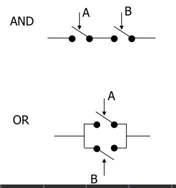
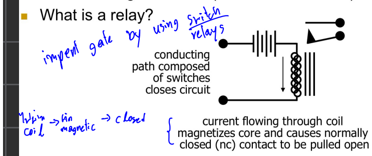
Basic / Logic gate

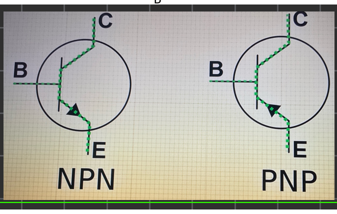
digital logic where voltage < 0.8v is a "0" and > 2.0v is a "1"

Compose switches into more complex ones (Boolean functions):



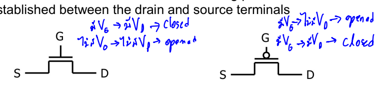
- A simple way to convert between conducting paths and switch settings is to use (electro-mechanical) relays.
- What is a relay?



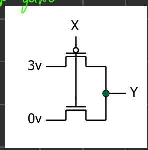


NPN : if (B∧C มีไฟ) → E มีไฟ = B+C
else → E ไม่มีไฟ
PNP : if (E มีไฟ) → B∧C มีไฟตามปริมาณ B+C=E
else → B∧C ไม่มีไฟ

- MOS transistors have three terminals: drain, gate, and source
  - they act as switches in the following way: (3 terminals)
    if the voltage on the gate terminal is (some amount) higher/lower than the source terminal then a conducting path will be established between the drain and source terminals



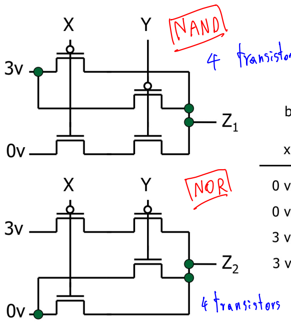
① Not gate



x	y
0V	3V
3V	0V

MOS = metal-oxide-semiconductor

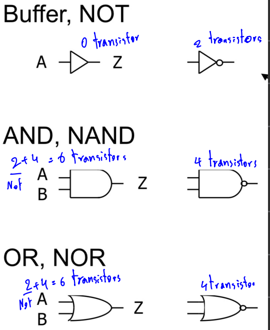
2 type transistor
n-channel & p-channel
↓ สามารถนำมาสร้าง
3 type basic gate
not, NAND, NOR
↓ สามารถนำมาสร้าง
Another gate on boolean Algebra
ex AND = NOT + AND
OR = NOT + OR
...



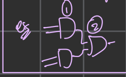
what is the relationship between x, y and z?

x	y	z1	z2
0 volts	0 volts	3	3
0 volts	3 volts	3	0
3 volts	0 volts	3	0
3 volts	3 volts	0	0
		NAND	NOR

- Buffer, NOT
- AND, NAND
- OR, NOR



จำนวน level-gate คือ maximum number of gate in 1 path



- Outputs of a combinational circuit depend only on current inputs
  - after sufficient time has elapsed
- Sequential circuits have memory
  - even after waiting for the transient activity to finish

เข้าใจ

- Combinational means "memory-less" ← just logic gate
  - a digital circuit is combinational if its output values only depend on its input values
- Sequential systems ← caused by clock memory
  - exhibit behaviors (output values) that depend not only on the current input values, but also on previous input values
- In reality, all real circuits are sequential

Math

Sum of product | product of sum

- identity
  1. X + 0 = X 1D. X • 1 = X
- null
  2. X + 1 = 1 2D. X • 0 = 0
- idempotency:
  3. X + X = X 3D. X • X = X
- involution:
  4. (X')' = X
- complementarity:
  5. X + X' = 1 5D. X • X' = 0
- commutativity:
  6. X + Y = Y + X 6D. X • Y = Y • X
- associativity:
  7. (X + Y) + Z = X + (Y + Z) 7D. (X • Y) • Z = X • (Y • Z)
- distributivity:
  8. X • (Y + Z) = (X • Y) + (X • Z) 8D. X + (Y • Z) = (X + Y) • (X + Z)
- uniting:
  9. X • Y + X • Y' = X 9D. (X + Y) • (X + Y') = X
- absorption:
  10. X + X • Y = X 10D. X • (X + Y) = X
  11. (X + Y') • Y = X • Y 11D. (X • Y') + Y = X + Y
- factoring:
  12. (X + Y) • (X' + Z) = X • Z + X' • Y 12D. X • Y + X' • Z = (X + Z) • (X' + Y)
- concensus:
  13. (X • Y) + (Y • Z) + (X' • Z) = X • Y + X' • Z 13D. (X + Y) • (Y + Z) • (X' + Z) = (X + Y) • (X' + Z)
- de Morgan's:
  14. (X + Y + ...)' = X' • Y' • ... 14D. (X • Y • ...)' = X' + Y' + ...
- generalized de Morgan's:
  15. $f'(X_1,X_2,...,X_n,0,1,+,•) = f(X_1',X_2',...,X_n',1,0,•,+)$

sense — property of sum of product & product of sum

SOP & POS

รูปแบบผลบวกของผลคูณ (Sum of Product)

- รูปแบบผลบวกของผลคูณ (Sum of Product) SOP
- การนำตัวแปรซึ่งอยู่ในรูปปกติและรูปคอมพลีเม้นท์มา AND กัน
  - เรียกเทอมที่ AND กันนี้ว่า มินเทอม (Minterm) — เช็คเฉพาะ case ①②③ แค่ 1 อัน → output = True
- แล้วจึงนำ Minterm แต่ละเทอมมา OR กันอีกที
- ทั้งฟังก์ชันเรียกว่า Canonical Sum
- เช่น

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

- อาจเขียนให้อยู่ในรูปของ Minterm ได้ดังนี้

$$F(A,B,C) = m_2 + m_6 + m_3 \quad (m = \text{มินเทอม})$$

ใน case ที่ output = True → นำทุก input มา AND กัน → ทุก case มา OR กัน (input A:0 → $\bar{A}$, A:1 → A)

รูปแบบผลคูณของผลบวก (Product of Sum)

- รูปแบบผลคูณของผลบวก (Product of Sum) POS
- การนำตัวแปรซึ่งอยู่ในรูปปกติและรูปคอมพลีเม้นท์มา OR กัน
  - เรียกเทอมที่ OR กันนี้ว่า แมกเทอม (Maxterm) — เช็คเฉพาะ case ①②③ แค่ 1 อัน → Output = False
- แล้วจึงนำ Maxterm แต่ละเทอมมา AND กันอีกที
- ทั้งฟังก์ชันเรียกว่า Canonical Product
- เช่น

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

- อาจเขียนให้อยู่ในรูปของ Maxterm ได้ดังนี้

$$F(A,B,C) = M_0 \cdot M_1 \cdot M_5 \quad (M = \text{แมกเทอม})$$

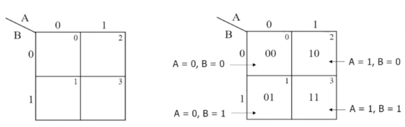
ใน case ที่ output = False → นำทุก input มา OR กัน → นำทุก case มา AND กัน (input A:0 → A, A:1 → $\bar{A}$)

- Minterm to maxterm conversion
  - use maxterms whose indices do not appear in minterm expansion
  - e.g., F(A,B,C) = Σm(1,3,5,6,7) = ΠM(0,2,4)
- Maxterm to minterm conversion
  - use minterms whose indices do not appear in maxterm expansion
  - e.g., F(A,B,C) = ΠM(0,2,4) = Σm(1,3,5,6,7)

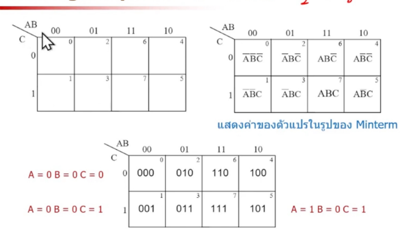
K-map n input

Karnaugh Map 2 ตัวแปร (2^n)

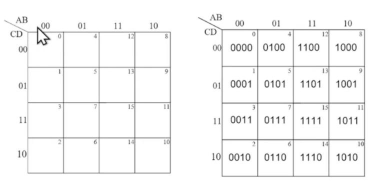
- Karnaugh Map ชนิด 2 ตัวแปร ประกอบด้วยช่องสี่เหลี่ยม จำนวน 2^2 = 4 ช่อง
- รหัสประจำแถว และรหัสประจำหลัก จะเรียงกันในรูปแบบรหัสเกรย์



Karnaugh Map 3 ตัวแปร (3 input, 2^3 = 8)



Karnaugh Map 4 ตัวแปร

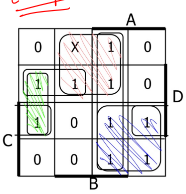


K-map / Karnaugh Map

Definition of terms for two-level simplification

- Implicant *every group ที่วงได้* // ◻ ใน ACB → A∧B = implicant
  - single element of ON-set or DC-set or any group of these elements that can be combined to form a subcube
- Prime implicant *bigest implicant*
  - implicant that can't be combined with another to form a larger subcube
- Essential prime implicant *cover all 1 ∧ by using prime implicant*
  - prime implicant is essential if it alone covers an element of ON-set
  - will participate in ALL possible covers of the ON-set
  - DC-set used to form prime implicants but not to make implicant essential
- Objective:
  - grow implicant into prime implicants (minimize literals per term)
  - cover the ON-set with as few prime implicants as possible (minimize number of product terms)

example



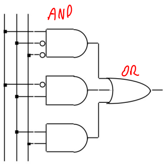
bigest cube *ตัวที่ใหญ่สุด*

6 prime implicants:
A'B'D, BC', AC, A'C'D, AB, B'CD

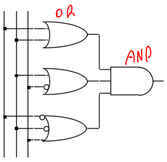
essential *จำเป็นต้องมีถึง cover 1* *prime + cover all element*

minimum cover: AC + BC' + A'B'D

- Sum-of-products
  - AND gates to form product terms (minterms)
  - OR gate to form sum
  - *ถ้า term AND = True → output = True (OR gate)*



- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product
  - *ถ้า term OR = False → output = False (AND gate)*

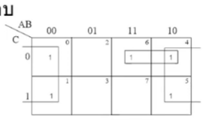


III - Working with

การลดรูปโดยใช้ Karnaugh Map

- การลดรูป โดยใช้ Karnaugh Map มีวิธีการดังนี้

1. ใส่ Minterm (1) หรือ Maxterm (0) ลงใน Karnaugh Map
2. จับกลุ่มตัวที่อยู่ติดกัน โดยมีหลักเกณฑ์ที่ว่า จับกลุ่มได้ครั้งละ 2^n ตัว คือ 1 ตัว หรือ 2 ตัว หรือ 4 ตัว หรือ 8 ตัว หรือ 16 ตัว ... (n คือเลขจำนวนเต็ม บวก)
3. การจับกลุ่มจะต้องจับกลุ่มให้ได้มากที่สุด เท่าที่ทำได้
4. ตัวที่ถูกจับกลุ่มไปแล้ว ก็สามารถนำไปจับกลุ่มกับตัวอื่นอีกก็ได้
5. Karnaugh Map มีคุณสมบัติม้วนโดยรอบ
   - ตัวริมซ้าย จับกลุ่ม กับตัวริมขวา ได้
   - ตัวริมบน จับกลุ่ม กับตัวริมล่าง ได้



Digital Circuit Design — Karnaugh Map I

// K-map : ทำเพื่อให้มองgroupที่ลูปได้ง่าย

true = 1, False = 0 ใช้ได้ทั้ง SOP = implicant 1, POS = implicant 0

↓

จับกลุ่ม Condition
① จับแค่ 2^n เป็นแบบ cube
② implicant สามารถเป็น candidate ซ้ำกันได้
③ map ทะลุกันได้

↓

1 implicant คือได้ 1 term โดยเลือกแค่ input ที่เหมือนกัน ในทุกตัวใน implicant นั้น

prove

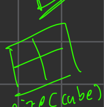
$(A+\bar{B}+C+\bar{D})(A+\bar{B}+\bar{C}+\bar{D})(A+\bar{B}+\bar{C}+\bar{D})(A+\bar{B}+C+\bar{D})$

$[(A+\bar{B}+\bar{D})+(C\bar{C})]^F \cdot [(A+\bar{B}+\bar{D})+(C\bar{C})]^F$

$(A+\bar{B}+\bar{D}) \cdot (A+\bar{B}+\bar{D})$

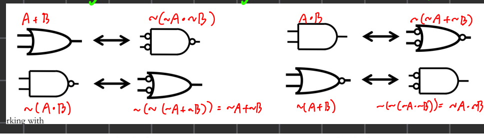
$(A+\bar{D})+(B\bar{B})^F$

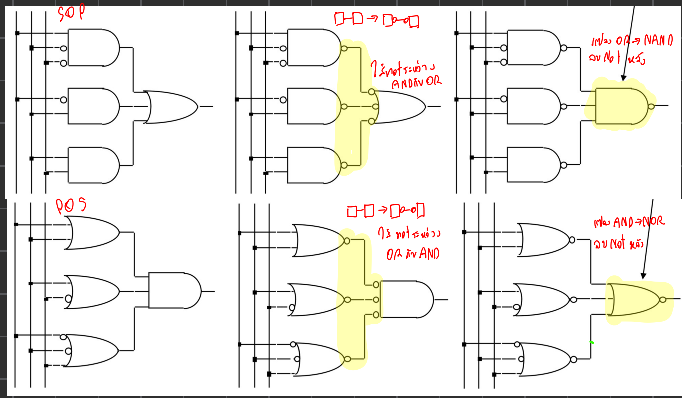
$A+\bar{D}$



size (cube) = 4

Convert using OR/AND gate → NAND/NOR gate



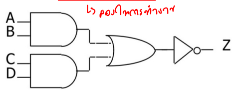


- AOI function: three stages of logic — AND, OR, Invert
  - multiple gates "packaged" as a single circuit block

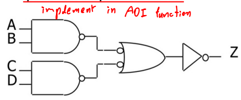
transistor gate that using in real circuit

→ AND OR invert gate

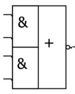
logical concept *→ รูปเก็บความคิดทำงาน*



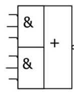
possible implementation *implement in AOI function*



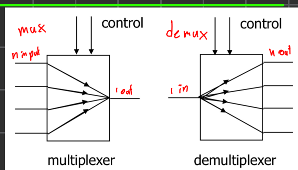
2x2 AOI gate symbol



3x2 AOI gate symbol

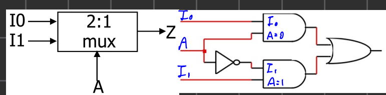


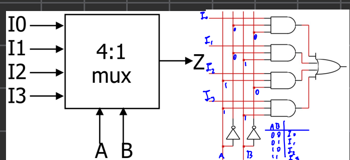
- Route one of many inputs to a single output --- multiplexer
- Route a single input to one of many outputs --- demultiplexer

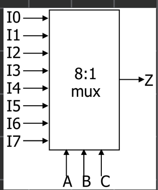


- Multiplexers/selectors: general concept

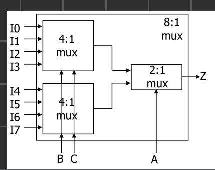
→ input + Selector = output : Selector เลือก index ของ input เพื่อเป็น output
→ ส่วนของ Select output ของแต่ละ process ว่า output ของprocessไหนเป็นค่าของ
→ ∀ (data bit / input) = data bit (output)
→ input = 2^n ตัว → Selector = n ตัว



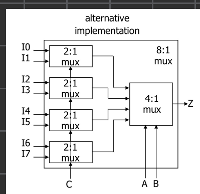




=



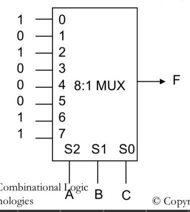
=



8:1 = 4:1 + 4:1

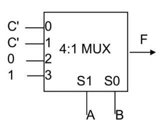
8:1 = (2:1 + 2:1 + 2:1 + 2:1) + (4:1)

- Example: *// 8 input +3 Selector → 2 input + 1 Selector*
  - F(A,B,C) = m0 + m2 + m6 + m7 *3 Variable*
    = A'B'C' + A'BC' + ABC' + ABC
    = A'B'(C') + A'B(C') + AB'(0) + AB(1)



A	B	C	F	
0	0	0	1	C'
0	0	1	0	
0	1	0	1	C'
0	1	1	0	
1	0	0	0	0
1	0	1	0	
1	1	0	1	1
1	1	1	1	

output (F) กับ C



IV - Combinational Logic Technologies

Decoders/demultiplexers: general concept

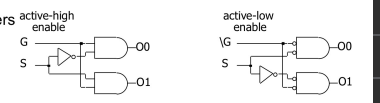
→ Selector selects [what output is equal to input] [else is default value]

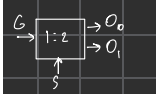
→ สามารถ give input แค่ process ว่า process ไหน ถ้า value = input แต่ value = default

→ ∀ (data bit / output) = data bit (input)

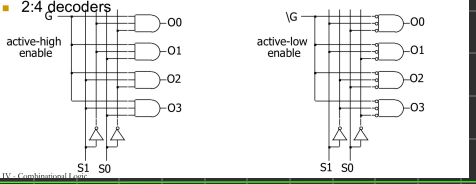
→ output = 2^n ตัว → Selector = n ตัว

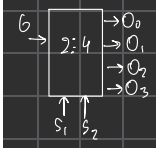
- 1:2 decoders



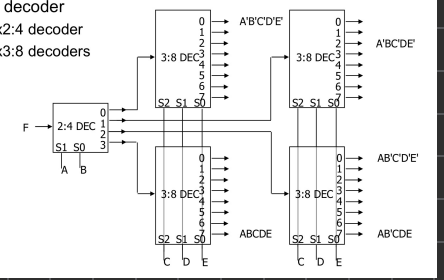


- 2:4 decoders





- 5:32 decoder
  - 1x2:4 decoder
  - 4x3:8 decoders



PROM & PAL & PLA // sum of product

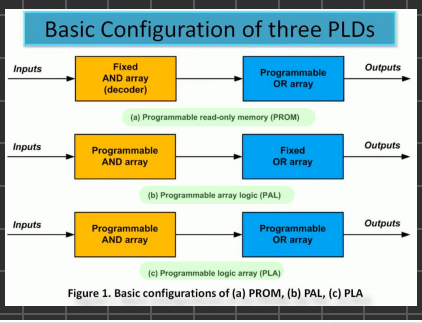


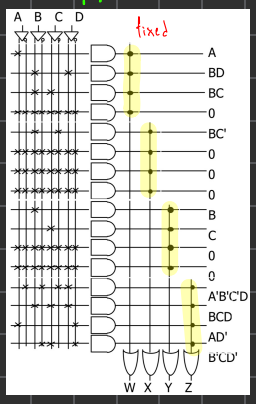
Figure 1. Basic configurations of (a) PROM, (b) PAL, (c) PLA

Feature	Programmable Logic	Fixed Logic
Flexibility	Reconfigurable for different tasks	Optimized for one specific function
Cost	More expensive for smaller volumes	Cheaper for large-scale production
Performance	Slightly slower due to reprogrammability	Faster, optimized for specific tasks
Power	Higher power consumption	Lower power consumption
Use Cases	Prototyping, dynamic systems	High-volume, performance-critical applications

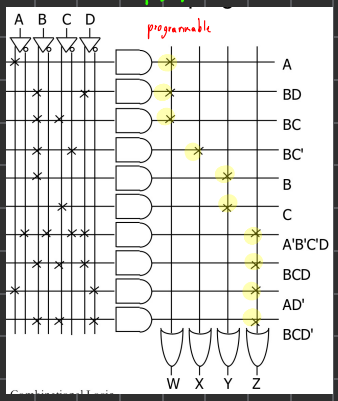
- ROM – full AND plane, general OR plane
  - cheap (high-volume component)
  - can implement any function of n inputs ← has all possible (Array)
  - medium speed
- PAL – programmable AND plane, fixed OR plane
  - intermediate cost
  - can implement functions limited by number of terms
  - high speed (only one programmable plane that is much smaller than ROM's decoder)
- PLA – programmable AND and OR planes
  - most expensive (most complex in design, need more sophisticated tools)
  - can implement any function up to a product term limit
  - slow (two programmable planes)

Feature	PAL (Programmable Array Logic)	PLA (Programmable Logic Array)
Structure	Fixed AND array, programmable OR array	Fully programmable AND and OR arrays
Flexibility	Less flexible due to fixed AND array	More flexible with programmable AND and OR arrays
Complexity	Simpler design, lower complexity	More complex, allowing for more customized logic
Programming	Only the OR array is programmable	Both AND and OR arrays are programmable
Speed	Typically faster due to simpler design	Slower due to additional complexity and flexibility
Power Consumption	Lower power consumption	Higher power consumption due to more programmable elements
Cost	Less expensive	More expensive
Logic Implementation	Can implement simpler logic functions	Can implement more complex logic functions
Typical Use	Used for simpler logic functions and faster performance	Used for more complex designs with greater logic flexibility
Reconfigurability	Limited reconfigurability due to fixed AND array	High reconfigurability with fully programmable logic arrays

PAL

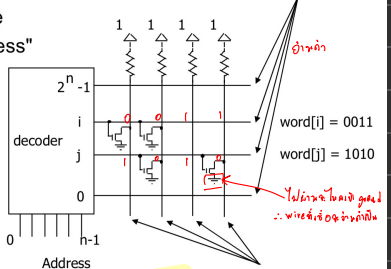


PLA



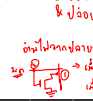
ROM

- Two dimensional array of 1s and 0s
  - entry (row) is called a "word"
  - width of row = word-size
  - index is called an "address"
  - address is input
  - selected word is output



internal organization

ข้อควรรู้: ปล่อยไฟ 1 Line ของ decode & ปล่อยไฟทุก Line ของ bit line

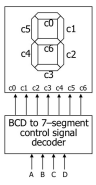


Problem: 7 segment / sensor / calendar

BCD to 7-segment display controller

problem show 0-6

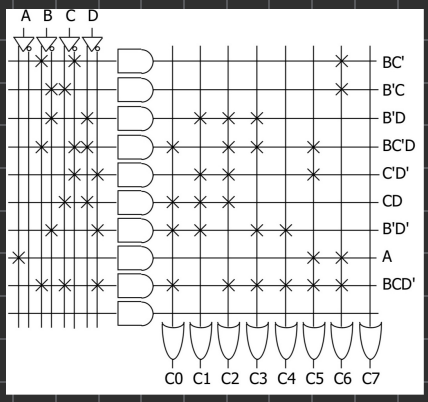
- Understanding the problem
  - input is a 4 bit bcd digit (A, B, C, D)
  - output is the control signals for the display (7 outputs C0 – C6)
- Block diagram



truth table

A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	–	–	–	–	–	–	–	–
1	1	–	–	–	–	–	–	–	–	–

PLA



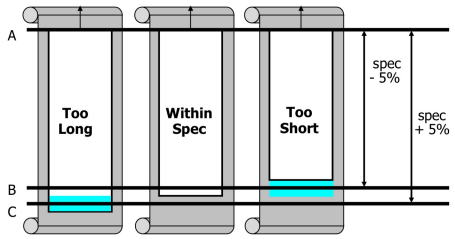
each term from k-map

C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

Position of sensors

problem size obj

- A to B distance = specification – 5%
- A to C distance = specification + 5%



Truth table

- show don't cares

A	B	C	Function
0	0	0	do nothing
0	0	1	do nothing
0	1	0	do nothing
0	1	1	do nothing
1	0	0	too short
1	0	1	don't care
1	1	0	in spec
1	1	1	too long

logic implementation now straightforward just use three 3-input AND gates

"too short" = AB'C'
(only first sensor tripped)

"in spec" = A B C'
(first two sensors tripped)

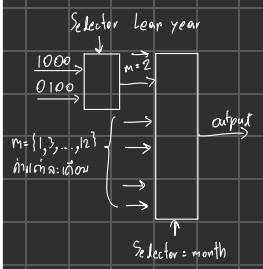
"too long" = A B C
(all three sensors tripped)

Calendar subsystem

problem

- Determine number of days in a month (to control watch display)
  - used in controlling the display of a wrist-watch LCD screen
  - inputs: month, leap year flag
  - outputs: number of days
- Use software implementation to help understand the problem

```
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
            case  1: return (31);
            case  2: if (leap_year_flag == 1)
                        then return (29)
                        else return (28);
            case  3: return (31);
            case  4: return (30);
            case  5: return (31);
            case  6: return (30);
            case  7: return (31);
            case  8: return (31);
            case  9: return (30);
            case 10: return (31);
            case 11: return (30);
            case 12: return (31);
            default: return (0);
    }
}
```



- Discrete gates
  - 28 = m8' m4' m2 m1' leap'
  - 29 = m8' m4' m2 m1' leap
  - 30 = m8' m4 m1' + m8 m1
  - 31 = m8' m1 + m8 m1'
- Can translate to S-o-P or P-o-S

month	leap	28	29	30	31
0000	–	–	–	–	–
0001	–	0	0	0	1
0010	0	1	0	0	0
0010	1	0	1	0	0
0011	–	0	0	0	1
0100	–	0	0	1	0
0101	–	0	0	0	1
0110	–	0	0	1	0
0111	–	0	0	0	1
1000	–	0	0	0	1
1001	–	0	0	1	0
1010	–	0	0	0	1
1011	–	0	0	1	0
1100	–	0	0	0	1
1101	–	–	–	–	–
111–	–	–	–	–	–

leap year
↳ %4==0 & %100 != 0 & %400 == 0

Activity: divisible-by-4 circuit

BCD coded year
YM8 YM4 YM2 YM1 – YH8 YH4 YH2 YH1 – YT8 YT4 YT2 YT1 – YO8 YO4 YO2 YO1

- BCD coded year
  - YM8 YM4 YM2 YM1 – YH8 YH4 YH2 YH1 – YT8 YT4 YT2 YT1 – YO8 YO4 YO2 YO1
- Only need to look at low-order two digits of the year
  all years ending in 00, 04, 08, 12, 16, 20, etc. are divisible by 4
  - if tens digit is even, then divisible by 4 if ones digit is 0, 4, or 8
  - if tens digit is odd, then divisible by 4 if the ones digit is 2 or 6.
- Translates into the following Boolean expression
  (where YT1 is the year's tens digit low-order bit, YO8 is the high-order bit of year's ones digit, etc.):

YT1' (YO8' YO4' YO2' YO1' + YO8' YO4 YO2' YO1' + YO8 YO4' YO2' YO1')
+ YT1 (YO8' YO4' YO2 YO1' + YO8' YO4 YO2 YO1')

→ YT1' (xx00) → YO2' · YO1'
→ YT1 (0xx0) → YO4' · YO1'

simplify

- Digits with values of 10 to 15 will never occur, simplify further to yield:

YT1' YO2' YO1' + YT1 YO2 YO1'

Divisible-by-100 and divisible-by-400 circuits

- Divisible-by-100 just requires checking that all bits of two low-order digits are all 0:

YT8' YT4' YT2' YT1' • YO8' YO4' YO2' YO1' → xx00

- Divisible-by-400 combines the divisible-by-4 (applied to the thousands and hundreds digits) and divisible-by-100 circuits

(YM1' YH2' YH1' + YM1 YH2 YH1') → AB00 ∧ AB%4==0
• (YT8' YT4' YT2' YT1' • YO8' YO4' YO2' YO1') → xx00

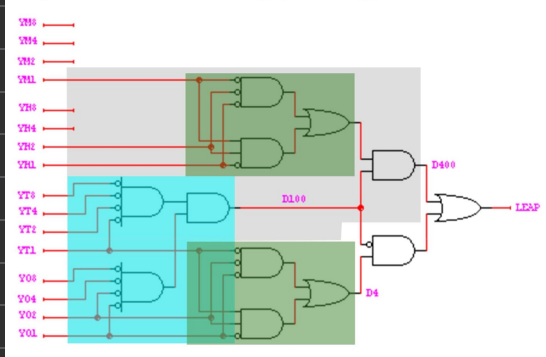
Label results of previous three circuits: D4, D100, and D400

combine 4 & 100 & 400

leap_year_flag = D4 (D100 • D400') '
= D4 • D100' + D4 • D400
= D4 • D100' + D400

leap year
↳ D4 • D100' + D400

Implementation of leap year flag



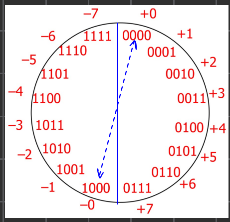
Number system

① Sign & magnitude

- → last bit = sign bit → 1 : − / 0 : +
- → Another bit = value bit

pro : easy implement

con : hard subtraction, lost 1 bit (1000 = 0000)



② 1's complement

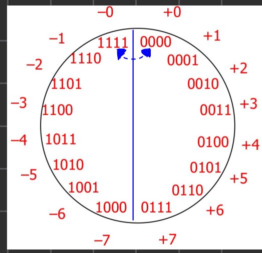
→ ⊕ → ⊖ หรือ ⊖ → ⊕

วิธี 1 toggle All bit

วิธี 2 $N' = (2^N - 1) - N \rightarrow X - N = X + N' = X + (2^N - 1) - N$ (loop to start)

pro : easy subtraction

con : hard implementation, lost 1 bit (1111 = 0000)



③ 2's complement

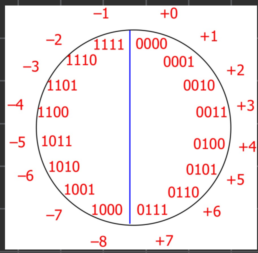
→ ⊕ → ⊖ หรือ ⊖ → ⊕

วิธี 1 toggle All bit → +Value (1)

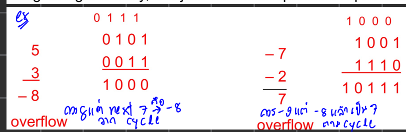
วิธี 2 $N' = 2^N - N \rightarrow X - N = X + N' = X + (2^N) - N$ (loop to start)

pro : easy subtraction, use all bit

con : hard implementation

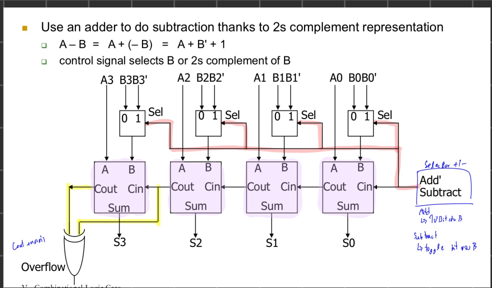


- Can't ignore it completely // 2' complement
  - needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can be ignored (ไม่ใช่ error แค่ value loop รอบ 2^N)
  - – M + N when N > M:
  - $M^* + N = (2n - M) + N = 2n + (N - M)$
  - ignoring carry-out is just like subtracting 2n
  - – M + – N where N + M ≤ 2n–1
  - $(-M) + (-N) = M^* + N^* = (2n - M) + (2n - N) = 2n - (M + N) + 2n$
  - ignoring the carry, it is just the 2s complement representation for – (M + N)

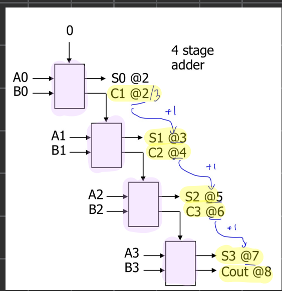


Adder

① Adder 1 ; ทำเหมือนตอนทด



เหมือนกัน ⟷



truth table for full Adder

Ai	Bi	Cin	Sum	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

pro : least transistor

con : last value sum & carry slow because $(sum \& carry)_i = input + carry_{i-1}$

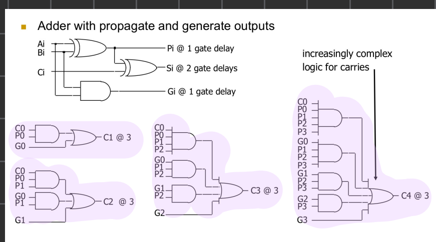
② Adder 2 ; คำนวณ carry ทุกตัวก่อน

- Carry generate: Gi = Ai Bi
  - must generate carry when A = B = 1 (หา function carry i)
- Carry propagate: Pi = Ai xor Bi
  - carry-in will equal carry-out here
- Sum and Cout can be re-expressed in terms of generate/propagate:
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Ci Pi

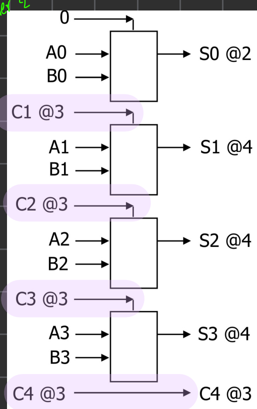
$(A_i + B_i) \rightarrow (A_i \text{ xor } B_i)$

Ai	Bi	+	xor
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	0

สร้างเป็น → function carry



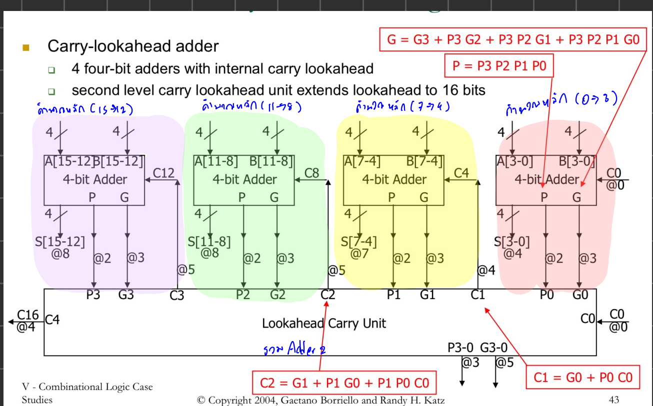
Adder 2



pro : fast calculation // not wait carry from before output

cons : Used a lot transistor for function carry

③ Adder 3 = Adder 1 + Adder 2 ; structure อย่าง Adder 1 แต่ function Adder เป็น Adder 2



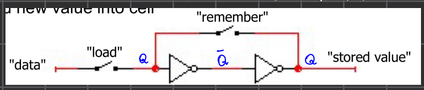
Sequential circuit : circuit with feed back

Output = function(inputs, {past inputs, memory})

① Simple circuit with feedback

• remember : ให้ remember value ปัจจุบัน

• load(on) : set "stored value" ตาม "data" ที่จะต่างๆมา



หรือ • เป็น →▷o→▷o เพื่อให้ stored value ได้

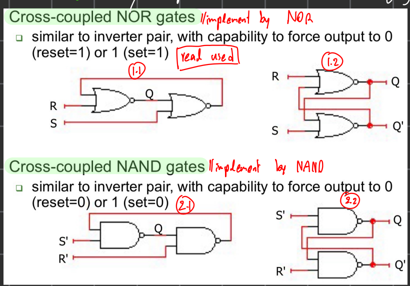
//case เป็น 1 แบบ load opened → เก็บไม่ได้ ∴ มี →▷o เพื่อจำไม่ได้ →▷o→▷o เพื่อ reverse value

• เป็น →▷o→▷o เพื่อเก็บค่าจาก end แล้ว start ไป end

remember \ load	closed	opened
closed	Value = load(now)	Value = load(now)
opened	Value = load(ล่าสุด)	Value = 0 //ไม่ได้ใช้งานจริง

ทบทวน Latch : memory ที่ไม่มี clock
flip flop: memory ที่มี clock

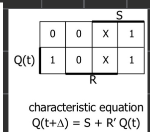
② SR Latch : easy implement but data changing all the time

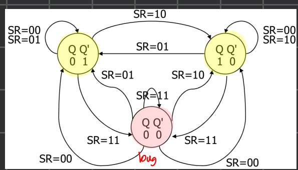


X.1 : → hard to understand (วงจรไม่สมมาตร)
→ easy to implement real circuit

X.2 : → easy to understand (สมมาตร)
→ hard to implement real circuit

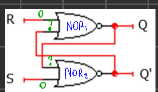
S (set)	R (Reset)	Q(t)	Q(t+Δ) (change to)	
0	0	0	0	hold
0	0	1	1	
0	1	0	0	reset
0	1	1	0	
1	0	0	1	set
1	0	1	1	
1	1	0	X	not allowed
1	1	1	X	bug: Q = Q̄

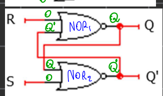




S = 0 , R = 0

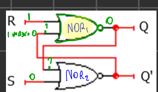
→ NOR₁ ไม่รู้ Output ไร (เพราะรู้แค่ wire เดียว 0) → รอรับ wire ของ Q (X NOR 0 = X̄)
∧ NOR₂ ไม่รู้ Output ไร (เพราะรู้แค่ wire เดียว 0) → รอรับ wire ของ Q (X NOR 0 = X̄)
→ NOR₁ อีก wire คือ Q' → Output Q = 0 + Q' →▷o Q
∧ NOR₂ อีก wire คือ Q → Output Q' = 0 + Q →▷o Q'

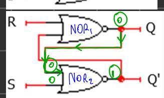




S = 0 , R = 1

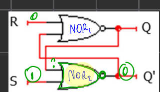
→ NOR₁ Output Q = 0 (1 NOR X = 0)
∧ NOR₂ ไม่รู้ Output ไร (เพราะรู้แค่ wire เดียว 0) → รอรับ wire ของ Q (X NOR 0 = X̄)
→ NOR₁ Output Q = 0
NOR₂ อีก wire คือ Q = 0 → Output Q' = 0 NOR 0 = 1

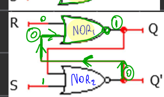




S = 1 , R = 0

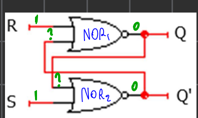
→ NOR₁ ไม่รู้ Output ไร (เพราะรู้แค่ wire เดียว 0) → รอรับ wire ของ Q' (X NOR 0 = X̄)
∧ NOR₂ Output Q' = 0 (1 NOR X = 0)
→ NOR₁ อีก wire คือ Q' = 0 → Output Q = 0 NOR 0 = 1
NOR₂ Output Q' = 0





S = 1 , R = 1

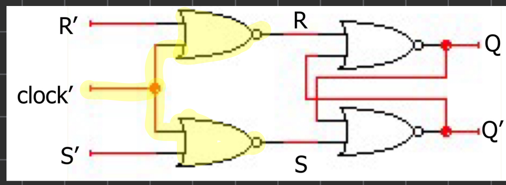
→ NOR₁ Output Q = 0 (1 NOR x = 0)
∧ NOR₂ Output Q' = 0 (1 NOR x = 0)
∴ Q = Q' → bug



③ RS Latch + clock = RS flip flop

pro : data changing when clock changing

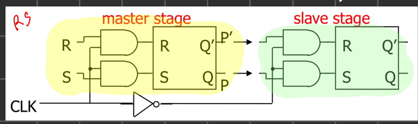
Con : Racing Condition (output ผิดพลาด เนื่องจากการรอ input จาก Q ∨ Q' ในช่วง ขณะนั้น)

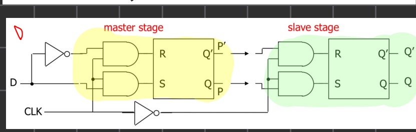


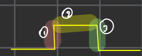
④ Master - Slave flip flop : ต่อ 2 flip flop

Pro : แก้ Racing condition ได้ จากการรอ input master state

Con : อาจเกิด wrong value จาก master stage





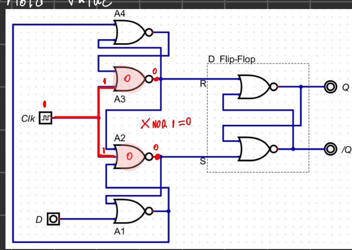
clock 

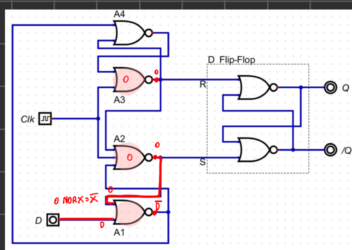
① clock เริ่ม = 1 → master state start
② clock เบา → master state work + hold output
③ clock ถึง = 0 → slave ทำงาน จาก output master state

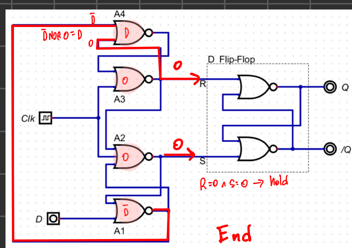
⑤ edge Trigger memory : ใช้ 2 flip flop hold memory

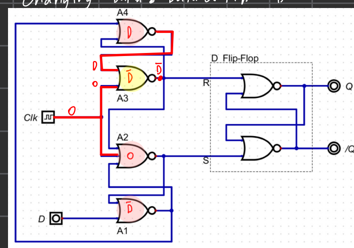
clock = 1 (clock(T) = 0 → clock(T+1) = 1)
↳ hold value

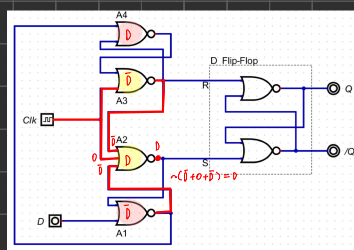
clock = 0 (clock(T) = 1 → clock(T+1) = 0)
↳ changing Data : Data ∝ input "D"

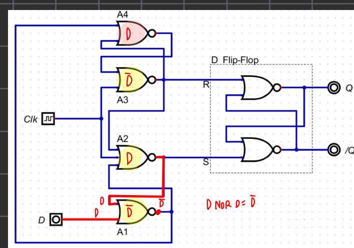


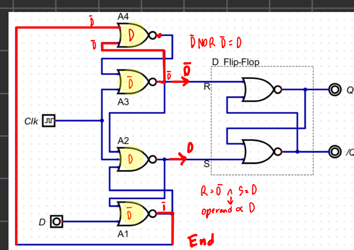












clock 

① clock 1→0 : } hold value
② clock 0 :
③ clock 0→1 : chaing value

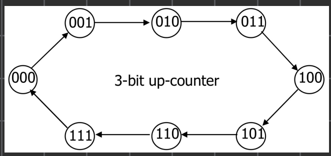
⑥ Register : collection of flip-flop/Latch for stored bits

FSM (Finite State Machine)

Common FSM

① example problem: bit counter

① create State diagram (problem)



: output ของแต่ละ clock จะต่างกันไปใน state ถัดไป

out(T) = x → out(T+1) = y

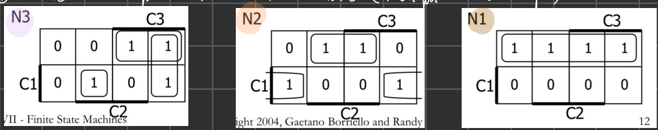
② create truth table (input: Each index of present State, Output: Each index of next State)

present state		next state	
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	101	5
5	101	110	6
6	110	111	7
7	111	000	0

→

C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

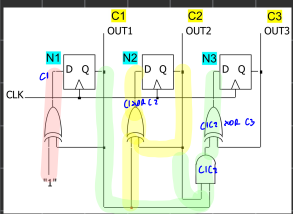
③ Create K-map from truth table (1 output = 1 K-map)

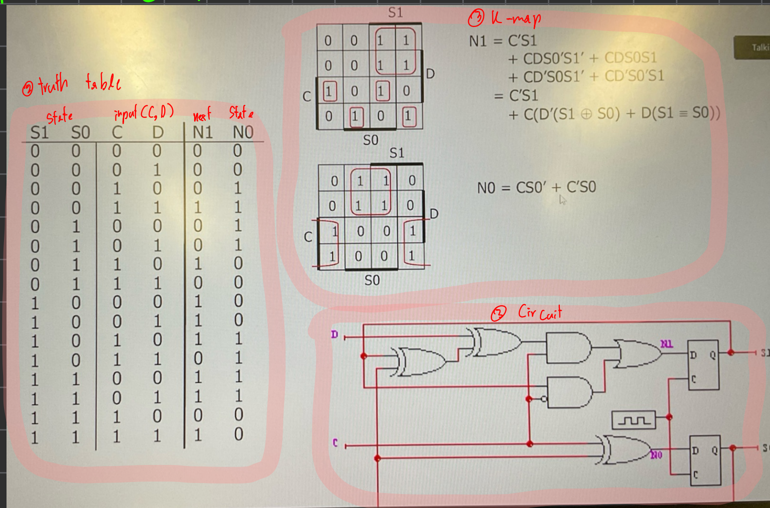


N1 <= C1'
N2 <= C1C2' + C1'C2
<= C1 xor C2
N3 <= C1C2C3' + C1'C3 + C2'C3
<= (C1C2)C3' + (C1' + C2')C3
<= (C1C2)C3' + (C1C2)'C3
<= (C1C2) xor C3

→ output from K-map

④ Create Circuit





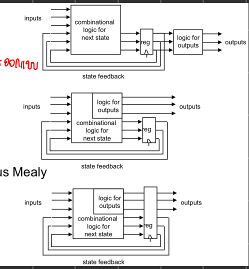
state S1	state S0	input (C, D) C	input (C, D) D	next state N1	next state N0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

Moore & Mealy: types of FSM

Feature	Mealy Machine	Moore Machine
Output Dependency	Depends on current state and input	Depends only on current state
Output Association	Associated with transitions	Associated with states
Generality	More general; can require fewer states	Less general; may require more states
Response Speed	Faster response to inputs (input change → state change (immediately))	Slower response; changes at clock edges
Implementation Complexity	More complex due to input dependency	Simpler; outputs tied to states

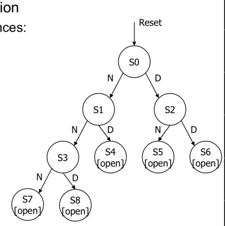
- Moore
- Mealy
- Synchronous Mealy

ต่างที่แนวทางการออกแบบ



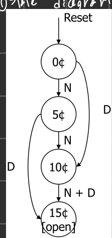
① example problem

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)



(1.1) implement by moore

① State diagram



② truth table

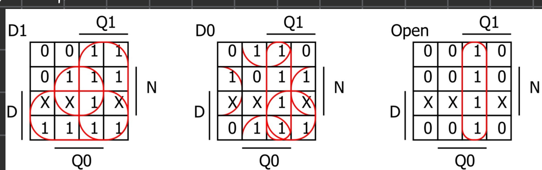
present state	inputs D	inputs N	next state	output open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	–	–
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	–	–
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	–	–
15¢	–	–	15¢	1

symbolic state table

→

present state Q1 Q0	inputs D	inputs N	next state D1 D0	output open
0 0 (0¢)	0	0	0 0	0
	0	1	0 1	0
	1	0	1 0	0
	1	1	– –	–
0 1 (5¢)	0	0	0 1	0
	0	1	1 0	0
	1	0	1 1	0
	1	1	– –	–
1 0 (10¢)	0	0	1 0	0
	0	1	1 1	0
	1	0	1 1	0
	1	1	– –	–
1 1 (15¢)	–	–	1 1	1

③ K-map

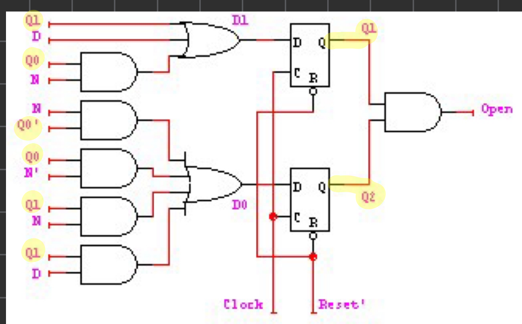


D1 = Q1 + D + Q0 N

D0 = Q0' N + Q0 N' + Q1 N + Q1 D

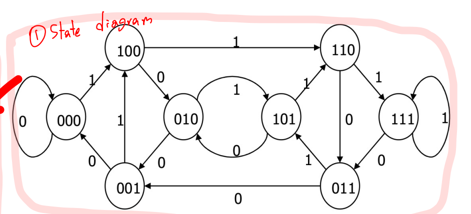
OPEN = Q1 Q0

④ Circuit



② example problem: Shift register (การ insert new input ที่ left ของ String 3 bit)

① State diagram



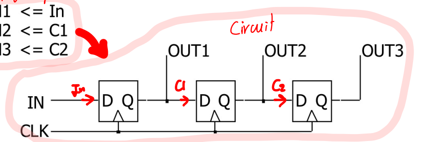
② truth table

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

out K-map

N1 <= In
N2 <= C1
N3 <= C2

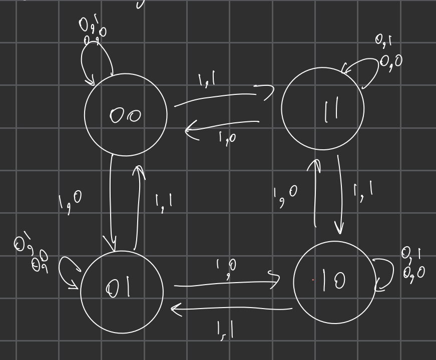
Circuit



③ example problem

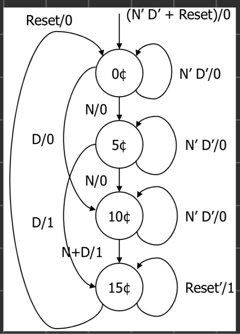
2-bit up-down counter (2 inputs)
- direction: D = 0 for up, D = 1 for down
- count: C = 0 for hold, C = 1 for count

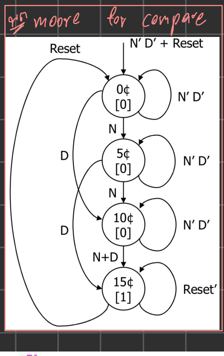
① state diagram



(1.2) implement by mealy

① State diagram

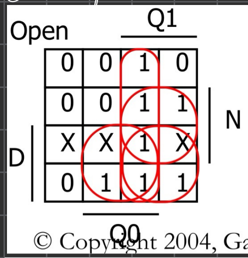




② truth table

present state Q1	Q0	inputs D	N	next state D1	D0	output open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	–	–	–
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	1
		1	1	–	–	–
1	0	0	0	1	0	0
		0	1	1	1	1
		1	0	1	1	1
		1	1	–	–	–
1	1	–	–	1	1	1

③ K-map

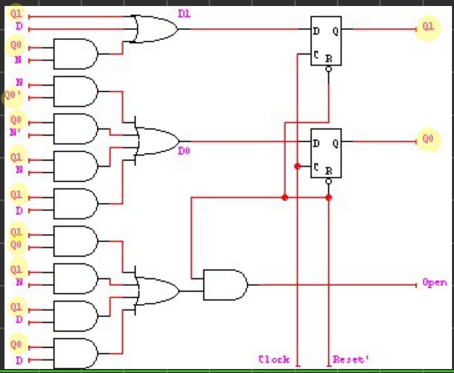


D0 = Q0'N + Q0N' + Q1N + Q1D

D1 = Q1 + D + Q0N

OPEN = Q1Q0 + Q1N + Q1D + Q0D

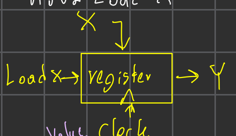
④ Circuit



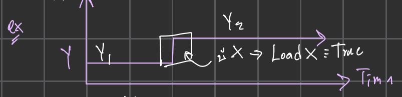
FSM Chart

① register :

- ใช้ปล่อย value Y ตลอดเวลา
- มี X มาตลอดเวลา ถ้าต้องการ Y เปลี่ยนค่า ต้องมี Load X



ex



Y_1 = Y ค่าเดิม

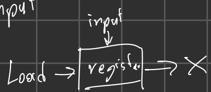
Y_2 = X

② variable ใน FSM

- ลองเขียนเป็น code จริง
- การ declare ตัวแปรคือ register

eg (I) X = input

ได้

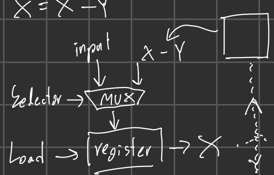


(II) X = input

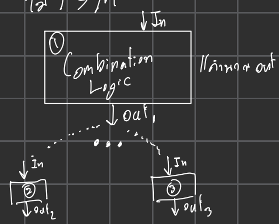
loop :

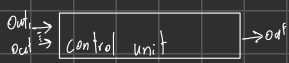
X = X − Y

ได้



② if ใน FSM

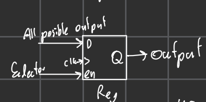




// ตอน control ในการใช้ In = True เพื่อทำงานของ State ใหม่ จาก out แต่ละ block

③ Loop ใน FSM : คือ If แต่แค่ย้อนกลับ จะสามารถสั่ง Control unit สั่งทำ State ก่อนหน้า

④ สร้าง output



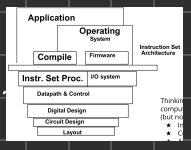
// Selector = output comparator สำหรับ Check this output

⑤ clear ใช้ที่ Control Unit Set State = 000 เสมอ (State Start)

Computer Architecture

①

- ISA = Instruction set Architecture
  ↳ set instructions the computer's processor can execute.
- Moore's Law
  ↳ the number of transisters on microchips x2 every 2 years



②

① Performace = $\frac{1}{\text{Execution time}}$

② Speedup (n) = $\frac{\text{Performance new (ใช้)}}{\text{Performance old (ไม่ใช้)}} = \frac{\text{Execution time (old)}}{\text{Execution time (new)}}$

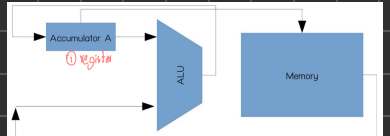
③ Elapsed time (sum Time) = CPU time + I/O time + Memory time (เวลาที่ผ่านไป)
↳ Response time = เวลาในการตอบสนอง

- Through put : speed (Task / time)
  ↳ ex Unit • MIPS = millions of instructions per second
  • FLOPS = Floating point Operations per second
- Benchmarks (เกณฑ์มาตรฐาน)
  ① Real Benchmark = real app
  ② Microbench mark = part of software
  ③ Kernels = specific operation
  ④ Toys benchmark = small programs (ex sorting)
  ⑤ Synthetics bench marks = mimic (เลียนแบบ) behavior of real programs
- SPEC Benchmark Suites
  ↳ Standard Performance evaluation Cooperation (SPEC)
- Performance Comparison
  ① Arithmetic Mean = $\frac{\sum x_i}{n}$
  ② Weighted Arithmetic Mean = $\sum_n (w_i x_i)$
  ③ Geometric Mean = $\sqrt[n]{\prod x_i}$
- CPU Time
  ① CPU Time = Instruction × CPI (clock per Instruction $\left(\frac{1}{\text{instruction}}\right)$) × cycle Time
  = (Number of clock) × cycle Time (time per 1 clock)
  ② CPI average = $\sum$ (CPI × Weight) $\left(\frac{1}{\text{Instruction}}\right)$
  ③ cycle Time (second) = $\frac{1}{\text{Clock Rate (Hz)}}$ $\left(T = \frac{1}{F}\right)$
- Amdahl's Law : Law of diminish return
  ① Overall Speedup = $\frac{\text{old Speed}}{\text{new Speed}} = \frac{1}{\sum\left(\frac{P}{r}\right)} = \frac{P}{r_1} + \frac{P_{rest}}{r_{rest}} + \ldots = \frac{1}{(1-P) + \frac{P}{n}}$
  ↳ time
  • P (percent) = อัตราร้อยที่ใช้ ใน overall time ∴ $\sum P_i = 100\%$
  • r (rate) = r_i เมื่อ อัตราร้อยที่เร็วขึ้น ตามรับ P_i (ยกตัวอย่าง เร็วขึ้นกี่เท่า)

③

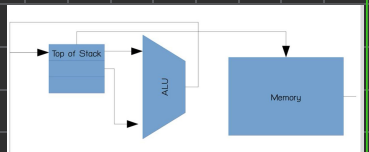
CPU Architecture

① Accumulator Architecture



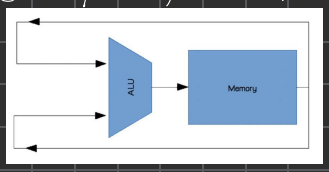
- Load acc A
  → acc A = acc A + b (a)
  → Store x ← acc A

② Stack Architecture



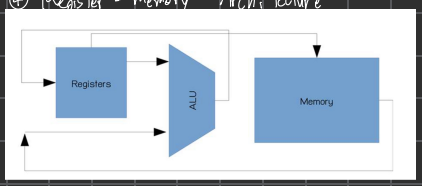
- implement like postfix

③ Memory - Memory Architecture



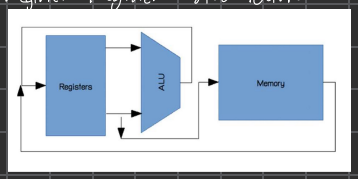
ex MUL k, a, b ; k = a * b

④ Register - memory Architecture



ex LD $r1, a ;
MUL $r1, b ; $r1 = $r1 * b = a * b

⑤ Register - Register Architecture



ex LD $r1, a ;
LD $r2, b ;
MUL $r3, $r1, $r2 ; // $r3 = $r1 * $r2

	Pros	Cons
①	easy implement	Bottleneck, Bad for parallelism, ↑ memory
②	easy compiler	Bottle neck, Difficult for optimization
③	Fewer instruction, Easy compiler	↑ memory traffic
④	easy encode, Good code density	Limited register
⑤	simple to encode, easy parallelism	↑ instruction count, Performance ∝ compiler

- 2 Paradigms of CPU Architecture
  ↳ CISC : powerful instruction (complex) ex Intel
  ↳ RISC : basic for building block ex ARM (low E consume)

Memory

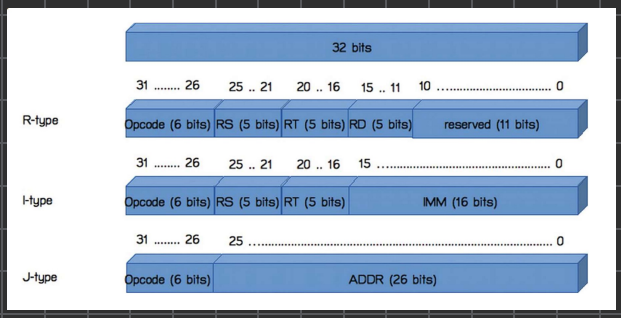
★ Little Endian easy implement Adding (Handle carry)
- Low-order byte stores at the lowest address
- Used by Intel

★ Big Endian easy compare ex sign bit
- High-order byte stores at the lowest address
- Used by IBM PowerPC, Sun

★ Note that modern processor (ie. ARM from ARM6 onwards) have the option of operating in either little-endian or big-endian mode.

→ @	78	56	34	12
→ @	12	34	56	78

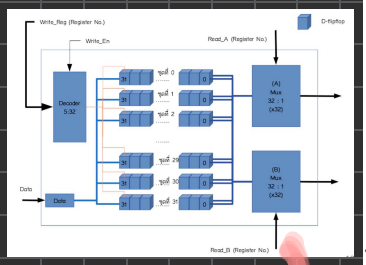
- Unrestricted Alignment : วาง ตัวแปร ตาม order declare
  ↳ Pros : Software simple
  ↳ Cons : Expensive logic, Bad Performance
- Restricted Alignment : Start index % size == 0
  ↳ Pros : Burden is on the software (Compiler)
  ↳ Cons : Good Performance
- caller & callee : caller calls callee
  ↳ ex main(caller) call function Add (callee)
- Handlers : handling of interrupts and exceptions
  ① Vector Table : Contains @ interruption/exception handler.
  ② Handler table : Contains functions/blocks that handler the interruption/exception
  ex If an interrupt occurs, the vector table point to @ in Handle table.

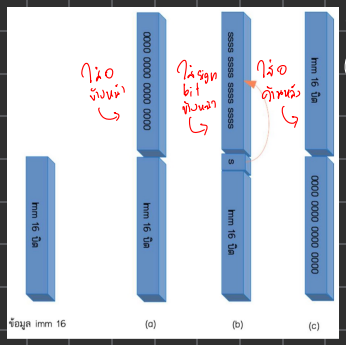


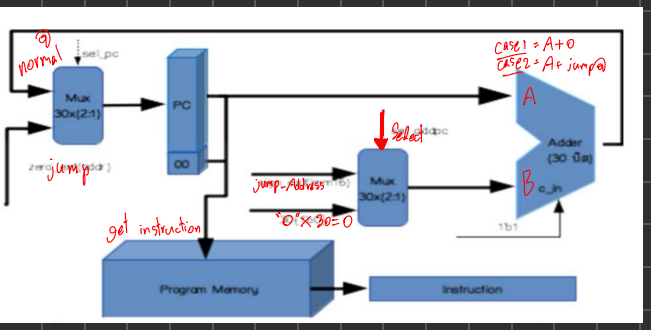
④ Single cycle processor

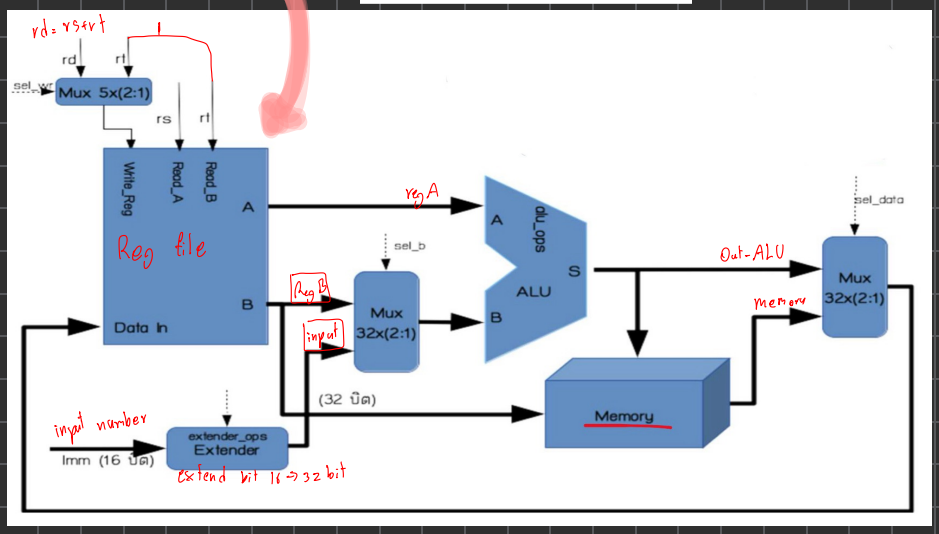
- 1 instruction per 1 clock
- ∴ 1 clock must be longest for any instruction time.
- ∴ Sum time) = Σ (max time instruction)

- ALU = Arithmetic Logic unit





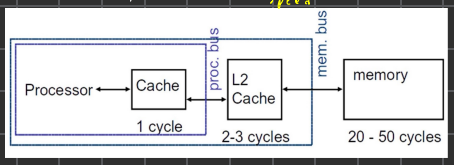




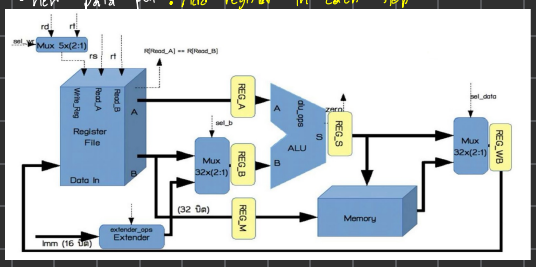
⑤ Multiple cycle processor

- ★ Instructions are explained with RTL.
- ★ ORI rt, rs, imm ; (I-type) ; Opcode 010000 (OR immediate)
  - R[rt] ← R[rs] | zero_ext(imm);
  - PC ← PC + 4
- ★ ORUI rt, rs, imm ; (I-type) ; Opcode 010001 (OR unsigned immediate)
  - R[rt] ← R[rs] | zero_pad(imm);
  - PC ← PC + 4
- ★ ADD rd, rs, rt ; (R-type) ; Opcode 000001
  - R[rd] ← R[rs] + R[rt];
  - PC ← PC + 4
- ★ LW rt, imm(rs) ; (I-type) ; Opcode 011000 (Load word)
  - R[rt] ← MEM[R[rs] + sign_ext(imm)];
  - PC ← PC + 4
- ★ SW rt, imm(rs) ; (I-type) ; Opcode 011100 (Store word)
  - MEM[R[rs] + sign_ext(imm)] ← R[rt];
  - PC ← PC + 4
- ★ BEQ rs, rt, imm*4 ; (I-type) ; Opcode 100100 (Branch if equal)
  - If (R[rs] == R[rt]) then
  - PC ← PC + 4 + (sign_ext(imm) * 4)
  - else
  - PC ← PC + 4
- ★ JMP addr*4 ; (J-type) ; Opcode 110000 (Jump)
  - PC ← (addr * 4)

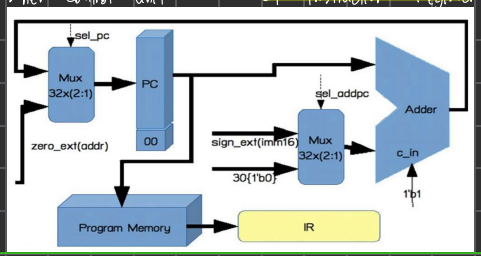
- new memory : Size memory ∝ $\frac{1}{\text{Speed}}$



- new Data path : Add register in each step

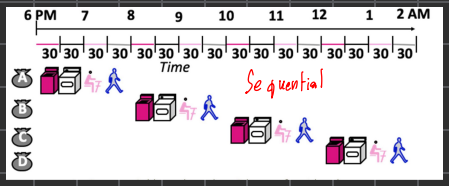


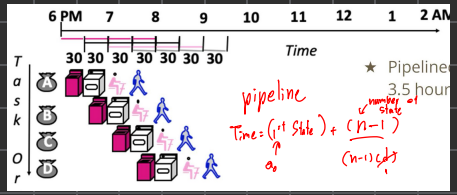
- new Control Unit IR = instruction Register

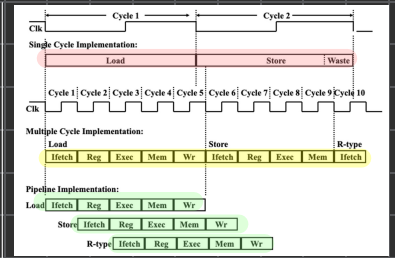


④ & ⑤ pipeline

- pipe line = The idea is to start a new task as soon as the resource are available.







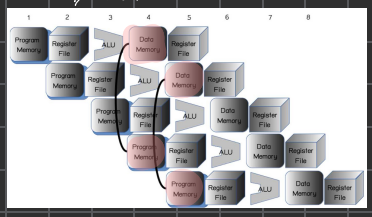
single cycle
↳ Time = Instruction x cycle time
multiple cycle
↳ Time = Instruction x CPI x cycle time
pipe line
↳ Time = (All state) x cycle time

Pipeline Hazards

① Structural hazard: Limited in resource
↳ some component required at the same time.

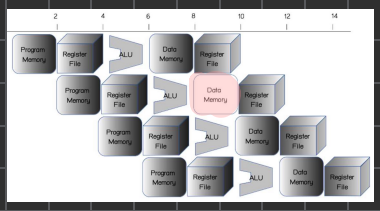
Sol ① Add more instances (eg ALU)

② seperate memory



popular

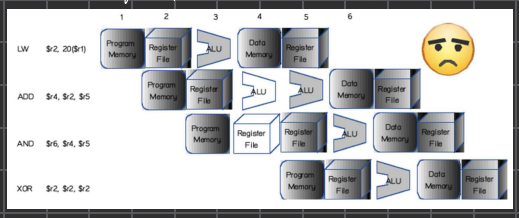
③ Redesign / Add dummy



② Data hazard : dependency of work / data.
↳ required data but data not yet.

Sol ① stall

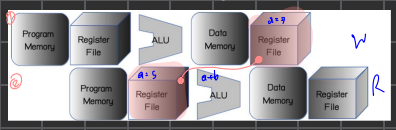
② Redesign / Reschedule (Reorder command)




```
LW      $r2,20($r1)
AND     $r4,$r2,$r5
```

```
LW      $r2,20($r1)
NOP
NOP
NOP
AND     $r4,$r2,$r5
```

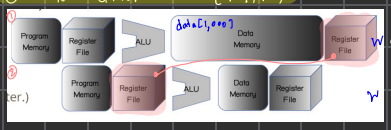
type ① Read after Write (RAW): Read ก่อน



ex `int a=5; a=7; int c=a+b;` } use a=5

∴ use (Read) old value (after write)

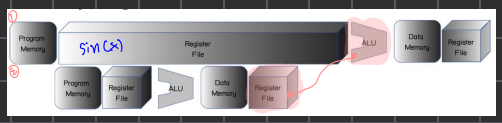
② Write after Write (WAW)



ex `a=data[1,000]; a=10; b=a+3;` } ∴ a=data[1,000]

∴ write old value after write new value

③ Write after Read (WAR) : Read ก่อน



ex `a=sin(x); x=10;` } a ใช้ x=10 (เดิม)

∴ use new value that just finished

! write = assign value
Read = use value / Arithmetic

③ Control hazard : don't know what to do next.
↳ In case jump. If work on jump → don't know pre instruction should be next-instruction / jump-instruction

Sol ① stall

② Guess: work ! → correct: no delay
→ incorrect: delay 1 clock

③ Reschedule

Without delay slot			With delay slot		
(1)	**ADD**	**$r1, $r2, $r3**	(2)	BEQ	$r2, $r3, BRANCH
(2)	BEQ	$r2, $r3, BRANCH	(1)	**ADD**	**$r1, $r2, $r3**
(3)	ORI	$r4, $r5, #100	(3)	ORI	$r4, $r5, #100
BRANCH:			BRANCH:		
(9)	SW	$r1, 100($r3)	(9)	SW	$r1, 100($r3)

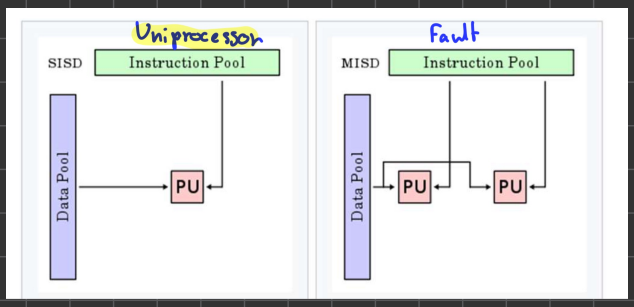
Flynn's Taxonomy ⇒ แบ่งตามจำนวน Instruction stream, Data stream

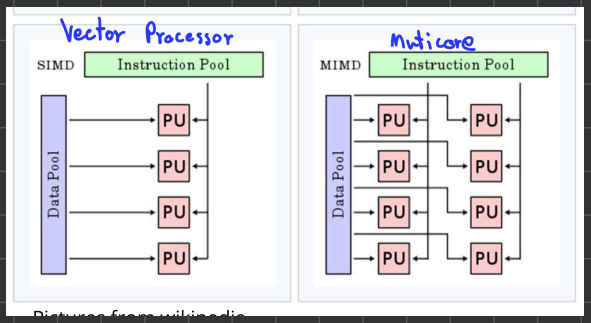
Uniprocessor : Single instruction stream & Single data stream (SISD)

Vector processor : Single instruction stream & Multi data stream (SIMD)

Fault tolerant Computing : Multi instruction stream & Single data stream (MISD)

Multi core : Multi instruction stream & Multi data stream (MIMD)





Pictures from wikipedia

ILP (Instruction level Parallelism)

↳ ค่าที่บอกถึง "จำนวน" ชิ้นส่วนที่ทำพร้อมกันได้

- Pipelining Processor ; (ILP = k)

$$T(n) = (k + (n-1))\,T_c$$

* instruction

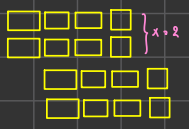
Time/clock ที่ใช้ได้

k = state of pipeline

- Super Scalar ; (ILP = kx)

$$T(n) = \left(k + \frac{(n-x)}{x}\right) T_c$$

x = degree of superscalar

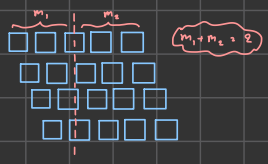


! Super Scalar อาจ may have out of order execute.
will get WAR / WAW
∴ solve with using In order completion

- Super Pipeline ; (ILP = km)

$$T(n) = \left(k + \frac{(n-1)}{m}\right) T_c$$

* minorcycle

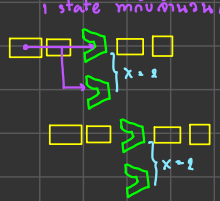


- Super Pipeline SuperScalar ; (ILP = kmx)

$$T(n) = \left(k + \frac{(n-x)}{mx}\right) T_c$$

Tip: mix Super Scalar & Super Pipeline

1 state ที่กับจำนวน ALU ตาม x degree



- VLIW (very long instruction word) ;

$$T(n) = \left(k + \left(\frac{n}{x} - x\right)\right) T_c$$

* degree of VLIW

Complier ⇒ ก็สามารถลดส่วนช่วยให้ ILM เพิ่มขึ้น

- Small basic block : less Toms for filling available slot.
- Large basic block : more Toms for filling slot.

Loop Unroll ⇒ ทำให้ loop ใหญ่ขึ้น ด้วยการซ้อน iteration

Normal Loop

```
LOOP:
    LW    $f0, $r1
(stall)
    ADD   $f2, $f0, $f1
(stall)
(stall)
    SW    $f2, $r1
    SUB   $r1, $r1, #4
    BNE   $r1, LOOP
```

8 cycles / Iteration

Unroll + Rescheduling

```
LOOP:
    LW    $f0, 0($r1)
    LW    $f3, 4($r1)
    LW    $f5, 8($r1)
    LW    $f7, 12($r1)
    ADD   $f2, $f0, $f1
    ADD   $f4, $f3, $f1
    ADD   $f6, $f5, $f1
    ADD   $f8, $f6, $f1
    SW    $f2, 0($r1)
    SW    $f4, 4($r1)
    SW    $f6, 8($r1)
    SW    $f8, 12($r1)
    SUB   $r1, $r1,#16
    BNE   $r1, LOOP
```

14 cycles / 4 Iteration
= 3.5 cycles per iteration

! แต่ทำให้ใช้เวลาเร็วขึ้น

Dynamic scheduling (Hardwere)	Static scheduling (Software/compiler)
- Out of order excute. - Loop dynamic unroll - Multiple instruction dynamic	- loop static unroll - Multiple instruction static

Chip with Multicore

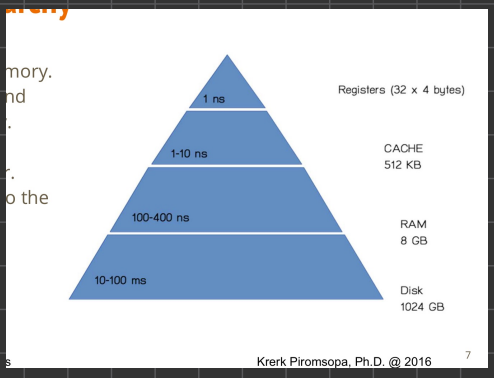
↳ have multiple superscalar inside
↳ also be viewed as multiple pipes

Thread level Parallelism

⇒ ถ้าเติมลง slot ที่ว่างอยู่ไม่ได้แล้ว ⇒ เลือก instruction จาก thread อื่นๆ
- instruction in pipeline come from many thread
- เทคโนโลยีนี้ชื่อ "Simultaneous Multitreading"(SMT)

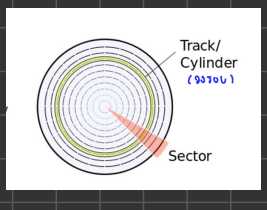
Cache : อยู่ใน CPU

คิดจากความคิดว่าทำให้มันทำเร็ว แต่ราคาถูกๆ
⇒ ทำ mem hierachy



Memory Tech

- RAM (random access memory)
- DRAM (Dinamic random access memory) (capacistor)
  ↳ high density, low power, slow, cheap (good for ram)
- SRAM (static random access memory) (transistor)
  (good for cache) ↳ low density, high power, expensive, fast
  * ต้องคอยการันตี Power on
- Semi random access tech.



Locality

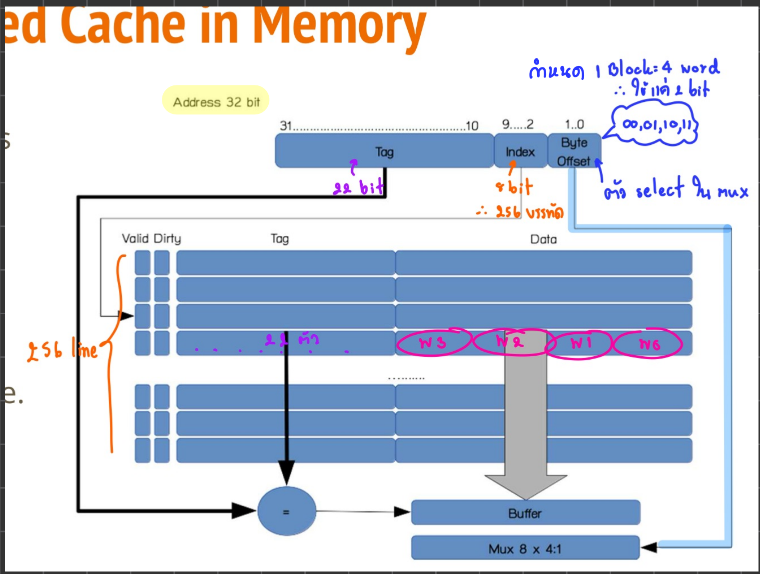
- Temporal = Time ~ ใช้สม่ำเสมอ
- Spacial = space ~ ข้อมูลอยู่ใกล้กัน

Hit : can find data in cache

Miss : cann't find data in cache

Block : เป็น chunk ที่ทำหน้าที่ ดูแล การ in/out ใน cache

Directed mapped



ดูจาก block size, cache size

block size = $2^{word} = 2^4 = 16$

cache size = 16x4 = 64

Performance

want to minimize ??

reduce this

- Mem access time = hit time + (miss rate × miss penalty)
- CPI = ideal CPI + (Miss rate × Miss Penalty)

Source of miss

- Compulsory miss (cold miss) ⇒ No data in cache
- Capacity miss ⇒ cache size is too small
- Conflict miss ⇒ have free space in cache but การ map prevent ไว้

Cache Config

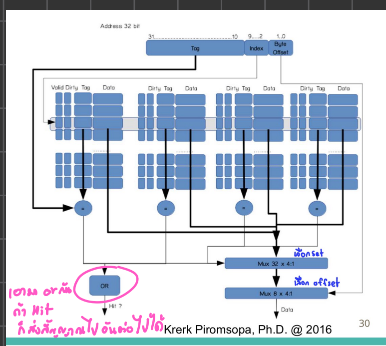
5 อย่าง ; cache size, block size, Associativity, replacement, write buffer

Block Size : larger size mean have more spatial but higer miss penalty (less temporal)

Two-way set associative : มี 2 ทางให้หาโดยใช้ index เดียวกัน (ตามด้วย Tag)

↳ less conflict :)

4-way set associative : more complex, 4 comparator, Large mux



replacement algorithm

- Least Recently Used (LRU) ⇒ แทนข้อมูลที่ไม่ได้ใช้แล้วนานสุด
- Round Robin ⇒ เอาตัวล่าสุดที่ไม่ได้ใช้ทับๆ วนกันไปเรื่อยๆ

Rep.	bench	Data	
		Miss	Miss rate
RR	gcc	36805	0.018
LRU	gcc	33126	0.017
RR	go	7255	0.005
LRU	go	9073	0.006

Write management

- write back : เขียนที่แคชอย่างเดียว ram จะถูกอัพเดท on replacement
  - Pro: good for repeat write, fast
  - Cons: incoherent data
- write through. เขียนทั้งที่ แคช และ ram
  - Pro: prefer for multiprocessor
  - cons: bad for repeat write, slow

Write allocate	write not allocate
- loading data before wire	- write direct without 1st loading

Mix ตัวย่อกันเถอะ

ISA (instruction set architecture)

ALU (arithmetic logic unit)

CPU (center processing unit)

MIPS (million instruction per seconds)

FLOPS (floating point operations per second)

SPEC (standard performance evaluation cooperation)

CPI (clock per instruction)

RISC (reduce instruction set computing)

CISC (complex instruction set computing

ABI (application binary interface)

RTL (register transfer language)

RTS (request to send)

Wr (write the data back to register)

WAR (write after read)

WAW (write after write)

RAW (read after write)

ILP (instruction-level parallelism)

VLIW (very long instruction word)

SISD (Single instruction stream & Single data stream)

SIMD (Single instruction stream & Multi data stream)

MISD (Multi instruction stream & Single data stream)

MIMD (Multi instruction stream & Multi data stream)

คำสั่ง
- LW (load word)
- SW (store word)
- ORI (or immediate) ที่ zero-extend
- ORUI (or unsiged immediate) ที่ zero-padded
- BEQ (branch if equl)

SMT (simultaneous multithreading)

ROM (read only memory)

RAM (random access memory)

DRAM (dynamic random access memory)

SRAM (static random access memory)

FIFO (first in first out)

LRU (least recenty used)

TLB (translation lookaside buffer)

NAS (network attached storage)

SAN (storage area network)

SSD (solid state drive)

HDD (hard disk drive)

JBOD (just a bunch of disk/drive)

replacement algorithm

- Least Recently Used (LRU) ⇒ แทนข้อมูลที่ไม่ได้ใช้แล้วที่นานสุด
- Round Robin ⇒ เอาตัวล่าสุดที่ไม่ได้ใช้ ทับๆ วนกันไปเรื่อยๆ
  - ↳ FIFO, Random

Rep.	bench	Data	
		Miss	Miss rate
RR	gcc	36805	0.018
LRU	gcc	33126	0.017
RR	go	7255	0.005
LRU	go	9073	0.006

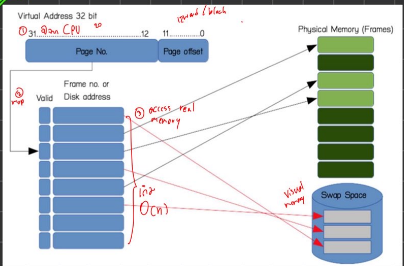
write management

- write back : เขียนที่แคชอย่างเดียว ram จะถูกอัพเดท (รอโดนแทน) on replacement
  - Pro: good for repeat write, fast
  - Cons: incoherent data
- write through: เขียนทั้งที่แคชและ ram
  - Pro: prefer for multiprocessor
  - cons: bad for repeat write, slow

Write allocate	write not allocate
- loading data before wire	- write direct without 1st loading

---

Sol ①



virtual Address = ⓐ in software

page table = ตัว map จาก physical

swap space = 2nd storage

page = visual block } size = จำนวน word ใน 1 block = 2^{offset}

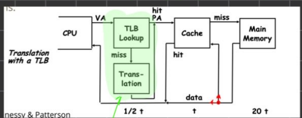
page frame = physical block

page faults = miss is called

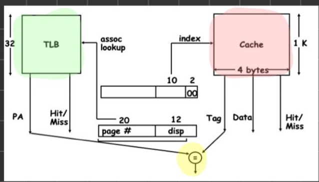
block | w_1 | w_2 | w_3 | w_4 |

size physical memory = $2^{(\text{len bit ⓐ})}$

Sol ②: Add TLB (translation look-aside Buffer)



upgrade →



: ดู table + cache พร้อมกัน

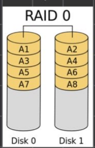
↓

Check ว่าตรงไหม

- ตรง → มีค่า
- ไม่ตรง → check memory

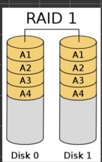
∴ Sol① ไล่ดูใน Cache "O(n)"

Sol② check ใน table "O(1)" → access cache ได้เลย

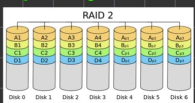


- Good performance
- low relibility

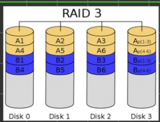
↘ รวม 0+1



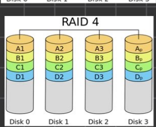
- security
- ซ้ำซ้อน



- ไม่นิยม & ไม่ใช้ประโยชน์

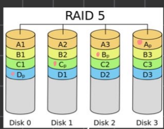


เก็บเป็น bits

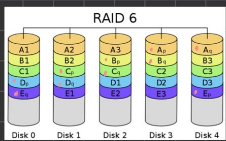


เก็บเป็น bytes

(sum) Disk 3 traffic เยอะ



กระจาย Load



เพิ่ม 1 disk ↓ มีรวมค่า ใน 2 disk (เพิ่ม security)

SAN = block storage
- fiber connect
- ↳ super hardisk

NAS = file storage
- ↳ file sever