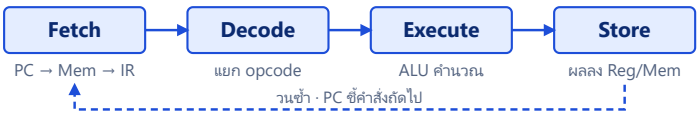
**Register PC IR MAR MBR** → น.1
**Interrupt · ISR · Nested** → น.1
**Cache · Tag/Index · Hit-Miss** → น.3
**DMA · I/O · RAID · HDD** → น.4
**CPU time · CPI · Speedup** → น.1
**2's Comp · Sign-Mag · IEEE** → น.2
**SRAM DRAM · Pyramid** → น.3
**Pipeline · Hazard** → น.4
**Bus · Daisy chain** → น.1
**RISC-V format · Branch** → น.2
**Write-through / back** → น.3
**Flynn · SIMD · GPU** → น.4

## Ch01–02 CPU ทำงานยังไง · Register ★ ดู flow ให้ออก




- **CPU** = Control Unit + ALU + Registers + Bus
- วงจร: **Fetch** → **Decode** → **Execute** → **Store** (Write back)
- **PC** เก็บ **ตำแหน่ง**คำสั่งถัดไป
- **IR** เก็บ**คำสั่งที่ fetch มาแล้ว**
- **Status/Flag** เก็บ overflow, negative
- **MAR** = address ใน Memory
- **MBR** = data ที่เข้า/ออก Memory
- **IOAR** = address ของ I/O
- **IOBR** = data ที่รับจาก I/O

**★ ตัวอย่างอาจารย์ (assignment)**
- PC = 300 → Mem[300] = **1940** → โหลดเข้า **IR**
- 1940: **4 บิตบน = opcode** (0001 = Load AC จาก Memory)
- ที่เหลือ **940 = address** ของข้อมูล
- Mem[940] = 0003 → **AC = 0003**

## Ch01 Performance · CPU time · CPI ★ ข้อคำนวณ

X เร็วกว่า Y กี่เท่า = **$Time_Y$ / $Time_X$**

CPU time = **IC × CPI × Cycle time**
= IC × CPI / Clock rate

Clock rate = 1 / Clock period

- เทียบ 2 เครื่อง ISA เดียวกัน → **ตัด IC ทิ้ง**
- ไม่รู้ค่า IC ให้ติดตัวแปร I ไว้
- ⚠ "เร็วกว่า" = ใช้เวลา**น้อยกว่า**
- อ่านโจทย์ดีๆ ว่าใครหารใคร

**ตย. สไลด์** A: cycle 250 ps, CPI 2.0 · B: cycle 500 ps, CPI 1.2
- A = I × 2.0 × 250 = **500I** ps
- B = I × 1.2 × 500 = **600I** ps
- → **A เร็วกว่า 1.2 เท่า**

**ตย. Clock** A: 2 GHz, 10 s · B ต้อง 6 s และใช้ cycles มากกว่า 1.2 เท่า
- $Clock_B$ = 1.2 × (10 × $2×10^9$) / 6 = **4 GHz**

## Ch02 Von Neumann · Harvard




| Von Neumann | Harvard |
|---|---|
| **Memory ก้อนเดียว** เก็บทั้งคำสั่ง + ข้อมูล | **แยก** Instruction Memory กับ Data Memory |
| คอมพิวเตอร์ทั่วไป | ส่วนใหญ่เป็น **Microcontroller** |
| **Hardware**: วงจรเฉพาะงาน เร็ว แต่ล็อกตายตัว | **Software**: ชุดคำสั่ง ยืดหยุ่น แต่ต้อง "คิด" ทุกคำสั่ง |

- **Interrupt** = สิ่งที่**ขัด**การทำงานแบบ sequence ของ CPU

## Ch02 Bus · Arbitration ★ Daisy chain = priority

- **Address bus** บอกตำแหน่งใน Memory
- **Data bus** ส่งข้อมูล
- **Control bus** สัญญาณเปิด/ปิด/เลือกอุปกรณ์
- **Point-to-point**: แยกทุกหน้าที่ ง่ายสุดแต่ **เปลืองเส้น**
- **Multipoint**: ประหยัด (ไฮเวย์ + ซอย) แต่ต้องมี **โปรโตคอลควบคุม**

| Arbitration | หลักการ |
|---|---|
| **1 Daisy chain** | ตาม **priority** (CPU > Memory > I/O)<br>**นิยมสุด ใช้ถึงวันนี้**<br>ข้อเสีย: priority ต่ำแทบไม่ได้ใช้ bus |
| **2 Centralized parallel** | ทุกอุปกรณ์ต่อตรงเข้า **ตัวควบคุมกลาง** |
| **3 Distributed self-detection** | ไม่มีตัวกลาง ตัดสินกันเอง เช็กเองว่ามีใครส่งอยู่ไหม |
| **4 Distributed + collision detection** | เหมือน 3 + ตรวจจับการชน ถ้าชน **ส่งซ้ำ** |




- ★ ตัวเลือกว่าอุปกรณ์ไหนได้ใช้ bus = **Bus Arbitrator**
- ⚠ **ไม่ใช่** Control Unit และ **ไม่ใช่** control line
- การ์ดจอเสียบเมนบอร์ดผ่าน **PCIe**
- **InfiniBand** = I/O ความเร็วสูงสุด ใช้กับ GPU ↔ memory

## Ch02 Interrupt ★ flow "จำไว้เลย"

**★ Interrupt flow**
1. **Save** register / PC / status ปัจจุบัน
2. เปิด **Interrupt Vector table** หา **ISR address** (เช่น เบอร์ 003 = คีย์บอร์ด)
3. ใส่ address นั้นใน **PC** → รัน **ISR**
4. **Restore** ค่าที่ save ไว้ → กลับไปทำงานเดิม




- **Sequential**: เกิด → ทำ → กลับ → เกิดอีก → ทำ → กลับ
- **Nested**: Y ซ้อนเข้ามาระหว่าง X → save X, ทำ Y, คืนค่า X, ทำ X ต่อ
- ซ้อนได้กี่ชั้น = จำนวน register ที่เก็บค่าได้

| ระบุว่าใครส่ง interrupt | ลักษณะ |
|---|---|
| Multiple lines | 1 เส้น/อุปกรณ์ แต่ I/O เยอะ เส้นก็เยอะ |
| Software poll | ไล่ถามทีละ module (ช้า) |
| Daisy chain | ส่ง acknowledge ต่อกันเป็นลูกโซ่ ตาม priority |
| Bus arbitration | ขอสิทธิ์คุม bus ก่อน แล้วส่ง vector ของ device |

## Ch03 Data Representation ★ "จะมีข้อนี้แน่ๆ" ต้องทำได้เต็ม

| | Sign-Magnitude | 2's Complement |
|---|---|---|
| **ค่าลบ** | กลับ **sign bit** (0 = +, 1 = −) ที่เหลือคือ magnitude | **กลับทุกบิต แล้ว +1** |
| **8 บิต** | −127 ... +127 | **−128 ... +127** |
| **ศูนย์** | มี 2 ตัว<br>0000 0000 / 1000 0000 | **ตัวเดียว** |
| **บวก/ลบ** | sign เหมือน → บวก magnitude<br>sign ต่าง → ลบตัวเล็กจากตัวใหญ่ ใช้ sign ตัวใหญ่ | A−B = A + (2's ของ B)<br>แล้ว **ทิ้ง carry-out** |



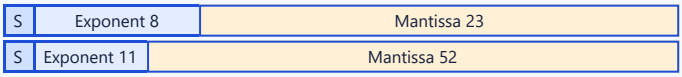

2's Complement แบบ 4 บิต ช่วง −8 ... +7 (ศูนย์ตัวเดียว)

**Shortcut 2's: จาก LSB เจอ 1 ตัวแรก คงไว้ แล้วกลับบิตทางซ้ายทั้งหมด**
**ตย. 0101011 → 1010101**

**ตย. สไลด์** 6 + (−5) แบบ 4 บิต
- 0110 + 1011 = 1|**0001**
- ทิ้ง carry → ได้ **1**
- Overflow: (+)+(+) ได้ลบ หรือ (−)+(−) ได้บวก

**ตย. 16 บิต (แนว assignment)**
- +100 = 0000 0000 0110 0100
- +37 = 0000 0000 0010 0101
- −100 SM = 1000 0000 0110 0100
- −100 2's = 1111 1111 1001 1100
- −37 2's = 1111 1111 1101 1011
- 100−37 = 1|0000 0000 0011 1111 → **63**
- 37−100 = 1111 1111 1100 0001 → **−63**

**IEEE 754** $N = (-1)^S \times 1.m \times 2^{E-127}$




- ★ **Single 32 บิต: exponent 8**
- ★ **Double 64 บิต: exponent 11** (อาจารย์ถาม)
- bias single 127 · double 1023

## Ch03 Assembly (RISC-V) ★ ข้อสอบ ไม่ต้องเขียนโค้ด

- ต้องรู้แค่ว่าคำสั่ง**ทำอะไร** ไม่เจาะจงตระกูล
- `ADD A,B,C` = **B + C เก็บใน A**
- **LOAD** = Memory → Register
- **STORE** = Register → Memory
- **ADDI** = register + **ค่าคงที่ (immediate)**
- `addi x22,x22,4` = x22 ← x22 + 4

### ★ Instruction Format (ถ้าออก **จะมี chart ให้**)

| | | | | | | |
|---|---|---|---|---|---|---|
| R | funct7 | rs2 | rs1 | f3 | rd | opcode |
| I | immediate 12 บิต | | rs1 | f3 | rd | opcode |

- **opcode 7 บิตล่าง** → $2^7$ แบบ
- **rs1, rs2, rd ละ 5 บิต** → $2^5$ = **32 register**
- I-format: **12 บิตบน = immediate**
- **ตย.** `add x9,x20,x21` → **opcode = 51**, rd = 9, rs1 = 20, rs2 = 21

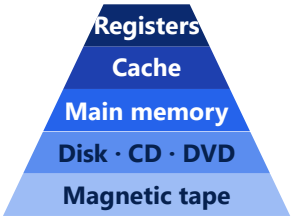
| กลุ่ม | คำสั่ง |
|---|---|
| Logic | **AND · OR · XOR** |
| Shift | **left = คูณ**<br>**right = หาร** |
| Branch<br>เช็กเงื่อนไขก่อน | `beq` เท่ากัน<br>`bne` ไม่เท่ากัน<br>`blt` rs1 < rs2<br>`bge` rs1 ≥ rs2<br>→ กระโดดไป L1 |
| Procedure | = ฟังก์ชัน = **software interrupt**<br>**save ทุกค่าลง stack** แล้ว return คืนจาก stack |

RISC-V ส่วน self-study ที่อาจารย์ทิ้งให้อ่านเอง: ไม่ออก

## Ch03 Endian ★ สลับกันแค่นี้

- **Little-Endian**: bit 0 (น้อยสุด) อยู่**ขวา**
- **Big-Endian**: bit 0 อยู่**ซ้าย**
- ตามสไลด์: Little เก็บ byte LSB ที่ address ต่ำสุด (RISC-V)

## Ch04 Memory Hierarchy ★ จำรูปพีระมิด




**เร็ว · แพง · จุน้อยสุด**
**SRAM** 0.5–2.5 ns
**DRAM** 50–70 ns
**Magnetic disk** 5–20 ms
**ช้า · ถูก · จุมากสุด**

- **Magnetic tape** = **ถูกสุด จุมากสุด** (offline) ★ **อาจารย์ถาม**
- **RAM** = ชั่วคราว (ไฟดับหาย)
- **ROM** = ถาวร (ปิดไฟแล้วอยู่)
- **Cache** คั่นกลาง CPU ↔ Main memory เก็บข้อมูลที่**ใช้บ่อย/ซ้ำ**
- **L1** อยู่ใน CPU · **L2/L3** อยู่นอก CPU
- **Hit** = เจอใน cache
- **Miss** = ไม่เจอ → ดึงจาก main memory มาใส่ cache แล้วส่ง CPU

## Ch04 Associative · Write Policy ★ ถูกถาม

- **Set-associative**: index เดียวเก็บได้หลายชุด
- **2-way** เก็บ 2 · **4-way** เก็บ 4 · **8-way** เก็บ 8
- แค่เพิ่มที่เก็บต่อ index → โอกาส hit มากขึ้น
- ★ **Fully associative = miss น้อยสุด** (hit มากสุด)
- Miss rate: **1-way 10.3% · 2-way 8.6% · 4-way 8.3% · 8-way 8.1%**
- ดีขึ้นแต่ได้น้อยลงเรื่อยๆ

| Write policy | ความหมาย |
|---|---|
| **Write-through** | เขียนลง cache แล้วส่ง CPU เลย |
| **Write-back** | ส่ง CPU **ก่อน** แล้วค่อยอัปเดต cache ทีหลัง |
| **Write-allocate** | อีกทางเลือกของ write-through |

ปัจจุบัน memory เร็วขึ้น กลไกการเขียนหมดความสำคัญ เน้นยัดโค้ดที่ใช้บ่อยเข้า L1 · แต่ต้องรู้ชื่อทั้ง 3 คำ

## Ch04 Direct-Mapped Cache ★ ข้อคำนวณ hit/miss

Cache index = (Block address) **mod** (#Blocks)
→ **ใช้ บิตล่าง เป็น index**

Address = **[ Tag | Index | Offset ]**

- เช็ก hit: ดู **Index** → เทียบ **Tag** → ดู **Valid (V) = Y**
- เริ่มต้น cache **ว่าง (V = N)**
- **ครั้งแรกของทุก address = Miss แน่นอน**
- ข้อเสีย: หลาย address ชิงช่อง index เดียวกัน

★ **ตย. สไลด์** cache 8 block · Tag = 2 บิตบน · Index = 3 บิตล่าง

| ขอ | 22 | 26 | 22 | 26 | 16 | 3 | 16 | 18 |
|---|---|---|---|---|---|---|---|---|
| Tag<br>Idx | 10<br>110 | 11<br>010 | 10<br>110 | 11<br>010 | 10<br>000 | 00<br>011 | 10<br>000 | 10<br>010 |
| **ผล** | Miss | Miss | **Hit** | **Hit** | Miss | Miss | **Hit** | Miss* |

*18: index 010 มี tag 11 แต่ขอ tag 10 → **tag ไม่ตรง = Miss** แล้วเขียนทับ

| Tag (บิตบน) | | Index (บิตกลาง) | | | Offset (บิตล่าง) | | |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| V=Y tag 10 **ข้อมูล 16** | V=N | V=Y tag 10 **ข้อมูล 18** | V=Y tag 00 **ข้อมูล 3** | V=N | V=N | V=Y tag 10 **ข้อมูล 22** | V=N |

สถานะ cache หลังขอครบตามตัวอย่าง (index 010: 18 เขียนทับ 26)

★ **ตย. บล็อกใหญ่** 64 block × 16 B/block, address **1200**

- Block address = ⌊1200 / 16⌋ = **75**
- Block number = 75 mod 64 = **11**

**Block address = ⌊address / bytes per block⌋**
**Block number = Block address mod #blocks**

## Ch04 คำนวณบิต Tag/Index/Offset เสริม

**Offset = $\log_2$(bytes ต่อ block)**
**Index = $\log_2$(จำนวน block)**
**Tag = บิต address − Index − Offset**

**ตย.** 4 KB direct-mapped, 16 B/block, address 32 บิต

- 4096 / 16 = **256 block**
- Index 8 · Offset 4 · Tag 32 − 8 − 4 = **20**
- ถ้า **2-way**: sets = 128 → Index 7 · Tag **21**

- Hit ratio = hits / accesses
- Miss ratio = 1 − hit ratio
- Block ใหญ่ขึ้น: miss rate ลด (spatial locality)
- แต่ cache เท่าเดิมมี block น้อยลง แย่งกันมากขึ้น miss penalty สูงขึ้น → **ใหญ่ ≠ ดีเสมอ**

★ อาจารย์เน้น · ⚠ ข้อหลอก · เทา = เสริมจากสไลด์

## Ch05 I/O ★ DMA เร็วสุด

- **I/O module** 5 หน้าที่:
  - Control & timing
  - สื่อสารกับ processor
  - สื่อสารกับ device
  - Data buffering (I/O ช้ากว่า memory จึงต้องมี buffer)
  - Error detection
- I/O command 4 แบบ: **Control · Test · Read · Write**

| เทคนิค | หลักการ |
|---|---|
| **Programmed I/O** | ง่ายสุด CPU สั่งแล้ว **นั่งรอ** → เสียเวลา CPU |
| **Interrupt-driven** | CPU สั่งแล้ว**ไปทำงานอื่น** I/O พร้อมค่อย interrupt ต้องรอ **จบ instruction cycle** ก่อนถูกขัด |
| **DMA** | **I/O คุย Memory ตรง** CPU ไม่ยุ่ง<br>**แทรกได้ทุก slot ของ cycle**<br>→ ★ **เร็วที่สุด** ใช้เป็นหลักปัจจุบัน |

## Ch06 Storage · HDD · RAID ★ เน้น HDD

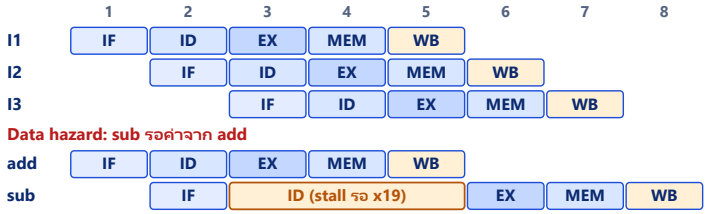
- **เขียน**: กระแสไฟเข้าหัวเขียน → สนามแม่เหล็ก → จัดทิศอนุภาคบนจาน (ซ้าย/ขวา = 0/1)
- **อ่าน**: ขดลวด**เหนี่ยวนำ**สัญญาณ
- **Platter** หลายชั้น · 1 หัวต่อ 1 ผิว
- **Track** = วงแหวน 1 วง
- **Sector** = **ส่วนย่อยของ track** ⚠ **ไม่ใช่กลุ่มของ track**
- ปัจจุบันใช้ **Multi-zone recording** (sector ขนาดใกล้เคียงกัน)
- **SSD** เร็วกว่า ประหยัดไฟกว่า แต่**แพงกว่า** HDD
- **DVD** จุมากกว่า CD (sector เล็กกว่า density สูงกว่า)
- **Tape** ถูกสุด




| RAID | ลักษณะ |
|---|---|
| **0** | **Striping** กระจายเป็นชิ้นเล็กลงหลายดิสก์ (ไม่มีสำรอง) |
| **1** | **Mirror** เขียนซ้ำ 2 ดิสก์ เปลืองแต่**ปลอดภัย** |
| **2** | เก็บ **parity bit / ECC** กู้ข้อมูลได้ |
| **3** | ระดับ **byte** + parity **ดิสก์เดียว** |
| **4** | ระดับ **block** + parity disk แยก → **คอขวด** |
| **5** | block + parity **หมุนเวียน**ทุกดิสก์ (แก้คอขวด RAID 4) |
| **6** | เหมือน 5 แต่ parity **2 ชุด** |

## Ch07 Pipeline · Hazard ★ ถามชนิด/วิธีแก้

- **Pipeline** = แบ่งงานเป็น **stage** ใช้ resource ไม่ชนกัน
- ทำหลายคำสั่งพร้อมกัน → **เร็วขึ้น** แต่เกิด hazard
- เปรียบ: ซักผ้า → อบ → รีด → เก็บ
- Stage: **IF → ID → EX → MEM → WB**
- ลำดับที่อาจารย์ย้ำ: **Fetch → Decode → Execute → Write back**
- ⚠ **ใช้ได้ทั้ง RISC (ARM) และ CISC (Intel)** ไม่ใช่เฉพาะ RISC




## Ch07 Hazard 3 แบบ ★ ถามชนิด/วิธีแก้

| Hazard | เกิดเมื่อ | วิธีแก้ |
|---|---|---|
| **Structural** | **ใช้ resource ชนกัน** ในเวลาเดียวกัน | **Stall** (bubble) จนกว่า resource ว่าง |
| **Data** | **ข้อมูลยังไม่พร้อม**<br>`add x19,x0,x1`<br>`sub x2,x19,x3`<br>(add ยังไม่ write back) | ① **Stall** ง่ายสุด แต่ช้า<br>② **Forwarding / Bypassing** ส่งผล Execute **กลับเป็น input** ไม่รอ write back (เพิ่ม hardware)<br>③ **Code scheduling** compiler เอาคำสั่งไม่เกี่ยวมา**คั่น** (ไม่เพิ่ม hardware) |
| **Control** | คำสั่งก่อนหน้าเป็น **branch/jump** → คำสั่งที่ fetch มา **ไม่ถูกใช้** | ① **Stall / bubble** ง่ายสุด<br>② **Branch prediction** เดาด้วยสถิติ แต่ **hardware แพง** → ส่วนใหญ่ **นิยม stall** |

- ทุก stage ต้องมี **pipeline register** เก็บข้อมูลจาก stage ก่อนหน้า
- ไม่งั้นข้อมูลถูกคำสั่งถัดไป**ทับ** (เพิ่ม hardware = ต้นทุน)

## Ch08 Parallel · Flynn · GPU ★ จับคู่ SISD/SIMD/MIMD

| Flynn | ที่อาจารย์อธิบาย |
|---|---|
| **SISD** | CPU ปกติ **1 คำสั่ง 1 ข้อมูล** |
| **SIMD** | **vector / array · GPU**<br>คำสั่งเดียว ข้อมูลหลายตัวพร้อมกัน (เช่นลบ background ภาพ) |
| **MISD** | แทบไม่มี |
| **MIMD** | **Shared memory = SMP** (PC / Intel)<br>**Distributed memory = cluster** |

- **Multi-core** = duplicate core จริง
- **Multithread** = ซอยงานเป็น thread แล้ว schedule ไม่เพิ่ม hardware
- VM = ซอย hardware ให้ ไม่มี core จริงรองรับ
- **Shared memory**: sync ง่าย แต่ทุก core **แย่งกัน** access
- **Distributed**: sync ข้อมูลยาก
- **GPU** = SIMD · **Fermi** = vector-based · **Tesla** = multiprocessor
- **DGX** ใช้ train AI · ตอนรันโมเดลใช้การ์ดทั่วไป (PCIe) ได้

## คลิป 5 เฉลยโจทย์ที่อาจารย์ถามทบทวน ★ แนวข้อสอบ

| โจทย์ | ตอบ | โจทย์ | ตอบ |
|---|---|---|---|
| DVD vs CD ใครจุมากกว่า | **DVD** sector เล็กกว่า density สูงกว่า | Cache แบบไหน miss น้อยสุด | **Fully associative** |
| Sector คืออะไร | **ส่วนย่อยของ track** ไม่ใช่กลุ่มของ track | วิธีเขียนลง cache | Write-through · Write-back · Write-allocate |
| SISD ทำหลายคำสั่งพร้อมกัน? | **ไม่ได้** 1 คำสั่ง 1 ข้อมูล | Daisy chain เกี่ยวกับ | **Priority** |
| Pipeline ใช้ได้เฉพาะ RISC? | **ผิด** RISC และ CISC มีหมด | ตัวเลือกว่าใครได้ใช้ bus | **Bus Arbitrator** (ไม่ใช่ Control Unit) |
| Exponent single / double | **8 / 11** บิต | I/O เทคนิคไหนเร็วสุด | **DMA** แทรกได้ทุก cycle |
| จุมากสุดในพีระมิด | **Magnetic tape** (offline) | การ์ดจอต่อเมนบอร์ดผ่าน | **PCIe** |
| RAM vs ROM | RAM **ชั่วคราว** · ROM **ถาวร** | SRAM vs DRAM | SRAM **เร็วกว่า** · DRAM ช้า **ถูกกว่า** |