# COMP 273: Intro to Computer Systems Review

Julian Lore

Last updated: April 12, 2017

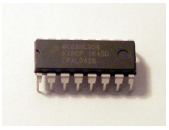
Adapted from Joseph Vybihal's Winter 2017 COMP273 slides.
Some things taken from Allan Wang's website.

## Contents

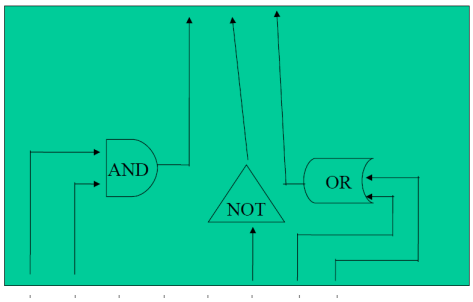
# 1 Basics

## 1.1 Bytes

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



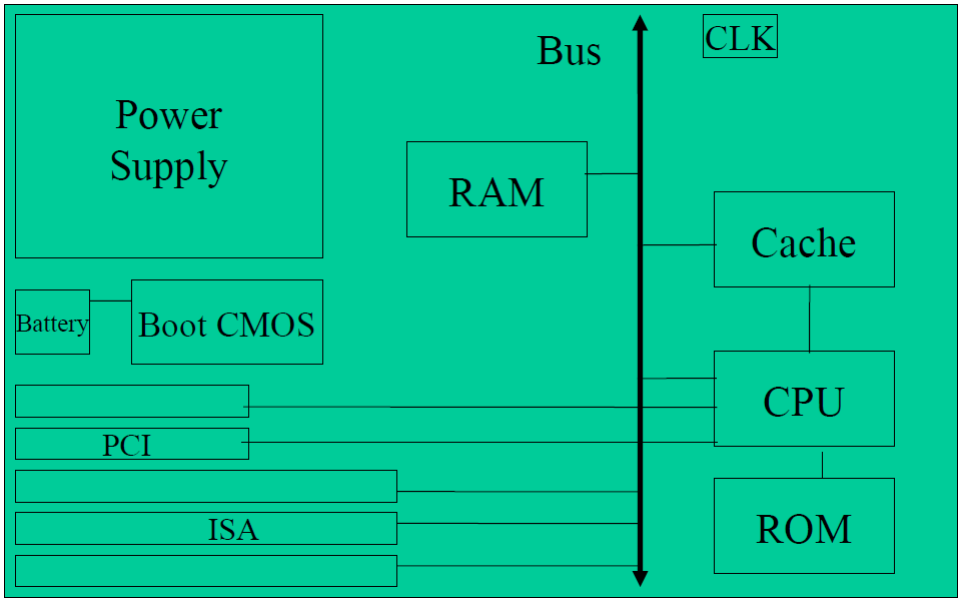
5v signifies T/1, whereas 2v signifies F/0.
Bytes consist solely of 3 gates (AND, OR, NOT), other gates can be made from these.




## 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.

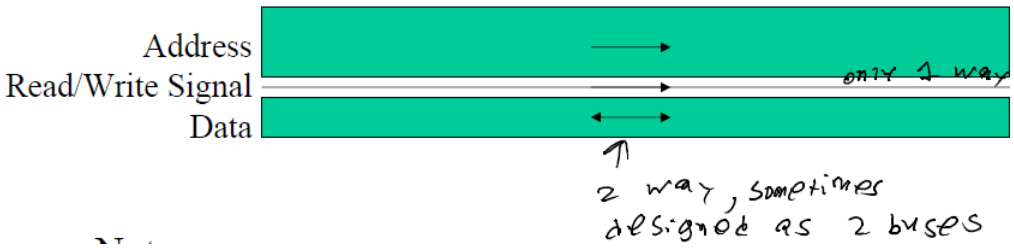
# 2 System Board




The system board consists of data paths, such that everything is connected.

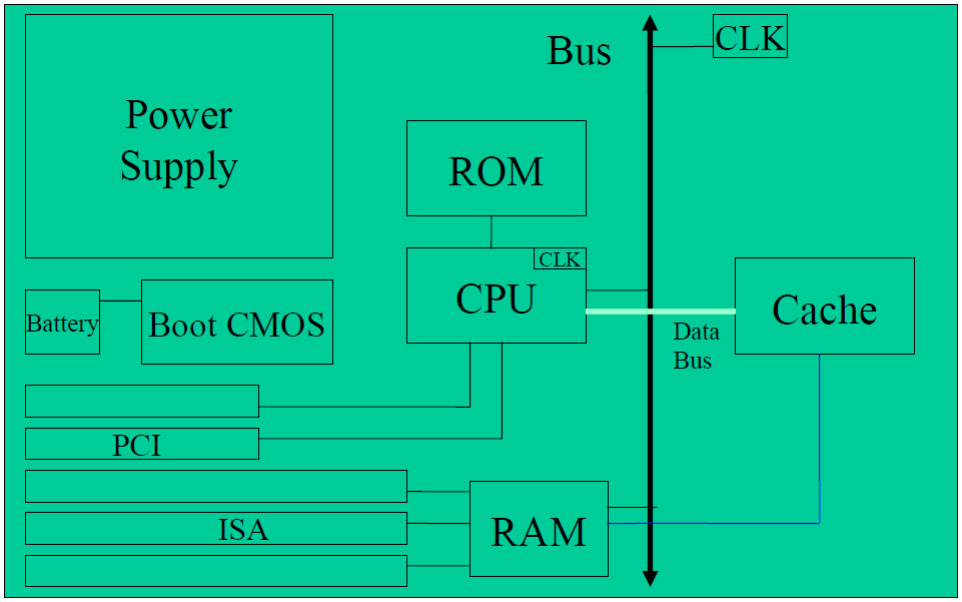
## 2.1 Bus

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:

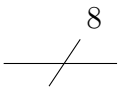
- From the CPU to the cache (cache can operate at CPU clock speed)
- Cache to RAM
- CPU to PCI
- All of these make things faster
- Great use of speediness of CPU




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

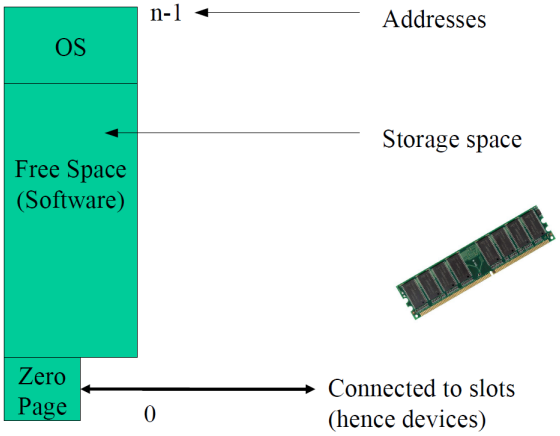
## 2.2 Pathways

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

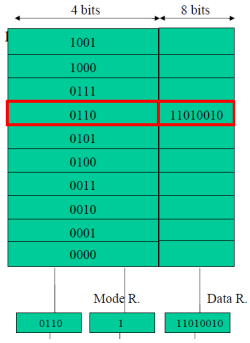
Volatile (live data) general purpose main memory bank, large & slow, **RAM** consists of addresses, storage space and the zero page, which connects to slots. Comparable to an array.




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.

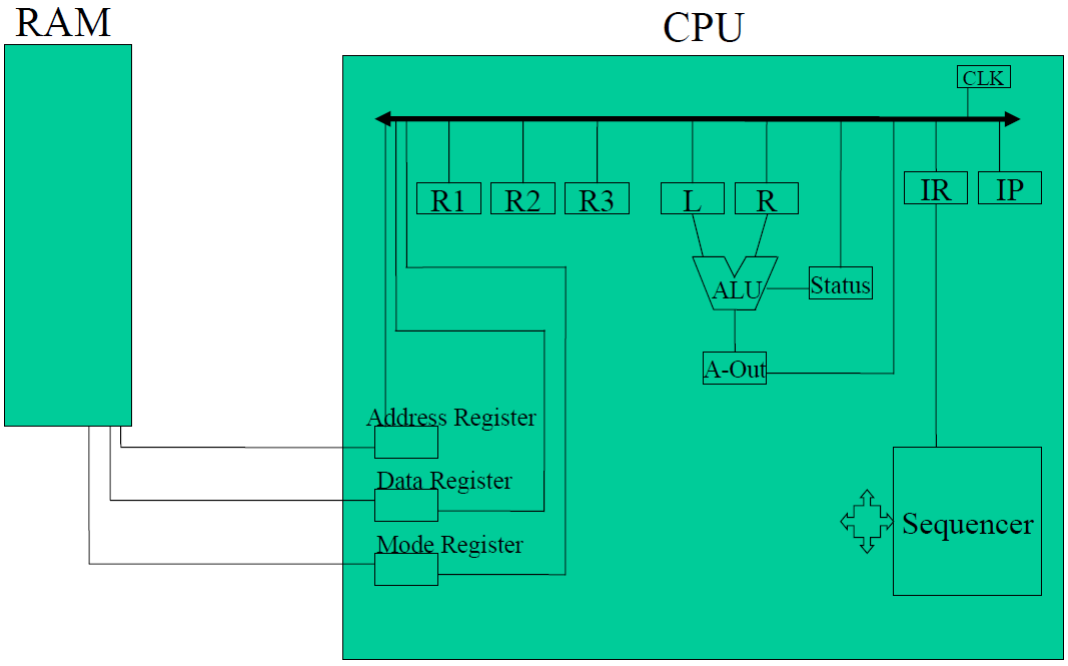
## 2.6 Addressing

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

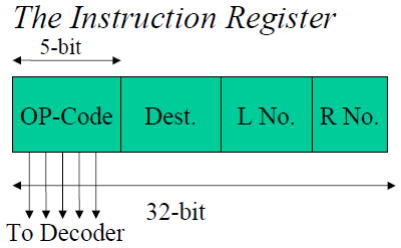
**IR (Instruction Register)**

- 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)




- 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.

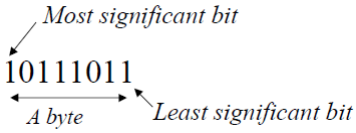
- RAM (primary storage). DRAM: Dynamic, must refresh. SRAM: Static, no refresh.
- ROM, read only memory (advanced instructions). ROM is hardwired. PROM programmable once (fuses). EPROM, erasable by heat. EEPROM electrically erasable. PAL, PLA.
- Cache. Very fast memory, usually on CPU. Store frequently 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.

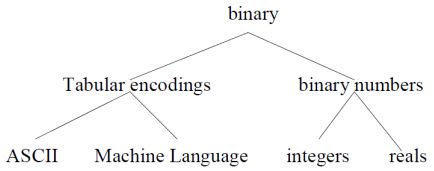
| | |
|---|---|
| Nibble | 4 bits |
| Byte | 8 bits |
| Word | 16 bits |
| Long Word/Word | 32 bits |
| Quad Word/Word | 64 bits |

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**

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

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

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

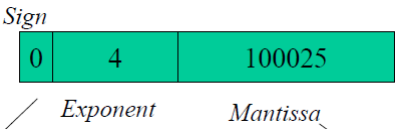
**Signed vs 2's Comp**

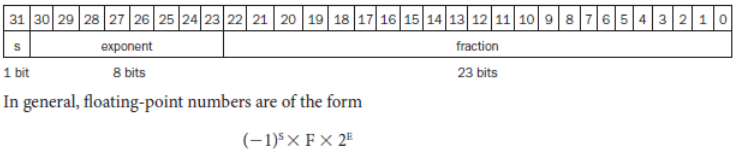
- Signed
  - Easy to read
  - No conversion
- 2's
  - Unique 0
  - Auto subtracts when adding

**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.




**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

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

Converting to decimal to check our results:

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

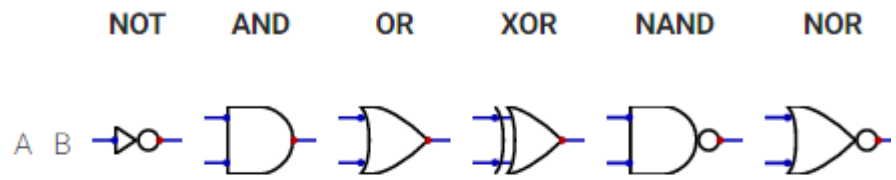
The product of $0.5_{ten}$ and $-0.4375_{ten}$ is indeed $-0.21875_{ten}$.

# 4 Logic & Gates

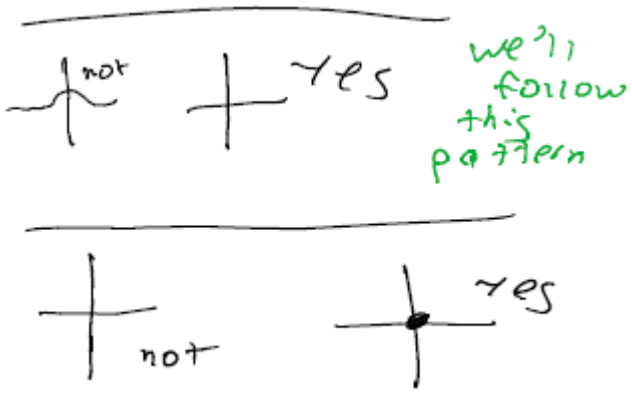
- Circle $\implies$ not
- Extra line $\implies$ exclusive

Figure 1: Taken from Allan Wang's website

Gates

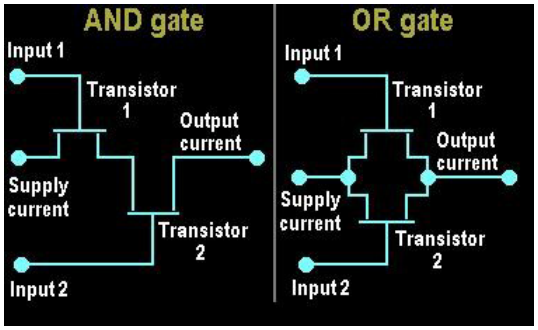


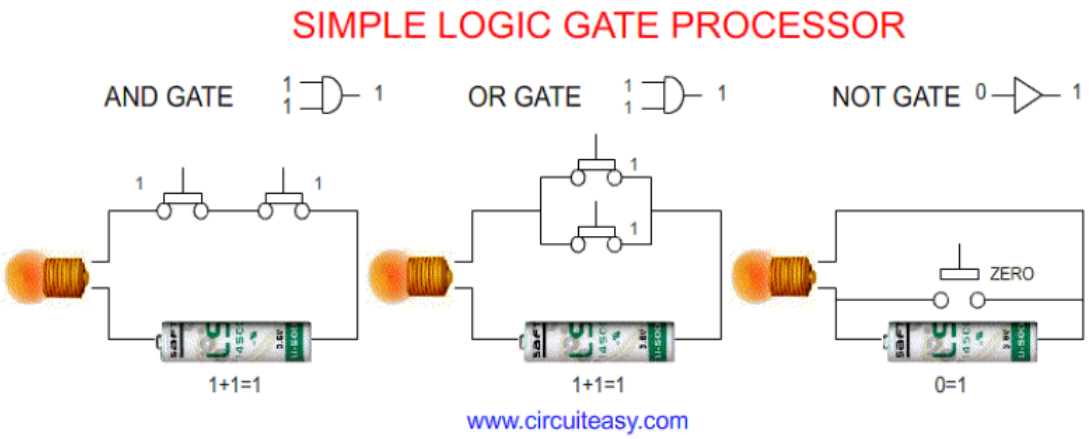
| A | B | NOT | AND | OR | XOR | NAND | NOR |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | X | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | X | 1 | 1 | 0 | 0 | 0 |




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

**Low-level Construction**

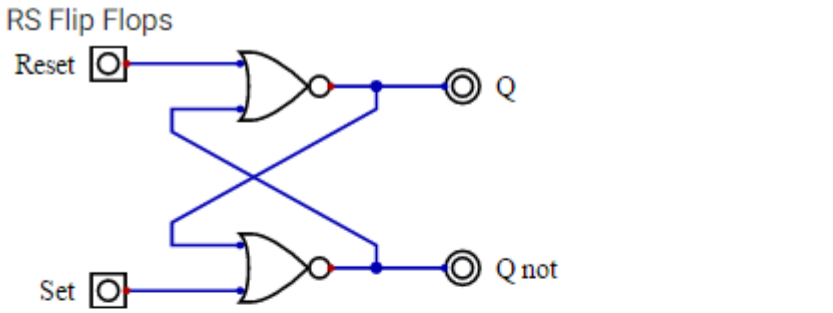



## 4.1 Basic Flip Flop Circuits

**Flip Flops** are bits!

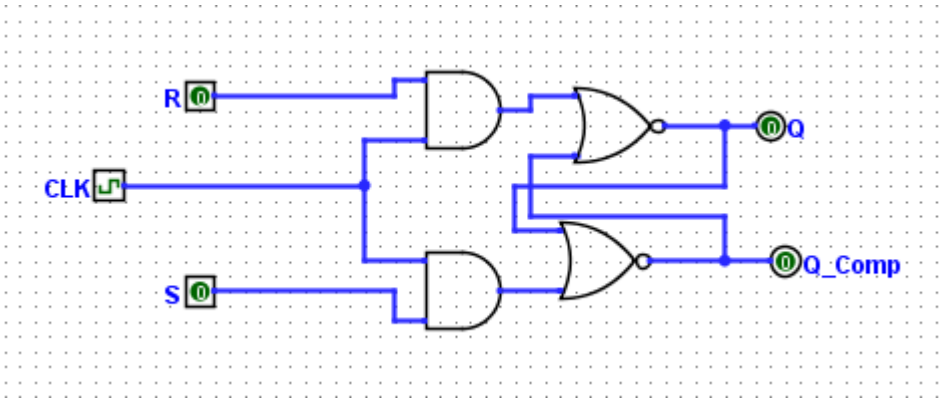
Figure 2: RS Flip Flops, Allan Wang




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

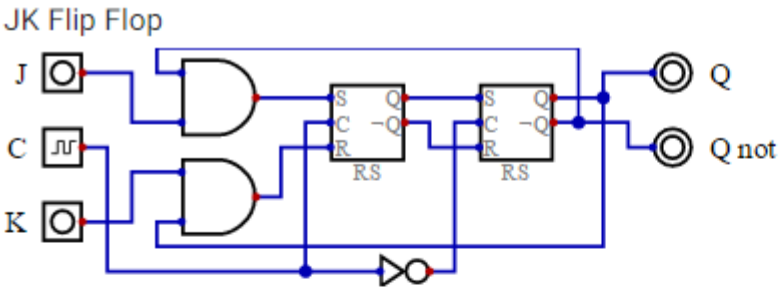
| R | S | Q | Q' | Result |
|---|---|---|---|---|
| 0 | 0 | Q | Q' | No Change |
| 0 | 1 | 1 | 0 | Set |
| 1 | 0 | 0 | 1 | Reset |
| 1 | 1 | 1 | 1 | Avoid |

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

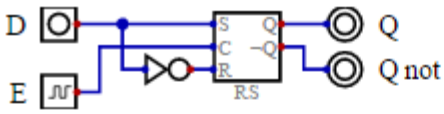



JK flip flops cycle at half the speed of its input, as only one SR flip flop is enabled at a time and it takes two clicks to pass data to the output.

| 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

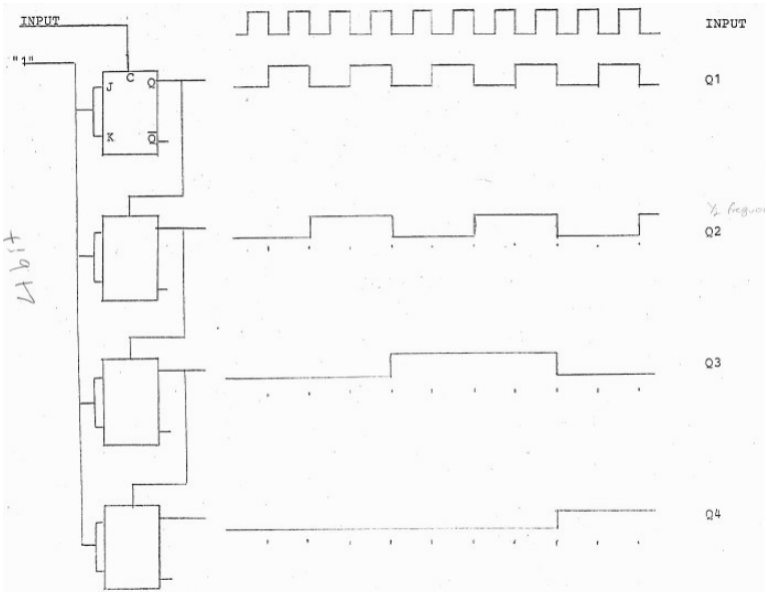



An addition to the RS flip flop to accept a single input. Together with the clock (E), the input (D) directly changes the output of the latch. This also eliminates the Reset = 1 & Set = 1 issue.

| 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.

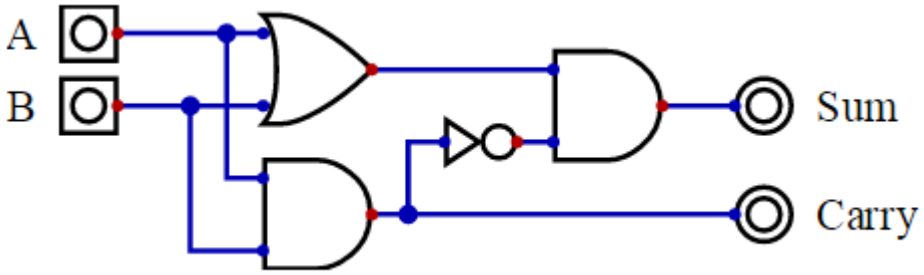


## 4.2 Adders

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

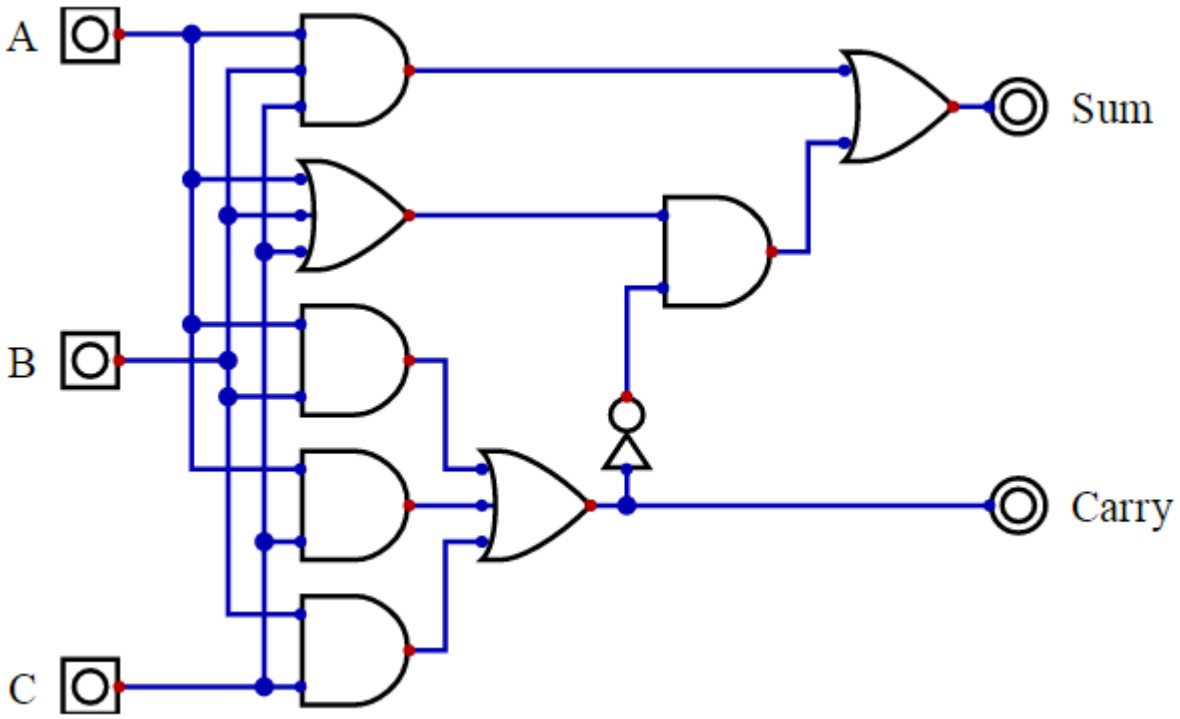
Figure 5: Half Adder, Allan Wang




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

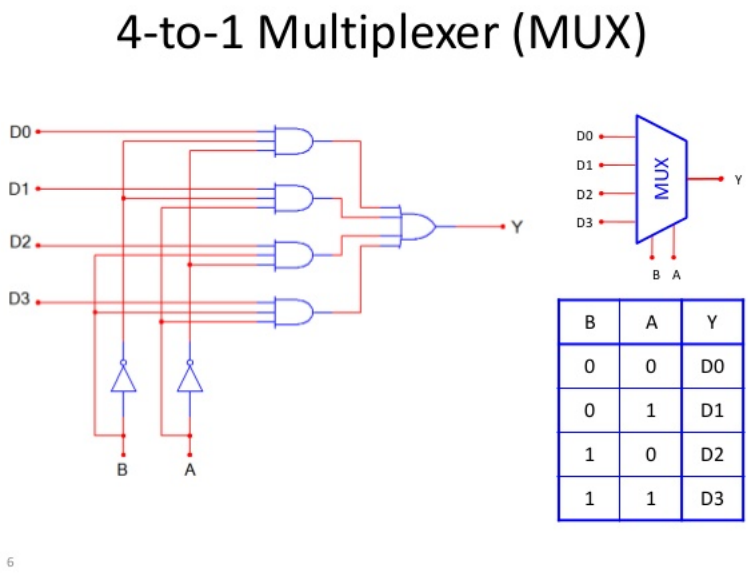



Above is the more optimized version. Can just have 2 half adders and an or. Include simpler version with half-adders?

## 4.3 Combinatorial Networks

Premade circuits for specific purposes, readily available.

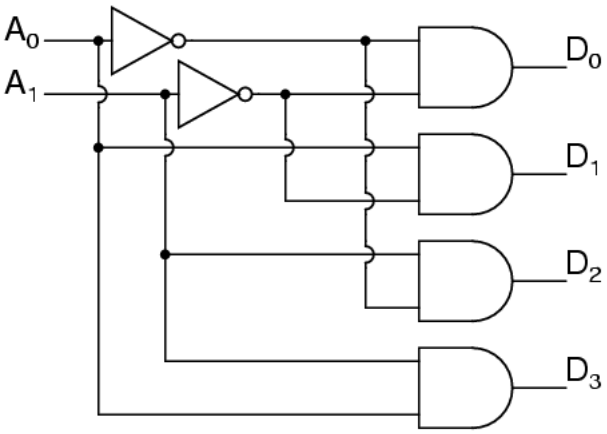
**Multiplexers** $2^a \to 1$ mux, $2^a$ inputs $x_0, \ldots, x_{n-1}$ and a single output z. Selector address a, with input signals $y_0, \ldots, y_{a-1}$, selects which $x_i$ to pass through z. Useful for when you have multiple signals but can only let one through at a time. Addressing, but with 1 output,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




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

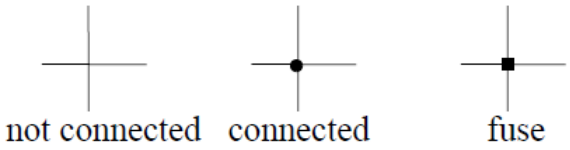
6

**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.



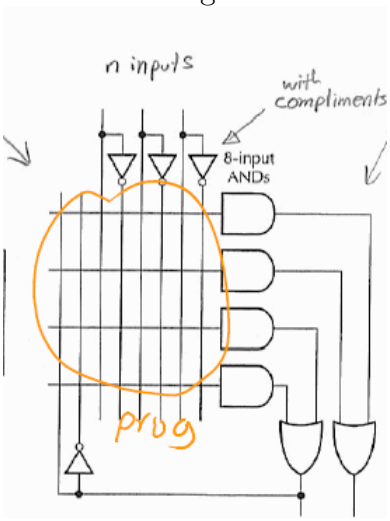

**Encoders** Outputs a-bit binary number, $y_i$ equal to index of single 1 signal on among $2^a$ inputs $x_0, \ldots, x_{a-1}$. All of the xs are connected to an or gate, so if they're all off but $x_0$ is on, will show 0.

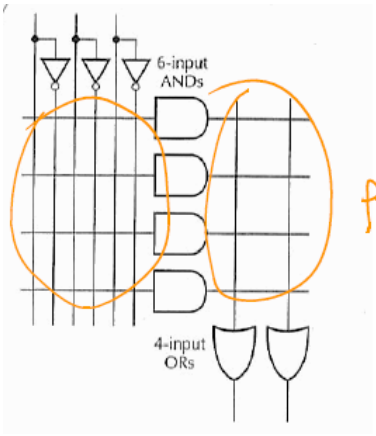
**Programmable Combinatorial Parts** VIA Fuses that can be blown (disconnected) from enough current or VIA anti-fuses that are set(connected) from enough current.




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

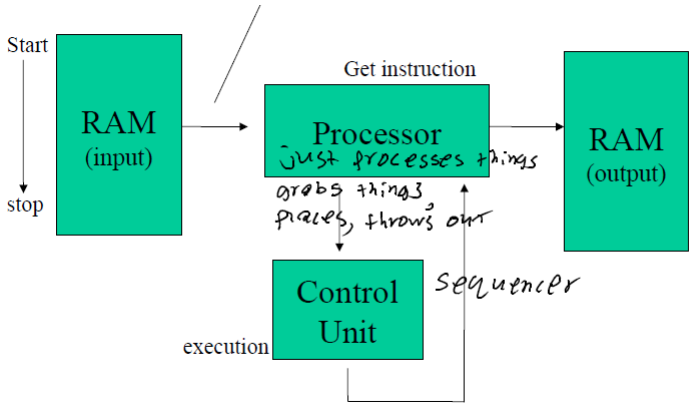
**PAL** Programmable Array Logic, input wires are programmable, but output isn't.



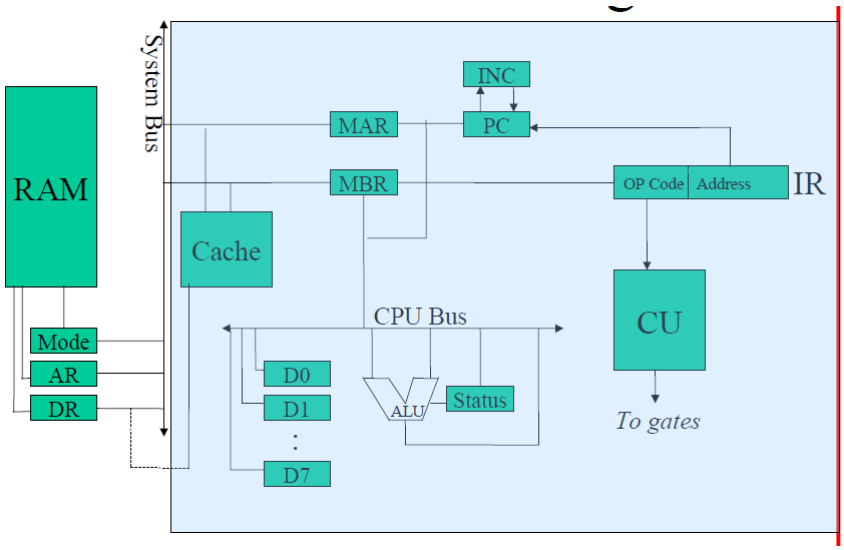




**PLA** Programmable Logic Array, input & output programmable.

# 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)

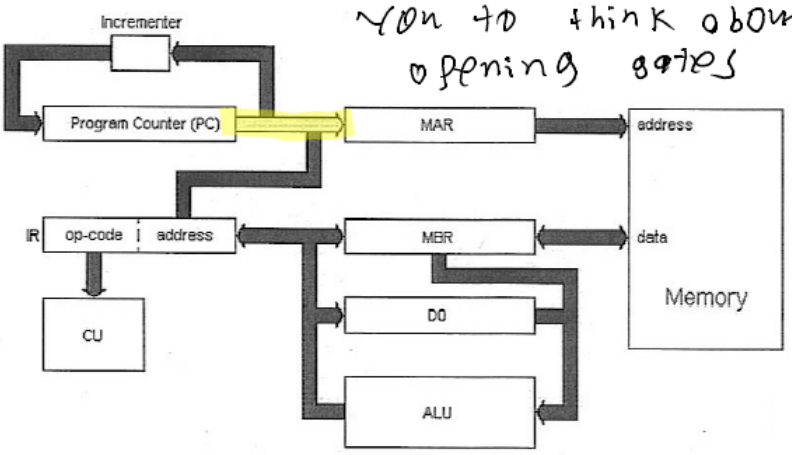
```
FETCH    [MAR] <- [PC]
         [PC] <- [PC] + 1
         [MBR] <- [M([MAR])]
         [IR] <- [MBR]

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

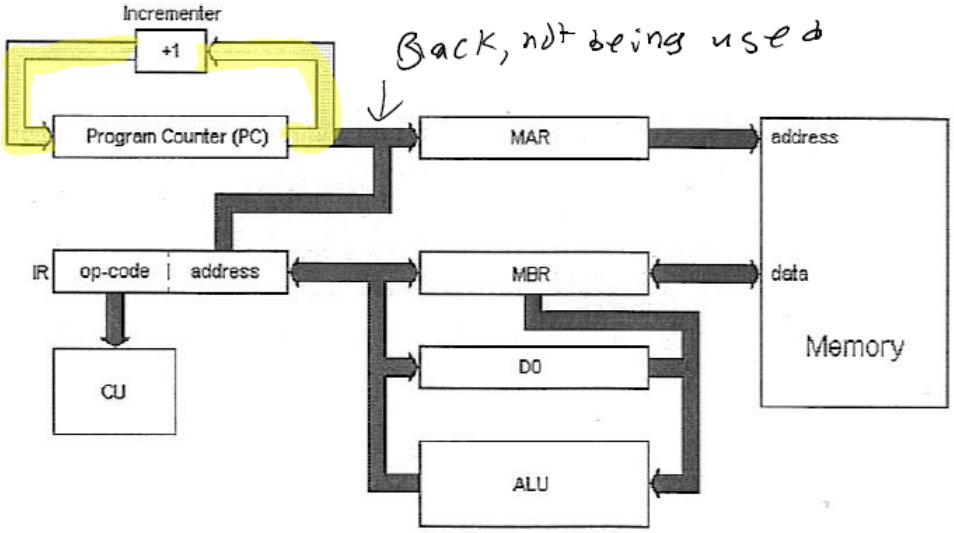
eg
ADD D0,D0,X

1) [MAR] <- [PC]

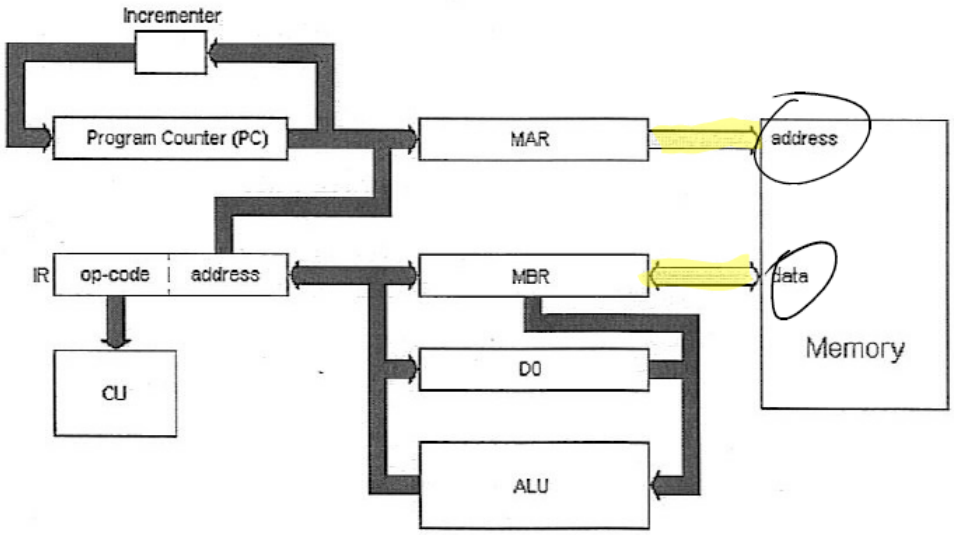
picture design allows you to think about opening gates



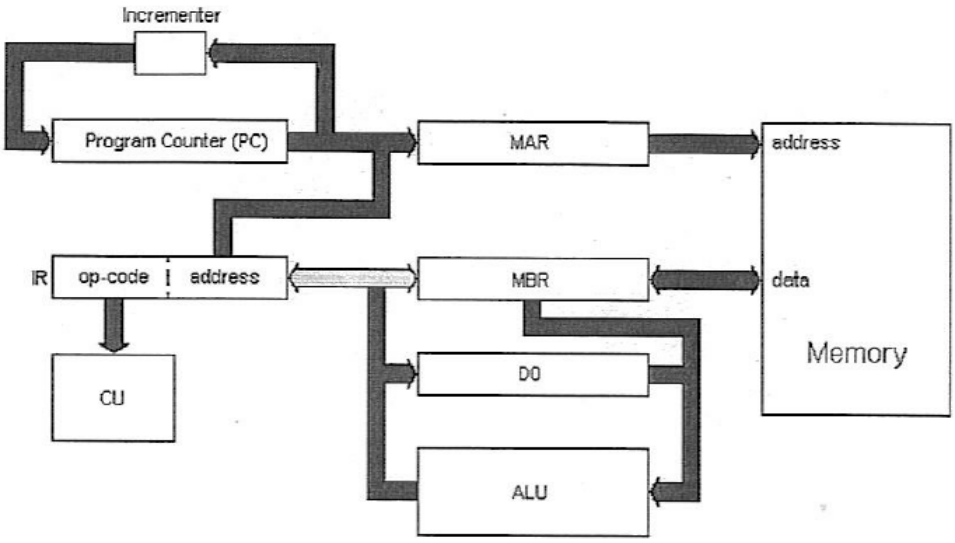

2) [PC] <- [PC] + 1



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

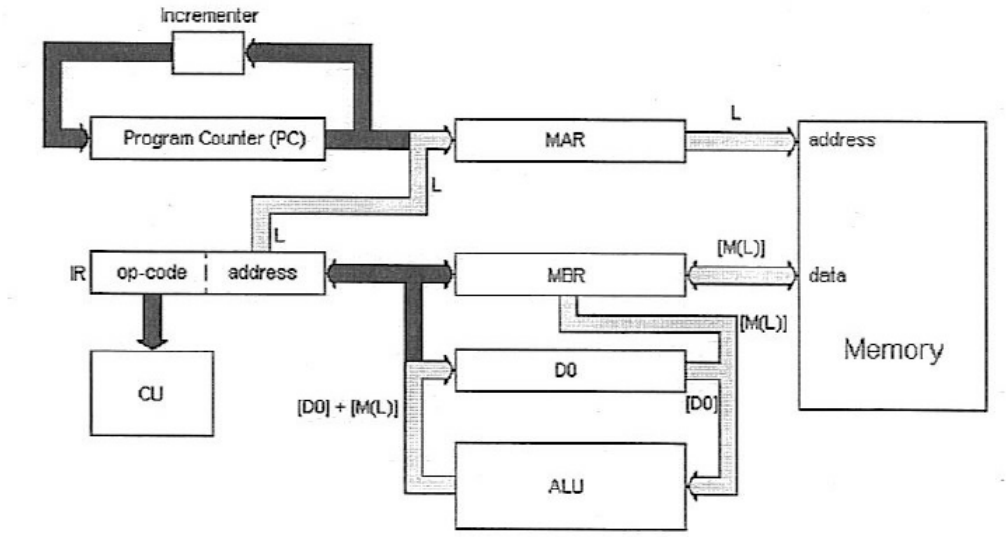



4) `[IR] <- [MBR]`

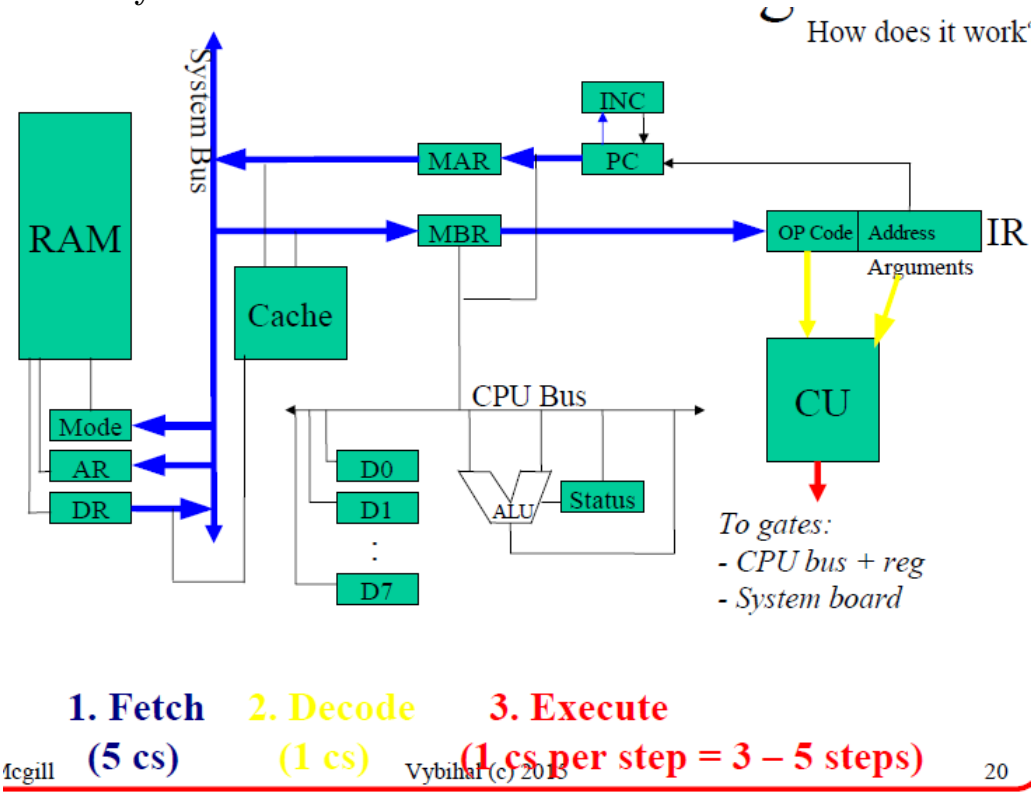


5) The execution phase:

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



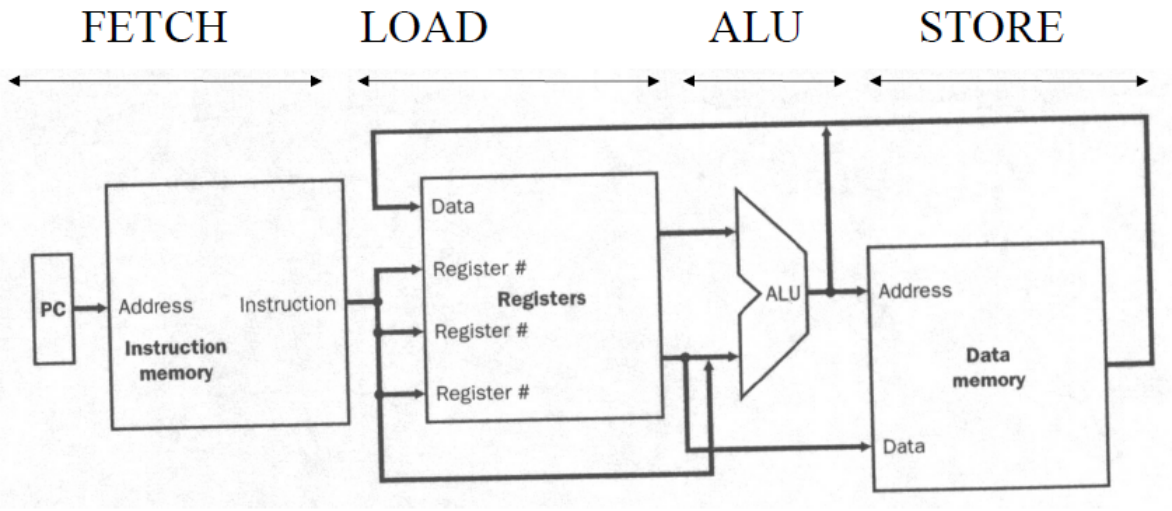

**Summary**



# 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.

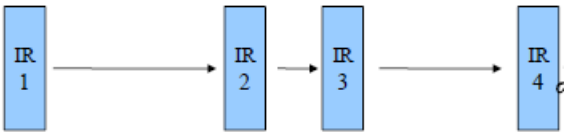
What about the pipeline CPU? The CPU **is** the bus. Only goes in one direction. Each instruction does a separate part at once, so you can have multiple instructions flowing through the pipeline at once (not executing simultaneously, more like doing different steps simultaneously). Pipeline instead of 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 go-to/jump). The linearity/one way of the bus imposes restrictions on the language. All instructions go through the pipelin, need a wide 32 bit+ size bus.

### Effects on IR and CU

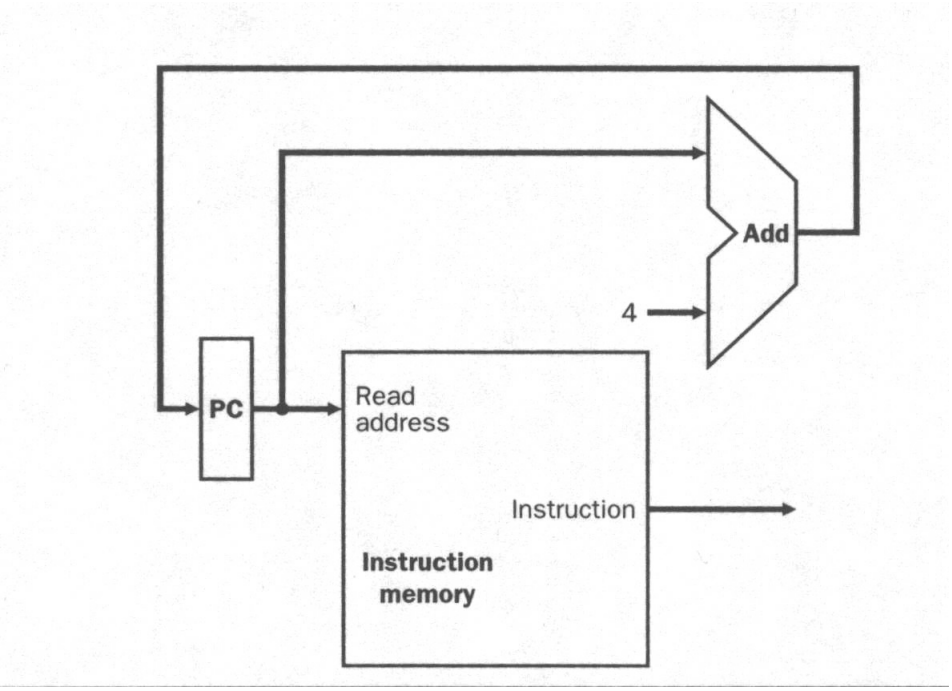
- No longer 1:1 IR and PC
- Either one long CU or many small CUs
- 1 IR for every silo (data lock/gates), i.e. every step




- Pipeline has to be gated

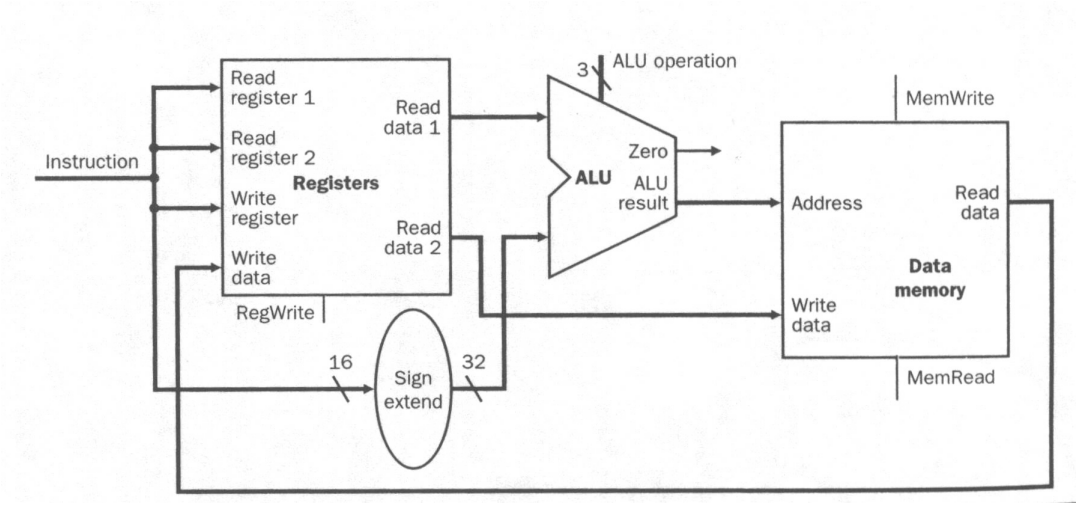
**Fetch**

- Read address from PC, get instruction and send it out.
- Increment by 4 since 32-bit instructions.



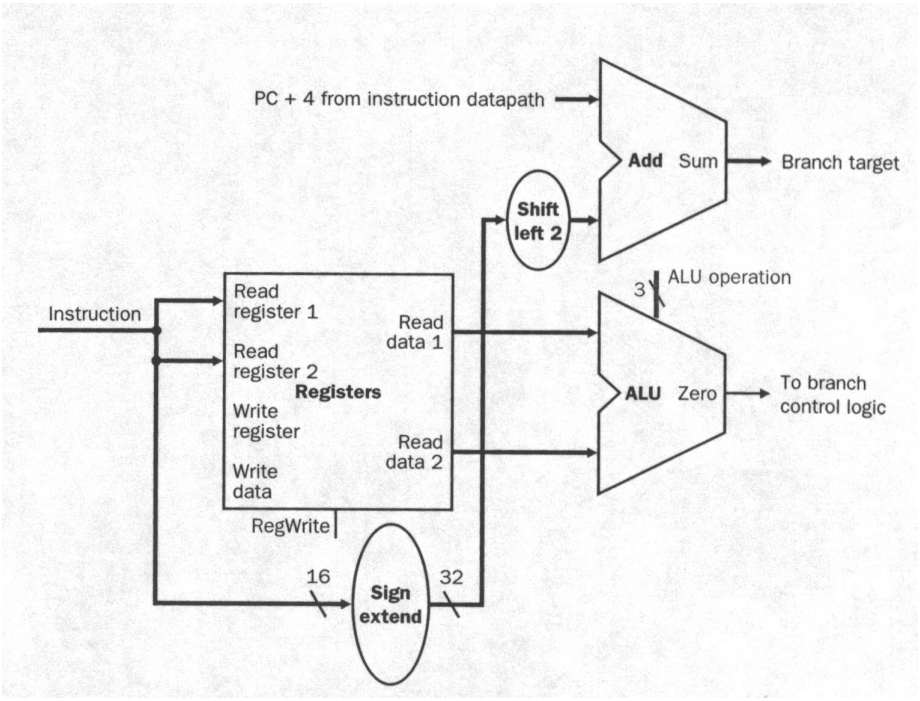

**Load & ALU**

- Instruction goes to registers
- After op code and registers that are arguments, have 16 bits leftover, have to sign extend, add 0s in front
  - This is for short constants, like offset of a LW instruction
  - Some instructions don't require ALU, like $x = 5$. Look at read data 2, skips ALU




**Delayed Branching ALU**

- For things like BEQ, bottom ALU checks conditions like $X > Y$
- If true, will send op to upper ALU, new address with offset specified
- Will have to dump everything loaded after branch if we branch

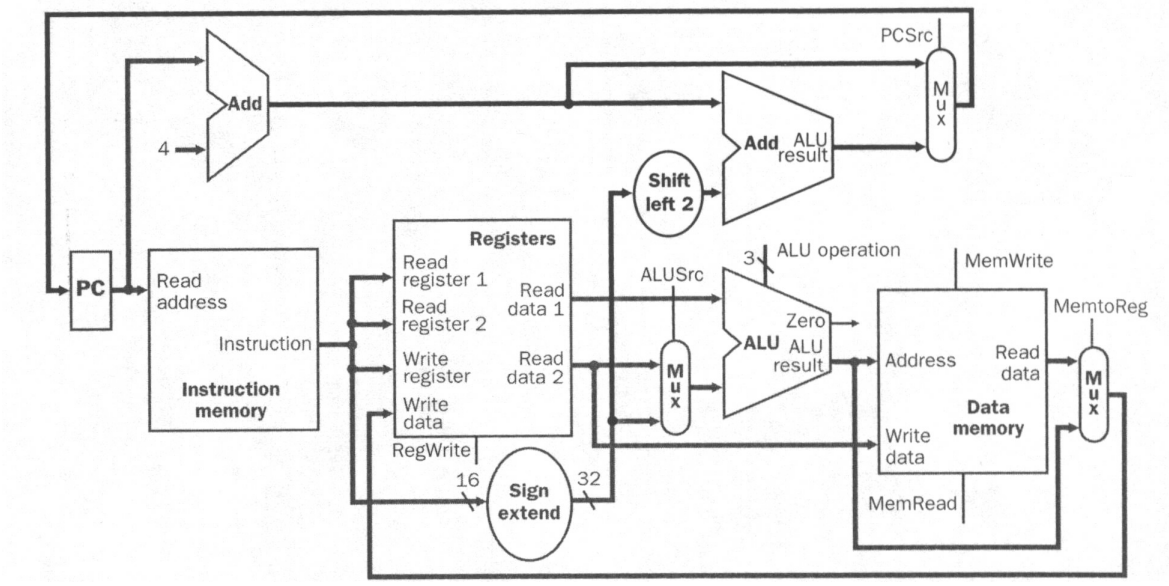



**Multi-Purpose ALU** ALU does different things depending on instruction, diff control codes.

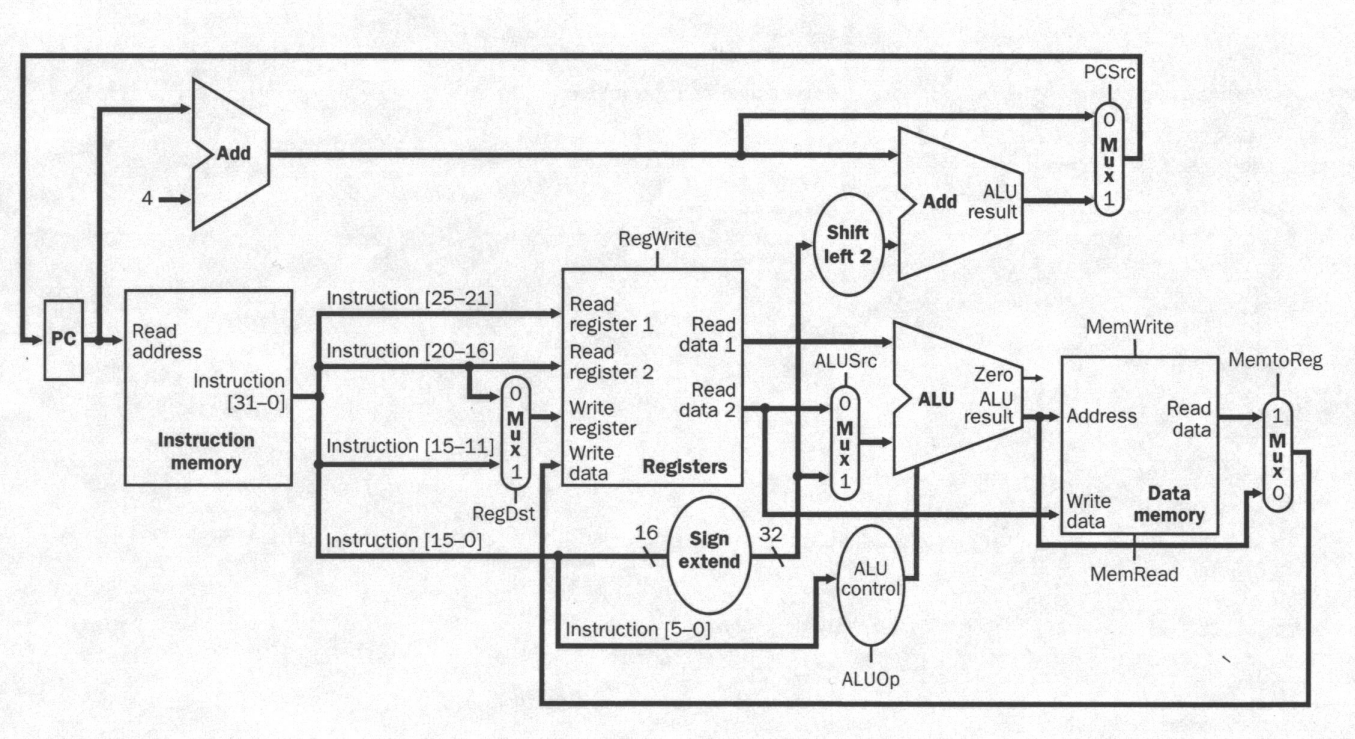
| Instruction opcode | ALUOp | Instruction operation | Funct field | Desired ALU action | ALU control input |
|---|---|---|---|---|---|
| LW | 00 | load word | XXXXXX | add | 010 |
| SW | 00 | store word | XXXXXX | add | 010 |
| Branch equal | 01 | branch equal | XXXXXX | subtract | 110 |
| R-type | 10 | add | 100000 | add | 010 |
| R-type | 10 | subtract | 100010 | subtract | 110 |
| R-type | 10 | AND | 100100 | and | 000 |
| R-type | 10 | OR | 100101 | or | 001 |
| R-type | 10 | set on less than | 101010 | set on less than | 111 |

## 6.1 Summary Layouts

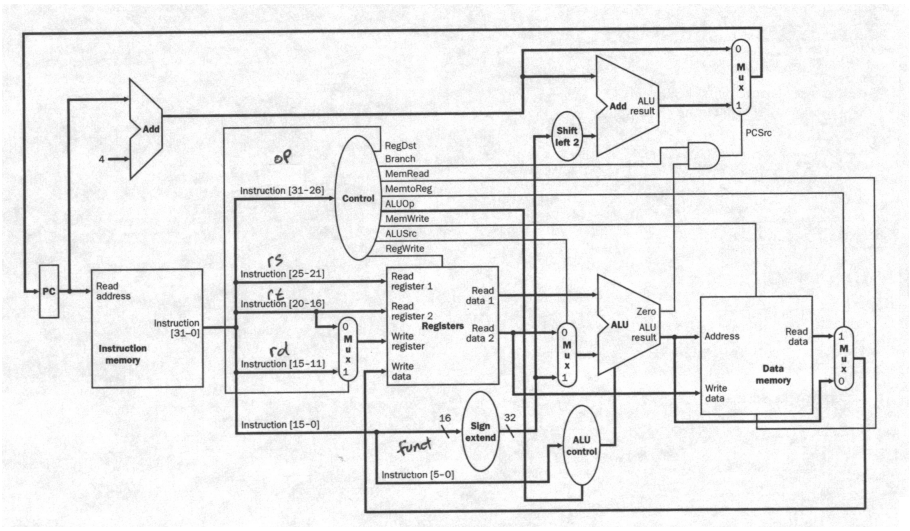
- Can see that there are 4 silos here
- Mux $\rightarrow$ multiplexer, decides which input to take in

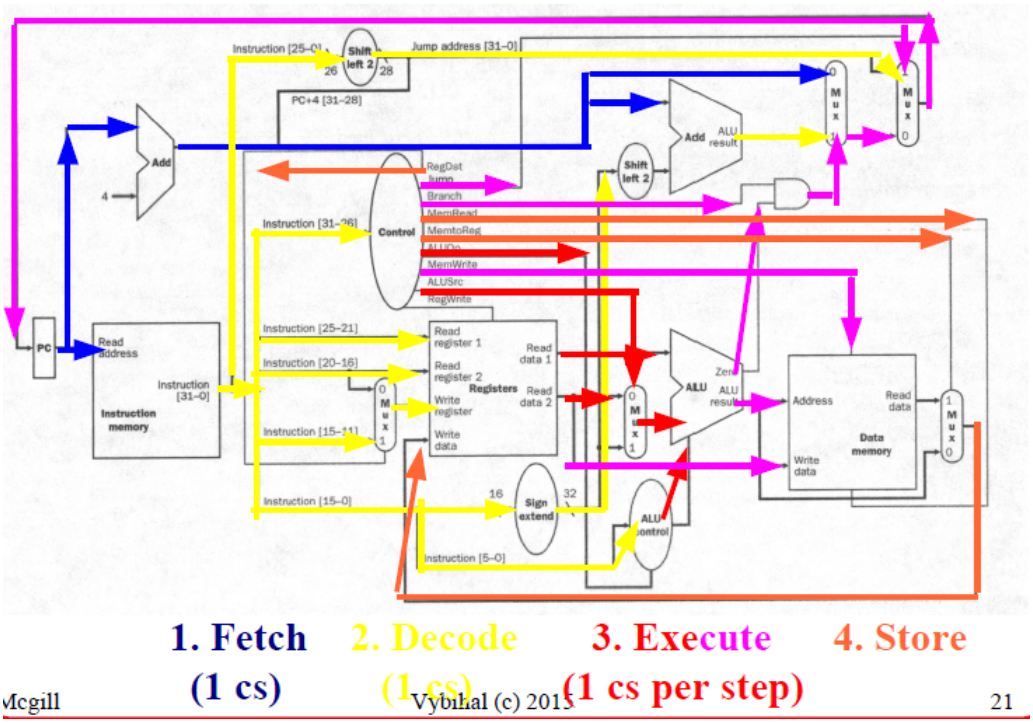



- Bit Layouts




- 2 control units




# 7 Control Unit/Sequencer

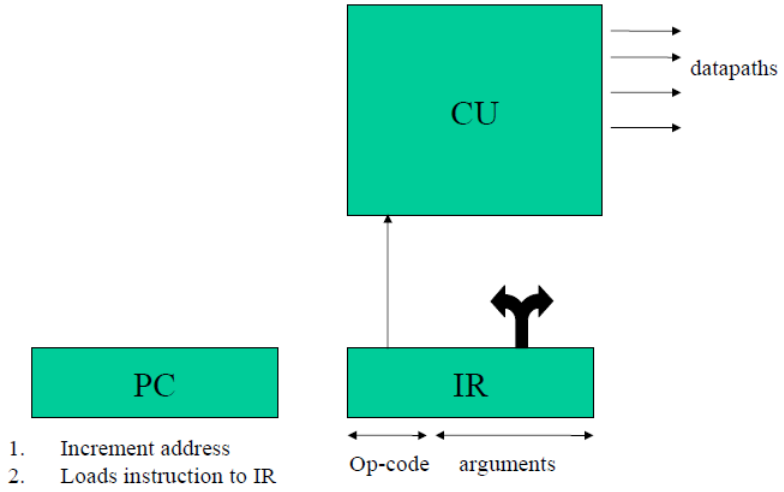
CU is the portion of the CPU responsible for timing and triggering of data paths.

**Datapaths** Wires of CPU needed for particular tick/instruction.

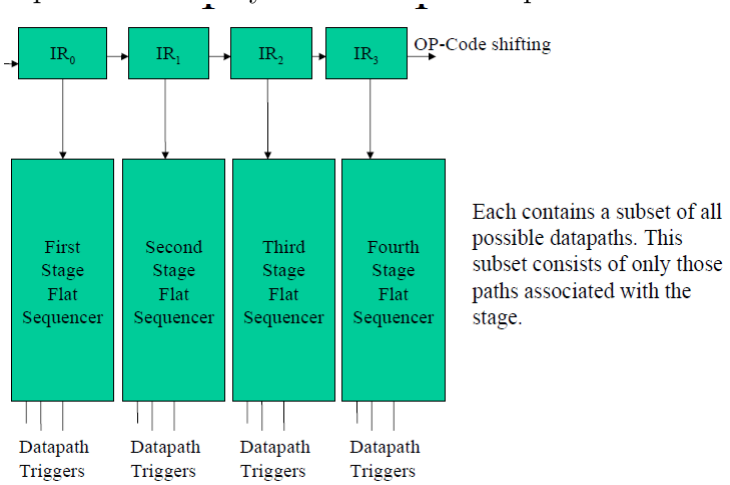
**Instruction Format** Organization of bits in IR

### Micro-programming

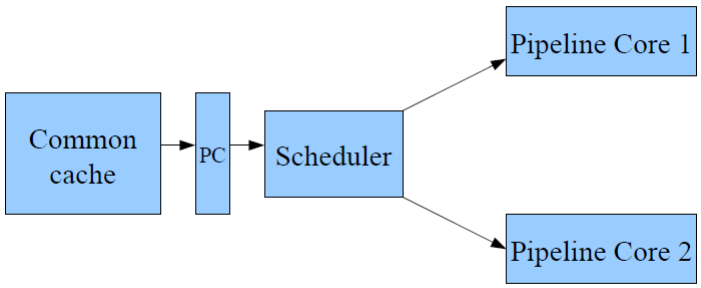
- Flat: one instr at a time




- Pipeline: Assembly execution of multiple




- Cores: Parallel exec



- Takes in an op-code register, associate numbers to instructions, like addressing. Open needed gates for specific instructions, mux
- The CU is like a decoder
- All commands belong to a certain family (see MIPS instructions)
- CU has a counter to tick the microinstructions of one instruction, op-code implemented via multiple sub-steps, works well if we make all instructions the same tick length
  - When it overflows (gets to 0), resets itself for next instruction
- Pipeline has IR at each silo and 2 CUs, with 0-1 count

**Buffers** Multiple IR registers (buffers) for each instruction in pipeline.

# 8 Performance

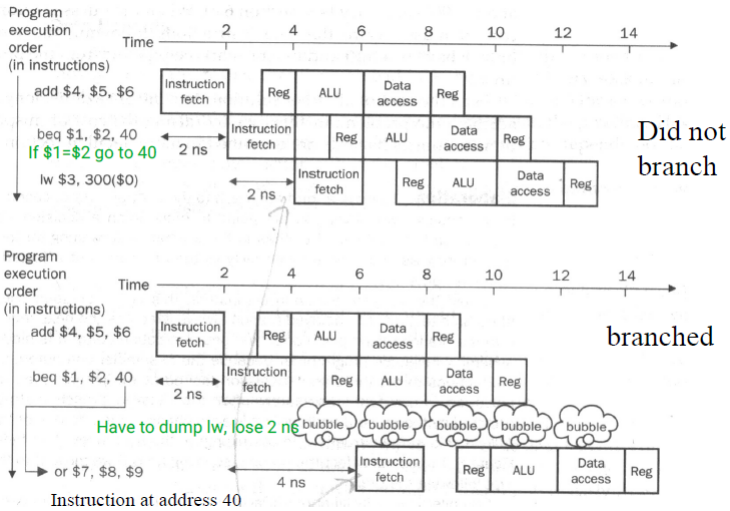
## 8.1 Hazards & Faults

- Hazard: danger to keep watch for
  - Not being able to control what user can divide by, hazard
- Fault: error has occurred
  - Dividing by 0, fault
- Stall: Normal CPU exec cycle lengthens
  - Cache smaller than RAM
  - Small loops, try to fit it all in cache to run quickly
  - Otherwise it slows down to get parts in RAM
  - Stall might happen if CPU is running faster than code → stop pipelining and do classical or no-op
  - Instructions going from CPU to RAM, slower clock speed, stalls
- Dump: Instructions after fault dumped from pipeline, re-loaded after fault handled
- No-op: Extra instruction that doesn't do anything, between fault and next instruction, allows CPU to exec problematic instruction without many side-effects

hazard→ fault → stall → dump or no-op
ALU has a status register. If CU is in a certain operation (ex. div) and ALU gives an error, it'll stop program. Go back to PC or OS register, stop incrementor, OS looks at error.

### Types of Hazards

- Structural Hazards
  - Bad combination of instructions, store/load in 1 Instructions
  - Illegal instruction or result
- Control Hazards
  - Branching (pipeline), making previous half executed instructions useless, comparison only happens at ALU




- Data Hazards
  - Instruction depends on result of previous instruction, save only happens at 4th stage for pipeline
    - Make CU check if instructions use same arguments
  - Use NOP or buffer to fix, can also reorder

**Loss** Calculate loss. At what stage did the fault happen? How many no-ops? How many things need to be reloaded after dump? How many instructions don't have faults (full speed)? The performance loss experienced → $E[Loss] = p(Loss) * cost$, stalls happen 17% of the time, still worth it to use pipeline.

**Solutions**

- Insert no-ops or reorder instructions (reordering branch or instructions that conflict)
- Assume branch will always fail (bad, not used)
- Assume branch success based on type
  - Function calls good
  - Backwards jumps, probably loops, good
  - If statements, no

## 8.2 Performance Issues

- Clock ticks: Number of clock ticks required to perform activity, only reliable measure
- Cycles: count of number of micro instructions to perform an activity
- CPU execution time = $n$ instructions in prog $\times$ $Y$ ticks per instr $\times$ $Z$ seconds per tick
  - Classical CPU, different ticks for different instructions
  - Pipeline, all the same

## 8.3 CPU Exception Handling

Why?

- Bad machine language Binary
- Bad arithmetic
- Incorrect address

Supporting registers:

- Expection Program Counter register (EPC), stores addr of bad instr
- Cause, has error code
- Jump to reserved cache mem for exception assembler code

Assembler implementations

- Jump to OS exception handler

- Jump to code in user program to handle it

What ends up happening: Exception handling hardware built into CPU, has default PC address locations (hardwired) to interrupt. Error happens, PC stored into EPC, PC gets default address relating to type of error. OS or programmer has code at that address to handle.

### 8.4 Performance Impacts

**Amdahl's Law** Speedup in performance proportional to new component and fraction of work it carries out

$$s = \frac{1}{(1-f) + f/k}$$

where s is speedup, f is fraction of work, k is advertised speedup

## 9 CPU & Chipsets

### 9.1 CPU

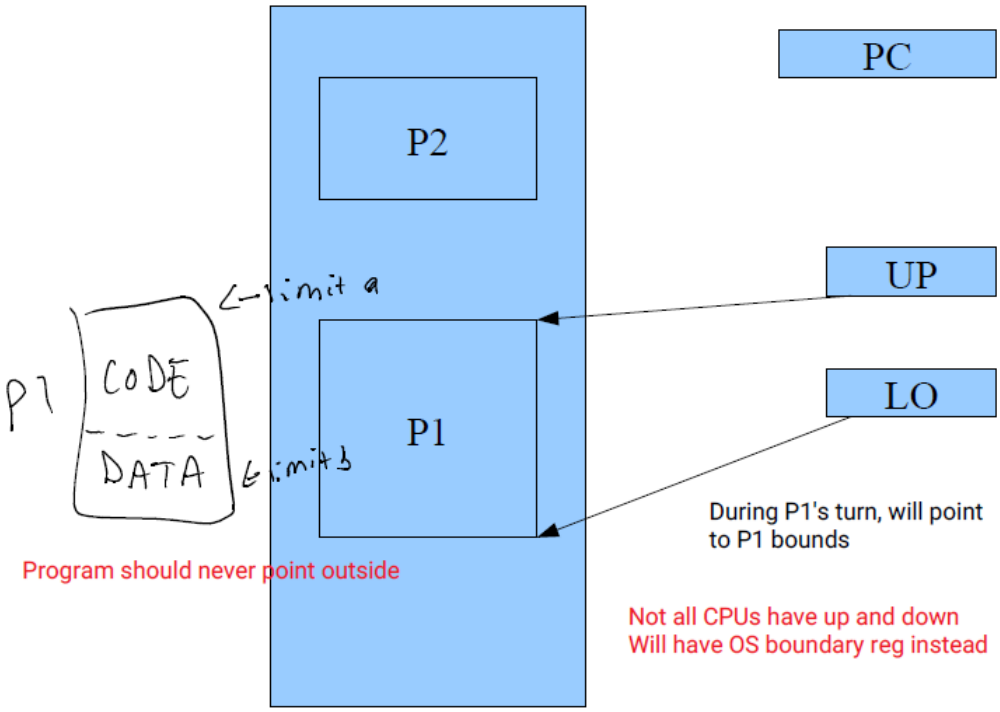
What does the CPU system consist of?

- General purpose registers
- ALU
- Cache
- OS registers
- System-Board registers
- Supporting chipsets

**Supporting CPU Registers** Running 2 programs on a computer, shouldn't interfere with each other.

- Process Boundary Register: has upper bound and lower bound for jump/branch statements compare PC with illegal situations. Protect processes, OS and zero page. OS'

job to load registers.




Some don't have upper and lower boundary registers, have OS boundary register instead, prevents processes from addressing into OS space, programs can damage each other, but not OS. For special instructions, we can use assembler instructions (like syscall).

## 9.2 Chipsets

Chips & circuitry to support CPU, your CPU needs a team.

- On Die chip sets: circuitry on CPU die, same Board
- On Board chip sets: circuitry on system board, usually clos to CPU, different bus

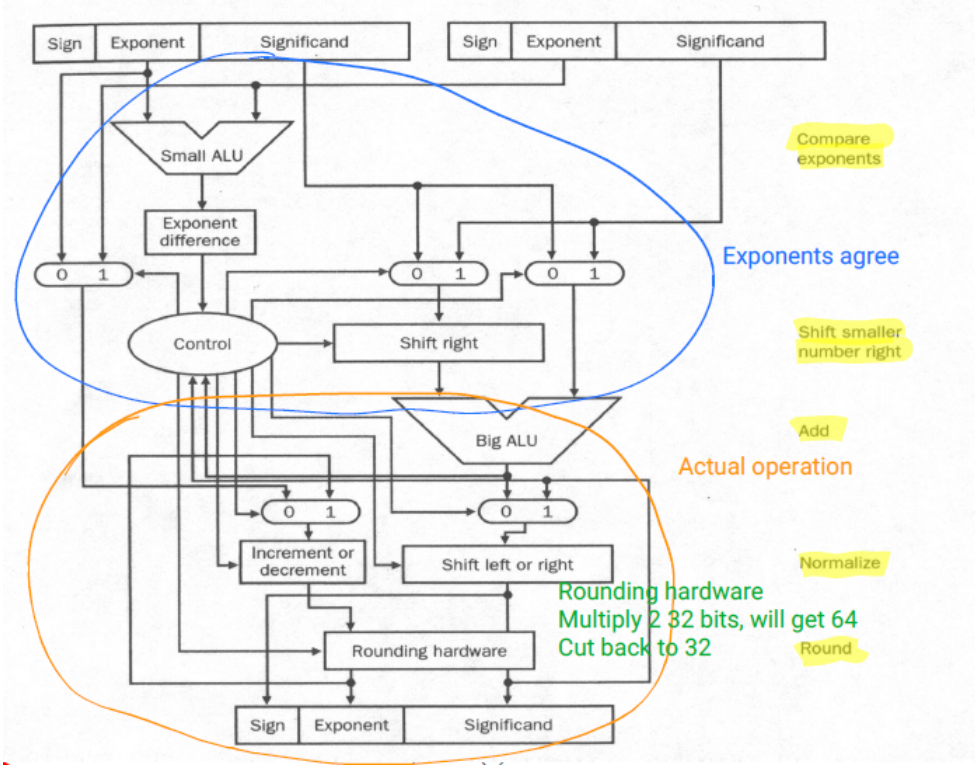
**Some chipsets**

- Co-processors
  - Math, matrix, graphics GPUs
  - MIPS only has an integer ALU, has a co-processor to do floating point arithmetic
- ROMS

- Support for video, basic graphics
- ASCII
- Ports to commmunicate, peripherals

## 9.3 Co-Processors

Like CPU, but CPU has everything, co-processor only has the things it needs to do what it was made for, i.e. math. Floating point numbers are stored very differently from integers, need a co-processor to do arithmetic.

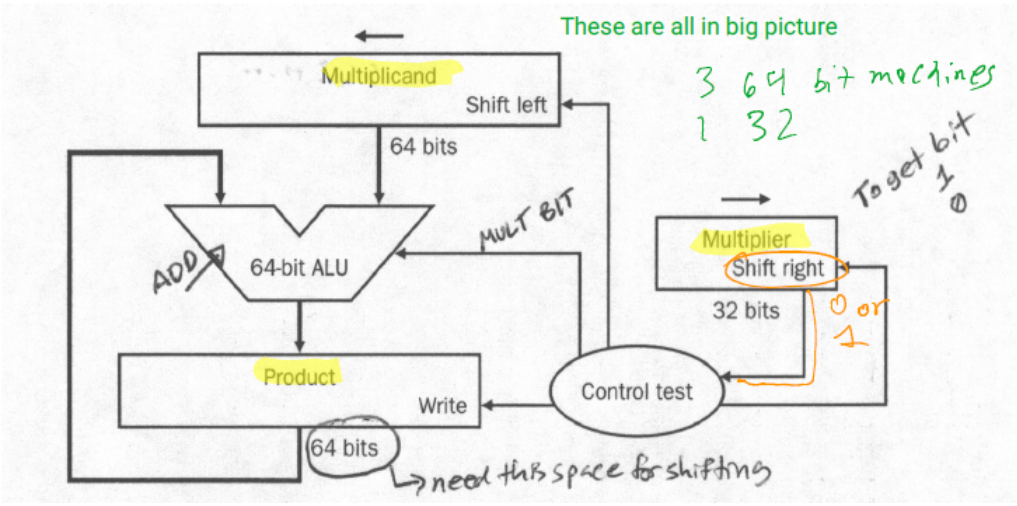



**Grade School Multiplication** Works the same way as decimal multiplication:

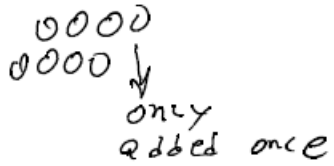
1. Right most digit of multiplier multiplies multiplicand
2. Write answer below digit
3. Next digit of multiplier, go to step 1 (writing underneath next digit implies shifting)
4. Sum

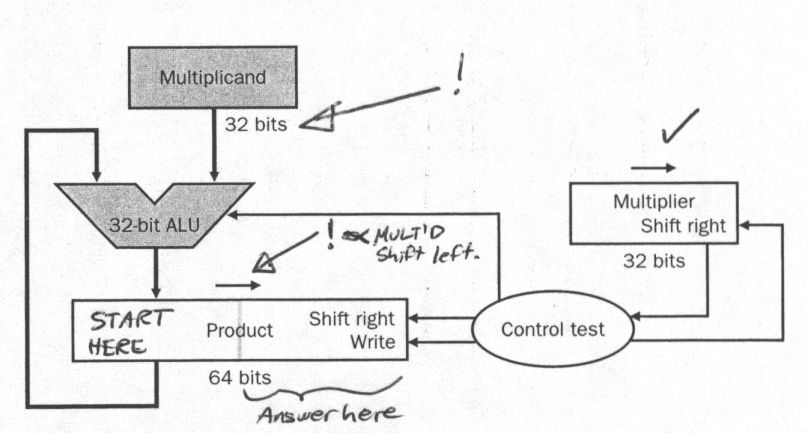
Really just need to AND everything, $1 \times 1 = 1, 0 \times 1 = 0$. Floating point numbers are 32 bits, will take 32 ticks per floating multiplication, convert to integer when possible as this is a huge stall.

**Integer Multiplier Circuit** Note that we have 3 64 bit registers here!

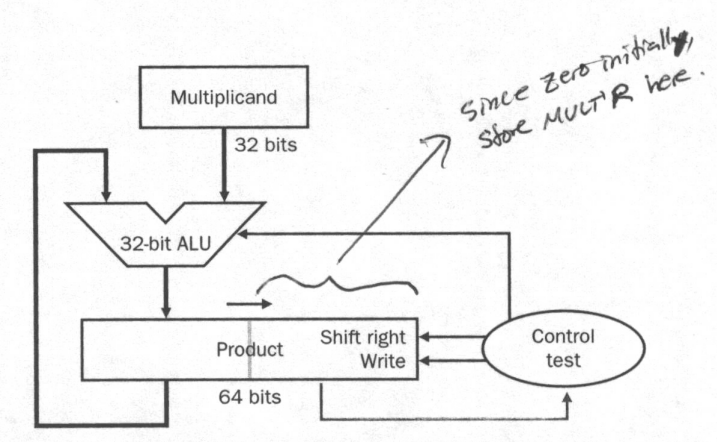



**Improvement I** Change multiplcand & multiplier to 32, right shift product register instead, Will constantly have a line splitting the 64 bit product register, because right most bits are only added a few times, least significant is not added, 2nd least is added once, etc.



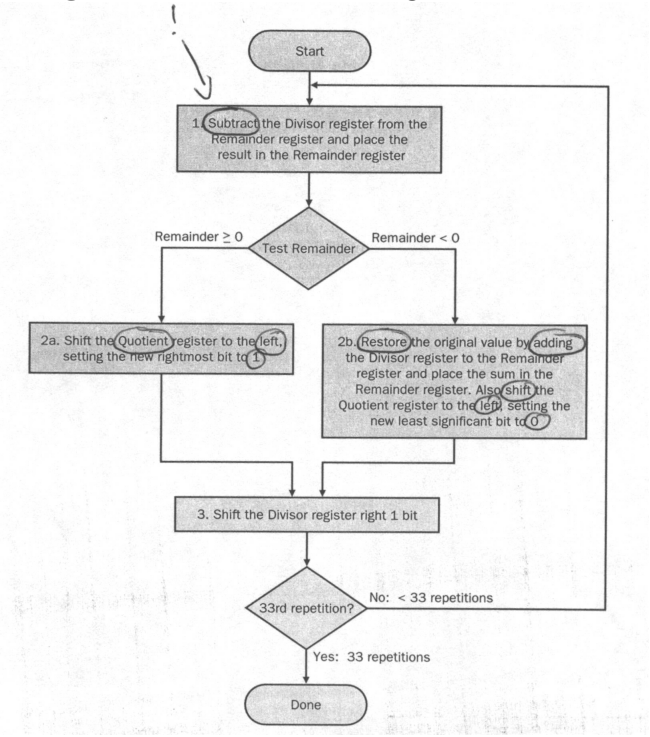



**Improvement II** No more multiplier register, placed in answer part of product instead.

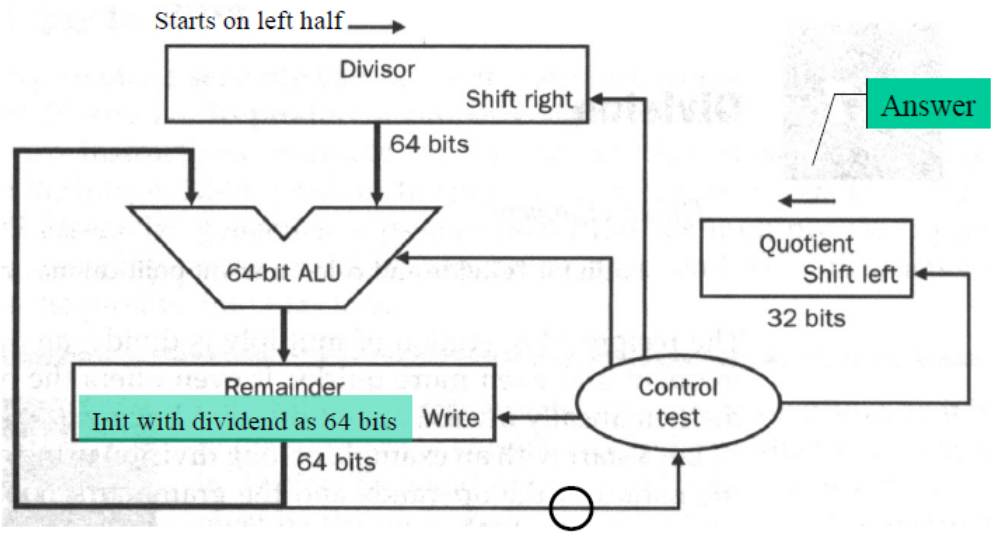



To do negative multiplication, convert to a positive value and check signs at the end
Multiplication by $2^i$ does not require multiplication (much faster), just shift left $i$ times

**Integer Division** Same as grade school division

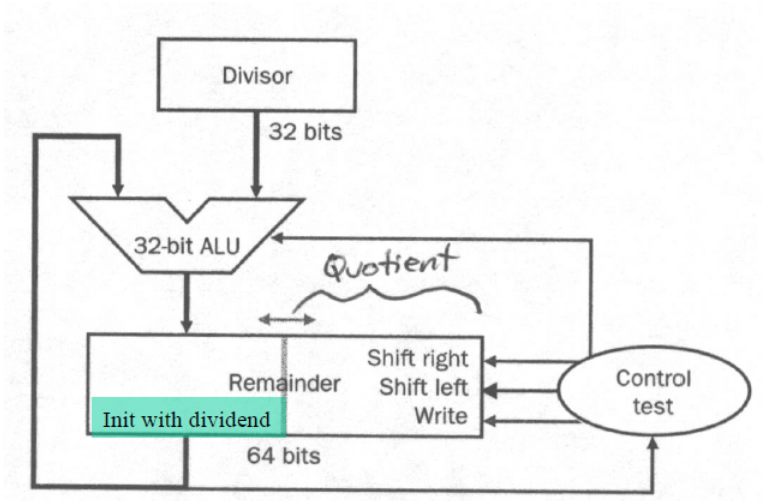


| Iteration | Step | Quotient | Divisor | Remainder |
|---|---|---|---|---|
| 0 | Initial values | 0000 | 0010 0000 | 0000 0111 |
| 1 | 1: Rem = Rem – Div | 0000 | 0010 0000 | 1110 0111 |
| | 2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0 | 0000 | 0010 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0001 0000 | 0000 0111 |
| 2 | 1: Rem = Rem – Div | 0000 | 0001 0000 | 1111 0111 |
| | 2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0 | 0000 | 0001 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 1000 | 0000 0111 |
| 3 | 1: Rem = Rem – Div | 0000 | 0000 1000 | 1111 1111 |
| | 2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0 | 0000 | 0000 1000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 0100 | 0000 0111 |
| 4 | 1: Rem = Rem – Div | 0000 | 0000 0100 | 0000 0011 |
| | 2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1 | 0001 | 0000 0100 | 0000 0011 |
| | 3: Shift Div right | 0001 | 0000 0010 | 0000 0011 |
| 5 | 1: Rem = Rem – Div | 0001 | 0000 0010 | 0000 0001 |
| | 2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1 | 0011 | 0000 0010 | 0000 0001 |
| | 3: Shift Div right | 0011 | 0000 0001 | 0000 0001 |




### Optimization

- Shift remainder left instead of divisor right, make divisor reg 32 bits
- Remove quotient register, put in right half of remainder
- 1 cannot be first bit in quotient, shift remainder left by 1 to do one less iteration

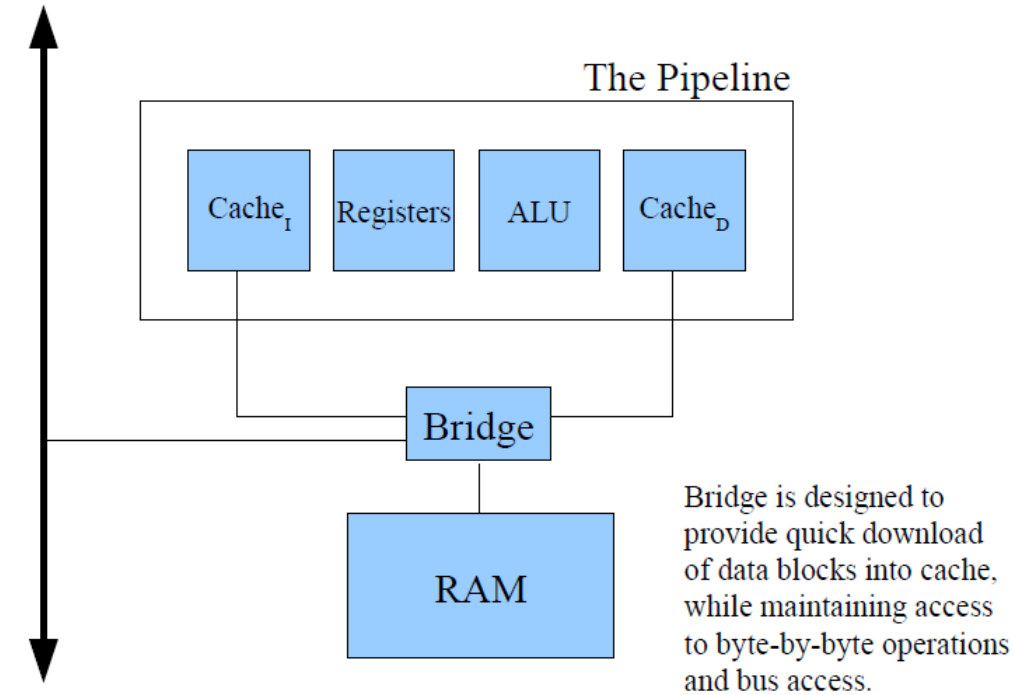



| Iteration | Step | Divisor | Remainder |
|---|---|---|---|
| 0 | Initial values | 0010 | 0000 0111 |
| | Shift Rem left 1 | 0010 | 0000 1110 |
| 1 | 2: Rem = Rem – Div | 0010 | 1110 1110 |
| | 3b: Rem < 0 ⇒ + Div, sll R, R0 = 0 | 0010 | 0001 1100 |
| 2 | 2: Rem = Rem – Div | 0010 | 1111 1100 |
| | 3b: Rem < 0 ⇒ + Div, sll R, R0 = 0 | 0010 | 0011 1000 |
| 3 | 2: Rem = Rem – Div | 0010 | 0001 1000 |
| | 3a: Rem ≥ 0 ⇒ sll R, R0 = 1 | 0010 | 0011 0001 |
| 4 | 2: Rem = Rem – Div | 0010 | 0001 0001 |
| | 3a: Rem ≥ 0 ⇒ sll R, R0 = 1 | 0010 | 0010 0011 |
| | Shift left half of Rem right 1 | 0010 | 0001 0011 |

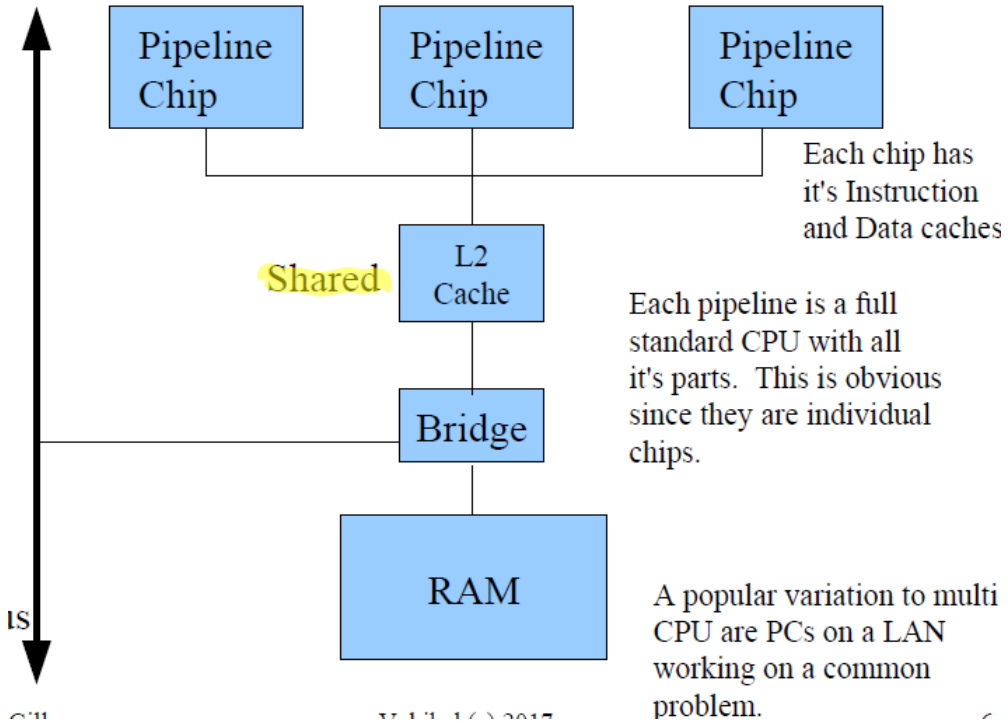
## 9.4 Multiprocessing

### Types of computers

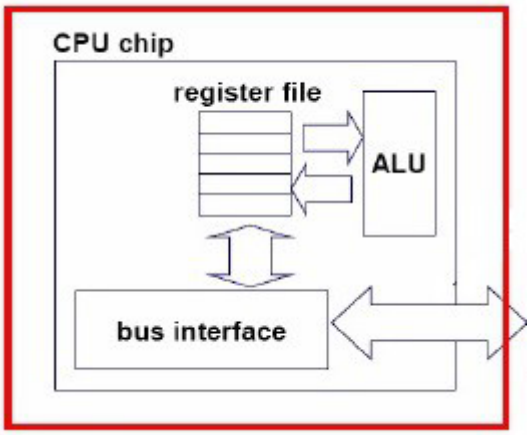
- Single CPU, one CPU chip, any type (flat, pipeline, hyper threaded)



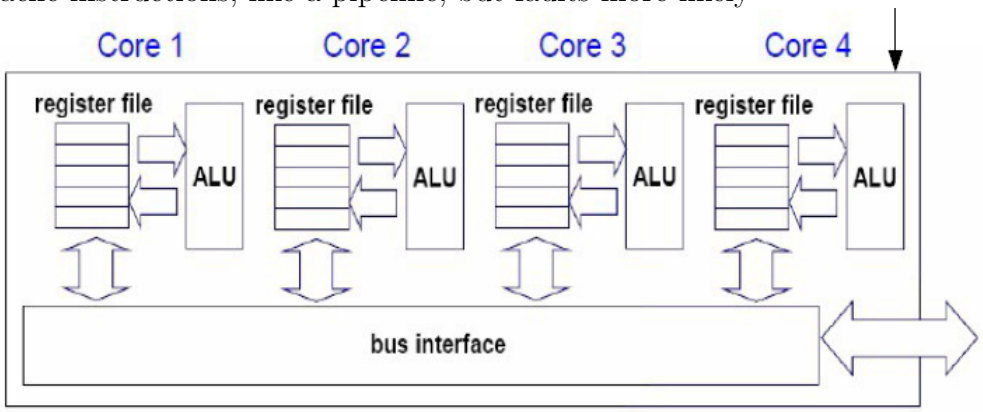

- Multi-CPU, more than one CPU chip, any type




- Single Core, like single CPU chip, but optimized. Has a bridge (connected to buses) to connect core with rest of system board




- Multi-Core, single cpu chip with multiple processors, aka Chip Multi-Processor (CMP).
  Bus interface shared (unlike Multi-CPU), to have same data
  Cache instructions, like a pipeline, but faults more likely




**Programs**

- Program is a compiled algorithm stored on disk, static
- Process is executing version of program in RAM, static & dynamic
- Thread are instructions currently being run
  - Process can have many threads
  - Thread is an instance of process

For multi-core, threads run in parallel, once for each core. Can have hyperthreading and have queues in each core. Fork a program to split across multiple cores. The more cores, the more the advantages get smaller since they all need to communicate with each other. Multicore because we can't make things go faster than speed of light, so we need more of them.

**OS Core Management** OS treats each core as a unique processor. OS schedules processes and threads to diff cores

- Special purpose core assignment: OS reserves cores for types of apps
- Dedicated application assignment: Application attached to core, always done on that core. Shareable with other processes
- Core pooled assignment: Queue manages all proceses, process at front gets any available core
- Simple Management
  - Queue of processes to execute
  - Each core assigned to waiting process
  - After n microsec, release core and loop
- Complex Management
  - One process has multiple threads in queue
  - Need to share data, put on a core that can share
  - Hardware for data access conflicts

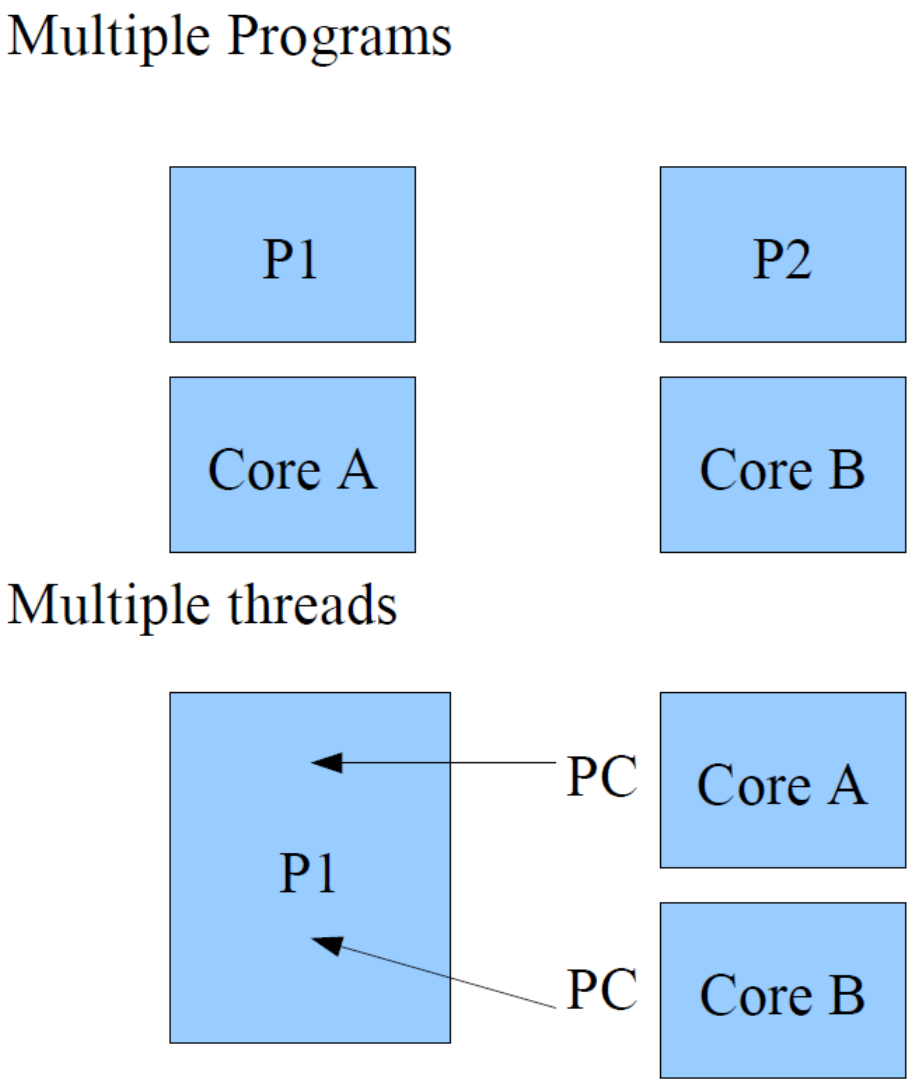
**Problem**

- Uniqueness of threads
  - 1 thread per program? Programs all execute at the same time
  - N threads per program? (like a browser with multiple windows)
    - They have to coordinate with each other, overhead to manage
    - $N > 6$, overhead starts to be more than speed improvement

**Parallelism**

- Single core, instruction level parallelism $\rightarrow$ pipeline
- Thread-level parallelism (TLP), multi-core
  - Each core gets a part of program to execute

– Single cores cannot take advantage of TLP, future will involve multi-cores exploiting TLP




**Single Instruction Multiple Data (SIMD)**

- Ex. Modern GPU
- Matrix of data
- 1 instruction executed on all cells at same time
- One cell is a core

### Multiple Instruction Multiple Data (MIMD)

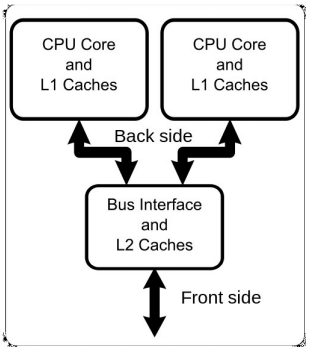
- Multi pipeline CPU
- 1 pipeline per core
- Shared RAM
- Each core has own caches
- Core itself is SIMD
- Chip as a whole is MIMD

### Simulataneous multi-threading (SMT)

- Multiple independent threads executing at same time on same core
- Can do integer operation on one thread and floating point on another (2 ALUs)
- Not really parallel, 30% threading, virtual processor, does not have resources for each core like Multi-core
- Can have multi-cores with SMT
- Intel calls SMT hyper-threads

### Memory Types

- Shared memory, shared between all processors
- Distributed memory, each processor has own memory
- Lots of computers have both



### Private vs Shared Cache

- Private is closer to core, faster access, less contention
- Shared, different cores can access the same thing, more space for big stuff

### Problems

- Contention: More than one core needs access to same cache, one has to wait, need circuitry to handle
- Cache coherence: Private caches, need data to be consistent, each core should perceive memory as shared
  - Core 1 changes a value, core 2 reads the old value
  - Solution: Inter-core bus, looks at MAR, addresses being used, issues faults (invalidation + snooping)
    - Core writes to cache, then all other copies → invalid
    - All cores snoop the bus connecting cores for write ops

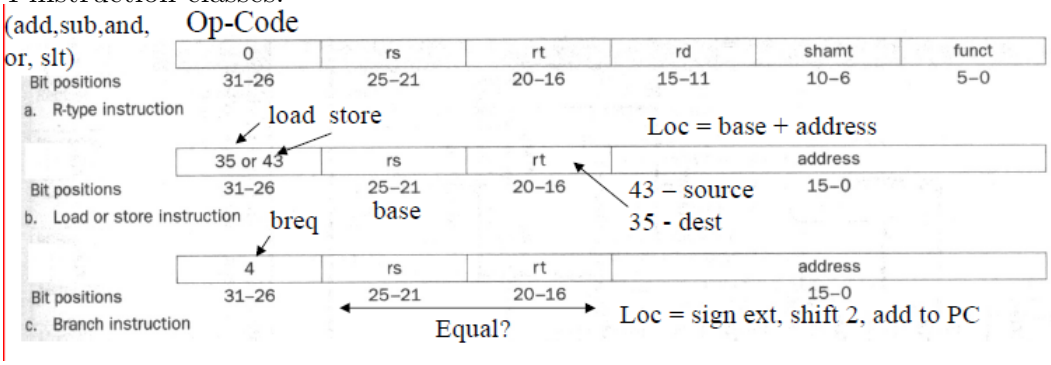
### Memory Hierarchy

- Not shared: Pipeline, L1 private
- Shared: L2 (sometimes private), L3, RAM
- Special: SMT, all caches shared

# 10 MIPS

## 10.1 Instructions

4 instruction classes:

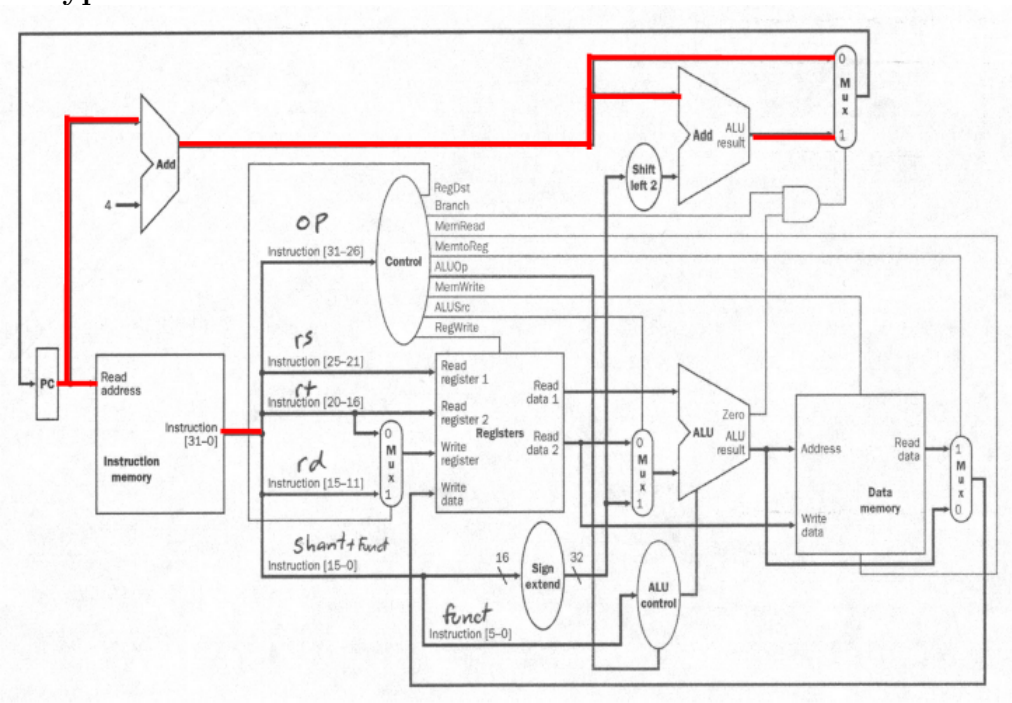


### Effects on CU

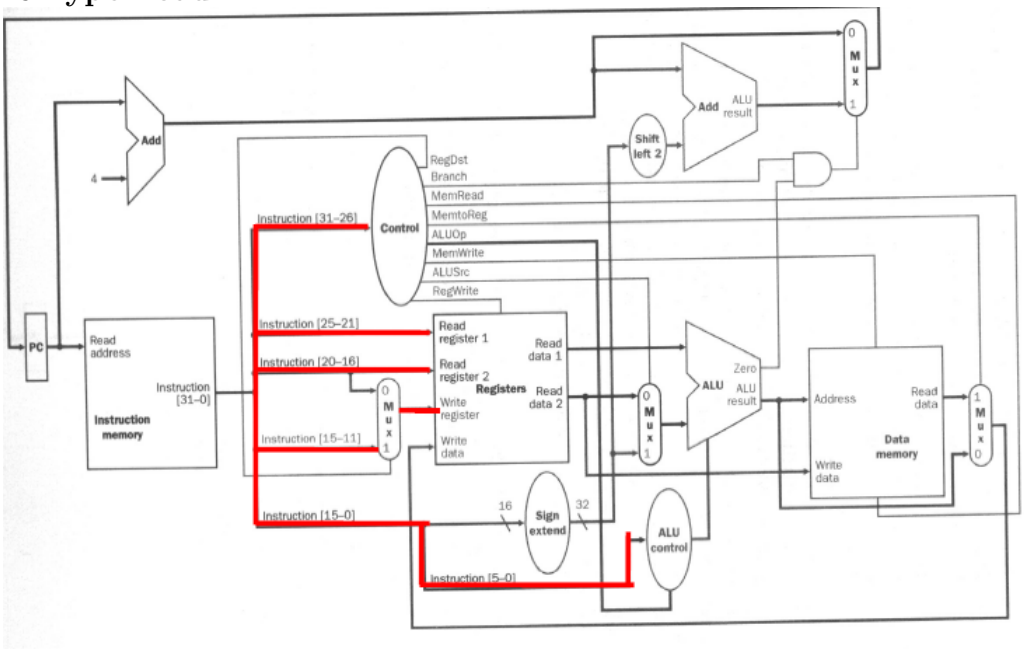
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp1 | ALUOp0 |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

Different pathways for different types of instructions.

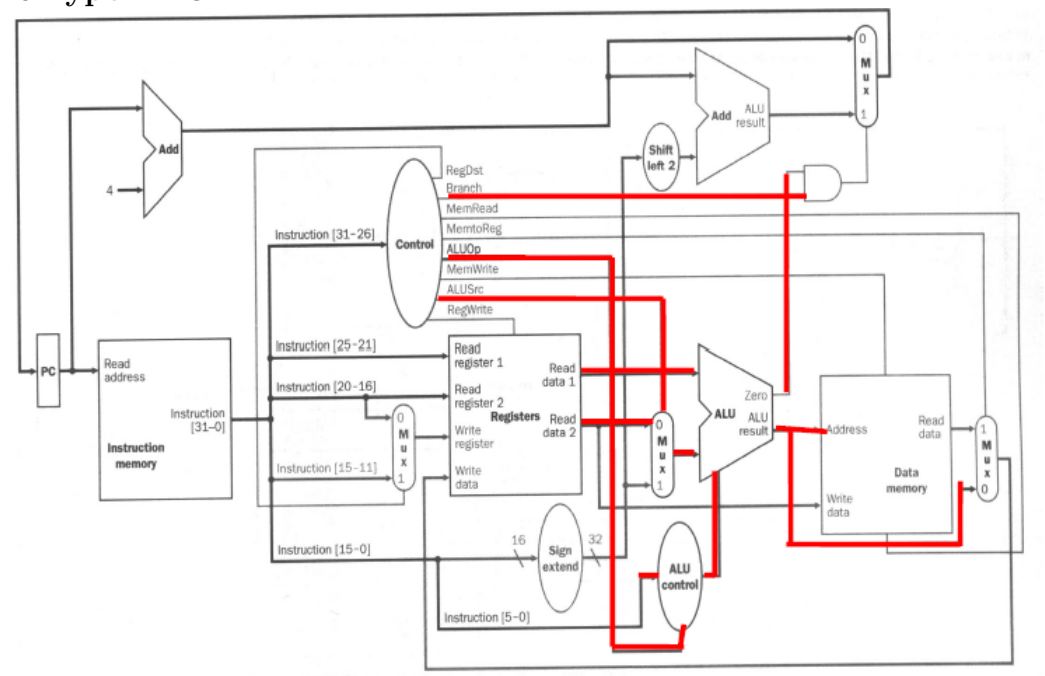
### R-Type Fetch



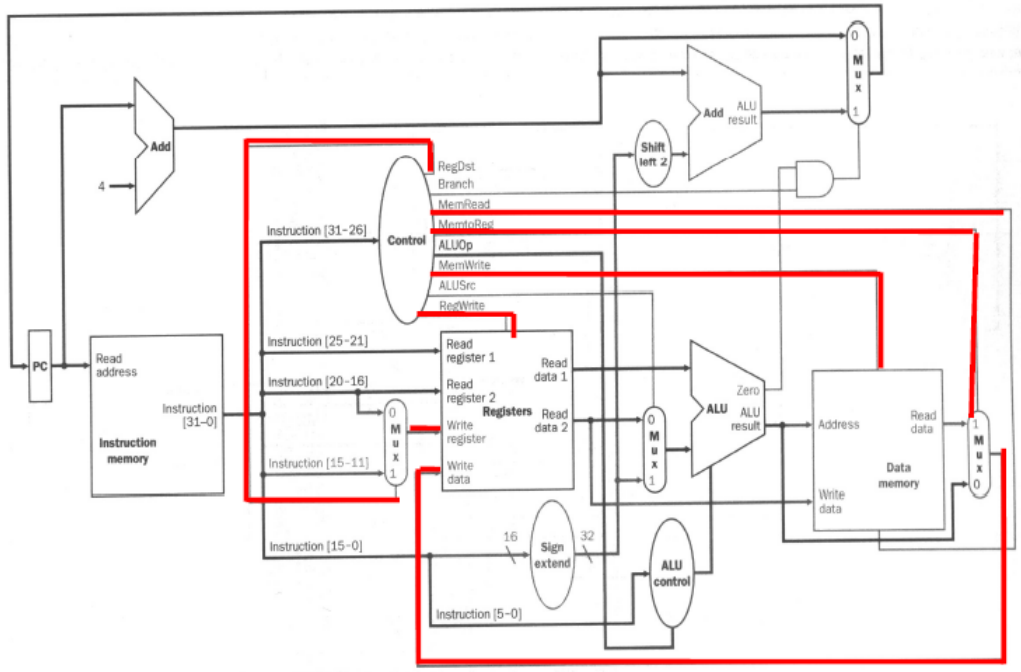
### R-Type Load




### R-Type ALU



### R-Type Store


Add
4
PC
Read address
Instruction [31-0]
Instruction memory
Instruction [31-26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25-21]
Instruction [20-16]
Instruction [15-11]
Instruction [15-0]
Instruction [5-0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
ALU result
Zero
ALU
ALU result
16
Sign extend
32
ALU control
Address
Read data
Data memory
Write data
M u x

**Instruction Table (given on final)**

| Category | Instruction | Example | | Meaning | Comments |
|---|---|---|---|---|---|
| Arithmetic | add | add | $s1,$s2,$s3 | $s1 = $s2 + $s3 | Three operands; overflow detected |
| | subtract | sub | $s1,$s2,$s3 | $s1 = $s2 – $s3 | Three operands; overflow detected |
| | add immediate | addi | $s1,$s2,100 | $s1 = $s2 + 100 | + constant; overflow detected |
| | add unsigned | addu | $s1,$s2,$s3 | $s1 = $s2 + $s3 | Three operands; overflow undetected |
| | subtract unsigned | subu | $s1,$s2,$s3 | $s1 = $s2 – $s3 | Three operands; overflow undetected |
| | add immediate unsigned | addiu | $s1,$s2,100 | $s1 = $s2 + 100 | + constant; overflow undetected |
| | move from coprocessor register | mfc0 | $s1,$epc | $s1 = $epc | Used to copy Exception PC plus other special registers |
| | multiply | mult | $s2,$s3 | Hi, Lo = $s2 × $s3 | 64-bit signed product in Hi, Lo |
| | multiply unsigned | multu | $s2,$s3 | Hi, Lo = $s2 × $s3 | 64-bit unsigned product in Hi, Lo |
| | divide | div | $s2,$s3 | Lo = $s2 / $s3, Hi = $s2 mod $s3 | Lo = quotient, Hi = remainder |
| | divide unsigned | divu | $s2,$s3 | Lo = $s2 / $s3, Hi = $s2 mod $s3 | Unsigned quotient and remainder |
| | move from Hi | mfhi | $s1 | $s1 = Hi | Used to get copy of Hi |
| | move from Lo | mflo | $s1 | $s1 = Lo | Used to get copy of Lo |
| Logical | and | and | $s1,$s2,$s3 | $s1 = $s2 & $s3 | Three reg. operands; logical AND |
| | or | or | $s1,$s2,$s3 | $s1 = $s2 \| $s3 | Three reg. operands; logical OR |
| | and immediate | andi | $s1,$s2,100 | $s1 = $s2 & 100 | Logical AND reg, constant |
| | or immediate | ori | $s1,$s2,100 | $s1 = $s2 \| 100 | Logical OR reg, constant |
| | shift left logical | sll | $s1,$s2,10 | $s1 = $s2 << 10 | Shift left by constant |
| | shift right logical | srl | $s1,$s2,10 | $s1 = $s2 >> 10 | Shift right by constant |
| Data transfer | load word | lw | $s1,100($s2) | $s1 = Memory[$s2+100] | Word from memory to register |
| | store word | sw | $s1,100($s2) | Memory[$s2 + 100] = $s1 | Word from register to memory |
| | load byte unsigned | lbu | $s1,100($s2) | $s1 = Memory[$s2 + 100] | Byte from memory to register |
| | store byte | sb | $s1,100($s2) | Memory[$s2 + 100] = $s1 | Byte from register to memory |
| | load upper immediate | lui | $s1,100 | $s1 = 100 * $2^{16}$ | Loads constant in upper 16 bits |
| Conditional branch | branch on equal | beq | $s1,$s2,25 | if ($s1 == $s2) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | bne | $s1,$s2,25 | if ($s1 != $s2) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | slt | $s1,$s2,$s3 | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than; two's complement |
| | set less than immediate | slti | $s1,$s2,100 | if ($s2 < 100) $s1 = 1; else $s1=0 | Compare < constant; two's complement |
| | set less than unsigned | sltu | $s1,$s2,$s3 | if ($s2 < $s3) $s1 = 1; else $s1=0 | Compare less than; natural numbers |
| | set less than immediate unsigned | sltiu | $s1,$s2,100 | if ($s2 < 100) $s1 = 1; else $s1 = 0 | Compare < constant; natural numbers |
| Unconditional jump | jump | j | 2500 | go to 10000 | Jump to target address |
| | jump register | jr | $ra | go to $ra | For switch, procedure return |
| | jump and link | jal | 2500 | $ra = PC + 4; go to 10000 | For procedure call |

## 10.2 Assembler Code

**Scope** All peripherals and system components accessible by instructions. Can access hardware through RAM (zero page), but you need to know address, binary format, command, status & data registers. Access to hardware requires privileged instructions.

- Privileged mode allows all registers protecting OS to be accessible (like boundary registers)
- Non-privileged is the opposite, used in this class

## 10.3 Compiling High Level Languages

Code → Compiler→ Assembler → linker (with library and loader) → executable

- Compiler checks syntax, coverts to assembly
- Assembler checks for syntax, converts to machine code
- Linker verifies library calls exist, merges library into program, adds special OS functions , determines memory locations that code will occupy, resolves references, ensures no undefined labels
- Loader & Loading
  - Loading: Puts program into RAM, notifies OS
  - OS responsible to pass CPU to program later
  - Loader: subroutine, knows how to put program into RAM and notify OS
    - Three types of OS
    - Loader function to be inserted as first instruction of program, done by linker
    - Built-in function, require program to call it
    - Require data record for parameters and do the whole process internally, no loader

We need the outputs and versions to match each other for the program to execute/work. Assembly program consists of the above, without a compiler, going directly from source code to assembler

## 10.4 Assembler

Converts assembler text file into .o file, has instructions and directives

**Directives**

- Comments indicated by #
- .text indicates code is after this
- .globl name of LABEL → where is main program
- .data indicates things stored and whatnot
- LABELS: define label's location, LABEL references its location

```
.align n                # Aligns data on specified size, 0=byte, 1=half, 2=wo
.ascii str              # Ascii String without '\0', put in quotes
.asciiz str             # Null terminating ascii String
.space n                # Reserve n Bytes
.byte b1,b2,...,bn      # Specify contents of a Byte (8 bits)
.half h1,..., hn        # 16 bits
.word w1,...,wn         # 32 bits
.float f1,...,fn        # 32
.double d1,...,dn       # 64
```

**Two-Pass Assembly** Building to linker:

- Pass 1: builds symbol table, indentifies labels, determines offset addresses of identifiers
  - Scans source, top to bottom
  - Count number of bytes in instructions passed to see how many bytes long the program is
  - Find a label, add to symbol table with offset from beginning of program
  - First instruction/label receives a base address in the symbol table, all others are just offsets from that address

| LABEL | OFFSET |
|---|---|
| main | x |
| int a | +3 (offset from x) |

- Pass 2: builds machine language program, 1-to-1 map
  - Scan source, top to bot again
  - One line of assembly is one line of machine, convert op-code & args
  - Assembly Template, just like Symbol Table, but has addresses (with offsets) and instructions in binary

**Unix Loading Algorithm**

- Read file header, determine CODE and DATA size req
- Find space in RAM for prog, set BASE addr
- Copy instructions and data into RAM

- Copy args onto run-time stack
- Assign SP to next free stack position, IP to first instr
- Clear registers

**Assembler Functionality** Assembler has functionality for:

- Math and Logic
- Data movement
- Pointer
- Branching
- Goto and subroutines

But does not have

- Functions (can be simulated)
- Objects (can be sim)
- Arrays or data structures (can be sim)
- Variables or constants in the normal way

Advanced things assembler can do:

- Access internal OS routines
- Access control ROMS of every peripheral

Assembler keeps it simple and modular, need to know how to use registers and stack.

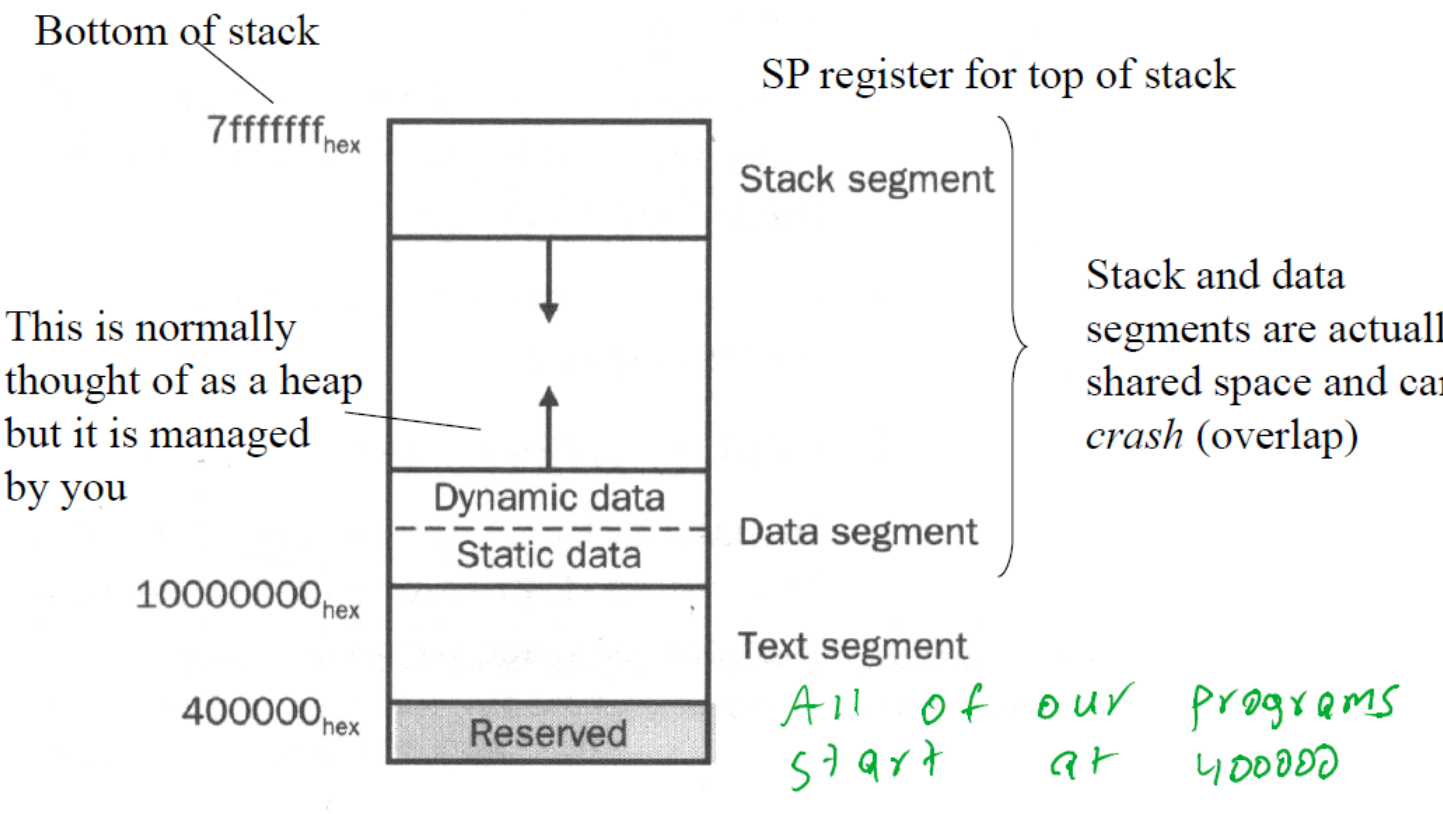
**Processors & RAM**

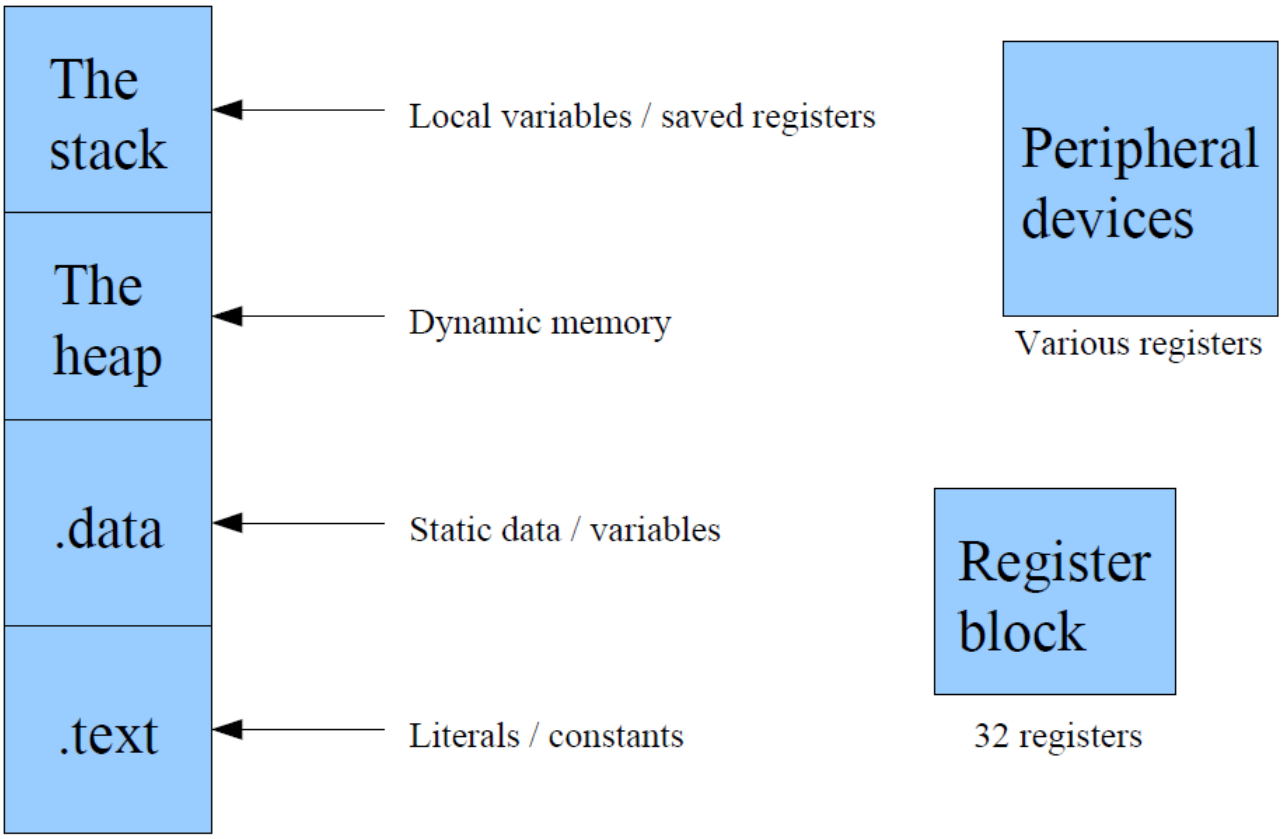
- Co-0 for exceptions and interrupts
- Co-1 for floating point ops
- 0xffff0000 to 0xffff000c for I/O ports (like keyboard and display)
- Syscall 1 to 10 for OS API

## 10.5 Virtual Memory

Mips has $2^{30}$ memory words, accessed by data transfer instructions

- Words are 4 Bytes
- Uses byte addresses
- Memory has data structures, arrays, saved registers, instructions, variables and constants


Bottom of stack
SP register for top of stack
7fffffff hex
Stack segment
This is normally thought of as a heap but it is managed by you
Stack and data segments are actuall shared space and ca crash (overlap)
Dynamic data
Static data
Data segment
10000000 hex
Text segment
400000 hex
Reserved
All of our programs start at 400000


The stack
Local variables / saved registers
The heap
Dynamic memory
.data
Static data / variables
.text
Literals / constants
Peripheral devices
Various registers
Register block
32 registers

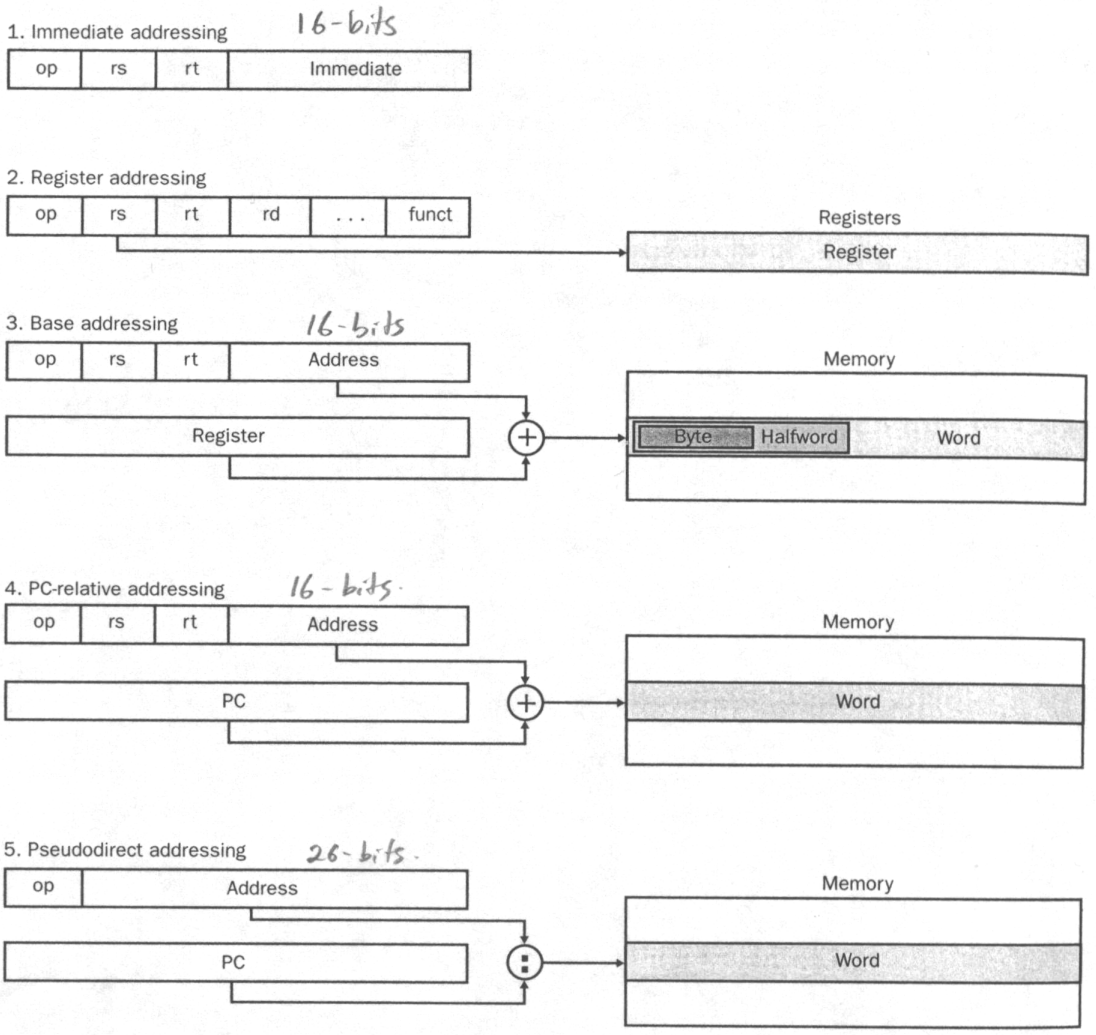
Heap is for things like allocating memory, stack is for local vars of functions, stack heap collisions happen if we have infinite recursion or ask for too much memory.

### MIPS Registers

### Addressing Modes

- Register addressing, operand is a register, add $s1, $s2, $s3
- Base or displacement addressing, operand is memory location, register+ offset, lw $s1, 100($s2)
- Immediate Addressing, operand is constant (16 bit), addi $s1, $s2, 100
- PC-Relative Addressing mem loc=PC+offset, j 2500
- Pseudo-Direct Addressing, mem loc = top 6 bits of PC with 26 bit offset, jal 2500

There are limitations to how far you can jump because of the size of an instruction.




**Buffer** Temporary location for for data during a move, could be in RAM.

- Using a buffer?
- Allocate space using syscall 9 or in .data
- Load the address of the label
- Store things

**Accessing OS Memory** Using syscalls for controlled access. Put the system call code in $v0 and then execute a syscall for the following:

| Service | System call code | Arguments | Result |
|---|---|---|---|
| print_int | 1 | $a0 = integer | |
| print_float | 2 | $f12 = float | |
| print_double | 3 | $f12 = double | |
| print_string | 4 | $a0 = string | |
| read_int | 5 | | integer (in $v0) |
| read_float | 6 | | float (in $f0) |
| read_double | 7 | | double (in $f0) |
| read_string | 8 | $a0 = buffer, $a1 = length | |
| sbrk | 9 | $a0 = amount | address (in $v0) |
| exit | 10 | | |

## 10.6 Code Format

```
        .text
        .globl main
main:   la $a0, MESG    # Loads address of String into $a0
        li $vo, 4       # Load 4 into $v0 for a syscall to print
        syscall         # Prints String pointed to by $a0

        li $vo, 10      # Code to exit
        syscall         # Exits

        .data
MESG:   .asciiz"Hello world!\n"
```

## 10.7 Calling Subroutines

**Systems Programming/Register Convention** Pass parameters through $a0-$a3, return with $v0, $v1.

- Benefits: fast and easy to code.
- Drawback: limited number of registers, no local vars.

**ANSI C Standard** Use the runtime stack, more work, get local vars
Return using either the stack of $v0, $v1, pass params through stack and push local vars to stack. Backup at beginning of functional call and restore register at the end.
Stack pointer goes “down”, subtract to advance it

- Benefits: stack very large, simulates local vars
- Drawback: slows down program, extra lines of code and moves

```
# Pushing
subi $sp,$sp,8  # Subtract 8 bytes from sp to reserve 8 bytes of space
sw   $s0, 4($sp)# Save $s0 at an offset of 4 from sp
sw   $s1, ($sp) # Save $s1 at sp

# Popping
lw $s0, 4($sp)  # Restores $s0
lw $s1, ($sp)
addi $sp,$sp,8  # Pseudo deletion of stack vars, still "there"
```

**Functions**

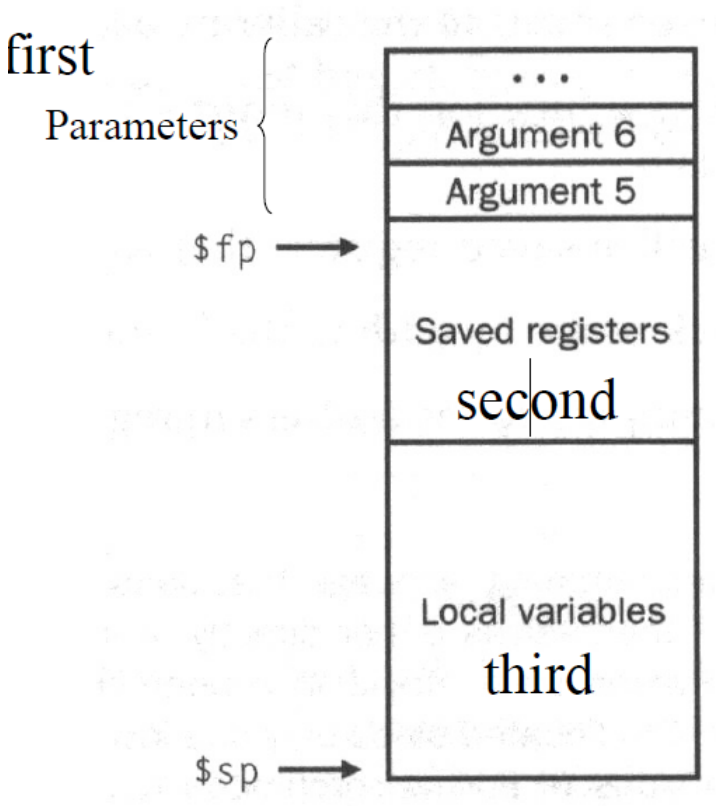
- Calling: JAL, pass params, stack or use $a registers
- Protecting: Stack registers to protect calling env
  - We usually don't protect $t registers, but we do protect $s registers, others optional
- Returning: restore calling env, return value by register, pointer or stack and jr $ra

**Memory**

- Protection
- Register, subroutine only uses register (CPU registers are considered temporary)
- Global, RAM and instructions lw, sw for subroutine memory
- Local, use the stack for memory
- Combination of above

**Stack pointer and frame pointer** Frame pointer stores upper bound of runtime stack, where you were before you decrement stack pointer. Move $sp to $fp before you decrement. Instead of adding to stack pointer to pop everything, can just move $fp into $sp. Don't let

things past $fp or $sp




**Global Pointer** $gp points to first byte of block of memory in .data/beginning of heap

**Recursion**

- Problematic, multiple calls
- Parameters for each call
- Local vars
- Only on $ra register

**Oversized variables** What do we do when data does not fit into a register? (Arrays, Structures, Objects) Use pointers.

## 10.8 Floating Point

Floating point instructions are different, uses co processor.

**MIPS floating-point operands**

| Name | Example | Comments |
|---|---|---|
| 32 floating-point registers | $f0, $f1, $f2, . . . , $f31 | MIPS floating-point registers are used in pairs for double precision numbers. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls. |

**MIPS floating-point assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | FP add single | add.s $f2,$f4,$f6 | $f2 = $f4 + $f6 | FP add (single precision) |
| | FP subtract single | sub.s $f2,$f4,$f6 | $f2 = $f4 – $f6 | FP sub (single precision) |
| | FP multiply single | mul.s $f2,$f4,$f6 | $f2 = $f4 × $f6 | FP. multiply (single precision) |
| | FP divide single | div.s $f2,$f4,$f6 | $f2 = $f4 / $f6 | FP divide (single precision) |
| | FP add double | add.d $f2,$f4,$f6 | $f2 = $f4 + $f6 | FP add (double precision) |
| | FP subtract double | sub.d $f2,$f4,$f6 | $f2 = $f4 – $f6 | FP sub (double precision) |
| | FP multiply double | mul.d $f2,$f4,$f6 | $f2 = $f4 × $f6 | FP multiply (double precision) |
| | FP divide double | div.d $f2,$f4,$f6 | $f2 = $f4 / $f6 | FP divide (double precision) |
| Data transfer | load word copr. 1 | lwc1 $f1,100($s2) | $f1 = Memory[$s2 + 100] | 32-bit data to FP register |
| | store word copr. 1 | swc1 $f1,100($s2) | Memory[$s2 + 100] = $f1 | 32-bit data to memory |
| Conditional branch | branch on FP true | bc1t 25 | if (cond == 1) go to PC + 4 + 100 | PC-relative branch if FP cond. |
| | branch on FP false | bc1f 25 | if (cond == 0) go to PC + 4 + 100 | PC-relative branch if not cond. |
| | FP compare single (eq,ne,lt,le,gt,ge) | c.lt.s $f2,$f4 | if ($f2 < $f4) cond = 1; else cond = 0 | FP compare less than single precision |
| | FP compare double (eq,ne,lt,le,gt,ge) | c.lt.d $f2,$f4 | if ($f2 < $f4) cond = 1; else cond = 0 | FP compare less than double precision |

$f0 and $f1 for returning

**Extra Instructions**

- FP absolute value double abs.d fd, fs
- FP absolute value single abs.s fd, fs
- FP addition double add.d fd, fs, ft
- FP addition single add.s fd, fs, ft
- Compare equal double c.eq.d fs, ft
- Compare less than c.le.d fs, ft
- Convert single to double cvt.d.s fd, fs
- Convert int to double cvt.d.w fd, fs
- FP divide double div.d fd, fs, ft

**Different Types** How to deal with different "types"

| C type | Data transfers | Operations |
|---|---|---|
| int | lw, sw, lui | add addi sub mult, div, and, andi, or, ori, slt, slti |
| unsigned int | lw, sw, lui | addu, addiu, subu, multu, divu, and, andi, or, ori, sltu, sltiu |
| char | lb, sb, lui | addu, addiu, subu, multu, divu, and, andi, or, ori, sltu, sltiu |
| bit field | lw, sw, lui | and, andi, or, ori, sll, srl |
| float | lwc1, swc1 | add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s |
| double | lwc1, swc1 | add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d |

# 11 I/O & Peripherals

**Peripherals** A peripheral is a device external to the system board. Used for sending and receiving data. Devices usually have their own data registers, status registers, command registers and rom, sometimes even a CLK or RAM. Usually accessed via zero page. 2 types of controllers:

- On board, registers integrated with system board
- External, registers part of a card plugged into slot

### Keyboard Registers

- Status:
  - 0: Key is being pressed
  - 1: Buffer empty
  - 2: Has power
  - 3: Shift being pressed

  - 4: Control being pressed

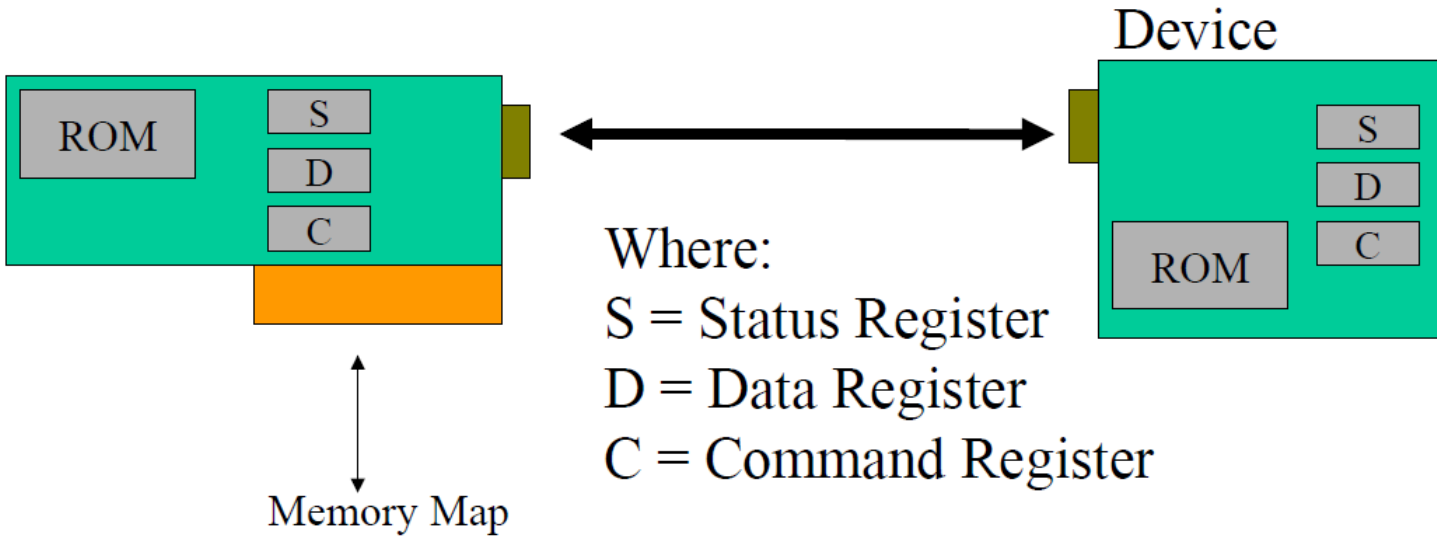
- Keyboard Data register stores ASCII of key pressed

## 11.1 I/O

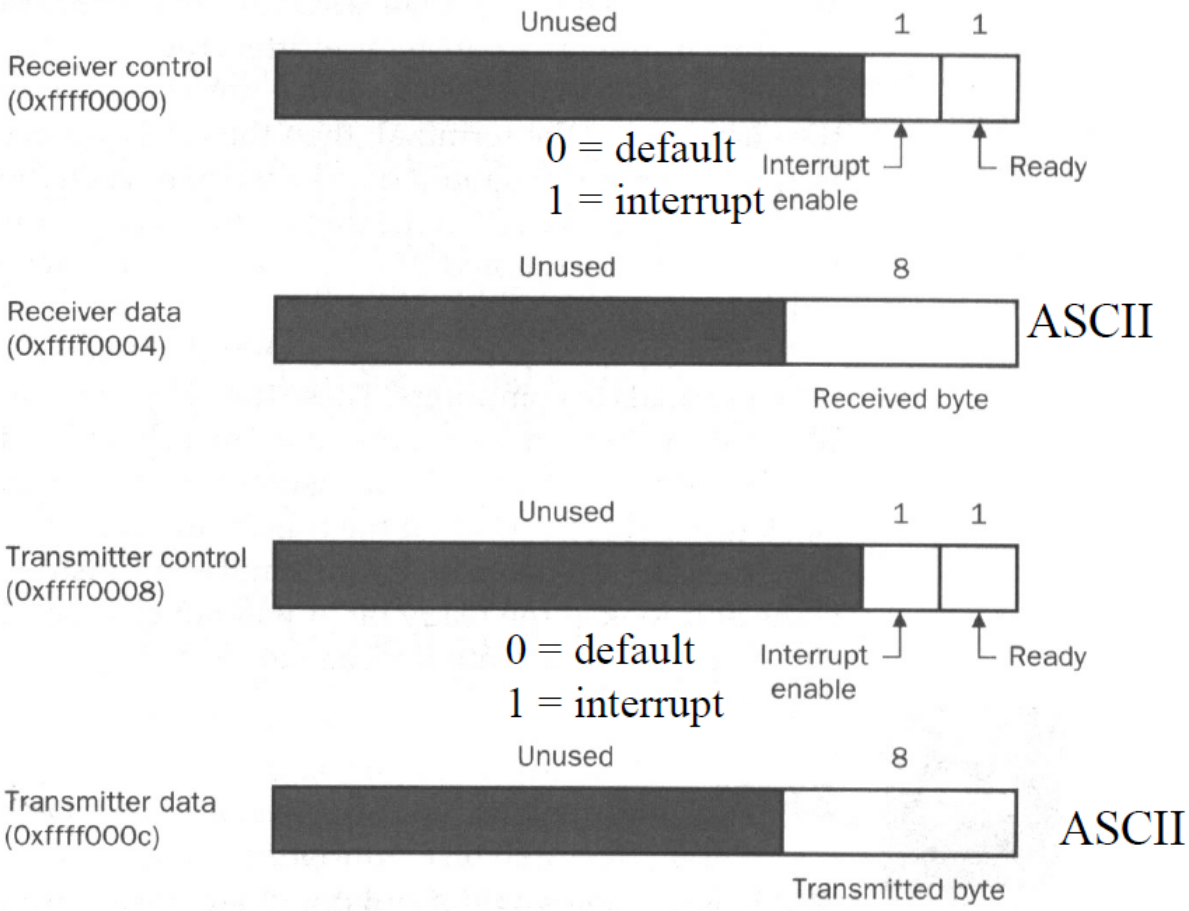
- Polling: keep checking status register until ready, inefficient
  - Busy loop, keeps checking until done
  - Internitent loop, polls on a timer, good if constantly taking input at a regular interval
- Interrupt: Do something else, let peripheral tell you when it's done. Requires special hardware. Sleeps current program, wakes up another until interrupted, then sleeps and wakes up, overhead is bigger.
  - Address register for address of the program so CPU knows where to go once the device is done
  - OS address interrupt register
  - Bus interrupt signal wire
  - Interrupt signal, event triggered by process to reroute CPU
  - Interrupt trap, event triggered by device to reroute CPU
  - Interrupt checks cause register to see what caused it and then deal with that case
  - Don't want to lose data (internet), use internet to send network card into a busy loop and let it interrupt you
- Direct Memory Interfacing (DMI/DMA), private bus, direct interface between program and RAM

Polling should be used when waiting for data that is required for the rest of the execution, or if there is only 1 process.

- Synchronous I/O, CPU monitors device, sends/reads 1 byte at a time (polling)
- Asynchronous I/O, CPU loads register with start address, another with limit, commands device to download or upload, does something else until device sends an interrup saying done



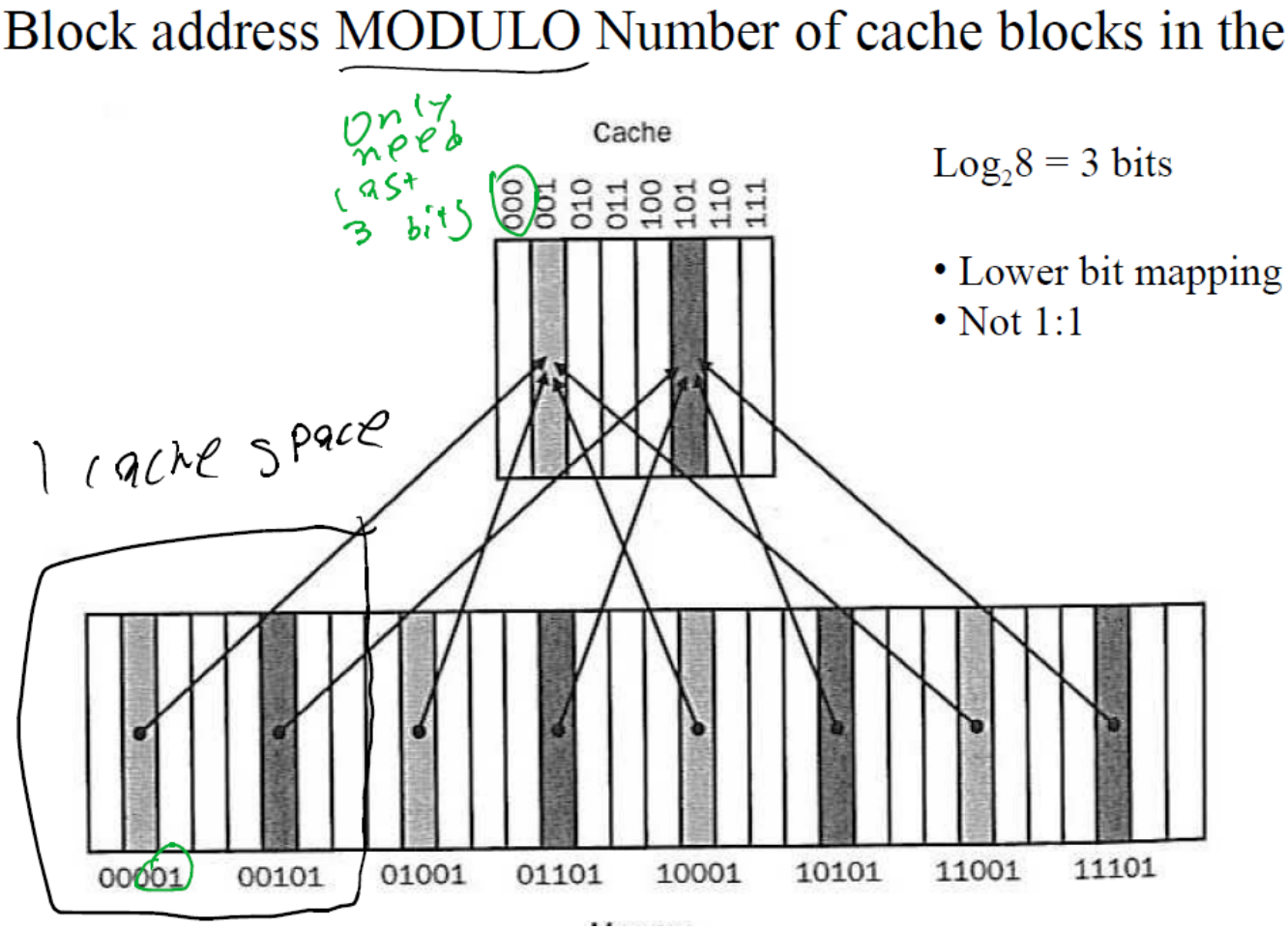

## 11.2 MIPS I/O




Polling, use andi with 1 to get if ready.

# 12 Cache

RAM clock is slow, can't fit RAM inside CPU. So we have a cache in the CPU, similar to RAM. We map RAM to cache, by using modulo. Naturally use the last few bits of the address and map to the cache. Problem: things start fighting in cache, overwriting each other. Cache is efficient if probability of collisions is low. Cache has different levels, primary, secondary and tertiary (usually RAM), getting bigger and slower as we distance from primary. 256 kb is a good block size.




**Locality**

- Temporal locality, item will be referenced again soon, library and Functions
- Spatial locality, item will probably be executed next, loops and functions

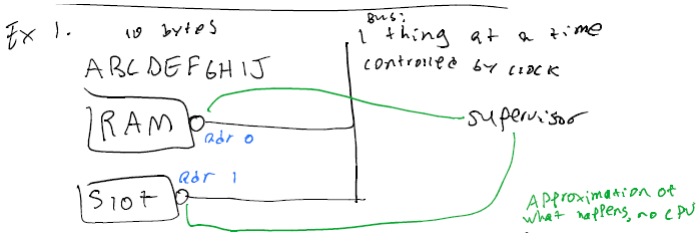
Cache takes advantage of locality, places code that will be repeatedly executed in part of cache. Can put two program's for loops, one at the top of cache, the other at the bottom.

But they will sometimes crash, how do we deal with that?
Speed vs amount of storage, work against each other in cost

**Issues** What do we load? Hit to miss ratio:

- Hit → finds instruction in cache
- Miss → need to go to RAM. Check if beginning of address (unique tag) is same as tag in cache, else miss
  - After a miss, need to access RAM, read, put data in cache, restart instruction from cache
  - miss penalty = cycles to upload data to cache
  - cost of miss = miss frequency × miss penalty
  - program speed = $n + m \times$penalty, $n$ and $n$ are number of instructions

# 13 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.

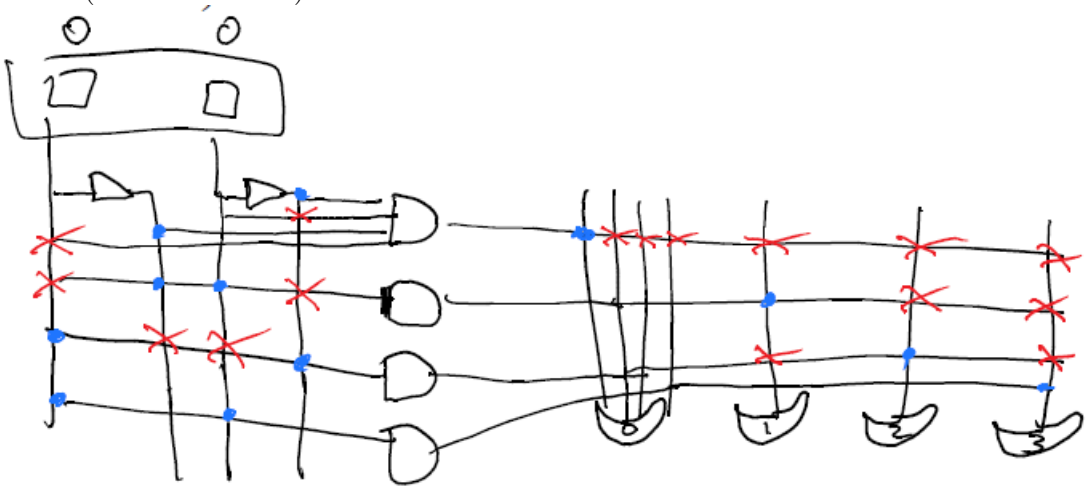
**How would the assumptions impact the design of the ALU?** Limit capacity to do arithmetic, depending on size of data.

**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.

2 **GHz computer** CPU that executes 1 instruction/20 ticks
How long to run linear program of 1000 lines of code?
$T = 1000 \times 20/2 \times 10^9 = 0.00001$ sec

**Comparing Pipeline & Classical** Given ratio of program instructions (how much % is loads, stores, etc.) and how long each step takes. Use: $time = instructions \times ticks/instr \times secs/ticks$

| Instruction class | Functional units used by the instruction class | | | | |
|---|---|---|---|---|---|
| ALU type | Instruction fetch | Register access | ALU | Register access | |
| Load word | Instruction fetch | Register access | ALU | Memory access | Register access |
| Store word | Instruction fetch | Register access | ALU | Memory access | |
| Branch | Instruction fetch | Register access | ALU | | |
| Jump | Instruction fetch | | | | |
| | 2 | 1 | 2 | 1 or 2 | 1 |

Like ADD

Memory access: 2 ns
ALU and adders: 2 ns
Register file (i/o): 1 ns

| Instruction class | Instruction memory | Register read | ALU operation | Data memory | Register write | Total |
|---|---|---|---|---|---|---|
| ALU type | 2 | 1 | 2 | 0 | 1 | 6 ns |
| Load word | 2 | 1 | 2 | 2 | 1 | 8 ns |
| Store word | 2 | 1 | 2 | 2 | | 7 ns |
| Branch | 2 | 1 | 2 | | | 5 ns |
| Jump | 2 | | | | | 2 ns |

Even here, where it's 2 for classical, still 2 for pipeline

Times for Classical

Variable vs. fixed cycle lengths?

Pipeline needs slowest, 8 ns
But only first guy needs 8
2 for remaining

ill Vybihal (c) 2015 38

# 14 Exercises

## 14.1 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

## 14.2 Assembler

How to create and use the following:

- Literals
- Static Variables
- Local variable
- Static data structures
- Local data structures
- Dynamic Memory
- Call by value vs call by reference