เฉลย A01 + วิธีคิด + เคสพลิกโจทย์

198322 สถาปัตยกรรมคอมพิวเตอร์ · แบบฝึกหัดครั้งที่ 1 · Processor / Memory / I/O / Bus & MIPS-32

4 หน้า · 2 แผ่น

⚑ 12 กฎทองที่ตอบได้เกินครึ่งข้อสอบ

1) Address / Read / Write = **ทางเดียว CPU → Mem** · Data bus = **สองทาง**
2) read active → **Memory ขับ** data bus · write active → **CPU ขับ**
3) address n บิต + data 8 บิต → อ้างได้ **2^n ไบต์**
4) เขียน bus ได้ **ทีละ 1 ตัว** (ตัวที่ถือสิทธิ) แต่ **อ่านได้ทุกตัว**
5) เขียนพร้อมกัน = **bus contention** → ค่าขยะ + กระแสลัดวงจร
6) อุปกรณ์ที่ interrupt ได้ = ตัวที่ **มีขา Int ต่อสายเข้า OR-gate** เท่านั้น
7) Interrupt: ไป vector → ทำ ISR → **กลับที่ address เดิม** (ยังไม่ได้ทำคำสั่งนั้น)
8) อุปกรณ์ไหนถูกเลือก = ดู **CS มาจากสายไหน** · วงกลม ○ = active-LOW
9) `beq/bne` = offset **สัมพัทธ์** (ย้าย code ไม่เปลี่ยน) · `j` = **สัมบูรณ์** (เปลี่ยน)
10) Big Endian: ไบต์ address **ต่ำสุด** = ไบต์ **ซ้ายสุด** ของ word
11) ใน `lw rt,off(rs)` เครื่องเรียง **rs ก่อน rt** (สลับกับที่เขียน)
12) Memory-mapped I/O: `lw/sw` ที่ address ตกในช่วงอุปกรณ์ = **ติดต่อ I/O**

คำถาม 1 — Bus & ทิศทางสัญญาณ (10 pt)

รูป 1 — ลูกศรคือคำตอบข้อ 1.1–1.4

ข้อ	สาย	คำตอบ	เหตุผล
1.1	Address bus	**CPU A → Memory**	CPU เป็นคนบอกว่าจะยุ่งกับช่องไหน Memory ไม่เคยกำหนดเอง
1.2	Data bus	**สองทาง (both)**	read = Mem → CPU, write = CPU → Mem ใช้เส้นเดียวกัน
1.3	Read	**CPU A → Memory**	สายควบคุม CPU สั่ง Memory
1.4	Write	**CPU A → Memory**	สายควบคุม CPU สั่ง Memory
1.7	addr 5 บิต, data 8 บิต	**32 ไบต์**	2^5 = 32 ช่อง × 1 ไบต์
1.8	addr 6 บิต, data 8 บิต	**64 ไบต์**	2^6 = 64 ช่อง × 1 ไบต์

1.5 — read active, write inactive, addr = 0x20004000

เขียนอธิบาย

คำตอบ (ต้องตอบ 2 ส่วน)

① ใครขับ: Memory เป็นตัวเขียนลง data bus (CPU เป็นฝ่ายอ่าน) — เพราะ `read` active แปลว่า CPU สั่ง "อ่าน"
② ค่าเป็นเท่าไร: ระบุไม่ได้จากข้อมูลที่โจทย์ให้ เพราะมันคือเนื้อข้อมูล 32 บิตที่ถูกเก็บอยู่ที่ช่อง 0x20004000 ซึ่งโจทย์ไม่ได้บอกว่าในหน่วยความจำมีอะไร

อธิบายให้เต็มคะแนน — เขียนแบบนี้

สาย `read` active ⇒ CPU ร้องขอข้อมูล ⇒ Memory ถอดรหัส address 0x20004000 แล้ว **ขับ (drive)** ข้อมูล 32 บิตออกมาบน data bus ให้ CPU latch เข้าไป ในจังหวะนี้ CPU ต้องปล่อย data bus เป็น **Hi-Z** ไม่งั้นชนกัน. ค่าที่ปรากฏจึงเป็น "ค่าที่เก็บไว้ในหน่วยความจำ" ไม่ใช่ตัวเลขที่คำนวณจาก address ⇒ **ตอบเป็นตัวเลขไม่ได้**

1.6 — read inactive, write active, addr = 0x20008000

เขียนอธิบาย

คำตอบ

① CPU A เป็นตัวเขียนลง data bus (Memory เป็นฝ่ายรับแล้วบันทึกลงช่อง 0x20008000)
② ค่า = ระบุไม่ได้ — คือข้อมูล 32 บิตที่โปรแกรมสั่งให้เขียน ซึ่งโจทย์ไม่ได้ให้มา

⚡ ถ้าอาจารย์พลิกโจทย์แบบนี้

read + write inactive ทั้งคู่ → ไม่มีใครขับ bus → data bus เป็น **Hi-Z / floating / ค่าไม่นิยาม** และ Memory ไม่ถูกเข้าถึงเลย
read + write active พร้อมกัน → ผิดกติกา เกิด **bus conflict** ทั้ง CPU และ Memory ขับพร้อมกัน ค่าไม่นิยาม + อาจเสียหาย
เพิ่ม CPU B / DMA ที่เป็น bus master ได้ → address bus กลายเป็น **สองทางในเชิงระบบ** (ยังทางเดียวเมื่อมองจาก master ตัวเดียว) และต้องมี **arbiter**
ถาม "ค่บน address bus" → ตอบได้ เพราะ CPU ใส่เอง = **0x2000 4000**
data 16 บิต, addr 10 บิต, ถามเป็นไบต์ → ถ้า 1 address = 1 word 16 บิต ⇒ $2^{10} \times 2$ = **2048 ไบต์**; ถ้า byte-addressable ⇒ 2^{10} = **1024 ไบต์** (อ่านโจทย์ให้ชาด)
"address bus 32 บิต อ้างได้เท่าไร" → 2^{32} = **4 GB** · **"ต้องใช้กี่บิตถึงอ้าง 1 MB"** → $2^n = 2^{20}$ ⇒ **20 บิต**
"อยากได้ 1 GB ด้วย data bus 32 บิต แบบ word-addressable" → 1 GB / 4 = 2^{28} ช่อง ⇒ **28 บิต**
"bus ทำงาน 100 MHz data 32 บิต bandwidth เท่าไร" → 100M × 4 ไบต์ = **400 MB/s**
"multiplexed bus คืออะไร" → ใช้สายชุดเดียวส่งทั้ง address และ data สลับเวลากัน ⇒ ประหยัดขา แต่ช้าลง (ต้อง 2 จังหวะ)

⚠ กับดัก

สูตรคือ **2^(จำนวนบิต address)** ไม่เกี่ยวกับสาย read/write ที่ให้มาหลอกๆ และ **ไม่ใช่** 2^addr × 2^data

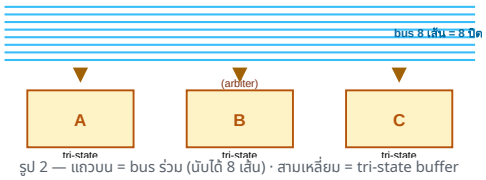
ตารางแปลงเลขที่ต้องใช้ตลอดข้อสอบ

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

2's complement 16 บิต (ใช้กับ offset ติดลบ)

วิธี: กลับบิตทุกตัวแล้ว +1 · หรือลัด: **0x10000 − |n|**
−1 = **0xFFFF** · −2 = 0xFFFE · −4 = **0xFFFC** · −8 = 0xFFF8 · −16 = 0xFFF0 · −32 = 0xFFE0 · −100 = 0xFF9C
ช่วงที่เก็บได้: **−32768 … +32767**

คำถาม 2 — Bus Arbitration (10 pt)



รูป 2 — แถวบน = bus ร่วม (นับได้ 8 เส้น) · สามเหลี่ยม = tri-state buffer

ข้อ	คำถาม	คำตอบ
2.1	bus นี้กี่บิต	**8 บิต** (นับเส้นสัญญาณแถวบนในรูป 2)
2.2	A ได้สิทธิ → ใครเขียนได้	**A เท่านั้น**
2.3	B ถือสิทธิ → ใครเขียนได้	**B เท่านั้น**
2.4	C ได้สิทธิ → ใครเขียนได้	**C เท่านั้น**
2.5	A ได้สิทธิ → ใครอ่านได้	**A, B, C**
2.6	B ถือสิทธิ → ใครอ่านได้	**A, B, C**
2.7	C ได้สิทธิ → ใครอ่านได้	**A, B, C**

หลักคิด 2 บรรทัด

เขียน = ขับสัญญาณ (drive) ⇒ ต้องมีตัวเดียว ⇒ เฉพาะตัวที่ถือสิทธิ
อ่าน = แค่วัดแรงดัน (sense) ⇒ ไม่รบกวนใคร ⇒ ทุกตัวบน bus อ่านได้เสมอ เพราะ bus เป็น **broadcast** — รวมตัวที่เขียนเองด้วย

⚠ กับดักข้อ 2.3

B เป็น **arbiter** ก็จริง แต่ arbiter คือ "คนแจกสิทธิ" ไม่ใช่ "คนเขียนได้ตลอด" ⇒ ตอบ **B ตัวเดียว**

2.8 — ถ้า A, B, C เขียนพร้อมกัน

เขียนอธิบาย

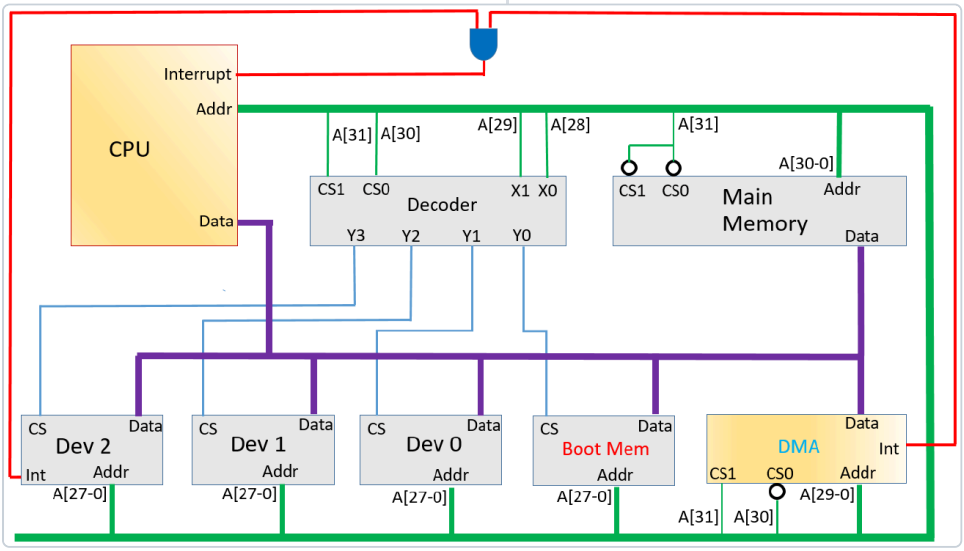
คำตอบ — เขียนให้ครบ 4 ชั้นนี้

① ชื่อปรากฏการณ์: เกิด **Bus contention / Bus conflict** — มี driver หลายตัวขับเส้นเดียวกันไปคนละระดับลอจิกในเวลาเดียวกัน
② ผลต่อข้อมูล: แรงดันบนเส้นถูกดึงไปอยู่ระหว่าง 0 กับ 1 → เป็นค่า **ไม่นิยาม (undefined)** ทุกตัวที่อ่านจะได้ข้อมูลผิด/ขยะ ระบบทำงานผิดพลาด
③ ผลต่อฮาร์ดแวร์: ตัวหนึ่งดันขึ้น V_{DD} อีกตัวดึงลง GND → เกิดเส้นทาง **ลัดวงจร** กระแสสูงไหลผ่าน output driver → ร้อน กินไฟ และ **ชิปอาจเสียหายถาวร**
④ วิธีป้องกัน: ใช้ arbiter + **tri-state buffer** บังคับให้ตัวที่ไม่ได้สิทธิอยู่ **Hi-Z** (คือสามเหลี่ยมในรูป 2) หรือใช้ open-drain + pull-up ให้เป็น wired-AND

⚡ ถ้าพลิกโจทย์

"ถ้า arbiter (B) พัง" → ไม่มีใครแจกสิทธิ อาจไม่มีตัวไหนเขียนได้เลย (ระบบค้าง) หรือเขียนพร้อมกันจนเกิด contention
"ถ้าใช้ open-drain แทน tri-state" → ไม่ลัดวงจร (ไม่มีใครดัน HIGH) แต่ผลลัพธ์เป็น **wired-AND** ของทุกตัว ⇒ ข้อมูลยังผิดอยู่ดี
"5 อุปกรณ์ต้องใช้ dedicated line กี่เส้น" → 1 เส้น/อุปกรณ์ = **5 เส้น**; ถ้าเข้ารหัสฐานสอง = $\lceil \log_2 5 \rceil$ = **3 เส้น**
"bus 8 บิต ส่งข้อมูล 32 บิต กี่รอบ" → 32/8 = **4 รอบ**
"นับเส้นได้ 16 เส้น" → ตอบ 16 บิต (ข้อนี้วัดการนับเส้นล้วนๆ)
"ข้อดี/ข้อเสียของ bus ร่วม" → ดี: สายน้อย ต่ออุปกรณ์เพิ่มง่าย ถูก · เสีย: ใช้ได้ทีละตัว (**bottleneck**) ต้องมี arbiter ความเร็วจำกัดโดยตัวช้าสุด
"แบบวิธี arbitration มีอะไรบ้าง" → **Daisy chain** (ต่อสิทธิเป็นทอดๆ ตัวใกล้ได้เปรียบ) · **Centralized / dedicated lines** (แบบในรูป) · **Distributed / self-selection** · **Round-robin** (ยุติธรรม ไม่มี starvation)
"starvation คืออะไร" → อุปกรณ์ที่ priority ต่ำไม่ได้สิทธิเลยสักที แก้ด้วย round-robin หรือ aging

คำถาม 3 — Interrupt + Memory Map (รูป 3) (10 pt)



รูป 3 — Main Memory = volatile · Boot Mem = non-volatile · สายสีแดง = interrupt · สายฟ้าบาง = chip-select จาก decoder

สูตร "ถอดรูป" ให้ได้ Address Map ใน 3 ขั้น (ใช้ได้กับทุกรูปที่อาจารย์ให้)

ขั้น 1 ดูว่า **CS ของแต่ละกล่องมาจากสายอะไร** และมี **วงกลม ○ (bubble)** หรือไม่ — มี ○ = active LOW (สายนั้นต้องเป็น **0** ถึงจะถูกเลือก), ไม่มี ○ = active HIGH (ต้องเป็น **1**)
ขั้น 2 Decoder ทำงานเมื่อ CS ทุกขา active เท่านั้น แล้วดูว่า `X1 X0` รับบิตไหน → เลือก Y0/Y1/Y2/Y3 ตามค่า
ขั้น 3 ดูว่าแต่ละกล่องรับ `A[k-0]` กี่บิต → ขนาดพื้นที่ของมัน = **2^(k+1) ไบต์**

ขา (รูป 3)	ต่อกับ	ความหมาย
Main Mem CS1, CS0	A[31] ผ่าน ○ ทั้งคู่	เลือกเมื่อ **A[31] = 0**
DMA CS1 / CS0	A[31] / A[30] ผ่าน ○	เลือกเมื่อ **A[31]=1, A[30]=0**
Decoder CS1, CS0	A[31], A[30]	ทำงานเมื่อ **A[31]=1 และ A[30]=1**
Decoder X1, X0	A[29], A[28]	Y0→Boot, Y1→Dev0, Y2→Dev1, Y3→Dev2
Dev / Boot Addr	A[27-0]	ตัวละ 2^{28} = **256 MB**

A[31:28]	hex หลักแรก	อุปกรณ์	ช่วง address
0xxx	**0–7**	**Main Memory**	0x0000 0000 – 0x7FFF FFFF
10xx	**8–B**	**DMA**	0x8000 0000 – 0xBFFF FFFF
1100	**C**	**Boot Mem**	0xC000 0000 – 0xCFFF FFFF
1101	**D**	**Dev 0**	0xD000 0000 – 0xDFFF FFFF
1110	**E**	**Dev 1**	0xE000 0000 – 0xEFFF FFFF
1111	**F**	**Dev 2**	0xF000 0000 – 0xFFFF FFFF

★ ทริกจำ 5 วินาที (รูป 3) — ดูแค่ hex หลักแรก
0–7 Main · 8–B DMA · C Boot · D Dev0 · E Dev1 · F Dev2

3.1 — ใครสั่ง interrupt CPU ได้

คำตอบ
DMA และ Dev 2 เท่านั้น

ทำไม
ในรูปมี **สายสีแดง** ออกจากขา `Int` ของ **Dev 2** (ซ้ายล่าง) และของ **DMA** (ขวาล่าง) วิ่งเข้า **OR-gate** แล้วเข้าขา `Interrupt` ของ CPU. กล่องที่เหลือ (Decoder, Main Memory, Boot Mem, Dev 0, Dev 1) **ไม่มีขา Int** จึงส่ง interrupt ไม่ได้ — ตอบตาม **สายที่เห็นในรูป** อย่าเดาจากชื่ออุปกรณ์

3.2 — ถ้าไม่มี Interrupt แล้ว CPU รู้ได้ยังไงว่าอุปกรณ์ทำเสร็จ เขียนอธิบาย

คำตอบ
ใช้ **Polling** (programmed I/O / busy-waiting): CPU **วนลูปอ่าน status register** ของอุปกรณ์ผ่าน memory-mapped I/O ซ้ำๆ แล้วตรวจบิต ready/done ว่าขึ้นหรือยัง ถ้ายังก็วนอ่านใหม่ ถ้าขึ้นแล้วจึงไปอ่านข้อมูลผลลัพธ์

```
# โครงโปรแกรม polling
poll: lw   $t0, 0($s0)    # $s0 = addr ของ status
      andi $t0, $t0, 1    # แยกบิต DONE
      beq  $t0, $zero, poll
      lw   $t1, 4($s0)    # เสร็จแล้ว → อ่านข้อมูล
```

ข้อเสีย: เปลือง CPU cycle ไปกับการถามซ้ำ (busy-wait) ทำงานอื่นไม่ได้ · ถ้าเว้นช่วง poll ห่างก็ตอบสนองช้า (latency สูง) · อุปกรณ์เยอะยิ่งไม่ scale
ข้อดี: ฮาร์ดแวร์ง่าย ไม่ต้องมีสาย Int / ISR / vector table และคาดเดาเวลาได้ (เหมาะกับ real-time ง่ายๆ)

	Polling	Interrupt	DMA
ใครเริ่ม	CPU ถามเอง	อุปกรณ์บอก CPU	อุปกรณ์ย้ายเอง
CPU เสียเวลา	มาก (busy-wait)	น้อย	น้อยมาก
ฮาร์ดแวร์	ง่ายสุด	ต้องมีสาย Int + ISR	ต้องมีตัวคุม bus
เหมาะกับ	อุปกรณ์เร็ว/น้อยตัว	เหตุการณ์ไม่แน่นอน	ย้ายข้อมูลก้อนใหญ่

3.3 – 3.5 — สถานการณ์ Interrupt

กติกา 4 ช่อง (a)(b)(c)(d) จำแค่นี้
(a) คำสั่งถัดไป = **ค่า interrupt vector**
(b) ISR อยู่ที่ = **ค่า vector เดียวกัน** (โจทย์นี้ vector ชี้ตรงไป ISR)
(c) Main / Boot = **เอา vector ไปเทียบ address map** (ดู hex หลักแรก)
(d) จบ ISR = **กลับไป address เดิม** ที่ยังไม่ได้ทำ — โจทย์เขียนว่า "ก่อนจะโหลดคำสั่งที่ ..." ⇒ คำสั่งนั้นยังไม่ถูก execute ⇒ ไม่ต้อง +4

ช่อง	ข้อ 3.3	ข้อ 3.4	ข้อ 3.5
คำสั่งที่ค้างอยู่	0x0000 1000	0xCFFE 1000	0x0041 0080
interrupt vector	0xCFFF 0000	0x0001 0000	0x0500 0000
(a) ไปทำต่อที่	**0xCFFF0000**	**0x00010000**	**0x05000000**
(b) ISR อยู่ที่	**0xCFFF0000**	**0x00010000**	**0x05000000**
(c) Main / Boot	**Boot Mem** (หลักแรก C)	**Main Mem** (หลักแรก 0)	**Main Mem** (หลักแรก 0)
(d) จบ ISR กลับไปที่	**0x00001000**	**0xCFFE1000**	**0x00410080**

⚠ เฉลยของอาจารย์ข้อ 3.4 (a) พิมพ์ว่า 0x00001000
อันนั้น **พิมพ์ผิด/ก๊อปมาจากข้อ 3.3** — เพราะ (a) ต้องเท่ากับ vector เสมอ (สังเกตข้อ 3.3 และ 3.5 ที่ (a) = (b)) ⇒ ที่ถูกคือ **0x00010000**. ถ้าเจอในห้องสอบให้ตอบ 0x00010000 แล้วเขียนกำกับว่า "= interrupt vector"

ลำดับที่ CPU ทำจริงเมื่อรับ interrupt (เขียนเพิ่มได้คะแนน)
① ทำคำสั่งปัจจุบันให้จบก่อน ② เช็คว่า interrupt enable อยู่ไหม ③ **เก็บ PC เดิม** (และ status/registers) ลง stack หรือ EPC ④ กระโดดไปที่ **interrupt vector** ⑤ ทำ ISR ⑥ ISR ตอบรับ/เคลียร์ flag ที่อุปกรณ์ ⑦ คืนค่า register แล้ว **กลับที่ PC เดิม** (คำสั่ง `eret / rfe`)

⚡ ถ้าพลิกโจทย์
"หลังจาก<u>ทำคำสั่งเสร็จ</u>ที่ 0x00410080 แล้วได้ interrupt" → (d) = **0x00410084** (+4)
"ถ้า CPU ปิดรับ interrupt (masked/disabled)" → ไม่กระโดด ทำคำสั่งเดิมต่อ (a) = 0x00410080 และไม่มี ISR
"vector = 0x9000 0000" → หลักแรก 9 → **DMA** ⇒ ผิดปกติ เพราะ ISR ต้องอยู่ในหน่วยความจำ
"vector = 0xE000 0100" → หลักแรก E → **Dev 1** · **"vector = 0xD000 0000"** → **Dev 0**
"ย้ายสาย X1,X0 ไปรับ A[27],A[26]" → ต้องกางบิตใหม่: A[31:30]=11 คงเดิม แล้ว A[27:26] เป็นตัวเลือก Y ⇒ พื้นที่แต่ละอุปกรณ์กระจายเป็นก้อน 64 MB สลับกัน ไม่ต่อเนื่อง
"ถ้าเอา ○ ออกจาก CS ของ Main Memory" → Main Mem ถูกเลือกเมื่อ A[31]=**1** ⇒ ชนกับ DMA/Decoder ⇒ ระบบผิด
"ใครเก็บโปรแกรมตอนปิดเครื่อง" → **Boot Mem** (non-volatile) · Main Memory เป็น volatile ข้อมูลหายเมื่อดับไฟ
"DMA มีไว้ทำไม" → ย้ายข้อมูลระหว่าง I/O กับหน่วยความจำ **โดยไม่ผ่าน CPU** แล้ว interrupt บอก CPU เมื่อเสร็จ ⇒ CPU เอาเวลาไปทำอย่างอื่น
"ถ้า Dev 0 กับ Dev 1 interrupt พร้อมกัน CPU รู้ได้ไง" → OR-gate บอกแค่ "มีคน interrupt" ⇒ ต้อง **poll status ของแต่ละตัว** ใน ISR หรือใช้ **vectored interrupt / priority encoder** แยกสาย

คำถาม 4 — เขียน MIPS Assembly แทน C (10 pt)

โจทย์กำหนด
`$s0` = `a` · `$s1` = `b` · `$s2` = `i` · `$s3` = base address ของ array A (int = 4 ไบต์) · **ใช้ pseudo-instruction = −1 คะแนน**

(a) `a = A[0] + 32;`

```
lw   $t0, 0($s3)      # $t0 = A[0]   offset = 0*4
addi $s0, $t0, 32     # a = A[0] + 32
```

(b) `A[1] = a;`

```
sw   $s0, 4($s3)      # offset = 1*4 = 4
```

(c) `a = -4;`

```
addi $s0, $zero, -4   # ห้ามใช้ li $s0,-4 (pseudo!)
```

(d) `if (a < b) { a = b; } a++;`

```
      slt  $t0, $s0, $s1     # $t0 = (a < b) ? 1 :
      beq  $t0, $zero, ENDIF # ไม่จริง → ข้าม
      add  $s0, $s1, $zero   # a = b  (แทน move)
ENDIF:
      addi $s0, $s0, 1       # a++
```

(e) `while (a <= b) { a += i; } A[0] = a;`

```
LOOP: slt  $t0, $s1, $s0     # $t0 = (b < a) = นิเสธ
      bne  $t0, $zero, ENDLOOP
      add  $s0, $s0, $s2     # a += i
      j    LOOP
ENDLOOP:
      sw   $s0, 0($s3)       # A[0] = a
```

★ สูตรจำ slt
`slt rd, X, Y` อ่านว่า "**X น้อยกว่า Y ไหม**" ⇒ อยากได้ **a ≤ b** ให้พลิกเป็น **ไม่ใช่(b < a)** ⇒ เขียน `slt $t0,$s1,$s0` แล้ว `bne` ออกจากลูป

ตารางแปลง pseudo → คำสั่งจริง (ท่องให้ได้)

Pseudo	เขียนแทนด้วย
`li $s0, N`	`addi $s0,$zero,N` (N ใน 16 บิต) หรือ `lui` + `ori`
`move $s0,$s1`	`add $s0,$s1,$zero`
`la $s0, LBL`	`lui $s0,upper ; ori $s0,$s0,lower`
`blt $a,$b,L`	`slt $t0,$a,$b ; bne $t0,$zero,L`
`bgt $a,$b,L`	`slt $t0,$b,$a ; bne $t0,$zero,L`
`ble $a,$b,L`	`slt $t0,$b,$a ; beq $t0,$zero,L`
`bge $a,$b,L`	`slt $t0,$a,$b ; beq $t0,$zero,L`
`bnez / beqz`	`bne / beq $x,$zero,L`
`nop`	`sll $zero,$zero,0`
`not $s0,$s1`	`nor $s0,$s1,$zero`

แม่แบบแปลง C → Assembly

C	โครง Assembly
`if(cond){X}`	ตรวจ **นิเสธ** ของ cond แล้ว branch ข้าม X
`if(c){X}else{Y}`	นิเสธ→ELSE / X / `j END` / `ELSE:` Y / `END:`
`while(c){X}`	`L:` ตรวจนิเสธ→END / X / `j L` / `END:`
`do{X}while(c)`	`L:` X / ตรวจ c → `bne/beq … L`
`for(init;c;up)`	init / `L:` ตรวจ→END / body / up / `j L` / `END:`

⚡ ถ้าพลิกโจทย์ข้อ 4
`A[i] = a;` (index เป็นตัวแปร) → ต้องคูณ 4 เอง:

```
sll  $t0, $s2, 2      # $t0 = i*4
add  $t0, $t0, $s3    # &A[i]
sw   $s0, 0($t0)
```

`a = A[2] - b;` → `lw $t0,8($s3) ; sub $s0,$t0,$s1`
`b = A[i+1];` → `addi $t0,$s2,1 ; sll $t0,$t0,2 ; add $t0,$t0,$s3 ; lw $s1,0($t0)`
`if(a==b){a=0;}else{a=1;}`

```
      bne  $s0,$s1,ELSE
      add  $s0,$zero,$zero
      j    ENDIF
ELSE: addi $s0,$zero,1
ENDIF:
```

`for(i=0;i<10;i++) a+=A[i];`

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

`a = a*8;` → `sll $s0,$s0,3` · **`a = a/4;`** (a≥0) → `srl $s0,$s0,2`
สลับค่า a กับ b → `add $t0,$s0,$zero ; add $s0,$s1,$zero ; add $s1,$t0,$zero`
ถ้า array เป็น `char` (1 ไบต์) → offset ไม่ต้อง ×4 และใช้ `lb/sb` แทน `lw/sw`
ถ้า array เป็น `double` (8 ไบต์) → offset = index × 8 ⇒ `sll $t0,$s2,3`
ถ้าห้ามใช้ `$t` → ยืม `$s4` – `$s7` (อย่าใช้ `$at` เพราะ assembler สงวนไว้)

คำถาม 5 — Assembly → Machine Code (10 pt)

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

R-type — `add rd, rs, rt` · op = 000000 เสมอ

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

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

op 31–26	target address 25–0

J-type — `j target`

สูตรที่ต้องท่อง
Branch offset = (address ของ label − (address ของ branch + 4)) ÷ **4** → เก็บเป็น 16 บิต signed
Jump field = address ปลายทาง ÷ **4** → เก็บ 26 บิตล่าง
ย้อนกลับ: PC ใหม่ = (PC+4)[31:28] : field(26) : **00**

5.1 & 5.2

คำสั่ง	op	rs	rt	imm (16 บิต)	hex
`lw $s0,4($s1)`	100011	10001 $s1=17	10000 $s0=16	0000 0000 0000 0100	**0x8E30 0004**
`sw $s0,-4($s1)`	101011	10001	10000	1111 1111 1111 1100 (−4 = 2's comp)	**0xAE30 FFFC**

⚠ ท่องให้ขึ้นใจ
ใน `lw rt, off(rs)` ตัวในวงเล็บคือ **rs** (ฐาน) ตัวหน้าคือ **rt** (ปลายทาง) ⇒ ในเครื่อง **rs มาก่อน rt** เสมอ สลับกับที่เขียน!

5.3 — เริ่มที่ 0x00400054 · 5.4 — เริ่มที่ 0x00408054

addr	Assembly	วิธีคิด	hex 5.3	hex 5.4
...054	`lab0: beq $t0,$s1,lab1`	offset = (...060 − (...054+4))/4 = 8/4 = **2** 000100·01000·10001·0002	**0x1111 0002**	**0x1111 0002**
...058	`addi $s1,$s1,1`	001000·10001·10001·0001	**0x2231 0001**	**0x2231 0001**
...05C	`j lab0`	0x00400054÷4 = 0x0100015 0x00408054÷4 = 0x0102015	**0x0810 0015**	**0x0810 2015**
...060	`lab1: add $s2,$s1,$zero`	000000·10001·00000·10010·00000·100000	**0x0220 9020**	**0x0220 9020**

★ ประเด็นที่อาจารย์ตั้งใจวัด
ย้ายโปรแกรมจาก 0x00400054 → 0x00408054 แล้ว **เปลี่ยนแค่ `j` ตัวเดียว** เพราะ `j` ใช้ address **สัมบูรณ์** ส่วน `beq` ใช้ offset **สัมพัทธ์** จึงไม่เปลี่ยน — นี่คือเหตุผลที่ branch ทำให้โค้ด **relocatable**

ตารางอ้างอิงเข้ารหัส

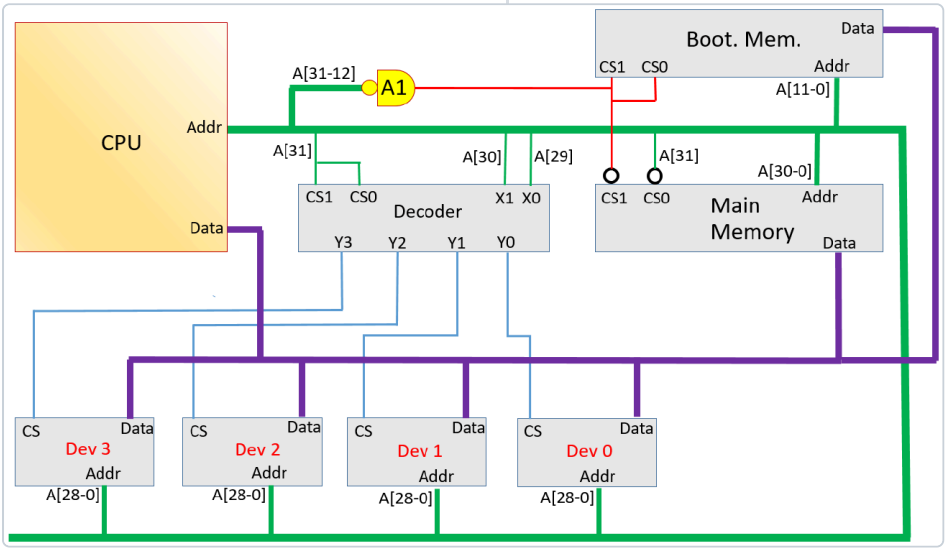
R-type	funct
`add` / `addu`	0x20 / 0x21
`sub` / `subu`	0x22 / 0x23
`and` / `or`	0x24 / 0x25
`xor` / `nor`	0x26 / 0x27
`slt` / `sltu`	0x2A / 0x2B
`sll` / `srl`	0x00 / 0x02
`jr`	0x08

I / J	op
`lw` / `sw`	0x23 / 0x2B
`lb` / `sb`	0x20 / 0x28
`addi` / `addiu`	0x08 / 0x09
`andi` / `ori`	0x0C / 0x0D
`lui` / `slti`	0x0F / 0x0A
`beq` / `bne`	0x04 / 0x05
`j` / `jal`	0x02 / 0x03

Reg	#	bin	Reg	#	bin
`$zero`	0	00000	`$s0`	16	10000
`$v0-$v1`	2–3	00010–11	`$s1`	17	10001
`$a0-$a3`	4–7	00100–111	`$s2`	18	10010
`$t0-$t7`	8–15	01000–1111	`$s3`	19	10011
`$t8-$t9`	24–25	11000–11001	`$s5`	21	10101
`$sp` / `$ra`	29/31	11101/11111	`$s7`	23	10111

⚡ ถ้าพลิกโจทย์ข้อ 5
ให้ hex มาแล้วให้ถอดเป็น assembly → กาง 32 บิต → ดู 6 บิตแรก: **000000 = R** (ไปดู funct 6 บิตท้าย), **000010/000011 = J**, ที่เหลือ = **I**
label อยู่ก่อน branch (ลูปย้อนกลับ) → offset ติดลบ เช่น `beq` ที่ ...060 กระโดดกลับ ...054 ⇒ (0x054 − 0x064)/4 = **−4** → imm = **0xFFFC**
“`j` ไปได้ไกลแค่ไหน” → 26 บิต × 4 = 2^{28} = **256 MB** ภายใน region เดียวกัน (บิต 31–28 มาจาก PC+4) ⇒ ข้าม region ต้องใช้ `jr`
“`beq` ไปได้ไกลแค่ไหน” → $\pm 2^{15}$ คำสั่ง = **−32768 ... +32767 คำสั่ง** (≈ ±128 KB)
ย้ายโปรแกรมไป 0x10400054 → `j` : 0x10400054÷4 = 0x04100015 → เอา 26 บิตล่าง = 0x0100015 ⇒ ยังเป็น **0x08100015** (บิตบนหายไปแต่ทำงานถูกเพราะอยู่ region เดียวกัน)
“ทำไมต้องหาร 4” → ทุกคำสั่ง MIPS ยาว 4 ไบต์และ address ลงตัว 4 เสมอ ⇒ 2 บิตล่างเป็น 0 ตลอด ไม่ต้องเก็บ ⇒ ประหยัดบิตและยิงได้ไกลขึ้น 4 เท่า
`sll $s1,$s2,4` →
000000·00000·10010·10001·00100·000000 = **0x00128900**
(R-type ที่ rs = 0 และใช้ shamt)

คำถาม 6 — Bootstrapping COE 2019 (รูป 4 + 5) (10 pt)



รูป 4 — Gate A1: $Y_{A1} = \bar{A}_{12}\cdot\bar{A}_{13}\cdot\ldots\cdot\bar{A}_{31}$ (เป็น 1 ก็ต่อเมื่อ A[31:12] เป็น 0 ทั้งหมด) · Big Endian · Bootstrapping Address = 0x00000000

ขั้นที่ 1 — ถอด Address Map ของ COE 2019

กล่อง	CS มาจาก	ช่วง address	ขนาด
Boot Mem	CS1 = CS0 = Y_{A1} (active-HIGH) ⇒ A[31:12] = 0 ทั้งหมด	0x0000 0000 – 0x0000 0FFF	4 KB A[11-0]
Main Memory	CS1 = Y_{A1} ผ่าน ○, CS0 = A[31] ผ่าน ○ ⇒ Y_{A1} = 0 **และ** A[31] = 0	0x0000 1000 – 0x7FFF FFFF	≈2 GB A[30-0]
Dev 0	Decoder เปิดเมื่อ CS1=CS0=A[31]=1 X1 = A[30], X0 = A[29] Y0→Dev0 Y1→Dev1 Y2→Dev2 Y3→Dev3	0x8000 0000 – 0x9FFF FFFF	ตัวละ 512 MB A[28-0]
Dev 1		0xA000 0000 – 0xBFFF FFFF	
Dev 2		0xC000 0000 – 0xDFFF FFFF	
Dev 3		0xE000 0000 – 0xFFFF FFFF	

★ ทริกจำ 5 วินาที (รูป 4)
`0x00000000–0FFF = Boot (4 KB แรกเท่านั้น)`
`หลักแรก 0–7 ที่เหลือ = Main` `8,9 = Dev0` `A,B = Dev1` `C,D = Dev2` `E,F = Dev3`
ระวัง: map ของรูป 4 **ไม่เหมือน** รูป 3 — อย่าจำสลับกัน!

ขั้นที่ 2 — อ่าน Boot Mem ทีละ 4 ไบต์ (Big Endian)

Big Endian
ไบต์ที่ address **ต่ำสุด** = ไบต์ที่ **มีนัยสำคัญสูงสุด** (ซ้ายสุดของ word) ⇒ `3C 15 A0 00` ที่ 0x0–0x3 = word **0x3C15A000**

รูป 5 · Boot Mem	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
...0000	3C	15	A0	00	34	10	00	27	AE	B0	00	00	8E	B1	00	04
...0010	08	10	00	14	27	A5	00	04	8F	A4	00	00	00	04	10	80
...0020	00	C2	30	21	0C	10	00	09	34	02	00	0A	3C	15	10	00

Address	4 ไบต์	Machine code	กางบิต	MIPS Assembly
0x0000 0000	3C 15 A0 00	**0x3C15A000**	001111·00000·10101·A000 op=0x0F=lui, rt=21=$s5	`lui $s5, 0xA000` ⇒ $s5 = 0xA000 0000
0x0000 0004	34 10 00 27	**0x34100027**	001101·00000·10000·0027 op=0x0D=ori, rs=$zero, rt=16=$s0	`ori $s0, $zero, 0x0027` ⇒ $s0 = 0x27
0x0000 0008	AE B0 00 00	**0xAEB00000**	101011·10101·10000·0000 op=0x2B=sw, rs=$s5, rt=$s0	`sw $s0, 0($s5)` ⇒ เขียน 0x27 ไป 0xA0000000
0x0000 000C	8E B1 00 04	**0x8EB10004**	100011·10101·10001·0004 op=0x23=lw, rs=$s5, rt=17=$s1	`lw $s1, 4($s5)` ⇒ อ่านจาก 0xA0000004
0x0000 0010	08 10 00 14	**0x08100014**	000010·0 0001 0000 0000 0000 0001 0100 field = 0x0100014 → ×4 = 0x00400050	`j 0x00400050`

ขั้นที่ 3 — ตอบข้อ 6.6 – 6.8

ข้อ	คำถาม	คำตอบ + เหตุผล
6.6	$s0 หลัง 5 คำสั่ง	**0x0000 0027** — `ori` ใส่ 0x27 ลง $s0 · `lui` ลง $s5 · `lw` ลง $s1 ⇒ ไม่มีใครแตะ $s0 อีก
6.7	คำสั่งที่ 6 อยู่ที่ไหน	**0x0040 0050** — เพราะคำสั่งที่ 5 คือ `j` ⇒ ไม่ไปต่อที่ 0x14
6.8	คำสั่งไหนติดต่อ I/O	**0x0000 0008** → **Dev 1** และ **0x0000 000C** → **Dev 1** เพราะ $s5 = 0xA000 0000 ⇒ hex หลักแรก **A** ⇒ ตกช่วง Dev 1 (memory-mapped I/O)

⚠ กับดักข้อ 6.8

คำสั่งอีก 3 ตัว (`lui`, `ori`, `j`) เป็นการคำนวณล้วน **ไม่แตะ data bus** · การ fetch คำสั่งเองอ่านจาก Boot Mem **ไม่นับเป็น I/O**

⚡ ถ้าพลิกโจทย์ข้อ 6

เปลี่ยนเป็น Little Endian → `3C 15 A0 00` กลายเป็น **0x00A0153C** ⇒ op = 000000 = R-type ⇒ คำสั่งเปลี่ยนหมด ต้องถอดใหม่ทุกบรรทัด

เปลี่ยน Bootstrapping Address เป็น 0x00000010 → คำสั่งแรกคือ `j 0x00400050` ทันที

เปลี่ยน `lui $s5,0xA000` → `0xC000` → ปลายทาง 0xC000 0000 ⇒ หลักแรก C ⇒ **Dev 2** แทน · เป็น `0x8000` ⇒ **Dev 0**

ถามค่า `$s1` → บอกไม่ได้ เพราะเป็นค่าที่อ่านมาจาก Dev 1 ซึ่งโจทย์ไม่ได้ให้

ถามค่า `$s5` → 0xA000 0000 (`lui` = ยกค่า 16 บิตขึ้นไปครึ่งบน ครึ่งล่างเป็น 0)

ถาม "คำสั่งไหนติดต่อ Main Memory" → ไม่มีเลย ใน 5 คำสั่งแรก

ถามคำสั่งที่ 6, 7 (สมมติไม่มี `j`) → 0x14 = `27 A5 00 04` = 0x27A50004 → op 001001 = `addiu $a1,$sp,4` · 0x18 = `8F A4 00 00` = 0x8FA40000 → `lw $a0,0($sp)`

เปลี่ยน Gate A1 เป็นรับ A[31-16] → Boot Mem โตเป็น **64 KB** (0x0000 0000–0x0000 FFFF) และ Main Mem เริ่มที่ 0x0001 0000

เปลี่ยน Gate A1 เป็น OR แทน AND → เงื่อนไขกลับด้าน ต้องคิดใหม่จากนิยาม Y_{A1} ที่โจทย์ให้เท่านั้น

"ทำไม Boot Mem ต้อง non-volatile" → ตอนเปิดเครื่อง RAM ยังว่าง ต้องมีโปรแกรมค้างอยู่จริงเพื่อเริ่มระบบ

"โปรแกรม boot นี้ทำอะไร" → ตั้งค่า base ของ Dev 1 (`lui`) → ส่งคำสั่ง/ค่า 0x27 ไปที่อุปกรณ์ (`sw`) → อ่านค่าตอบกลับ (`lw`) → กระโดดไปรันโปรแกรมหลักที่ 0x00400050 ใน Main Memory

★ ถ้าเจอโจทย์ COE รุ่นใหม่ ทำตาม 6 ขั้นนี้

① เขียน address map จากรูปก่อนเสมอ (CS + bubble + X1X0)
② จด bootstrapping address ③ ตัดไบต์เป็นก้อนละ 4 ตาม endian ④ แปลง hex → 32 บิต ⑤ ดู op 6 บิตแรก → รู้ format → กางฟิลด์ ⑥ ตามรอย register ทีละคำสั่ง แล้วเช็คทุก `lw/sw` ว่า address ตกช่วงไหน

สรุปกับดักที่คนพลาดบ่อยที่สุด

① ตอบ 1.5/1.6 แค่ "CPU" หรือ "Memory" แต่ **ไม่บอกว่าค่าบน data bus ระบุไม่ได้** ⇒ เสียครึ่งคะแนน
② คิดว่า arbiter เขียน bus ได้ตลอด (ข้อ 2.3)
③ ข้อ 2.5–2.7 ตอบว่า "อ่านได้เฉพาะตัวที่ไม่ได้เขียน" — ที่ถูกคือ **ทุกตัว รวมตัวที่เขียนเอง**
④ ข้อ 3 ช่อง (d) ตอบ address+4 — โจทย์บอก "**ก่อนจะโหลด**" ⇒ ยังไม่ได้ทำ ⇒ กลับที่เดิม
⑤ ลืมว่า `lw/sw` เรียง **rs ก่อน rt** ในเครื่อง
⑥ คิด branch offset เป็นไบต์ (ลืมหาร 4) หรือลืมบวก **PC+4**
⑦ `j` ลืม **หาร 4** ก่อนใส่ field 26 บิต
⑧ ใช้ `li` / `move` / `blt` ในข้อ 4 ⇒ โดนหักข้อละ 1 คะแนน
⑨ อ่าน Big Endian กลับด้าน
⑩ ลืมว่า **Boot Mem ในรูป 4 มีแค่ 4 KB** (0x0–0xFFF) ไม่ใช่ทั้ง region 0
⑪ จำ address map ของ **รูป 3 สลับกับรูป 4** — คนละวงจร คนละ map
⑫ ลืมว่า array int ต้องคูณ index ด้วย **4**

ภาคผนวก — ของแถมที่ออกสอบได้

ตารางกำลังของ 2 (ใช้ตอบ "อ้างหน่วยความจำได้เท่าไร")

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

ตัวอย่างถอด hex → Assembly ทีละขั้น (ถ้าโจทย์กลับด้าน)

ตัวอย่าง ① 0x02328020
กาง 32 บิต: `000000 10001 10010 10000 00000 100000`
op = **000000** ⇒ R-type ⇒ ไปดู funct 6 บิตท้าย = **100000** = 0x20 = add
rs = 10001 = 17 = `$s1` · rt = 10010 = 18 = `$s2` · rd = 10000 = 16 = `$s0`
⇒ **`add $s0, $s1, $s2`** (R-type เรียงในเครื่อง rs·rt·rd แต่เขียน rd·rs·rt)

ตัวอย่าง ② 0x1230FFFC (สมมติอยู่ที่ 0x00400020)
กาง: `000100 10001 10000 1111111111111100`
op = 000100 = 0x04 = `beq` · rs = `$s1` · rt = `$s0` · imm = 0xFFFC = **−4**
ปลายทาง = (PC + 4) + (−4 × 4) = 0x00400024 − 16 = **0x00400014** (กระโดดย้อนกลับ)

คำศัพท์ที่ต้องอธิบายเป็นคำพูดตัวเองได้

คำ	ความหมายสั้น
Volatile	ข้อมูลหายเมื่อดับไฟ (RAM / Main Memory)
Non-volatile	ข้อมูลอยู่ต่อเมื่อดับไฟ (ROM / Boot Mem)
Tri-state	เอาต์พุตมี 3 สถานะ: 0, 1, และ **Hi-Z** (ลอย ไม่ขับ)
Hi-Z	สถานะไม่ขับสัญญาณ ปล่อยให้ตัวอื่นใช้ bus ได้
Chip Select (CS)	ขาที่บอกว่า "ชิปนี้ถูกเลือกให้ทำงาน" ○ = active-low
Decoder	แปลงเลข n บิต เป็นสายเอาต์พุต 2^n เส้น เลือกทีละเส้น
Memory-mapped I/O	อุปกรณ์ถูกจัดสรร address ปนกับหน่วยความจำ ใช้ `lw/sw` ปกติ
Isolated I/O	แยกพื้นที่ I/O ออกจาก memory ต้องใช้คำสั่งเฉพาะ (`in/out`)
Bus master	ตัวที่มีสิทธิควบคุม/ขับ bus ในจังหวะนั้น
Arbiter	ตัวตัดสินว่าใครได้เป็น bus master
ISR	โปรแกรมย่อยที่ทำงานเมื่อเกิด interrupt
Interrupt vector	ค่า address ที่ CPU กระโดดไปเมื่อรับ interrupt
Bootstrapping	โปรแกรมชุดแรกหลังเปิดเครื่อง อยู่ใน non-volatile memory
Latency / Bandwidth	เวลาที่ต้องรอ / ปริมาณข้อมูลต่อวินาที

ส่วนขยาย: เรียกฟังก์ชัน (เผื่อออก)

```
      jal  FUNC        # $ra = PC+4 แล้วกระโดด
      ...
FUNC: addi $sp,$sp,-4  # จองที่ใน stack
      sw   $ra,0($sp)
      ...
      lw   $ra,0($sp)
      addi $sp,$sp,4
      jr   $ra         # กลับที่เดิม
```

ต่างกันตรงไหน
`j` ไปแล้วไม่กลับ · `jal` จำที่กลับไว้ใน `$ra` · `jr $ra` ใช้ค่าใน register เป็นปลายทาง จึงกระโดดได้ **ทั่วทั้ง 4 GB** (ไม่ติดข้อจำกัด 256 MB ของ `j`)

โจทย์ซ้อมมือ 10 ข้อ (เฉลยอยู่ในตาราง)

#	โจทย์	เฉลย
1	address bus 24 บิต, data 8 บิต อ้างได้กี่ไบต์	**2^{24} = 16 MB**
2	`lw $t1, 8($s2)` → hex	**0x8E49 0008** `100011·10010·01001·0008`
3	`beq $s0,$s1,L` ที่ 0x00400100, L = 0x004000F0	offset = (0xF0−0x104)/4 = **−5** = 0xFFFB **0x1211 FFFB**
4	`sub $s3,$s4,$s5` → hex	**0x0295 9822** `000000·10100·10101·10011·00000·100010`
5	`j 0x00500000` → hex	0x500000÷4 = 0x140000 ⇒ **0x0814 0000**
6	address 0xB000 0000 คืออุปกรณ์อะไร	**รูป 3** → **DMA** · **รูป 4** → **Dev 1**
7	address 0x0000 0800 (รูป 4)	**Boot Mem** (อยู่ใน 4 KB แรก)
8	Little Endian ของไบต์ `3C 15 A0 00`	**0x00A0 153C**
9	interrupt ก่อนโหลด 0xD0001000, vector 0xF0000000 (รูป 3)	(a)(b) 0xF0000000 · (c) **Dev 2** ⇒ ผิดปกติ ISR ไม่ควรอยู่ I/O · (d) 0xD0001000
10	`a = A[3];` ($s0 = a, $s3 = base)	`lw $s0, 12($s3)`

เทียบวงจร รูป 3 กับ รูป 4 — อย่าจำสลับกัน

	รูป 3 (คำถาม 3)	รูป 4 — COE 2019 (คำถาม 6)
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	**ไม่มี DMA** ในรูปนี้
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
interrupt ได้	Dev 2, DMA	โจทย์ไม่ถาม (ไม่มีสาย Int ในรูป)

คำสั่ง MIPS เพิ่มเติมที่อาจโผล่มา

คำสั่ง	ชนิด	ความหมาย
`lb / lbu rt,off(rs)`	I 0x20/0x24	โหลด 1 ไบต์ (มี/ไม่มีเครื่องหมาย)
`lh / sh`	I 0x21/0x29	โหลด/เก็บ 2 ไบต์
`sb rt,off(rs)`	I 0x28	เก็บ 1 ไบต์
`slti rt,rs,imm`	I 0x0A	rt = (rs < imm)
`sra rd,rt,sh`	R 0x03	เลื่อนขวาแบบคงเครื่องหมาย
`sllv / srlv`	R 0x04/0x06	เลื่อนตามค่าใน register
`mult / div`	R 0x18/0x1A	ผลลง hi, lo
`mfhi / mflo rd`	R 0x10/0x12	ดึงค่าจาก hi / lo
`jalr rs`	R 0x09	เรียกฟังก์ชันด้วย address ใน register

Σ สูตรทั้งหมดในกล่องเดียว

ความจุ = 2^(บิต address) × (ไบต์ต่อ 1 ช่อง)
ขนาดอุปกรณ์จากรูป = 2^(จำนวนบิตของสาย Addr ที่ต่อเข้ากล่อง)
Branch imm = (addr(label) − addr(branch) − 4) ÷ 4 (16 บิต signed)
Branch ปลายทาง = addr(branch) + 4 + imm × 4
Jump field = addr(target) ÷ 4 (26 บิตล่าง)
Jump ปลายทาง = (PC+4)[31:28] ‖ field ‖ 00
offset ของ A[k] = k × sizeof(ชนิดข้อมูล)
Big Endian word = byte[a]·byte[a+1]·byte[a+2]·byte[a+3] เรียงซ้าย → ขวา
2's complement = 0x10000 − |n| (16 บิต)

★ ลำดับการทำข้อสอบที่แนะนำ

① ข้อ 5 + 6 ก่อน (คำนวณล้วน ได้คะแนนแน่) ② ข้อ 1 + 2 (ทฤษฎีสั้น) ③ ข้อ 3 (วาด address map ก่อนตอบ) ④ ข้อ 4 (เขียนโค้ด ใช้เวลามากสุด)
ก่อนส่ง: เช็ค 3 อย่าง — ① เผลอใช้ pseudo-instruction ไหม ② หาร 4 ครบทุก branch/jump ไหม ③ ตอบครบทุกช่อง (a)(b)(c)(d) ไหม