2. Register with data → MBR
3. Write signal sent to RAM AR and DR

5.3 Micro Instructions

Express operation of CPU, step by step. Syntax to follow.
CHANGE_STATE ← OPERATION

- MACHINE(addr)

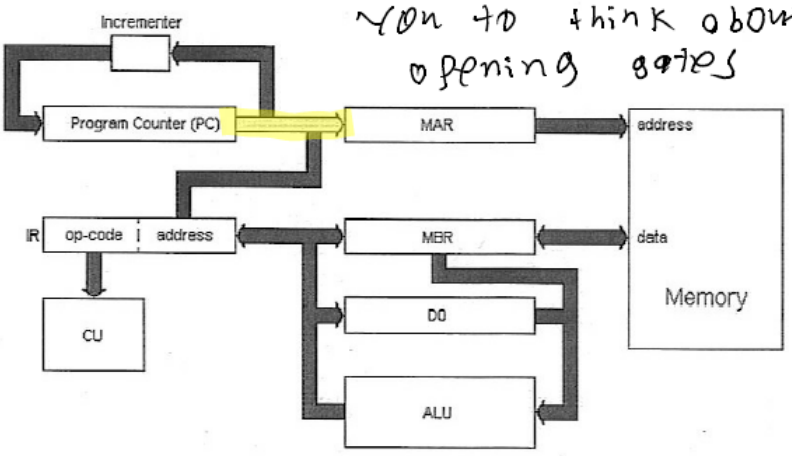
```
FETCH    [MAR] <- [PC]
         [PC] <- [PC] + 1
         [MBR] <- [M([MAR])]
         [IR] <- [MBR]

EXECUTE  [MAR] <- [IR(Address_Field)]
         [MBR] <- [M([MAR])]
         [D0] <- [D0] + [MBR]
```

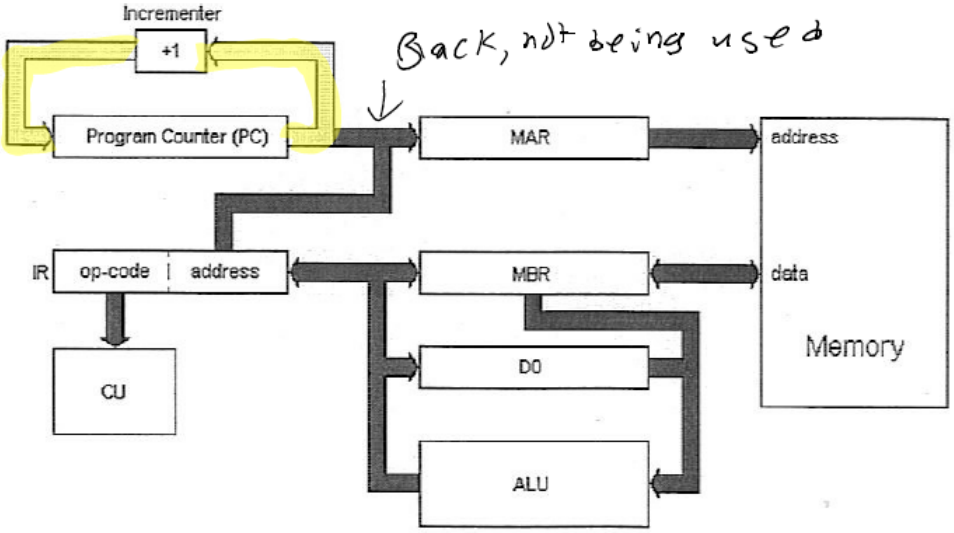
eg
ADD D0,D0,X

1) [MAR] <- [PC]

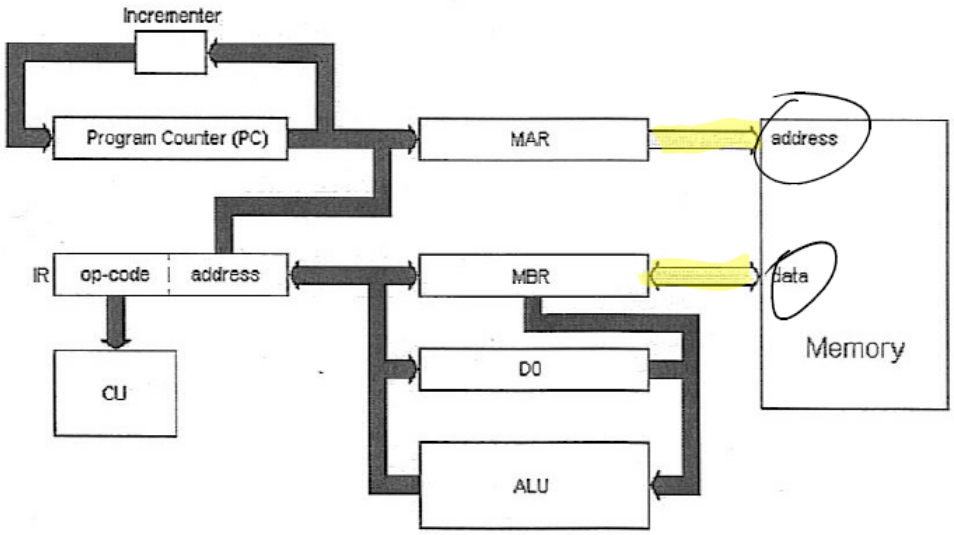
picture design allows you to think about opening gates



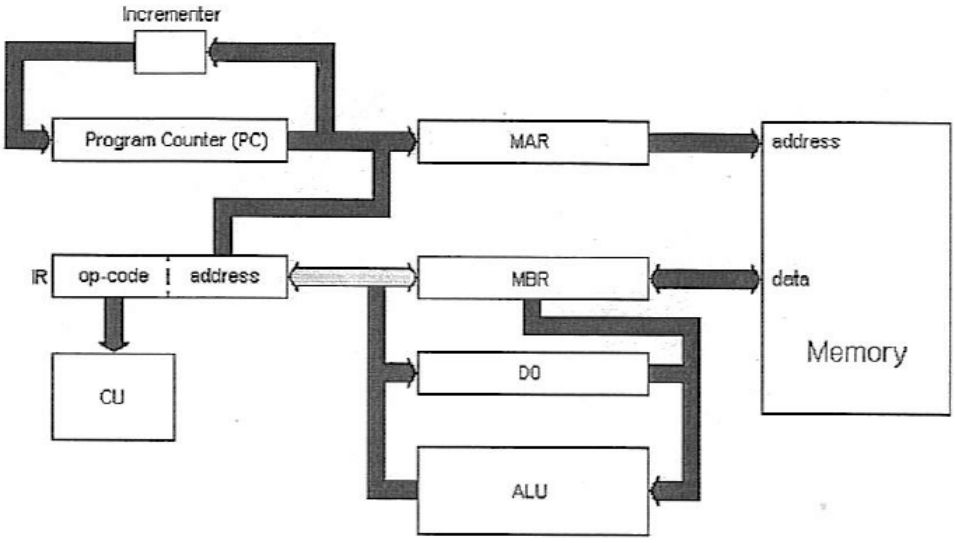
2) [PC] <- [PC] + 1



3) `[MBR] <- [M([MAR])]`

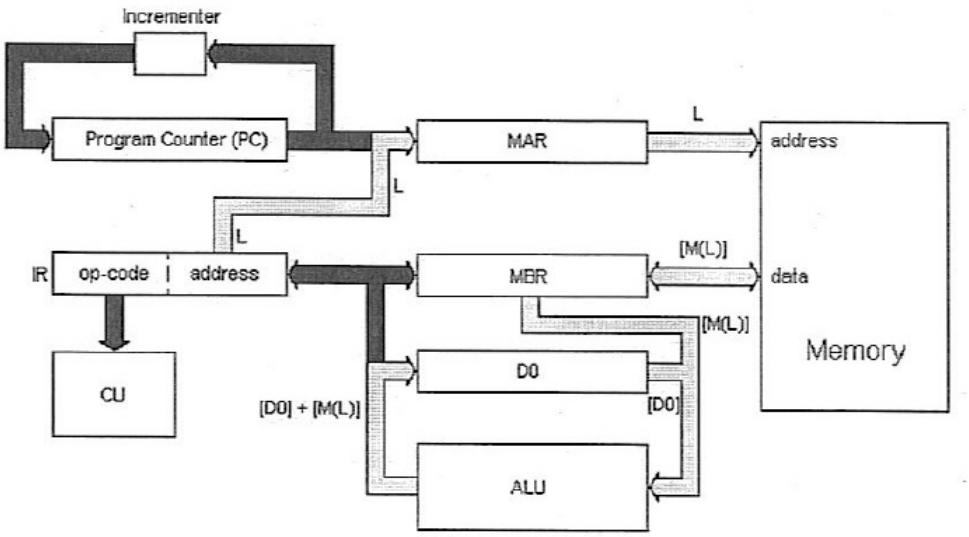


4) `[IR] <- [MBR]`



5) The execution phase:

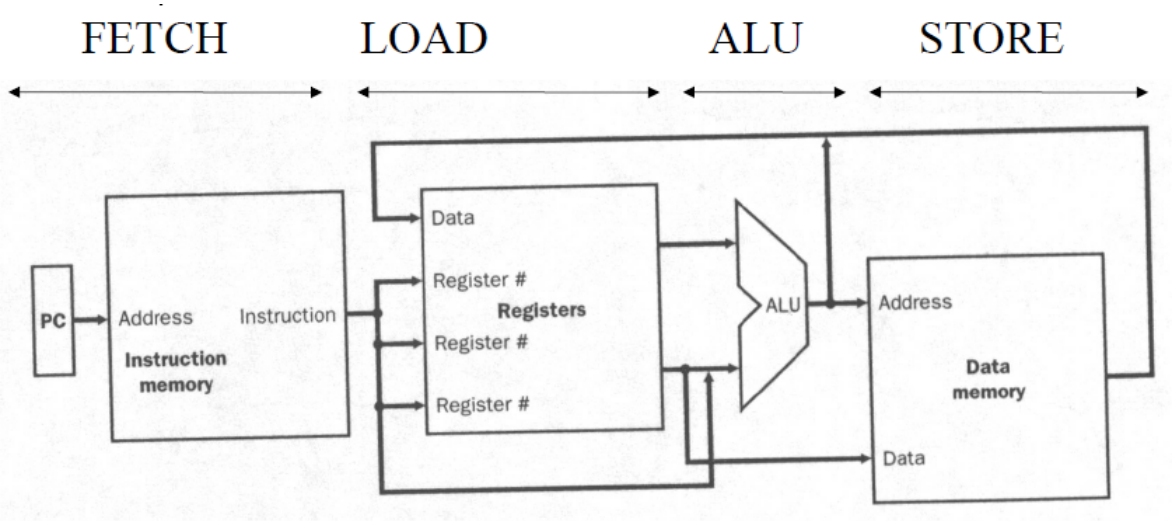
```
[MAR] <- [IR(Address_Field)]
[MBR] <- [M([MAR])]
[D0] <- [D0] + [MBR]
```



6 Pipeline CPU Structure

When we look at the classical CPU architecture, we see that it is easy to build/cheaper and the bus allows flexibility of data movement. But, there are unused circuits and everyone has to share the same clock tick.

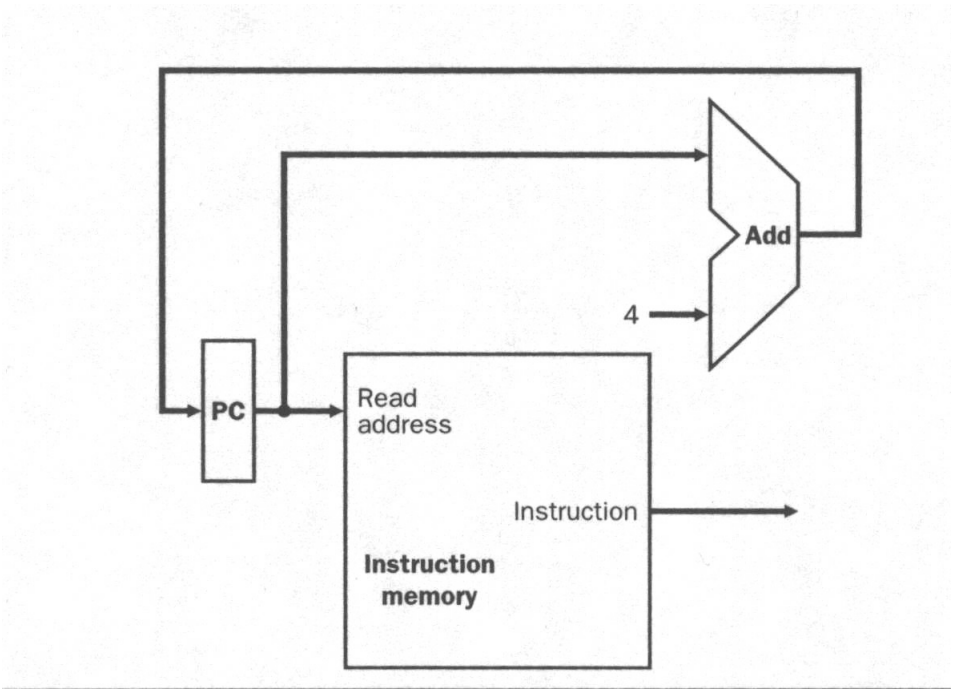
What about the pipeline CPU? The CPU **is** the bus. Only goes in one direction. Each instruction does a separate part at once, so you can have multiple instructions flowing through the pipeline at once (not executing simultaneously, more like doing different steps simultaneously). Pipeline instead of CPU loop.



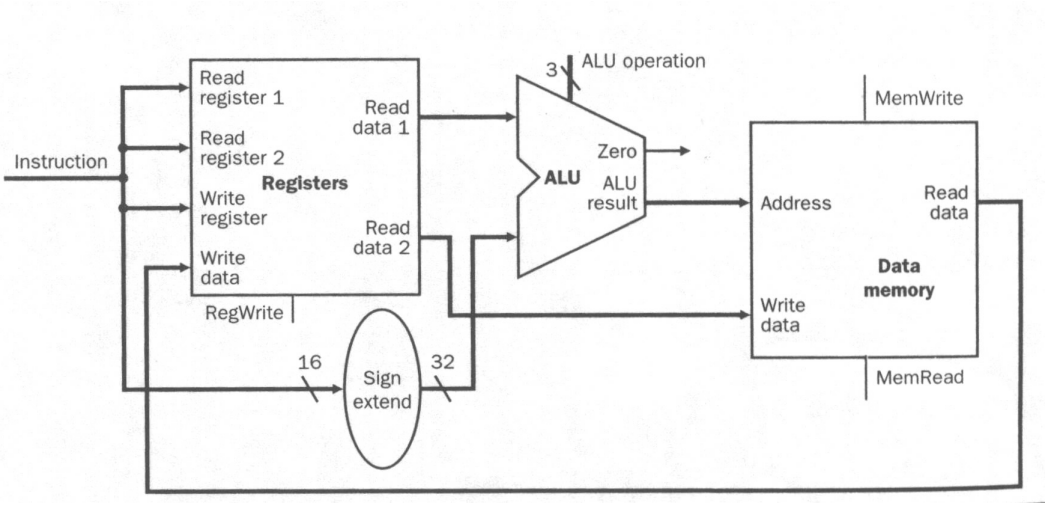
While fetching next instruction, can load current instruction. Can do 4 instructions at different phases at the "same time". Need a lot of instruction registers, one at each stage (4). Two caches. Code/Load prediction has dumb(blind dump of code) vs smart(look ahead for goto/jump).

Effects on IR and CU

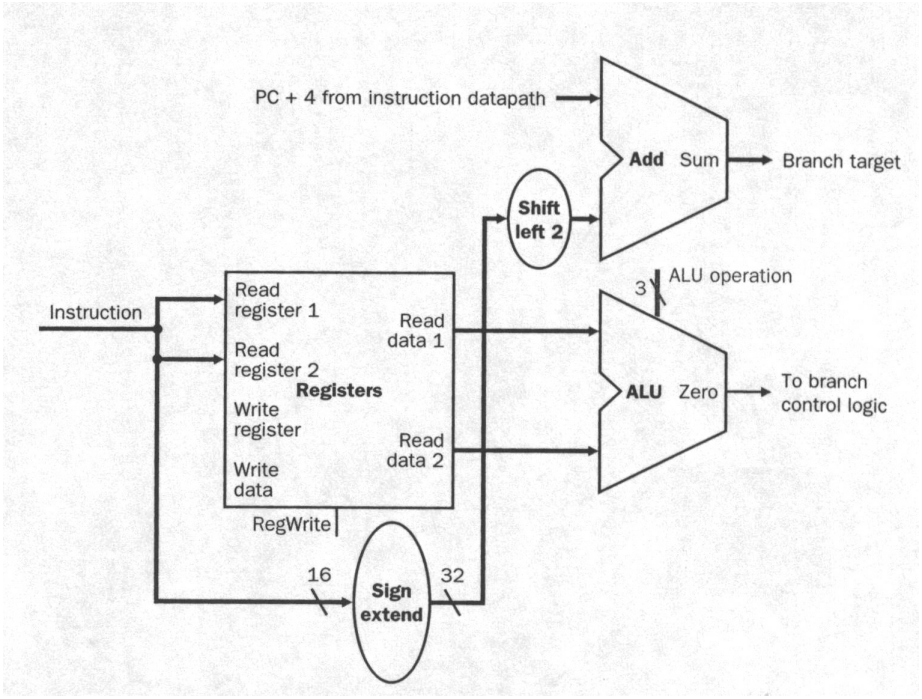
- No longer 1:1 IR and PC
- Either one long CU or many small CUs
- 1 IR for every silo (data lock/gates)
- Pipeline has to be gated



Fetch

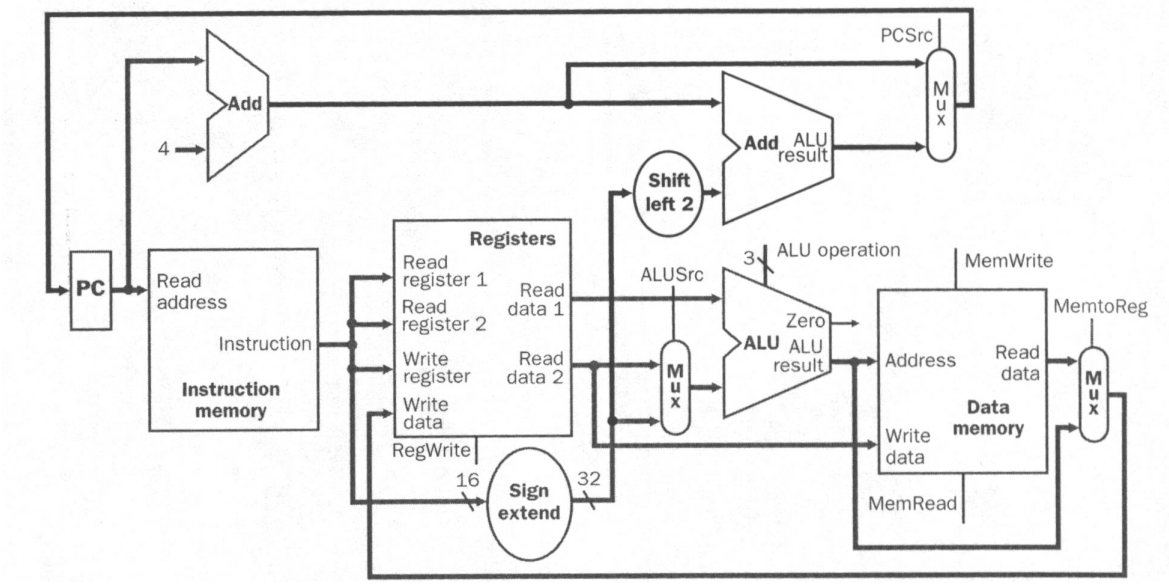


Load & ALU



Delayed Branching ALU

6.1 Summary Layouts

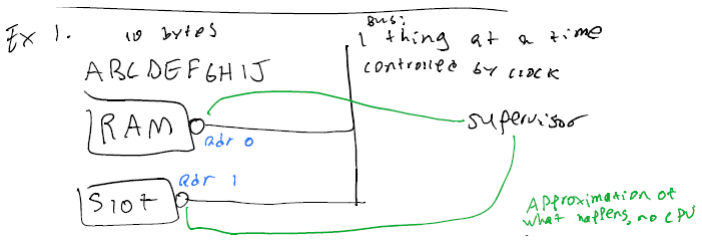


7 Assembler

- Comments indicated by #
- .text indicates code is after this

- .globl something → where is main program
- syscall calls system to temporarily wake up to do something, like print
- data indicates things stored and whatnot

8 Examples



An example of Copying from RAM to a Slot
In this example, we'd like to copy 10 letters from RAM to an arbitrary slot on a traditional system board. We must loop through as follows:

1. Open gate 0 & 1, close all other gates.
2. Wait for tick.
3. 1 character passes.
4. Go back to step 2.
5. Once all 10 characters are written, "supervisor" (later, central unit?) closes all gates and the operation is done.

Can a single byte in RAM and a single CPU instruction in RAM both exit RAM at once?
No, not possible with traditional system board. You can do each one very quickly, but not simultaneously.

What does the circuit of a full 2-byte r/w memory look like? Size of RAM is 2-byte, need 16 flip flops. Need a r/w register, data register and some sort of addressing system (many gates, only opening access to 1 bit at a time). Clock controls it all.

ALU Questions

What do we need to assume about our data to build an ALU Size of inputs & outputs.

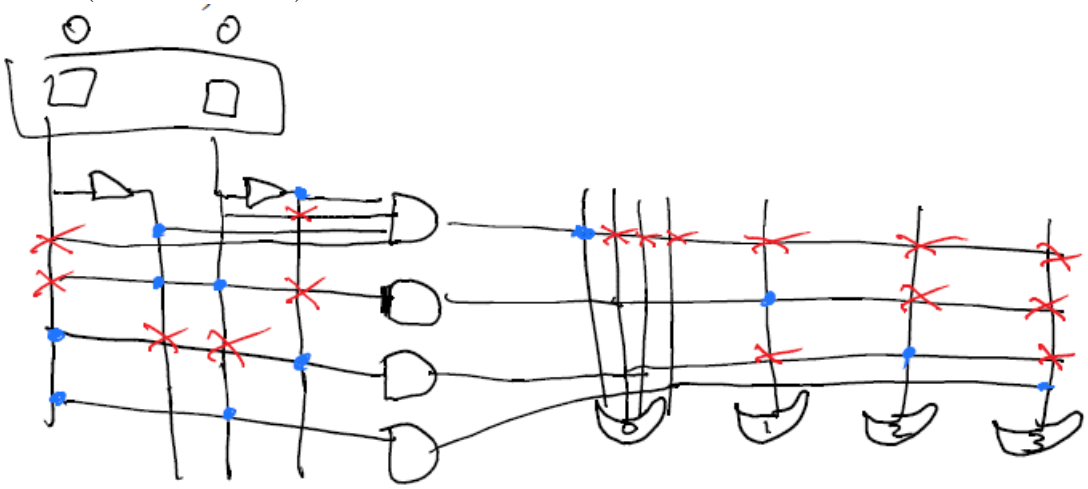
How would the assumptions impact the design of the ALU? Limit capacity to do arithmetic, depending on size of data.

4-bit integer ALU with addition and subtraction? 2 Half-adders.

Upgrading size? Add more half adders, bigger output.

Binary Code Decimal 7 segments (calculator LED), depending on which bits are on, display that number. Uses addressing.

Given 2 bit binary number, use PLA to send out single bit value for each binary value. (Like a decoder)



Temperature stored in a register as a binary number, set off alarm when temp is $\geq$ 500 Do all the situations 5+, then put them all into an or.

9 Exercises

9.1 Circuits

- Make a circuit with a truth table
- Simple 3 bit increment with 1 circuit, no adder
- 2's complement circuit
- Circuit picking 1 bit out of 2 bits, given a 1 bit number

10 Readings

Computer Organization & Design Textbook Readings correspond to the 4^{th} edition of the textbook, not the 5^{th}.

- Chapter 1
- Chapter 2.4
- Appendix C1 to C11
- Chapter 4
- Appendix D