CS 2115

Project (group - 4 person)

- report & presentation
- 5 bonus (up to 30%)

Lecture 01

---

1. History:

- Analytical Engine
  - Charles Babbage
  - Made by mechanical
  - Started in 1833; never finished
- Turing machine (1937) => basis of modern computing
  - Theoretically, as poweful as other computers
  - Conceptually, a finite set of states, a finite alphabet and a finite set of instructions
  - Physically, it has a head (read&write), and move along an infinitely long tape that is divided into cells storing a letter
- Stored-program concept (1944 by John von Neumann and Alan Turing)
  - Store program and data in memory
  - A computer can read them from memory
  - Program can be altered
- Structure of von Neumann machine
  - Main Memory $\Longleftrightarrow$ (CPU (CA and ALU)) $\Longleftrightarrow$ I/O
- Transistor(晶体管)
  - solid state device made from Silicon (sand)
  - Act as a variable value
- Integrated Circuit (1964)
  - Make computers smaller
  - Cost of a chip virtually unchanged with the growth in density
  - Components placed closer => faster to access
  - Reduction in power and cooling requirement
  - The interconnection is much more reliable than solder connections
- Moore's Law
  - The number of transistors that could be put on a single chip will be doubling every year (slows to 18 months in 70s(1970)) => speed of CPU also doubling
- The Gap between IC Capacity/ Design Productivity and Memory
  - Use **wide data buses** so we can retrieve more bits at the same time when we read or write to memory

    - Include a cache(between CPU and memory) or hierarchical buffering scheme to make memory chip work more efficiently
        - Faster but size is small and expensive
        - Reduce miss match (memory输出速度跟不上CPU需求)
    - Put cache into processors (increasingly, processor design dedicates over 50% transistors for cache)
    - Use high-speed buses to interconnect processor and memory

---

2. Turing Machine

- Hierarchy of Machines : Combinational Logic $\in$ Finite-state machine $\in$ Push down automation $\in$ Turing machine
- A Thinking Machine
    - basic ideas
        - Compute anything that a human can compute
        - Studied one of Hilbert's 10th mathematical conjecture on "Entscheidung problem"
        - Turing's machine is a thought experiment. Any algorithm can be carried
    - Method
        - Imagine a computer that writes everything down in a form that is completely specified using one symbol at a time
        - The computer follows a finite set of rules that are referred to every time after a symbol is written down
        - Rules are such that at any given time, only one rule is active so no ambiguity can arise
        - Each rule activates another rule depending on what letter/number is currently read.
- Rule of add 1 to a number (from lower bits)
    - If read 1, write 0, go right(the paper strip)
    - If read 0, write 1
    - If read blank, write 1
- Rule of substract 1 from a number
    - If read 1, write 0
    - If read 0, write 1, go right

Lecture 02

1.1 Organization And Architecture

- **Computer Architecture** : Refers to those attributes of a system visible to a programmer or, put another way, those attributes that have a direct impact on the logical execution of a program
- **Computer Organization** : Refers to the *operational units* and their *interconnections* that realize the architectural specifications (Same architecture may have different organizations)
- Organization attributes : Hardware details transparent to the programmer

    - Control signals
    - Interfaces between computers and peripherals
    - Memory technology used

1.2 Sturcture And Function

- Key of clearly describe millions of elementary electronic components : Recognize the hierarchical nature of most complex systems (Divide the system into different levels to implement)
- At each Level, the designer is concerned with structure and function
    - Structure : The way in which the components are interrelated
    - Function : The operation of each individual component as part of the structure
- Two choice to understand the Organization
    - Start at the bottom and building up a complete description
    - Beginning with a top view (Clearest and most effective)
- Function
    - Data processing
        - The data may take a wide variety of forms => range of processing requirements is broad
    - Data storage
        - Even if the computer is processing data on the fly (in and out immediately), the computer must temporarily store at least those pieces of data that are being worked on at any given moment => short-term data storage function (RAM)
        - Equally important, the computer performs a long-term data storage function (File of data are sotred on the computer for subsequent retrieval and update) => ROM
    - Data movement
        - **move data** between itself and outside world => I/O
        - Device to provide this function => **peripheral** (外设)
        - data moved over longer distances => *data communications*
    - Control
        - Within the computer, a control unit manages the computer's resources and orchestrates (编排) the performance of its functional parts in response to those instructions

- Structure :
    - Out : All the linkages to the external environment can be classified as peripheral devices or communication lines
    - Internal:
        - **Central processing unit (CPU)**: Controls the operation of the computer and performs its data processing functions; often simply referred to as **processor** (1 or more)
            - **Control unit**: Controls the operation of the CPU and hence the computer
                - Parts
                    - Sequecing logic
                    - Control unit registers and decoders

        - Control memory
    - Different implementations of Control unit
        - *microprogrammed* implementation : Operate by executing microinstructions that define the functionality of the control unit
- **Arithmetic and logic unit (ALU)** : Performs the computer's data processing functions
- **Registers** : Provides storage internal to the CPU
- **CPU interconnection** : Some mechanism that provides for communication among the control unit, ALU and Registers

- **Memory unit** : Stores data
    - **Primary memory** (Main memory): A fast memory that operates at electronic speeds, programs must be stored in this memory while they are being executed
        - Consist of a large number of semiconductor storage cells, each capable of storing one bit of information. These cells are handled in group of fixed size called *words* instead of read or written individually
        - word length : number of bits in each word (16,32,64 bits)
        - *Address* : To provide easy access to any word in the memory, address is related with each word location
        - A memory in which any location can be accessed in a short and fixed amount of time after specifying its address is called *random-access-memory*(RAM), the time required to access one word is called the *memory access time*
    - **Cache Memory** : As an adjunct(附件) to the main memory, a smaller, faster RAM unit, called a *cache* => hold section of program that currently being executed, along with any associated data
    - **Secondary Storage** : When large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently. (magnetic disks, optical disks(CD,DVD), flash memory devices)
- **I/O**: Moves data between the computer and its external environment
- **System interconnection**: Some mechanism that provides for communication among CPU, main memory, and **I/O**. A common example : **System bus**, consisting of a number of conducting wires to which all the other components attach.

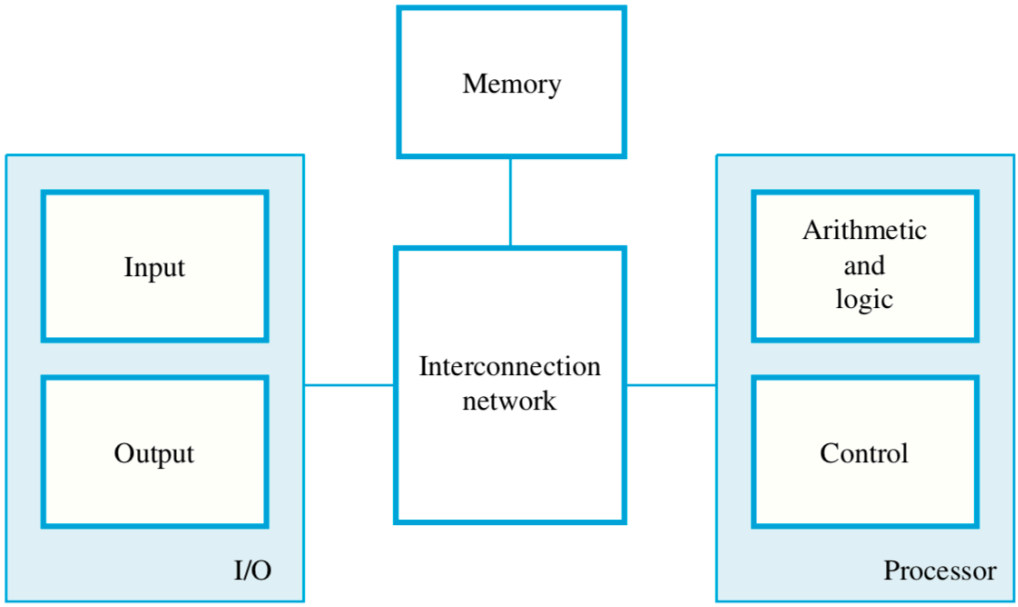


Figure 1.1 Basic functional units of a computer.

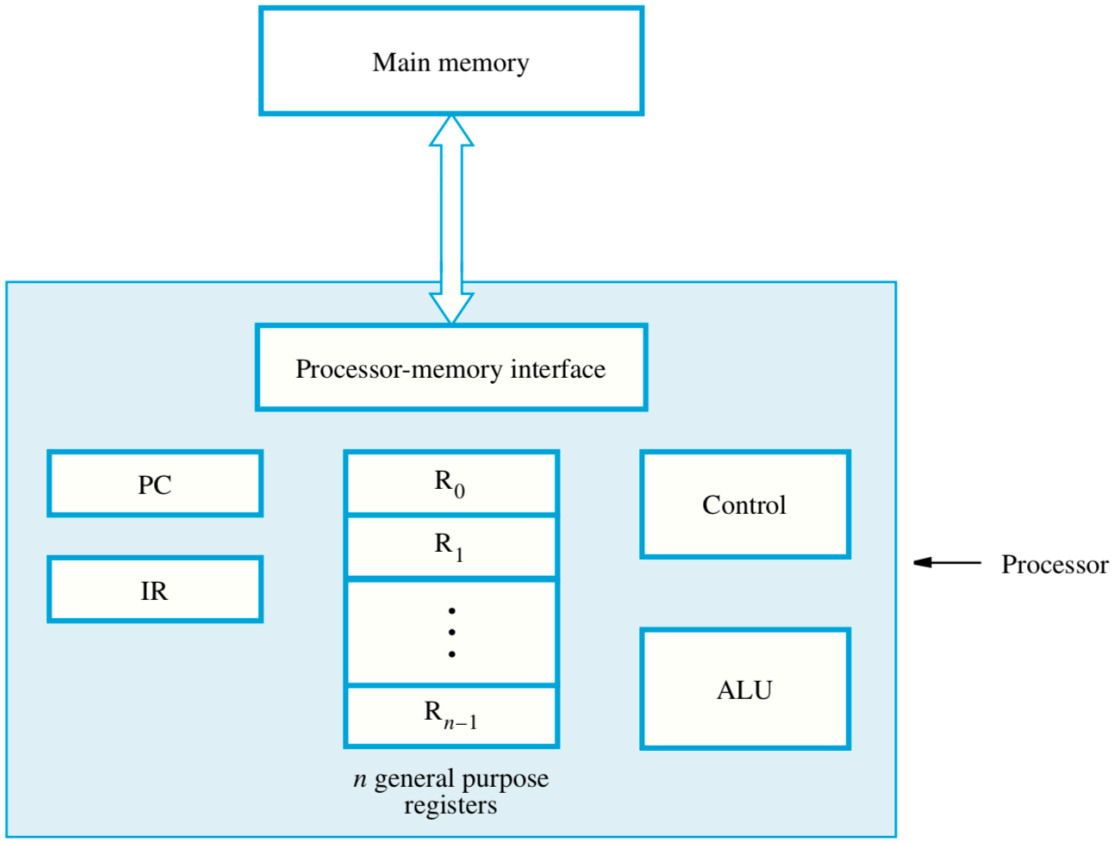


Figure 1.2 Connection between the processor and the main memory.

1.4 Number representation and arithmetic operations

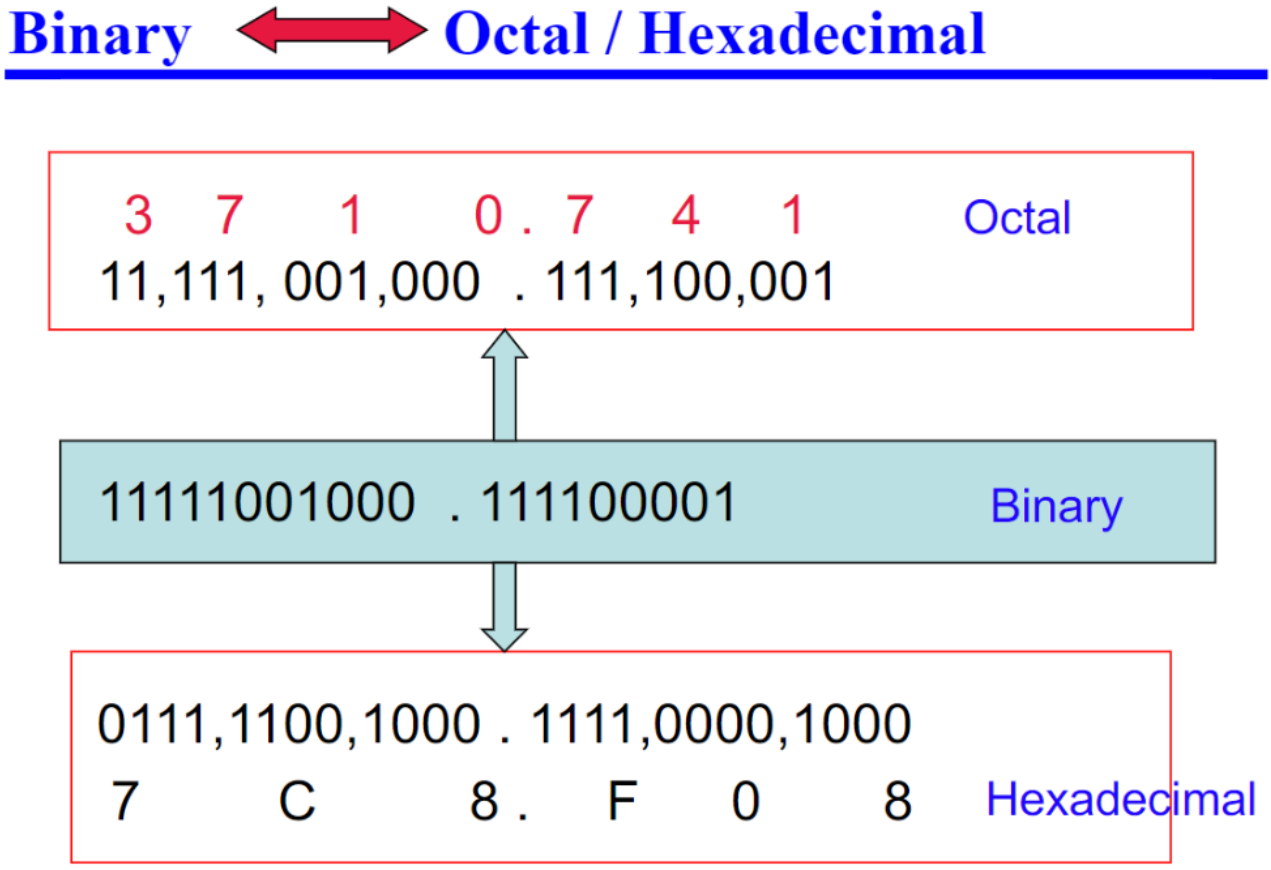
LEC

- Decimal, Binary, Octal , Hexadecimal systems
  - Radix-R system

$$N = (a_{n-1}a_{n-2}\dots a_1a_0.\, a_{-1}a_{-2}\dots a_{-m})_R \tag{1}$$

$$N = \sum_{i=-m}^{n-1} a_i * R^i (a_i = 0, \dots, R-1) \tag{2}$$

- R = 2 binary, R = 8 Octal, R = 16 Hexadecimal



- 10 to R (mod by R,divide by R each time)
- for fractional number (multiple by 2 and record the integer part)
- Signed Number Representation
  - Sign Magnitude Number (0 for positive, 1 for negative) but:
    - Addition and subtraction need to consider sign and magnitude.

        - Two representation of 0
        - Complicate circuit and more computation time
    - Two's Complement
    - Basic Rules:
        1. Positive numbers are represented in the same fashion as in sign magnitude numbers.
        2. Negative numbers are represented as the complement of the corresponding positive numbers.
    - Direct Method

$$[N]_2(two's\ complement) = 2^n - (N)_2 \tag{3}$$

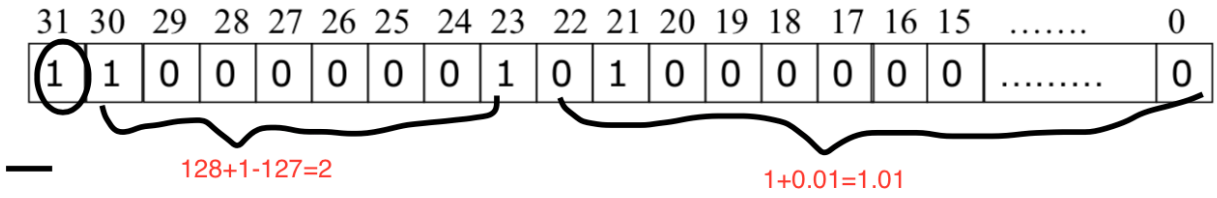
        - delete the first bits of 2^n it is 0, so we can maintain addition of (N) and (-N) is 0
    - Fast Method
        - Flit the bits and add 1
    - Conversion Between Lengths
        - Positive number pack with leading zeros
        - Negative numbers pack with leading ones
        - Reason : the sum is 0
    - Addition and Subtraction
        - Operation A-B by $(A)_2 + [B]_2$
        - We only need a binary adder and do all the things
    - Range of Two's complement system : $-2^{n-1} \leq N \leq 2^{n-1} - 1$
        - If after addition/ substraction, the result is out of the range, it is called **overflow**
        - Negative : 2^{n-1} (start with 1), positive and zero: 2^{n-1} (start with 0)
    - Overflow Rule
        - All positive number with first bit 0, negative with first bit 1
        - If add 2 negative get a positive or 2 positive get 1 negative, overflow occurs
- Codebook Representation
    - Binary Floating-point
        - Method to record floating-point number => use a number multiply with 2^n
        - first bits : sign 0(+) 1(-)
        - k bits bias exponent : record $(指数)_2 + bias$ where bias is $(2^{k-1} - 1)_2$ => to maintain k digits
        - last digits to record the significant digit (小数点后) => There is only one digit before the dot (scientific)
    - Binary Floating-point => decimal
        - when calculating convert the exponent to decimal first

$$(N)_{10} = (-1)^s(1 + Significand)2^{(E)_2 - 127} \tag{4}$$

        - S : Sign Bit (0 + 1 -)

$$Significand = s_1 * 2^{-1} + s_2 * 2^{-2} + \ldots + s_n * 2^{-n} \tag{5}$$

What decimal number is represented by this binary single floating point representation?



-(1.01)*2^2

- The Range of 32-bit Binary Floating Point number (8 exponent bits)
  - Biggest positive number : $(1 + s_1 * 2^{-1} + s_2 * 2^{-2} + \ldots + s_n * 2^{-23}) * 2^{128}$
  - Smallest positive number : $(1) * 2^{-127}$
  - Smallest negative number : $-(1 + s_1 * 2^{-1} + s_2 * 2^{-2} + \ldots + s_n * 2^{-23}) * 2^{128}$
  - Biggest negative number : $-(1) * 2^{-127}$
- IEEE Standard for Binary Floating Point Numbers

单精度格式位模式	值
$0 < e < 255$	$(-1)^s \times 1.f \times 2^{e-127}$ (二进制正规数)
$e = 0, f \neq 0$	$(-1)^s \times 0.f \times 2^{-126}$ (二进制次正规数)
$e = 0, f = 0$	$(-1)^s \times 0.0$ (有符号的零)
$e = 255, f = 0, s = 0$	+inf (正无穷大)
$e = 255, f = 0, s = 1$	-inf (负无穷大)
$e = 255, f \neq 0$	NaN (非数、非确定值)

  - Single format (1+8+23=32bits)
  - Double format (1+11+52=64bits)
- Encode
  - BCD: each digits have it binary form
  - ASCII : use two one digit hexdecimal number
    - 0-9 $(30 - 39)_{16}$
  - Gray code
    - each two neighbor number different(hamming distance) is 1
    - Distance between two binary codewords is equal to the number of bits that these two codewords are different.

$$N_{Gray} = (N)_2 \; XOR \; (N >> 1)_2 \tag{6}$$

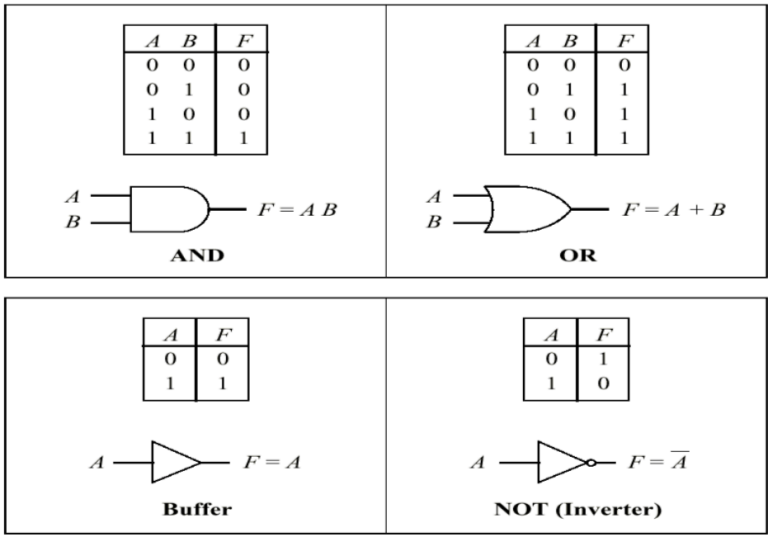
Lec 3 Fundamentals of Combinational Logic Circuits

- Basics of Combinational Circuits
  - Boolean Algebra
  - Truth Table(Blueprint or your target) & Logic Gates
    - And : 串联开关, in math multiplication (one zero cause all zero)
    - or： 并联开关, in math add (if one none zero, add them get non zero)

Logic Gates and Their Symbols

Logical symbols for AND, OR, and NOT Boolean functions.

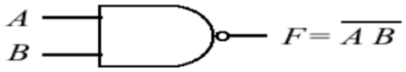
Buffer simply means equivalence mathematically (but interpret as “holding area” in circuits)



- Note the use of the “inversion bubble.”
- Be careful about the “nose” of the gate when drawing AND vs. OR.

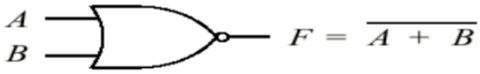
Logic symbols for NAND, NOR, XOR, and XNOR Boolean functions

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0



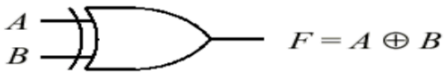
NAND

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0



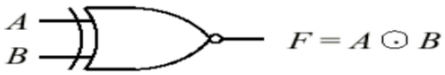
NOR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0



Exclusive-OR (XOR)

A	B	F
0	0	1
0	1	0
1	0	0
1	1	1



Exclusive-NOR (XNOR)

- Given n boolean inputs, there are 2^{2^n} functions :
  - 2^n combinations => define how many rows
  - 2 represents the output may has 2 values (True or False) => consider each row => 2^{2^n}

Inputs		Outputs							
A	B	*False*	*AND*	$A\overline{B}$	A	$\overline{A}B$	B	*XOR*	*OR*
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Inputs		Outputs							
A	B	*NOR*	*XNOR*	$\overline{B}$	$A+\overline{B}$	$\overline{A}$	$\overline{A}+B$	*NAND*	*True*
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Basics of Boolean Algebra

Relationship	Dual	Property
$A\,B = B\,A$	$A+B = B+A$	Commutative
$A\,(B+C) = A\,B+A\,C$	$A+B\,C = (A+B)\,(A+C)$	Distributive
$1\,A = A$	$0+A = A$	Identity
$A\,\overline{A} = 0$	$A+\overline{A} = 1$	Complement
$0\,A = 0$	$1+A = 1$	Zero and one theorems
$A\,A = A$	$A+A = A$	Idempotence
$A\,(B\,C) = (A\,B)\,C$	$A+(B+C) = (A+B)+C$	Associative
$\overline{\overline{A}} = A$		Involution
$\overline{A\,B} = \overline{A}+\overline{B}$	$\overline{A+B} = \overline{A}\,\overline{B}$	DeMorgan's Theorem
$AB+\overline{A}C+BC = AB+\overline{A}C$	$(A+B)(\overline{A}+C)(B+C) = (A+B)(\overline{A}+C)$	Consensus Theorem
$A\,(A+B) = A$	$A+A\,B = A$	Absorption Theorem

A, B, etc. are called Boolean variables or Literals.
The values 0 and 1 are called constants.

Principle of Duality:
The **dual** of a Boolean function statement is obtained by:
- replace AND with OR;
- replace OR with AND,
- replace 1's by 0's,
- replace 0's by 1's

- Canonical Implementation (guess the function by in/output)
  - SOP(sum-of-products) （这样做才对）
    - minterm : to make it true

- Find all true value(output)
- find the minterm function to satisefy it
- add them together (use or for each minterm)

The Sum-of-Products (SOP) Form

Minterm Index	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

What can this logic function be used for?

- Minterm is defined as the Boolean expression with minimum terms satisfying the output.
- F is true when:

  A=0 B=1 C=1 Minterm 3

  A=1 B=0 C=1 Minterm 5

  A=1 B=1 C=0 Minterm 6

  A=1 B=1 C=1 Minterm 7

  $F = \Sigma\ (3, 5, 6, 7)$

- $F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

The SOP (Sum-of-Products) form is a two-level AND-OR equation.

- POS (product-of-sum) (不这样做就对了)
  - Find all false vale (input)
  - find the minterm function to satisfy it (make it true)
  - get not of minterms
  - multiple them together
  - maxterm = NOT(minterm)， to make it wrong

The Product-of-Sum (POS) Form

Minterm Index	A	B	C	F
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

- Minterms: $F = \Sigma\,(3, 5, 6, 7)$
- SOP:

$$F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

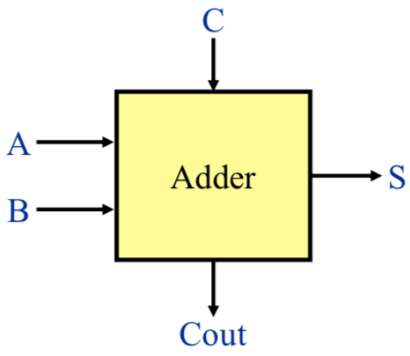
- POS: Product-of-Sum

$$F = (\overline{\bar{A}\bar{B}\bar{C}})\,(\overline{\bar{A}\bar{B}C})\,(\overline{\bar{A}B\bar{C}})(\overline{A\bar{B}\bar{C}})$$
$$= (A+B+C)(A+B+\bar{C})(A+\bar{B}+C)(\bar{A}+B+C)$$

The Product-of-Sum form (POS) is a two-level OR-AND equation

- Circuit Simplification
  - One-bit Adder : need one bit (进位) , thus three value in total
    - A : input digit 1
    - B : input digit 2
    - C : 进位 from last digit
    - S : answer
    - Cout : this digit 进位
    - 逐位计算， 每一位运行一次下图

One-bit Adder



Truth Table

A	B	C	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

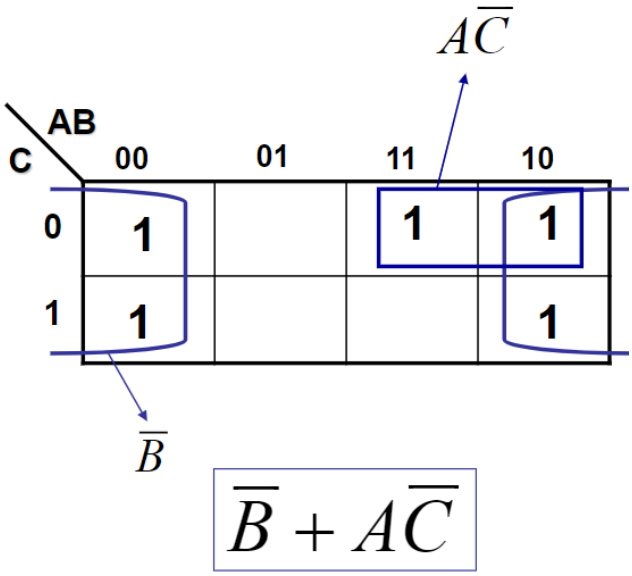
SOP Forms:

$$S = \overline{A}\,\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\,\overline{C} + ABC$$

$$C_{out} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

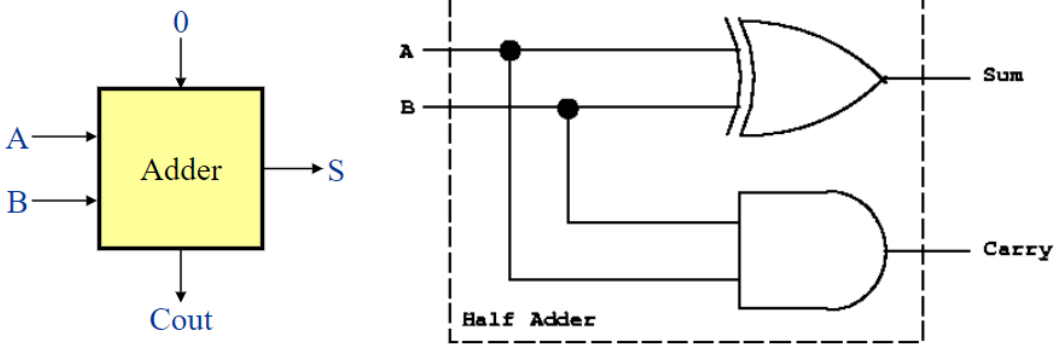
- Karnaugh Map (K-map)
  - Often used to simplify logic problems with 2, 3 or 4 variables
  - # of cell : 2^n
  - use gray code to number the entries => to make it easy to absorb i.e. use (A+not A=1)
  - Now calculation of SOP is simplified
    - fill the answer of truth table to the K-map
    - Use absorb

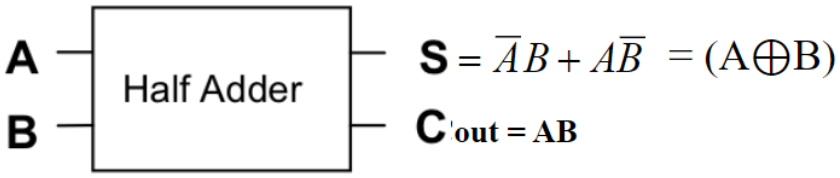
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



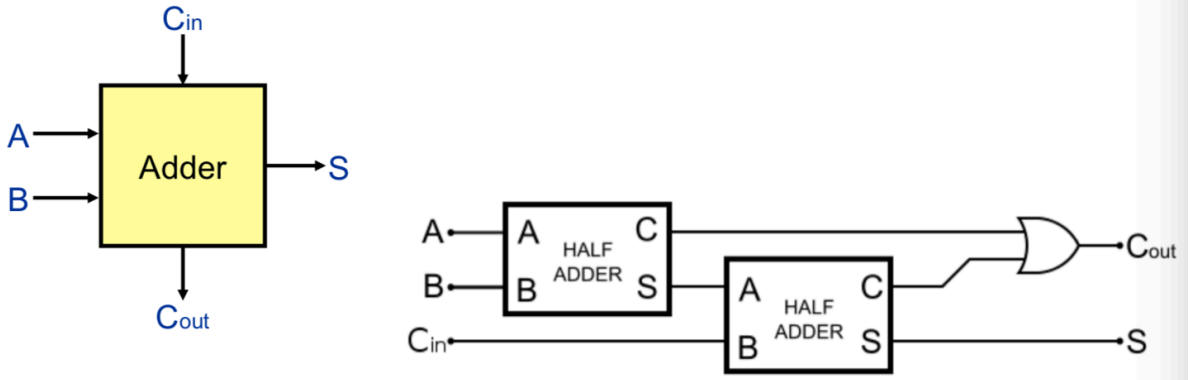
- Half adder (C=0)
  - remove C

- $S = A\ XOR\ B; C_{out} = A\ AND\ B$



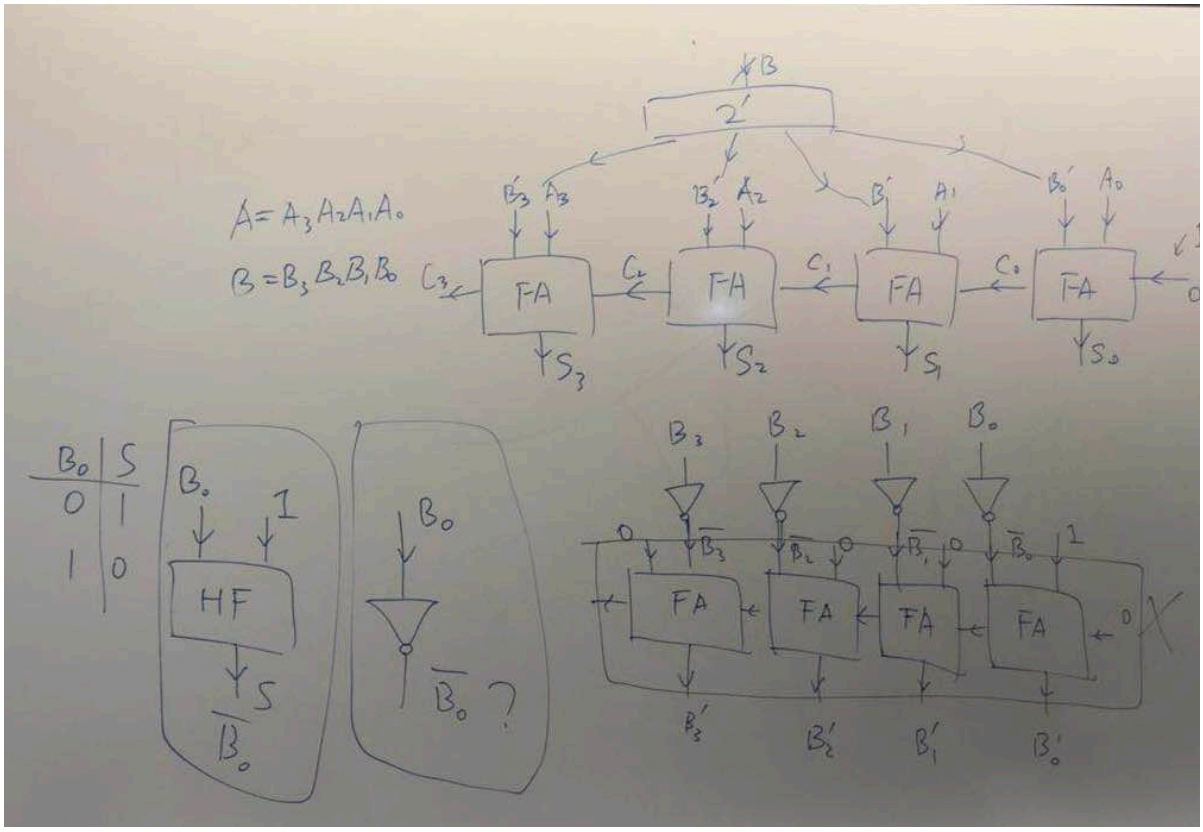


- Final Cout = Cout1 OR Cout2 OR ...
  - what if 2 Cout equal to 1 => impossible because when Cout1 is 1, A HF B = 0, Cout2 is impossible to be 1
- use 2 half adder to simplify the circuit(reuse a XOR gate)



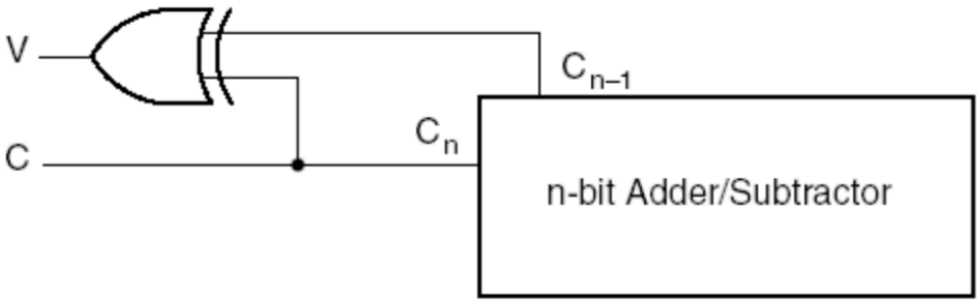
$$S = \overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C}_{in} + A\overline{B}\,\overline{C}_{in} + ABC_{in}$$
$$= \mathrm{A} \oplus \mathrm{B} \oplus \mathrm{C_{in}}$$

- Binary Ripple Carry Adder
  - Use half adder to implement addition or subtraction(2's complement)



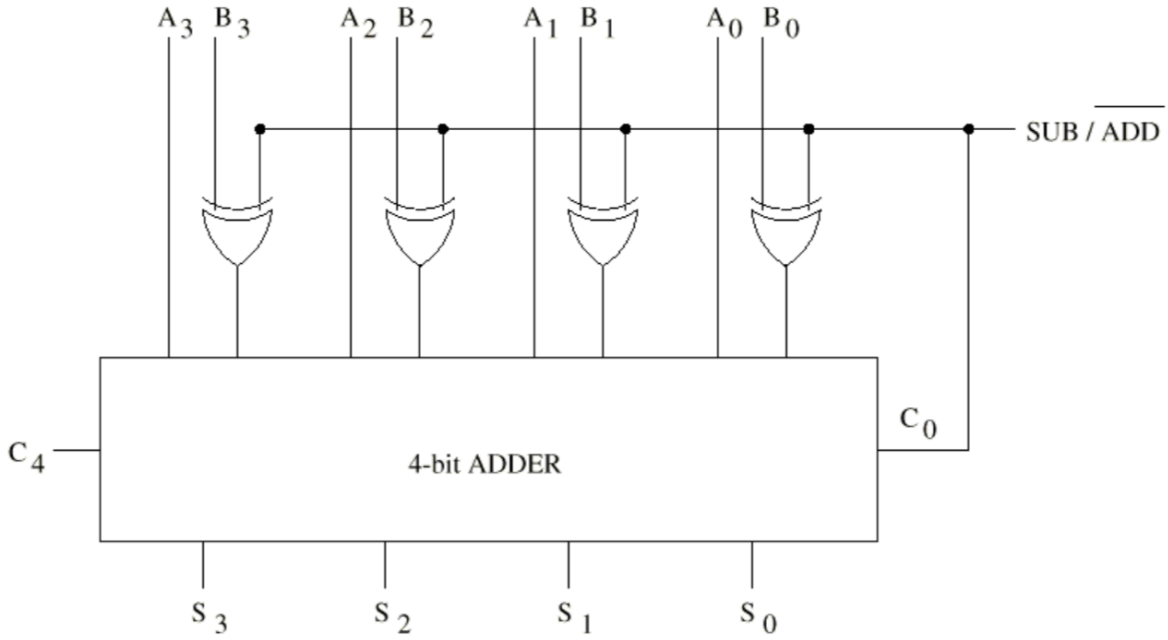
- avoid overflow (judge)
  - Unsigned : check whether Cout is 1
  - Signed : check whether the value of last two digits equal to the output value
    - [A(HA)S](FA)[B(HA)S] output the carry bits (carry bits is the answer whether it is overflow)
  - Also can be detected by checking whether the carry bits of last bit and second last bit are same (if do not same, the overflow occurs)
    - 1 1 0 => 2 negative add to 0
    - 0 0 1 => 2 positive add to 1

$$V = C_{n-1} \oplus C_n$$



- Combine addition add substraction

- when substraction : input 1 => 2's complement
- when addition : input 0



- Carry-Lookahead Adder
  - The delay in developing S_0 through S_{n-1} and C_n
  - Every digit have to wait for the carry bit of last digit => have to calculate one by one
    - calculate C : 2 gate delay
  - speed up => generation of the carry signal => can calculate at the same time
  - method

$$S_i = A_i \oplus B_i \oplus C_i$$

$$\begin{aligned} C_{i+1} &= A_iB_i + A_iC_i + B_iC_i \\ &= A_iB_i + (A_i + B_i)C_i \\ &= G_i + P_iC_i \end{aligned}$$

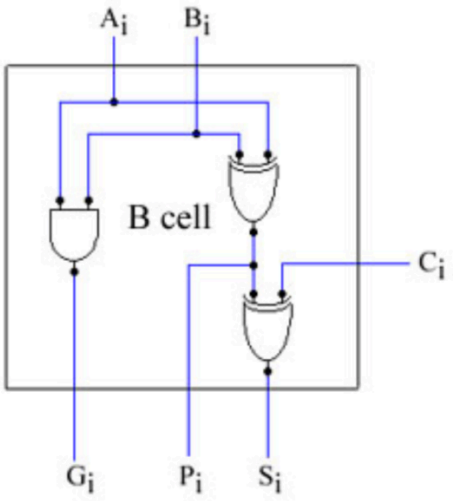
where $G_i = A_iB_i$ and $P_i = A_i + B_i$

A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

  - G is *generation* function(only consider the situation of current digit) and P called *propagate* function (have to consider the carry bits of last digit)
  - If A_i and B_i equal to 1 , then G is 1 and the result is alread gotten => don't need to consider this situation in the next product
  - Thus we can rewrite the function

$$C_{i+1} = A_iB_i + (A_i \oplus B_i)C_i \tag{7}$$

A bit-stage cell



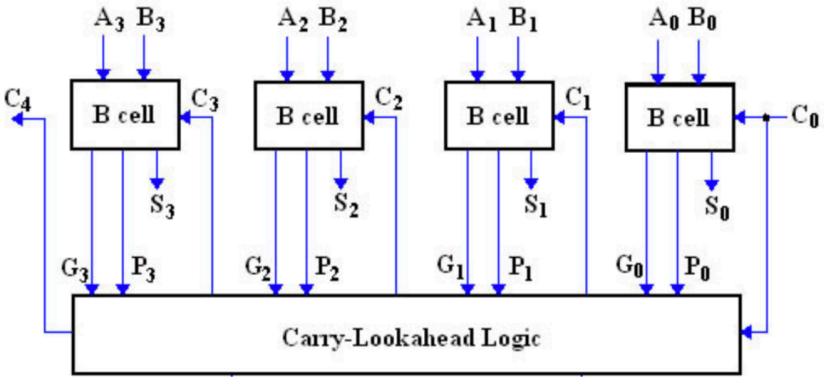
$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = G_i + P_i C_i$$

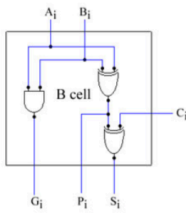
$$G_i = A_i B_i$$
$$P_i = A_i \oplus B_i$$

- Expand the equation

$$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 C_0$$

  - **all carries can be obtained three gate delays** after applying input A,B and C_0 (because only C_0, G, P are used), 4 gates delay for S

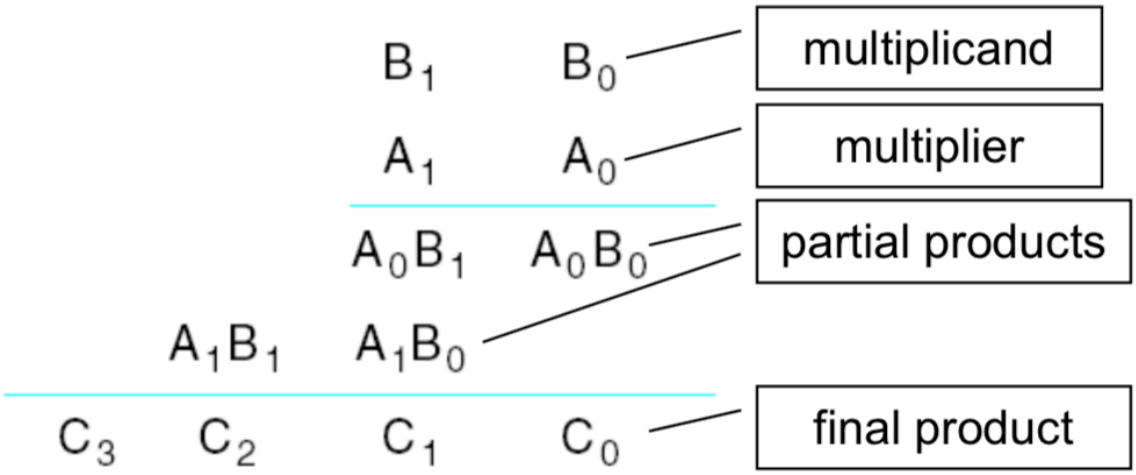




  - All carries are calculated indepently
  - basic idea : calculation of G and P is faster than C

- Binary Multiplier
  - For unsigned numbers

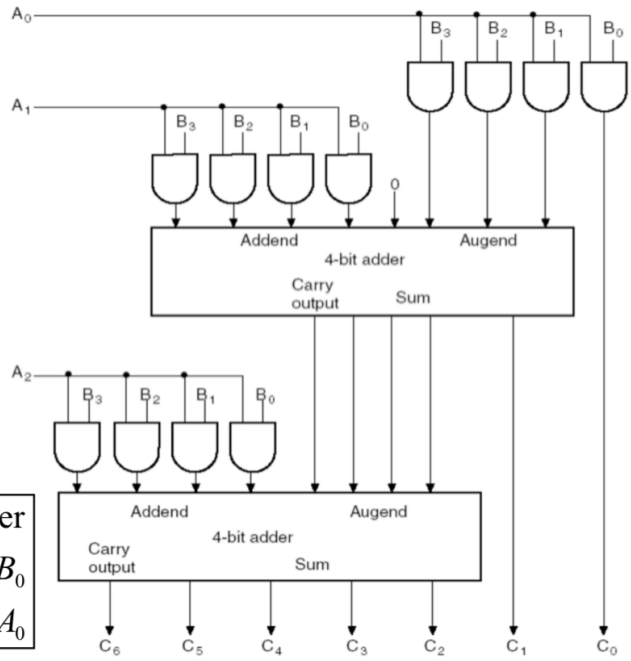
- Iteratively multiply the multiplicand by each bits of the multiplier, and properly add the answer



  - Partial product means just AND gate, it works for binary bits
- The logical circuit

For a binary multiplier with more bits, a multi-level circuit can be built.

In each level, a bit of the multiplier is ANDed with each bit of the multiplicand.



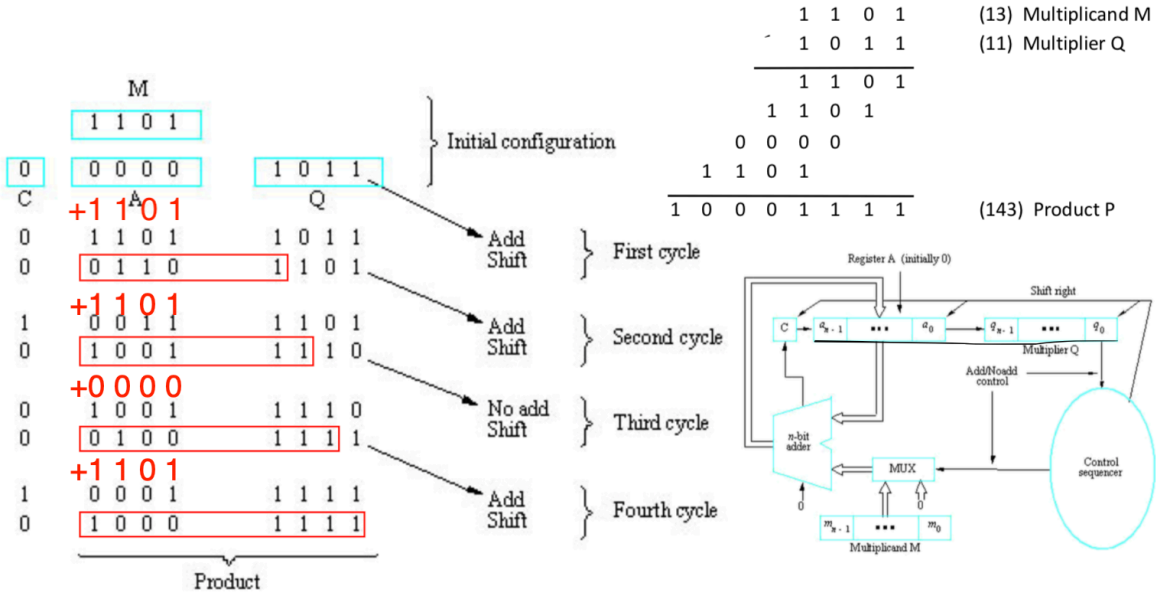
A 4 - Bit by 3 - Bit Binary Multiplier

$$B_3B_2B_1B_0$$
$$\times \quad A_2A_1A_0$$

  - AND gate and fill empty bits by zero
  - Left is Addend left is Augend
- Can be performed more easily by using adder circuitry in the ALU for a number of sequential steps
  - Q : multiplier
  - M : multiplicand
  - C : carry
  - A and Q : Used to hold the parital product
- Procedure
  1. q_i represents whether to add multiplier in this digit
     - q_i = 0 => add 0
     - q_i = 1 => add M

2. Shift right C, A and Q one bit

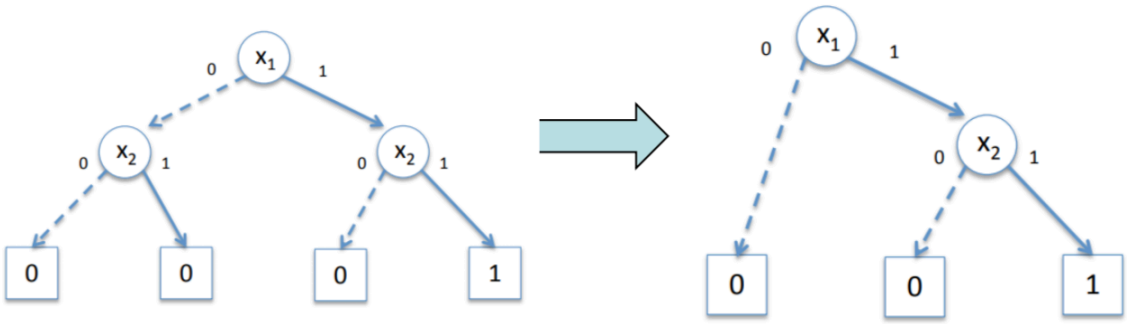
- ***Example*: 1101 × 1011**



- Shannon's Expansion
  - keep going => there will be 2^n min-term

- **Shannon's expansion theorem**
- $F(X_1, X_2, \ldots, X_n) = X_1 F(1, X_2, \ldots, X_n) + \overline{X_1} F(0, X_2, \ldots, X_n)$
- **Proof is simple**
  - $X_1 = 0,\ F(X_1, X_2, \ldots, X_n) = 0 + F(0, X_2, \ldots, X_n)$
  - $X_1 = 1,\ F(X_1, X_2, \ldots, X_n) = F(1, X_2, \ldots, X_n) + 0$

- Binary Decision Diagram
  - Root is the variable => each root has 2 child represent root=0 and root = 1 respectively
  - child can be merged into parent if their values are same

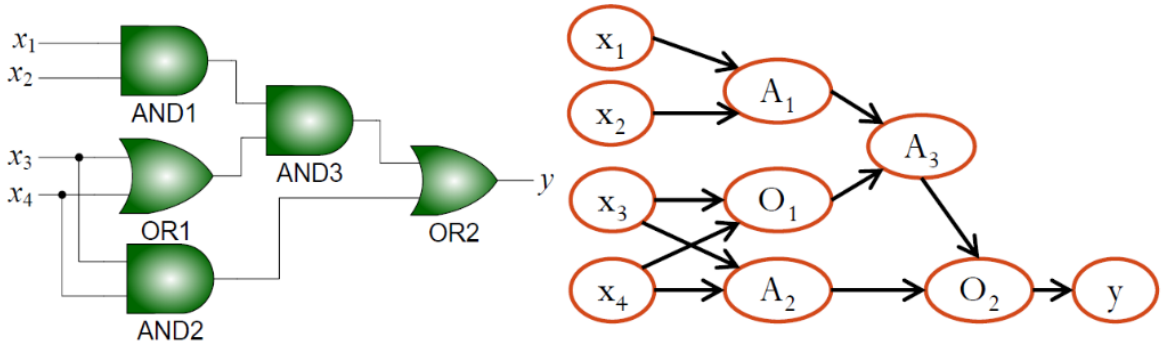


  - Combinational Logic Flow Diagram
    - Has no loop : The graph is a $directed\ acyclic\ graph(DAG)$

- Given an input pattern, how to find the output ? => Use Topological sorting to find the correct path to go through from the input to the output (BFS)

Combinational Logic Flow Diagram

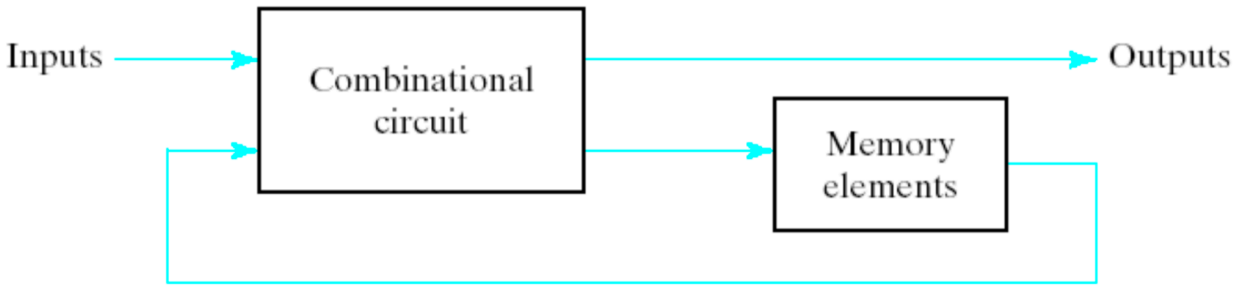
- Represented as a directed graph.
  - Inputs, outputs, and gates ➔ nodes
  - Wires ➔ directed edges.
    - Why directed edges? Signal flow has direction.



Lec 04 Sequential Logic Circuits

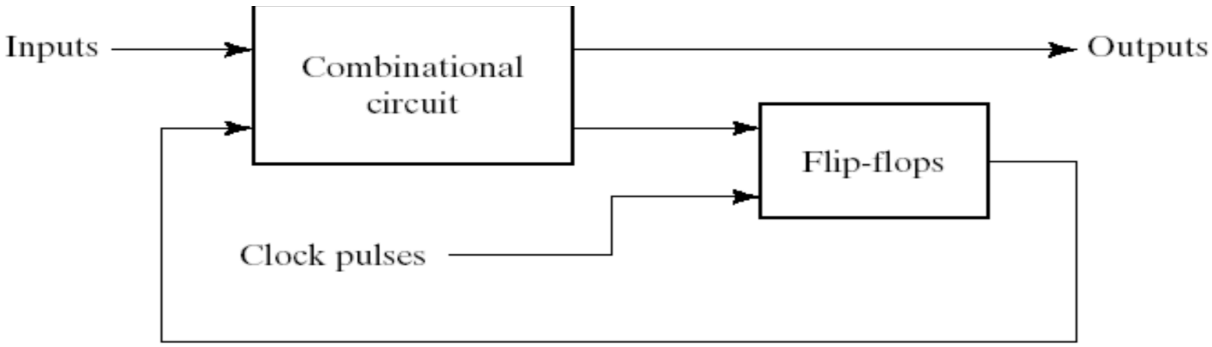
Concept of sequential circuits

- Sequential Circuits : Every digital system is likely to have combinational circuits, most systems encountered in practice also include *storage elements*, which require that the system be described in term of *sequential logic* (The output of last operation will be the next input of the next operation).

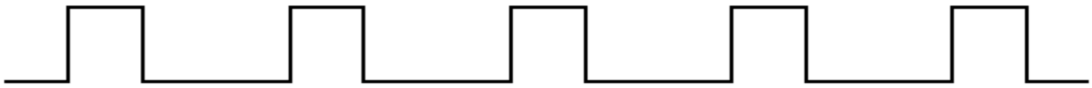


Block Diagram of Sequential Circuit

- Asynchronous vs. Synchronous(同步) Sequential Circuits
  - asynchronous : change its outputs and internal states at any instant of time (difficult to design large circuits)
  - synchronous : changes its outputs and internal states at discrete points of time
  - classify : whether has clock (The Clock is a periodic external input to the circuit)



(a) Block diagram

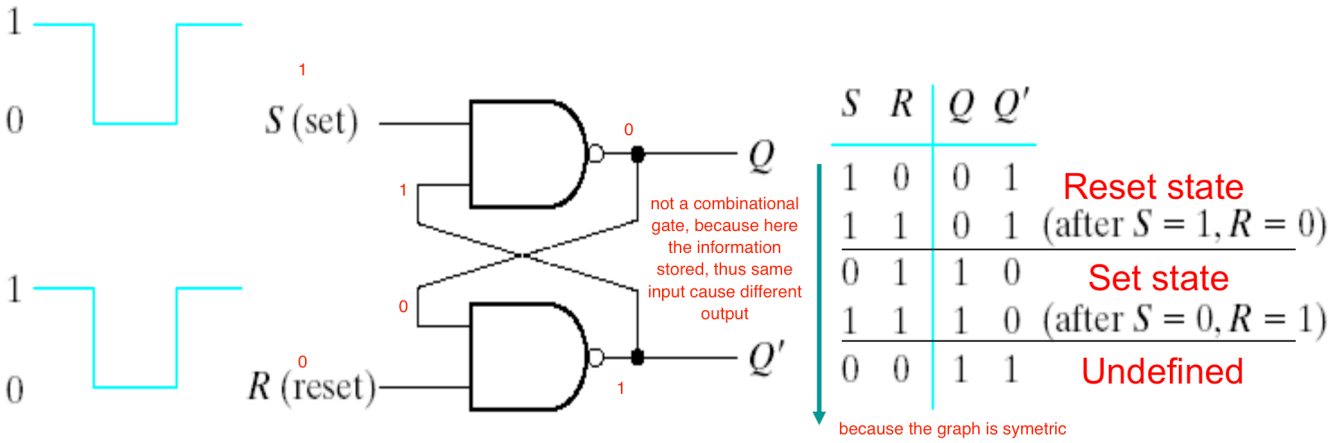


(b) Timing diagram of clock pulses

Latches and flip-flops

- SR Latch with NAND gates
  - initially , set the S and R to be 1 => The circuit is balanced and not changed
  - Set *Reset value* to be 0 (push the reset button), Q becomes 0
  - Set *Set value* to be 0 (push the set button), Q becomes 1
  - push one button once will change the value, push twice continuously will just change the value once
  - in SR latch, the value of Q is always different with S

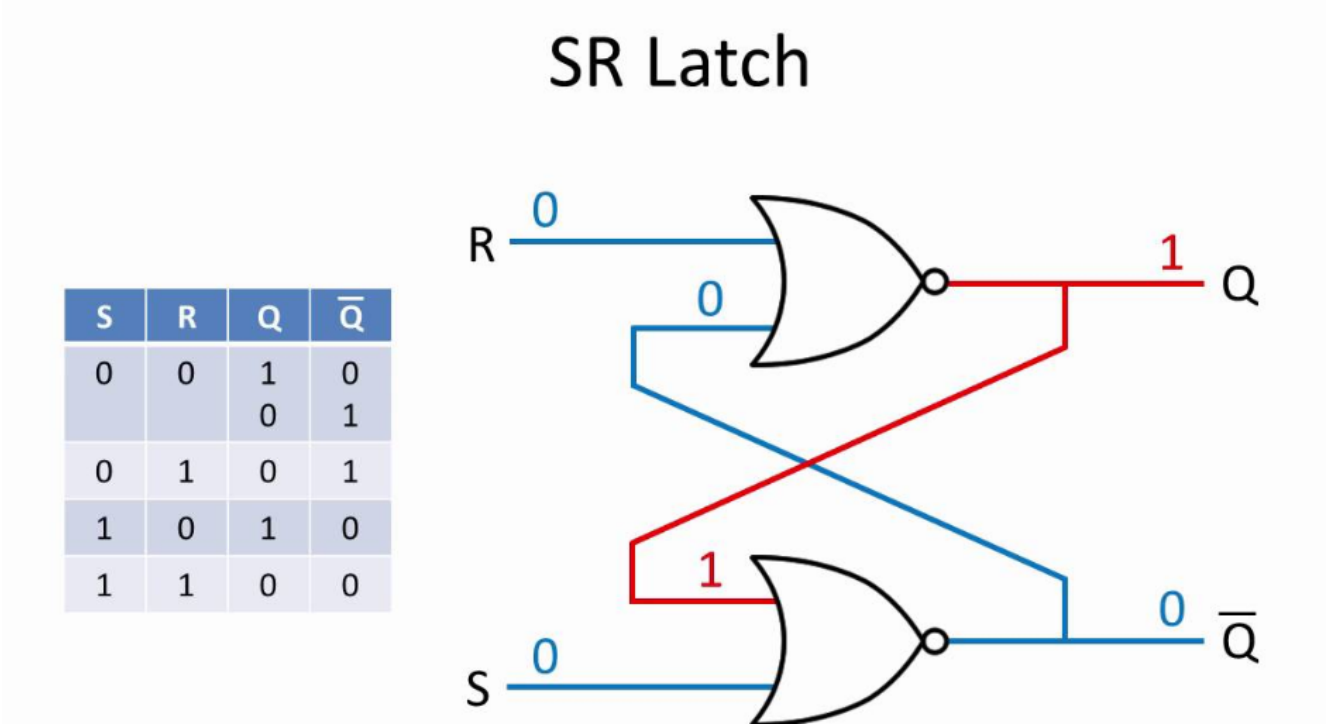
The SR latch is a circuit with two cross-coupled NOR gates or two cross-coupled NAND gates. It has two inputs labeled S for set and R for reset.



(a) Logic diagram (b) Function table

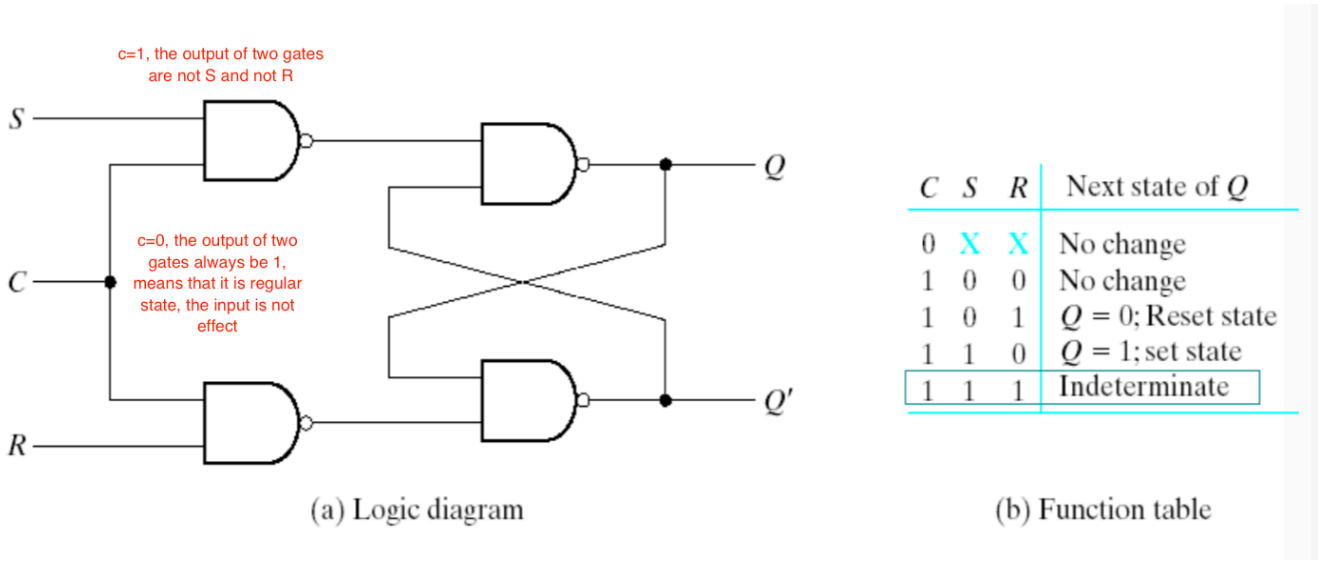
SR Latch with NAND Gates

- SR Latch with NOR Gates



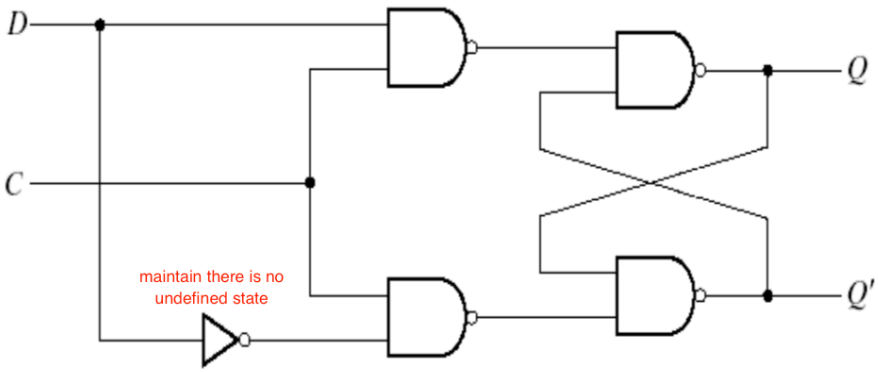
- The Q always equals S
- 0, 0 is the balance state; 1,1 is the undefined state

- Set control input



- If not valid, keep the stable status of the SR latch
- If valid, change s to not s, R to not R and run SR latch
  - now if S is 1 ,R is 0 then Q becomes 1
  - now if S is 0 ,R is 1 then Q becomes 0
  - Q equals S => The SR latch store the value of S

- D Latch => One way to eliminate the undesirable condition (0,0) , S and R cannot equal to 1 at the same time



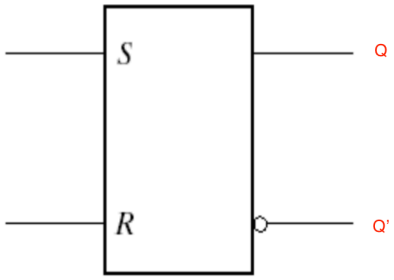
(a) Logic diagram

C	D	Next state of Q
0	X	No change
1	0	$Q = 0$; Reset state
1	1	$Q = 1$; Set state

(b) Function table

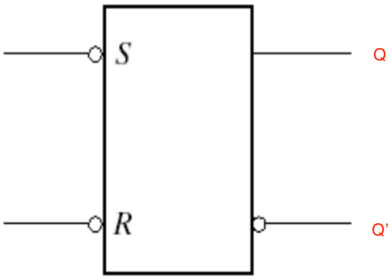
- reduce the indetermined states, and Store D as Q

- Graphic Symbol



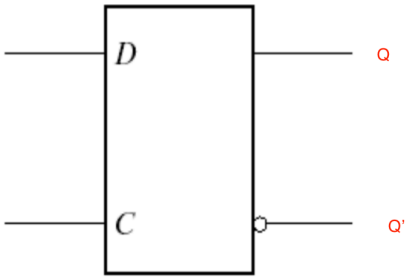
SR latch with NOR gates

SR



SR latch with NAND gates

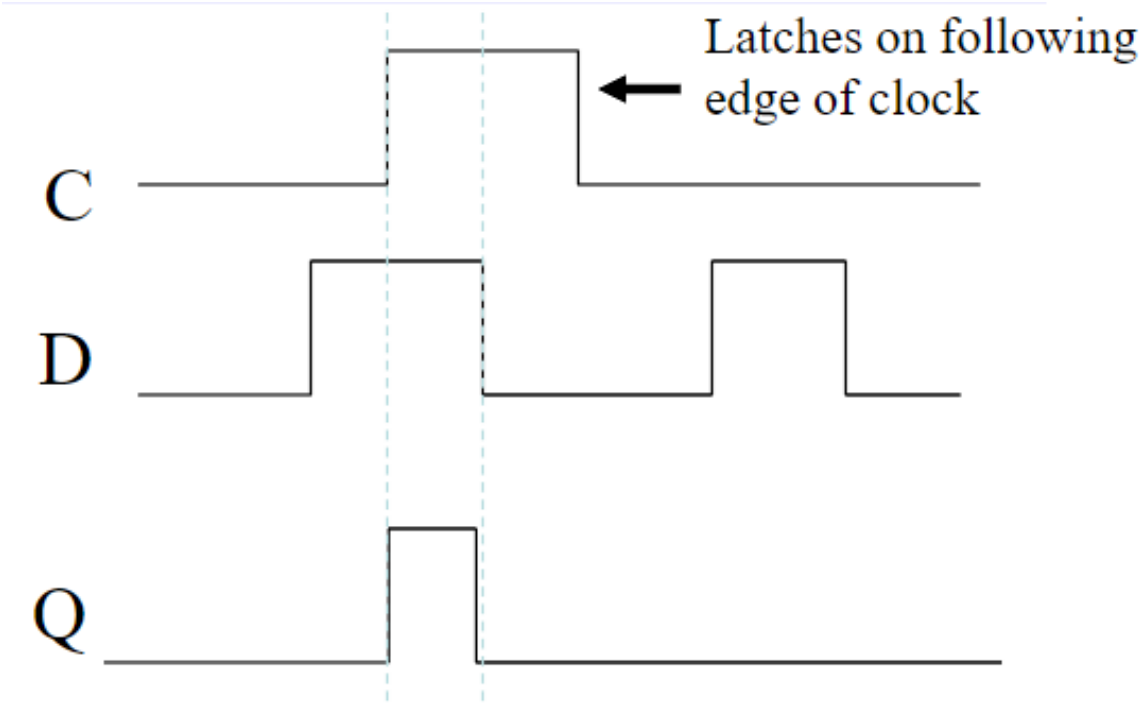
$\overline{S}\overline{R}$



D latch

D

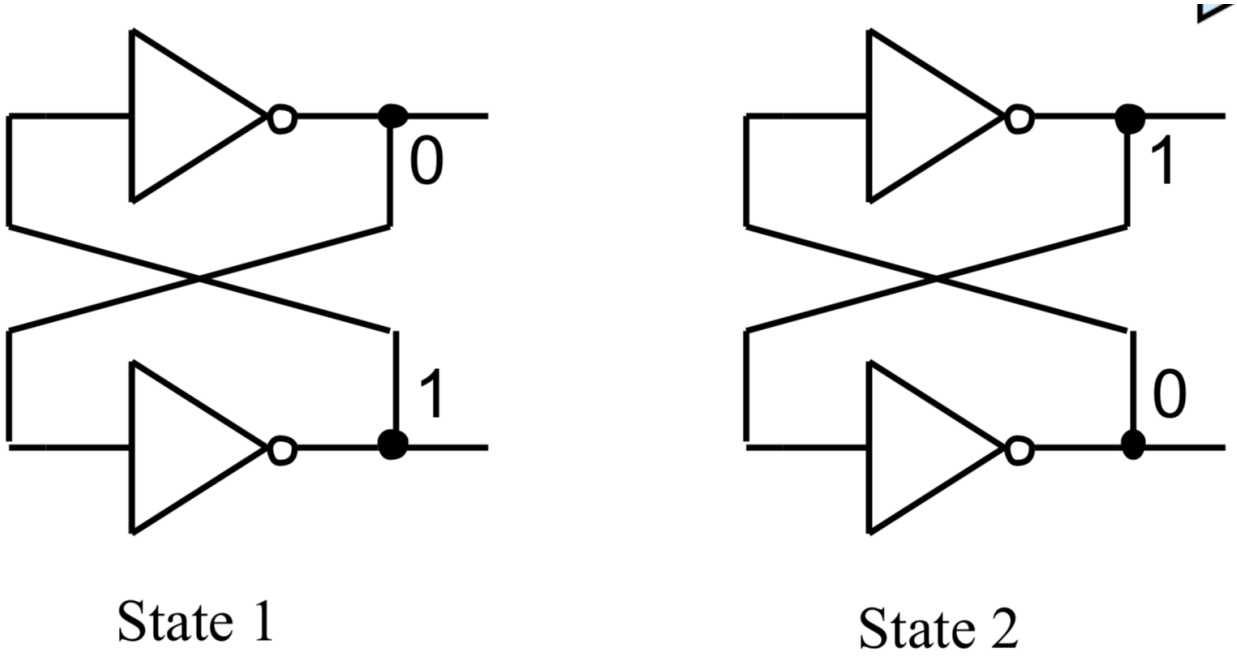
- Clock Response of D latch



- when clock is true, Q will change with the input

- find the intervals that C is true, draw down a line and move the input part to the output part, keep the value when CLK is 0

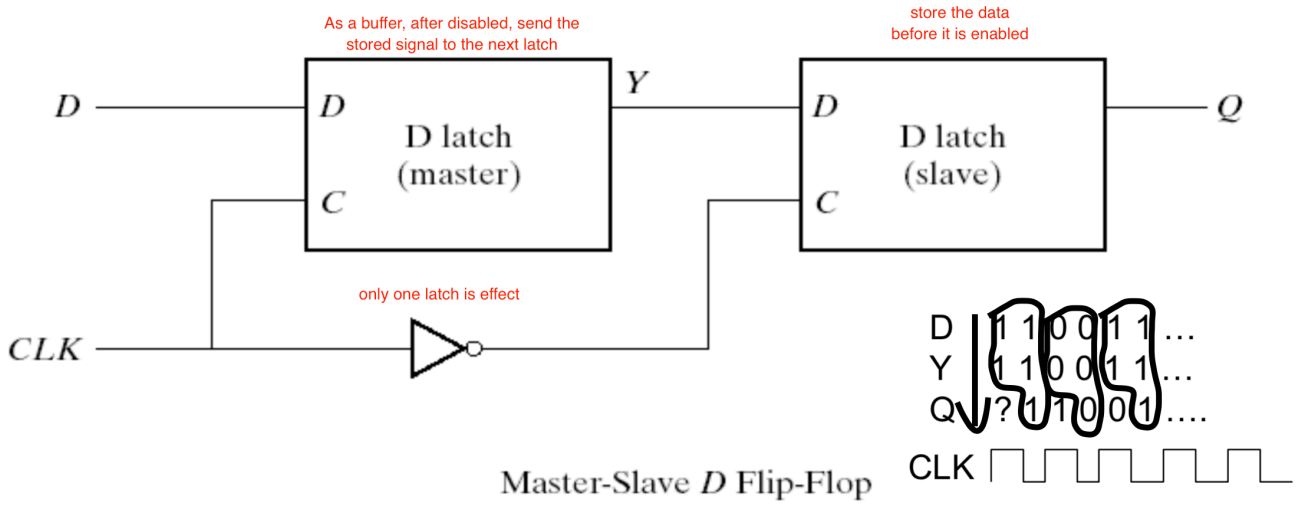
- Flip-Flops with Inverters



- Edge-Triggered D Flip-Flop

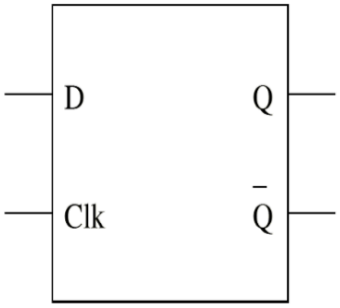
The first latch is called the master and the second the slave. The circuit samples the D input and changes its output Q only at the negative-edge of the controlling clock.

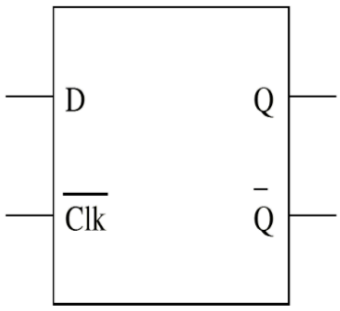
Store 1-bit information.

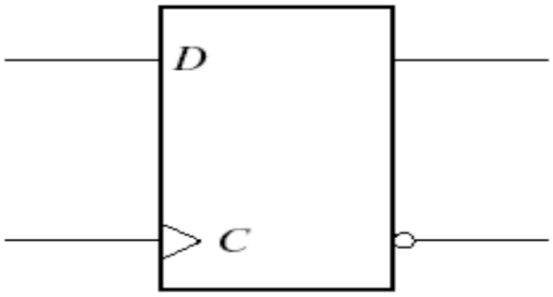


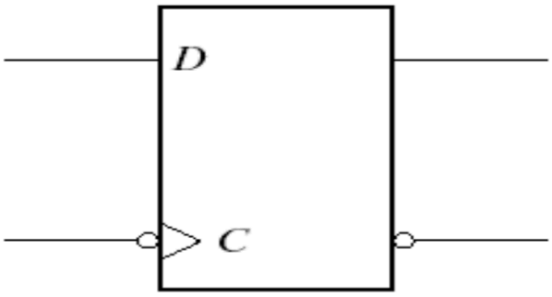
  - Just used to maintain at the time that CLK changed, Y will a valid value to change Q (but Y will not change because D is not valid now), implement the edge triggered property
  - In this case, Q only changes when the CLK turns to negative, and it will not change with D, because D is not effect now

- Representation
  - **positive** edge means **Q will only change at the point CLK turns to 1**
  - **negative** edge means **Q will only change at the point CLK turns to 0**





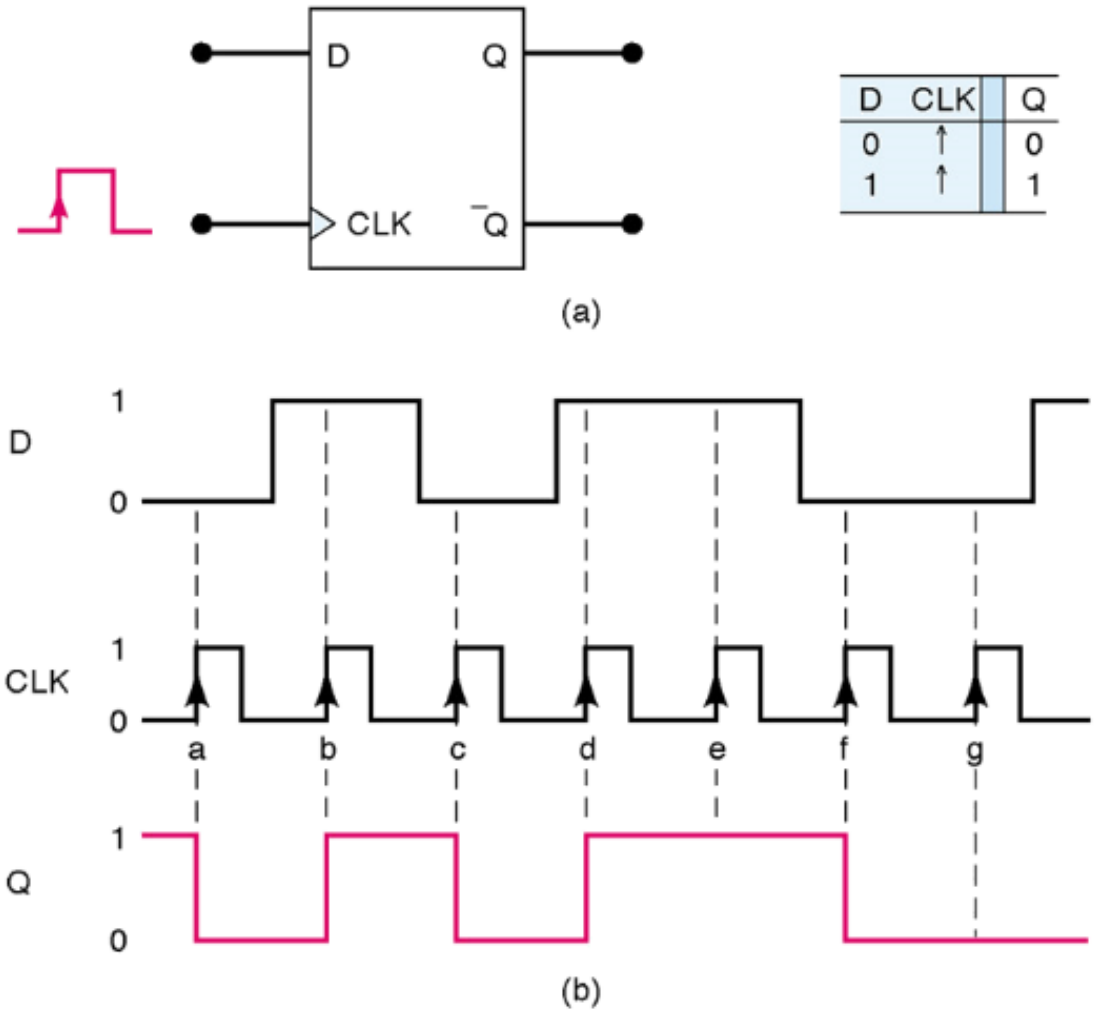




(a) Positive-edge

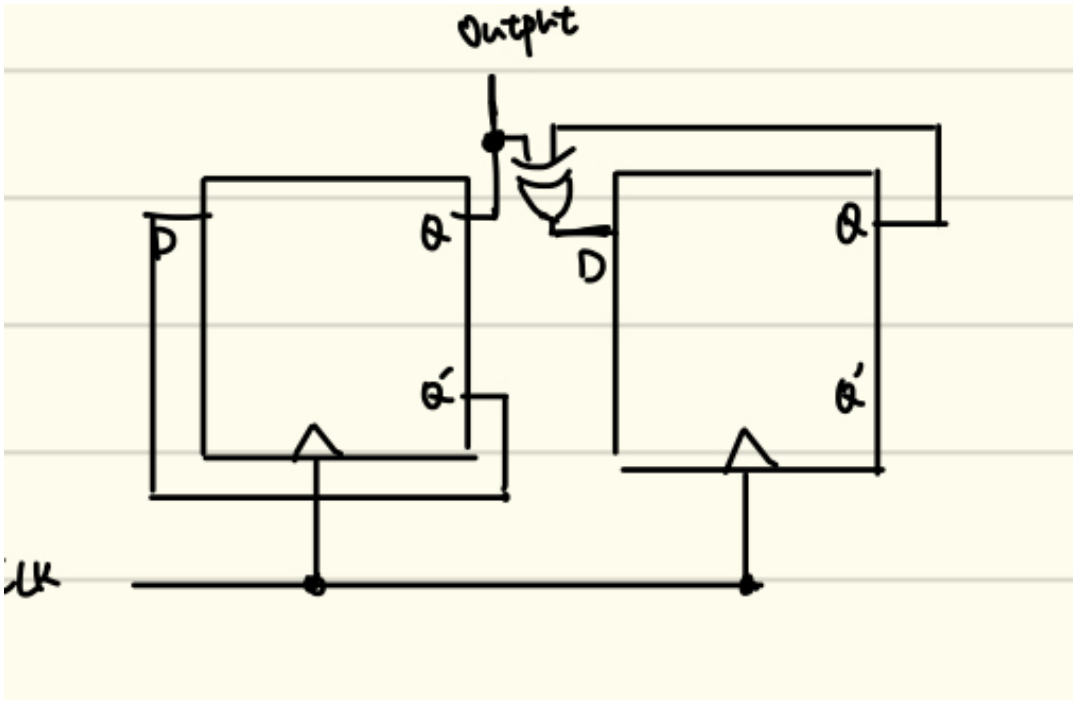
(a) Negative-edge

- Difference :
  - Latch copy the data when Clock signal is 1
  - Flip-Flops just change the value when the Clock signal changed
    - Positive : change when the clock from 0 to 1
    - Negative : change when the clock from 1 to 0
  - Find the critical points and draw line, copy the initial value of the input as the value during the interval



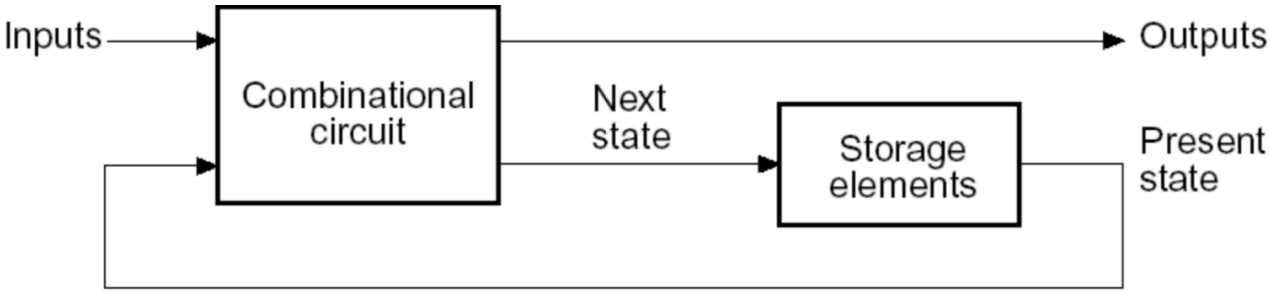
- Counting 0,1,2,3
  - **Input: start signal**
  - **When the start signal is pressed, output a sequence of binary numbers 0 1 2 3, each one in a clock cycle. Wait until next time the start signal is pressed and repeat the same.**

first bit: every time change
next bit: depends on the first bit(if it is 1, next bit changes)



Finite State Machine

- FSM => (external inputs, externally visible outputs, internal states)
  - output and next state depend on => inputs&present state



$$M = (S,\ I,\ O,\ \delta)$$

S: Finite set of states (internal)

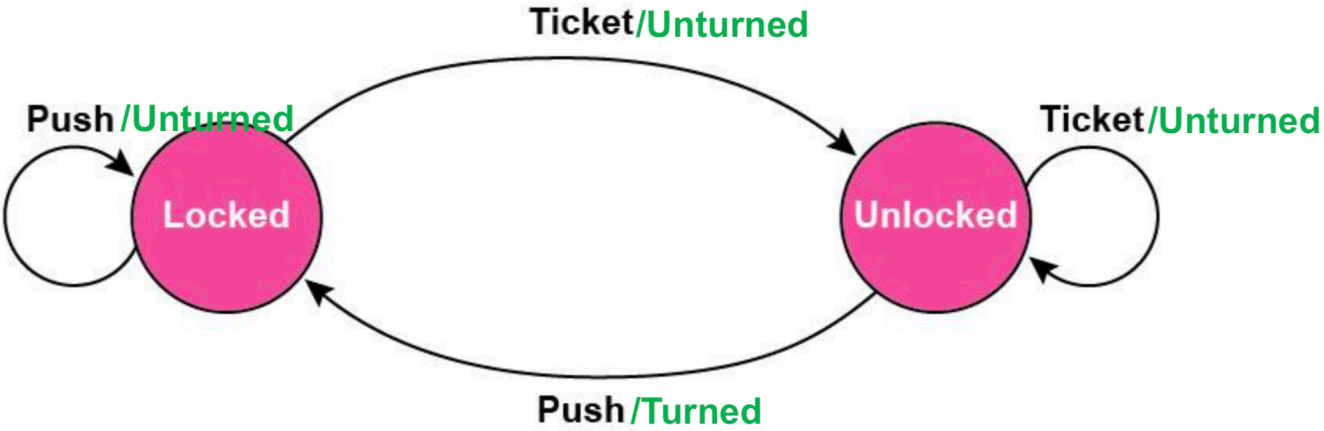
I: Finite set of inputs (external)

O: Finite set of outputs

present output = O (present state, present input)

δ: State transition function

next state = δ (present state, present input)

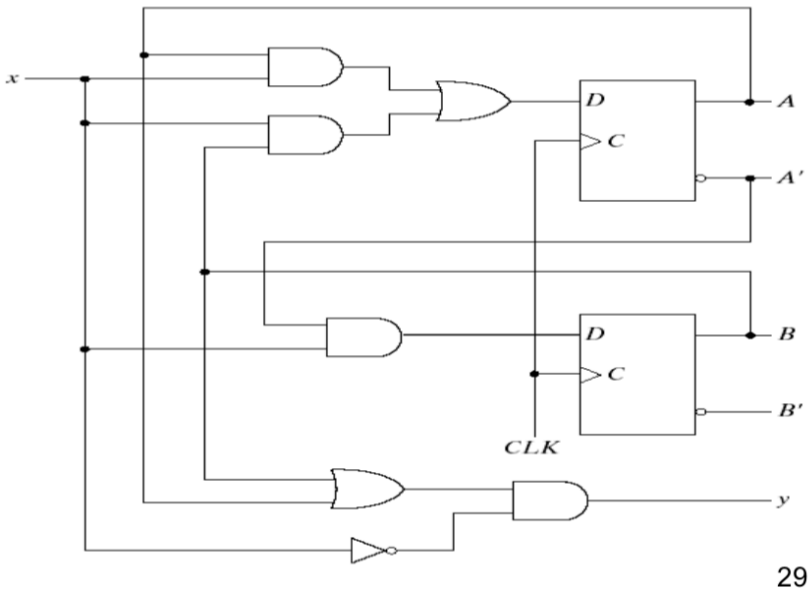


- State equation => use the circuit to summarize an equation
  - output = function1(present input, present state)
  - Next state = function2(present input, present state)

$A(t+1) = A(t)\,x(t) + B(t)\,x(t)$

$B(t+1) = A'(t)\,x(t)$

$Y(t) = (A(t) + B(t))\,x'(t)$



29

- State Table : Use the equation to draw a partial Truth table (partial because the input may not exist)

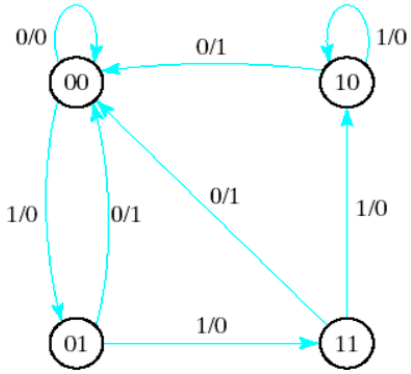
Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Don't have these two later

30

- State Diagram => use the table to draw the diagram (state in the circle, event is consist of input/output, event changes state)

- **Detect whether the input signal is of the following pattern**

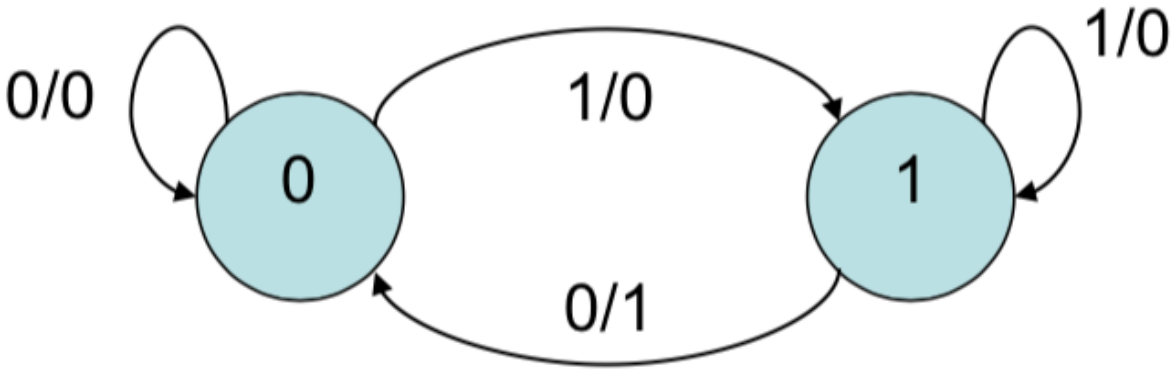


  - Initial state: 00. Sequential inputs.
  - If following this pattern, output 0, else output 1.
  - Examples

Input stream Output stream State stream

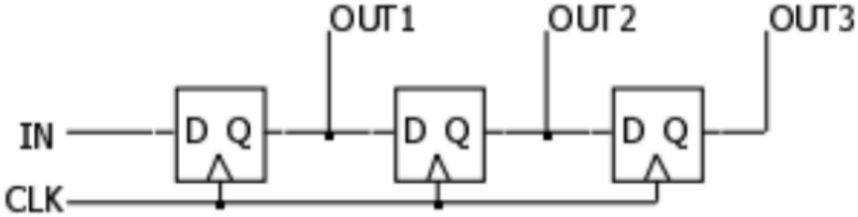
    - 001111 → 000000 (state: initial 00 → 00 → 00 → 01 → 11 → 10 → 10)
    - 001100 → 000010 (state: initial 00 → 00 → 00 → 01 → 11 → 00 → 00)

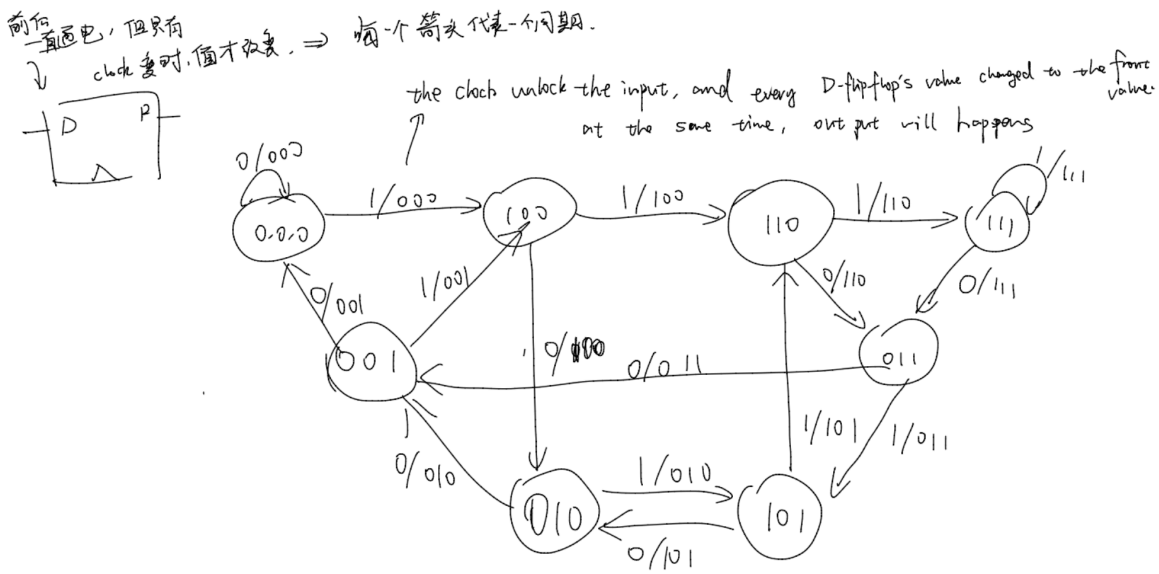
- Simplification (merge 3 states to a cluster, because when they are merged into 1 cluster, the behavior will be same) merge 01,11,10 to 1 => similar to combinational circuit
  - can be done when there are a lot of similar transitions



- Draw a state graph

2. Draw the state transition diagrams for the three-bit shift register in the below diagram with different initial values in the shift register.



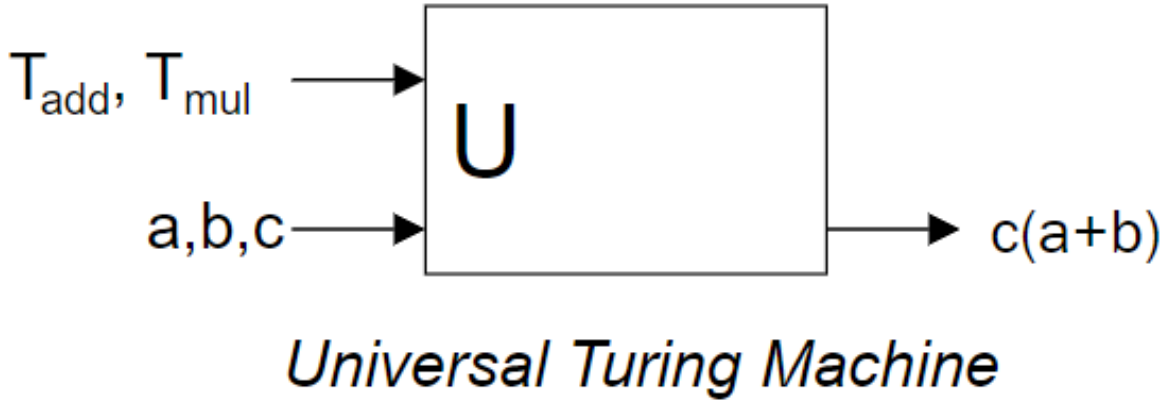


Lec05 Computer Architecture Overview

Universal Computing Device

All computers, given enough time and memory, are capable of computing exactly the same things

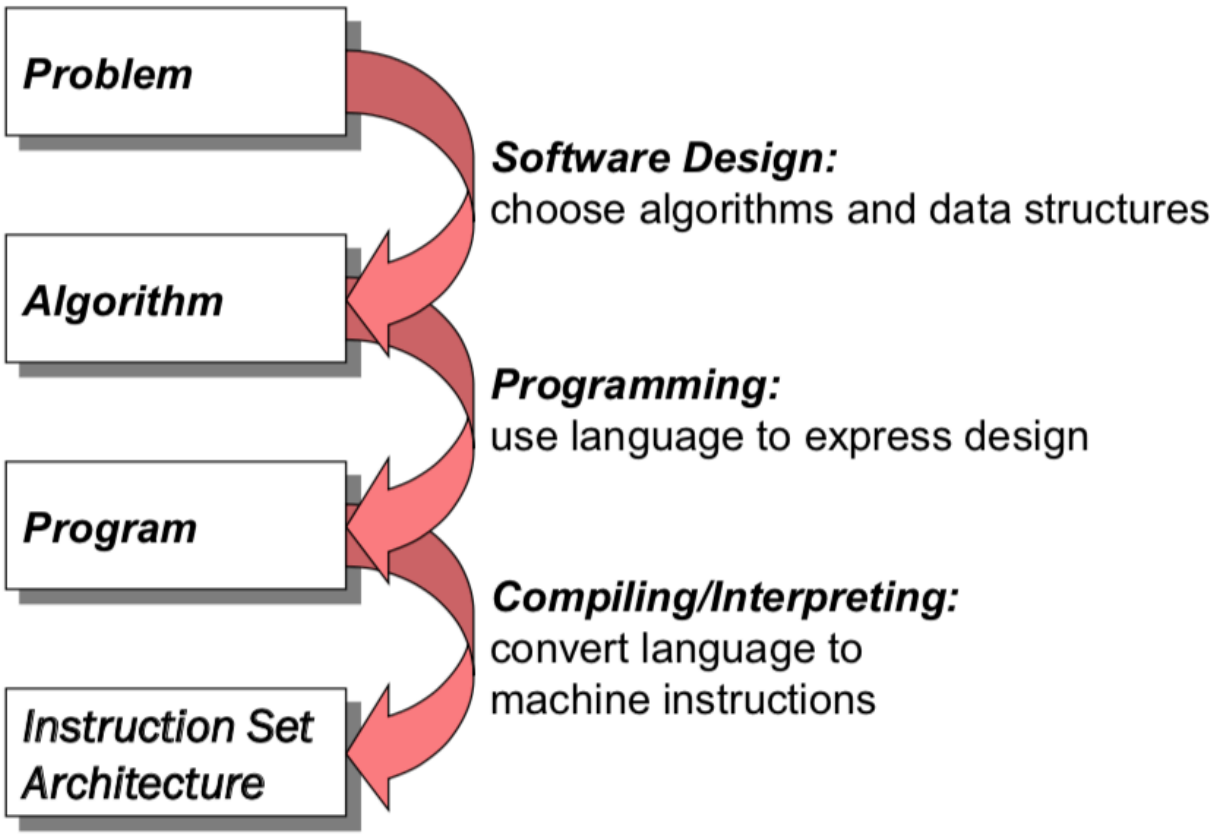
- Turing machine : Mathmatical model of a device that can perform any computation （Turing's thesis : Every computation can be performed by some turing machine）
  - Tape => store a finite set of alphabets
  - Head => read and write from/to the tape, move left or right one cell at a time
  - state register => with a start state (and often accept state)
  - Finite table of instructions : given current state and the symbol read pointed by the head
    - Write/erase symbol
    - move head
    - state transition
  - ability to read/write symbols on an infinite "tape"
  - State transitions, based on current state
- Universal Turing machine (extends to Turing machine)

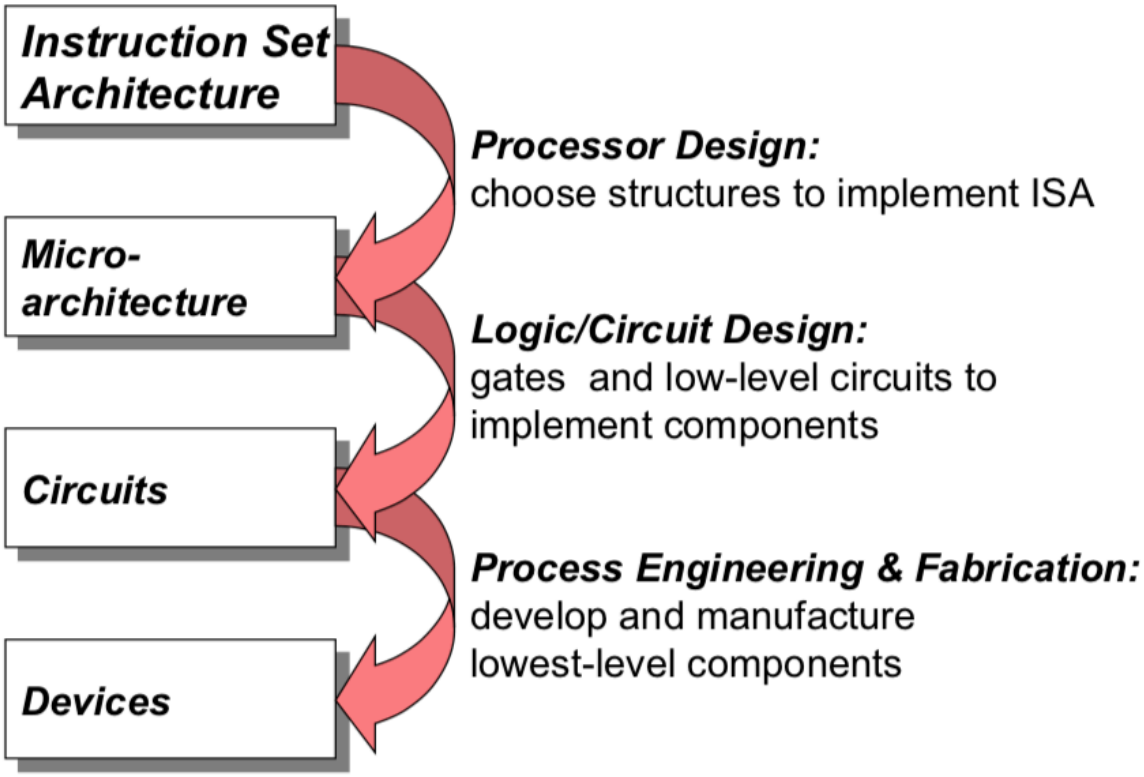


  - treat instructions at part of input data (command as Input)

    - can implement all Turing machines
    - Can be programmed (programmable)
    - Can be emulate by a computer
- Theory to practice : time, memory, cost, power.. are limited

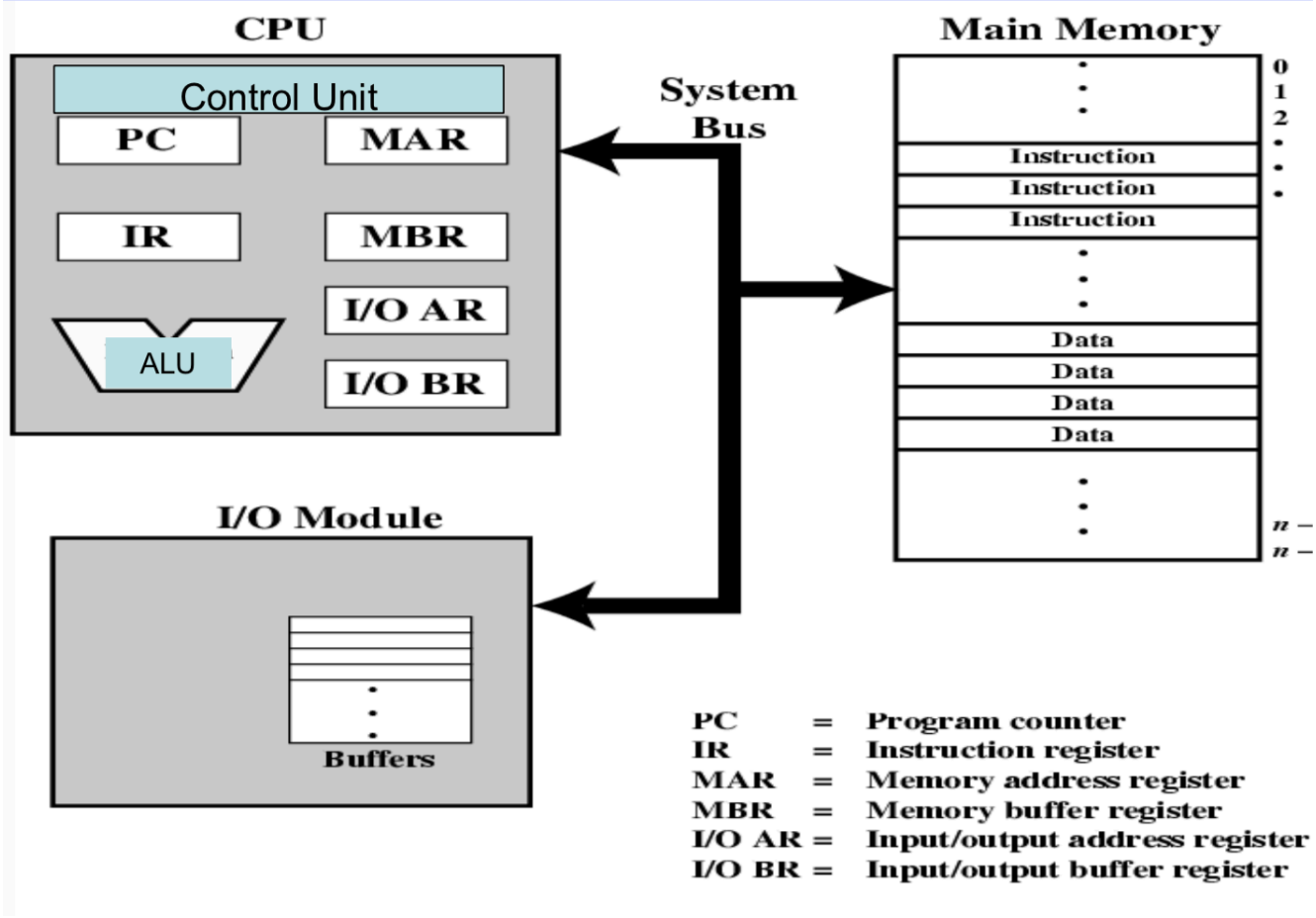
Transformations between layers

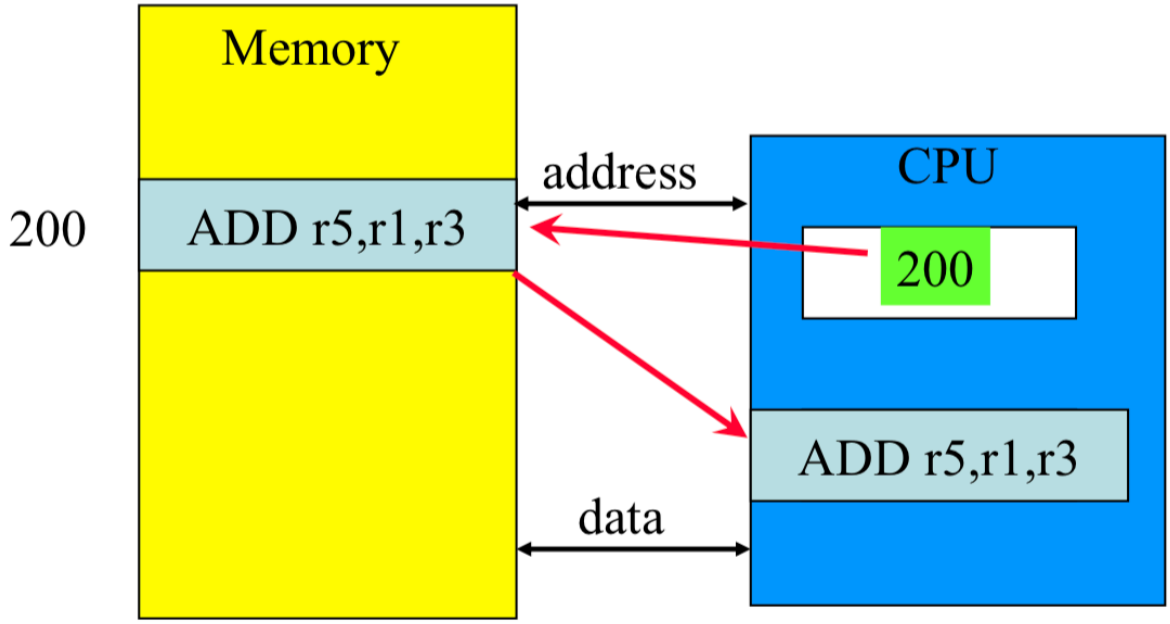




Hardware: Microarchitecture

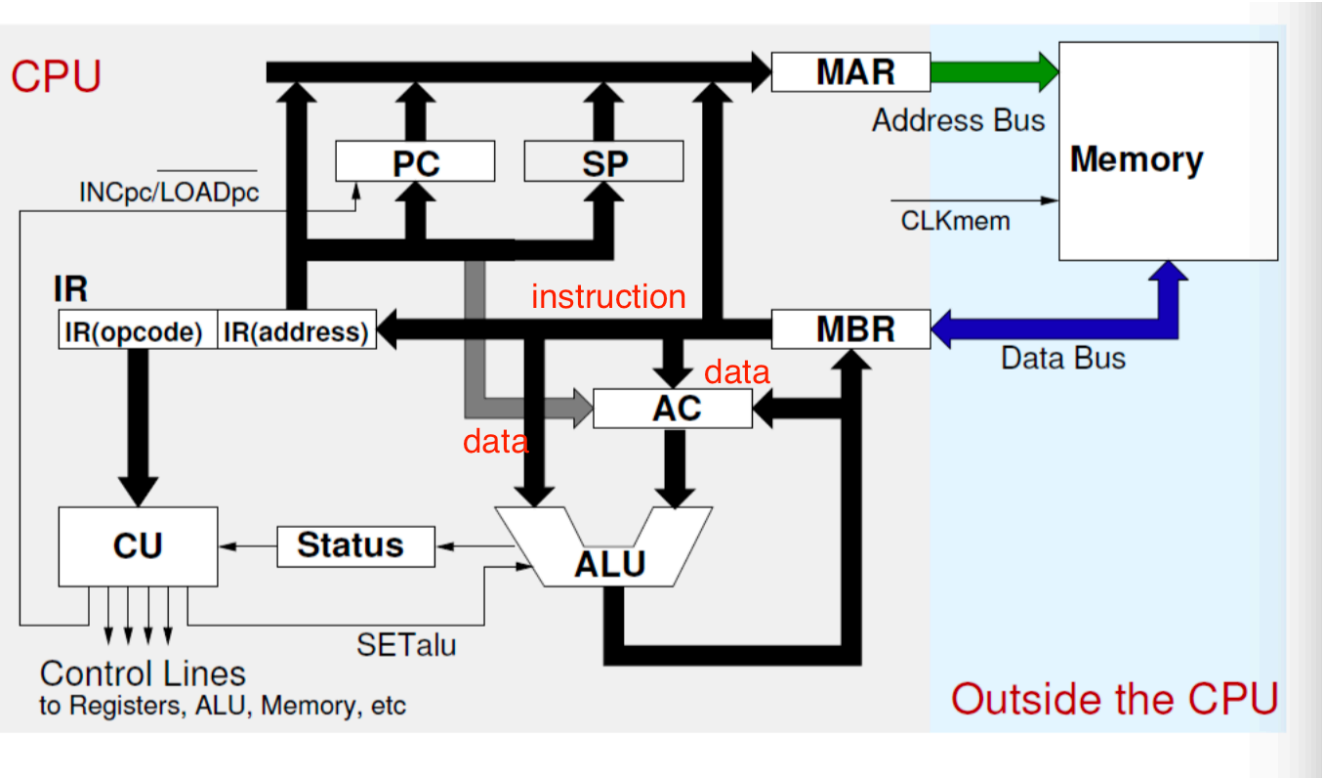
- CPU : constructed from digital logic gates
  - CA : Central Arithmetic part
  - CC : Central Control part
- System bus
- Memory : store instructions as well as data
- Von Neumann Architecture
  - Components
    - Central Arithmetic part (CA)
    - Central Control part (CC)
    - Memory (M)
    - Input (I)
    - Output (O)
  - Use binary instead of decimal system
  - Single storage structure hold both instructions and data in binary form
  - Computer can fetch instructions and execute them automatically





- CPU Registers
  - MAR : Store the address to access memory
  - MBR : Store information that is being sent to, or received from, the memory along the bidirectional data bus
  - AC : **Accumulator** used to store data that is being worked on by the ALU and is the key register in the data section of the CPU (store some intermediate value of calculation) (memory cannot directly access, but can go through MBR)

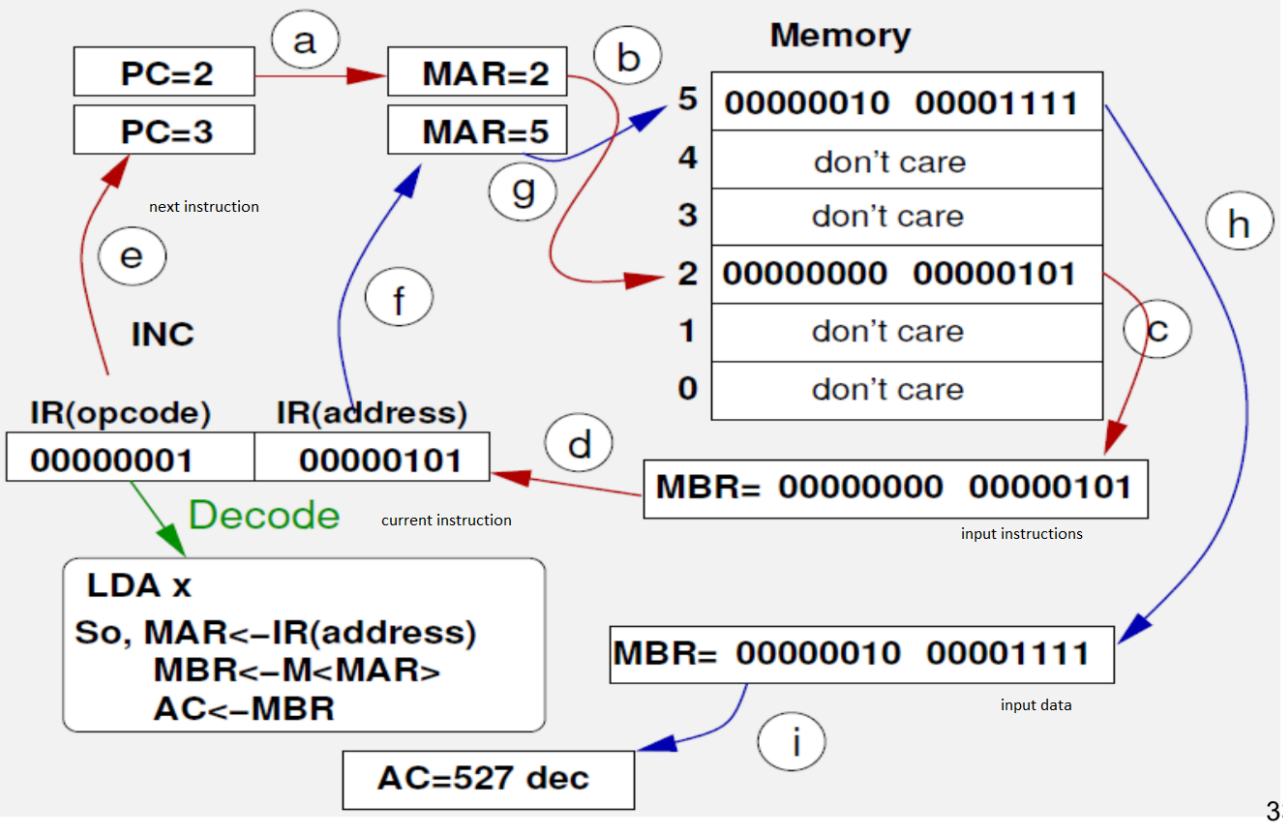
- PC : The **Program Counter** holds the address in memory of the next program instruction (doesn't connect directly to memory but via MAR)(both a register and a counter(计数器))(send to memory)
- IR (get from and send to memory): When memory is read, the data first goes to the MBR. If the data is an **instruction** it gets moved to the **Instruction Register** , has 2 parts
  - IR (opcode) : Store the most significant bits of instruction, tells CPU what to do (instruction here gets decoded and executed by the CU) => instruction part
  - IR (address) : Store the least significant bits As the name suggests they usually form all or part of an address for later use in MAR => **address of data**
- SP : **Stack Pointer** connected to the internal address bus used to hold address of **special chunk of main memory** used for temp storage during program execution



- Instruction Cycle : Each instruction in an assembly-language program goes through two states
  - Fetch (Deliver the instruction stored at main memory to the CPU)
    - PC holds the address of next instruction to fetch
    - CPU fetches instruction from memory location pointed to by PC
    - Increment PC
    - Instruction loaded into IR
    - processor interprets instruction and performs required actions
  - Execute (Execute the fetched instruction in the CPU to completion)
    - Data transfer (load/store moving data from CPU to memory)
    - Data processing
    - Control to modify program flow

```
while(somecondition){
    fetch();
    execute();
}
```

Example: Loading Data



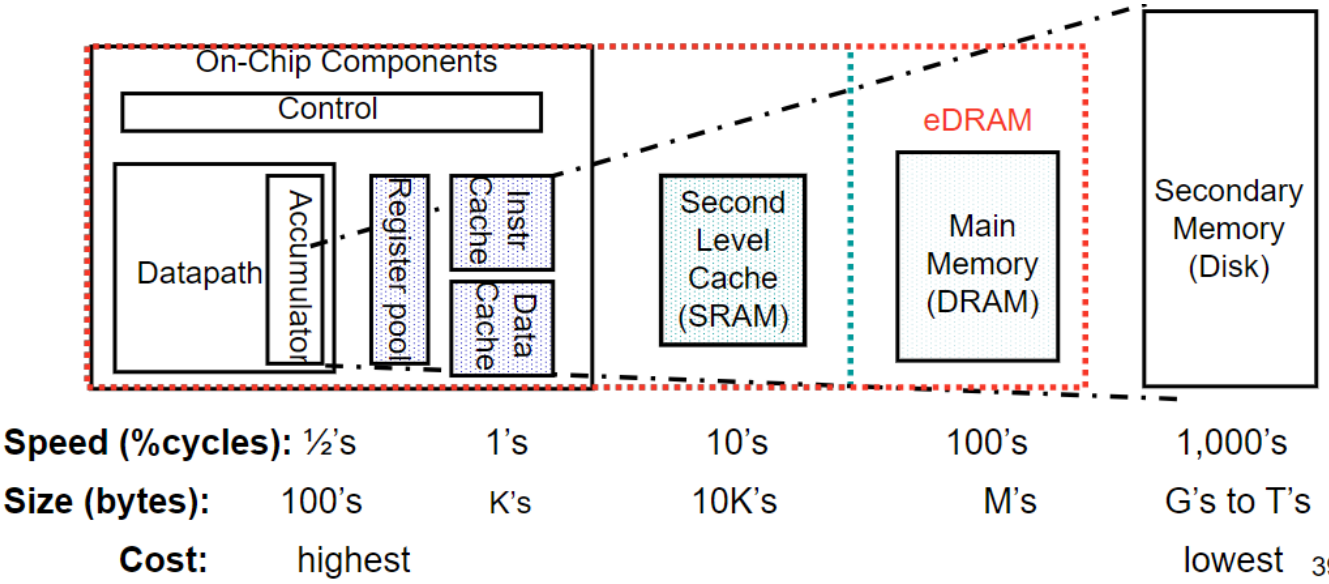
- Instruction Set Architecture :
  - what it is
    - interface between hardware and software
    - operations to CPU (A complete collection of instructions to control the CPU)
    - standard set of instructions
    - registers addressing mode
  - Compose
    - Operation Code
    - Source Operands reference / Destination Operands reference / Next Instruction Reference
  - e.g. :
    - `add $t1,$t2,$t3` ("$" prefix denotes a register onboard the CPU)
    - Transfer data between registers and main memory: `lw $t1, 0x12abcdef` (in assembly language programming, `lw` means "load word" to transfer data)
    - Data movement between registers : `mov $t1, $v0`
    - Flow control : `jump loop1`

Memory Hierarchy

Large memories are slow (but cheap) and fast memories are small (but expensive)

Target : large, cheap, fast

Thus exploiting memory hierarchy and principle of locality

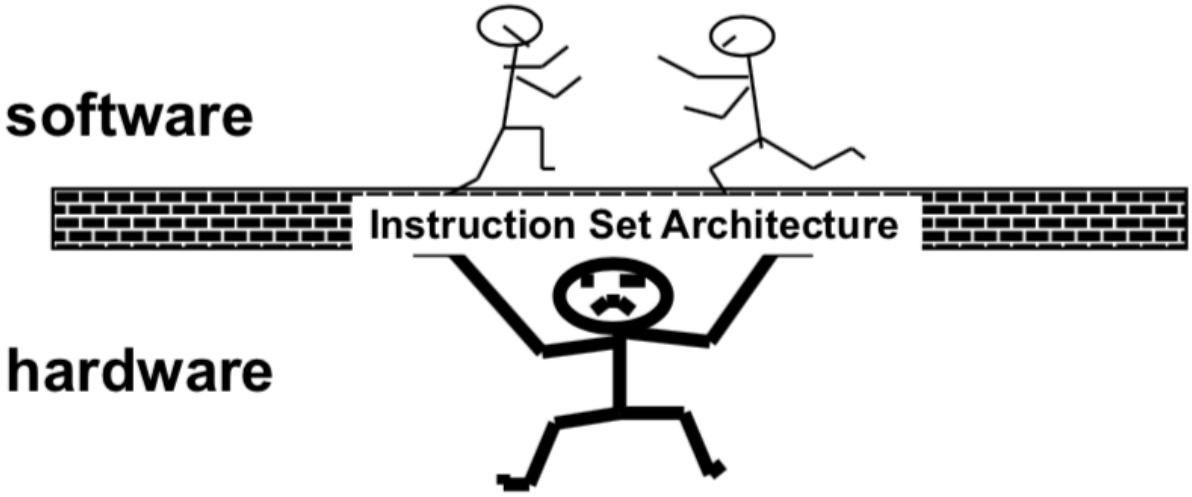


- Cache : A small amount of fast memory hardware component located physically near to the CPU and serves as a "middle man"
    - When requesting contents from main memory, CPU first looks for the content in the cache and, if content is present, retrieves it from the cache. Otherwise, the CPU accesses the main memory directly to fetch the content, which is also placed on the cache.

Lec 06 Instruction Set Architecture

ISA overview

- A very important abstraction:



  - **interface** between hardware and low-level software
  - **standardizes** instructions, machine language bit patterns, etc.
  - advantage: **allows different implementations of the same architecture**
  - disadvantage: sometimes prevents adding new innovations（新方法） (e.g. machine learning to use GPU)
- Interface Design
  - Completeness, orthogonality, regularity and simplicity, compactness(简单紧凑)
  - portability, compatibility (lives long)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels
  - **Issues**:
    - Operation: add, sub, mul
    - Type&size of operands are supported : byte, int, float, double, string
    - operands stored: registers, memory, stack, accumulator
    - How many explicit operands are there: 0,1,2,3
    - Operand location specified: register, immediate, indirect

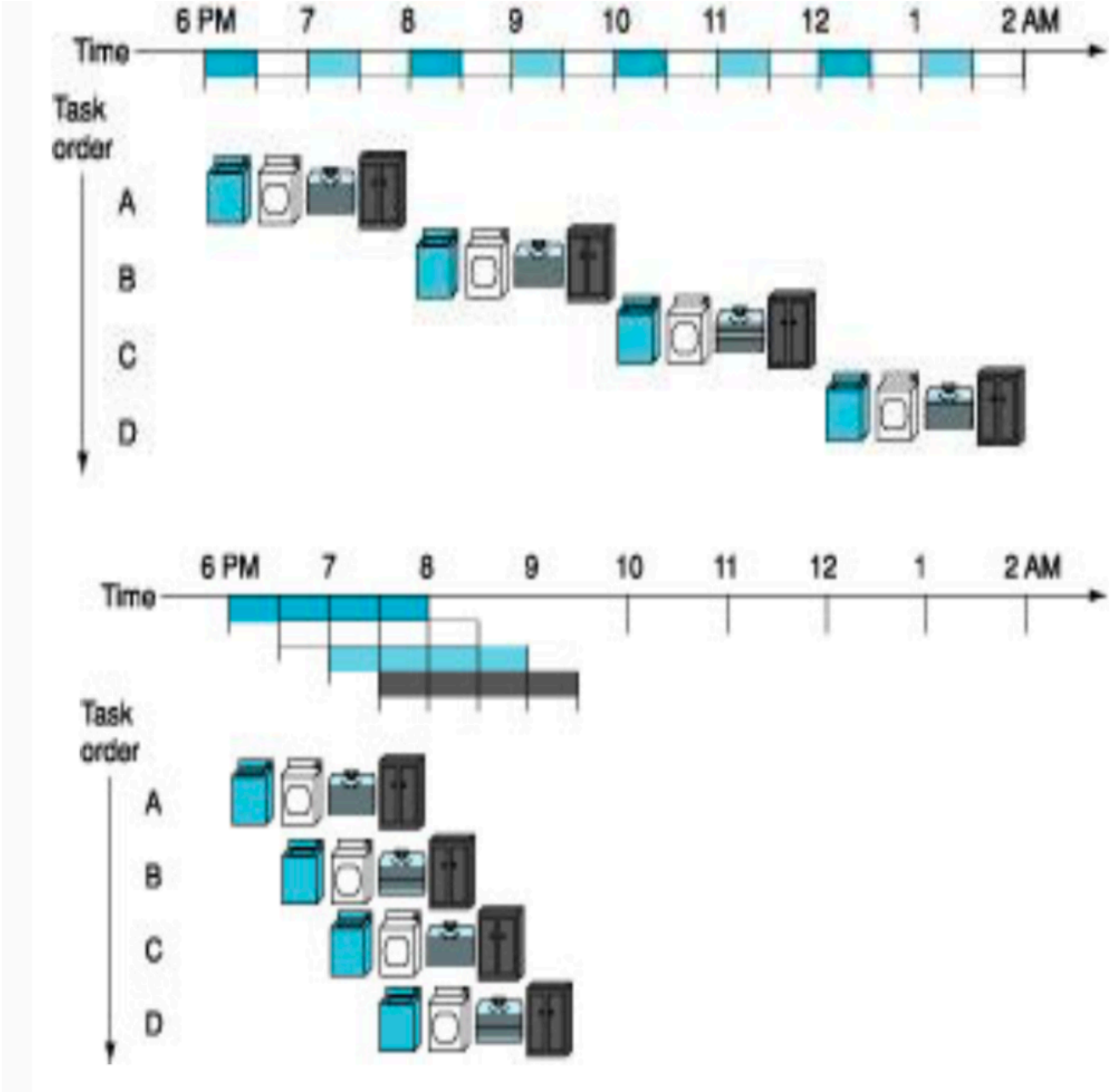
---

CISC vs. RISC

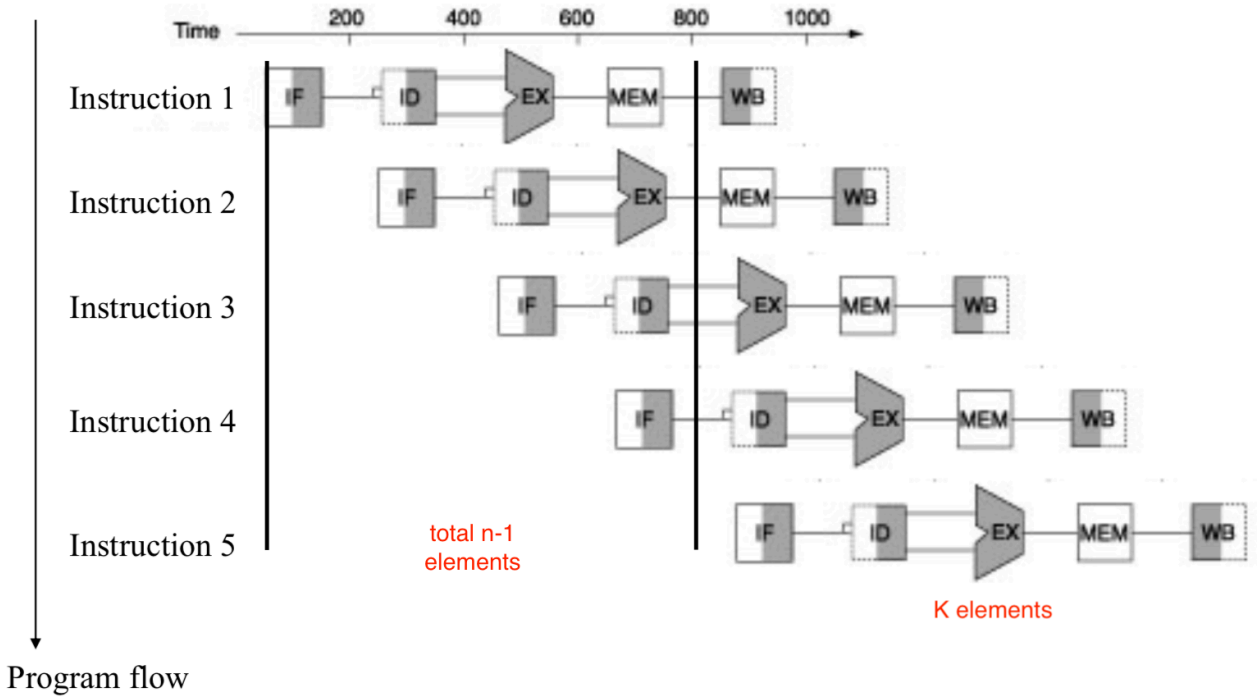
- CISC (**Complex** Instruction Set Computer): Intel x86=>variable length instructions, lots of addressing modes, lots of instructions
- RISC (**Reduced** Instruction Set Computer): MIPS, Sun SPARC, IBM
  - fixed instruction lengths, uniform instruction formats
  - Load-store instruction sets
  - Limited number of addressing modes
  - Limited number of operations

CISC	RISC
large number of instructions and many addressing modes	one instruction per cycle a simple instruction set with a few addressing modes
slower clock rate	fast clock rate
complex control unit, thus requiring microprogrammed implementation	hardwired control Unit
more difficult to pipeline	more efficient pipelining
complex programs require fewer instructions in CISC	RISC requires a large number of instructions to accomplish the same task

- Pipeline :



  - **pipeline** : A set of data processing elements connected in series, output of one element is input of next element
  - Divide a task into small pieces (data processing elements) connected in series, where output of one element is the input of the next one
    - Thus the **elements**(洗衣机) of a pipeline are often executed in **parallel** (originally, the task is not divided, thus the used part of the elements cannot be used for next operation during the first operation is not finished)
  - More number of stages => Enhances parallelism and speeds up computation
- Analysis of Instruction Cycle Pipeline
  - The execution of an instruction cycle can be divided into several stages
    - IF : fetch instructions
    - ID: decode instruction
    - EX: Execute instruction
    - MEM : Memory Operation
    - WB: Write back result to registers/memories



- Suppose one clock cycle lasts τ seconds, there are K stages for each instruction and there are n instructions in total
  - Computer with pipeline :

$$T_{pipeline} = K\tau + (n-1)\tau \tag{8}$$

  - Comupter without pipeline :

$$T_{non-pipeline} = Kn\tau \tag{9}$$

  - Speed Ratio :

$$\frac{T_{non-pipeline}}{T_{pipeline}} = \frac{Kn}{K+n-1} = \frac{K}{\frac{K-1}{n}+1} \tag{10}$$

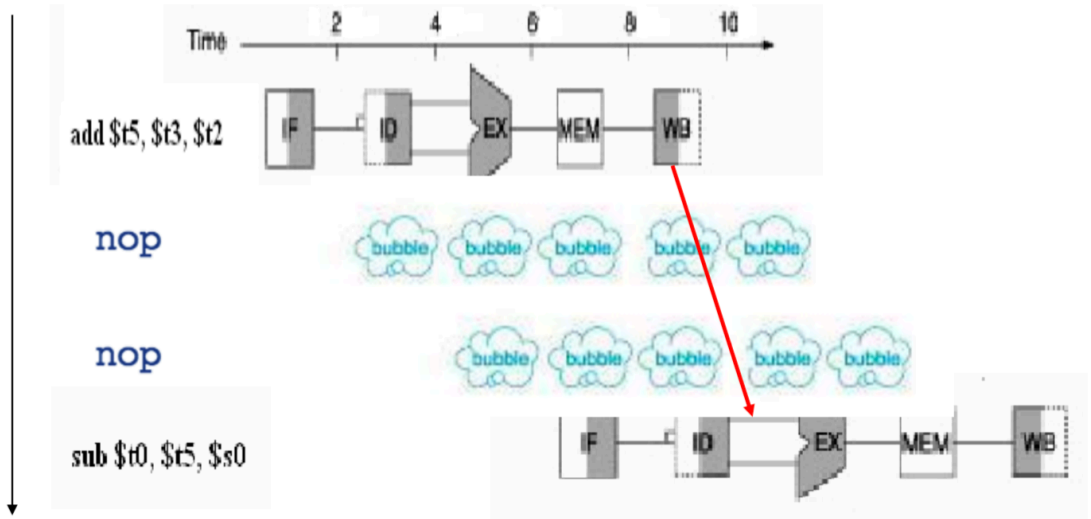
    - As n goes to infinity, the value goes to K
  - Instruction Throughput of a computer with pipeline(average number of instructions per unit time):

$$\frac{n}{(K+n-1)\tau} = \frac{1}{(\frac{K-1}{n}+1)\tau} \tag{11}$$

    - as n goes to infinity, the value goes to $\frac{1}{\tau}$

- Pipeline Hazard(流水线风险): Situations whereby the next instruction should not be allowed to execute in the following clock cycle
  - *Control Hazards* : Dependency on the **execution outcome of a previous instruction** leads to uncertainty on what the next correct instruction should be (转移或异常改变执行流程，顺序执行指令在目标地址产生前已被取出)
  - Data Hazards: Dependency on shared data used by a previous instruction still in pipeline (shared data,data is used without initialization)

- Insert No-operation (nop) instruction between instructions that have dependencies
- Compiler software automates the task of inserting nop in program to resolve hazards
- Speedup ratio of pipelining is thus much less than the number of stages in general



Classifying ISAs

Accumulator (before 1960, e.g. 68HC11):

1-address	add A	acc ← acc + mem[A]

Stack (1960s to 1970s):

0-address	add	tos ← tos + next

Memory-Memory (1970s to 1980s):

2-address	add A, B	mem[A] ← mem[A] + mem[B]
3-address	add A, B, C	mem[A] ← mem[B] + mem[C]

Register-Memory (1970s to present, e.g. 80x86):

2-address	add R1, A	R1 ← R1 + mem[A]
	load R1, A	R1 ← mem[A]

Register-Register (Load/Store) (1960s to present, e.g. MIPS):

3-address	add R1, R2, R3	R1 ← R2 + R3
	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

Registers

- Benefit of general purpose registers:
  - much faster(variable attained immediately)
  - covenient for tmp variable storage

- Advantages
  - Faster than cache or main memory (no addressing mode or tags)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic

- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number

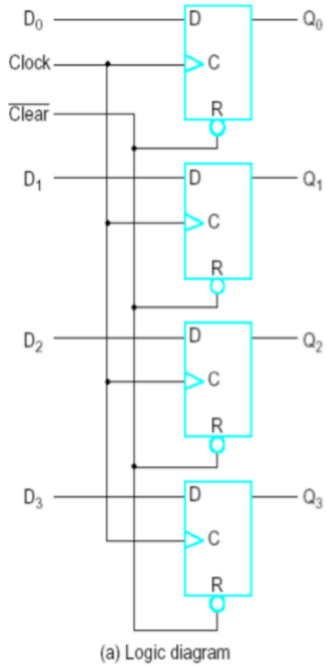
- A flip-flop can be used to store bit => a register can handle data

A FF can be used to store one bit.

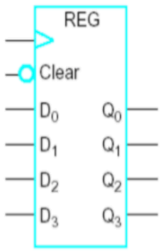
A register can handle data in multiple bits.

A 4-bit register

- To write new data into the register, a positive clock transition applied to the *Clock* input of the register will load all four *D* inputs into the flip-flops in parallel.
- To read data stored in the register, simply sample the four *Q* outputs from the flip-flops.
- The $\overline{\text{Clear}}$ input is used for clearing the register to all 0's in a system reset.



(a) Logic diagram



(b) Symbol

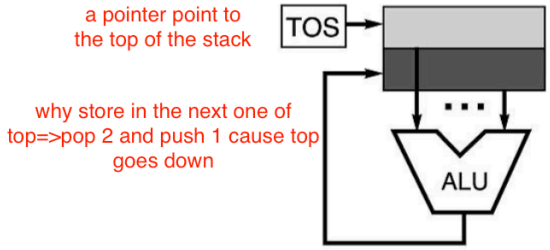
- Shift Register (4-bit shift register)
  - It consists of D flip-flop

- Each clock pulse will cause the transfer of the contents of F_i to F_{i+1} => effecting a "right "
- Data are shifted serially into and out of the register

- Stacks
  - **Instruction set:**

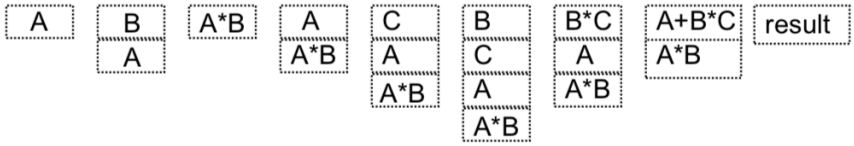
    add, sub, mult, div, . . .

    push A, pop A



  - **Example: A*B - (A+C*B)**

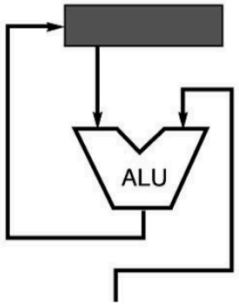
    push A
    push B
    mul
    push A
    push C
    push B
    mul
    add
    sub



    - data has originally pushed into the stack
    - pros :
      - good code density (implicit top of stack)
      - Low Hardware requirements (than memory-memory)
      - Easy to write compiler
    - Cons :
      - must use stack
      - little ability for parallelism of pipelining
      - Data is not always at the memory block pointed by the TOS pointer => need additional instructions like TOP and SWAP
      - Difficult to write an optimizing compiler
- Accumulator

Accumulator Architectures

- Instruction set:
  add A, sub A, mult A, div A, . . .
  load A, store A



acc = acc +,-,*,/ mem[A]

- Example: A*B - (A+C*B)
  load B
  mul C
  add A
  store D put D to the memory as result of previous
  load A
  mul B
  sub D

B	B*C	A+B*C	A+B*C	A	A*B	result

  - Pros :
    - low hardware requirments
    - Easy enough
  - Cons :
    - must need accumulator
    - little ability for parallelism or pipelining
    - High memory traffic
- Memory-Memory

Memory-Memory Architectures

- Instruction set:

(3 operands) add A, B, C	sub A, B, C	mul A, B, C
(2 operands) add A, B	sub A, B	mul A, B

- Example: A*B - (A+C*B)
  - 3 operands

```
mul D, A, B
mul E, C, B
add E, A, E
sub E, D, E
```

  2 operands

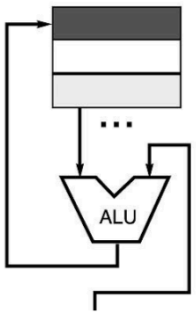
```
mov D, A
mul D, B
mov E, C
mul E, B
add E, A
sub E, D
```

    - Data is store to the first one after the operator
    - pros :
      - Requires fewer instructions
      - Easy to writer compiler
    - Cons :
      - Very high memory traffic
      - Variable(可变的) number of clocks per instruction
      - variables are not equivalent
      - With 2 operands, more data movements are required
- Register-Memory

Register-Memory Architectures

- Instruction set:

```
add R1,  A        sub R1, A        mul R1, B
load R1, A        store R1, A
```



R1 = R1 +,-,*,/ mem[B]

- Example: A*B - (A+C*B)

```
load R1, A
mul R1, B          /*    A*B          */
store R1, D        Store R1 into D in the main memory
load R2, C
mul R2, B          /*    C*B          */
add R2, A          /*    A + CB       */
sub R2, D          /*    AB - (A + C*B) */
```

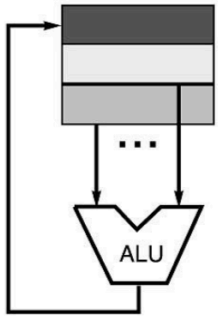
different with the register-register store

  - memory in front of the operator
  - Pros
    - **Some data** can be accessed without loading first
    - Instruction format easy to encode – Good code density
  - Cons
    - Operands are not equivalent (register != memory)
    - Variable number of clocks per instruction
    - May limit number of registers
- Load-Store (Register-Register)

Load-Store Architectures

- Instruction set:

  add R1, R2, R3 sub R1, R2, R3 mul R1, R2, R3
  load R1, &A store R1, &A move R1, R2



R3 = R1 +,-,*,/ R2

- Example: A*B - (A+C*B)

```
load R1, &A
load R2, &B
load R3, &C
mul R7, R3, R2          /*   C*B          */
add R8, R7, R1          /*   A + C*B      */
mul R9, R1, R2          /*   A*B          */
sub R10, R9, R8         /*   A*B - (A+C*B) */
```

  - store the results to the available memory of register
  - Pros
    - Simple, fixed length instruction encodings
    - Instructions take similar number of cycles
    - Relatively easy to pipeline and make superscalar
  - Cons
    - **Higher instruction count**
    - Not all instructions need three operands
    - Dependent on good compiler

Instruction types

- Three Basic Types of Instructions
  - Arithmetic and Logic : AND, ADD
  - Data Transfer : MOVE, LOAD, STORE
  - Program Control : BRANCH, JUMP, CALL

Addressing mode and Immediate Mode

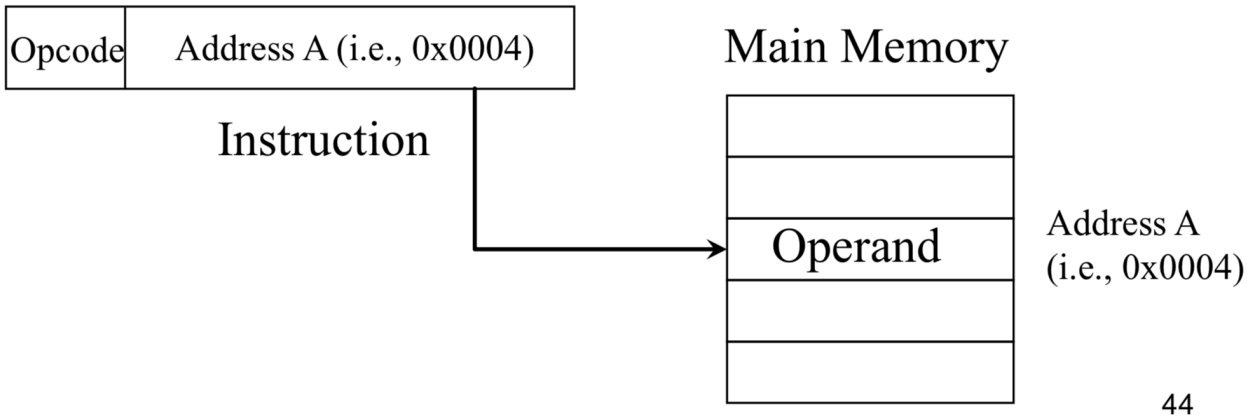
- Addressing mode : Define how machine language instructions identify the operand(s) of each instruction. (has many types as below)

$$Opcode + Mode + Address\ or\ operand \tag{12}$$

- Immediate Mode

- Operand value is encoded **directly** in the instruction
- Number of operands depends on operation (arithmetic or logical)
- No need to read from Memory=>fast but **limited in computation**
- e.g. add 5 (add 5 to the content of accumulato)

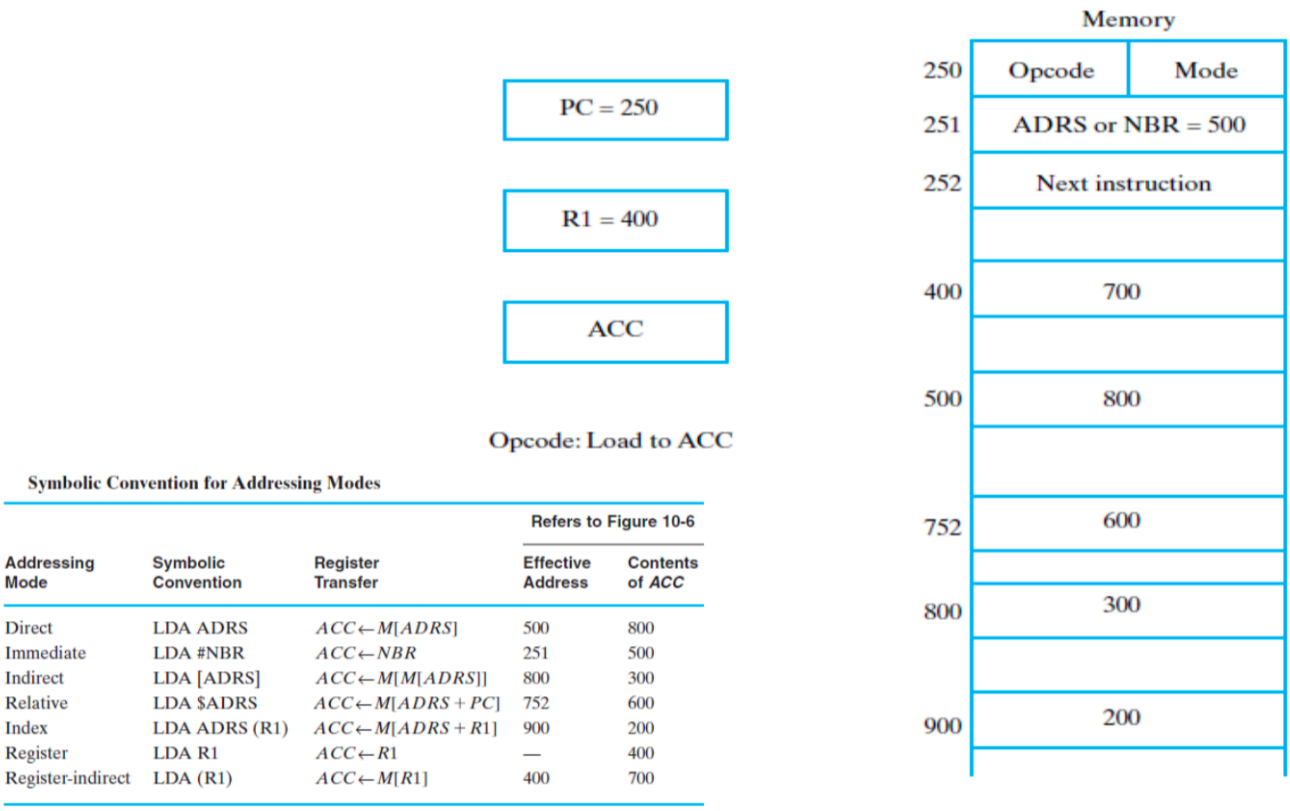
- Direct Addressing Mode
  - To access the content at a particular memory address in main memory
  - Memory address field value is encoded directly in the instruction
  - e.g. add 0x004
  - limited by size of main memory



- Register and register-indirect mode
  - Register mode : the address field **specifies a processor register** -- Add R1, R2
  - Register-indirect mode: the instruction specifies a **register** in the processor whose **content gives the address of the operand in the memory**. (register stores the address) -- Add R1,(R2)
- Addressing mode
  - Indirect address mode : the address field of the instruction gives the address at which the effective address is sotred in memory (memory stores the true address)
  - Relative addressing mode : Effective address = base + offset(often in PC register) => move from base => similar to an array
- Indexed addressing Mode: The content of an indexed register is added to the address part of the instruction to obtain the effective address => move from base

  Effective address = base + offset (**stored in register**)

Numerical Example



Symbolic Convention for Addressing Modes

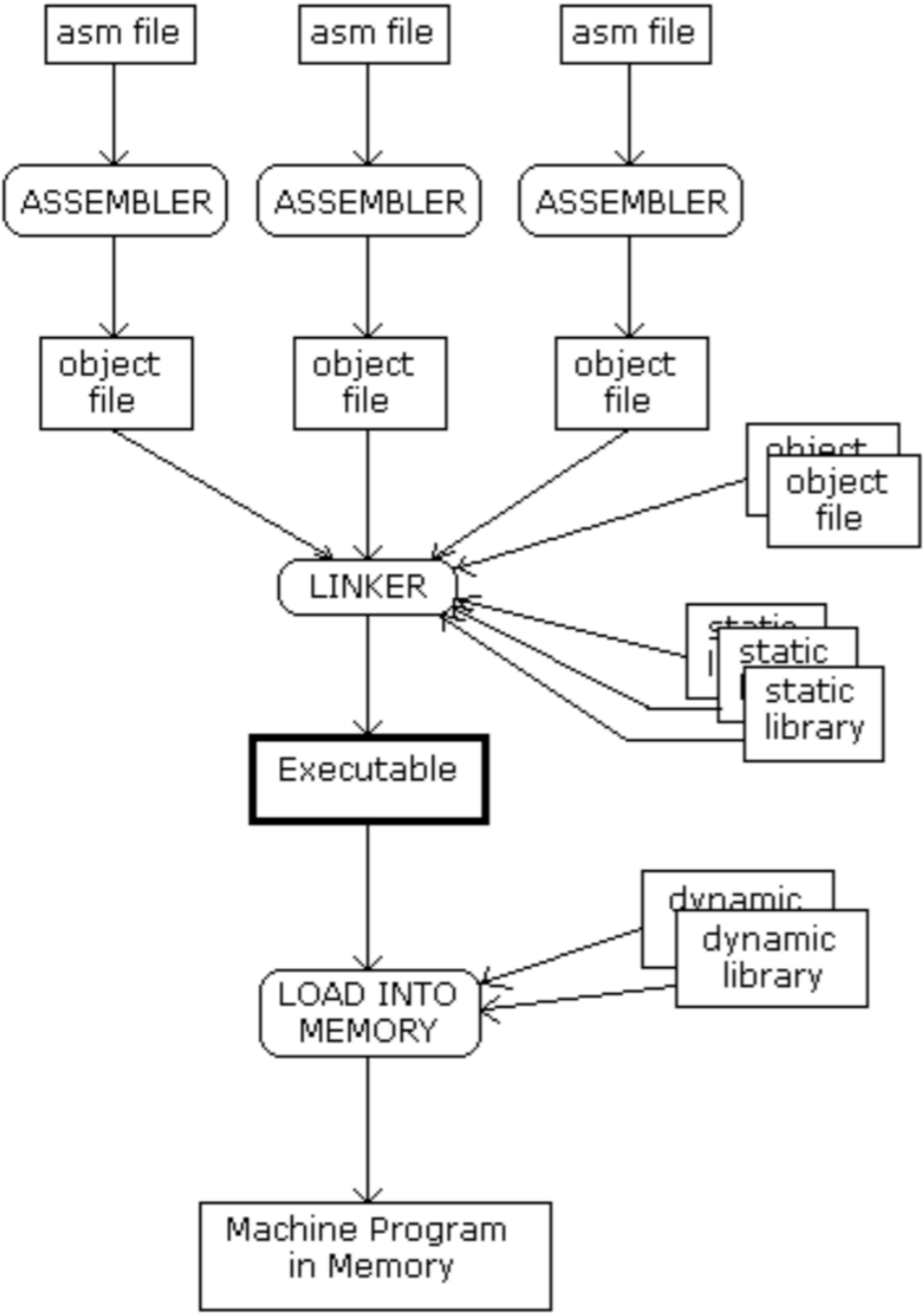
Addressing Mode	Symbolic Convention	Register Transfer	Refers to Figure 10-6: Effective Address	Refers to Figure 10-6: Contents of *ACC*
Direct	LDA ADRS	$ACC \leftarrow M[ADRS]$	500	800
Immediate	LDA #NBR	$ACC \leftarrow NBR$	251	500
Indirect	LDA [ADRS]	$ACC \leftarrow M[M[ADRS]]$	800	300
Relative	LDA $ADRS	$ACC \leftarrow M[ADRS + PC]$	752	600
Index	LDA ADRS (R1)	$ACC \leftarrow M[ADRS + R1]$	900	200
Register	LDA R1	$ACC \leftarrow R1$	—	400
Register-indirect	LDA (R1)	$ACC \leftarrow M[R1]$	400	700

Lec 07 Assembly Language (do not test)

Tutorial of Assembly language

- Windows

```
  global main
  extern puts
  section .text
main:
  sub rsp 20h     ;20h means 32 bits => Reserve the shadow space
  move rcx,message  ;rcx is general perpose register
  call puts     ;use the function puts(message) add
  add rsp, 20h        ; Remove shadow space
  ret         ; return
message:
  db 'Hello, World!', 0 ; C strings need a zero byte at the end, db is double
; run it: nasm -f win64 helloWorld.asm -o helloWorld.obj
;     gcc -m64 helloWorld.obj -o helloWorld.txt
```

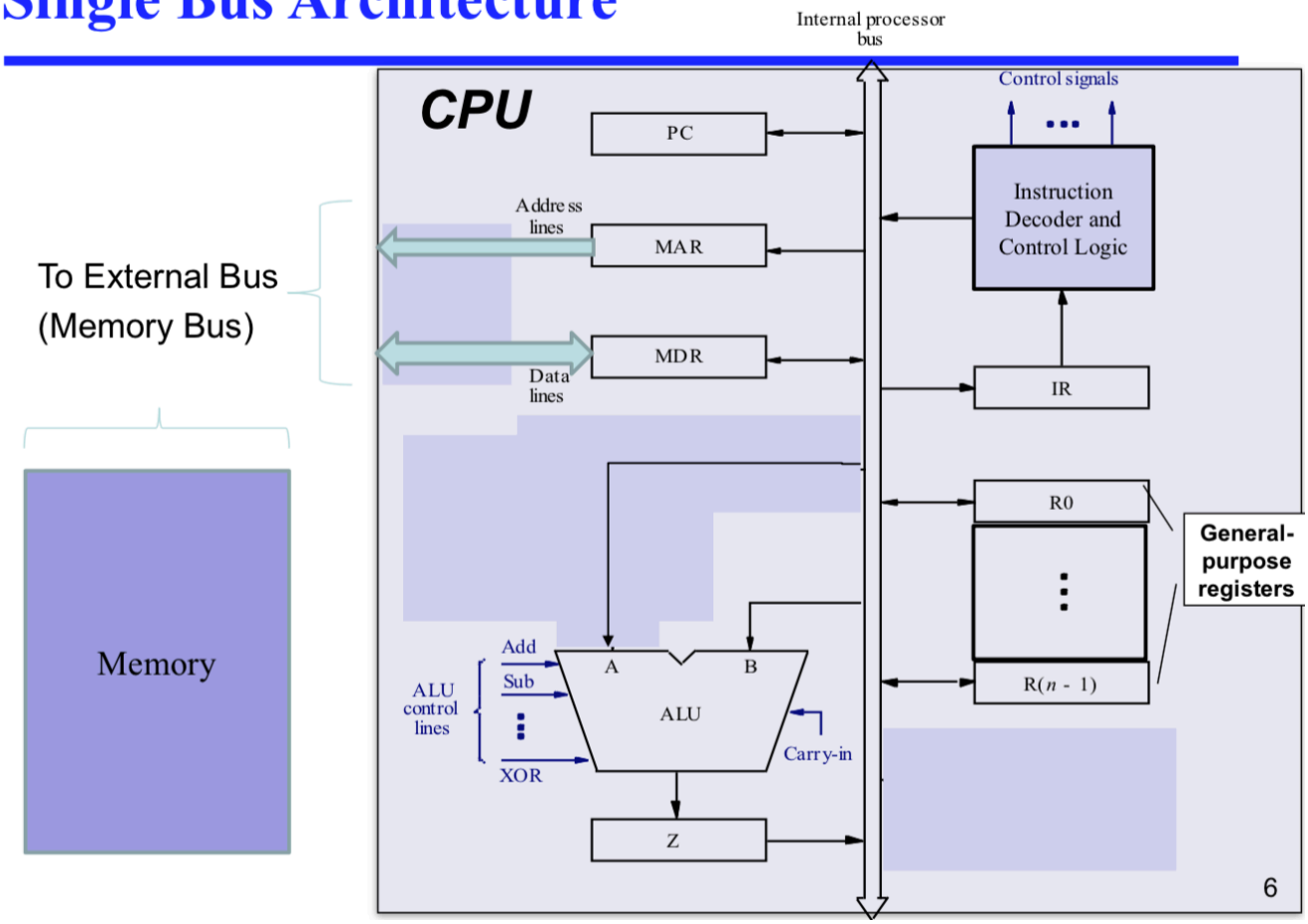

asm file
asm file
asm file
ASSEMBLER
ASSEMBLER
ASSEMBLER
object file
object file
object file
object file
LINKER
static library
Executable
dynamic library
LOAD INTO MEMORY
Machine Program in Memory

Lec 08 Central Processing Unit

CPU organization

- Aim of processor
  - transfer an instruction from memory through the bus
  - interpret and decode the instruction;
  - carry out(do) the instruction;
  - identify the next instruction ready to be fetched and executed
- Three components of the processor
  - ALU (*Arithmetic and Logical Unit*)
    - Made up of circuits that perform the **arithmetic and logical execution** within the processor.
    - It has no internal storage.
  - CU (*Control Unit*)
    - It contains circuits that direct and coordinate proper sequence, interpret each instruction and apply the **proper signals** to the ALU and registers.
  - Registers
    - Registers are high speed temporary data storage area within the processor to support execution activities.
    - Both instructions or data can be stored in registers for processing by the ALU.
    - All processors have a certain number of registers, the exact number varies between different CPUs.

Single Bus Architecture



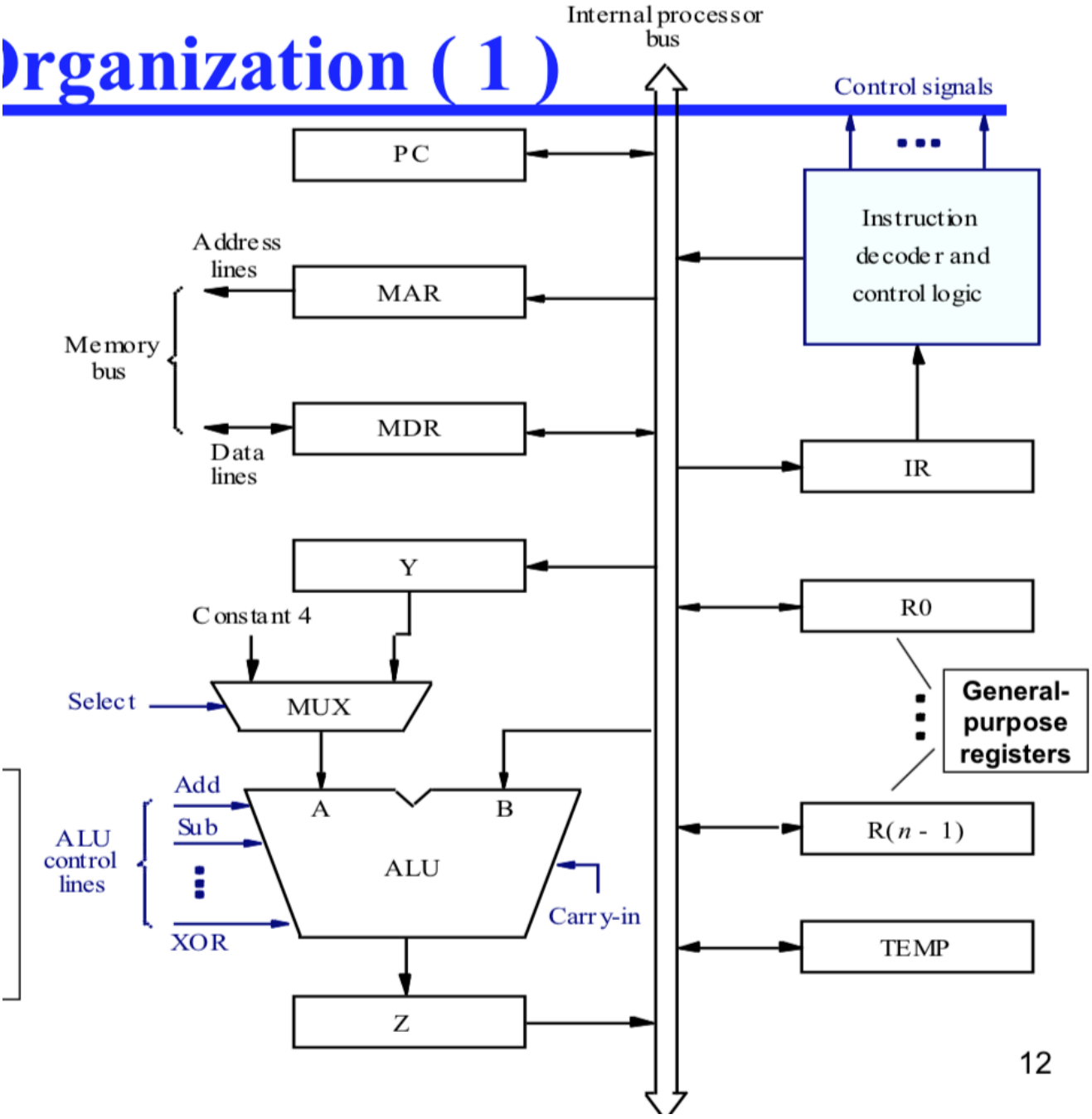
- Fetch and Execute :
  - Fetch: deliver the instruction stored at main memory to the CPU
  - Execute: execute the fetched instruction in the CPU to completion
  - To perform a task, a program consisting of a list of instructions is stored in the memory. Data also stored in memory as operands
  - To execute, fetch one instruction at a time from the memory by using the address in the PC (stores the next instruction to be fetched), and the instruction will be sent to the IR

```
IR<-[[PC]] ; PC is address of program counter
```

  - After fetching, the PC is updated (increase by 32 bits i.e. 4 bytes a time for a 32 bits system)

```
PC<-[PC]+4
```

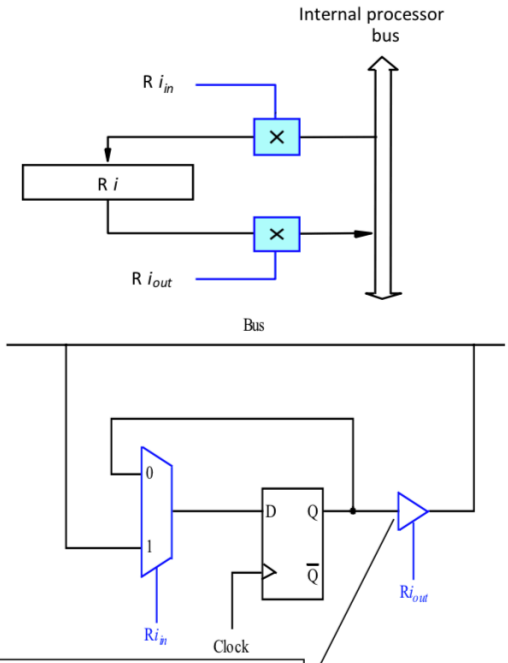
  - Examining the instruction in IR to determine which operation is to be performed (decoding)
  - Perform the specified operation by the processor
    - fetch operands from memory or registers
    - perform an arithmatic or logic operation
    - store the result in the destination
  - New instruction until the end of the program
- Single bus Organization



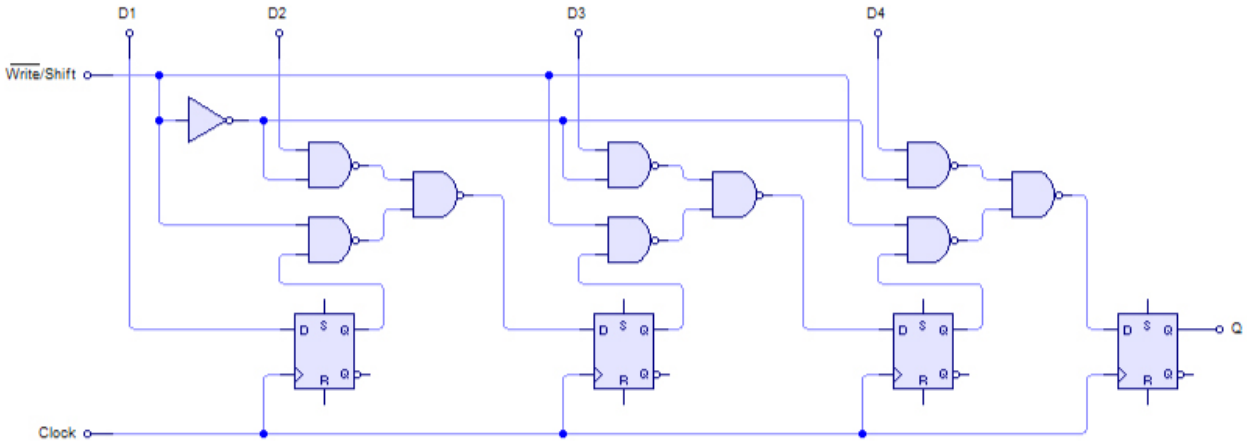
- Instruction decoder and control logic : Implementing the actions specified by the instruction loaded in IR by
  - issuing signals that control the operation of all the unit inside the processor(on one side)
  - interacting with the memory bus via *control* lines(on the other side)
- MUX selects either the output of register Y or a constant value 4(to increment the PC) to be provided as input A of the ALU
- **datapath** : Registers, ALU and the interconnecting bus are collectively referred to as the datapath

- Functions of the CPU(most of operations needed to execute an instruction can be carried out by performing a **sequence**: get/store data from register/memory)
  - transfer a word of data from one register to another or to the ALU

- **To transfer the contents of R1 to R4, the following actions are needed.**
  - Enable the output of R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the CPU bus.
  - Enable the input of R4 by setting $R4_{in}$ to 1. This loads data from the CPU bus into R4.

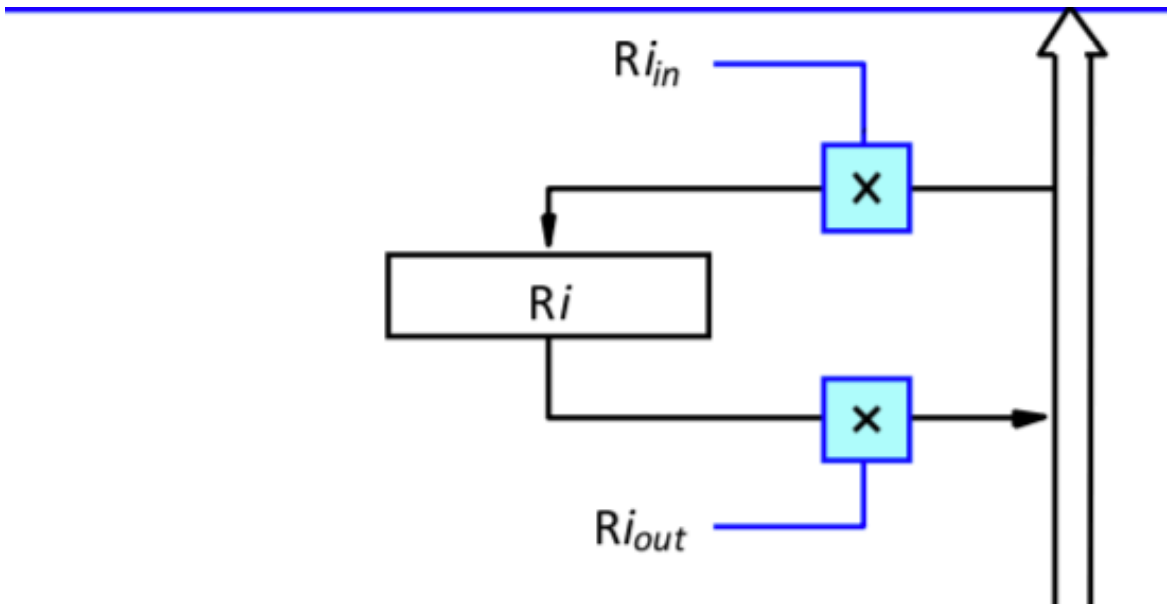


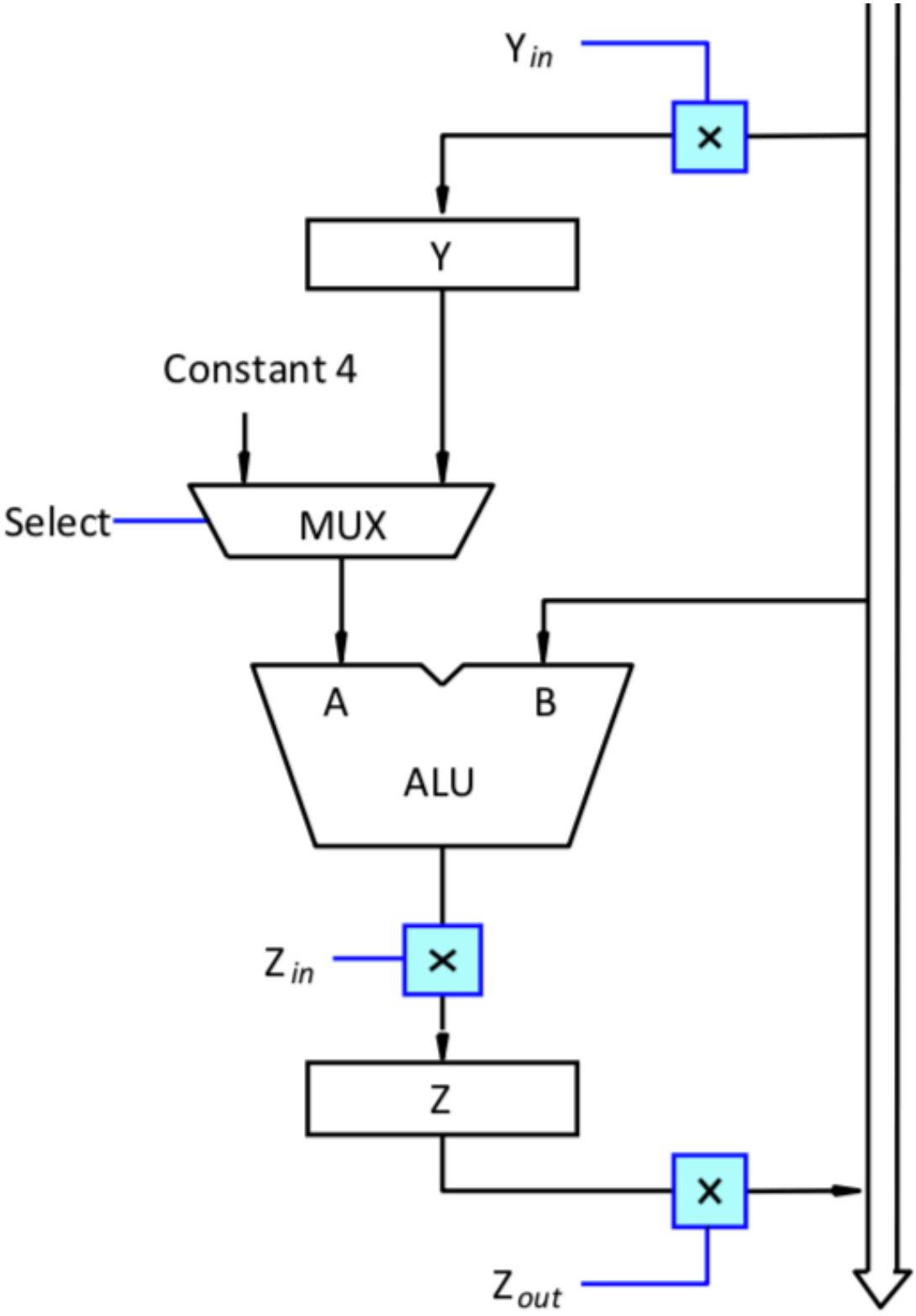
A tri-state gate:
when Ri_{out}=0, disconnected;
when Ri_{out}=1, output=input



- If R_{in} is 1, read from outside, else input the old data
- R_{out} is connected with Q by a AND gate(may be)

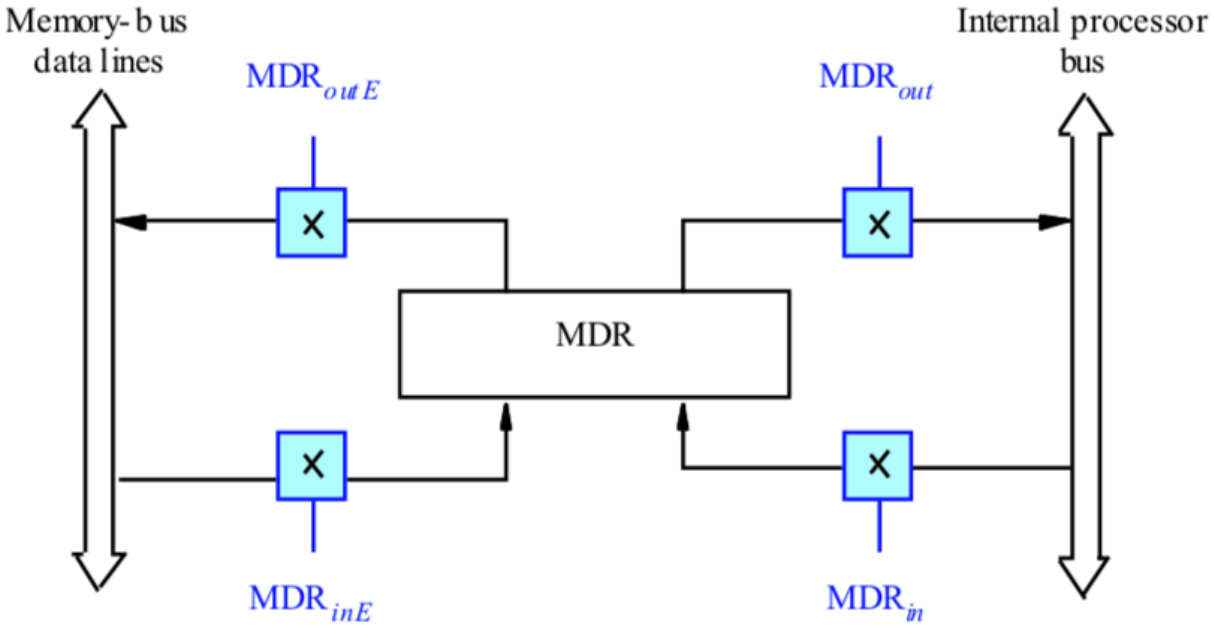
- perform an arithmetic or logic operation and store the result in a register



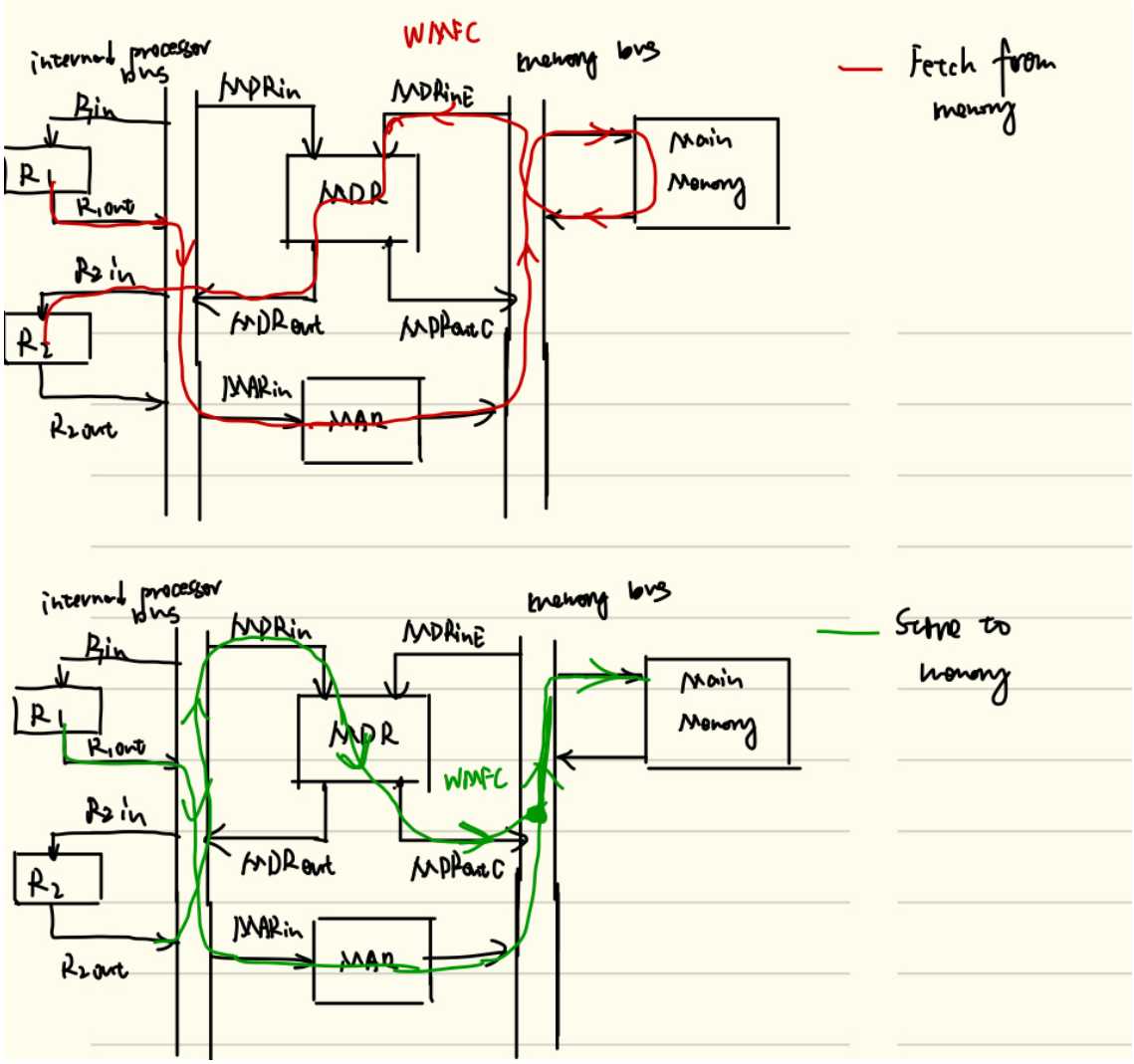


- $R1_{out}, Y_{in}$
- $R2_{out}, Select\ Y, Add, Z_{in}$ (Value of R2 automatically input to B) => why not lock B? => don't need because ALU doesn't receive any instructions
- $Z_{out}, R3_{in}$

- Fetch(or store) data in(or to) memory(address stored in R1) and load into(or write from) the register(R2). Suppose MAR_{out} is enabled all the time



- MDR can receive data from memory and CPU internal bus both (memory side has subscript E)
- Memory signal
  - During memory Read and Write operations, the timing must be coordinated with the (different) response of the memory.
  - A **MFC (Memory-Function-Completed)** signal is set by the memory when the contents of the specified location have been read and are available on the data lines of the memory bus. (memory tells MDR it is end)
  - The **WMFC (Wait for MFC)** signal causes the processor waits for the arrival of the MFC signal. (Memory tells MDR to wait)
- Fetching from memory step
  - $R1_{out}, MAR_{in}, Read\ : MAR <- [R1]$
  - $MDR_{inE}, WMFC$
  - $MDR_{out}, R2_{in}$
- Storing in memory step
  - $R1_{out}, MAR_{in}$
  - $R2_{out}, MDR_{in}, write$
  - $MDR_{outE}, WMFC$



- A complete instruction : to add the content of a memory location pointed to by R3 to R1

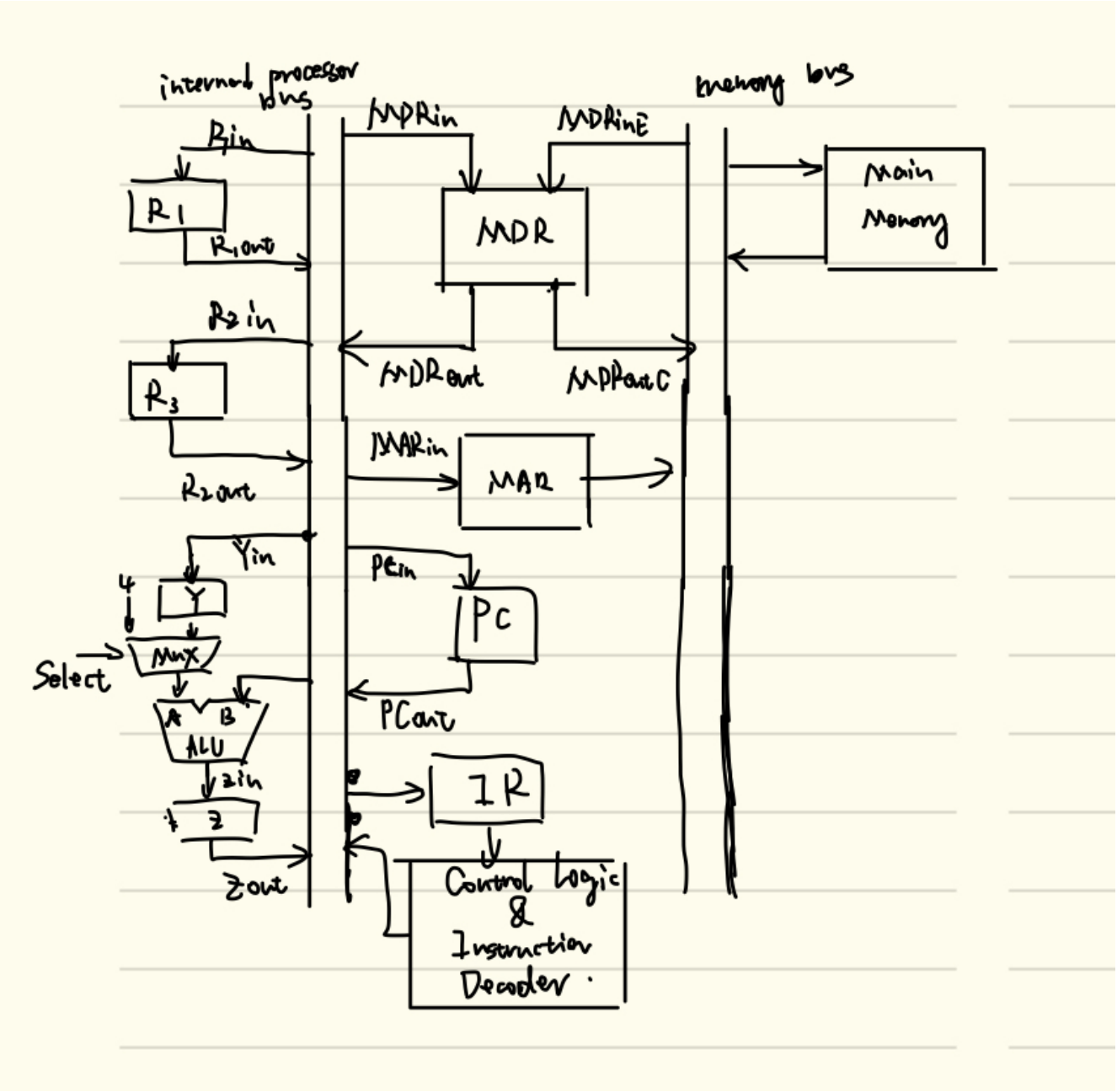
1. PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2. Z_{out}, PC_{in}, Y_{in}, MDR_{inE}, WMFC
3. MDR_{out}, IR_{in}
4. $R3_{out}$, MAR_{in}, Read
5. $R1_{out}$, Y_{in}, MDR_{inE}, WMFC
6. MDR_{out}, SelectY, Add, Z_{in}
7. Z_{out}, $R1_{in}$, End

Fetch phase

The instruction decoder interprets the contents of IR.

Execution phase

Selec

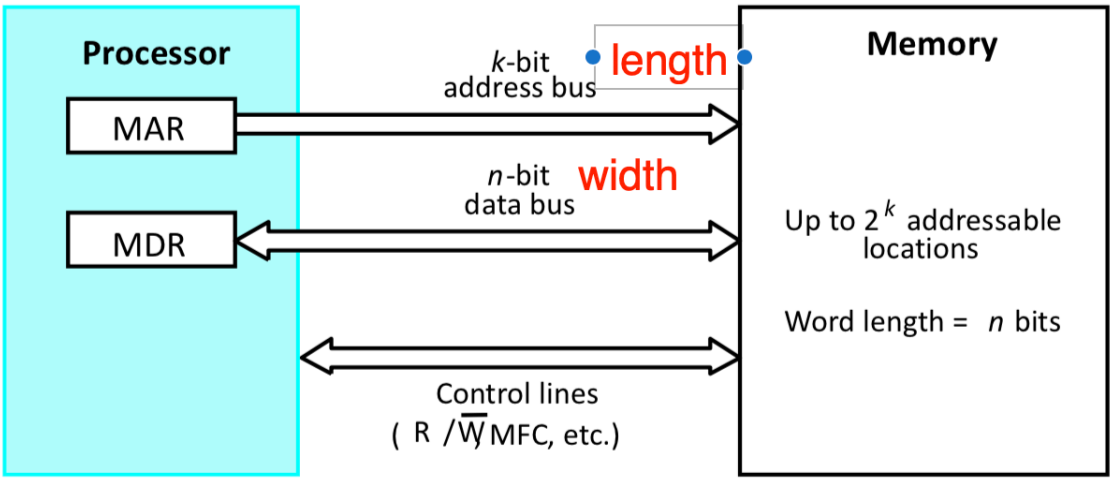


- First 3 steps is to load the **instruction**(Fetch phase, load addition instruction), in this example, the loaded instruction is ADD, it is decoded by control unit connected to the IR
  - Output the address stored in PC to MAR and B, add the PC by 4
  - Output the answer to PC and Y(because it may used to increase the PC again). Let MDR wait for memory
  - Output the data received by MDR from memory to IR
- Remaining 4 steps do the data fetching and do the calculation(Execution phase)
  - output the address stored in R3 to MAR
  - output the data stored in R1 to Y. Let MDR wait for the data pointed by R3
  - output the data received by MDR (to B), select Y and do the calculation
  - store the answer into R1

Lec 09 Memory

Main memory

- Main memory : where program and data are stored during execution
  - Consists of a number of cells, each of which can store a piece of information (data, instruction, character or number)
  - The size of the cell can be single byte or several successibe bytes(world)
    - Byte-addressable computer
    - Word-addressable computer
- Address : Reference number of each cell, refered by which program can refer to it
  - k-bits address => 2^k cells directly addressable
  - maximum size of address references available in main memory => **address space**



Connection of the memory to the processor.

  - MAR : k-bit address bus to memory
  - MDR : n-bit data bus to memory
- **Memory Access Time** : The time that elapses between the initiation and the completion of a memory access operation => How fast the memory responds to a read/write request
- **Memory Cycle Time** : The minimum time delay required between the initiation of 2 successibe memory operations (Usually slightly longer than the access time)

Types of Memory Unit

- Random-Access Memory (RAM)
  - Any location can be accessed for a Reads or Write operation in some fixed amount of time that is independent of the memory location
  - Static RAM (SRAM)
    - Memories that consist of circuits capable of retaining their state as long as power is applied. (Never changed until end of the power)
    - SRAMs are fast (a few nanoseconds access time) but their cost is high.

    - Dynamic RAM (DRAM)
        - These memory units are capable of storing information for only tens of milliseconds, thus require periodical refresh to maintain the contents.
- Read-Only Memory (ROM)
    - Nonvolatile memory
    - Data are written into a ROM when it is manufactured. Normal operation involves only reading of stored data.
    - ROM are useful as control store component in a micro- programmed CPU.
    - ROM is also commonly used for storing the bootstrap loader, a program whose function is to load the boot program from the disk into the memory when the power is turned on.
    - Types

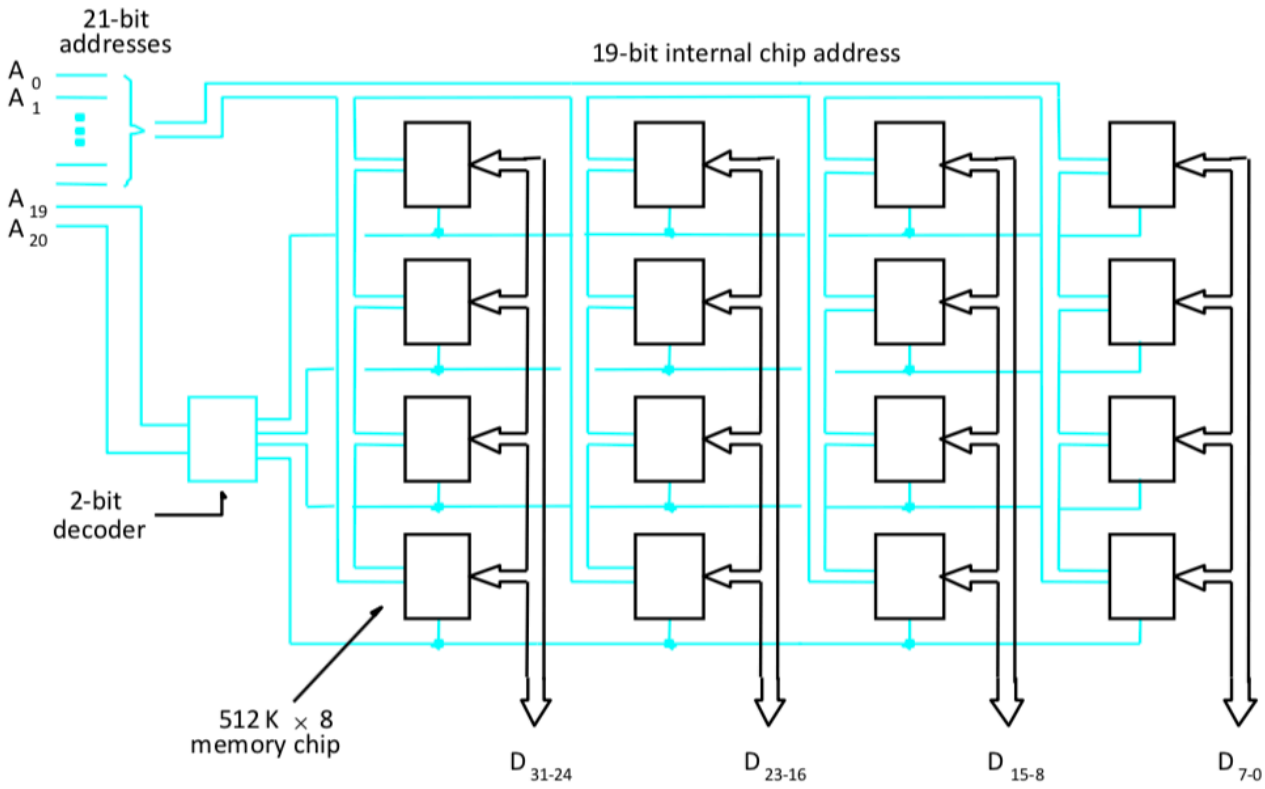
Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory
ROM	Read-only	Not possible	No	No	Large volume appliances
PROM	Read-only	Not possible	No	No	Small volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

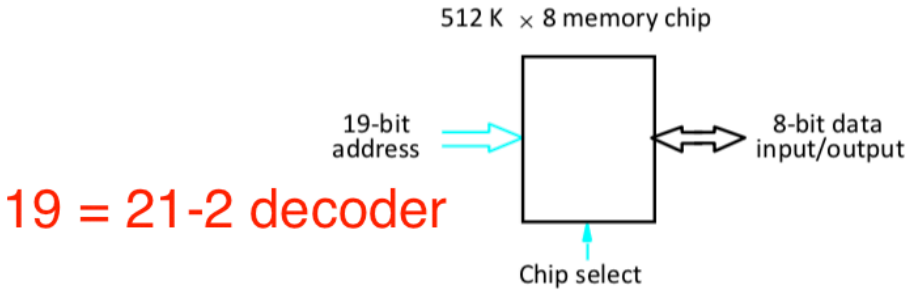
A comparison of various memory types.

        - PROM (programmable ROM)
            - all load data => not reversible
            - faster and less expensive approach
        - EPROM (Erasable, reProgrammable ROM)
            - data can be erased by UV(ultraviolet 红外线) light
        - EEPROM (Electrically Erasable Programmable)
            - Stored data can be electrically and selectively => different voltages are needed for erasing, writing and reading the stored the data
        - Flash memory : Similar to EEPROM tech
            - it is possible to read the contents of a single cell, but it is only possible to write an entire block of cells
            - greater density, higher capacity, lower cost per bit, low power consumption
            - typical applications: hand-held computers, digital cameras, MP3 music players
            - large memory modules implementation: flash cards and flash drives

Memory Systems

A memory system with 2M words(32 bits for each word) formed by 512K * 8 memory chips





$512 * 1024 * 8 * 16 = 2 * 1024 * 1024 * 32$

$512 * 1024 = 2^{19}$

- For each chip :
  - There is a control input called Chip Select (CS) used to enable the chip
  - 21 address bits are needed to select a 32-bit word
    - High-order 2 bits are decoded to determine which of the 4 CS control signals are activated (select one row from 4 rows, in binary, 2 bits can represent a 4 numbers)
    - The remaining 19 bits are used to access specific byte locations inside **each chip of the selected row**. => each output 8 bits, totally 4*8 = 32 bits => 1 word

Cache Memory

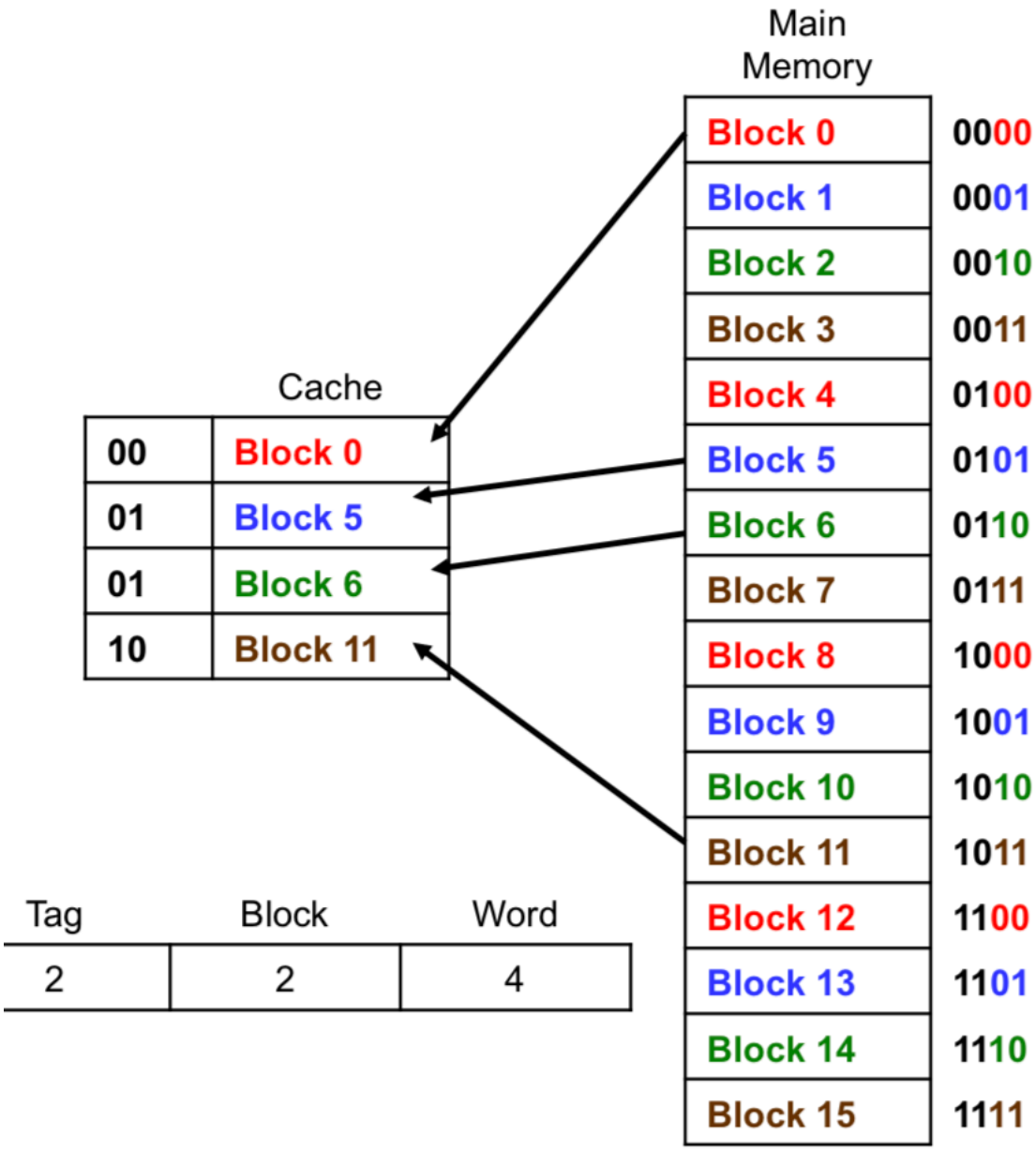
- *locality of reference* : During the execution of a typical program it is **often occurred in a few localized areas** of the program
  - temporal : recently executed instruction is likely to be used again
  - Spatial(空间的): instructions(data) in close proximity to a recently executed instruction(accessed data) are also likely to be used again
- SRAM or DRAM
  - Cache memory used to store the active segments of the program (reduce the average memory access time)
  - It is usually implemented by SRAM(no need refresh periodically, and it is fast)
- Basic operation
  - In a read operation, the word *block* asked by CPU is transferred into the cat=che from the main memory
  - *miss* : The block is not in the cache
  - *hit* : The block is in the cache

$$hit\ ratio = \frac{\#hits}{\#hits + \#miss} \tag{13}$$

- Ways to **write** access for systems with cache memory
  - *Write-through* method : the cache and the main memory locations are updated simultaneously(at the same time)
  - *write-back* method : cache location updated first, and mark the modified bit . Update main memory when the block is to be removed from the cache
- Ways to **read** from cache (mapping the address in main memory with the data block stored in the cache)
  - Suppose there is 2K(word) cache with 128 16-word-blocks ($128 * 16$) and 64K main memory addressable by a **16-bit** address, 4096 6-word-blocks
  - associative mapping
    - main memory block can (randomly) be placed into any cache position => efficient cache memory using
    - Divide the main memory address into 2 parts
      - 12 bits for block of cache ($2^{12} > 128$)
      - 4 bits for selecting 1 word in 16 words ($2^4 = 16$)
    - Cost is high because when searching, need to traverse 128 tags => thus for performance, it must be done in parallel
  - direct mapping
    - For main memory block j, assign a number j%128. Then for each block i in cache , it only stores j%128 = i
    - division of 16 bits address
      - 7 bits block field to determine block position in the cache ($2^7 = 128$)
      - 5 bits for tag, determine which super block in the 32 blocks it belongs to (there are 64/2

= 32 super blocks)

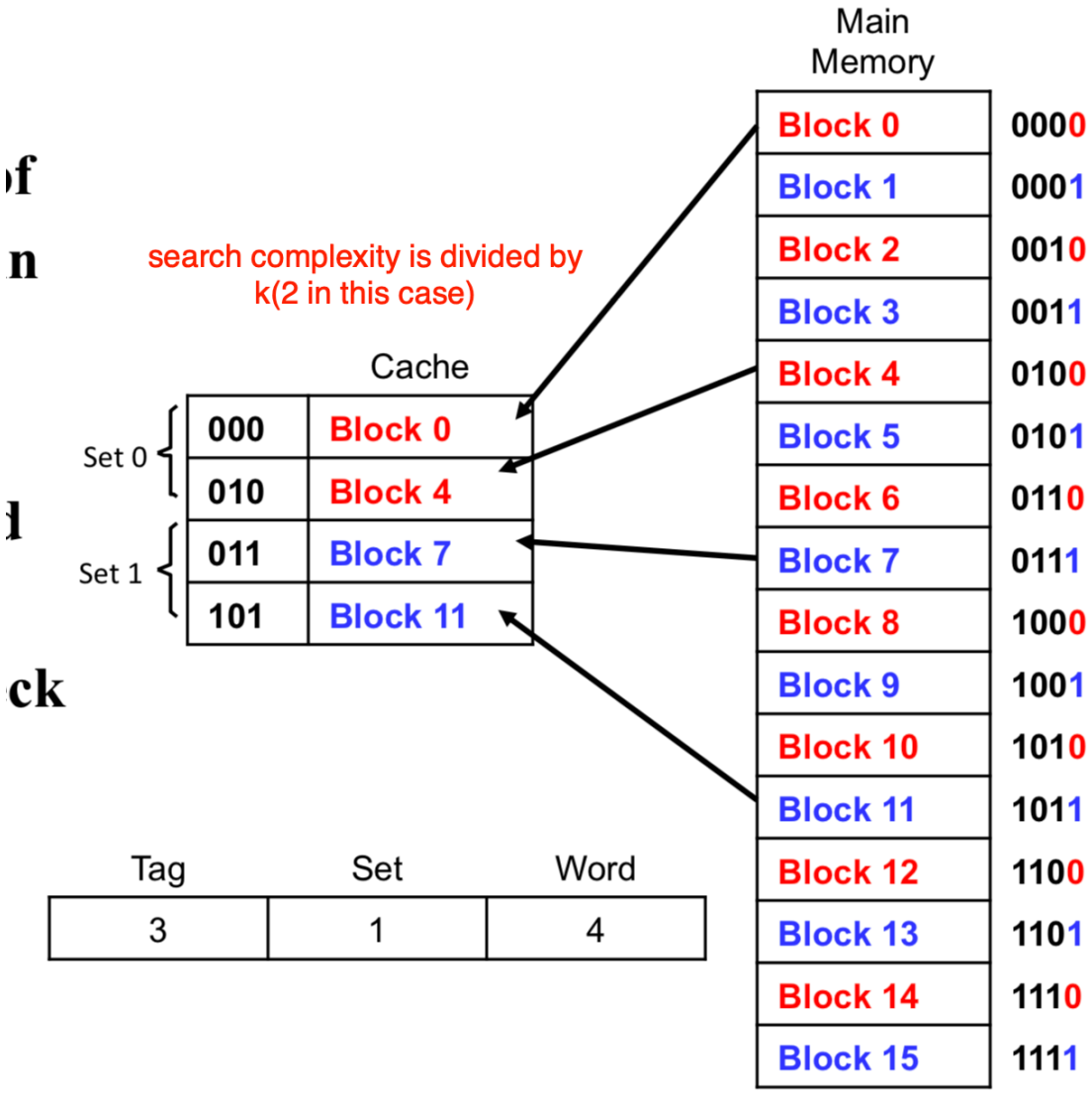
- An example of 4-block cache



- Drawbacks
  - More than one memory block is mapped onto a given cache block => contention may arise for that position even when the cache is not full
  - easy to implement but not flexible

- Set-associative mapping
  - Blocks of the cache are grouped into sets, and each time store a main memory block in any block of a specific set
  - A cache that has *k* blocks per set is referred to as a *k-way* set-associative cache
  - combine the benefit of direct method and associative method
  - Compare with associative
    - associative : each main memory block correspond with **only one** cache block
    - Set-associative: each main memory block correspond with **a set of** cache blocks

- Division of main memory address
  - 6-bit for set field to determine set in cache blocks
  - 6-bit for tag field corresponding to which block in the cache set
  - 4-bit word field
- 4-block cache example

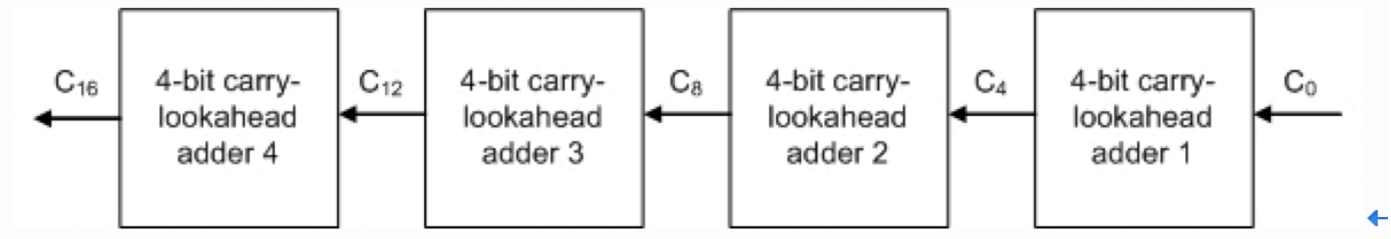


- Replacement algorithm
  - Cache contoller should decide to remove which block to create space for new bloc kwhen the cache is full
  - Least recently used (LRU) replacement algorithm
    - The block that has gone the longest time without being referenced is chosen to be overwritten.

Tutorial 9

1.

The word-width of a machine is 16 bits. There are **four** 4-bit carry-lookahead adders, which are stringed together as shown in the following.



- Disadvantage of carry-lookahead : since the formula is complete, it is hard to calculate when the digit is too large
- Disadvantage of Binary Ripple Carry Adder : for each C_i there is 2 gates delay (2 half adder)
- Solution : Mix them and take advantage of both of them

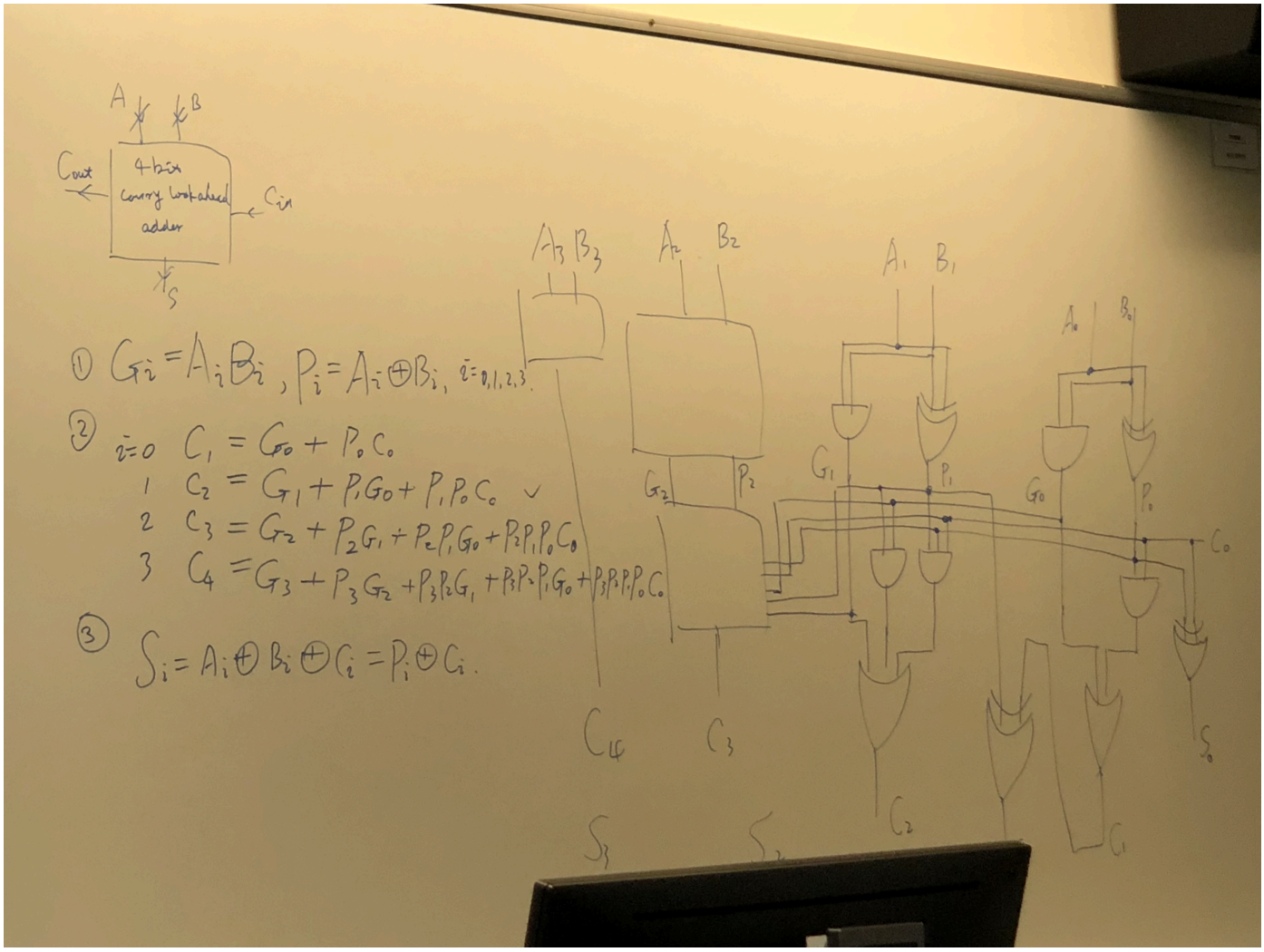
(a) Assume the gate delay to develop the carry of each bit is **2**, and the gate delay to obtain all G_i and P_i (in all the carry-lookahead adders) is **1**. Then what's the total delay from C_0 to C_{16}? What if there are **two** 8-bit carry-lookahead adders stringed together?from C_0 to C_4 :

3 gate delay = 1 for G and P + 2 gates for C

3 gate delay among k blocks => why not 3*k?

G and P => 1 gate delay => for all the block, this step can be done at the same time

Thus the total delay is reduced to **2*k+1**



2. Suppose a computer has a cache with the following parameters.

- Cache access time = 1 clock cycle
- Hit rates are 0.95 for instructions and 0.9 for data
- Cache miss penalty = 17 clock cycles (time used for cache miss and get from CPU)
- Consider a typical program with 100 instructions, 30% of the instructions access data in memory, i.e., 130 memory accesses for 100 instructions executed. What is the performance loss compared to an ideal cache with a hit rateof 100% (time with cache misses / time without cache miss)?

$$
\begin{aligned}
time\ without\ miss &= 130 * 1\ colckcycle \\
time\ with\ miss &= 130 * 1 + 100 * 0.05 * 17 + 30 * 0.1 * 17 = 266 \\
performance &= \frac{266}{130} \\
time\ without\ cache &= 12 * 130
\end{aligned}
\tag{14}
$$

Lec 10

Lec 11 Advanced Topic