ชีทสรุป สถาปัตยกรรมคอมพิวเตอร์ — Bus · I/O · Interrupt · MIPS-32

198322 · สรุปทฤษฎี สูตร และวิธีคิด (ไม่มีตัวข้อสอบ) · เข้าห้องสอบ 2 แผ่น 4 หน้า

SUMMARY

1 · โครงคอมพิวเตอร์ & ระบบ Bus

4 องค์ประกอบ

Processor (CPU) ประมวลผล + ควบคุมทุกอย่าง · **Memory** เก็บคำสั่งและข้อมูล · **I/O** ติดต่อโลกภายนอก · **Bus** สายสัญญาณร่วมที่เชื่อมทุกอย่างเข้าด้วยกัน

ชนิด bus	ทิศทาง	หน้าที่
Address bus	**ทางเดียว** CPU → Mem/IO	ระบุว่าจะคุยกับ “ช่องไหน” — เจ้าของ bus เป็นคนกำหนดเสมอ
Data bus	**สองทาง**	ขนเนื้อข้อมูล · read = Mem ขับ · write = CPU ขับ
Control bus (read, write, CS, ...)	**ทางเดียว** CPU → Mem/IO	บอกจังหวะและชนิดของ transaction

กฎที่ถามบ่อยที่สุด

read active → Memory / อุปกรณ์ เป็นคน **ขับ (drive)** data bus, CPU เป็นฝ่ายอ่าน
write active → CPU เป็นคนขับ data bus, Memory เป็นฝ่ายรับ
ค่าบน data bus = เนื้อข้อมูลที่เก็บ/จะเก็บ ⇒ **ระบุเป็นตัวเลขไม่ได้** ถ้าโจทย์ไม่บอกเนื้อในหน่วยความจำ
ค่าบน address bus = ระบุได้เสมอ เพราะ CPU เป็นคนใส่เอง

สถานะ read/write	เกิดอะไร
read=1, write=0	Memory ขับ data bus · CPU ปล่อย Hi-Z
read=0, write=1	CPU ขับ data bus · Memory รับไปเก็บ
read=0, write=0	**ไม่มีใครขับ** → data bus เป็น Hi-Z / floating / ค่าไม่นิยาม
read=1, write=1	**ผิดกติกา** → bus conflict ค่าไม่นิยาม อาจเสียหาย

ความจุ & การคำนวณ

สูตรหลัก

จำนวนช่องที่อ้างได้ = 2^(จำนวนบิตของ address bus)
ความจุเป็นไบต์ = 2^(บิต address) × (ขนาด data bus ÷ 8) เมื่อ 1 ช่อง = 1 word
ถ้าระบบเป็น **byte-addressable** (data bus 8 บิต) ⇒ ความจุ = **2^n ไบต์** ตรงๆ
ขนาดของอุปกรณ์ในรูปวงจร = 2^(จำนวนบิตของสาย Addr ที่ต่อเข้ากล่องนั้น)
Bandwidth = ความถี่ × (ขนาด data bus ÷ 8) ไบต์/วินาที
ส่ง N บิตด้วย bus M บิต ต้องใช้ ⌈N/M⌉ รอบ

บิต	ค่า	บิต	ค่า	บิต	ค่า
5	32	16	64 K	28	256 M
6	64	20	1 M	29	512 M
8	256	24	16 M	30	1 G
10	1 K	26	64 M	31	2 G
12	4 K	27	128 M	32	4 G

2 · Tri-state, Bus Contention & Arbitration

ทำไมต้องมี tri-state

bus เป็นสายร่วม ถ้าปล่อยให้ทุกตัวขับพร้อมกันจะชนกัน ⇒ ทุกอุปกรณ์ต้องต่อผ่าน **tri-state buffer** ที่มี 3 สถานะ: **0**, **1**, และ **Hi-Z** (ไม่ขับ ปล่อยลอย) — ตัวที่ไม่ได้สิทธิต้องอยู่ Hi-Z เสมอ

Bus contention / Bus conflict (ออกสอบแน่)

สาเหตุ: driver ≥ 2 ตัวขับเส้นเดียวกันไปคนละลอจิกพร้อมกัน
ผลต่อข้อมูล: แรงดันค้างกลาง → ค่า **undefined** ทุกตัวอ่านได้ข้อมูลผิด
ผลต่อฮาร์ดแวร์: ตัวหนึ่งดัน V_{DD} อีกตัวดึง GND ⇒ **ลัดวงจร** กระแสสูง ร้อน ชิปพังได้
ป้องกัน: arbiter + tri-state (ให้ตัวอื่น Hi-Z) หรือ open-drain + pull-up (กลายเป็น wired-AND ไม่พังแต่ข้อมูลยังผิด)

กฎ “ใครเขียน / ใครอ่าน”

เขียนลง bus = ขับสัญญาณ ⇒ ได้ **ทีละ 1 ตัวเท่านั้น** คือตัวที่ถือสิทธิ (bus master)
อ่านจาก bus = แค่วัดแรงดัน ⇒ **ทุกตัวอ่านได้เสมอ** (bus เป็น broadcast) รวมตัวที่กำลังเขียนเองด้วย
arbiter = คนแจกสิทธิ **ไม่ใช่** คนที่เขียนได้ตลอดเวลา

วิธี arbitration	ลักษณะ
Daisy chain	ส่งสิทธิเป็นทอดๆ · สายน้อย แต่ตัวใกล้ได้เปรียบ เกิด starvation
Centralized / dedicated lines	arbiter มีสายตรงถึงทุกตัว · เร็ว ยุติธรรม แต่สายเยอะ
Distributed / self-selection	อุปกรณ์ตกลงกันเอง ไม่มีศูนย์กลาง
Round-robin	วนสิทธิให้ทั่วถึง แก้ปัญหา starvation

ข้อดี / ข้อเสียของ bus ร่วม

ดี: สายน้อย ราคาถูก ต่ออุปกรณ์เพิ่มง่าย (ขยายได้)
เสีย: ใช้ได้ทีละตัว ⇒ เป็น **bottleneck** · ต้องมี arbiter · ความเร็วถูกจำกัดโดยอุปกรณ์ที่ช้าที่สุด · สายยาว = สัญญาณรบกวน

3 · Chip Select, Decoder และการถอด Address Map จากรูป

สูตรถอดวงจร 4 ขั้น — ใช้ได้กับทุกรูปที่โจทย์ให้

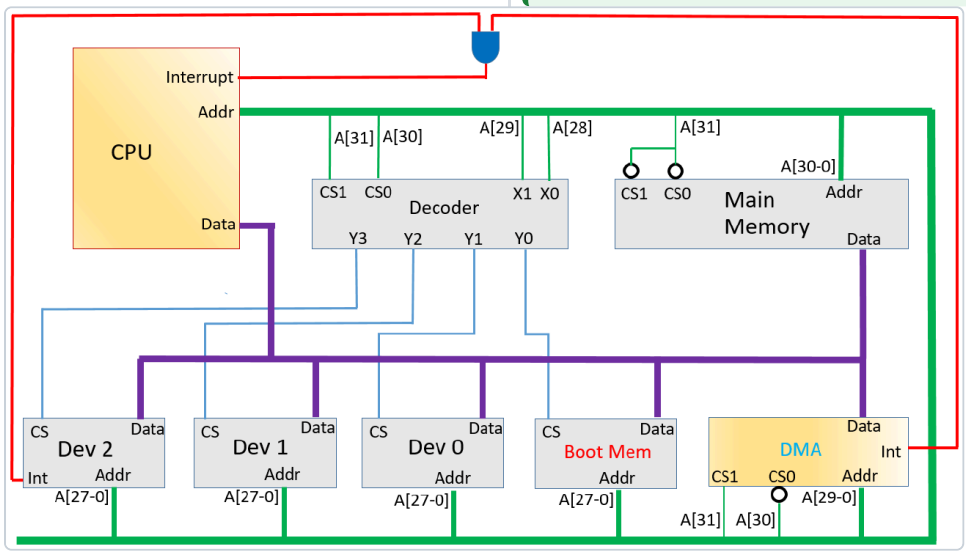
ขั้น 1 ไล่ดูทุกกล่อง ว่า **ขา CS ของมันรับสายอะไร** และมี **วงกลม ○** คั่นไหม
มี ○ = active LOW ⇒ สายนั้นต้อง = **0** ถึงจะเลือก · **ไม่มี ○ = active HIGH** ⇒ ต้อง = **1**
มีหลายขา CS ⇒ ต้อง active **ครบทุกขา** (AND กัน)
ขั้น 2 Decoder ทำงานเมื่อ CS ครบ แล้วดู `X1 X0` รับบิตไหน → เลือก Y ตามค่าฐานสอง (00→Y0, 01→Y1, 10→Y2, 11→Y3)
ขั้น 3 ไล่ดูว่า Y ตัวไหนวิ่งไปเข้า CS ของกล่องไหน (ในรูปสายมักไขว้กัน — ไล่ทีละเส้น)
ขั้น 4 ดูว่ากล่องรับ `A[k-0]` กี่บิต ⇒ ขนาดพื้นที่ = **2^(k+1) ไบต์** · ส่วนบิตบนที่เหลือคือ “รหัสประจำตัว” ของกล่องนั้น

★ เทคนิคเร็ว: อ่าน hex หลักแรก

hex 1 หลัก = 4 บิต = A[31:28] ⇒ ถ้าเงื่อนไขเลือกอุปกรณ์ใช้แค่ A[31]...A[28] ก็ **ดู hex ตัวแรกตัวเดียวจบ**
A[31] คือบิตซ้ายสุด: **0–7 ⇒ A[31]=0** · **8–F ⇒ A[31]=1**
A[30]: 0-3 และ 8-B ⇒ 0 | 4-7 และ C-F ⇒ 1

ตัวอย่างวงจรอ้างอิง — ให้ฝึกไล่สาย

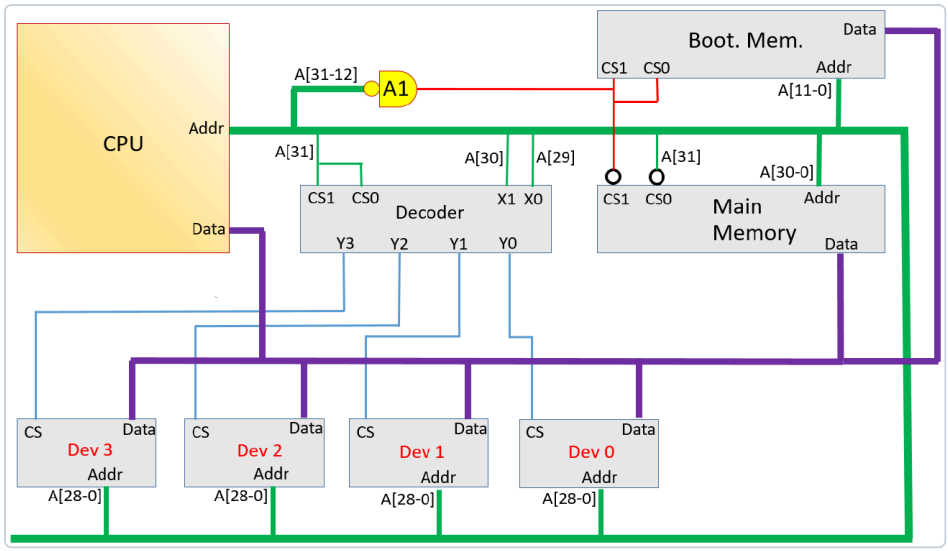
วงจร A: Main Mem CS ○○ ← A[31] · DMA CS1←A[31], CS0 ○←A[30] · Decoder CS←A[31],A[30], X1X0←A[29],A[28]
⇒ **0–7 Main · 8–B DMA · C Boot · D Dev0 · E Dev1 · F Dev2** (แต่ละตัว 256 MB, A[27-0])



วงจรอ้างอิง A — Main Memory volatile · Boot Mem non-volatile · เส้นแดง = interrupt · เส้นฟ้าบาง = chip-select จาก decoder

วงจรอ้างอิง B (COE 2019)

Gate A1 ให้ Y = $\bar{A}_{12}\cdot\bar{A}_{13}\cdot\ldots\cdot\bar{A}_{31}$ ⇒ เป็น 1 เมื่อ A[31:12] เป็น 0 ทั้งหมด ⇒ Boot Mem = **4 KB แรกเท่านั้น**
Main Mem CS ○←Y_{A1}, ○←A[31] ⇒ ต้อง Y=0 และ A[31]=0 ⇒ 0x0000 1000–0x7FFF FFFF
Decoder CS1=CS0=A[31], X1X0 = A[30],A[29] ⇒ **8,9 Dev0 · A,B Dev1 · C,D Dev2 · E,F Dev3** ตัวละ 512 MB (A[28-0])



วงจรอ้างอิง B (COE 2019) — Big Endian · Bootstrapping Address = 0x00000000 · Gate A1 รับ 20 บิต ออก 1 บิต

Memory-mapped I/O

อุปกรณ์ I/O ถูกจัดสรร **ช่วง address ปนอยู่กับหน่วยความจำ** ⇒ CPU ติดต่อ I/O ด้วย `lw` / `sw` ธรรมดา ไม่มีคำสั่งพิเศษ
วิธีตอบว่า "คำสั่งไหนติดต่อ I/O": หา `lw` / `sw` ทุกตัว → คำนวณ address จริง (ค่าใน base register + offset) → เอาไปเทียบ address map
ไม่นับเป็น I/O: คำสั่งคำนวณ (`add`, `lui`, `ori`, `slt`, `j`, `beq`) และการ **fetch คำสั่ง** เอง
เทียบกับ Isolated I/O: แยกพื้นที่ address ของ I/O ออกจาก memory ต้องใช้คำสั่งเฉพาะ (`in` / `out`) และสายควบคุมเพิ่ม

4 · Interrupt · Polling · DMA

อุปกรณ์ตัวไหน interrupt CPU ได้

ตอบจาก **สายในรูปเท่านั้น**: ตัวที่มีขา `Int` และมีเส้นวิ่งเข้า OR-gate ที่ต่อกับขา `Interrupt` ของ CPU · ตัวที่ไม่มีขา Int (Decoder, Memory, Boot Mem, ...) **ทำไม่ได้**

ลำดับที่ CPU ทำเมื่อรับ interrupt

① ทำคำสั่งปัจจุบันให้จบ ② เช็คว่าเปิดรับ interrupt อยู่ไหม (ถ้า masked = ไม่สนใจ) ③ **เก็บ PC** และ status/registers ④ กระโดดไปที่ **interrupt vector** ⑤ ทำ ISR ⑥ เคลียร์ flag ที่อุปกรณ์ ⑦ คืน register แล้ว **กลับที่ PC ที่เก็บไว้**

โจทย์แบบ "ค้างที่ address X แล้วได้ interrupt vector V"

(a) คำสั่งถัดไป = **V** · **(b)** ISR อยู่ที่ = **V** · **(c)** อยู่หน่วยความจำไหน = เอา **V** ไปเทียบ address map
(d) จบ ISR กลับไปที่ = **X** ถ้าโจทย์บอก "ก่อนจะโหลดคำสั่งที่ X" · = **X + 4** ถ้าโจทย์บอก "ทำคำสั่งที่ X เสร็จแล้ว"

ถ้าไม่มี interrupt → ใช้ Polling

CPU **วนลูปอ่าน status register** ของอุปกรณ์ซ้ำๆ ผ่าน memory-mapped I/O แล้วตรวจบิต ready/done

```
poll: lw   $t0, 0($s0)   # อ่าน status
      andi $t0, $t0, 1   # แยกบิต DONE
      beq  $t0, $zero, poll
      lw   $t1, 4($s0)   # เสร็จ → อ่านข้อมูล
```

เสีย: busy-waiting กิน CPU · latency ขึ้นกับคาบ poll · ไม่ scale
ดี: ฮาร์ดแวร์ง่าย ไม่ต้องมี ISR/vector · เวลาคาดเดาได้

	Polling	Interrupt	DMA
ใครเริ่ม	CPU ถามเอง	อุปกรณ์บอก CPU	อุปกรณ์ย้ายเอง
CPU เสียเวลา	มาก	น้อย	น้อยมาก
ฮาร์ดแวร์	ง่ายสุด	สาย Int + ISR	ต้องเป็น bus master ได้
เหมาะกับ	อุปกรณ์เร็ว/น้อย	เหตุการณ์ไม่แน่นอน	ย้ายข้อมูลก้อนใหญ่

DMA

ย้ายข้อมูลระหว่าง I/O กับหน่วยความจำ **โดยไม่ผ่าน CPU** (ตัวมันเองเป็น bus master ชั่วคราว) แล้วส่ง interrupt บอก CPU เมื่อย้ายเสร็จ ⇒ CPU เอาเวลาไปทำอย่างอื่นได้

Volatile vs Non-volatile

Main Memory (RAM) = volatile ข้อมูลหายเมื่อดับไฟ · **Boot/Bootstrapping Memory (ROM)** = non-volatile ข้อมูลอยู่ต่อ ⇒ จึงเป็นที่เก็บโปรแกรมชุดแรกตอนเปิดเครื่อง

5 · Bootstrapping & Endianness

Bootstrapping — ขั้นตอนถอดโปรแกรม boot

① เขียน **address map** จากรูปให้เสร็จก่อน ② จด **Bootstrapping Address** (คำสั่งแรกอยู่ที่นั่น) ③ ตัดข้อมูลเป็นก้อนละ **4 ไบต์** ตาม endian ④ แปลง hex → 32 บิต ⑤ ดู **op 6 บิตแรก** → รู้ format → กางฟิลด์ ⑥ ไล่ค่า register ทีละคำสั่ง ⑦ ทุก `lw`/`sw` ให้คำนวณ address แล้วเทียบ map ว่าไปโดนอุปกรณ์ไหน

Endianness

Big Endian — ไบต์ที่ address **ต่ำสุด** = ไบต์ที่ **มีนัยสำคัญสูงสุด** (ซ้ายสุดของ word)
ไบต์ `3C 15 A0 00` ⇒ word = **0x3C15A000**
Little Endian — กลับกัน ไบต์ที่ address ต่ำสุด = ไบต์ขวาสุด
ไบต์ `3C 15 A0 00` ⇒ word = **0x00A0153C**
เช็คเร็ว: ถอดแล้ว op ต้องเป็นค่าที่มีอยู่จริง (lui/ori/lw/sw/j/...) ถ้าถอดแล้วได้คำสั่งประหลาด แปลว่าอาจสลับ endian ผิด

ลำดับการรันหลังเปิดเครื่อง

เปิดไฟ → PC = Bootstrapping Address → fetch จาก **Boot Mem (non-volatile)** → boot program ตั้งค่าอุปกรณ์ / ทดสอบ / โหลดโปรแกรมหลัก → `j` หรือ `jr` ไปยังโปรแกรมหลักใน **Main Memory**

6 · MIPS-32 — Register & รูปแบบคำสั่ง

Register	เลข	bin	ใช้ทำอะไร
`$zero`	0	00000	ค่าคงที่ 0 เขียนทับไม่ได้
`$at`	1	00001	assembler สงวนไว้
`$v0`-`$v1`	2–3	00010–00011	ค่าส่งกลับจากฟังก์ชัน
`$a0`-`$a3`	4–7	00100–00111	อาร์กิวเมนต์
`$t0`-`$t7`	8–15	01000–01111	ตัวแปรชั่วคราว
`$s0`-`$s7`	16–23	10000–10111	ตัวแปรถาวร (saved)
`$t8`-`$t9`	24–25	11000–11001	ชั่วคราวเพิ่ม
`$k0`-`$k1`	26–27	11010–11011	สงวนให้ OS/kernel
`$gp`/`$sp`/`$fp`/`$ra`	28–31	11100–11111	global / stack / frame / return addr

★ ท่องเลข register

`$t0` =8 · `$s0` =**16** ⇒ `$sN` = **16 + N** · `$tN` = **8 + N** (N ≤ 7) · `$ra` =31

op 31–26	rs 25–21	rt 20–16	rd 15–11	shamt 10–6	funct 5–0

R — `add rd, rs, rt` · op = 000000 เสมอ · แยกด้วย **funct**

op 31–26	rs 25–21	rt 20–16	immediate / offset 15–0

I — `lw rt,imm(rs)` · `addi rt,rs,imm` · `beq rs,rt,imm`

op 31–26	target address 25–0

J — `j target` · `jal target`

⚠ ลำดับฟิลด์สลับกับที่เขียน

`add` **`rd`**`, rs, rt` แต่ในเครื่องเรียง **rs · rt · rd**
`lw` **`rt`**`, off(rs)` แต่ในเครื่องเรียง **rs · rt** ⇒ register ในวงเล็บมาก่อนเสมอ
`beq rs, rt, L` เรียงตามที่เขียนพอดี

R-type	funct
add / addu	0x20/0x21
sub / subu	0x22/0x23
and / or	0x24/0x25
xor / nor	0x26/0x27
slt / sltu	0x2A/0x2B
sll / srl / sra	0x00/0x02/0x03
sllv / srlv	0x04/0x06
mult / div	0x18/0x1A
mfhi / mflo	0x10/0x12
jr / jalr	0x08/0x09

I / J	op
lw / sw	0x23/0x2B
lb / lbu	0x20/0x24
lh / sh / sb	0x21/0x29/0x28
addi / addiu	0x08/0x09
andi / ori	0x0C/0x0D
xori / lui	0x0E/0x0F
slti / sltiu	0x0A/0x0B
beq / bne	0x04/0x05
j / jal	0x02/0x03

★ op ที่ต้องจำให้ได้ 6 ตัว

`lw` **100011** · `sw` **101011** · `addi` **001000** · `beq` **000100** · `j` **000010** · R-type **000000**

10 · ลอจิกพื้นฐานที่ต้องใช้อ่านรูปวงจร

A	B	AND	OR	XOR	NAND	NOR
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	1	0
1	1	1	1	0	0	0

X1	X0	Decoder 2→4 (เมื่อ CS ครบ)
0	0	**Y0 = 1**, Y1 = Y2 = Y3 = 0
0	1	**Y1 = 1**, อื่นๆ = 0
1	0	**Y2 = 1**, อื่นๆ = 0
1	1	**Y3 = 1**, อื่นๆ = 0
CS ไม่ครบ		**Y0=Y1=Y2=Y3=0 ทั้งหมด** (ไม่สนใจ X1 X0)

Active HIGH vs Active LOW — จุดที่พลาดกันมากที่สุด

ไม่มีวงกลม ○ ที่ขา ⇒ active **HIGH** ⇒ สายที่ต่อเข้ามาต้องเป็น **1** ถึงจะ "ทำงาน"
มีวงกลม ○ ที่ขา ⇒ active **LOW** ⇒ สายที่ต่อเข้ามาต้องเป็น **0** ถึงจะ "ทำงาน"
กล่องที่มี CS หลายขา ⇒ ต้อง active **ครบทุกขาพร้อมกัน** (AND) ถึงจะถูกเลือก

รูปแบบที่เจอบ่อย	เงื่อนไขถูกเลือก
CS1 ← A[31], CS0 ← A[30] (ไม่มี ○)	A[31]=1 **และ** A[30]=1
CS1 ○← A[31], CS0 ○← A[31]	A[31]=0
CS1 ← A[31], CS0 ○← A[30]	A[31]=1 **และ** A[30]=0
CS1=CS0 ← Y ของ gate	เงื่อนไขของ gate เป็นจริง

Gate แบบ "AND ของบิตที่กลับค่า"

$Y = \bar{A}_k \cdot \bar{A}_{k+1} \cdot \ldots \cdot \bar{A}_{31}$ ⇒ Y = 1 **ก็ต่อเมื่อบิต A[31:k] เป็น 0 ทั้งหมด**
⇒ ครอบคลุม address **0 ถึง 2^k − 1** ⇒ ขนาด = **2^k ไบต์**
ตัวอย่าง: รับ A[31-12] ⇒ 2^{12} = **4 KB** · รับ A[31-16] ⇒ 2^{16} = **64 KB** · รับ A[31-10] ⇒ 1 KB

แม่แบบเปล่า — ใช้ร่างในห้องสอบก่อนตอบเสมอ

กล่อง	CS มาจาก (○?)	เงื่อนไขบิตบน	ช่วง address	สาย Addr	ขนาด

11 · เลขฐานที่ต้องคล่อง

hex	bin	dec	hex	bin	dec
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

hex หลักแรก	ช่วง 256 MB
0	0x0000 0000–0x0FFF FFFF
1	0x1000 0000–0x1FFF FFFF
...	ทุกหลักกินพื้นที่ **256 MB**
8	0x8000 0000–0x8FFF FFFF
C	0xC000 0000–0xCFFF FFFF
F	0xF000 0000–0xFFFF FFFF
0–7	= A[31] เป็น 0 (ครึ่งล่าง 2 GB)
8–F	= A[31] เป็น 1 (ครึ่งบน 2 GB)

2's complement (จำนวนเต็มมีเครื่องหมาย)

วิธี: **กลับบิตทุกตัวแล้ว +1** · ลัด: **0x10000 − |n|** (16 บิต) หรือ **0x100000000 − |n|** (32 บิต)
−1 = 0xFFFF · −2 = 0xFFFE · −3 = 0xFFFD · −4 = **0xFFFC** · −5 = 0xFFFB · −8 = 0xFFF8 · −16 = 0xFFF0 · −32 = 0xFFE0 · −64 = 0xFFC0 · −128 = 0xFF80 · −256 = 0xFF00
บิตซ้ายสุดเป็น 1 ⇒ ค่าติดลบ · ช่วง 16 บิต = **−32768 ... +32767**
Sign extension: ขยาย 16 → 32 บิต ให้ก๊อปบิตซ้ายสุดไปเติมข้างหน้า (0xFFFC → 0xFFFFFFFC = −4)

★ คูณ/หาร 4 ในเลขฐานสิบหก (ใช้ตอนคิด j และ branch)

÷ 4 = เลื่อนขวา 2 บิต · × 4 = เลื่อนซ้าย 2 บิต
วิธีทำมือ: hex → เขียนเป็น binary → เลื่อน → จับกลุ่มละ 4 บิตกลับเป็น hex
ตัวอย่าง 0x0040 0054 ÷ 4 → `0000 0000 0100 0000 0000 0000 0101 0100` เลื่อนขวา 2 → `0000 0000 0001 0000 0000 0000 0001 0101` = **0x0010 0015**
ทางลัด: address ที่ลงท้าย `0,4,8,C` เท่านั้นที่หาร 4 ลงตัว — ถ้าโจทย์ให้ address ที่ลงท้ายด้วยเลขอื่น แสดงว่าอ่านผิด

7 · เข้ารหัส ↔ ถอดรหัส Machine Code

สูตรที่ต้องท่อง

Branch (beq/bne)
imm = (addr(label) − addr(branch) − 4) ÷ 4 → เก็บ 16 บิต
signed
ปลายทาง = addr(branch) + 4 + imm × 4
Jump (j/jal)
field = addr(target) ÷ 4 → เก็บ **26 บิตล่าง**
ปลายทาง = (PC+4)[31:28] || field(26) || **00**
2's complement 16 บิต = 0x10000 − |n| ⇒ −1=0xFFFF, −4=0xFFFC, −5=0xFFFB, −8=0xFFF8

ขั้นตอนเข้ารหัส (assembly → hex) 6 ขั้น

① ดูว่าเป็น R / I / J ② เปิดตาราง op (และ funct ถ้าเป็น R) ③ แปลงชื่อ register เป็นเลข 5 บิต ④ ถ้าเป็น branch/jump ให้คำนวณ offset/field ก่อน ⑤ เรียงบิตให้ครบ 32 ⑥ จับกลุ่มละ **4 บิต** แปลงเป็น hex 8 หลัก

ขั้นตอนถอดรหัส (hex → assembly) 5 ขั้น

① hex 8 หลัก → 32 บิต ② อ่าน **6 บิตแรก**: **000000 = R** (ไปดู funct 6 บิตท้าย) · **000010/000011 = J** · อื่นๆ = **I** ③ กางฟิลด์ตามรูปแบบ ④ แปลงเลข register กลับเป็นชื่อ ⑤ ถ้าเป็น branch/jump คำนวณ address ปลายทางกลับ

★ ประเด็นที่ชอบออก: ย้ายโปรแกรมไปคนละที่

`beq` / `bne` ไม่เปลี่ยน เพราะใช้ offset **สัมพัทธ์** ⇒ โค้ด **relocatable**
`j` / `jal` เปลี่ยน เพราะใช้ address **สัมบูรณ์**
`j` ไปได้ไกล 2^{28} = **256 MB** ภายใน region เดียวกัน (บิต 31–28 มาจาก PC+4) ⇒ ข้าม region ต้องใช้ `jr` (ได้ทั้ง 4 GB)
`beq` ไปได้ **−32768...+32767 คำสั่ง** (≈ ±128 KB)
ทำไมหาร 4: คำสั่ง MIPS ยาว 4 ไบต์เสมอ address จึงลงท้ายด้วย 00 ตลอด ⇒ ไม่ต้องเก็บ 2 บิตนั้น

ตัวอย่างถอด — 0x02328020

`000000·10001·10010·10000·00000·100000` ⇒ op=R, funct=0x20= add , rs=17=$s1, rt=18=$s2, rd=16=$s0 ⇒ **`add $s0,$s1,$s2`**

ตัวอย่างเข้ารหัส — `lw $t1, 8($s2)`

op=100011 · rs=$s2=18=10010 · rt=$t1=9=01001 · imm=0x0008 ⇒ **0x8E49 0008**

8 · แปลง C → MIPS Assembly

หลักคิด 3 ข้อ

① **if / while ให้ตรวจ "นิเสธ" ของเงื่อนไข แล้ว branch ข้ามออกไป**
② MIPS มีแค่ `beq`, `bne`, `slt` ⇒ เงื่อนไข <, ≤, >, ≥ ต้องประกอบเอง
③ array `A[k]` ⇒ offset = **k × sizeof** (int=4, char=1, double=8) · index เป็นตัวแปรต้อง `sll` ก่อน

C	โครง Assembly
`if(c){X}`	ตรวจนิเสธ c → `b.. END` / `X` / `END:`
`if(c){X}else{Y}`	นิเสธ→ `ELSE` / `X` / `j END` / `ELSE:` `Y` / `END:`
`while(c){X}`	`L:` นิเสธ→ `END` / `X` / `j L` / `END:`
`do{X}while(c)`	`L:` `X` / ตรวจ c → `b.. L`
`for(i;c;u){X}`	`i` / `L:` นิเสธ→ `END` / `X` / `u` / `j L` / `END:`

เงื่อนไข C	เขียนด้วย slt (ให้กระโดดเมื่อ "จริง" ไป L)
`a < b`	`slt $t0,$s0,$s1 ; bne $t0,$zero,L`
`a > b`	`slt $t0,$s1,$s0 ; bne $t0,$zero,L`
`a <= b`	`slt $t0,$s1,$s0 ; beq $t0,$zero,L`
`a >= b`	`slt $t0,$s0,$s1 ; beq $t0,$zero,L`
`a == b` / `a != b`	`beq` / `bne $s0,$s1,L`

ตารางแปลง pseudo → คำสั่งจริง (ห้ามใช้ pseudo ในข้อสอบ)

Pseudo	ใช้อะไรแทน
`li $s0,N`	`addi $s0,$zero,N` (16 บิต) · ใหญ่กว่านั้นใช้ `lui` + `ori`
`move $s0,$s1`	`add $s0,$s1,$zero`
`la $s0,LBL`	`lui` + `ori`
`blt/bgt/ble/bge`	`slt` + `bne` / `beq` (ดูตารางด้านบน)
`beqz/bnez $x,L`	`beq` / `bne $x,$zero,L`
`not $s0,$s1`	`nor $s0,$s1,$zero`
`nop`	`sll $zero,$zero,0`
`mul $s0,$s1,$s2`	`mult $s1,$s2 ; mflo $s0`

สำนวนที่ใช้บ่อย

```
a = A[k];  →  lw $s0, k*4($s3)
A[k] = a;  →  sw $s0, k*4($s3)
a = A[i];  →  sll $t0,$s2,2 ; add $t0,$t0,$s3 ; lw $s0,0($t0)
a = a * 8;  →  sll $s0,$s0,3   ·   a = a / 4;  (a≥0) →  srl $s0,$s0,2
a = 0;  →  add $s0,$zero,$zero   ·   a++;  →  addi $s0,$s0,1
สลับ a,b →  add $t0,$s0,$zero ; add $s0,$s1,$zero ; add $s1,$t0,$zero
```

```
# เรียกฟังก์ชัน
      jal  FUNC        # $ra = PC+4 แล้วกระโดด
FUNC: addi $sp,$sp,-4
      sw   $ra,0($sp)  # เก็บที่กลับ
      ...
      lw   $ra,0($sp)
      addi $sp,$sp,4
      jr   $ra         # กลับ
```

9 · สูตรรวม & คำศัพท์ & กับดัก

Σ สูตรทั้งหมด

ความจุ = 2^(บิต address) × ไบต์ต่อช่อง | ขนาดกล่องในรูป = 2^(บิตของสาย Addr)
Branch imm = (addr(label) − addr(branch) − 4) ÷ 4 | ปลายทาง = addr(branch)+4+imm×4
Jump field = addr(target) ÷ 4 | ปลายทาง = (PC+4)[31:28] || field || 00
offset ของ A[k] = k × sizeof | 2's comp 16 บิต = 0x10000 − |n|
Bandwidth = ความถี่ × (data bus ÷ 8) | รอบส่งข้อมูล = ⌈N/M⌉

คำ	ความหมาย
Volatile / Non-volatile	ข้อมูลหาย / ไม่หายเมื่อดับไฟ
Tri-state / Hi-Z	เอาต์พุต 3 สถานะ (0,1,ลอย) / สถานะไม่ขับสัญญาณ
Chip Select (CS)	ขาที่บอกว่าชิปถูกเลือก · ○ = active-low
Decoder	แปลง n บิต → เลือกเอาต์พุต 1 ใน 2^n เส้น
Bus master / Arbiter	ตัวที่ควบคุม bus / ตัวตัดสินว่าใครได้เป็น master
Bus contention	หลายตัวขับ bus พร้อมกัน → ค่าผิด + ลัดวงจร
Memory-mapped I/O	I/O ใช้ address ปนกับ memory ⇒ `lw`/`sw` ธรรมดา
ISR / Interrupt vector	โปรแกรมตอบ interrupt / address ที่ CPU กระโดดไป
Polling	CPU วนถาม status เอง (busy-wait)
DMA	ย้ายข้อมูลเองโดยไม่ผ่าน CPU
Bootstrapping	โปรแกรมชุดแรกหลังเปิดเครื่อง อยู่ใน non-volatile
Relocatable	ย้ายโค้ดไปที่อื่นแล้วยังทำงานได้ (เพราะใช้ branch สัมพัทธ์)

⚠ 12 กับดักที่คนพลาดบ่อย

① ตอบว่า data bus มีค่าเท่าไร ทั้งที่โจทย์ไม่ได้ให้เนื้อในหน่วยความจำ ⇒ ต้องตอบ "ระบุไม่ได้"
② คิดว่า arbiter เขียน bus ได้ตลอด ③ คิดว่าอ่าน bus ต้องขอสิทธิ ⇒ อ่านได้ทุกตัว
④ ตอบ interrupt ช่อง (d) เป็น address+4 ทั้งที่โจทย์บอกว่า "ก่อนจะโหลด" ⇒ กลับที่เดิม
⑤ ลืมว่า `lw`/`sw` เรียง rs ก่อน rt ⑥ ลืมหาร 4 / ลืม PC+4 ในการคิด branch
⑦ ลืมหาร 4 ใน `j` ⑧ ใช้ pseudo-instruction (`li`, `move`, `blt`) ทั้งที่ห้าม
⑨ อ่าน Big Endian กลับด้าน ⑩ ลืมว่าขนาดของ Boot Mem ถูกกำหนดโดย gate ไม่ใช่ทั้ง region
⑪ จำ address map ของวงจรคนละรูปสลับกัน ⑫ ลืมคูณ index ด้วย 4 สำหรับ array int

★ ก่อนส่งกระดาษ เช็ค 5 อย่าง

① ตอบครบทุกช่องย่อย (a)(b)(c)(d) ② ไม่มี pseudo-instruction ③ หาร 4 ครบทุก branch/jump ④ hex เขียนครบ 8 หลัก ⑤ ข้อที่ให้ "อธิบาย" เขียนเหตุผล ไม่ใช่ตอบคำเดียว

12 · ความหมายของแต่ละคำสั่ง MIPS

คำสั่ง	ชนิด	ทำอะไร
`add rd,rs,rt`	R	rd = rs + rt (ตรวจ overflow)
`sub rd,rs,rt`	R	rd = rs − rt
`and/or/xor/nor`	R	ดำเนินการบิตต่อบิต
`slt rd,rs,rt`	R	rd = (rs < rt) ? 1 : 0
`sll rd,rt,sh`	R	rd = rt เลื่อนซ้าย sh บิต (= × 2^sh)
`srl rd,rt,sh`	R	เลื่อนขวาเติม 0 (= ÷ 2^sh เมื่อไม่ติดลบ)
`sra rd,rt,sh`	R	เลื่อนขวาคงเครื่องหมาย
`jr rs`	R	PC = rs (กระโดดได้ทั้ง 4 GB)
`addi rt,rs,imm`	I	rt = rs + sign-extend(imm)
`andi/ori rt,rs,imm`	I	ใช้ zero-extend(imm) ไม่ใช่ sign-extend
`lui rt,imm`	I	rt = imm || 0x0000 (ยกไปครึ่งบน)
`lw rt,off(rs)`	I	rt = Mem[rs + off] (4 ไบต์)
`sw rt,off(rs)`	I	Mem[rs + off] = rt
`beq rs,rt,L`	I	ถ้า rs == rt → PC = PC+4+imm×4
`bne rs,rt,L`	I	ถ้า rs != rt → กระโดด
`slti rt,rs,imm`	I	rt = (rs < imm) ? 1 : 0
`j target`	J	PC = (PC+4)[31:28] || field || 00
`jal target`	J	$ra = PC+4 แล้วกระโดด

★ ทางลัดคิด hex ของ I-type

32 บิต = op(6) || rs(5) || rt(5) || imm(16)
⇒ **hex 4 หลักแรก** มาจาก op·rs·rt (16 บิตบน) และ **hex 4 หลักหลัง = imm ตรงๆ**
เช่น `lw $s0,__($s1)` : 100011·10001·10000 = `1000 1110 0011 0000` = **0x8E30** ⇒ คำสั่ง = **0x8E30**||imm
`sw $s0,__($s1)` = **0xAE30**||imm · `addi $s1,$s1,__` = **0x2231**||imm
⇒ ถ้าโจทย์ให้คำสั่งเดิมแต่เปลี่ยน offset ก็แค่เปลี่ยน 4 หลักหลัง

วงจรการทำงานของคำสั่ง (Instruction Cycle)

① **Fetch** — อ่านคำสั่งจากหน่วยความจำที่ address = PC (ใช้ address bus + read) ② **Decode** — ถอด op/ฟิลด์ อ่านค่า register ③ **Execute** — ALU คำนวณ (บวก offset, เทียบค่า) ④ **Memory** — เฉพาะ `lw` / `sw` ที่แตะ data bus จริง ⑤ **Write back** — เขียนผลกลับ register
PC: ปกติ PC ← PC + 4 ทุกคำสั่ง · ถ้าเป็น branch/jump ที่เป็นจริง ถึงจะเปลี่ยนเป็นค่าอื่น

13 · Interrupt เชิงลึก (เผื่อถามต่อยอด)

คำ	ความหมาย
Maskable	ปิดรับได้ด้วยซอฟต์แวร์ (interrupt enable bit)
Non-maskable (NMI)	ปิดไม่ได้ ใช้กับเหตุร้ายแรง เช่น ไฟตก
Vectored interrupt	อุปกรณ์แต่ละตัวมี vector ของตัวเอง ⇒ CPU รู้ทันทีว่าใครเรียก
Polled interrupt	สาย Int รวมกัน (OR) ⇒ ISR ต้องไล่ถามทีละตัวว่าใครเรียก
Priority	ถ้าหลายตัวเรียกพร้อมกัน ใช้ priority encoder ตัดสิน
Nested interrupt	ยอมให้ interrupt ที่สำคัญกว่าขัดจังหวะ ISR ที่กำลังทำอยู่
Exception / Trap	เหตุผิดปกติจากภายในซีพียูเอง (หารศูนย์, overflow, คำสั่งไม่รู้จัก)
Context switch	เก็บ/คืนสถานะ register + PC เพื่อสลับงาน

ทำไมสาย Int ถึงรวมผ่าน OR-gate

CPU มีขา `Interrupt` ขาเดียว ⇒ ต้อง OR สัญญาณจากทุกอุปกรณ์เข้าด้วยกัน
ผลคือ CPU รู้แค่ว่า "**มีใครสักคน** ขอ interrupt" แต่ไม่รู้ว่าใคร ⇒ ISR ต้อง **ไล่ poll status ของแต่ละอุปกรณ์** เพื่อหาตัวที่เรียก
ถ้าอยากรู้ทันทีว่าใครเรียก ต้องแยกสาย Int + ใช้ vectored interrupt / priority encoder

⚠ อ่านโจทย์ให้ขาด

"**ก่อนจะโหลด**คำสั่งที่ X" ⇒ คำสั่งนั้น **ยังไม่ถูกทำ** ⇒ จบ ISR กลับมาที่ **X**
"**ทำคำสั่งที่ X เสร็จแล้ว**" ⇒ จบ ISR กลับมาที่ **X + 4**
"CPU **ปิดรับ** interrupt" ⇒ ไม่กระโดดเลย ทำคำสั่งที่ X ต่อตามปกติ

14 · Recipe สรุป — เจอโจทย์แบบไหนทำยังไง

โจทย์ถามว่า	ทำตามนี้
ทิศทางของสายสัญญาณ	address/read/write = CPU→Mem ทางเดียว · data = สองทาง
ค่าบน data bus	ดู read/write ว่าใครขับ แล้วบอกว่า “ค่าระบุไม่ได้ ถ้าไม่รู้เนื้อในหน่วยความจำ”
อ้างหน่วยความจำได้กี่ไบต์	2^(บิต address) × (data bus ÷ 8) — ถ้า data 8 บิต ก็ 2^n ไบต์
ใครเขียน/อ่าน bus ได้	เขียน = ตัวถือสิทธิตัวเดียว · อ่าน = ทุกตัว
เขียนพร้อมกันเกิดอะไร	bus contention → ค่า undefined + ลัดวงจร + ป้องกันด้วย tri-state/arbiter
ใคร interrupt ได้	ไล่สายจากขา Int → OR-gate → ขา Interrupt ของ CPU
ไม่มี interrupt ทำยังไง	Polling — วนอ่าน status register + บอกข้อดีข้อเสีย
interrupt แล้วไปไหน/กลับไหน	(a)(b) = vector · (c) = เทียบ address map · (d) = address เดิม
address นี้คืออุปกรณ์อะไร	ถอดวงจร → ทำ address map → เทียบ hex หลักแรก
เขียน assembly แทน C	ตรวจนิเสธของเงื่อนไข → branch ข้าม · array คูณ 4 · ห้าม pseudo
assembly → machine code	เลือก format → เปิด op/funct → register 5 บิต → branch ÷4 · jump ÷4 → รวม 32 บิต → hex
machine code → assembly	กาง 32 บิต → ดู op 6 บิตแรก → กางฟิลด์ → แปลง register กลับ
คำสั่งไหนติดต่อ I/O	หา lw/sw ทุกตัว → คำนวณ address จริง → เทียบ map
โปรแกรม boot ทำอะไร	ไล่ทีละคำสั่ง จด register ทุกตัว แล้วสรุปเป็นภาษาคน

★ ก่อนส่งกระดาษ เช็ค 6 อย่าง
① ตอบครบทุกช่องย่อย (a)(b)(c)(d) ② ไม่มี pseudo-instruction ③ หาร 4 ครบทุก branch/jump ④ hex ครบ 8 หลัก ⑤ ข้อที่ให้ “อธิบาย” ต้องมีเหตุผล ไม่ตอบคำเดียว ⑥ ตรวจว่าใช้ address map ของรูปที่ถูกต้อง

17 · ตารางสำเร็จรูป — offset ของ branch

นิยาม
ให้ **m** = จำนวนคำสั่งนับจาก branch **ลงไป** ถึง label (label ที่อยู่ถัดจาก branch ทันที ⇒ m = 1)
ให้ **k** = จำนวนคำสั่งนับจาก branch **ขึ้นไป** ถึง label

label อยู่ข้างหน้า	imm	hex
m = 1 (ถัดไปเลย)	0	0x0000
m = 2	1	0x0001
m = 3	**2**	**0x0002**
m = 4	3	0x0003
m = 5	4	0x0004
ทั่วไป	**m − 1**	—

label อยู่ข้างหลัง	imm	hex
k = 1	−2	0xFFFE
k = 2	−3	0xFFFD
k = 3	−4	0xFFFC
k = 4	−5	0xFFFB
k = 5	−6	0xFFFA
ทั่วไป	**−(k + 1)**	—

⚠ ตรวจซ้ำเสมอ
ใช้สูตรกลับ: ปลายทาง = addr(branch) + 4 + imm × 4 ⇒ ต้องได้ address ของ label พอดี ถ้าไม่ตรงแปลว่าคิดผิด

18 · โค้ดแม่แบบ C → MIPS (ลอกไปใช้ได้เลย)

if — เงื่อนไขเดียว

```
# if (a < b) { a = b; }
      slt  $t0, $s0, $s1
      beq  $t0, $zero, ENDIF
      add  $s0, $s1, $zero
ENDIF:
```

if – else

```
# if (a == b) a = 0; else a = 1;
      bne  $s0, $s1, ELSE
      add  $s0, $zero, $zero
      j    ENDIF
ELSE: addi $s0, $zero, 1
ENDIF:
```

while

```
# while (a <= b) { a += i; }
LOOP: slt  $t0, $s1, $s0   # b < a ?
      bne  $t0, $zero, ENDL
      add  $s0, $s0, $s2
      j    LOOP
ENDL:
```

do – while

```
# do { a += i; } while (a < b);
LOOP: add  $s0, $s0, $s2
      slt  $t0, $s0, $s1
      bne  $t0, $zero, LOOP
```

for + array

```
# for (i=0; i<10; i++) a += A[i];
      add  $s2, $zero, $zero
      addi $t1, $zero, 10
FOR:  slt  $t0, $s2, $t1
      beq  $t0, $zero, ENDF
      sll  $t0, $s2, 2      # i*4
      add  $t0, $t0, $s3    # &A[i]
      lw   $t2, 0($t0)
      add  $s0, $s0, $t2
      addi $s2, $s2, 1
      j    FOR
ENDF:
```

สลับค่า + เรียกฟังก์ชัน

```
# swap(a, b)
      add  $t0, $s0, $zero
      add  $s0, $s1, $zero
      add  $s1, $t0, $zero

# เรียกและกลับ
      jal  FUNC
FUNC: addi $sp, $sp, -4
      sw   $ra, 0($sp)
      lw   $ra, 0($sp)
      addi $sp, $sp, 4
      jr   $ra
```

19 · เทียบวงจรอ้างอิง A กับ B — อย่าจำสลับกัน

	วงจรอ้างอิง A	วงจรอ้างอิง B (COE 2019)
Decoder เปิดเมื่อ	A[31]=1 **และ** A[30]=1	A[31]=1 (CS1 = CS0 = A[31])
X1 X0 รับจาก	A[29], A[28]	A[30], A[29]
Y0 / Y1 / Y2 / Y3 →	Boot Mem / Dev 0 / Dev 1 / Dev 2	Dev 0 / Dev 1 / Dev 2 / Dev 3
Main Memory	A[31]=0 ⇒ 0x0000 0000–0x7FFF FFFF	Y_{A1}=0 และ A[31]=0 ⇒ 0x0000 **1000**–0x7FFF FFFF
Boot Mem	0xC000 0000–0xCFFF FFFF (256 MB)	0x0000 0000–0x0000 0FFF (**4 KB**)
DMA	0x8000 0000–0xBFFF FFFF	ไม่มีในวงจรนี้
ขนาดอุปกรณ์	A[27-0] ⇒ 256 MB / ตัว	A[28-0] ⇒ 512 MB / ตัว
ทริก hex หลักแรก	0–7 Main · 8–B DMA · C Boot · D Dev0 · E Dev1 · F Dev2	0x0–0xFFF Boot · 0–7 Main · 8,9 Dev0 · A,B Dev1 · C,D Dev2 · E,F Dev3
ใคร interrupt ได้	ตัวที่มีขา Int ต่อเข้า OR-gate เท่านั้น	วงจรนี้ไม่มีสาย Int

★ วิธีจำที่ปลอดภัยที่สุด
อย่าท่องช่วง address — ให้ท่อง **วิธีถอดวงจร** แทน แล้วถอดสดในห้องสอบจากรูปที่ให้มา ใช้เวลาไม่ถึง 2 นาที และไม่มีทางจำสลับ

15 · ตารางสำเร็จรูป — machine code ของคำสั่งที่ใช้บ่อย

วิธีใช้
ถ้าโจทย์ให้คำสั่งคล้ายกันแต่เปลี่ยน register หรือ offset ให้จำ **โครง**: I-type เปลี่ยน offset = เปลี่ยนแค่ **4 หลักหลัง** · เปลี่ยน register = เปลี่ยน 5 บิตในตำแหน่ง rs/rt

Assembly	Machine code	หมายเหตุ
`add $s0,$s1,$s2`	**0x02328020**	R funct 0x20
`sub $s0,$s1,$s2`	**0x02328022**	R funct 0x22
`and $s0,$s1,$s2`	**0x02328024**	R funct 0x24
`or $s0,$s1,$s2`	**0x02328025**	R funct 0x25
`slt $t0,$s0,$s1`	**0x0211402A**	R funct 0x2A
`add $s0,$s1,$zero`	**0x02208020**	= move $s0,$s1
`sll $s0,$s1,2`	**0x00118080**	rs=0, shamt=2
`srl $s0,$s1,2`	**0x00118082**	rs=0, shamt=2
`jr $ra`	**0x03E00008**	rs=31
`lw $s0,0($s3)`	**0x8E700000**	A[0]
`lw $s0,4($s3)`	**0x8E700004**	A[1]
`lw $s0,12($s3)`	**0x8E70000C**	A[3]
`sw $s0,0($s3)`	**0xAE700000**	A[0]=a
`sw $s0,4($s3)`	**0xAE700004**	A[1]=a
`addi $s0,$s0,1`	**0x22100001**	a++
`addi $s0,$zero,-4`	**0x2010FFFC**	a = -4
`addi $sp,$sp,-4`	**0x23BDFFFC**	จอง stack
`ori $s0,$zero,0x27`	**0x34100027**	a = 0x27
`lui $s5,0xA000`	**0x3C15A000**	rs = 0 เสมอ
`beq $t0,$zero,+2`	**0x11000002**	ข้าม 2 คำสั่ง
`bne $t0,$zero,-4`	**0x1500FFFC**	ย้อน 4 คำสั่ง
`j 0x00400000`	**0x08100000**	field=0x100000
`j 0x00400050`	**0x08100014**	field=0x100014
`jal 0x00400000`	**0x0C100000**	เก็บ $ra

★ ตรวจคำตอบเร็วๆ
hex หลักแรก บอกชนิดคำสั่งได้ทันที: **0**=R-type · **08/09**=j/jal · **1**=beq/bne · **2**=addi/andi(2x) · **3**=ori/lui(3x) · **8**=lw · **A**=sw
ถ้าถอด hex แล้วได้ op ที่ไม่มีในตาราง ⇒ แปลว่าคำนวณผิดหรือสลับ endian

16 · คำถามชิงคะแนน 25 ข้อ (ถาม–ตอบสั้น)

#	คำถาม	คำตอบสั้น
1	ทำไม address bus เป็นทางเดียว	มีแต่ bus master (CPU) ที่กำหนดตำแหน่ง อุปกรณ์ปลายทางเป็นฝ่ายถูกเลือกเท่านั้น
2	ทำไม data bus ต้องสองทาง	ต้องรองรับทั้ง read และ write ด้วยสายชุดเดียว ประหยัดจำนวนขา/สาย
3	Hi-Z คืออะไร จำเป็นยังไง	สถานะ “ไม่ขับสัญญาณ” ทำให้อุปกรณ์อื่นใช้ bus ได้โดยไม่ชนกัน
4	CS ต่างจาก read/write ยังไง	CS บอกว่า “ชิปไหน” ถูกเลือก · read/write บอกว่า “จะทำอะไร”
5	ทำไม decoder ต้องมีขา CS	เพื่อ “ปิด” เอาต์พุตทุกเส้นเมื่อ address ไม่อยู่ในช่วงที่มันดูแล
6	ถ้า decoder เลือก 2 เส้น พร้อมกัน	อุปกรณ์ 2 ตัวขับ data bus พร้อมกัน ⇒ bus contention
7	memory-mapped I/O ดี/เสียยังไง	ดี: ใช้ `lw`/`sw` เดิม ฮาร์ดแวร์ง่าย · เสีย: กินพื้นที่ address ของหน่วยความจำ
8	ทำไม Boot Mem ต้อง non-volatile	ตอนเปิดเครื่อง RAM ยังว่าง ต้องมีโปรแกรมค้างอยู่จริงถึงจะเริ่มระบบได้
9	Bootstrapping Address คืออะไร	ค่า PC เริ่มต้นตอน reset — คำสั่งแรกของเครื่องอยู่ที่นั่น
10	ทำไมคำสั่ง MIPS ยาว 4 ไบต์เท่ากันหมด	ถอดรหัสง่าย ทำ pipeline ได้ และคำนวณ PC+4 ตรงไปตรงมา
11	ทำไม branch ใช้ offset ไม่ใช่ address เต็ม	ประหยัดบิต (เหลือ 16 บิต) และทำให้โค้ด **relocatable**
12	ทำไม `j` ต้องหาร 4	address ของคำสั่งลงตัว 4 เสมอ ⇒ 2 บิตล่างเป็น 0 ไม่ต้องเก็บ ⇒ ยิงได้ไกลขึ้น 4 เท่า
13	ทำไม `j` ข้าม region ไม่ได้	บิต 31–28 ของ PC ใหม่ถูกยืดมาจาก PC+4 ไม่ได้อยู่ในคำสั่ง
14	`j` ต่างจาก `jr` ยังไง	`jr` ใช้ค่าใน register เป็นปลายทาง ⇒ กระโดดได้ทั้ง 4 GB
15	`$zero` มีไว้ทำไม	ทำให้เขียน move / clear / compare ได้โดยไม่ต้องเพิ่มคำสั่งใหม่
16	`lui` มีไว้ทำไม	immediate มีแค่ 16 บิต ⇒ ใส่ค่า 32 บิตต้องใช้ `lui` + `ori` สองคำสั่ง
17	ทำไม `andi`/`ori` zero-extend แต่ `addi` sign-extend	คำสั่งลอจิกทำงานกับรูปแบบบิต ส่วนคำสั่งคณิตต้องรองรับค่าติดลบ
18	`slt` มีไว้ทำไมทั้งที่มี `beq`/`bne`	`beq`/`bne` เทียบได้แค่ “เท่า/ไม่เท่า” ⇒ ต้องมี `slt` เพื่อเทียบ < และ >
19	pseudo-instruction คืออะไร	คำสั่งที่ assembler แปลงเป็นคำสั่งจริง 1 ตัวขึ้นไป — ฮาร์ดแวร์ไม่มีจริง
20	Endianness สำคัญตรงไหน	ลำดับไบต์ในหน่วยความจำ มีผลตอนถอดรหัสคำสั่งและตอนแลกข้อมูลข้ามเครื่อง
21	polling ต่างจาก interrupt ยังไง	polling = CPU ถามเอง (เสียเวลา) · interrupt = อุปกรณ์เรียก CPU เมื่อพร้อม
22	ทำไม DMA ยังต้อง interrupt CPU	เพื่อบอกว่า “ย้ายข้อมูลเสร็จแล้ว” CPU จะได้มาใช้ข้อมูลต่อ
23	ISR ต้องเก็บอะไรก่อนทำงาน	PC เดิม (ที่กลับ) + register ที่จะใช้ + สถานะ/flag ของ CPU
24	interrupt vector table คืออะไร	ตารางเก็บ address ของ ISR ของแต่ละสาเหตุ/อุปกรณ์
25	exception ต่างจาก interrupt ยังไง	exception เกิดจากภายใน CPU เอง (หารศูนย์ overflow คำสั่งไม่รู้จัก) · interrupt มาจากอุปกรณ์ภายนอก