# COMP2611 Computer Organizations

# 1. Introduction

- Binary to Decimal:
  - The right-most digit represents $2^0$, to its left is $2^1$, $2^2$...
  - Sum up the powers of 2 if its corresponding digit is 1.
- Decimal to Binary:
  - Continue divide the decimal value by 2 **until the quotation is 0**, record the remainder. The binary value is equal to reading the remainder from the bottom to the top.
  - e.g. 34:
    - $34/2 = 17$ remain **0**
    - $17/2 = 8$ remain **1**
    - $8/2 = 4$ remain **0**
    - $4/2 = 2$ remain **0**
    - $2/2 = 1$ remain **0**
    - 1/2 = **0** remain **1**
    - Binary value: 100010.
- Decimal to Hexadecimal:
  - Continue divide the value by 16 until the quotation is 0. The hexadecimal is the remainder reading from bottom to top.
  - e.g. 280:
    - $280/16 = 17$ remain **8**
    - $17/16$ = 1 remain **1**
    - $1/16$ = **0** remain **1**
    - The hexadecimal value will be 118
- Size of **memory of file**:
  - 1 Kilo/Mega/Giga/Tera/Peta is $2^{10}/2^{20}/2^{30}/2^{40}/2^{50}$
    - e.g. 4GB = $4 * 2^{30}$ Bytes
- For **rate/frequency**:
  - 1 Kilo/Mega/Giga/Tera/Peta is $10^3/10^6/10^9/10^{12}/10^{15}$
- 1 Byte = 8 bits
  - e.g. GB

# 2. Digital Logic

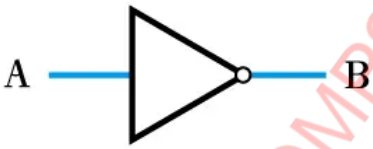
- **Analog**: Continuous values over a broad。/ range
- **Digital**: Only assumes discrete values (e.g. Computer)
  - 0~0.5V: Binary 0
  - 2.4~2.9V: Binary 1
  - Other: illegal voltage.

Analog Signal Digital Signal

- **Bits** are the basis for binary number representation in digital computers
- Truth Table:

| S | L |
|---|---|
| 0 | 0 |
| 1 | 1 |

  - input on left and output on right
  - Each row contains one possible configuration of the input variables.
  - Total combination of inputs given n variables: $2^n$
- **Most Significant Bit** (MSB): The bit on the leftmost, which affect the most
- **Least Significant Bit** (LSB): The bit on the rightmost.
- Boolean Algebra:
  - Consist of Boolean **values**, **variables** and **operations**
  - Values: True/False (0/1) only
  - Boolean variable: A variable that can take only 2 values, 0 or 1
  - Logic Operations: **AND, OR, NOT**, NAND, NOR, XOR
- Logic gates implement the above logic operations and built up by transistors.
- **NOT gate**:




(a) Circuit symbol

| A | B |
|---|---|
| 0 | 1 |
| 1 | 0 |

(b) Truth table

$$B = \overline{A}$$

(c) Boolean expression

- **AND gate**:


(a) Circuit symbol

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(b) Truth table

$$C = A \cdot B$$

(c) Boolean expression

- **OR gate**:


| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$$C = A + B$$

(a) Circuit symbol (b) Truth table (c) Boolean expression

- **NAND gate**:


| A | B | C |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$C = \overline{A \cdot B}$$

(a) Circuit symbol (b) Truth table (c) Boolean expression

  - Output true when both inputs are not 1.

- **NOR gate**:


| A | B | C |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

$$C = \overline{A + B}$$

(a) Circuit symbol (b) Truth table (c) Boolean expression

  - Output true if and only if both inputs are 0.

- **XOR gate** (Exclusive OR):


| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$C = A \oplus B$$

(a) Circuit symbol (b) Truth table (c) Boolean expression

  - Only output true when both inputs are different

- Any operation in digital circuit can be described by truth table or logic functions.

- Logic function: A function on binary variables whose outputs are also binary variables

# Combinational Logic Circuit

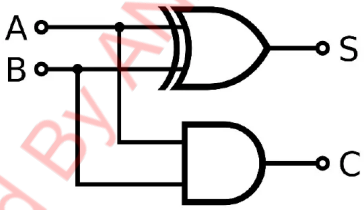
- It does not have memory
  - The output **depends only** on the current inputs and the circuit
- They **can be specified fully with a truth table or logic equation**
- 1 Bit half adder: It only takes 2 input and sum them up, give carry if needed
  - C: Carry on
  - S: Sum

| A | B | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Truth Table

$$S = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$$
$$C = A \cdot B$$

Logic Function




Circuit Implementation

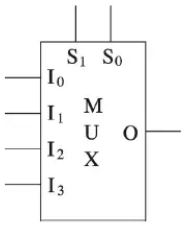
- **Multiplexor**: It selects one of the data inputs as output by a separate selection input (Job interview)
  - For a $2^n$-to-1 multiplexor:
    - It can take in $2^n$ inputs
      - If inputs not $2^n$, take n's ceiling (e.g. 9 to 1 needs 4 selection input)
    - Have n selection input
    - 1 Output
  - AND OR NOT operations only inside the multiplexor
  - e.g. 2:1, 4:1, 8:1, 16:1 MUXs.
  - 2:1 MUX
    - Logic Function: $\overline{S_1} \cdot I_0 + S_1 \cdot I_1$

| S1 | I0 | I1 | OUT |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

  - The truth table can be simplified into:

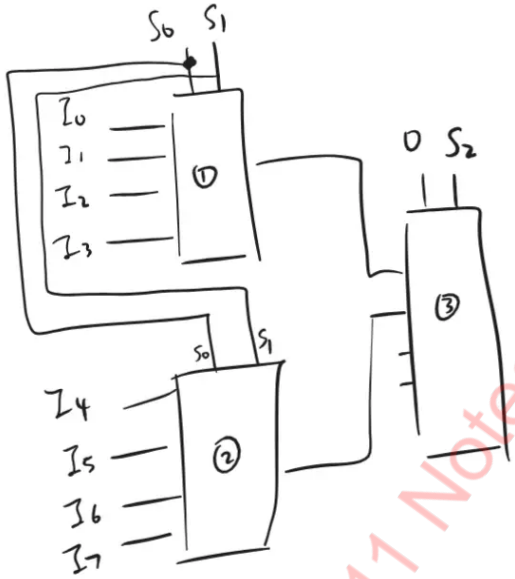
| S1 | OUT |
|---|---|
| 0 | I0 |
| 1 | I1 |

- e.g. 4:1 MUX:
  - Logic Function: $O = \overline{S_1} \cdot \overline{S_0} \cdot I_0 + \overline{S_1} \cdot S_0 \cdot I_1 + S_1 \cdot \overline{S_0} \cdot I_2 + S_1 \cdot S_0 \cdot I_3$




| $S_1$ | $S_0$ | O |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

  - If there are more inputs than the MUX can handle, it needs to have another MUX with some inputs left disconnected.

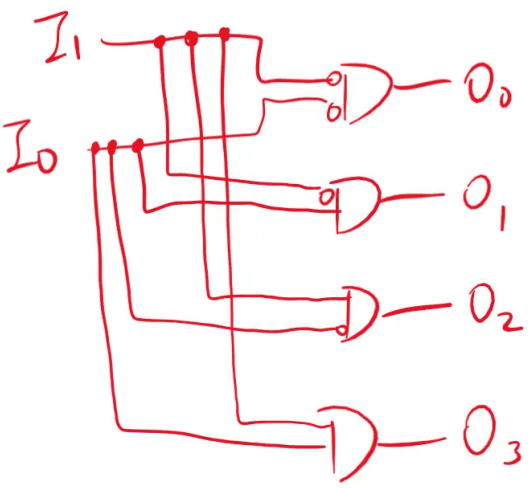


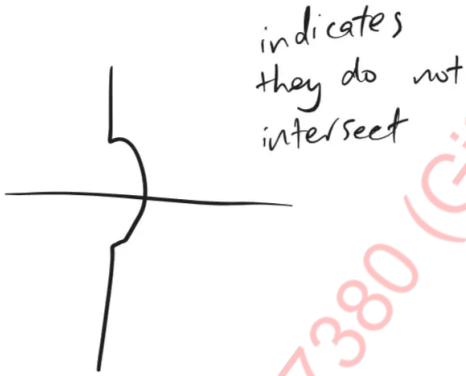

    - There are 8 inputs, so 3 selection lines are enough to decide the final output.
    - For the third MUX, one bit is enough to decide the final output, so one selection input can be unplugged.
- **Decoder** (Open gate by decoding the input):
  - n bit input
  - $2^n$ outputs, but only one of them will be true, others are false

| $I_1$ | $I_0$ | $O_3$ | $O_2$ | $O_1$ | $O_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

  - It convert the input back to integers and set the ith gate to true.
  - Logic Function: (The output are independent of each other)
    - $O_0 = \overline{I_1} \cdot \overline{I_0}$
    - $O_1 = \overline{I_1} \cdot I_0$

- $O_2 = I_1 \cdot \overline{I_0}$
- $O_3 = I_1 \cdot I_0$




- **3 Person Majority Vote**:
  - 3 inputs, if 2 or more is true, the output will be true

| Inputs | | | Output |
|---|---|---|---|
| X | Y | Z | D |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

  - Logic Function: $D = \overline{X} \cdot Y \cdot Z + X \cdot \overline{Y} \cdot Z + X \cdot Y \cdot \overline{Z} + X \cdot Y \cdot Z$
- **Minterm**: The *product* of Boolean variables
- **Maxterm**: The *sum* of Boolean variables

| Variable | | | Minterm | | Maxterm | |
|---|---|---|---|---|---|---|
| x | y | z | Term | Designation | Term | Designation |
| 0 | 0 | 0 | x'y'z' | $m_0$ | x+y+z | $M_0$ |
| 0 | 0 | 1 | x'y'z | $m_1$ | x+y+z' | $M_1$ |
| 0 | 1 | 0 | x'yz' | $m_2$ | x+y'+z | $M_2$ |
| 0 | 1 | 1 | x'yz | $m_3$ | x+y'+z' | $M_3$ |
| 1 | 0 | 0 | xy'z' | $m_4$ | x'+y+z | $M_4$ |
| 1 | 0 | 1 | xy'z | $m_5$ | x'+y+z' | $M_5$ |
| 1 | 1 | 0 | xyz' | $m_6$ | x'+y'+z | $M_6$ |
| 1 | 1 | 1 | xyz | $m_7$ | x'+y'+z' | $M_7$ |

  - Minterm **starts from all false**, **AND** statements connecting the relationship between the three input variables
  - Maxterm **starts from all true**, **OR** statements connecting them
- **Canonical Form** (2 Level Logic):
  - **Any boolean function can be expressed as a sum of minterm or a product of maxterm**

| Inputs | | | Output |
|---|---|---|---|
| A | B | C | D |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

$$\begin{aligned} \mathrm{D} &= m_3 + m_5 + m_6 + m_7 \\ &= \bar{\mathrm{A}}\mathrm{BC} + \mathrm{A}\bar{\mathrm{B}}\mathrm{C} + \mathrm{AB}\bar{\mathrm{C}} + \mathrm{ABC} \end{aligned}$$

$$\begin{aligned} \mathrm{D} &= M_0 M_1 M_2 M_4 \\ &= (A + \mathrm{B} + \mathrm{C})(A + \mathrm{B} + \bar{\mathrm{C}})(\mathrm{A} + \bar{\mathrm{B}} \\ &\quad + \mathrm{C})(\bar{A} + \mathrm{B} + \mathrm{C}) \end{aligned}$$

- If all cases that will fail does not happen, it is **Logically Equivalent** that one of the cases that will be true is happening.
- Any logic function can be expressed in canonical form as 2 level logic
  - One level consist of AND only
  - Another level consist of OR only
- **Sum of Product** (SOP) form:
  - e.g. $D = \overline{A}BC + A\overline{B}C + ABC$
  - More common
- **Product of Sum** (POS) form:
  - e.g. $D = (A + \overline{B} + \overline{C}) \cdot (\overline{A} + B + \overline{C}) \cdot (\overline{A} + \overline{B} + \overline{C})$
  - It is the reverse of the SOP examples with AND and OR switched between each other.

- Programmable Logic Array: An electronic device that can implement combinational logic circuits.

Show a PLA implementation of this example:

| Inputs | | | Outputs | | |
|---|---|---|---|---|---|
| A | B | C | D | E | F |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

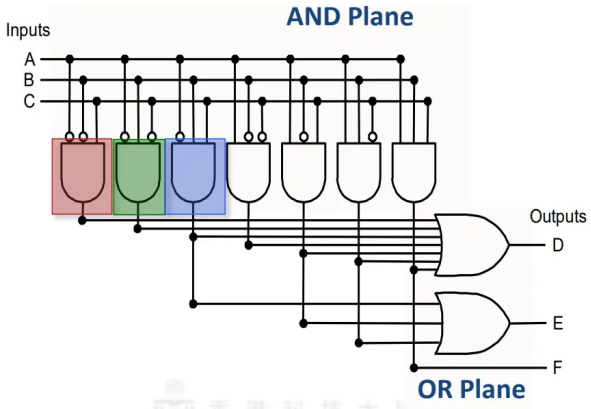
Sum-of-product representation

$$D = \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C$$

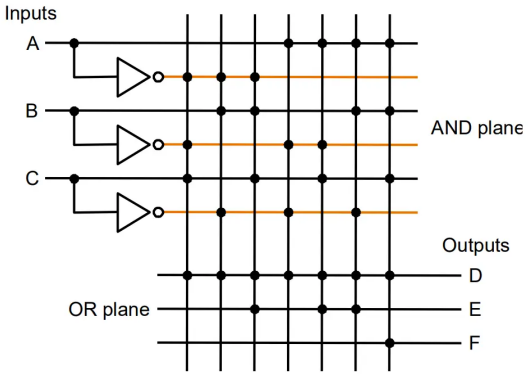
$$E = \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C}$$

$$F = A \cdot B \cdot C$$

- Implementing circuit diagram:




- Implementing PLA diagram:




  - This is SOP level implementation, each node represents one minterm connection, by reading downward in the AND plane then reading to the right at the OR plane, we can see the result at different output.
- Ways to identify logical equivalence:
  - Truth table (Good for small number of inputs)
  - Algebraic manipulation (with help of boolean algebra)
- Laws or Boolean Algebra:

| Name | AND form | OR form |
|---|---|---|
| Identity law | 1A = A | 0 + A = A |
| Null law | 0A = 0 | 1 + A = 1 |
| Idempotent law | AA = A | A + A = A |
| Inverse law | $A\overline{A} = 0$ | $A + \overline{A} = 1$ |
| Commutative law | AB = BA | A + B = B + A |
| Associative law | (AB)C = A(BC) | (A + B) + C = A + (B + C) |
| Distributive law | A + BC = (A + B)(A + C) | A(B + C) = AB + AC |
| Absorption law | A(A + B) = A | A + AB = A |
| De Morgan's law | $\overline{AB} = \overline{A} + \overline{B}$ | $\overline{A + B} = \overline{A}\,\overline{B}$ |

  - Be careful on De Morgan's Law and distributive law
  - Sometimes it is useful to duplicate some terms (Still logically equivalent) and it could be further simplified. For example:
- $O = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC = \overline{A}BC + A\overline{B}C + AB(C + \overline{C})$
$= ABC + \overline{A}BC + ABC + A\overline{B}C + AB = (A + \overline{A})BC + A(\overline{B} + B)C + AB$
$= BC + AC + AB$

- **K map**: A graphical representation of the truth table or logic function
  - Cells are arranged in **Gray Code**, which **a cell only differs by one bit between its neighbors**.

| A \ B | 0 | 1 |
|---|---|---|
| 0 | m0 | m1 |
| 1 | m2 | m3 |

2 inputs

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | m0 | m1 | m3 | m2 |
| 1 | m4 | m5 | m7 | m6 |

3 inputs

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | m0 | m1 | m3 | m2 |
| 01 | m4 | m5 | m7 | m6 |
| 11 | m12 | m13 | m15 | m14 |
| 10 | m8 | m9 | m11 | m10 |

4 inputs

  - If more than 4 inputs, it will need a 3D table to preform K mapping
- Rules of K map:
  - Find the largest size of circles displaying 1 with size $2^n$
    - The cells on the leftmost are connected to the cells at the rightmost (They can form a circle even they are on opposite sides) And the top cells are connected to the bottom cells.
  - If lots of circles formed, this means the resultant equation have less AND gates
  - If less circles formed, this means there are less AND gates as well as OR gates
- Examples

Simplify $F = A \cdot \overline{B} + A \cdot B + \overline{A} \cdot B$

| A \ B | 0 | 1 |
|---|---|---|
| 0 | $\overline{A} \cdot \overline{B}$ | $\overline{A} \cdot B$ |
| 1 | $A \cdot \overline{B}$ | $A \cdot B$ |

| A \ B | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

$F = A + B$

Simplify majority vote

$D = (\overline{A} \cdot B \cdot C) + (A \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot C) + (A \cdot B \cdot C)$

| A \ BC | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |

D = AB + AC + BC

- 7 Segment Display Example: (Only consider Segment G: The horizontal center part)

| Inputs | | | | Output | |
|---|---|---|---|---|---|
| $i_3$ | $i_2$ | $i_1$ | $i_0$ | G | A |
| 0 | 0 | 0 | 0 | 0 | |
| 0 | 0 | 0 | 1 | 0 | |
| 0 | 0 | 1 | 0 | 1 | |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 1 | |
| 0 | 1 | 0 | 1 | 1 | |
| 0 | 1 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 0 | 1 | |
| 1 | 0 | 0 | 1 | 1 | |
| 1 | 0 | 1 | 0 | 1 | |
| 1 | 0 | 1 | 1 | 1 | |
| 1 | 1 | 0 | 0 | 0 | |
| 1 | 1 | 0 | 1 | 1 | |
| 1 | 1 | 1 | 0 | 1 | |
| 1 | 1 | 1 | 1 | 1 | |

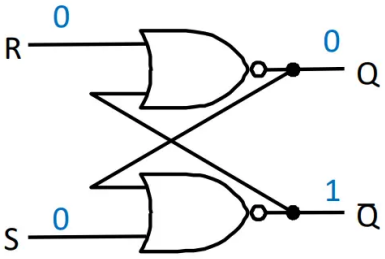
| $i_3i_2$ \ $i_1i_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| **00** | 0 | 0 | 1 | 1 |
| **01** | 1 | 1 | 0 | 1 |
| **11** | 0 | 1 | 1 | 1 |
| **10** | 1 | 1 | 1 | 1 |

$$G = i_1i_0' + i_3i_2' + i_3i_0 + i_2'i_1 + i_3'i_2i_1'$$

- Compare: Before: 12 AND with 4 input each + 1 OR gate with 12 inputs. Now: 4 AND gates with 2 inputs and 1 AND gate with 3 input + 1 OR gate with 5 inputs

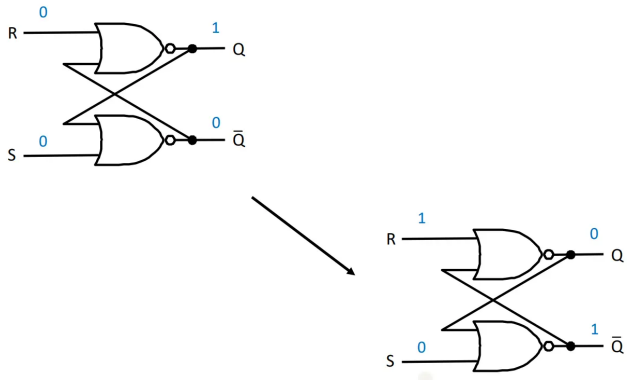
# Sequential Logic Circuits

- They are logic circuits that **have memory**
- The output will **depend on both** the current inputs and the values stored in memory (called state)
- Full adder: Similar function to half adder, but it also take carry on value from neighbor bit, that takes 3 inputs
- S-R latch
  - S is set
  - R is reset

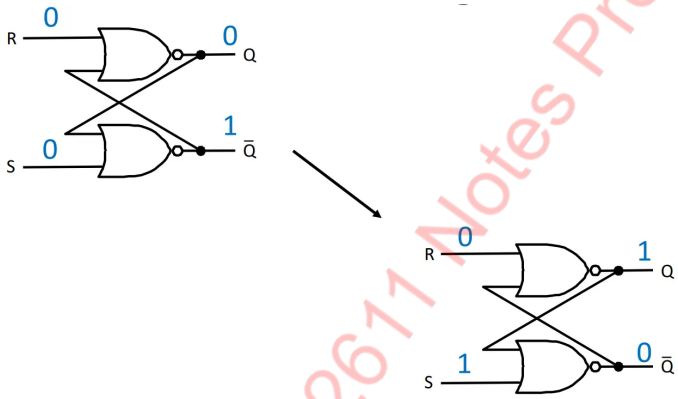



A S-R latch built by NOR gates
in quiescent state when previous value is 0

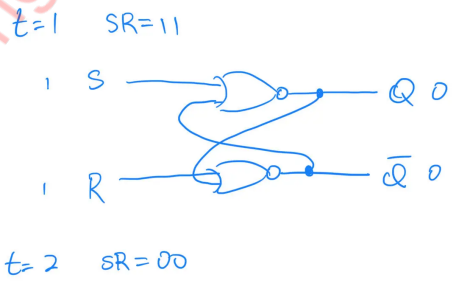
- 00: Latch state, hold current output (Both inputs are **de-asserted**)
- 01: Reset, set output Q to 0 (Reset is **in-effect** when input R is **asserted**)

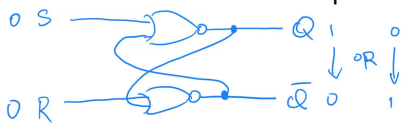



- 10: Set, set output Q to 1 (S is **asserted**)

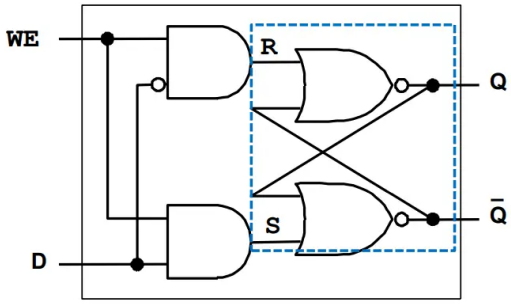



- 11: Invalid, forbidden state
  - Both inputs are 1 will cause both Q and $\overline{Q}$ become 0, which contradict its function. Then if the next input are 00 (latch), the new output is unpredictable, either 01 or 10, depending on which data travel faster

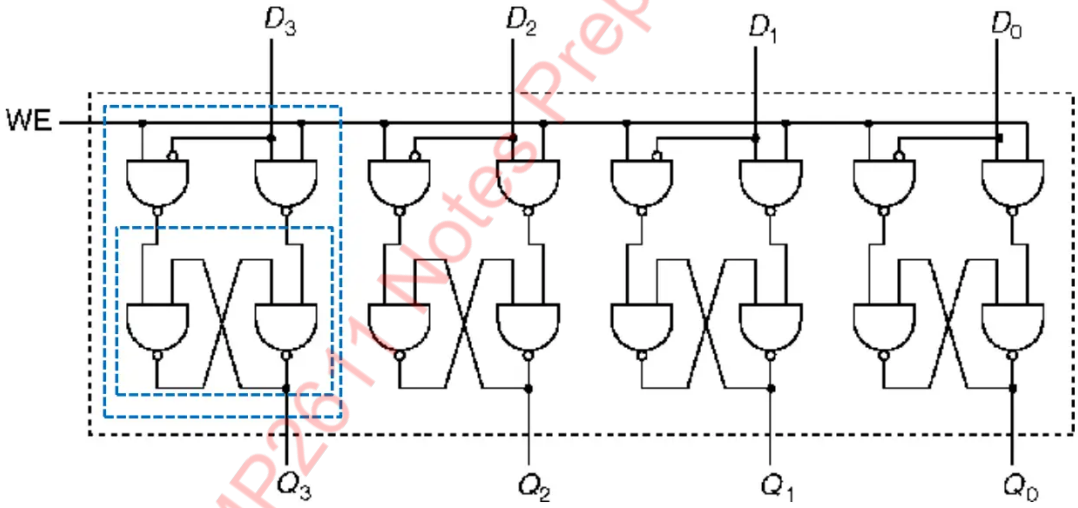





- D latch: A SR latch with **Write-Enabled** function
  - 2 inputs, WE and D
  - If WE = 0, latch
  - If WE =1, then output Q will change to D
    - If D = 1, then S = 1, R = 0
    - If D = 0, then S = 0, R = 1

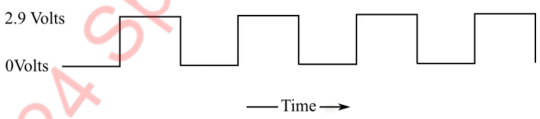



- Register: It is a collection of D latches, using the same WE input.
  - Stores multiple bits



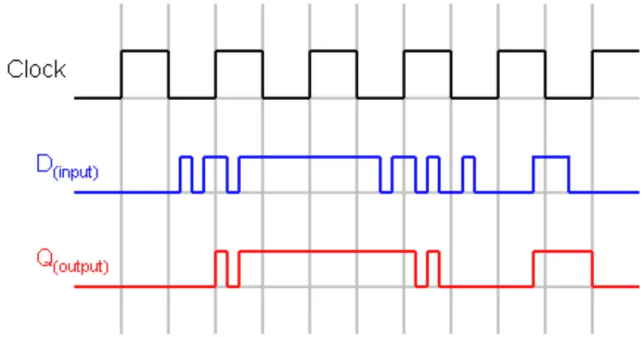

# Clock

- A free-running signal with a fixed cycle time (**clock period**)
- All signals on a ship share a clock signal, which they must accept inputs, process and produce outputs in regulated timing.

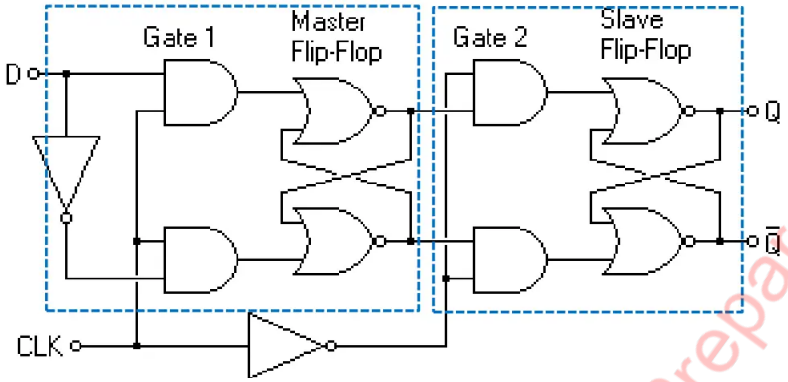



- **Edge triggered clocking**: For sequential logic circuits, all changes of state occur at either the rising edge (left) or falling edge(right).
  - No change during cruising
- **Timing Diagram**:

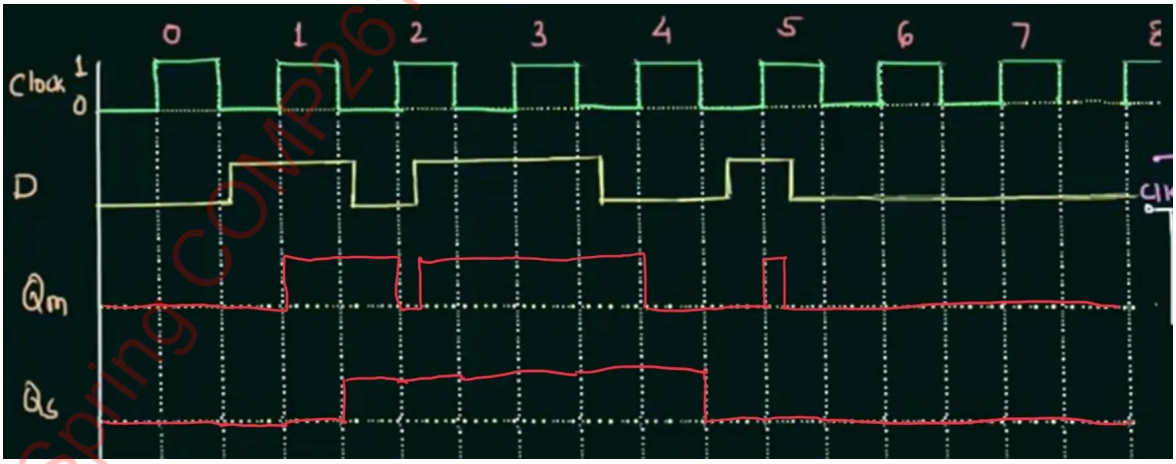
- Only when the clock signal is 1 will the output copy the original input
- When the clock signal is 0, the output will hold at its last state.
- The following diagram is a **Positive level triggered** device:



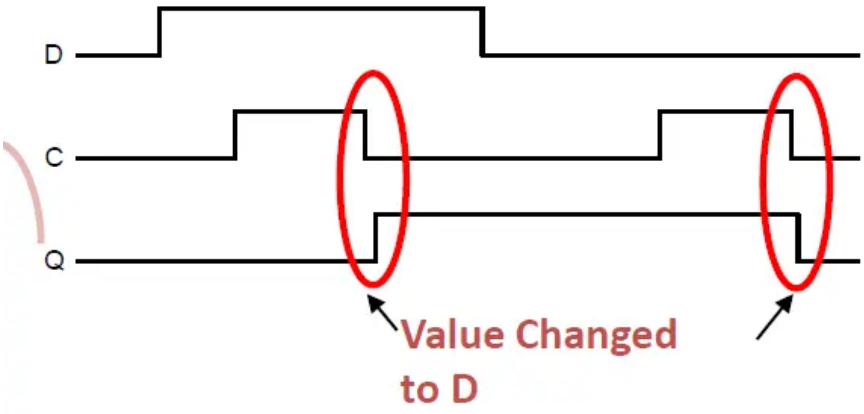

- **Master-Slave D flip-flop**:

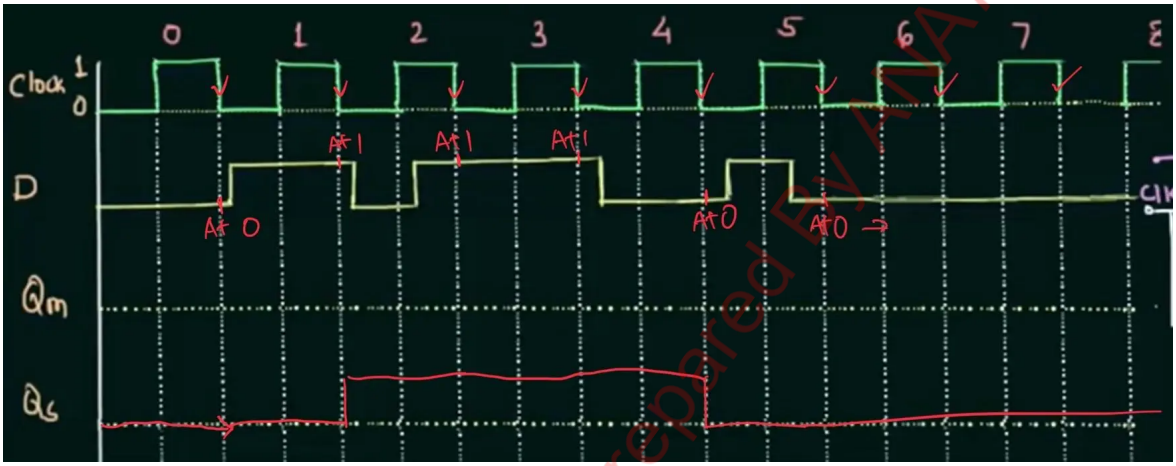



  - An improved D latch which prevents inputs 11 which causes uncertainty.
    - Done by using a clock feed to master D latch, and the same but **reversed** clock is feed to the slave D latch.
  - Flip-Flop: Outputs can only be changed when the clock
  - **The output from master flip-flop is feed into the slave flip-flop together with the reversed clock**.




  - When clock is asserted, the input D is updated in the master flip flop
  - When the clock is de-asserted, the input D (= Output of master flip flop) is updated in the slave flip flop and being output to Q
  - Summary: The output Q of the master-slave flip flop is related to the **falling edge** of the clock, that is, $Q_s$ agrees with D when clock de-asserts to 0.

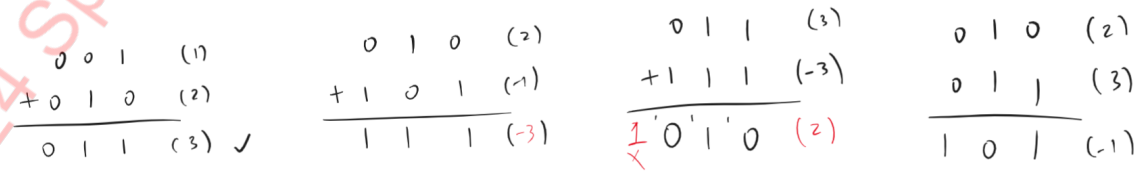



# 3. Data Representation

## Signed and Unsigned Integer

- **Unsigned Integer**:
  - Min: 000..
  - Max: 111...
  - Range: 0~$2^k - 1$
- **Sign-n-magnitude**:
  - The most significant bit represent whether the number is positive or negative. **0 = positive, 1 = negative**
    - There are n-1 bits represent the actual number. No change to these bits.
  - There exist +0 and -0, which duplicates
  - Disadvantage: Errors in arithmetic computation



- **1's complement**:
  - If the number is positive, convert to binary directly

    - If the number is negative, convert the positive number to binary, and flip all bits (0 change to 1 and 1 change to 0).
    - Drawback of 1's complement:
        - Exist positive and negative 0
        - Need to take care of carry for addition
    - Representable range: $-2^{k-1}+1$~ $2^{k-1}-1$
- **2's complement**:
    - It still use the most significant bit as the sign bit.
    - The positive side is calculated normally and added a 0 to the front
    - The negative side:
        - **Flip all bits**
        - **Add 1** to the least significant bit (or the current decimal integer), 000 -> 001
    - Need to make sure the answer falls within representable range (e.g. Need 4 bits to represent -5)
    - Representable range: $-2^{k-1}$~ $2^{k-1}-1$
        - The 0 occupies one bit from the positive half
        - For the negative half, there is no such requirement

| Bit Sequence | Unsigned | Sign-n-magnitude | 2's Complement | 1's complement |
|---|---|---|---|---|
| 0000 | 0 | 0 | 0 | 0 |
| 0001 | 1 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 | 7 |
| 1000 | 8 | -0 | -8 | -7 |
| 1001 | 9 | -1 | -7 | -6 |
| 1010 | 10 | -2 | -6 | -5 |
| 1011 | 11 | -3 | -5 | -4 |
| 1100 | 12 | -4 | -4 | -3 |
| 1101 | 13 | -5 | -3 | -2 |
| 1110 | 14 | -6 | -2 | -1 |
| 1111 | 15 | -7 | -1 | -0 |

- Converting a negative integer to 2's complement:
    - e.g. -5

    - Convert 5 to binary (0101).
    - Flip all bits (1010) to get 1s complement
    - Add 1 to get the result (1011)
- Converting from 2's complement to a negative number:
    - Minus 1 from the bit sequence.
    - Flip all bits and convert the result to decimal. Add a negative sign.
- If value is too big (out of representable range): **Overflow**
- If value is smaller than the smallest integer that can be represented: **Underflow**

### Largest integer represented by a 32 bit word

☐ $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = (2^{31} - 1)_{10} = 2{,}147{,}483{,}647_{10}$

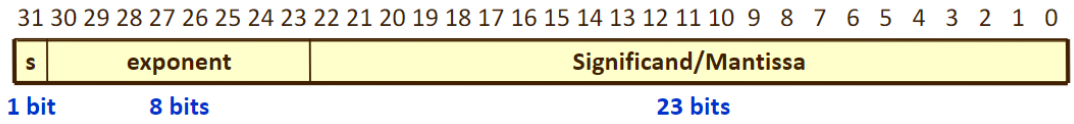
### Smallest integer represented by a 32 bit word

☐ $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2^{31}{}_{10} = -2{,}147{,}483{,}648_{10}$

- **Sign extension**: The sign (0 for positive and 1 for negative) is used to extend the bit sequence if the integer does not need that much bits to represent.
- **Zero extension**: Fills the missing bits with 0.

# Floating Point Numbers

- Converting a number with decimal point into binary:
    - e.g. 5.75 = 4 + 1 + 0.5 + 0.25 = $2^2 + 2^0 + 2^{-1} + 2^{-2} = 101.11_2$
- **Normalized Scientific Notation**: By shifting the binary decimal to left x places, we times a $2^x$ at the end of the expression
    - Target: Make the first bit to be 1 and make only 1 bit appears to the left of binary decimal
    - e.g. $101.11_2 = 1.0111 \times 2^2$
    - Allow representing numbers that are very large and very small
- **Significand (mantissa)**: The part of bits that appears to the right of the binary decimal.
- **Exponent**: The power of 2 in scientific notation, can be positive or negative
- Floating point representation is **approximate** arithmetic, values between two points need to be approximated.
    - Finite range and limited precision.
    - Equal comparison between floating point will always return false.
- IEEE754 floating point representation:
    - Single Precision: Use 32 bits to represent floating point.
        - 1 bit for sign (MSB), 8 bits for exponent, 23 bits for the mantissa.




- This allows easy comparison between two floating points. (Sign > Exponent > Mantissa)
- More exponents and less mantissa bits will cause each value to be farther away (less precise, range bigger).
- More mantissa and less exponent bits will cause the range to be smaller, and each value will be closer (better precision)

  **Interpretation**

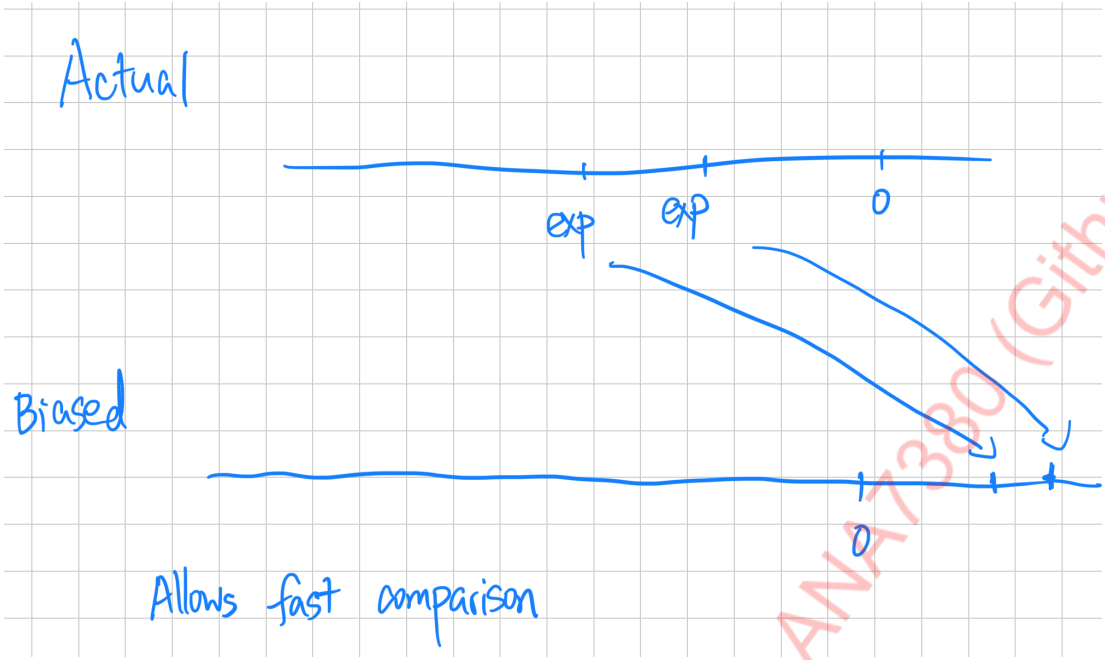
  - ☐ Roughly gives 7 decimal digits in precision
  - ☐ Exponent scale of about $10^{-38}$ to $10^{+38}$

- Double Precision:
  - 1 bit for sign, 11 bits for exponent and 52 bits for mantissa

    **Interpretation**

    - ☐ Provides precision of about 16 decimals
    - ☐ Exponent scale from $10^{-308}$ to $10^{+308}$

- e.g. $-85.125 = -(64+16+4+1)+(0.125) = -(2^6+2^4+2^2+2^0+2^{-3})$
  - In binary: $-(1010101.001)$
  - Normalize: $-(1.010101001 \times 2^6)$
  - If single precision: 1+8+23
    - Set 1 to the sign bit (negative)
    - Set the **mantissa part** (Ignore the leading 1 to increase precision, also called implicit 1) and pad 0s to the end
    - 6 is the actual exponent, we need to save the biased exponent, which is 6+127 = 133 = 128 + 4 +1 = $10000101_2$
      - Bias for single precision: 127 $(2^7-1)$
      - Bias for double precision: 1023 $(2^{10}-1)$
    - IEEE754 Answer: 1 10000101 01010100100000000000000
- Bias is for ensure exponent is unsigned value.
  - Now actual exponent (might be negative) are shifted along the axis to become positive.




- Converting from single precision back into real number:
  - e.g. 1 10000001 01000000000000000000000
  - Convert the biased exponent to integer and minus 127 (bias)
    - 128+1-127 = 2
  - Write the binary form: $-1.01 \times 2^2 = -101_2 = -5.0_{10}$
- From IEEE754 -> binary: $x = (-1)^{sign} \times (1.mantissa) \times 2^{Biased\ Exponent - Bias}$
- Example:

**Give the IEEE754 representation of -0.75$_{10}$ in single & double precisions**

**Answer**

- Scientific notation: -0.75 = -0.11$_2$ x $2^0$
- Normalized scientific notation: -1.1$_2$ x $2^{-1}$
- Sign = 1 (negative), exponent = -1

- Single precision:

  S = 1, Significand = 100...00 (23 bits)

  Biased exponent = -1+127 = 01111110

  1 01111110 10000000000000000000000

- Double precision:

  S = 1, significand = 100...00 (52 bits)

  Biased exponent = -1+1023 = 01111111110

  1 01111111110 1000000000000000 ....000

- Another example:

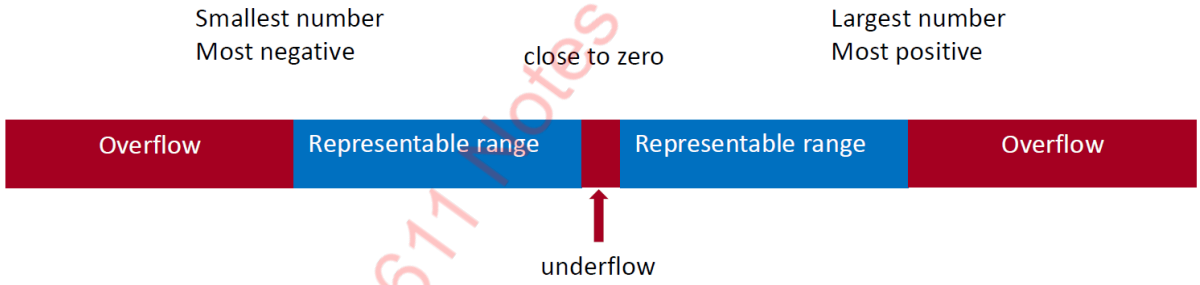
## What decimal number is represented by this word (single precision)?

| 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 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Answer:

$$
\begin{aligned}
&(-1)^{s} \times (1 + Significand) \times 2^{(Exponent - Bias)} \\
&= (-1)^{1} \times (1 + 0.25) \times 2^{(129-127)} \\
&= -1 \times 1.25 \times 2^{2} \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

- Range of Single Precision without considering Denormalized/Special Case:
  - Exponent bits 00000000 and 11111111 are reserved
  - Smallest positive value: 0 00000001 00000.....
    - Exponent: 1 - 127 = -126, so smallest binary = $1.0_2 \times 2^{-126} = 1.2 \times 10^{-38}$
  - Largest positive value: 0 11111110 111111.....
    - Exponent: 254 - 127 = 127, so largest binary = $1.111\ldots \times 2^{127} = 2_2 \times 2^{127} = 3.4 \times 10^{38}$




  - Both ends will overflow because the most negative and the most positive number only differs by the sign bit. Any number greater than the maximum representable range will be overflow.
- If the exponents are 0000 0000 (**Denormalized**) or 1111 1111 (**Special Case**):

## Single precision:

Denormalized | Normalized | Special Cases

| Significand \ Exponent | 0 | 1 - 254 | 255 |
|---|---|---|---|
| 0 | 0 | $(-1)^S \times (1.F) \times (2)^{E-127}$ | $(-1)^S \times (\infty)$ |
| ≠ 0 | $(-1)^S \times (0.F) \times (2)^{-126}$ | | non-numbers e.g. 0/0 , $\sqrt{-1}$ |

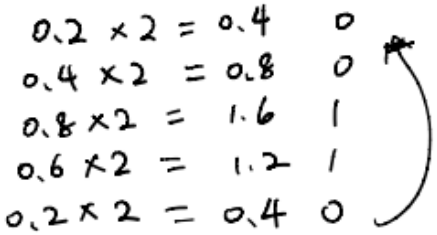
## Double precision:

| Significand \ Exponent | 0 | 1 - 2046 | 2047 |
|---|---|---|---|
| 0 | 0 | $(-1)^S \times (1.F) \times (2)^{E-1023}$ | $(-1)^S \times (\infty)$ |
| ≠ 0 | $(-1)^S \times (0.F) \times (2)^{-1022}$ | | non-numbers e.g. 0/0 , $\sqrt{-1}$ |

- The power -126 comes from 1-bias

- Smallest representation (except 0) is when all exponent bits are 0 and the last mantissa bit is 1. (Different from above)
- Largest representation (except infinity) is when all exponent bits except the last one is 1 and all mantissa bits are 1. (Same as above)
- Converting from fraction to binary:
  - Times the decimal number by 2, and record its integer part
  - If the integer part of the result is $1.x$, the next calculation starts with $0.x$
  - Continue multiplying by 2 until the result is 1.0 (can be represented by IEEE754) or it appears a result that appeared before (Cannot represented by IEEE754, need to approximate to the closest value by repeating)
    - e.g. 0.125
    - $0.125 * 2 = 0.25$ take out 0
    - $0.25 * 2 = 0.5$ take out 0
    - $0.5 * 2 = 1.0$ take out 1 and stop
    - Read from top to bottom, $0.125_{10} = 0.001_2$

2) 0.2 (6 points)




$0.2 \approx 0.0011$

Mantissa = 1.1

Exponent = -3

$-3+127 = 124 = 64+32+16+8+4 = 1111100$

0 01111100 1001 1001 1001 1001 1001 100

- Converting from decimal value to hexadecimal:
  - Divide the decimal by 16, and record its result and remainder.
  - Continue dividing the result by 16 until it becomes 0.
  - Construct the remainder part together, if the remainder of the division is greater than 10, change it to A (10),B (11)...E(15) instead.
  - The result is now hexadecimal, $(?)_{16} = 0x(?) = (?)_{HEX}$
    - e.g. $39 \div 16 = 2$ remain 7
    - $2 \div 16 = 0$ remain 2
    - Hex = 27
- Converting from binary to octal:
  - Separate the bit sequence into bits of 3, and convert them to decimal
  - Construct the octal form by combining the decimal values together.

| OCTAL | BINARY |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

- Converting from decimal to octal:
  - Divide the decimal by 8 and record the result and remainder
  - Continue divide the result by 8 until it becomes 0.
  - Construct the octal bits by reading the remainder from bottom to top.

# Characters

- ASCII table:

| $b_7$ → | | | | | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_6$ → | | | | | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| $b_5$ → | | | | | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| Bits | $b_4$ ↓ | $b_3$ ↓ | $b_2$ ↓ | $b_1$ ↓ | Row ↓ / Column → | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | NUL | DLE | SP | 0 | @ | P | ` | p |
| | 0 | 0 | 0 | 1 | 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| | 0 | 0 | 1 | 0 | 2 | STX | DC2 | " | 2 | B | R | b | r |
| | 0 | 0 | 1 | 1 | 3 | ETX | DC3 | # | 3 | C | S | c | s |
| | 0 | 1 | 0 | 0 | 4 | EOT | DC4 | $ | 4 | D | T | d | t |
| | 0 | 1 | 0 | 1 | 5 | ENQ | NAK | % | 5 | E | U | e | u |
| | 0 | 1 | 1 | 0 | 6 | ACK | SYN | & | 6 | F | V | f | v |
| | 0 | 1 | 1 | 1 | 7 | BEL | ETB | ' | 7 | G | W | g | w |
| | 1 | 0 | 0 | 0 | 8 | BS | CAN | ( | 8 | H | X | h | x |
| | 1 | 0 | 0 | 1 | 9 | HT | EM | ) | 9 | I | Y | i | y |
| | 1 | 0 | 1 | 0 | 10 | LF | SUB | * | : | J | Z | j | z |
| | 1 | 0 | 1 | 1 | 11 | VT | ESC | + | ; | K | [ | k | { |
| | 1 | 1 | 0 | 0 | 12 | FF | FC | , | < | L | \ | l | \| |
| | 1 | 1 | 0 | 1 | 13 | CR | GS | - | = | M | ] | m | } |
| | 1 | 1 | 1 | 0 | 14 | SO | RS | . | > | N | ^ | n | ~ |
| | 1 | 1 | 1 | 1 | 15 | SI | US | / | ? | O | _ | o | DEL |

  - Each characters are represented by a 7 bit binary number (total 128)
  - In binary representation, we add a 0 in the beginning then $b_7, b_6, \ldots$
  - By knowing the binary representation, we can convert it into decimal (multiply by $2^n$) or hexadecimal (base 16, separate into bits of 4 and convert it to decimal, with 10 = A, 11 = B, ...15 = E)
- e.g. Convert $32363131_8$ to characters using ASCII:
  - 0x32: 011 + 010, because hexadecimals are bits of 4, so we add 0 to the front:
    - 0011 0010 = 2
  - 0x36: 0011 0110 = 6
  - 0x31: 0011 0001 = 1
  - ANS = 2611
- e.g. Convert 0x32363131 from 2's complement to integer
  - Bit sequence: 0011 0010 0011 0110 0011 0001 0011 0001
  - Positive, $2^0 + 2^4 + 2^5 + \ldots = 842412337$
  - Same result if unsigned
- e.g. Convert 0x32363131 from IEEE754 to floating point
  - Bit sequence: 0 01100100 01101100011000100110001
  - Positive, exponent = 100-127 = -27
  - $1.01101\ldots \times 2^{-27} = 0.000000010604979$

# 4. MIPS ISA and Assembly

- Programmers write high level language and converted to **assembly language** by compiler
- Computers only understand **machine language** translated by assembler
- **Instruction Set Architecture (ISA)**: The set of instructions that a CPU can understand and execute
  - Hardware manufacturers need to follow ISA architecture to build the product
  - Two processors using the same ISA can run the same program directly.
- Assembly language: Simple English instructions that represent the operations the processor can perform.

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

C compiler

```
swap:
    muli $2, $5,4
    add  $2, $4,$2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

# Microprocessor without Interlocked Pipeline Stages (MIPS)

- RISC Design

## Arithmetic Operations:

- Add, subtract: Two sources and one destination
- `add a,b,c` Means `a = b + c`
- `sub x,y,z` Means `x = y - z`
- The first variable indicate the destination

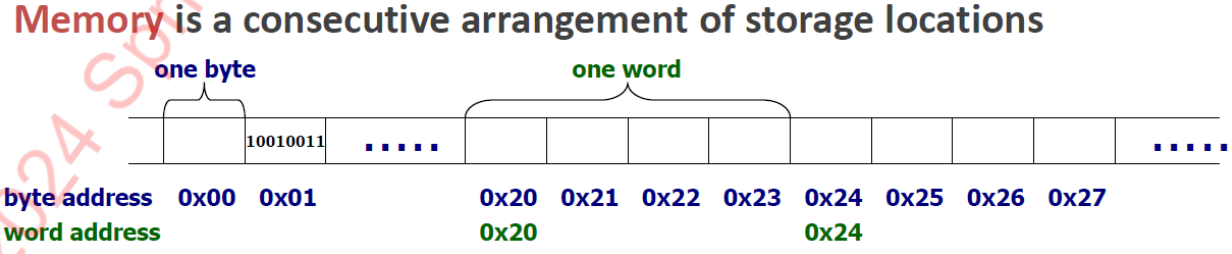
C++ code: `f = (g + h) - (i + j);`

Using registers `$s0`, `$s1`, `$s2`, `$s3`, `$s4`, to hold variables f, g, h, i, j

Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

- Comment: `#`
- **Registers**: **Fast temporary storage inside** the processor used to hold variables
  - Variable: A logical storage, its amount can be unlimited
  - Register is a physical storage, its amount is limited. Low level language must use physical storage to perform computations.
- MIPS has **32** general purpose registers, each in **32 bits** (word)
  - Saved registers `$s0, $s1, ... , $s7` or `$16 ~ $23`, used to store variables in high-level program
  - Temporary registers `$t0, $t1, ... , $t7` or `$8 ~ $15`, used to store temporary data
  - `$zero` or `$0` hold constant zero, read-only.
    - `add $s2, $s1, $zero`
    - `addi $s1, $zero, 5`
- Main memory provides large storage for millions of data elements
  - Storing arrays, structures and dynamic data
- MIPS does not allow values stored in memory be manipulated directly.
  - Data must be loaded from memory to a register before MIPS perform calculations and the results are stored back to the memory.
  - `lw` (load word) load data from memory to register
    - `lw $s1, 0($s2)` means load the value of `$s2` into `$s1`
    - `$s2` should be holding an address, if not, then need to `la` first
  - `sw` (store word) moves data from register to memory
    - `sw $s1, 0($s2)` means save the value of `$s1` into `$s2`
    - The **offset** (In bytes) must be a constant, and `()` must be provided
- Memory: from 0x00 ~ 0x0F represents 16 bytes

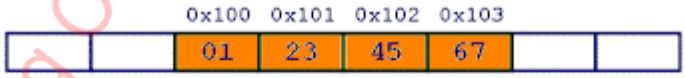



  - To specify the address of the memory location of any **array** element in assembly language, we need:

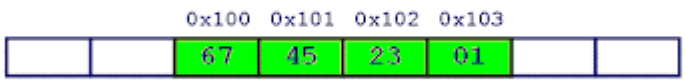
    - **Base address**: starting address of an array
    - **Offset**: distance of target location from starting address. A constant that can be either positive or negative. Offset = **Index** $\times$ **Num of bytes of the data type**
        - e.g. `A[3] = &A[0] + 4*3`
- For a main memory space of 16 bit addresses, it can store from 0000.... to 1111.... If each space can store one Byte of information, in total it can store $2^{16}$ bytes.
    - To store data with more than one byte, it will occupy the next few bytes as required

      Data types:

      - Instructions are all 32 bits
      - byte(8 bits), halfword (2 bytes), word (4 bytes)
      - a character requires 1 byte of storage
      - an integer requires 1 word (4 bytes) of storage
    - For data with more than one bit, it is enough to know their starting position (similar to pointer arithmetic)
        - Decide using `lw` or `lb` ... depending on the data types
- E.g. For an word array (each space 4 bytes) of 100 words. We want to load the 8th element in the array. If the starting position is stored in `$s0`
    - `lw $t0, 32($s0)` //load the 8th element ($8 * 4 = 32$ bytes from the start) from `$s0` and store it in `$t0`
    - `add $t0, $t1, $t0` //add the loaded value to the value stored in temp 1 register, and store it back into t0
    - `sw $t0, 48($s0)` //Store the result in temp 0 into 48th byte after the starting point of the array (the 12th element)
- **Endianness** is the order of storing data in the computer memory
    - Little endian: Meaning the **end of the word (Least Significant Bit)** being stored in the *smaller* address
    - Big endian: Meaning the end of the word being stored in the *larger* address
        - MIPS use big endian




Big Endian




Little Endian

- `addi` : adding a constant value (**immediate**) to a value stored in register
    - Syntax: `addi $t0, $s1, 8` // `$t0 = $s1 + 8;` the constant must be at the end

  - There is no `subi`, in order to preform immediate subtractions, *make the third argument to be negative numbers.*
  - We can store the result back into the original register: `addi $s0, $s0, 2`
- A program is a mixture of these 3 types of operations:
  - Memory is **outside the processor**, takes a *long* time to load and store
  - Register is **inside the processor**, takes a *short* time to get the values.
  - Constant are **already encoded in the instructions**, and are *immediately available*

## Logical Operations: and/or/nor they are bit-by-bit operations

- `and`: by calling `and $t0, $s1, $s2`, it will compare `$s1` and `$s2` by preforming AND on every single bits, and save the result to `$t0`
  - e.g.

    $t1 = 0011 1100 0000 0000$_2$
    $t2 = 0000 1101 0000 0000$_2$

    $t0 = 0000 1100 0000 0000$_2$
- `or`: `or $t0, $t1, $t2`

  $t1 = 0011 1100 0000 0000
  $t2 = 0000 1101 0000 0000
  $t0 = 0011 1101 0000 0000

- `nor`: `nor $t0, $t1, $t2`

  $t1 = 0011 1100 0000 0000
  $t2 = 0000 1101 0000 0000
  $t0 = 1100 0010 1111 1111

- `andi`: and with an immediate operand
  - `andi $t0, $t1, 4`: `$t1` stores the binary bit sequence of the loaded word, to compare the bit sequence with 4, convert 4 to binary (0100) and pad zeroes to its left. Then perform and bit-wise operation to store it in `$t0`
- `ori`: or with an immediate operand
- shift: Move all the bits in a word to the left or right
  - Filling the empty bits with 0, bits left the window will be ignored
  - `sll`: Shift left logical. `srl`: Shift right logical
  - `sll $t0, $t1, 2` means shift the bit sequence in `$t1` to the left by 2 bits.
  - Shifting left (Right) by k bits is equivalent to multiplying (Dividing) the original result by $2^k$ (Beware of overflow!)
- Summary of instructions in MIPS:

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $s1, $s2, $s3 | $s1 = $s2 + $s3 | 3 operands |
| | subtract | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 | 3 operands |
| | add immediate | addi $s1, $s2, 100 | $s1 = $s2 + 100 | 2 operands, 1 constant |
| Logical | and | and $s1, $s2, $s3 | $s1 = $s2 & $s3 | 3 operands, bit-by-bit and |
| | or | or $s1, $s2, $s3 | $s1 = $s2 \| $s3 | 3 operands, bit-by-bit or |
| | nor | nor $s1, $s2, $s3 | $s1 = ~($s2 \| $s3) | 3 operands, bit-by-bit or |
| | and immediate | andi $s1, $s2, 100 | $s1 = $s2 & 100 | 2 operands, 1 constant, bit-by-bit |
| | or immediate | ori $s1, $s2, 100 | $s1 = $s2 \| 100 | 2 operands, 1 constant, bit-by-bit |
| | 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[$2+100] | Word from mem to reg |
| | store word | sw $s1, 100($s2) | Memory[$2+100] = $1 | Word from reg to mem |

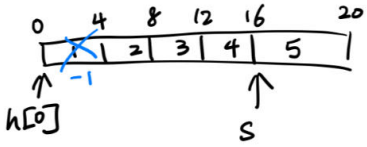
# MIPS Program

```
#############
# - We need to declare "variables" & "Arrays" used in the program in a data segment.
# - The compiler recognize .data as the beginning of data segment
.data -1                    # assembler directive
h: .word 1 2 3 4            # h is an array of size 4, each element is a word (32 bit)
s: .word 5                  → They are put in sequential memory

# The 3 lines below let the system know the program begins here
.text
.globl __start
__start:

# Write your program code here
la $s0, h                   # Obtain starting address of array h,  s0 = x (a constant)
lw $s1,8($s0)               # $s1 = content in memory address x + 8 = 3 = h[2]

la $s2, s
lw $s3, -12($s2)            # $s2 = content of address of s -12 = ?
sub $s3, $s3, $s1           # Q1: Guess what is the value of $s3 ?
sw $s3, 0($s0)              # Q2: How are the values of array h changed ?
```




when defining .data they are put in consecutive memory

$s0: base addr of h[]
$s1: stores h[2] = 3

$s2: addr of s
$s3: stores 3 element before $s2's address = 2
↳ $s3 = $s3 − $s1 = 2 − 3 = −1

Save the result to 0 offset of $s0 = h[0]

**h: .word 1 2 3 4 # h is an array of size 4**

**s: .word 5**

| | Address | Value | Array element |
|---|---|---|---|
| h → | X -th byte | 1 | h[0] |
| | X+4 -th byte | 2 | h[1] |
| | X+8 -th byte | 3 | h[2] |
| | X+12 -th byte | 4 | h[3] |
| s → | X+16 -th byte | 5 | |

# MIPS Instructions for Control Flow

- `beq`: branch if equal
  - `beq reg1, reg2, L1`
  - If `reg1 == reg2`, jump to L1
- `bne`: branch if not equal
  - `bne reg1, reg2, L1`
  - If `reg1 ! = reg2` jump to L1.
- Unconditional jump `j`
  - `j L1` (Direct jump)
- Converting from C++ codes to MIPS assembly language:

```
## IN C++ ##
        if (i == j)
                f = g + h;
        else if (i == g)
                f = g - h;
        else
                f = g + j;
## APPENDIX ##
        f = $s0
        g = $s1
        h = $s2
        i = $s3
        j = $s4
## MIPS Solution 1 ##
        beq $s3, $s4, IJ
        beq $s3, $s1, IG
        add $s0, $s1, $s4
        j exit
        IJ: add $s0, $s1, $s2
        j exit
        IG: sub $s0, $s1, $s2
        exit:
## MIPS Solution 2 ##
        bne $s3, $s4, ELSEIF
        add $s0, $s1, $s2
        j exit
        ELSEIF: bne $s3, $s1, ELSE
        sub $s0, $s1, $s2
        j exit
```

```
        ELSE: add $s0, $s1, $s4
        exit:
```

- **While loops** in MIPS

**Here is a traditional loop in C:**

```
while (save[i] == k)    i += 1;
```

**Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C Segment?**

**Answer:**

```
Loop:   sll $t1, $s3, 2          # Temp reg $t1 = 4 * i
        add $t1, $t1, $s6        # $t1 = address of save[i]
        lw  $t0, 0($t1)          # Temp reg $t0 = save[i]
        bne $t0, $s5, Exit       # go to Exit if save[i] != k
        addi $s3, $s3, 1         # i = i + 1
        j   Loop
Exit:
```

- **Less than** comparison in MIPS
  - `slt reg1, reg2, reg3`
  - If `reg2 < reg3` then store 00..01 (true) to `reg1`, else store 0000... (false)
  - The `for` loops in C++ will need 2 operations to check condition:
    - Compare the values in 2 registers and save the result to another register
    - The result register is compared with `$zero` register and decide branch to enter.

```
slt $t0, $s0, $s1       #If $s0 is less than $s1, $t0 will be 1
bne $t0, $zero, BRANCH  #If $t0 is not 0, then jump to BRANCH
```

- Compare with immediate: `slti reg1, reg2, immediate`
- MIPS use `beq`, `bne`, `slt`, `slti` and `$zero` to create all comparison operations:
  - Equal, not equal
  - less than, less than or equal
    - `$s0 ≤ $s1` is equivalent to `!($s0>$s1)`:

```
slt $t0, $s1, $s0
beq $t0, $zero, BRANCH
```

  - greater than, greater than or equal
    - `$s0 ≥ $s1` is equivalent to `!($s0<$s1)`:

```
slt $t0, $s0, $s1
beq $t0, $zero, BRANCH
```

- Pseudo Code: Commands that are defined in assemblers but not in the ISA, it recognizes the pseudo code and will translate it into machine codes automatically.
- **Branch With Zero**:
  - `bgez reg1, BRANCH` branch if `reg1` greater than or equal to 0
  - `bgtz reg1, BRANCH` branch if `reg1` greater than to 0
  - `blez reg1, BRANCH` branch if `reg1` less than or equal to 0
  - `blze reg1, BRANCH` branch if `reg1` less than zero
- **Jump Register**:
  - `jr reg`, where `reg1` stores the instruction address that point to the command.
  - used for procedure call and case/switch statements
- `la` are **pseudo instructions** which can be used to load an address into a register. It is not a native command in the hardware, but will be converted to **pure MIPS command** during translation in assembler.
  - `la $s0, label` stores the head of the data/instruction

```
Array: .word 4 8 12 16 20

la $t0, Array
lw $t1, 4($t0)
lw $t2, 8($t0)
```

i) So, t1 = 8, t2 = 12

```
la $s0, Label
add $s0, $s0, $t1
jr $s0
```

$s0 = 10000 + 8 jump to 10008 = sw $t1, 12($t0)

| Array → | Address | Value | Array element |
|---|---|---|---|
| | t0 | 4 | Array1[0] |
| | t0+4 | 8 | Array1[1] |
| | t0+8 | 12 | Array1[2] |
| | t0+12 | 16 | Array1[3] |
| | t0+16 | 20 | Array1[4] |

ii) s0 = 10000 after la $s0, Label1 is executed.
Hence, the next instruction to be run after jr $s0 is stored at 10000 + 8 = 10008th byte of the memory

```
Label:
add $t1, $t1, $t1
add $t1, $t2, $t2
sw $t1, 12($t0)
```

| Label → | Address | Instruction |
|---|---|---|
| | 10000 | add $t1, $t1, $t1 |
| | 10004 | add $t1, $t1, $t2 |
| | 10008 | sw $t1, 12($t0) |

iii) All MIPS instructions are fixed as 4 bytes long. So, sw $t1, 12($t0) should be executed after jr $s0 (2 instructions skipped). Array1[3] = t1 = 8 at the end

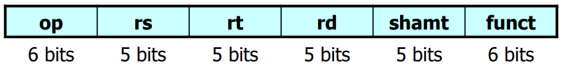
# Implement MIPS Instructions

- MIPS instructions are encoded as 32-bit instruction words
  - MIPS have 32 registers which each store 32 bits.
- Three types of instruction formats in MIPS:
  - R-type (register)
  - I-type (immediate)
  - J-type (jump)

- Each format is assigned a distinct set of values for the 1st field (opcode)
- R-type:
  - The opcode are **all zero** for R type instructions
  - The **Function Field** codes distinguish between different R type operations.
  - The first operand used to perform calculation is stored in `rs` field, and the second operand is stored in `rt` field. The destination to store the result is put **at last**, in `rd`

    - They are entered as the number of register in binary (e.g. `$s0` = `$16` so storing `10000`)

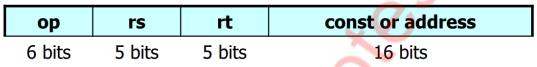
    **R-type or R-format**

    | op | rs | rt | rd | shamt | funct |
    |---|---|---|---|---|---|
    | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

    - **Instruction fields: (6 fields)**
      - **op**: basic operation of instruction, traditionally called **opcode**
      - **rs**: first register source operand
      - **rt**: second register source operand
      - **rd**: register destination operand, which gets result of operation
      - **shamt**: shift amount (number of positions to shift)
      - **funct**: function code (extends opcode)
  - `sll` and `srl` will use the shift bits
- I-type: Immediate arithmetic and load/store instructions

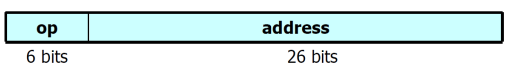
  **I-type or I-format**

  | op | rs | rt | const or address |
  |---|---|---|---|
  | 6 bits | 5 bits | 5 bits | 16 bits |

  - **Immediate arithmetic and load/store instructions**
  - **Instruction fields:**
    - **op**: as before
    - **rs**: base register
    - **rt**: register source operand (for **`sw`**) *or* destination operand (for **`lw`**)
    - **constant**: $-2^{15}$ to $+2^{15} - 1$
    - **address**: offset added to base address in rs
  - The register storing the result will be the closest to the end
  - If performing `addi`, the 16 bits for storing constant will be stored as **Two's complement**.
- J-type: Jump instructions:

  **J-type or J-format**

  | op | address |
  |---|---|
  | 6 bits | 26 bits |

- MIPS Instructions Encoding

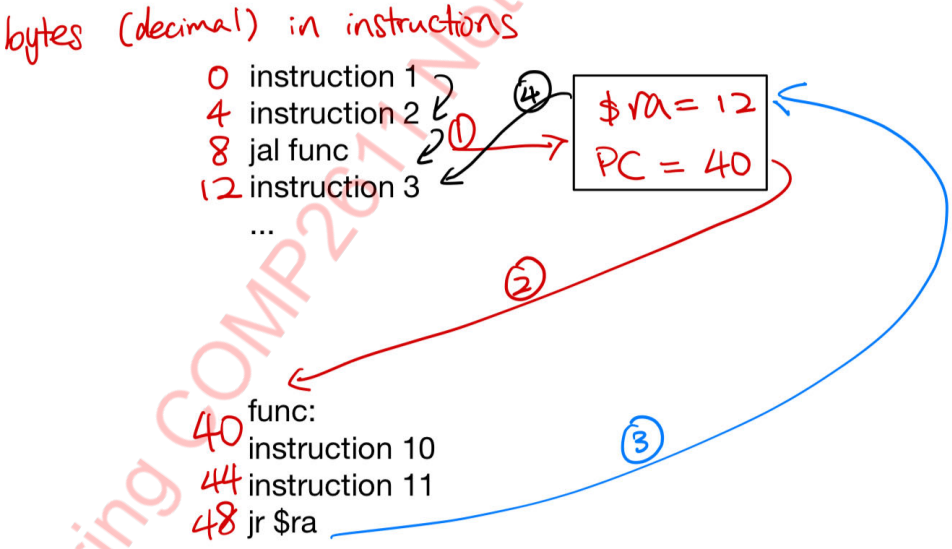
| Instruction | Type | op | rs | rt | rd | shamt | funct | const/address |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | reg | reg | reg | 0 | $32_{10}$ | - |
| sub | R | 0 | reg | reg | reg | 0 | $34_{10}$ | - |
| and | R | 0 | reg | reg | reg | 0 | $36_{10}$ | - |
| or | R | 0 | reg | reg | reg | 0 | $37_{10}$ | - |
| sll | R | 0 | 0 | reg | reg | constant | 0 | - |
| srl | R | 0 | 0 | reg | reg | constant | $2_{10}$ | - |
| addi | I | $8_{10}$ | reg | reg | - | - | | constant |
| lw | I | $35_{10}$ | reg | reg | - | - | | address |
| sw | I | $43_{10}$ | reg | reg | - | - | | address |

    - `sll` and `srl` will have 5 zeroes in the first register location
- `addu $reg1, $reg2, $reg3` and `subu`: Unsigned addition/subtraction, they will continue running the program even overflow/underflow happens.
    - `add` and `sub` will terminate the program if such case happen.
- `lw $t0, offset(reg2)` **require offset + reg2 be divisible by 4.** (To complete loading in one execution)
    - To load arbitrary byte, use `lb` instead
- The value loaded from `lb` is only 8 bits, and to store the data inside a 32 bit register, **Sign extension** is used. (extend MSB)
- `lbu $reg1, offset(reg2)`: load byte unsigned, **Zero Extended**
- `sb`: store byte in arbitrary address
    - It only stores the last 8 bits of the register into the memory
- `nor $dest, $reg1, $reg2` does NOR in bit-wise comparison
- `sllv $reg1, $reg2, $reg3` shift the value in `reg2` to left **by the value** stored in `reg3` and save it to `reg1`
    - Also shift right logical variable `srlv $reg1, $reg2, $reg3`

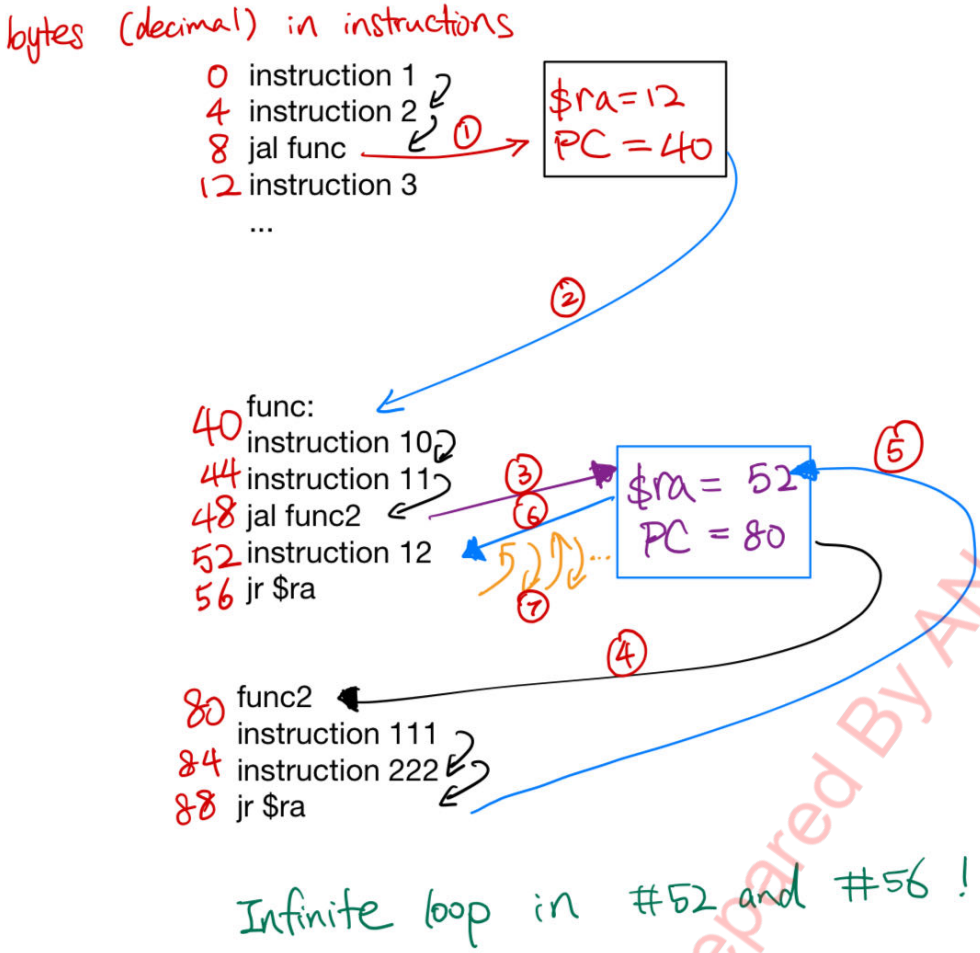
# Procedure (Function)

- Steps for executing a function in MIPS:
    1. Store the parameters in some registers
    2. Transfer control to the function
    3. Acquire the storage resources needed for the procedure
    4. Perform desired task
    5. Place the result value in a register
    6. Return control to the point of origin
- Registers:
    - `$a0` – `$a3`: arguments
    - `$v0`, `$v1`: result values
    - `$t0` – `$t9`: temporaries
        - Can be overwritten by callee
    - `$s0` – `$s7`: saved
        - Must be saved/restored by callee

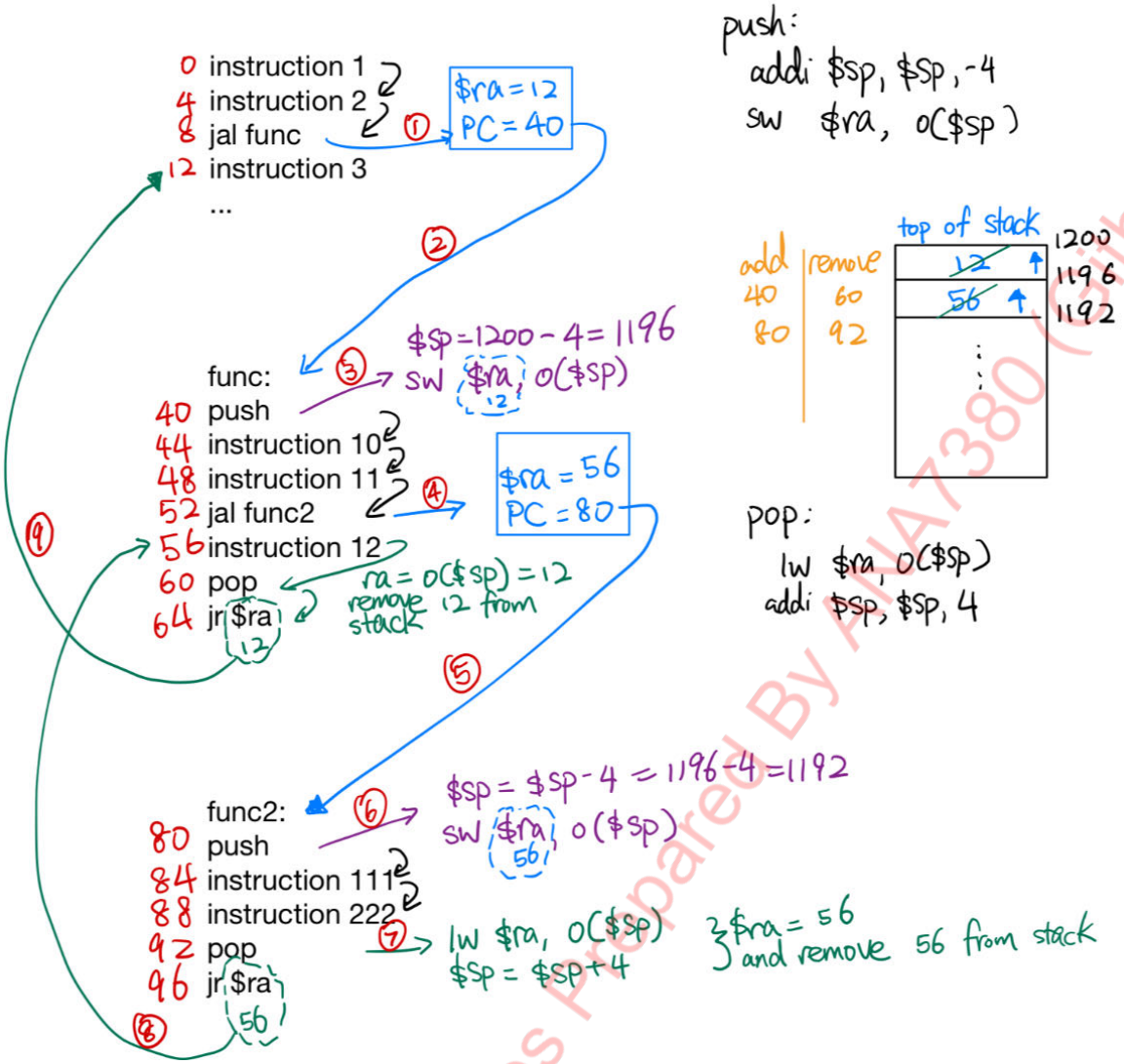
    - `$gp`: global pointer for static data
    - `$sp`: stack pointer
    - `$fp`: frame pointer
    - `$ra`: return address
- Caller: The function that calls another function (main())
- Callee: The function called by caller (Other Global functions)
- **Program counter**: A register that holds the address of the current instruction being executed.
    - When this instruction completes execution, it will update to the next instruction by adding 4 (bytes) to itself.
    - If branch, the PC counter will update itself to the branch address directly.
    - `jr $ra` means setting the next operation to start at the **return address**
- Procedure Call:
    - first stores the arguments inside the argument register
    - `jal label`: Jump and Link to label. Making a function call, which memorises the return address and jump to the function body
        - `$ra` set to store the next instruction after `jal`
        - Jump to label and start executing from there.
    - In the procedure, after calculations, the values to be returned will be stored in `v` registers
    - `jr $ra`: Return to the main block by jumping to `$ra` set by `jal`.
      - The program counter now stores `$ra`




- If a function calls another function, then the return address will be modified and we cannot return back to the original point we left main!




- Solution: Before making function call inside another function, remember the `$ra` into memory, and before returning to main function, get the saved return address from the memory (backup)
  - Save the `$ra` into stack which is pointed by `$sp` (stack pointer)
- **Stack**: Last in first out, the first element in the stack has biggest memory
  - Push: Store an element to the top of the stack
    - `addi $sp, $sp, -4`
    - `sw $ra, 0($sp)`
  - Pop: Removing the last element from the stack
    - `lw $ra, 0($sp)`
    - `addi $sp, $sp, 4`




## MIPS Memory Convention

- Functions should backup the data in `$s0 - $s7` in stack before they modify any of them, and restore the data back into the register before the function returns.
  - So that other functions (e.g. main) will hold the same values in the register before making that function call
- Stack pointers should remain the same by having balanced `push` and `pop`. Else undefined behaviour
- Return address should be same before and after a function call.

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| `$zero` | 0 | constant value 0 | n.a. |
| `$at` | 1 | reserved for assembler | n.a. |
| `$v0-$v1` | 2-3 | values for results and expression evaluation | no |
| `$a0-$a3` | 4-7 | arguments | no |
| `$t0-$t7` | 8-15 | temporaries | no |
| `$s0-$s7` | 16-23 | saved temporaries | yes |
| `$t8-$t9` | 24-25 | more temporaries | no |
| `$k0-$k1` | 26-27 | reserved for operating system kernel | n.a. |
| `$gp` | 28 | pointer to global area | yes |
| `$sp` | 29 | stack pointer | yes |
| `$fp` | 30 | frame pointer | yes |
| `$ra` | 31 | return address | yes |

- Most computers use **ASCII** coding to represent characters, one character will occupy 8 bits
  - Load to MIPS: **extend** from 1 byte to 4 bytes using sign extension to the left
    - When call `lb $s0, 0($sp)`, it loads one byte from the stack pointer's location and store it to `$s0` with sign extension
    - Use `lbu` for zero extension
  - Store from MIPS: store the last 8 bits of the register into memory
    - `sb $s0, 0($sp)`
- String Copy Example:

```
//strcpy(dest, source) is equivalent to:
int i = 0;
do{ dest[i] = source[i]; i++; }while (source[i]≠ '\0');
```

```
i:                      $s0
address of array x[]:   $a0
address of array y[]:   $a1
address of y[i]:        $t1
y[i]:                   $t2
address of x[i]:        $t3
```

```
strcpy:
        addi $sp, $sp, -4       # adjust stack for 1 item
        sw   $s0, 0($sp)        # save $s0
        add  $s0, $zero, $zero # i = 0
Loop:
        add  $t1, $s0, $a1      # addr of y[i] in $t1
        lbu  $t2, 0($t1)        # $t2 = y[i]
        add  $t3, $s0, $a0      # addr of x[i] in $t3
        sb   $t2, 0($t3)        # x[i] = y[i]
        beq  $t2, $zero, exit  # exit loop if y[i] == 0
        addi $s0, $s0, 1        # i = i + 1
        j    Loop               # next iteration of loop
exit:
        lw   $s0, 0($sp)        # restore saved $s0
        addi $sp, $sp, 4        # pop 1 item from stack
        jr   $ra                # and return
```

- For immediate (constants) with 32 bit length, we need a new operation to load them into a register `lui` (load upper immediate) and `ori` (or immediate)
  - Because I type operations only have 16 bits of space to put specify immediate
  - `lui $reg, constant` loads the upper 16 bit of the immediate into the first 16 bits of the register, and setting the last 16 bits to be all zero.
    - The constant field accepts hexadecimal values by including `0x` before hex values to indicate.
  - `ori $reg, $reg, constant` compares the 16 bit (**with 0 extension in the front**) to the value stored with register which have the upper 16 bits.
    - It does bit wise comparison with 16 bits of 0, so it will update the register to the complete 32 bit sequence.
    - `ori` can be replaced by `add` in the condition that the upper 16 bits MUST be zero

### Example: Loading a 32-bit Constant

- How to load the 32-bit constant below into register `$s0`?
  0000 0000 0011 1101 0000 1001 0000 $0000_2$ (0x003D0900)
- Solution: (assuming the initial value in `$s0` is 0)

```
lui $s0, 61          # 61_10 = 0000 0000 0011 1101
# value of $s0 becomes 0000 0000 0011 1101 0000 0000 0000 0000
ori $s0, $s0, 2304   # 2304_10 = 0000 1001 0000 0000
# now, we get the value desired into the register
```

# Addressing Mode

- Immediate addressing mode: The operand is encoded in the instruction, immediately available
  - e.g. `addi`

- Register addressing mode: The operand is stored in register
  - e.g. `add`
- Base addressing mode: the operand is the sum of base address + offset
  - e.g. `lw` `sw`
- PC-relative addressing ( `beq,bne` )
  - For I type instructions, they don't have enough bits to store the memory address of branch. So they instead use Relative address
  - Before `beq` finish executing, the PC will be set to PC + 4, and if the branch condition is true, it will calculate how many bytes are away from PC + 4.

| Address | Instruction |
|---|---|
| 40000008 | Instruction 1 |
| 4000000C | beq $zero, $s0, label |
| 40000010 | Instruction 3 |
| 40000014 | Instruction 4 |
| 40000018 | label: Instruction 5 |
| 4000001C | Instruction 6 |
| 40000020 | etc… |

**Machine code to `beq` is `0x10100002`, which means 2 instructions from the next instruction**

**PC = 0x4000000C**
**PC+4 = 0x40000010**
**Add 4*2 = 0x00000008**
**Target = 0x40000018**

| op | rs | rt | const or address |
|---|---|---|---|
| 000100 | 00000 | 10000 | 0000000000000010 |

  - In the above example, after `beq` the PC should be set to 40000010, but if the condition is true, it should jump to 4000018, the difference between the two values are 8 bytes, which means 8/4 = 2 words or instructions. So it will store 2 in binary format in the I type instruction code.
    - It stores word instead of byte because every instructions are 4 bytes, by converting to word it extend the offset by 4 times.
    - `beq` can branch backward as well! so the 16 bit storing offset is 2s complement
    - Recall: With 16 bits available to store 2s complement, it can represent $-2^{15}$ instructions ($-2^{15} \times 4$ bytes) ahead of PC+4 or $2^{15} - 1$ instructions ( $(2^{15} - 1) \times 4$ bytes) after PC+4
      - Earliest: $-2^{15} \times 4 + PC + 4$
      - Latest: $(2^{15} - 1) \times 4 + PC + 4$
- (Pseudo) Direct addressing:
  - `j` or `jal` instructions: They have only 26 bits to store the destination address of 32 bits.
  - Recall each memory address in MIPS are 32 bits
  - Also recall that each instruction are stored in address that is divisible by 4.
  - Assumption: The address of the first 4 bits of the instruction we want to jump to is the same as the current instruction.
  - So for J-type instructions, it will store the 32 bit instruction address by: Omitting the first 4 bits of the address AND remove the last 2 bits of the address (Because

it is always divisible by 4, so last 2 bits must be 00)
    - $32 - 4 - 2 = 26$
- For CPU to reconstruct the complete address to branch:
    - Copy the current address' first 4 bits
    - Fill with the 26 bits
    - Add 00 to the end
    - Jump
- Maximum distance to jump:
    - The first 4 bits are fixed, and the last 28 bits can be changed.
    - Max bytes: $2^{28}$ bytes = 256MB
- If `j` instructions cannot jump to the desired destination (e.g. first 4 bits changed), we need to load the full address into a register using `lui` and `ori` and call `jr`, `jr` can access everywhere in the memory because it is 32 bit full address.

## Summary of MIPS Addressing Modes

1. **Immediate addressing**
    - ☐ The operand is a constant within the instruction itself
2. **Register addressing**
    - ☐ The operand is a register
3. **Base addressing or displacement addressing**
    - ☐ The operand is at the memory location with address = (register) + constant
4. **PC-relative addressing**
    - ☐ The address is = (PC) + 4 + constant
5. **Pseudodirect addressing**
    - ☐ The jump address is a constant in the instruction concatenated with the upper 4 bits of the PC

## Pseudo instructions

- Instructions not implemented in the hardware, they are provided by assembler. Assembler will replace these instructions to real instructions.
- Cost of supporting pseudo instructions: Reserve a register `$at` for assembler to use.
- e.g. `move`, `la`, `blt`, `ble`, `bgt`, `bge`

# 5. Arithmetic For Computers

- Revisit converting a number to negative using 2's complement:
    - Step 1: Reverse all bits. This is equivalent to use bit sequence of 1111.. to minus by the positive binary number.
        - Sequence of all 1 is equivalent to $2^n - 1$

    - After reversing the bit sequence becomes $2^n - 1 - A$
  - Step 2: Add 1, Therefore the equation now becomes $2^n - 1 + 1 - A = 2^n - A$, note that $2^n$ will be out of window, so it will not appear in our bit sequences, remaining $-A$.
- e.g. $7 - 6$ in 32 bit sequence is equivalent to $7 + (-6) = 7 + [2^{32} - 1 - 6] + 1 = 7 + 2^{32} - 6 = 7 - 6$ because $2^{32}$ is out of representable window.
- Overflow: When the computation returns wrong answers due to numbers that are too big.
  - e.g. sign bit modified when it shouldn't be.

**Overflow detection**

| Operation | Sign Bit of X | Sign Bit of Y | Sign Bit of Result |
|---|---|---|---|
| X + Y | 0 | 0 | 1 |
| X + Y | 1 | 1 | 0 |
| X – Y | 0 | 1 | 1 |
| X – Y | 1 | 0 | 0 |

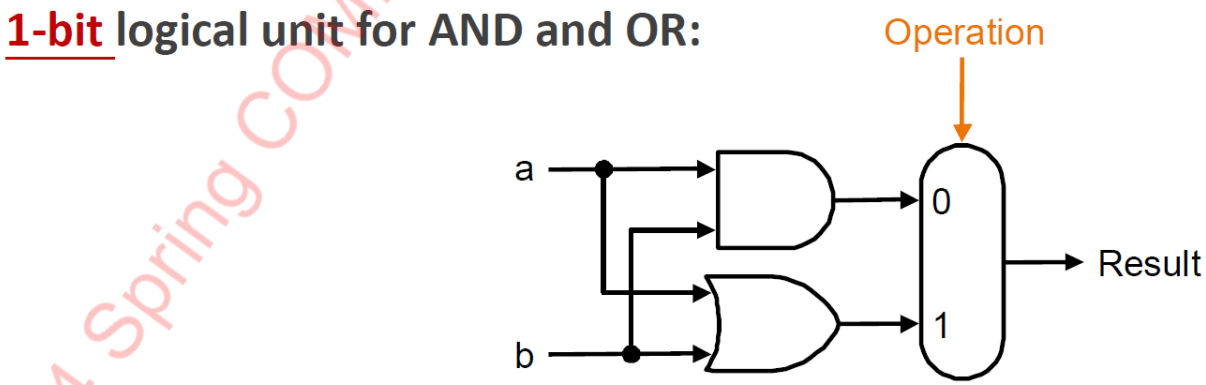
- MIPS support signed (int) and unsigned (unsigned int) data.
  - Based on the data types some instructions will output different results under same arguments.
- `slt` and `sltu` (Comparison):
  - `slt` treats both arguments to be signed integers, (The most significant bit represents +ve or -ve)
  - `sltu` treats both arguments to be unsigned integers, (The MSB is also part of the binary integer)
- In arithmetic instructions:
  - `addu/subu`: It treats both arguments to be singed, and adds/subtracts two values and does not detect overflow (No error)
  - `addiu`: It treats the immediate to be signed, because the immediate is only 16 bits, so it will perform sign extension and make it 32 bits. Then adds it to the (signed) register provided, and does not detect overflow
  - Summary: It treats all values signed, but does not detect for overflow
- In comparison instructions:
  - `sltu`: Consider registers to be unsigned
  - `sltiu`: First converts the immediate to 16 bits (Because I type instruction) and does sign extension to 32 bits, then it compares it with the given register, treating both of them unsigned
- For data transfer instructions:
  - `lhu $t0, (a)$s0`: load half word unsigned: It loads 16 bits and performs zero extension
    - The immediate field (a) in the instruction is treated as signed integers, and added to the 32 bit sequence.

  - `lbu` : load byte unsigned: It loads 8 bits and performs zero extension
    - The immediate field in the instruction is treated as signed integers, will have sign extension and added to the 32 bit sequence.
  - `lb` : Sign extend the immediate value and sign extend the bit to 32 bits.
- For Logical instruction: `and/andi/or/ori/xor/xori ...` :
  - They perform bit-wise comparison, the immediate values will have zero extension. (So the 0s in the front does not affect the bitwise comparison)
- The unsigned flag in arithmetic instructions is misleading, it mean not detecting for overflow.
- The immediate values are always sign extended (except `andi/ori/ ...` ).

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add ~~unsigned~~ | addu $s1, $s2, $s3 | $s1 = $s2 + $s3 | binary addition with overflow ignored |
| | subtract ~~unsigned~~ | subu $s1, $s2, $s3 | $s1 = $s2 - $s3 | binary subtraction with overflow ignored |
| | add immediate ~~unsigned~~ | addiu $s1, $s2, immd | $s1 = $s2 + sign-extend(immd) | binary addition, with the 16-bit immediate value sign-extended to 32 bits, overflow ignored |
| Comparison | set less than unsigned | sltu $s1, $s2, $s3 | $s1 = 1 if $s2 < $s3, 0 otherwise | $s2 and $s3 hold unsigned integers |
| | set less than immediate unsigned | sltiu $s1, $s2, immd | $s1 = 1 if $s2 <sign-extend(immd), 0 otherwise | the 16-bit immediate value sign-extended to 32-bits; comparison on two unsigned integers |
| Data transfer | load half byte unsigned | lhu $s1, immd($s2) | Memory address: $s2 + sign-extended(immd)<br>Load 2 bytes and then zero-extend to 32-bit, store it to $s1 | |
| | load byte unsigned | lbu $s1, immd($s2) | Memory address: $s2 + sign-extended(immd)<br>Load a single byte and then zero-extend to 32-bit, store it to $s1 | |

# Arithmetic Logic Unit

- ALU: The hardware component that performs arithmetic operations (add, sub) and logical operations (and, or)




  - This circuit performs both AND and OR comparison for 1 bit, and the operation selector line decides to use the result from AND or OR.
- 1 bit full adder (Also called 3,2 adder) :

- It has adds A and B and overflow from previous calculation to output result and carry over

## Truth Table and Logic Equations for 1-Bit Adder

### Truth table

| Inputs | | | Outputs | | Comments |
|---|---|---|---|---|---|
| a | b | CarryIn | CarryOut | SumOut | |
| 0 | 0 | 0 | 0 | 0 | $0 + 0 + 0 = 00_2$ |
| 0 | 0 | 1 | 0 | 1 | $0 + 0 + 1 = 01_2$ |
| 0 | 1 | 0 | 0 | 1 | $0 + 1 + 0 = 01_2$ |
| 0 | 1 | 1 | 1 | 0 | $0 + 1 + 1 = 10_2$ |
| 1 | 0 | 0 | 0 | 1 | $1 + 0 + 0 = 01_2$ |
| 1 | 0 | 1 | 1 | 0 | $1 + 0 + 1 = 10_2$ |
| 1 | 1 | 0 | 1 | 0 | $1 + 1 + 0 = 10_2$ |
| 1 | 1 | 1 | 1 | 1 | $1 + 1 + 1 = 11_2$ |

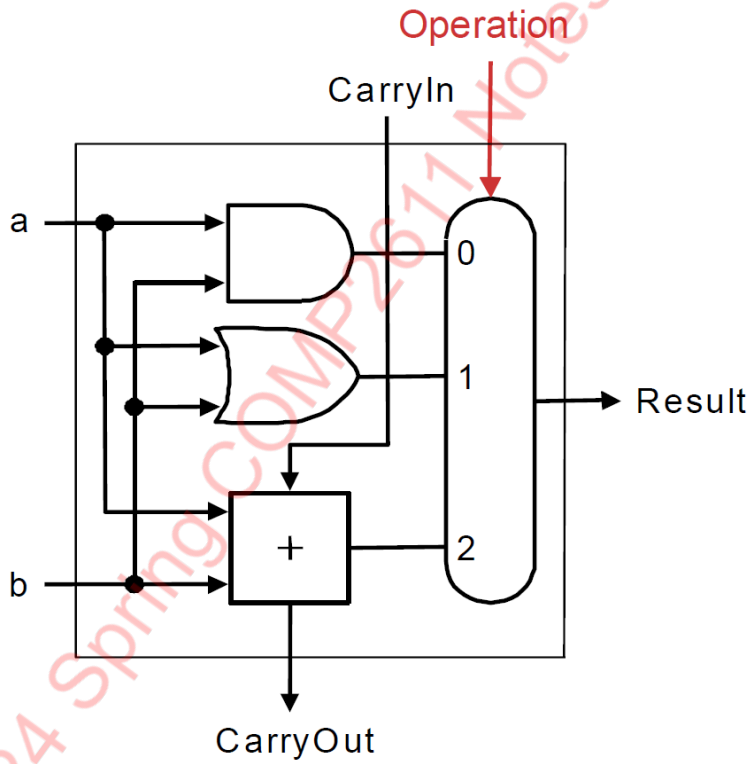
### Logic equations

$$\begin{aligned} CarryOut &= (\bar{a} \cdot b \cdot CarryIn) + (a \cdot \bar{b} \cdot CarryIn) + (a \cdot b \cdot \overline{CarryIn}) + (a \cdot b \cdot CarryIn) \\ &= (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b) \end{aligned}$$

$$\begin{aligned} SumOut = &(a \cdot \bar{b} \cdot \overline{CarryIn}) + (\bar{a} \cdot b \cdot \overline{CarryIn}) + (\bar{a} \cdot \bar{b} \cdot CarryIn) \\ &+ (a \cdot b \cdot CarryIn) \end{aligned}$$

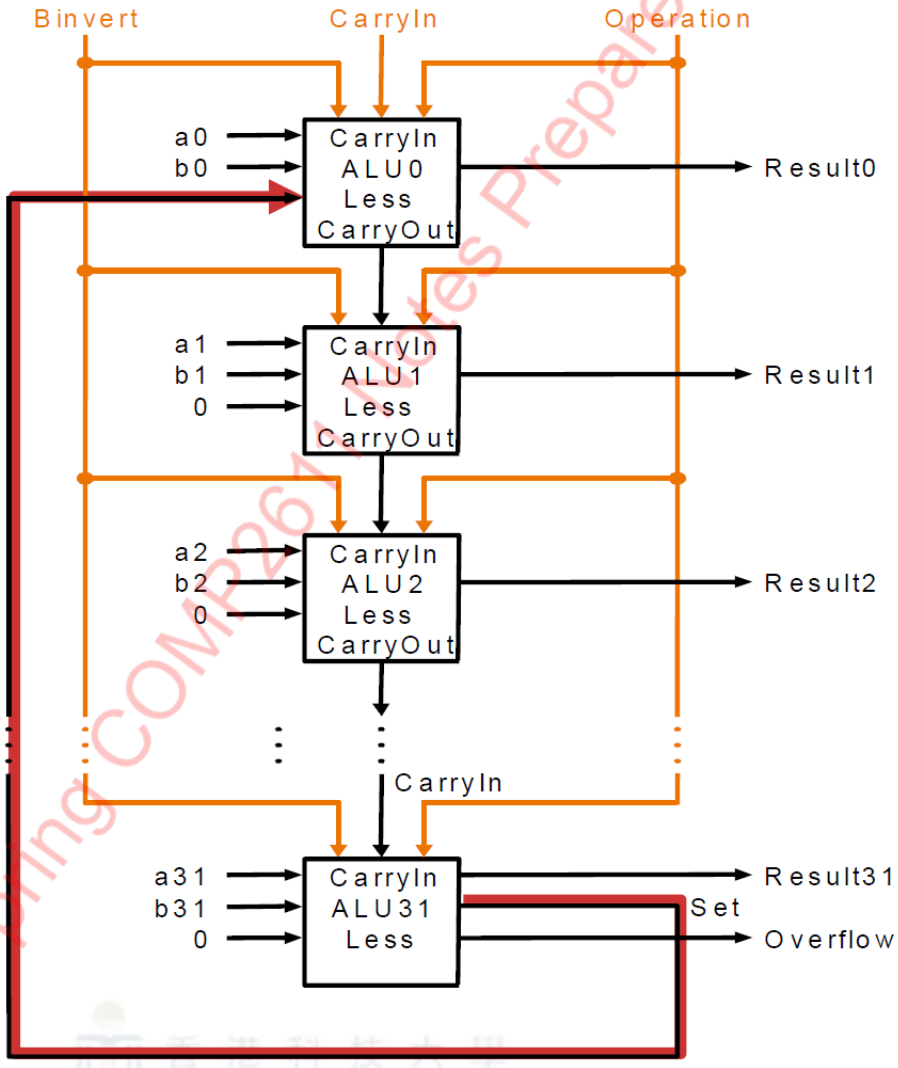
- `CarryOut` can be simplified using k-map
- `SumOut` : Equivalent to if A and B are different (XOR = 1), then C must be 0. Or A and B are same (XOR = 0), then C must be 1.
  - Circuit: XOR(xor(AB), C)

- 1 bit ALU:



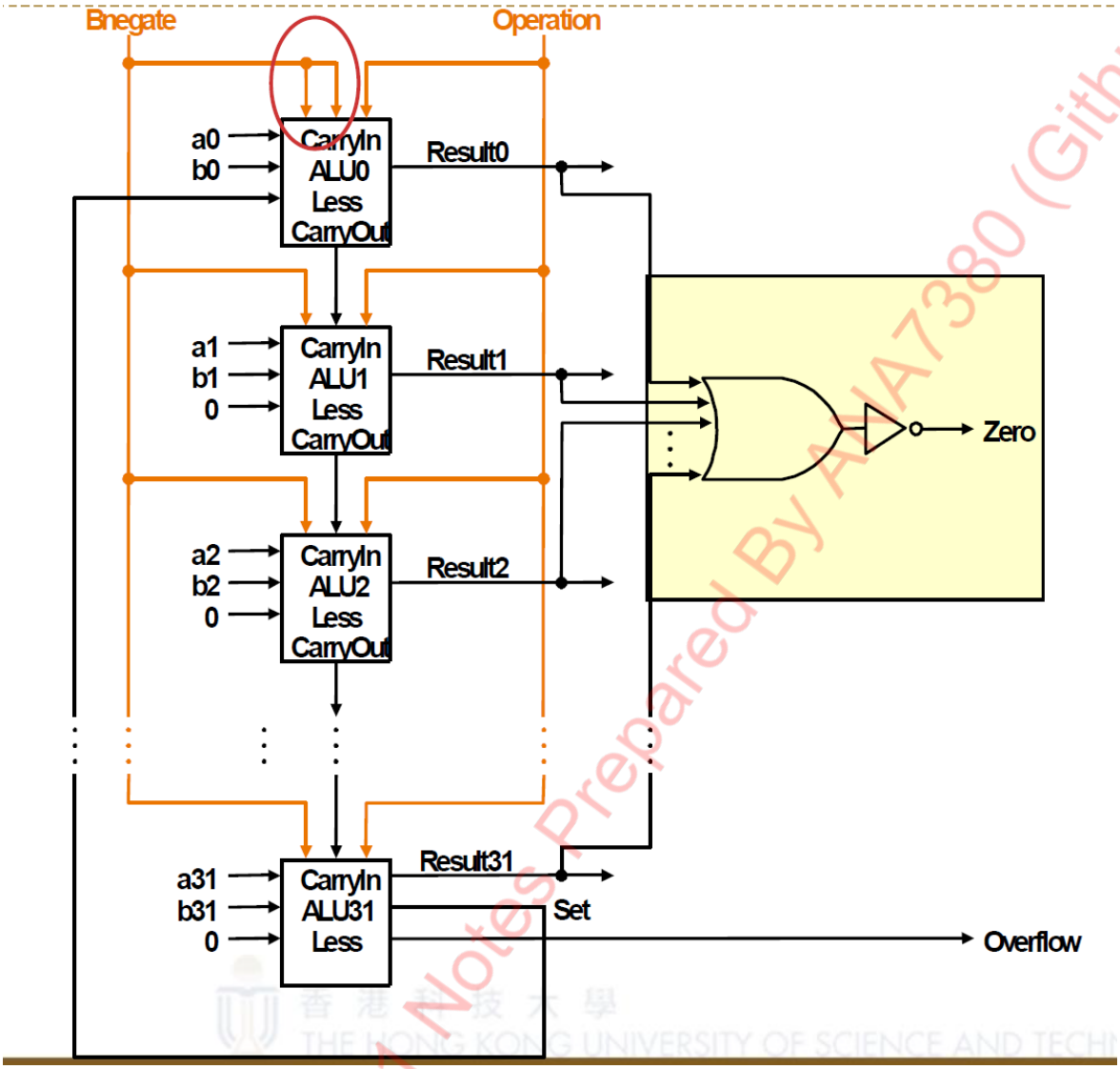

  - The Operation selector needs 2 bits.
- 32 Bit ALU is formed by assembling 32 1-Bit ALU into one large component.
  - The carry in of the current ALU is the carry out from the previous ALU, and the current carry out is passed to the next ALU's carry in.

    - The last ALU need to handle overflow checking.
- Subtraction in ALU:
    - We need to invert all bits and add 1.
    - Invert: Use a 2:1 multiplexor (B invert), one path is original bit and the other path is the inverted bit using NOT gate.
    - Add 1: By setting 1 to the first ALU's carry in, it achieves the add 1 effect (first carry in must be 0 in normal cases)
- `slt` in ALU:
    - It is equivalent to perform subtraction and check the MSB of the result. If 0, A < B
        - If 1, A > B
    - The destination register will have 31 zeroes and the LSB being determined by subtraction.
      - To implement this, the first ALU (LSB) will accept the adder result from the last ALU (MSB) via Set
      - `slt` is added to the multiplexor input lines. When Multiplexor selects Less, then Result0 will tell if its less than or not.



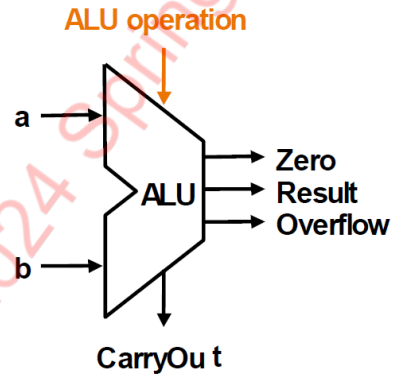

- `beq` in ALU:
    - Equivalent to `(A-b)==0?`

- Performs subtraction and feed all the Results into an OR gate. The or gate is connected to a NOT gate to indicate true if all zeroes, and false for at least one non-zero.




- If subtraction, then `Bnegate` and `carryin` in ALU0 will be 1!

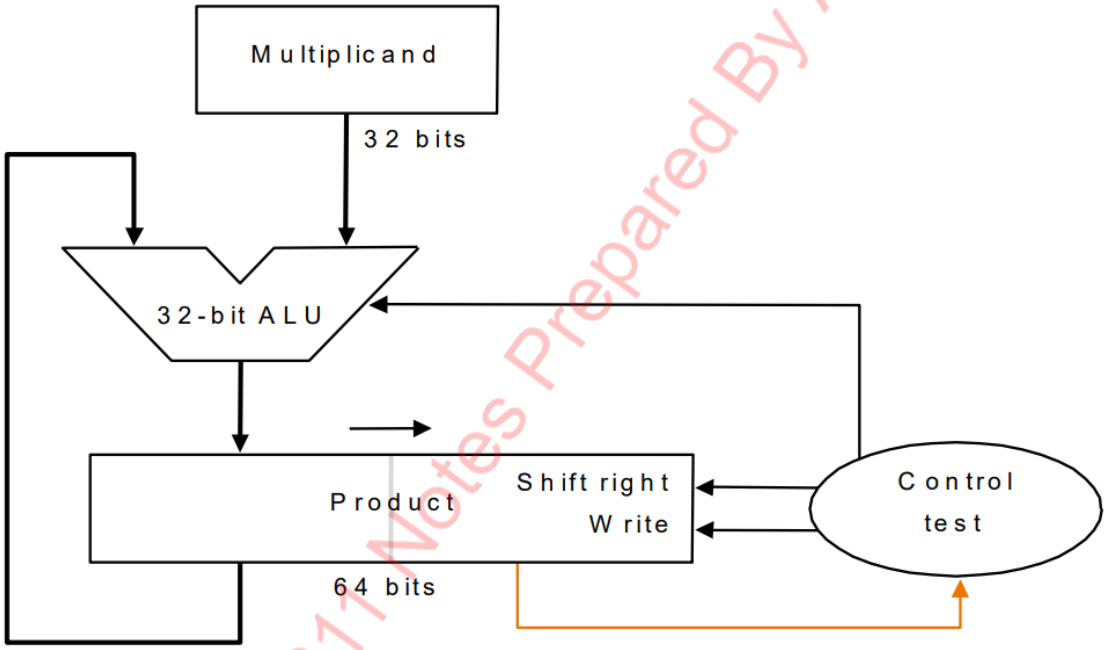
- Ripple carry ALU: The ALUs are connected such that they wait for the previous ones to output result before itself can start to execute calculation.
  - If each ALU take time $t$ to do calculation, then the ith ALU need to wait $(i-1) \times t$ before it computes.
  - Slow!
- Universal Representation of 32 bit ALU:




| ALU Control lines | Operation |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | ADD |
| 110 | SUB |
| 111 | SLT |

# Multiplication

- If a m bit number times a n bit number (Both unsigned), the maximum bits required to store the result will be m + n bits.
  - In MIPS, require 64 bits to store result.
- Optimized multiplication algorithm:
  - Multiplicand: 32 bits.
  - Product: 64 bits, where the upper 32 bits are zeroes, and the lower 32 bits are the multiplier
  - Use the LSB of Product to times with multiplicand, then add the results to the upper 32 bit window via 32 bit ALU.
  - Shift the Product to right by 1 bit. Repeat 32 times. **(Recursion time depend on the bits of multiplier)**




- Example:

**Multiplication of two 4-bit unsigned numbers (0110 and 0011)**

Multiplier

| Iteration | Multiplicand (M) | Product (P) | Remark |
|---|---|---|---|
| 0 | 0110 | 0000 0011 | Initial state |
| 1 | | 0110 0011<br>0011 0001 | Left(P) = Left(P) + M<br>P = P >> 1 |
| 2 | | 1001 0001<br>0100 1000 | Left(P) = Left(P) + M<br>P = P >> 1 |
| 3 | | 0100 1000<br>0010 0100 | No operation<br>P = P >> 1 |
| 4 | | 0010 0100<br>0001 0010 | No operation<br>P = P >> 1 |

- Hardware Speedup:
  - By using 31 adders instead of using one single adder 32 times, the delay to multiply decreases by 5 times.
- If the multiplicand or multiplier is negative, Convert it to positive number, and do the normal multiplication. At last negate the result to get the negative version.
- Multiplication in MIPS:
  - A pair of register `Hi` and `Lo` are used to store the product (because 32 bits are not enough to store product)
    - The two registers are read-only
  - `mult $r1, $r2` (Treat them as signed)
    - `$r1 x $r2` and store it in `Hi` and `Lo`
    - It ignores overflow (because the max value of result must be smaller than $2^{64}-1$)
  - `multu $r1, $r2` (Treat them as unsigned)
    - `$r1 x $r2` and store it in `Hi` and `Lo`
    - Ignore overflow
  - To move data from `Hi` and `Lo` into general registers, use `mfhi $dest` (move from high) and `mflo $dest` (move from low)
- Unsigned Overflow (in terms of data representation)
  - Occurs when the upper 32 bits (`Hi`) is not equal to zero (Likely to happen)

```
multi $s0, $s1
mfhi $s2
mflo $s3
beq $s2, $zero, no_overflow
j overflow
```

- Signed Overflow:
  - Occurs when the upper 32 bits is not equal to Sign Extended of MSB of Lo

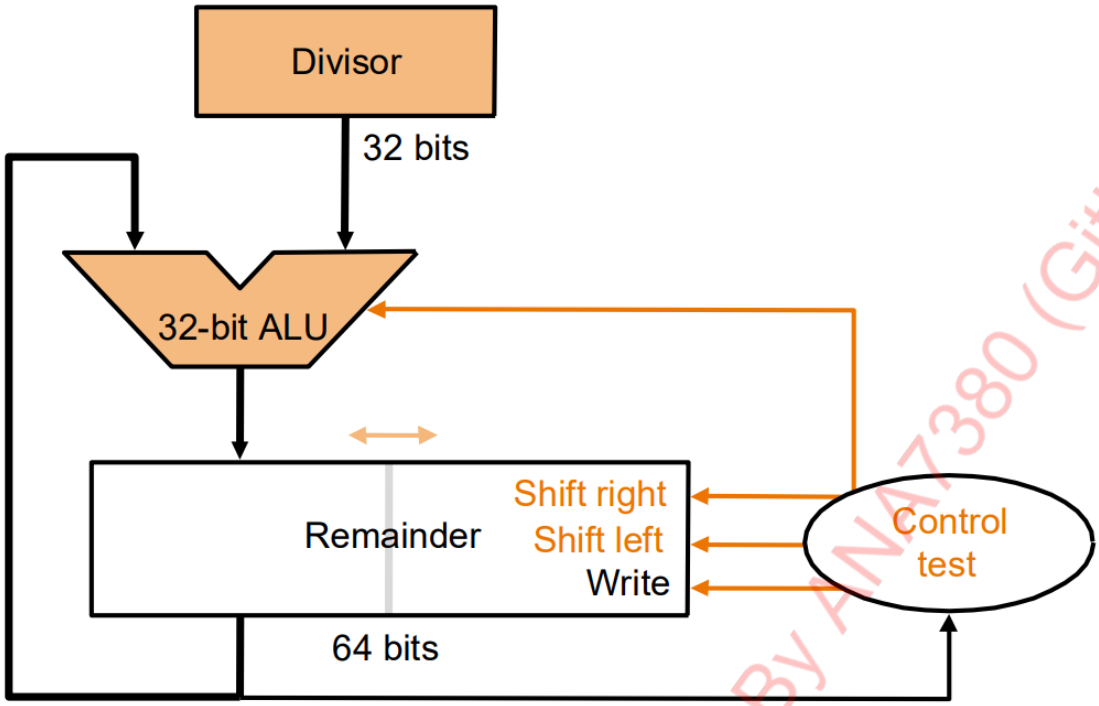
```
multi $s0, $s1
mfhi $s2
mflo $s3
sra $t0, $s3, 31 # extend the MSB of Lo to right 31 bits
# Now $t0 = all sign bits
beq $t0, $s2, no_overflow
# If no overflow, the Hi register will be all 0 or all 1
j overflow
```

- `sra`: Shift right arithmetic:

- `sra $dest, $source, bits` similar to shift right logical, instead of zero extension, it performs sign extension to the right.
- `sla` does not exist.

# Division

- Optimized Version:
  - Remainder: 64 bits, lower 32 bits store dividend and upper all zero
  - Divider: 32 bits
  - First shift left the whole remainder to left by 1 bit.
  - Try to minus the divisor from the left half of the remainder. If result is negative, undo, Shift left by 1 bit and set the LSB to 0
    - If result is positive, shift right by 1 bit and set LSB to 1.
    - Repeat for 32 times.
  - At the end we will need to do correction (ONLY Left 32 bits shift right by 1 bit) then the remainder will be the upper 32 bits and quotient be the lower 32 bits.




**Division of a 4-bit unsigned number (0111) by another one (0011)**

| Iteration | Divisor (D) | Remainder (R) | Remark |
|---|---|---|---|
| 0 | 0011 | 0000 0111<br>0000 1110 | Initial state<br>R = R << 1 |
| 1 | | 1101 1110<br>0000 1110<br>0001 1100 | Left(R) = Left(R) – D<br>Undo<br>R = R << 1, $R_0$ = 0 |
| 2 | | 1110 1100<br>0001 1100<br>0011 1000 | Left(R) = Left(R) – D<br>Undo<br>R = R << 1, $R_0$ = 0 |
| 3 | | 0000 1000<br>0001 0001 | Left(R) = Left(R) – D<br>R = R << 1, $R_0$ = 1 |
| 4 | | 1110 0001<br>0001 0001<br>0010 0010 | Left(R) = Left(R) – D<br>Undo<br>R = R << 1, $R_0$ = 0 |
| extra | | 0001 0010 | Left(R) = Left(R) >> 1 |

Remainder ↳ correction Quotient

- Signed Division:
  - If the signs of the divisor and dividend are different, then the quotient should be negated
  - If the remainder is nonzero, then its sign should be the same as that of the dividend

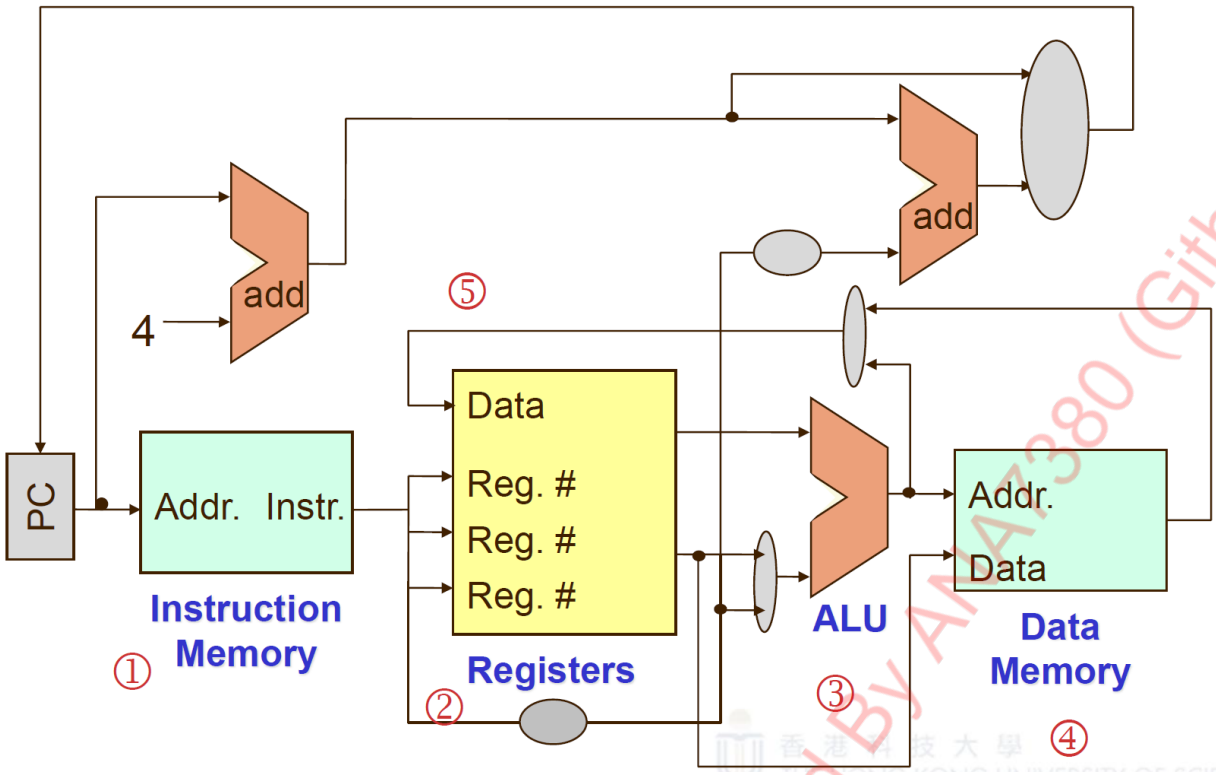
| Dividend | Divisor | Quotient | Remainder |
|---|---|---|---|
| +7 | +2 | +3 | +1 |
| −7 | +2 | −3 | −1 |
| +7 | −2 | −3 | +1 |
| −7 | −2 | +3 | −1 |

- Division in MIPS:
  - `div`: Consider the registers as signed
    - Storing 32 bit numbers into register: `lui` then `ori`
    - Result: low: Quotient, high: Remainder, use `mfhi` to get remainder and `mflo` to get quotient
  - `divu`: consider the registers as unsigned

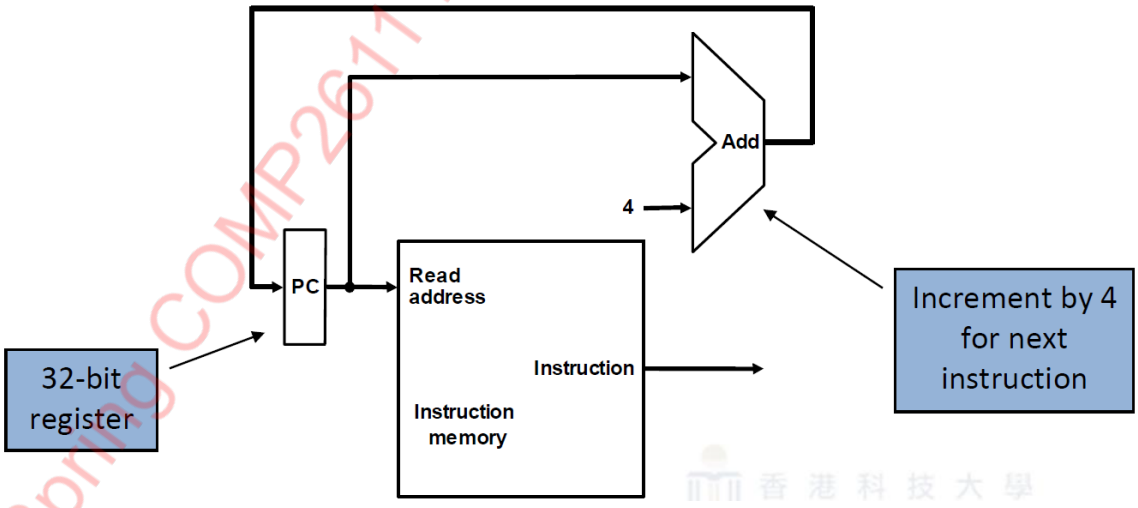
# 6. Processor

- MIPS Processor in **single-cycle implementation**: Only one instruction is executed within one cycle, all other instructions need to wait for their turns.
  - The time for each clock must allow the slowest instruction to finish.
    - Slowest: `lw/sw` etc
- Instruction execution flowchart:

1. **Fetch instruction** from memory location indicated by *program counter*
2. **Decode the instruction** and prepare the operands
   - Also prepare the Control signals to allow read/modification etc
3. Perform operation using ALU
4. Memory access: Only instructions such as `lw` and `sw` require to execute this part
5. Write the result into the destination register, update PC (increment by 4 or overwrite PC to branch target)



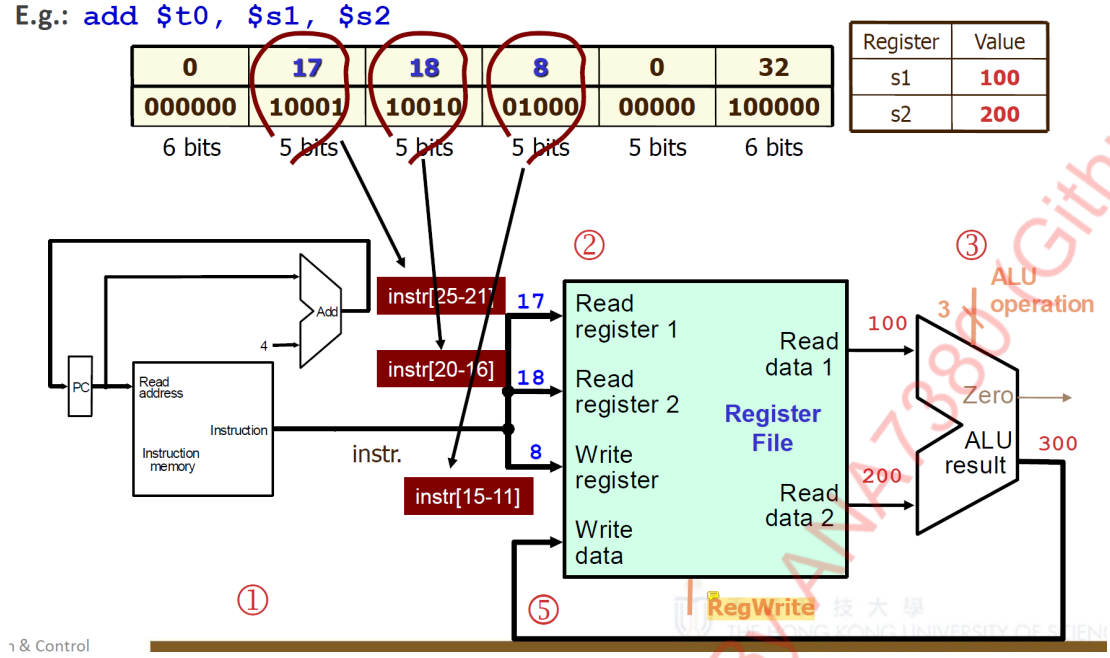

- The PC counter adder is a D flip flop, the effect of PC+4 happens in falling-edge
  - The second adder is responsible for branching conditions, it will calculate how many bytes away from PC+4 is the target address
- Datapath: Elements that process data and addresses in the CPU. e.g. ALU, registers, multiplexor, memories
- Instruction Fetch:
  - Instruction memory: A memory unit shat stores the instructions of a program.
  - Output the 32 bit instruction




- Operations for Different Types of Instructions:
  - R type:
    1. Read two register operands
    2. ALU performs arithmetic/logical operation (Different R type instruction differs in this step only)

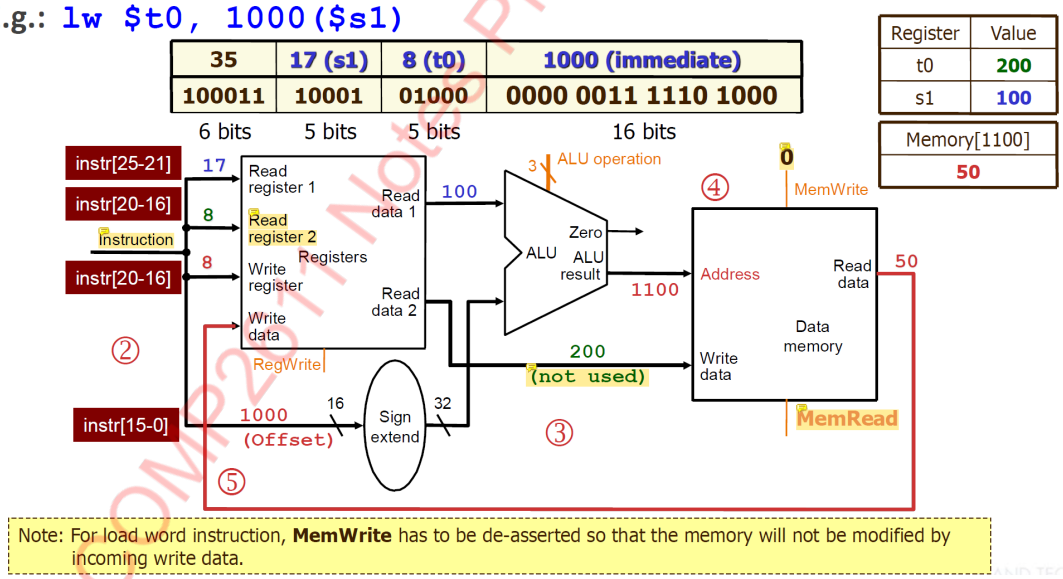
3. Write register result

E.g.: `add $t0, $s1, $s2`




- I type: `sw` and `lw`:
  1. load the base address
  2. Sign extend the immediate offset to 32 bits
  3. ALU adds the two result to find the target address
  4. Write/Save data to register/memory

E.g.: `lw $t0, 1000($s1)`

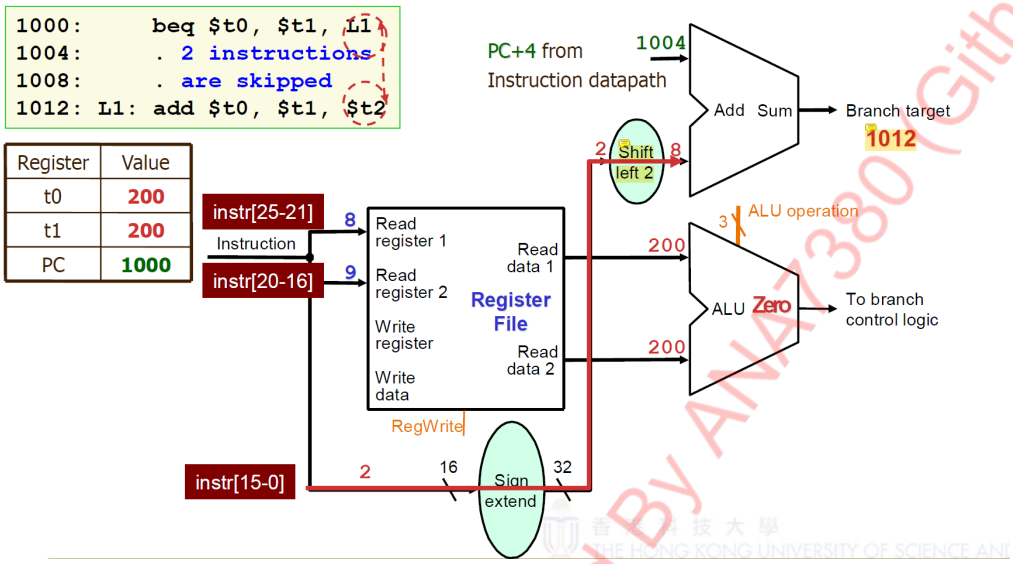



Note: For load word instruction, **MemWrite** has to be de-asserted so that the memory will not be modified by incoming write data.

- In this diagram, `$t0` is supplied to both *read register 2* and *Write Register*. When `sw` is called, we need to store value in `$t0` to data memory.
  - Write Register port will have undetermined values
- When `lw` is called, we need to find the value in the target memory and pass it to *Write data* port.
  - `RegWrite` allows writing to register
  - `MemWrite` allows modifying memory
  - `MemRead` prevents unrelated operations reading invalid memory.
- I type: `beq`:

1. Read registers
2. Subtract operands and check the **ZERO** output
3. if True, calculate the relative address between PC+4 and target address

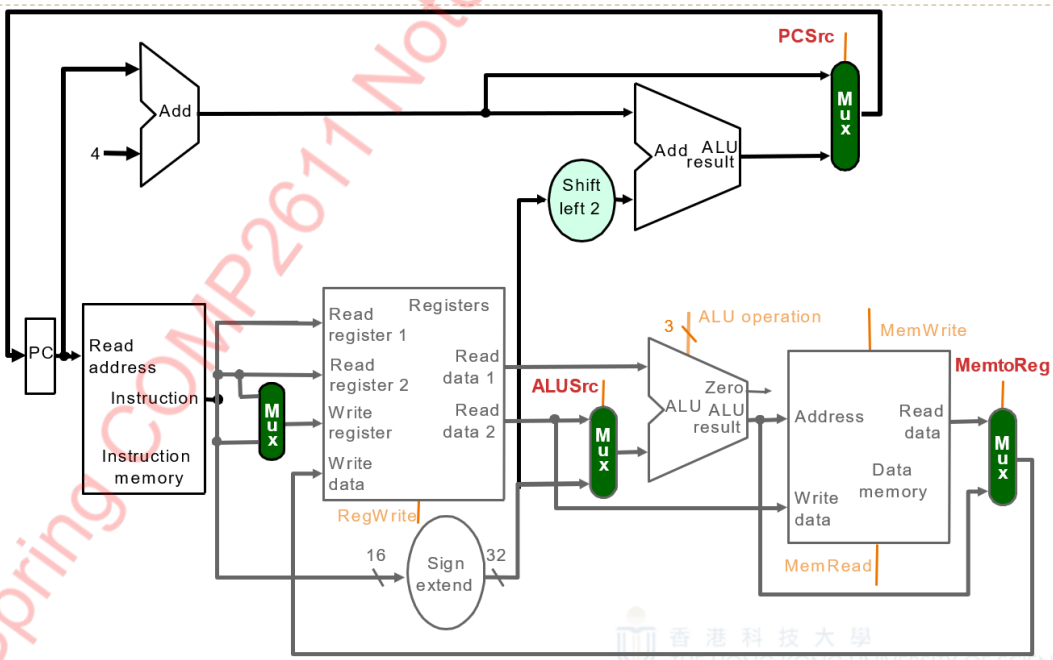
E.g.: `beq $t0, $t1, 2`




- The immediate field stores the number of instructions (words) away from the next instruction (PC+4) to the target instruction.
- It will perform `sll` to convert word to bytes.
- If `beq` returns true, the PC will be modified to the branch target.

- This implementation is simple, but unrealistic (Inefficient).

## Full Datapath: Muxing Two Possible Destination Registers

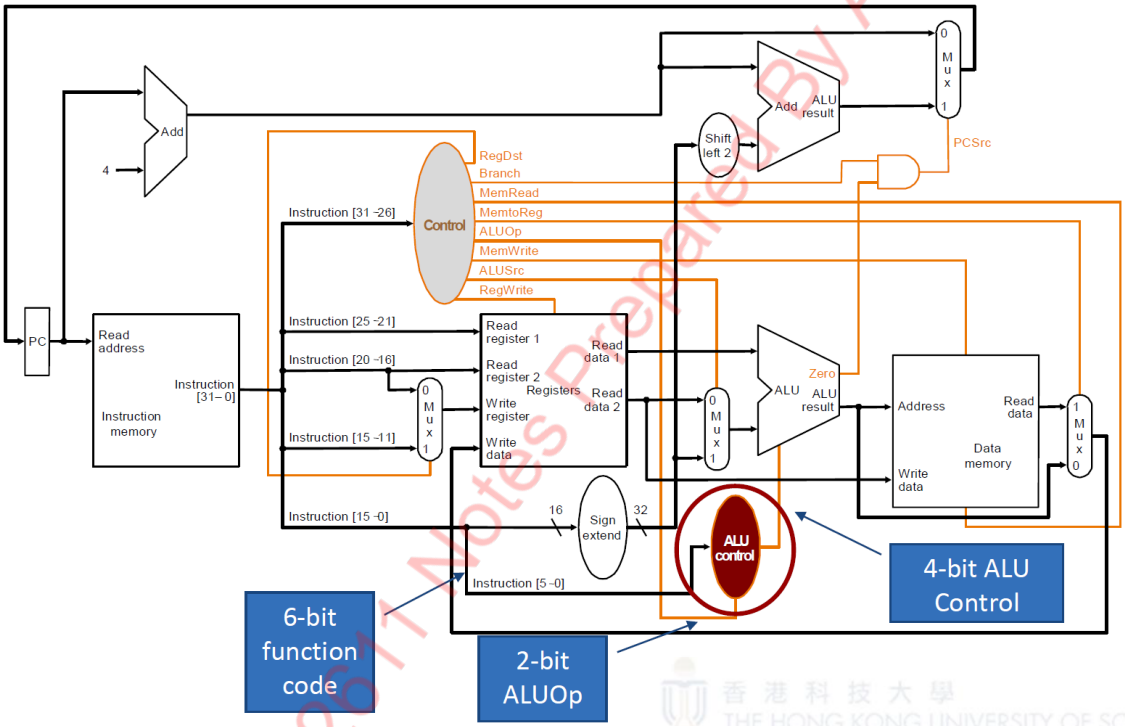



- Need to understand the path of:
  - R type instruction
  - `lw` and `sw`
    - Base address passed to Read Register 1 and data passed to Read register 2 AND Write register

- `beq`

# Single Cycle Control

- Two controls in the data path:
  - Big control: Handles instructions other than R type instructions.
    - When the opcode is supplied to the Big control, it will generate correct config that needs to be fed to ports requiring the control signal.
      - e.g. RegWrite, MemWrite, MemRead, ALUSrc, MemtoReg, PCSrc, RegDst
    - In R type instructions, all opcodes will be 0, it will output a `ALUOp` to the ALU control.
  - ALU control: Handles instructions for R type instructions.




- ALU is used in R type instructions, `lw/sw`, and `beq`
- ALU control:
  - First method: pass the opcode to big control, then big control generate a two bit ALUOp to pass to ALUControl. At the same time, the 6 bit function code is passed to ALUControl.
    - Total 2+6 = 8 bits of input
  - Second method: Pass the 6 bit opcode and 6 bit function code to the ALUControl, total 12 bits of input.
  - The first method (2 level decoding) is faster ($2^8 < 2^{12}$})
- ALUOp:
  - `00`: select addition for ALU output
  - `01`: select subtraction for ALU output

- `10`: R type instructions, need to check function code, and output the correct selection.
- `11`: Invalid

- Implementing ALU Control:

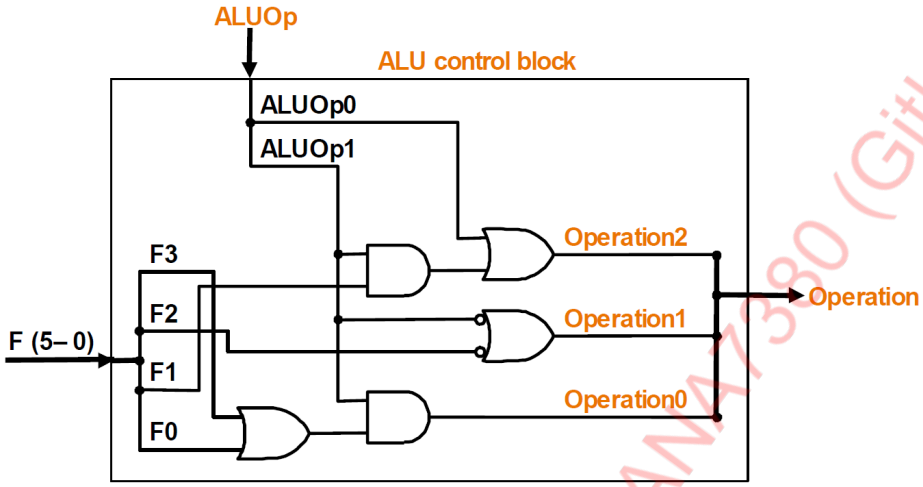
| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|---|---|---|---|---|---|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract | 100010 | subtract | 0110 |
| | | AND | 100100 | AND | 0000 |
| | | OR | 100101 | OR | 0001 |
| | | set-on-less-than | 101010 | set-on-less-than | 0111 |

input input output

| ALUOp | | Function code | | | | | | Operation | |
|---|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | | |
| 0 | 0 | X | X | X | X | X | X | 0010 | lw, sw |
| X | 1 | X | X | X | X | X | X | 0110 | beq |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 0010 | R-type Instr. |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 0110 | |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 0000 | |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 0001 | |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 0111 | |

- Cross outs:
- In ALUOp, there is no `11`, therefore if one if given as 1, the other must be 0. (simplify minterm)
- In Function code, if the instruction is not R type, then no need to examine the function code
- If the instruction is R type, observed that the first 2 digits are same, therefore also cross out,
- When Big Control outputs `10` to ALU Control, it will look up the function code to generate corresponding operation code to feed to ALU

# Hardware Implementation of ALU Control Block

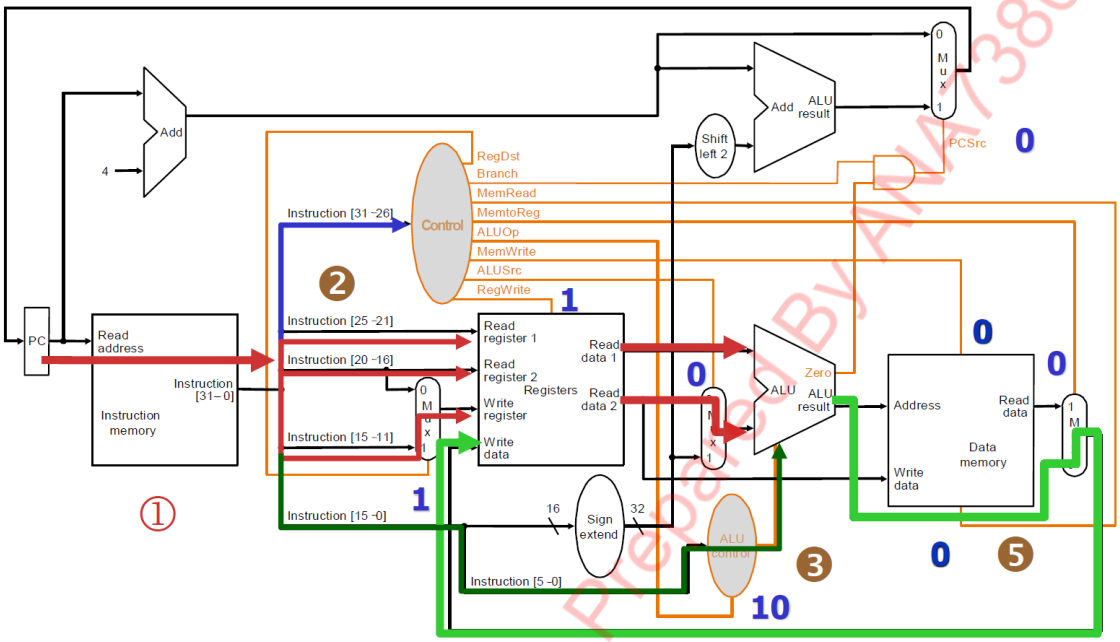



| Signal name | Effect when deasserted | Effect when asserted |
|---|---|---|
| RegDst | The register destination number for the Write register comes from **rt** field (bits 20-16) | The register destination number for the Write register comes from **rd** field (bits 15-11) |
| RegWrite | None | Enable data write to the register specified by the register destination number |
| ALUSrc | The second ALU operand comes from the second register file output (Read data port 2). | The second ALU operand is the sign-extended, lower 16 bits of the instruction |
| PCSrc | The next PC picks up the output of the adder that computes PC+4 | The next PC picks up the output of the adder that computes the branch target |
| MemRead | None | Enable read from memory. Memory contents designated by the address are put on the Read data output |
| MemWrite | None | Enable write to memory. Overwrite the memory contents designated by the address with the value on the Write data input |
| MemtoReg | Feed the Write data input of the register file with output from ALU | Feed the Write data input of the register file with output from memory |

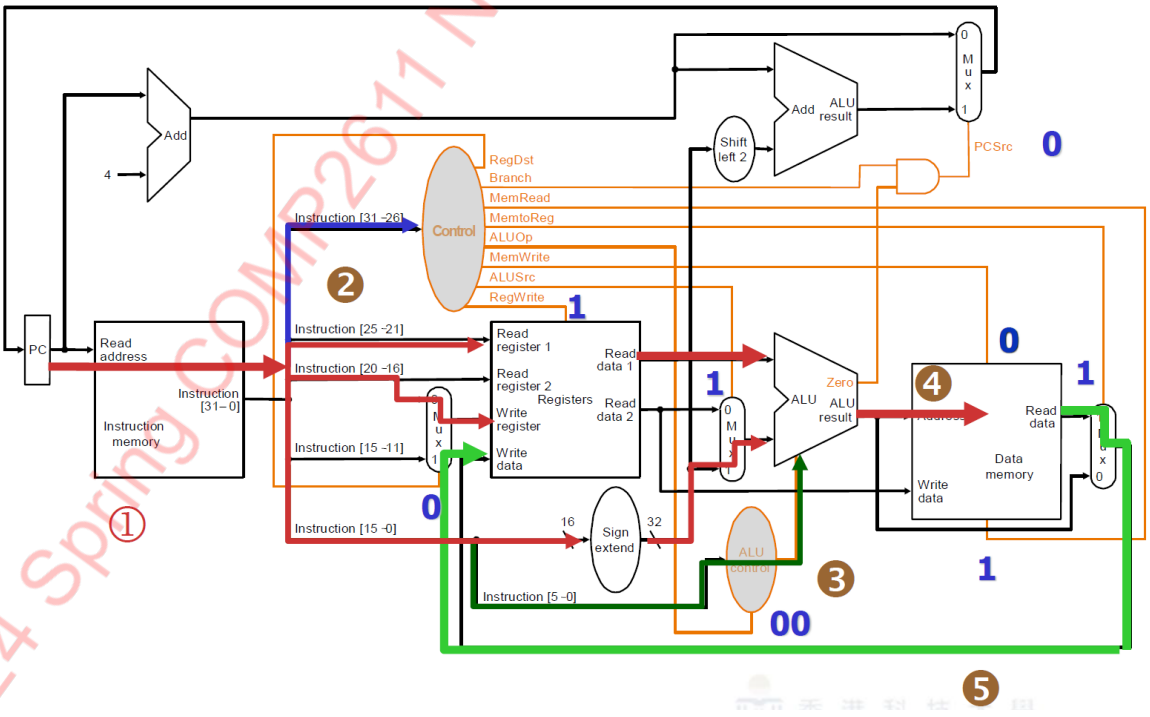
- RegDst: selecting which register is the destination register
- RegWrite: allows writing to the destination register
- ALUSrc: select second operand for ALU, either sign extended immediate value(I type, offset) or value from register(R type)
- PCSrc: The control Branch (Whether performing `beq` operations) AND zero output from ALU decides the value for PCSrc
- MemRead: Controls the access to memory for reading data, if 0 meaning the instruction(e.g. R type) executing is not a instruction that needs to look into the memory, prevents accessing invalid location
  - Only `lw` requires MemRead to asserted

- MemWrite: Only `sw` will require writing to memory, will not mess up the memory
- MemToReg: Decide whether the calculation result from ALU(R type) or the data read from the memory( `lw` ) is passed to modify the register
- All control signals can be determined based on the opcode, except the PCSrc.
  - PCSrc is determined by opcode `beq` AND zero output be true.

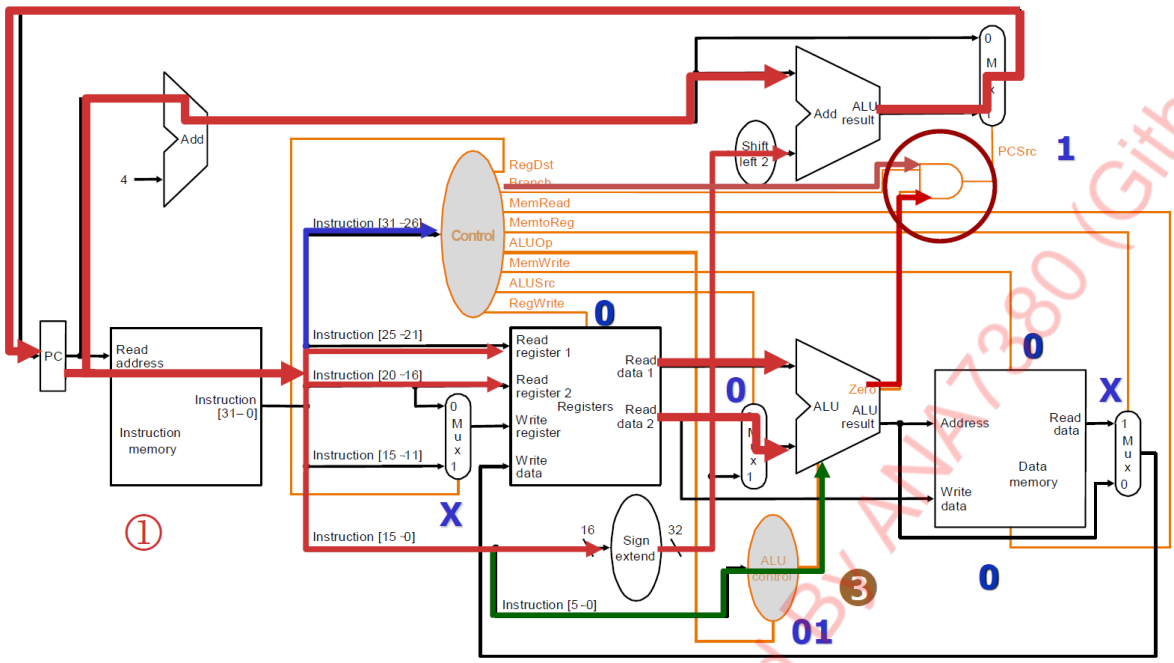
# R-Type Instr. with Control




# Load Instr. with Control



# Branch-on-Equal Instr. with Control




- The X in the `beq` data path represents any value can be used as it does not affect the calculation
- `beq`'s third operand is using relative address jumping, it holds the number of instructions to jump from the **NEXT** Instruction to the target instruction, and by sign extension + shift left logical, the PC ALU will calculate the amount of bytes need to jump if `beq` is true.
- Implementing the data path with truth table:

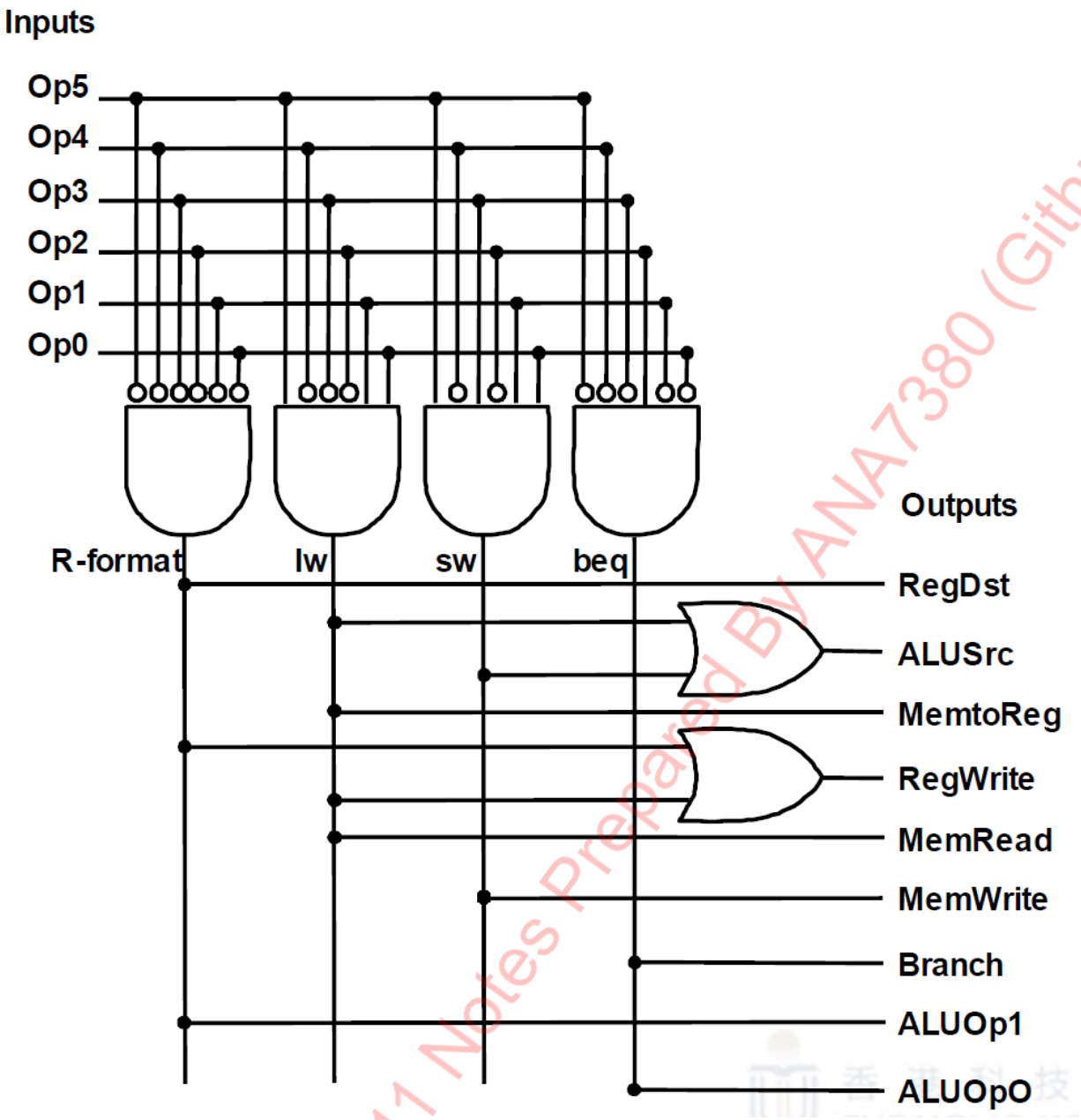
### Setting of control lines (output of control unit):

| Instruction | Reg-Dst | ALU-Src | Mem-toReg | 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 |

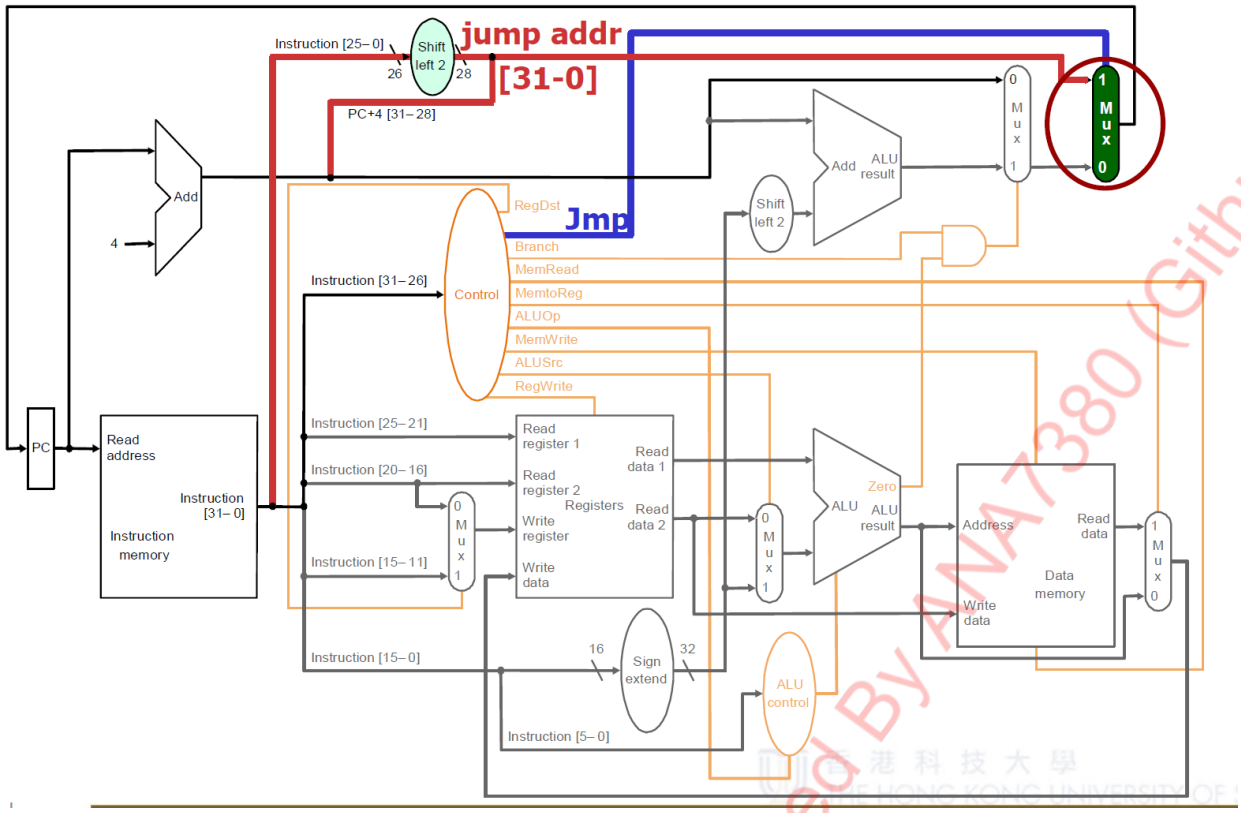
`sw` & `beq` will not modify any register, it is ensured by making RegWrite to 0
So, we don't care what write register & write data are

### Input to control unit (i.e. opcode determines setting of control lines):

| Instruction | Opcode in decimal | Opcode in binary | | | | | |
|---|---|---|---|---|---|---|---|
| | | Op5 | Op4 | Op3 | Op2 | Op1 | Op0 |
| R-format | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| `lw` | 35 | 1 | 0 | 0 | 0 | 1 | 1 |
| `sw` | 43 | 1 | 0 | 1 | 0 | 1 | 1 |
| `beq` | 4 | 0 | 0 | 0 | 1 | 0 | 0 |




- Jump Instruction `j`:
  - 6 bits of opcode + 26 bits of target address
  - When `j` instruction is used, it will **Update the PC** to the following 32 bits branch target address:
    - `PC+4[31:28] 26Bits 00`

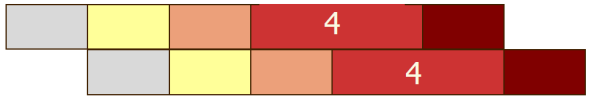



- For Single Cycle data path, instructions are slow due to every instructions are given a long enough time to allow the slowest instruction to finish executing.

# Pipelined Control

- Instead of one instruction occupies 5 stages and most hardware are waiting, it is better to feed the next instruction into the idling hardware that the current instruction finished using.
  - Multiple tasks can be processed simultaneously
  - The tasks should be independent of each other
  - The time for each instruction did not change, it is only utilizing the idling hardware.
    - Improves entire workload
- The potential speedup of all instructions is equal to the number of pipeline stages
  - e.g. MIPS pipelined control will be at most 5 times faster than single control. (Best performance when each stage use same time)
  - Instead of giving a fixed amount of time to allow slowest instruction to finish executing through 5 stages, now each stage is assigned an amount of time that

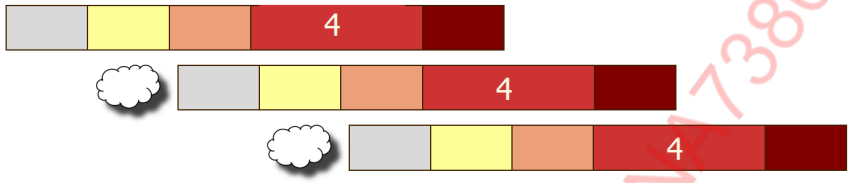
allows the slowest instruction to complete that stage.




**Can I align the pipeline stages as above?**

**Answer: NO, because the tasks executing in parallel are not independent (task 4 overlaps task 4)**

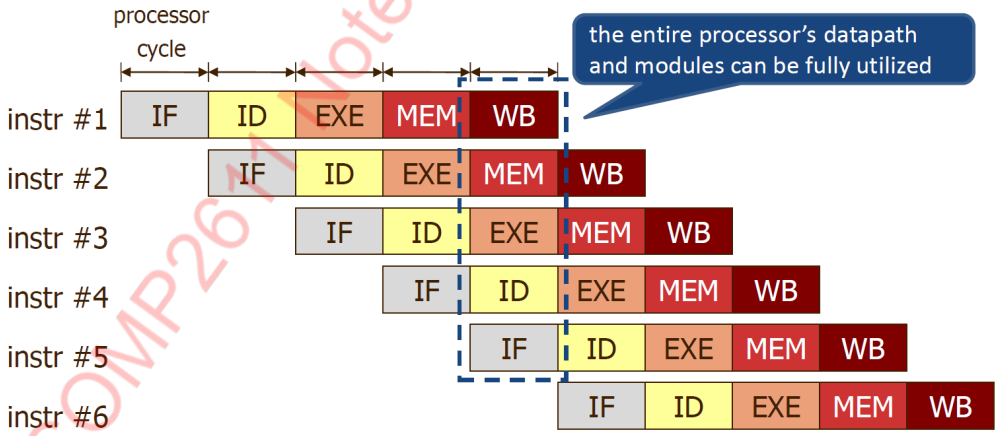
**The condition to align is to make sure NO OVERLAP of any stages?**



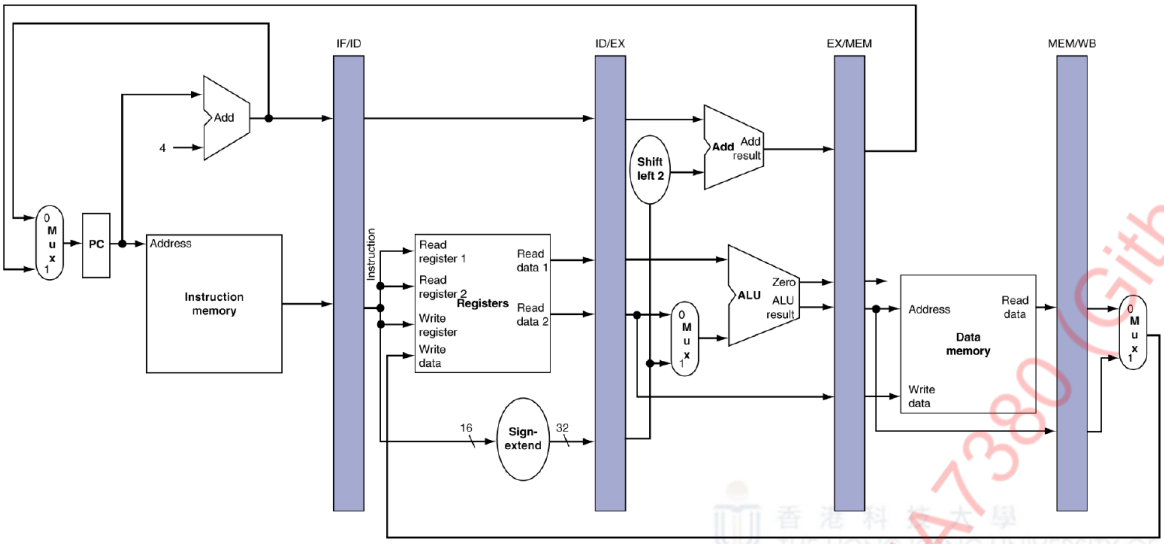

- MIPS is suitable for pipelining because:
  1. All instructions are 32 bits, can be done in one fixed clock cycle
  2. Only a few similar instruction formats, so each formats take approximately same time to complete
  3. Memory operands only appear in load and store
  4. Alignment of memory operands, each operand's starting address can be divisible by 4, can be read within one fetch

**In The Processor: Datapath & Control,**

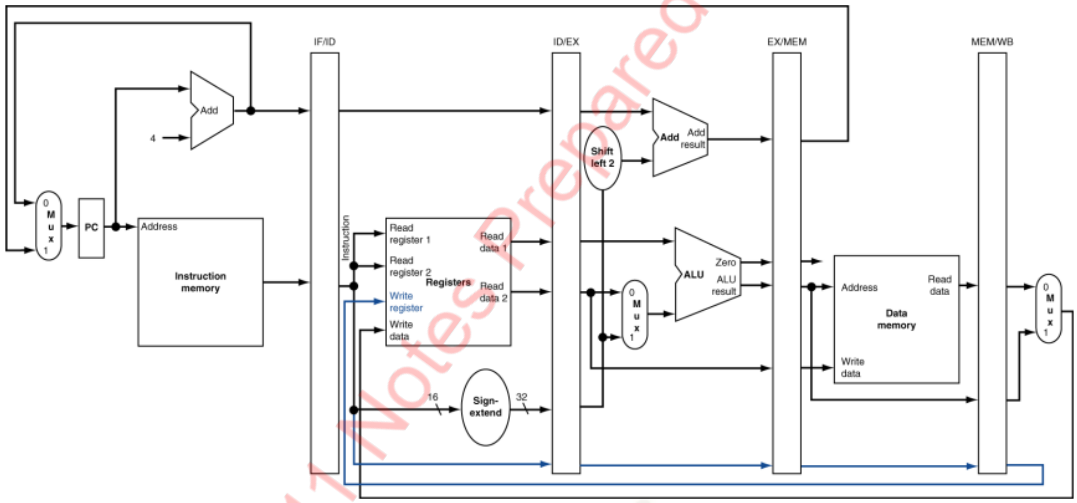
- **Each instruction takes multiple steps**
- **Each step is independent of each other and takes different datapath**




- **Additional Pipeline Registers** (In D flip flops) are required between each stages to hold information produced in the previous cycle.
  - When falling edge, the values are written into the registers and allows another set of data to be fed into that stage




- Issue with this pipelined stages: In the **Write Back** stage of `lw`, the value returned to Stage 2 will potentially write to the wrong **Write Register** for other instructions currently performing Instruction decode.
  - Solution: Pass the write register data along to the end

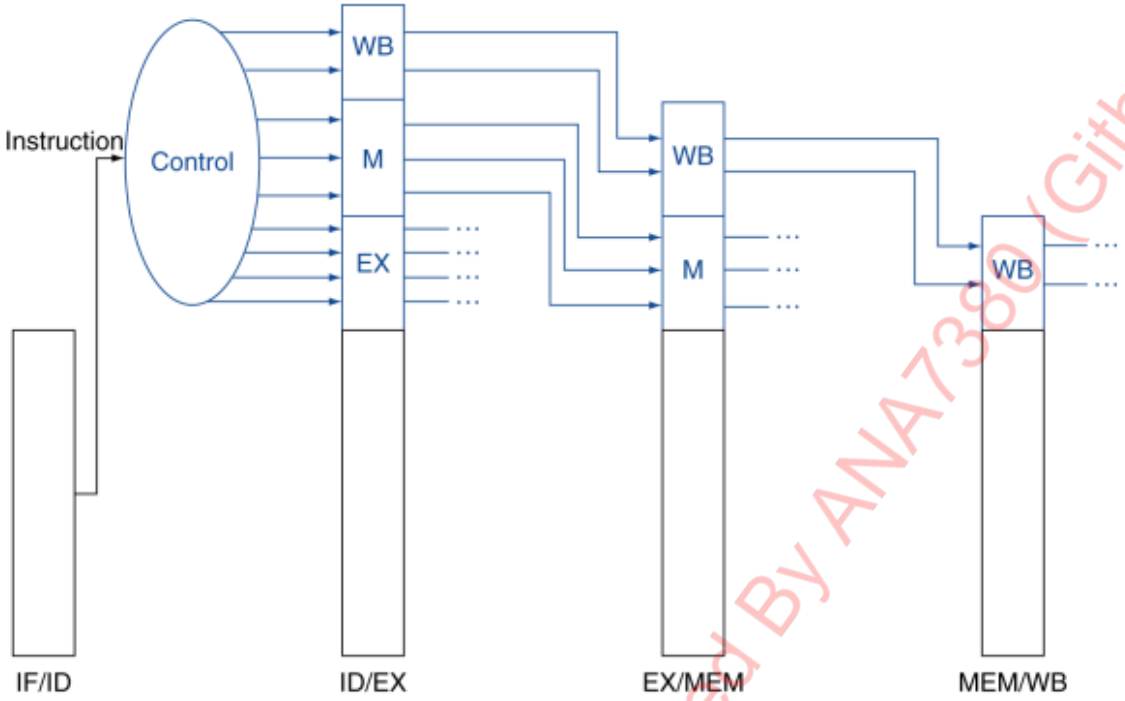


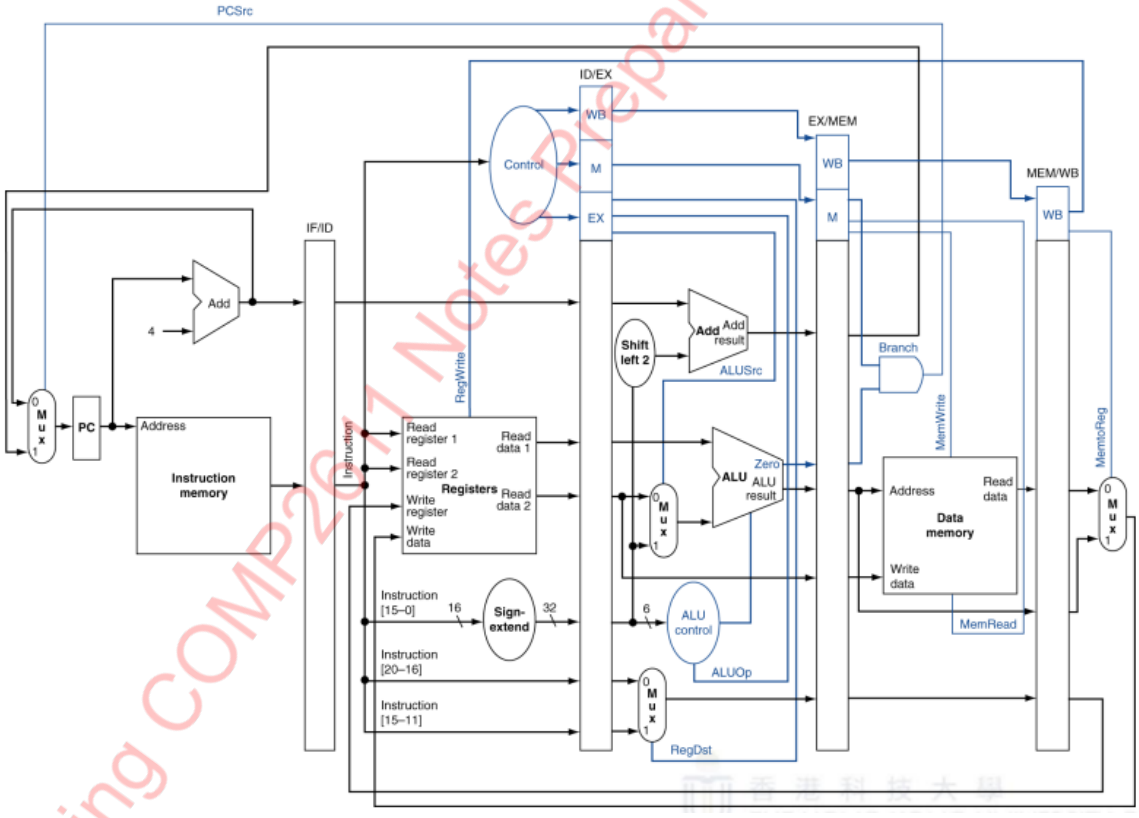

- Control signals for each stage:
  1. Instruction fetch: No control signal required
  2. Instruction decoding: No control signal required
  3. Execution: Require `RegDst, ALUOp, ALUSrc`
  4. Memory: Require `Branch, MemRead, MemWrite`
  5. Write Back: Require `MemToReg, RegWrite`

| Instructions | EX | | | | MEM | | | WB | |
|---|---|---|---|---|---|---|---|---|---|
| | RegDst | ALUOp1 | ALUOp2 | ALUSrc | Branch | MemRead | MemWrite | RegWrite | MemToReg |
| R-format | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

- When the control signals are generated in stage 2, the signals that are needed will be passed to the pipeline register.

- The signals that will not be used later (Or used in current stage) will not be passed to the pipeline register.



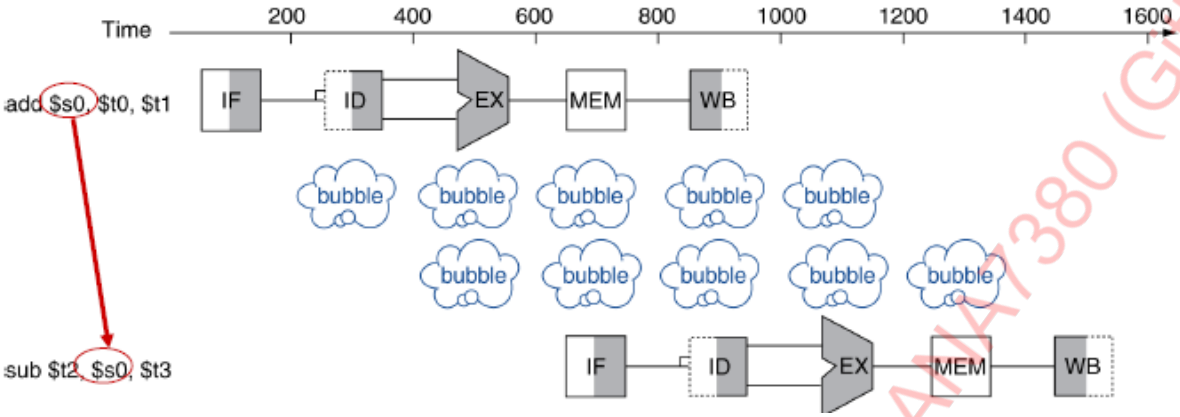




# Pipeline Hazards

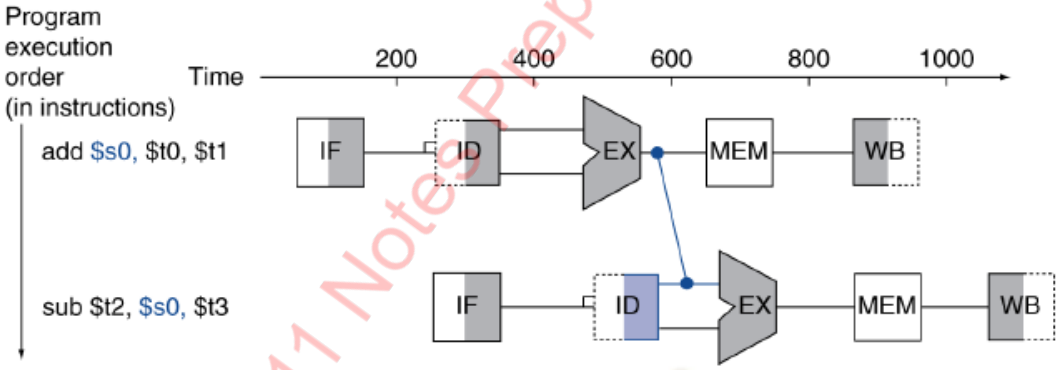
- Hazards: Situations in pipelining when the next instruction cannot be executed in the following clock cycle (Need to wait)
- **Structural hazards**: A required hardware is busy
  - Won't happen in MIPS because for instruction memory and main memory are separated.

- Also the write process are set to happen in the upper half of the cycle, and read process happen in the lower half of the cycle, due to extreme fast registers

- **Data hazards**: Previous instruction not yet written to registers and current instruction need to read data from that register

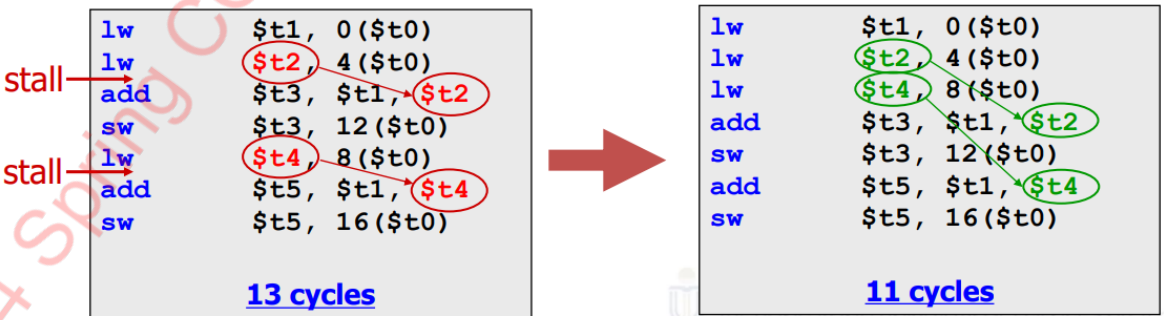



  - Solution: **forward** ALU calculation result from previous instruction directly to current read register stage to overwrite the incorrect data
    - The data should be used in the next time cycle.
    - For `lw` instruction, the data is only read after `MEM` state, so it requires one additional bubble time compared to R type instructions




  - Another solution: MIPS compiler tries to find any unrelated instruction that can be filled and does not change the logic during waiting

Assume a to e are stored in memory address `0($t0)`, `4($t0)`, `8($t0)`, `12($t0)` and `16($t0)` respectively. Assume **forwarding is used**.

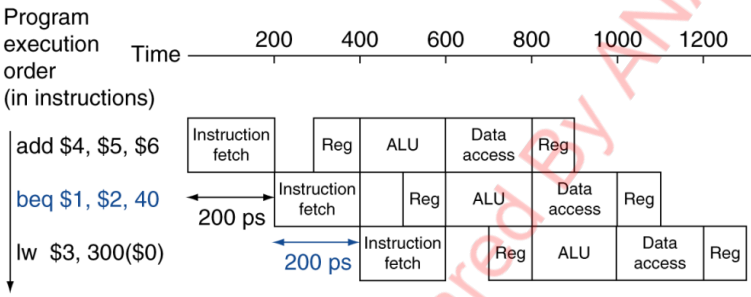



    - *Stall*: Delay time slot
  - Or delaying

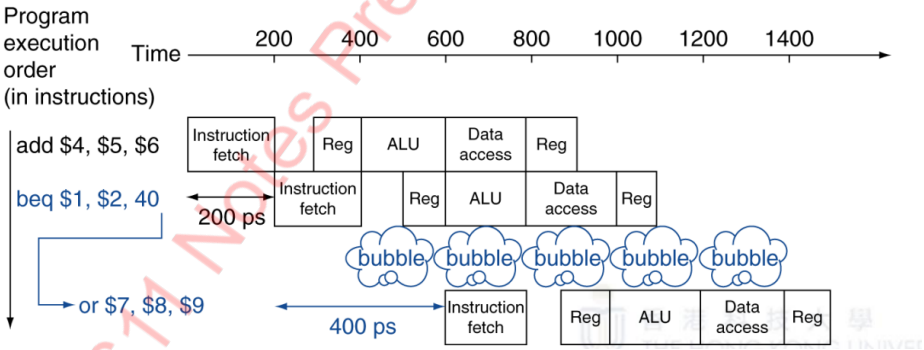
- **Control hazards**: Decision branching depends on previous instruction

- Need to wait until ALU's ZERO Output (If no optimization, the `IF` stage should start at t=800 → 2 bubbles)
- Solution: For `beq`, add additional hardware to perform bitwise AND during stage 2 decoding.
- Or predict branching. We always predict no branching taken.
    - If the comparison from the **additional hardware** indicate no branching, continue executing current instruction
    - If the comparison from the **additional hardware** indicate need to branch, then set all the control values to zero (in current instruction) and jump to the label via `PCSrc`

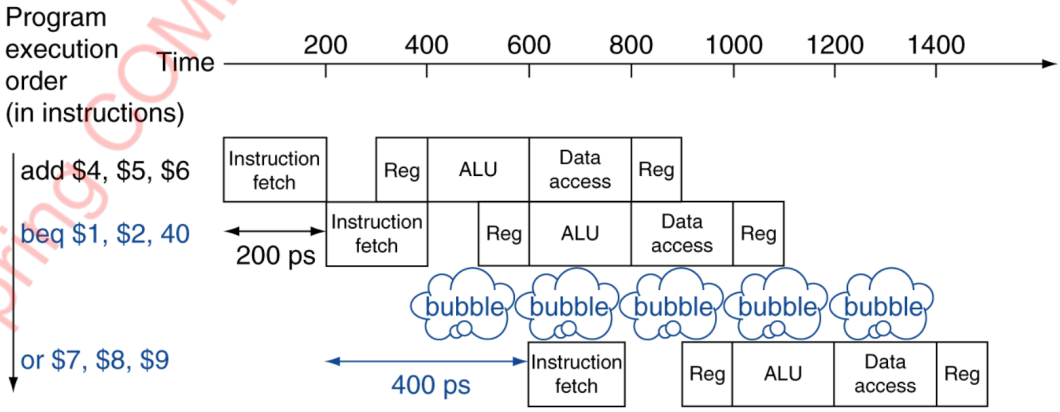
Prediction correct




Prediction incorrect




- Note: The above diagram is after fetching the 3rd instruction and found the second instruction should branch, therefore erasing the 3rd instruction




- Or delaying

- All hazards can be solved by waiting.

# Concluding Remarks

- Pipeline allows more instruction to execute in a given time compared to single cycle control due to utilization of different hardware.
  - Need careful design and additional registers to store results between each stages.
  - Need registers to forward control signals down the pipeline.

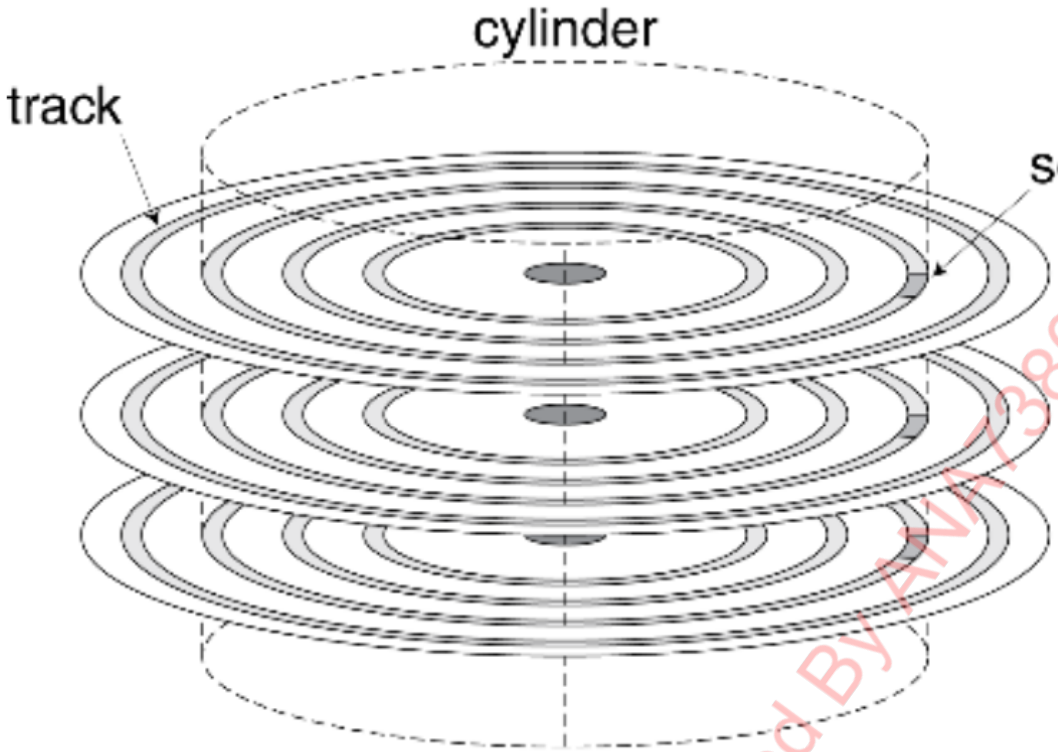
# 7. Memory Hierarchy

- Register is the fastest memory (Inside CPU)
- Cache is the second fastest memory (Inside CPU)
- Main memory is the slowest memory inside CPU
- Hard disks, CD, USB are the slowest (Outside CPU) **Not covered**

## Memory Technology

- RAM (Random Access Memory) composed of two types:
  - Static RAM (SRAM): Used in cache, expensive and fast
  - Dynamic RAM (DRAM): Used in main memory, cheaper and slower
- In DRAM, each bit is stored using one transistor (Read/Write control) and one capacitor (Charge electron to store 1, and discharge to store 0)
  - High density and low cost
  - Need lot of recharging because capacitor slowly leak electrons.
  - When recharging, read/write need to wait. When reading, the electron will be discharged, therefore require another recharge after read $\rightarrow$ slow
- SRAM: Each cell of data is constructed using transistors only
  - No need to recharge, always available $\rightarrow$ Static
  - But more expensive and less density

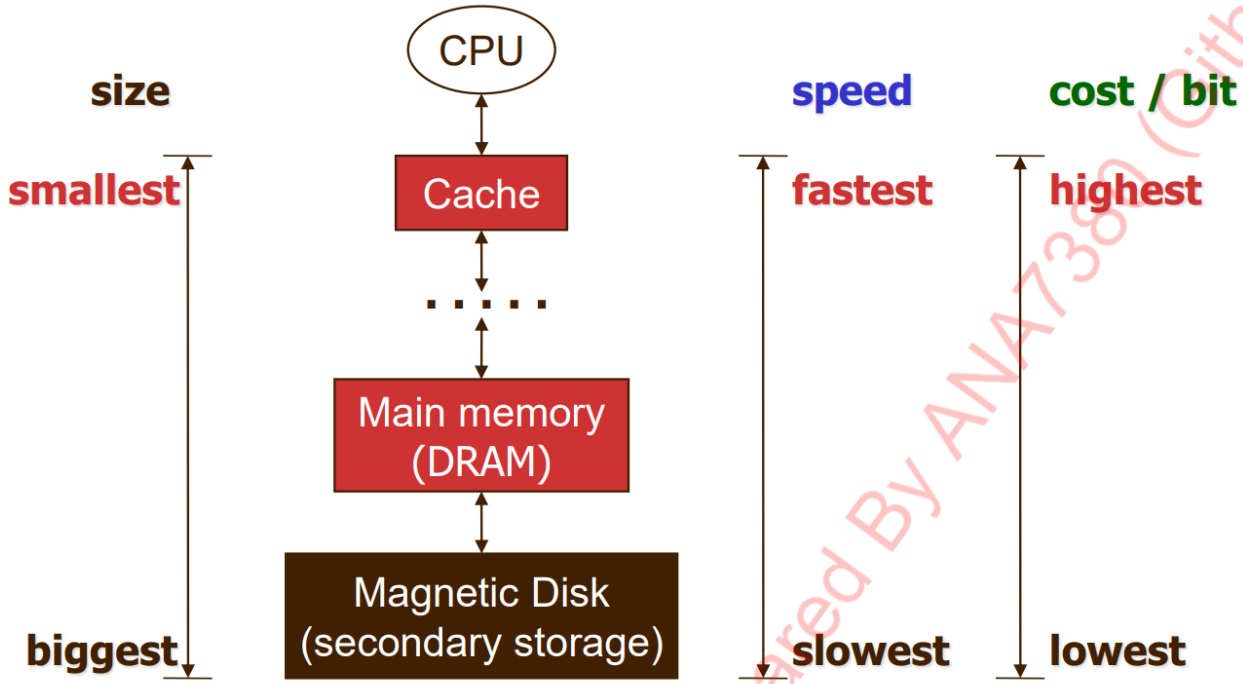
- Hard Disk Drive (HDD): Mechanical, rotating magnetic storage




  - Each Sector can hold 4096 Bytes of data
  - To read data from HDD:
    1. Wait until previous access complete
    2. Move the arm to the correct track (change radius)
    3. Wait until the sector rotates to under the arm
    4. Data transfer
    5. Controller overhead
- Processor-Memory Bottleneck:
  - **CPU performance is faster than DRAM** overtime and the gap is increasing.
    - Make waiting time long
- Principle of Locality:
  - Programs access a small proportion of their address space at any time.
  - Temporal locality: Items accessed recently are likely to be accessed again soon. (e.g. `i` used in for loop)
  - Spatial locality: Items near those accessed recently are likely to be accessed soon (e.g. array)

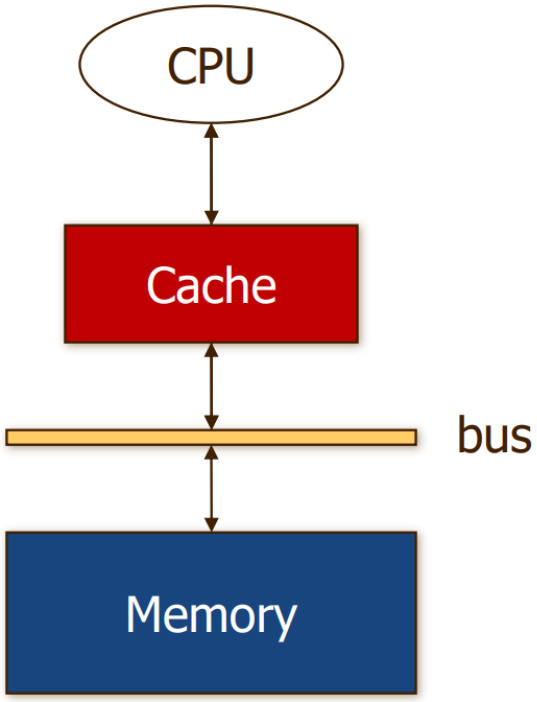

  - The dots along x axis over a specific memory address indicate temporal locality, and the neighbors are also accessed can be proved by the vertical lines.

- With temporal locality, the first iteration will be loaded from main memory(DRAM), since then it will be loaded to cache (SRAM) and have huge improvement.
- With spatial locality, when reading an array, it will bring the whole array from main memory(DRAM) to the cache (SRAM)




- Memory Hierarchy Operation:
  - Store everything on hard disk
  - When the program start to execute:
    - Load all instructions and data from disk to main memory (DRAM)
    - Copy instructions and data (Also nearby data) from DRAM to cache (SRAM)
    - If the required data is not in the cache, load it from DRAM then pass it to CPU.
    - Every instruction and data should be passed from SRAM to CPU, DRAM only passes to SRAM.
- Levels of Cache:
  - Level1 cache is closest to CPU (fastest but smallest)
  - If level1 cache does not contain the required data or instruction, it will check level2, 3 cache until it finds the data. Then the data is transferred up to level1 cache then deliver to the CPU.
  - L2 cache must be holding all the data that currently exist in L1 cache with some additional data. L3 cache must be holding all data in L1 AND L2 plus additional data.
    - DRAM contains all the data in L1, L2, L3 and contains more than that.
- Hit time: If the required data is in the cache, then hit time represents the time to pass from cache to CPU
- Miss penalty: If the required data is not in cache, then miss penalty represents the time to copy the data blocks from main memory to cache.

- Performance measures:
  - Hit Rate (ratio): probability of hit
  - Miss Rate: $1-$Hit Rate
  - Cache Hit time: Time to determine miss or hit + Time to access the cache
  - Cache Miss time: Time to bring a block of data from main memory to cache.




  - Bus: handles transfer of data between hardware
- Cache performance:
  - Average Memory access latency = hit time + miss rate * miss penalty
    - Hit time occurs in both hit and miss case (both need to bring data from cache to CPU at the end)

For two levels of caches, average memory latency
= $\text{hit time}_1 + \text{miss ratio}_1 * \text{miss penalty}_1$
= $\text{hit time}_1 + \text{miss ratio}_1 * (\text{hit time}_2 + \text{miss ratio}_2 * \text{miss penalty}_2)$
= $\text{hit time}_1 + \text{miss ratio}_1 * \text{hit time}_2 + \text{miss ratio}_1 * \text{miss ratio}_2 * \text{miss penalty}_2$

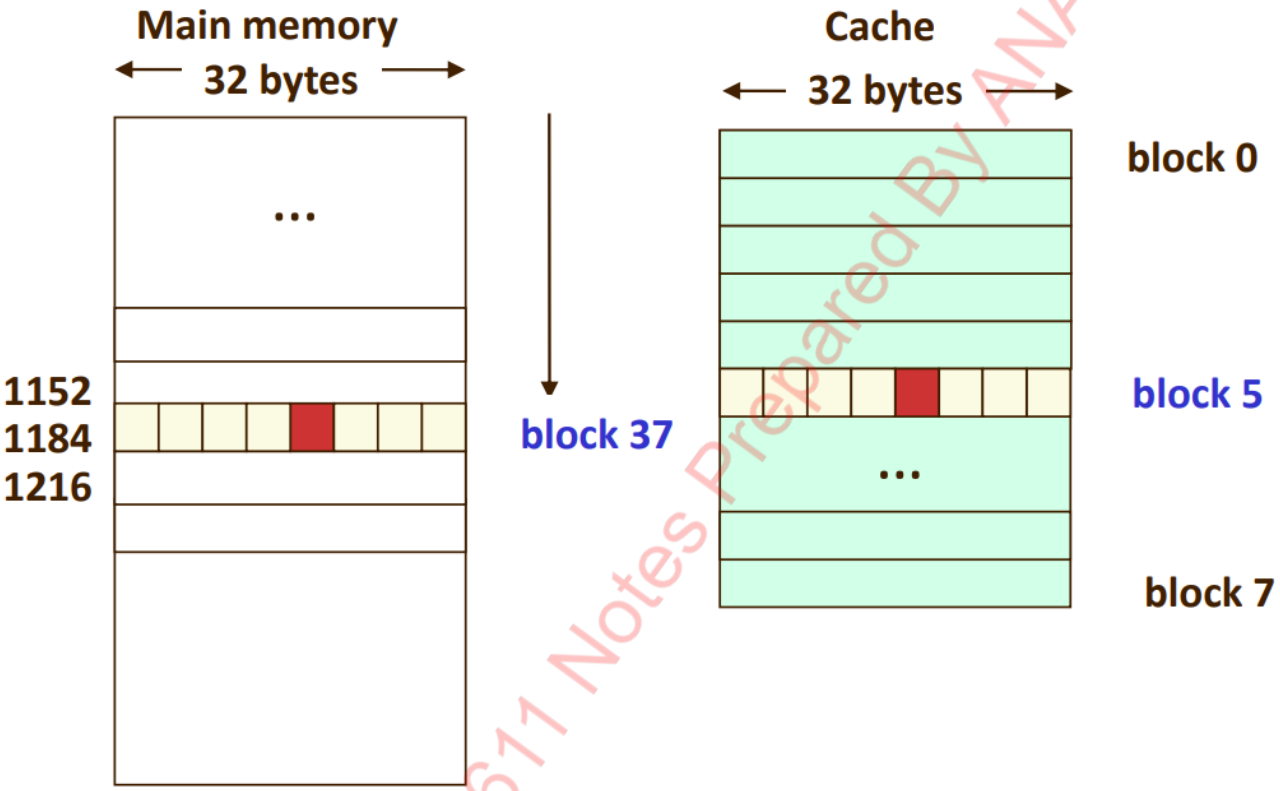
    - When the level1 cache does not contain the data, the level2 cache will need to look up for that data, it takes **hit time**$_2$ to bring the data from level2 cache to level1 cache.
    - This happens in both hit and miss case, if miss, it needs to look up in the main memory (DRAM) and then pass it back to level1 cache.
- Summary:

$$AMAL = t_1 + MR_1 * MP_1 = t_1 + MR_1 * (t_2 + MR_2 * MP_2)$$

# Cache Operations

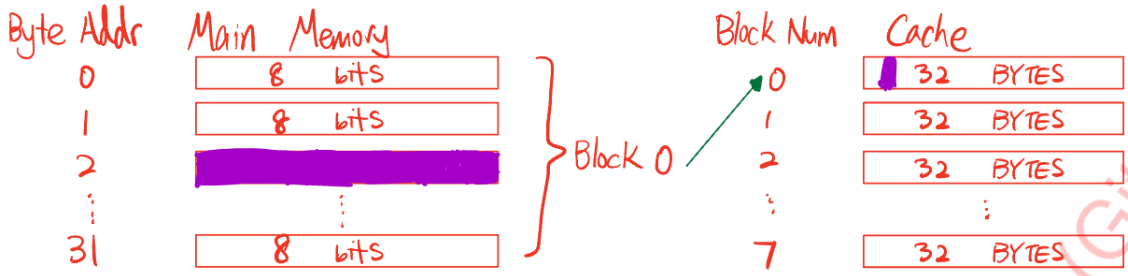
- Cache is an array of cache block. One cache block is 32 bytes or 64 bytes (spatial locality)
- The main memory is split into blocks of 32 bytes. Each block will have a block number, and the block will be assigned to a typical cache block.
- When transfer a block from main memory to cache, its location in cache is determined by memory address

## 1. Direct mapped




- One memory block assigned to one possible cache block
- Formula: Cache Block = (Block address) % (#Total Blocks in cache)
  - e.g. to pass block 65 to a cache with 8 blocks, 65 % 8 = 1, therefore it will be stored on cache block# 1
- Block address: Byte address divide by bytes in one block, then take lower bound.
- For a cache with $2^n$ blocks, each storing $2^m$ bytes:
  - Convert the byte address from decimal to binary
  - The last m bits indicate the offset into a specific cache block.
  - The next n bits indicate the cache block index.
- A valid bit is stored in front of the cache block to indicate if its valid or not (default 0)
- tag bits are stored in front of the cache block to indicate which particular block of main memory it come from.
- The last 5 bits of the bytes address indicate the offset of bytes into a specific block

- e.g. last 5 bits of byte addr = `11111` means the last 8 bits of that block



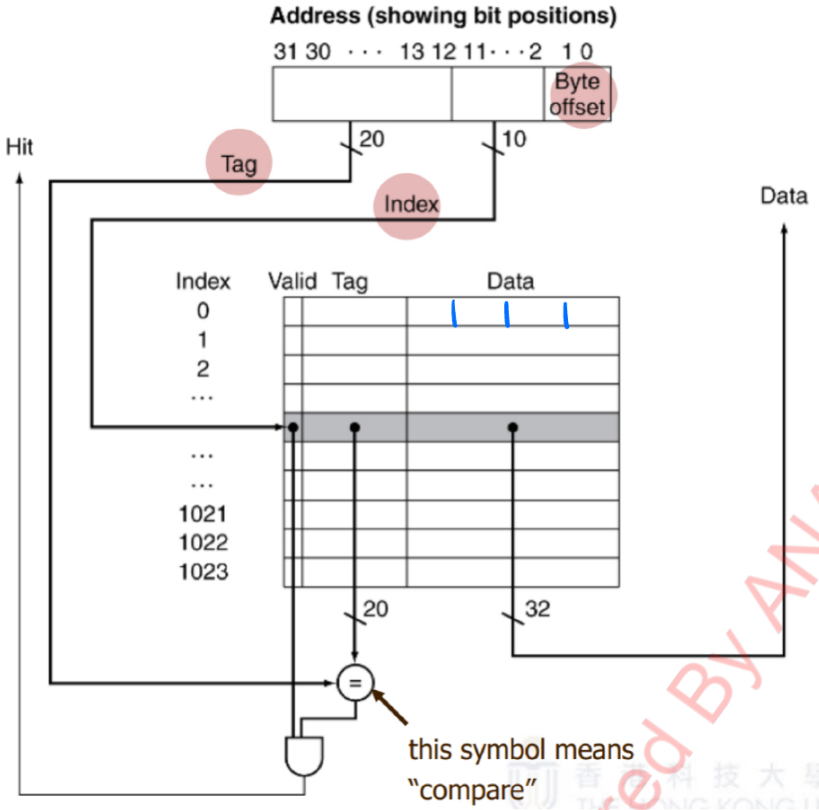

0 % 8 = 0 So stored in Cache Block #0

e.g. Byte address $(200)_{10}$ (10010|10000)

tag

Block 5

located at index 16 in the block




In this Cache, there are $2^{10}$ cache blocks

Each block stores 32 bits. (4 bytes)
↳ The last 2 bits of the byte address indicate byte offset in that cache block

↳ The next 10 bits indicate which block it belongs to.

↳ The remaining 20 bits is the tag, Combining with above gives the byte address.

- If tag matches, then its a hit.

- Disadvantage: Previous block in the cache will be overwritten

| Memory (byte) address generated by processor | Block address | Hit or miss in cache | Assigned cache block (where found or placed) |
|---|---|---|---|
| $0010\ 1100\ 0010_2$ | $0010\ 110_2$ | **Miss** | $(0010110_2 \bmod 8) = 110_2$ |
| $0011\ 0100\ 0000_2$ | $0011\ 010_2$ | **Miss** | $(0011010_2 \bmod 8) = 010_2$ |
| $0010\ 1100\ 0100_2$ | $0010\ 110_2$ | **Hit** | $(0010110_2 \bmod 8) = 110_2$ |
| $0010\ 1100\ 0010_2$ | $0010\ 110_2$ | **Hit** | $(0010110_2 \bmod 8) = 110_2$ |
| $0010\ 0000\ 1000_2$ | $0010\ 000_2$ | **Miss** | $(0010000_2 \bmod 8) = 000_2$ |
| $0000\ 0110\ 0000_2$ | $0000\ 011_2$ | **Miss** | $(0000011_2 \bmod 8) = 011_2$ |
| $0010\ 0001\ 0000_2$ | $0010\ 000_2$ | **Hit** | $(0010000_2 \bmod 8) = 000_2$ |
| $0010\ 0100\ 0001_2$ | $0010\ 010_2$ | **Miss** | $(0010010_2 \bmod 8) = 010_2$ |

  - First two trials are all miss because they are invalid, so the tag is updated after loading from main memory
  - The third trial tried to access block 110, which the tag matches, therefore hit
  - The last trial tried to access block 010, but the tag does not match, so new data copied to cache block 010 and tag overwritten
  - **miss rate = # miss / # total = 5/8**

**A direct-mapped cache with 16KB ($16 * 2^{10}$ Bytes = $2^{14}$ Bytes) of data and 8 word (32 Bytes = $2^5$ Bytes) blocks.**
**Assume 32 bit memory address from CPU**

Solution:
Amount of blocks = $2^{14} \div 2^5 = 2^9$ Blocks
So the last 5 bits of the byte address indicate byte offset
The next 9 bits indicate the block index
Remaining 18 bits are stored in the tag

Also a valid bit is added
Therefore one block contains: $1 + 18 + 32*8 = 275$ bits
For $2^9$ Block, total bits = $275 * 2^9 = 140800$ bits
Slightly more than $16\text{KB} = 2^{14} * 8 = 131072$ Bits

- Block Size and Miss Rate:
  - When the block size is small (many blocks), then there need to have frequent loading of data from main memory
  - When the block size is large (few blocks), then in case there is data competing for the same block requires more writing.
- If cache hit, the CPU performs normally
  - If miss, the CPU pipeline stalls
  - Then fetch the block data into cache
    - If it is instruction cache miss, then restart instruction fetching

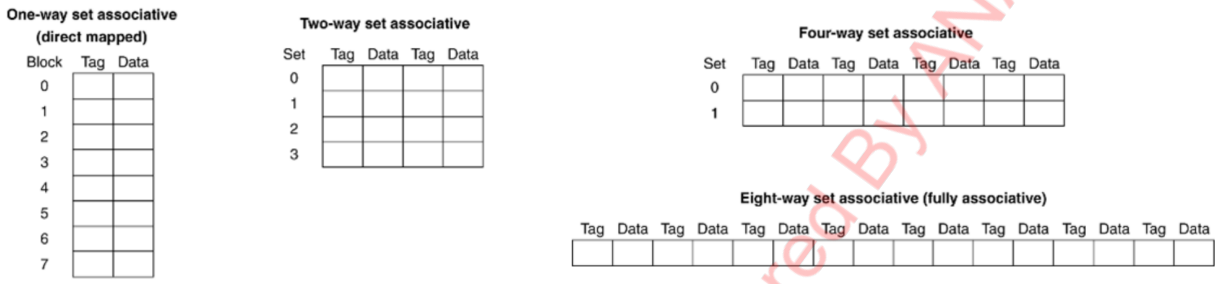
- If it is data cache miss, then restart data access

## 2. Fully Associative

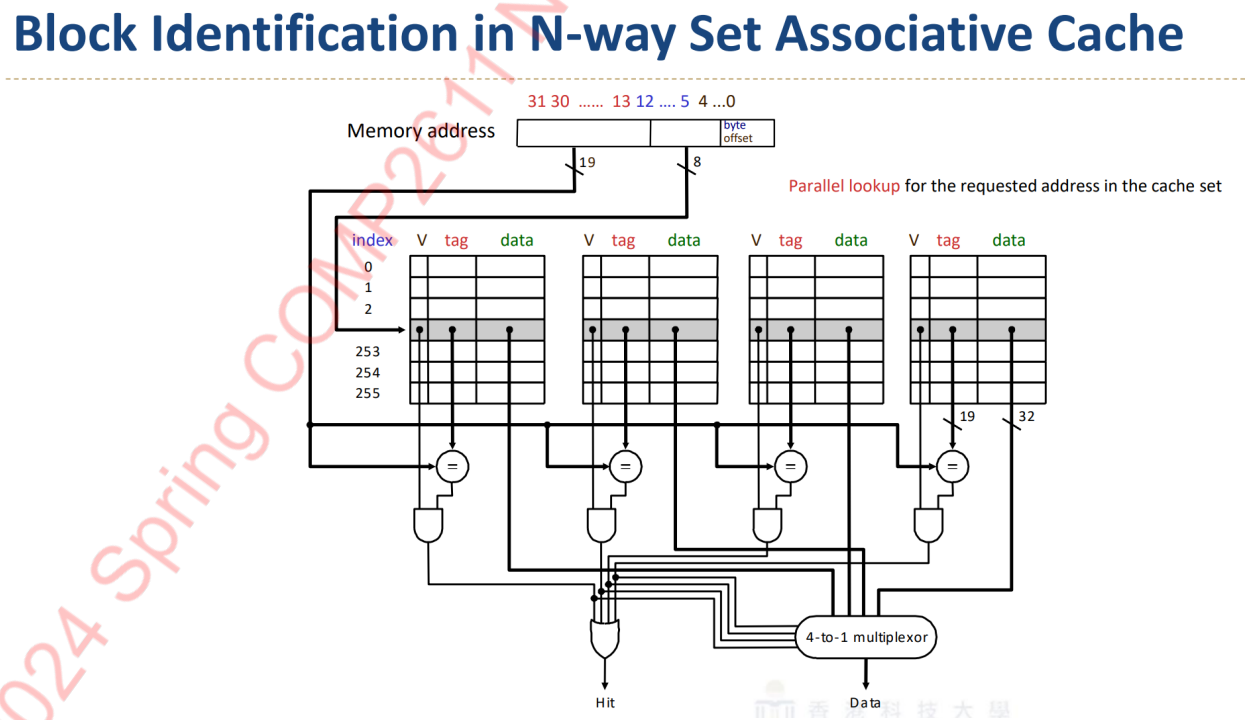
- Place the block from main memory to any block that is vacancy in the cache block
  - Advantage: No cache conflict, better hit rate
  - Disadvantage: When all cache fills, still misses. Also require more hardware and time to look up.

## 3. N-Way Set associative

- Reduce the number of cache sets by increasing the capacity one set can hold




- Example: CPU want to access block 0 and 8, in Direct mapped both will result in accessing block 0
  - In Eight Way set associative, both mod results in 0, but they were put one after another, will hit eventually.
- Advantage: Decrease in miss rate
- Disadvantage: Increase in hit time (due to longer search time)




- Given the set number for cache, loop through each of the blocks to check the tag.
- There is also instruction cache which stores the instruction for the program.

- Work identical to data cache, CPU send the PC to cache and cache check if it contain the instruction or not to decide whether need to contact the main memory.
- When instruction not found, CPU stalls and waits and re-fetch later

# Block Replacement

- For direct mapping, when the target block in cache is already occupied, the values in the cache will be written back into memory before writing the new block to cache.
- For Set associative, it first finds whether there is a block with valid bit 0, otherwise:
  1. Randomly select one block
  2. Find the least recently used block (temporal locality)
- Least Recently Used Block example: