ชีทรวม — สรุป + เฉลยข้อสอบ A01

198322 สถาปัตยกรรมคอมพิวเตอร์ · ครึ่งแรก = สรุปทฤษฎี/สูตร · ครึ่งหลัง = เฉลยข้อสอบทุกข้อ · 2 แผ่น 4 หน้า

SUMMARY + KEY

ส่วนที่ 1 · สรุปทฤษฎี & สูตร

1.1 Bus & ทิศทางสัญญาณ

สาย	ทิศทาง	หมายเหตุ
Address bus	**ทางเดียว** CPU → Mem/IO	CPU เป็นคนกำหนดช่องเสมอ
Data bus	**สองทาง**	read = Mem ขับ · write = CPU ขับ
Read / Write	**ทางเดียว** CPU → Mem/IO	สายควบคุม

กฎที่ต้องตอบให้ครบ 2 ส่วน

ใครขับ data bus: read active ⇒ **Memory/อุปกรณ์** · write active ⇒ **CPU**

ค่าเป็นเท่าไร: ระบุไม่ได้ ถ้าโจทย์ไม่บอกเนื้อในหน่วยความจำ (address บอกได้เสมอ)

read=write=0 ⇒ ไม่มีใครขับ = **Hi-Z / floating** · read=write=1 ⇒ **bus conflict**

สูตรความจุ

จำนวนช่อง = **2^(บิต address)** · ความจุ = 2^n × (data bus ÷ 8) ไบต์ · ถ้า data 8 บิต ⇒ **2^n ไบต์**

ขนาดกล่องในรูป = 2^(จำนวนบิตสาย Addr) · Bandwidth = ความถี่ × (data ÷ 8) · รอบส่ง = ⌈N/M⌉

2^5=32 · 2^6=64 · 2^{12}=4K · 2^{16}=64K · 2^{20}=1M · 2^{28}=256M · 2^{29}=512M · 2^{32}=4G

1.2 Arbitration & Bus Contention

เขียน bus = ขับสัญญาณ ⇒ ได้ **ทีละ 1 ตัว** คือตัวที่ถือสิทธิ · **arbiter =/คนที่เขียนได้ตลอด**

อ่าน bus = แค่วัดแรงดัน ⇒ **ทุกตัวอ่านได้เสมอ** รวมตัวที่กำลังเขียน

ขนาด bus = นับเส้นสัญญาณในรูป

เขียนพร้อมกัน = Bus contention (ตอบ 4 ชั้น)

① ชื่อ: **bus contention / conflict** ② ข้อมูล: แรงดันค้างกลาง ⇒ ค่า **undefined** อ่านได้ขยะ

③ ฮาร์ดแวร์: ตัวหนึ่งดัน V_{DD} อีกตัวดึง GND ⇒ **ลัดวงจร** ร้อน ชิปพัง

④ ป้องกัน: arbiter + **tri-state** (ตัวอื่น Hi-Z) หรือ open-drain (wired-AND ไม่พังแต่ข้อมูลยังผิด)

1.3 สูตรถอด Address Map จากรูป (4 ขั้น)

① ดูขา CS ของทุกกล่อง ว่ารับสายอะไร และมี **วงกลม** ○ ไหม — มี ○ = ต้องเป็น **0** · ไม่มี ○ = ต้องเป็น **1** · หลายขา = ต้องครบทุกขา

② Decoder ทำงานเมื่อ CS ครบ แล้ว X1X0 เลือก Y0–Y3 (00→Y0, 01→Y1, 10→Y2, 11→Y3)

③ ไล่เส้น Y ว่าไปเข้า CS ของกล่องไหน (สายในรูปมักไขว้กัน)

④ กล่องรับ A[k-0] ⇒ ขนาด = 2^(k+1) ไบต์ · บิตบนที่เหลือคือรหัสประจำตัว

Gate Y = $\bar{A}_k \cdot \ldots \cdot \bar{A}_{31}$ ⇒ เป็น 1 เมื่อ A[31:k]=0 หมด ⇒ ครอบคลุม 0 … 2^k−1

★ เทคนิคเร็ว

hex หลักแรก = A[31:28] ⇒ **0–7 ⇒ A[31]=0** · **8–F ⇒ A[31]=1** · แต่ละหลัก hex กิน 256 MB

1.4 Interrupt / Polling / DMA

ใคร interrupt ได้: เฉพาะกล่องที่มีขา `Int` และมีสายวิ่งเข้า OR-gate ที่ต่อกับขา `Interrupt` ของ CPU

โจทย์ "ค้างที่ X, vector = V": (a) = **V** · (b) ISR อยู่ที่ = **V** · (c) = เอา V เทียบ address map · (d) = **X** (ถ้าโจทย์ว่า "ก่อนจะโหลด") หรือ **X+4** (ถ้า "ทำเสร็จแล้ว")

ไม่มี interrupt ⇒ Polling: CPU วนอ่าน status register ซ้ำๆ ตรวจบิต ready/done · เสีย = busy-wait กิน CPU, latency สูง · ดี = ฮาร์ดแวร์ง่าย เวลาคาดเดาได้

DMA: ย้ายข้อมูล I/O ↔ memory เองโดยไม่ผ่าน CPU แล้ว interrupt บอกเมื่อเสร็จ

volatile = หายเมื่อดับไฟ (RAM) · **non-volatile** = ไม่หาย (Boot ROM)

Memory-mapped I/O: I/O ใช้ address ปนกับ memory ⇒ ติดต่อด้วย `lw/sw` ธรรมดา · การ fetch คำสั่งและคำสั่งคำนวณ **ไม่นับเป็น I/O**

1.5 MIPS-32 — รูปแบบคำสั่ง

⚠ ลำดับฟิลด์สลับกับที่เขียน

`add rd,rs,rt` → เครื่องเรียง **rs·rt·rd** · `lw rt,off(rs)` → เครื่องเรียง **rs·rt** (ตัวในวงเล็บมาก่อน!)

R	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
jr/jalr	0x08/0x09

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

★ ท่อง

$sN = **16+N** · $tN = **8+N** · op ที่ต้องจำ: lw **100011** · sw **101011** · addi **001000** · beq **000100** · j **000010** · R **000000**

1.6 สูตรเข้ารหัส/ถอดรหัส

Branch imm = (addr(label) − addr(branch) − 4) ÷ 4 (16 บิต signed)

label อยู่ข้างหน้า m คำสั่ง ⇒ imm = **m − 1** · อยู่ข้างหลัง k คำสั่ง ⇒ imm = **−(k+1)**

Branch ปลายทาง = addr(branch) + 4 + imm × 4

Jump field = addr(target) ÷ 4 (26 บิตล่าง) · **ปลายทาง** = (PC+4)[31:28] ‖ field ‖ 00

2's comp 16 บิต = 0x10000 − |n| ⇒ −1=0xFFFF · −4=0xFFFC · −5=0xFFFB · −8=0xFFF8

ถอดรหัส: hex→32 บิต → op 6 บิตแรก: **000000=R** (ดู funct) · **000010/11=J** · อื่น=**I**

ย้ายโปรแกรม: beq/bne **ไม่เปลี่ยน** (สัมพัทธ์) · `j/jal` **เปลี่ยน** (สัมบูรณ์)

ระยะ: `j` = 256 MB ใน region เดียว · `beq` = ±32768 คำสั่ง · `jr` = 4 GB

1.7 C → MIPS & ห้ามใช้ pseudo

Pseudo	ใช้แทน
`li $s0,N`	`addi $s0,$zero,N`
`move $s0,$s1`	`add $s0,$s1,$zero`
`la $s0,L`	`lui + ori`
`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`
`not/nop`	`nor $s0,$s1,$zero / sll $zero,$zero,0`

หลักคิด

① if/while ⇒ ตรวจ **นิเสธ** แล้ว branch ข้าม ② array `A[k]` ⇒ offset = **k × 4** (int) · index เป็นตัวแปรต้อง `sll $t0,$s2,2` ก่อน ③ `a*8` ⇒ `sll ,3` · `a/4` ⇒ `srl ,2`

```
# if (a<b){a=b;} a++;
      slt  $t0,$s0,$s1
      beq  $t0,$zero,ENDIF
      add  $s0,$s1,$zero
ENDIF:addi $s0,$s0,1

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

1.8 ลอจิก & เลขฐาน ที่ต้องใช้จริง

X1	X0	Decoder 2→4
0	0	**Y0=1** ที่เหลือ 0
0	1	**Y1=1**
1	0	**Y2=1**
1	1	**Y3=1**
CS ไม่ครบ		**ทุกเส้น = 0**

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

รูปแบบ CS ที่เจอบ่อย	ถูกเลือกเมื่อ
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

2's complement & กำลังของ 2

−1=0xFFFF · −2=0xFFFE · −3=0xFFFD · −4=**0xFFFC** · −5=0xFFFB · −8=0xFFF8 · −16=0xFFF0 · −32=0xFFE0

2^5=32 · 2^6=64 · 2^{10}=1K · 2^{12}=4K · 2^{16}=64K · 2^{20}=1M · 2^{24}=16M · 2^{28}=256M · 2^{29}=512M · 2^{30}=1G · 2^{32}=4G

÷4 = เลื่อนขวา 2 บิต · address ที่หาร 4 ลงตัวต้องลงท้ายด้วย **0, 4, 8, C** เท่านั้น

label ข้างหน้า m คำสั่ง	imm
m=1	0
m=2	1
m=3	**2**
m=4	3
ทั่วไป	**m−1**

label ข้างหลัง k คำสั่ง	imm
k=1	−2 = FFFE
k=2	−3 = FFFD
k=3	−4 = FFFC
k=4	−5 = FFFB
ทั่วไป	**−(k+1)**

1.9 ตารางสำเร็จรูป — machine code ที่ใช้บ่อย

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

★ ตรวจเร็ว

hex หลักแรกบอกชนิด: **0**=R · **08/09**=j/jal · **1**=beq/bne · **2**=addi · **3**=ori/lui · **8**=lw · **A**=sw

I-type: 4 หลักแรก = op·rs·rt (เช่น `lw $s0,_($s1)` = **0x8E30**) · 4 หลักหลัง = offset ตรงๆ

1.10 · 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 ที่เป็นจริงถึงจะเปลี่ยนเป็นค่าอื่น

1.11 · 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

1.12 · Recipe — 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

1.13 · โค้ดแม่แบบ C → MIPS (ลอกไปใช้ได้เลย)

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:
```

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
      add  $t0,$t0,$s3
      lw   $t2,0($t0)
      add  $s0,$s0,$t2
      addi $s2,$s2,1
      j    FOR
ENDF:
```

do – while

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

swap

```
add  $t0,$s0,$zero
add  $s0,$s1,$zero
add  $s1,$t0,$zero
```

เรียกฟังก์ชัน

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

1.14 · เทียบวงจร รูป 3 กับ รูป 4 — จุดที่คนจำสลับกันที่สุด

	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	0x0000 0000 – 0x7FFF FFFF	0x0000 **1000** – 0x7FFF FFFF
Boot Mem	0xC000 0000 – 0xCFFF FFFF (256 MB)	0x0000 0000 – 0x0000 0FFF (**4 KB**)
DMA	0x8000 0000 – 0xBFFF FFFF	ไม่มีในรูปนี้
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

★ วิธีจำที่ปลอดภัยที่สุด

อย่าท่องช่วง address — ท่อง **วิธีถอดวงจร** แล้วถอดสดจากรูปที่โจทย์ให้ ใช้เวลาไม่ถึง 2 นาที และไม่มีทางจำสลับ

1.15 · โจทย์ซ้อมมือ (เฉลยในตาราง)

#	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**
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	`sll $s0,$s1,2` → hex	**0x0011 8080** (R-type: rs = 0, shamt = 2)
10	`a = A[3];` ($s0 = a, $s3 = base)	`lw $s0, 12($s3)`

1.16 · 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 เส้นพร้อมกัน	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 ใหม่ถูกยึดมาจาก 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

ส่วนที่ 2 · เฉลยข้อสอบ A01 ทุกข้อ

คำถาม 1 — Bus (10 pt)

ข้อ	คำตอบ
1.1	Address bus → **CPU A ไป Memory** (ทางเดียว)
1.2	Data bus → **สองทาง**
1.3	Read → **CPU A ไป Memory**
1.4	Write → **CPU A ไป Memory**
1.7	addr 5 บิต, data 8 บิต → **32 ไบต์** (2^5)
1.8	addr 6 บิต, data 8 บิต → **64 ไบต์** (2^6)

1.5 (อธิบาย) read active, addr 0x20004000

Memory เป็นตัวเขียน (ขับ) ลง data bus, CPU เป็นฝ่ายอ่าน
ค่าบน data bus ระบุไม่ได้ — เป็นข้อมูล 32 บิตที่เก็บอยู่ที่ช่อง 0x20004000 ซึ่งโจทย์ไม่ได้ให้
เขียนเสริม: ตอนนี้ CPU ต้องปล่อย data bus เป็น Hi-Z ไม่งั้นชนกัน

1.6 (อธิบาย) write active, addr 0x20008000

CPU A เป็นตัวเขียนลง data bus, Memory เป็นฝ่ายรับไปเก็บที่ช่อง 0x20008000
ค่าระบุไม่ได้ — เป็นข้อมูลที่โปรแกรมสั่งเขียน โจทย์ไม่ได้ให้

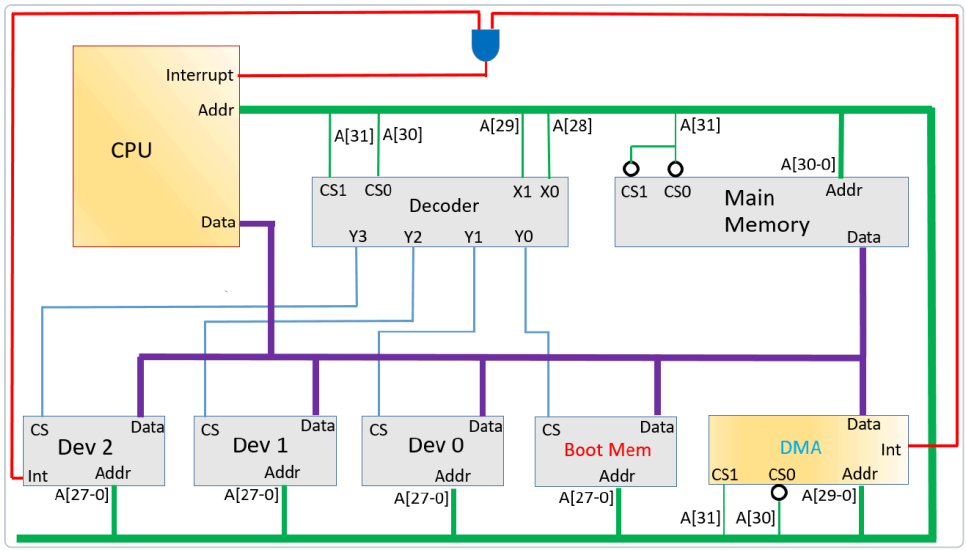
คำถาม 2 — Bus Arbitration (10 pt)

ข้อ	คำตอบ	ข้อ	คำตอบ
2.1	**8 บิต**	2.5	**A, B, C**
2.2	**A** เท่านั้น	2.6	**A, B, C**
2.3	**B** เท่านั้น	2.7	**A, B, C**
2.4	**C** เท่านั้น	—	อ่านได้ทุกตัวเสมอ

2.8 (อธิบาย) ถ้า A, B, C เขียนพร้อมกัน

เกิด **Bus contention / bus conflict** — driver หลายตัวขับเส้นเดียวกันคนละลอจิก
⇒ แรงดันเป็นค่า **undefined** ทุกตัวอ่านได้ข้อมูลผิด · เกิด **ลัดวงจร** จาก V_{DD} ลง GND ผ่าน driver ⇒ ร้อน กินไฟ ชิปพังได้
⇒ ป้องกันด้วย arbiter + **tri-state buffer** ให้ตัวที่ไม่ได้สิทธิอยู่ **Hi-Z**

คำถาม 3 — Interrupt + Memory Map (10 pt)



รูป 3 — Main Memory volatile · Boot Mem non-volatile

hex หลักแรก	อุปกรณ์	เงื่อนไข
0–7	**Main Memory**	A[31]=0 (CS ○○ ← A[31])
8–B	**DMA**	A[31]=1, A[30]=0
C	**Boot Mem**	A[31]=A[30]=1 เปิด decoder
D	**Dev 0**	X1X0 = A[29]A[28]
E	**Dev 1**	Y0→Boot Y1→Dev0
F	**Dev 2**	Y2→Dev1 Y3→Dev2
		ตัวละ 256 MB (A[27-0])

3.1
DMA และ Dev 2 — เป็นสองกล่องเดียวที่มีขา `Int` ต่อสายแดงเข้า OR-gate ของ CPU

3.2 (อธิบาย) ถ้าไม่มี interrupt
ใช้ **Polling** — CPU วนลูปอ่าน **status register** ของอุปกรณ์ผ่าน memory-mapped I/O ซ้ำๆ แล้วตรวจบิต ready/done จนกว่าจะขึ้น

```
poll: lw   $t0,0($s0)
      andi $t0,$t0,1
      beq  $t0,$zero,poll
```

ข้อเสีย: busy-waiting กิน CPU cycle ทำงานอื่นไม่ได้ · latency ขึ้นกับคาบ poll · ไม่ scale
ข้อดี: ฮาร์ดแวร์ง่าย ไม่ต้องมีสาย Int / ISR / vector table

ช่อง	3.3	3.4	3.5
ค้างที่	0x00001000	0xCFFE1000	0x00410080
vector	0xCFFF0000	0x00010000	0x05000000
(a)	**0xCFFF0000**	**0x00010000**	**0x05000000**
(b)	**0xCFFF0000**	**0x00010000**	**0x05000000**
(c)	**Boot Mem**	**Main Mem**	**Main Mem**
(d)	**0x00001000**	**0xCFFE1000**	**0x00410080**

⚠ เฉลยอาจารย์ข้อ 3.4 (a) พิมพ์ 0x00001000
เป็นการพิมพ์ผิด (ก๊อปจาก 3.3) — (a) ต้องเท่ากับ vector เสมอ ⇒ ที่ถูกคือ **0x00010000**

คำถาม 4 — เขียน MIPS แทน C (10 pt)

$s0=a · $s1=b · $s2=i · $s3=base ของ A (int 4 ไบต์) · **ห้าม pseudo-instruction**

```
(a) a = A[0] + 32;
lw   $t0, 0($s3)
addi $s0, $t0, 32

(b) A[1] = a;
sw   $s0, 4($s3)

(c) a = -4;
addi $s0, $zero, -4

(d) if (a<b){a=b;} a++;
      slt  $t0, $s0, $s1
      beq  $t0, $zero, ENDIF
      add  $s0, $s1, $zero
ENDIF:
      addi $s0, $s0, 1

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

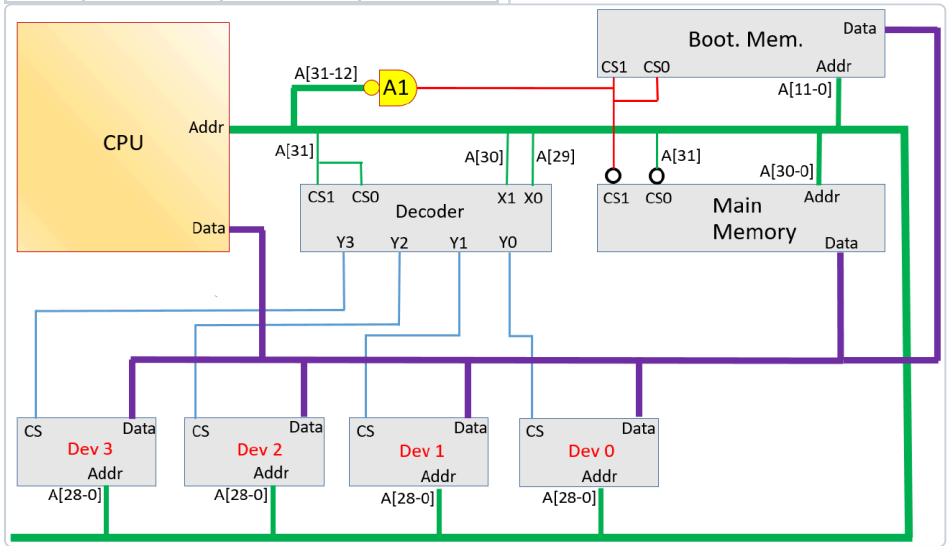
คำถาม 5 — Assembly → Machine code (10 pt)

คำสั่ง	การกางบิต	hex
`lw $s0,4($s1)`	100011·10001·10000·0004	**0x8E30 0004**
`sw $s0,-4($s1)`	101011·10001·10000·FFFC	**0xAE30 FFFC**

addr	Assembly	5.3 (เริ่ม 0x00400054)	5.4 (เริ่ม 0x00408054)
...054	`lab0: beq $t0,$s1,lab1`	**0x1111 0002**	**0x1111 0002**
...058	`addi $s1,$s1,1`	**0x2231 0001**	**0x2231 0001**
...05C	`j lab0`	**0x0810 0015**	**0x0810 2015**
...060	`lab1: add $s2,$s1,$zero`	**0x0220 9020**	**0x0220 9020**

วิธีทดที่ต้องเขียนลงกระดาษ
beq: offset = (...060 − (...054+4)) ÷ 4 = 8 ÷ 4 = **2**
j (5.3): 0x00400054 ÷ 4 = 0x0100015 ⇒ 000010‖0100015 = 0x08100015
j (5.4): 0x00408054 ÷ 4 = 0x0102015 ⇒ 0x08102015
add: 000000·10001·00000·10010·00000·100000
ประเด็นที่วัด: ย้ายโปรแกรมแล้วเปลี่ยนแค่ `j` เพราะ `beq` เป็น offset **สัมพัทธ์**

คำถาม 6 — Bootstrapping COE 2019 (10 pt)



รูป 4 — $Y_{A1} = \bar{A}_{12}\cdot\ldots\cdot\bar{A}_{31}$ (เป็น 1 เมื่อ A[31:12] = 0 หมด) · Big Endian · Boot address = 0x00000000

กล่อง	เงื่อนไข CS	ช่วง address
Boot Mem	CS1=CS0=Y_{A1} ⇒ A[31:12]=0	0x0000 0000–0x0000 0FFF (**4 KB**)
Main Memory	$\circ Y_{A1}$ และ ○A[31] ⇒ ทั้งคู่ = 0	0x0000 1000–0x7FFF FFFF
Dev 0	Decoder เปิดเมื่อ A[31]=1 X1=A[30], X0=A[29] ตัวละ 512 MB (A[28-0])	0x8000 0000–0x9FFF FFFF
Dev 1		0xA000 0000–0xBFFF FFFF
Dev 2		0xC000 0000–0xDFFF FFFF
Dev 3		0xE000 0000–0xFFFF FFFF

★ ทริก (รูป 4)
0x0–0xFFF = Boot · หลักแรก 0–7 ที่เหลือ = Main · **8,9**=Dev0 · **A,B**=Dev1 · **C,D**=Dev2 · **E,F**=Dev3

ข้อ	Address	Machine code	MIPS Assembly
6.1	**0x00000000**	**0x3C15A000**	`lui $s5, 0xA000` ⇒ $s5 = 0xA0000000
6.2	**0x00000004**	**0x34100027**	`ori $s0, $zero, 0x0027` ⇒ $s0 = 0x27
6.3	**0x00000008**	**0xAEB00000**	`sw $s0, 0($s5)` ⇒ เขียนไป 0xA0000000
6.4	**0x0000000C**	**0x8EB10004**	`lw $s1, 4($s5)` ⇒ อ่านจาก 0xA0000004
6.5	**0x00000010**	**0x08100014**	`j 0x00400050` (0x0100014 × 4)
6.6	$s0 = **0x00000027**		เพราะ `ori` ใส่ลง $s0 · lui ลง $s5 · lw ลง $s1
6.7	คำสั่งที่ 6 = **0x00400050**		เพราะคำสั่งที่ 5 เป็น `j`
6.8	**0x00000008 → Dev 1** **0x0000000C → Dev 1**		$s5 = 0xA000 0000 ⇒ หลักแรก **A** ⇒ Dev 1 `lui/ori/j` ไม่แตะ bus · fetch ไม่นับเป็น I/O

Big Endian — วิธีตัด word
ไบต์ที่ address ต่ำสุดคือไบต์ซ้ายสุด ⇒ `3C 15 A0 00` = **0x3C15A000** · ถ้าเป็น Little Endian จะกลายเป็น 0x00A0153C แล้วคำสั่งเปลี่ยนหมด

10 กับดักที่ต้องเช็คก่อนส่งกระดาษ

① ข้อ 1.5/1.6 ต้องตอบ **ทั้งใครขับ และค่าระบุไม่ได้** ② arbiter ไม่ได้เขียน bus ได้ตลอด ③ อ่าน bus ได้ **ทุกตัว** ④ ข้อ 3 ช่อง (d) = address **เดิม** ⑤ `lw/sw` เรียง **rs ก่อน rt** ⑥ branch ต้อง **−4 แล้วหาร 4** ⑦ `j` ต้อง **หาร 4** ⑧ ห้าม `li/move/blt` ⑨ Big Endian อย่ากลับด้าน ⑩ อย่าจำ address map ของรูป 3 กับรูป 4 สลับกัน

ส่วนที่ 3 · เคสพลิกโจทย์ (ถ้าอาจารย์แก้ข้อสอบ)

ข้อ	ถ้าโจทย์เปลี่ยนเป็น…	คำตอบ
1	read + write inactive ทั้งคู่	ไม่มีใครขับ ⇒ data bus เป็น **Hi-Z / floating / ไม่นิยาม**
1	read + write active พร้อมกัน	ผิดกติกา ⇒ **bus conflict** ค่าไม่นิยาม อาจเสียหาย
1	ถามค่าบน **address** bus	ตอบได้ = ค่าที่โจทย์ให้ (เช่น 0x2000 4000) เพราะ CPU ใส่เอง
1	data 16 บิต, addr 10 บิต ถามเป็นไบต์	ถ้า 1 ช่อง = 1 word 16 บิต ⇒ **2048 ไบต์** · ถ้า byte-addressable ⇒ 1024 ไบต์
1	addr 32 บิต / 24 บิต อ้างได้เท่าไร	**4 GB / 16 MB**
1	เพิ่ม CPU B หรือ DMA ที่เป็น master ได้	address bus เป็น **สองทางในเชิงระบบ** ต้องมี arbiter
2	นับเส้นได้ 16 เส้น	**16 บิต** (ข้อนี้วัดการนับเส้นล้วนๆ)
2	arbiter พัง	ไม่มีใครแจกสิทธิ ⇒ ระบบค้าง หรือเขียนพร้อมกันจนเกิด contention
2	ใช้ open-drain แทน tri-state	ไม่ลัดวงจร แต่ผลเป็น **wired-AND** ⇒ ข้อมูลยังผิด
2	5 อุปกรณ์ ต้องใช้ dedicated line กี่เส้น	1 เส้น/ตัว = **5 เส้น** · ถ้าเข้ารหัสฐานสอง = **3 เส้น**
2	bus 8 บิต ส่ง 32 บิต กี่รอบ	**4 รอบ**
3	**"ทำคำสั่งที่ X เสร็จแล้ว"** ค่อยได้ interrupt	(d) = **X + 4** ไม่ใช่ X
3	CPU ปิดรับ interrupt (masked)	ไม่กระโดด ทำคำสั่งที่ X ต่อ ไม่มี ISR
3	vector = 0x9000 0000 / 0xE000 0100	รูป 3: **DMA / Dev 1** ⇒ ผิดปกติ ISR ไม่ควรอยู่ที่ I/O
3	ย้าย X1,X0 ไปรับ A[27],A[26]	ต้องกางบิตใหม่ ⇒ พื้นที่แต่ละอุปกรณ์กระจายเป็นก้อน 64 MB สลับกัน
3	เอา ○ ออกจาก CS ของ Main Memory	Main Mem ถูกเลือกเมื่อ A[31]=**1** ⇒ ชนกับ DMA/Decoder ⇒ ระบบผิด
3	ถามว่าใครเก็บโปรแกรมตอนปิดเครื่อง	**Boot Mem** (non-volatile) · Main Memory เป็น volatile
3	Dev 0 กับ Dev 1 interrupt พร้อมกัน	OR-gate บอกแค่ "มีคนเรียก" ⇒ ISR ต้อง poll กันตัว หรือใช้ vectored interrupt
4	`A[i] = a;` (index เป็นตัวแปร)	`sll $t0,$s2,2` ; add `$t0,$t0,$s3` ; `sw $s0,0($t0)`
4	`if(a==b){..}else{..}`	`bne $s0,$s1,ELSE` / then / `j ENDIF` / `ELSE:` else / `ENDIF:`
4	`a = a*8;` / `a = a/4;`	`sll $s0,$s0,3` / `srl $s0,$s0,2` (a ≥ 0)
4	array เป็น `char` / `double`	offset ×1 ใช้ `lb/sb` / offset ×8 ⇒ `sll ,3`
5	ให้ hex มา แล้วให้ถอดเป็น assembly	กาง 32 บิต → op 6 บิตแรก: 000000=R (ดู funct) · 00001x=J · อื่น=I
5	label อยู่**ก่อน** branch (ลูปย้อน)	offset ติดลบ เช่น ย้อน 4 คำสั่ง ⇒ imm = **−5** = 0xFFFB
5	`j` / `beq` ไปได้ไกลแค่ไหน	**256 MB** ใน region เดียว (บิต 31–28 มาจาก PC+4) / **±32768 คำสั่ง**
5	ย้ายโปรแกรมไป 0x10400054	`j` ยังเป็น **0x08100015** (เก็บแค่ 26 บิตล่าง) และทำงานถูกเพราะอยู่ region เดียวกัน
6	เปลี่ยนเป็น **Little Endian**	`3C 15 A0 00` ⇒ **0x00A0153C** ⇒ op = 000000 = R-type ⇒ ถอดใหม่หมด
6	Bootstrapping Address = 0x00000010	คำสั่งแรกคือ `j 0x00400050` ทันที
6	`lui $s5,0xC000` / `0x8000`	ปลายทาง 0xC000 0000 ⇒ **Dev 2** / 0x8000 0000 ⇒ **Dev 0**
6	ถามค่า `$s1` / `$s5`	$s1 **บอกไม่ได้** (อ่านมาจาก Dev 1) · $s5 = **0xA000 0000**
6	คำสั่งไหนติดต่อ **Main Memory**	**ไม่มีเลย** ใน 5 คำสั่งแรก
6	คำสั่งที่ 6, 7 (สมมติไม่มี `j`)	0x14 = 0x27A50004 = `addiu $a1,$sp,4` · 0x18 = 0x8FA40000 = `lw $a0,0($sp)`
6	Gate A1 รับ A[31-16] แทน	Boot Mem โตเป็น **64 KB** (0x0–0xFFFF) · Main Mem เริ่มที่ 0x0001 0000
6	"โปรแกรม boot นี้ทำอะไร"	ตั้ง base ของ Dev 1 → ส่งค่า 0x27 ไปอุปกรณ์ → อ่านค่าตอบกลับ → กระโดดไปโปรแกรมหลักที่ 0x00400050

★ ลำดับทำข้อสอบ + เช็คก่อนส่ง
ทำข้อ **5 → 6 → 1 → 2 → 3 → 4** (คำนวณล้วนก่อน เขียนโค้ดทีหลัง) ก่อนส่ง: ① ตอบครบ (a)(b)(c)(d) ② ไม่มี pseudo ③ หาร 4 ครบ ④ hex ครบ 8 หลัก ⑤ ข้อ "อธิบาย" มีเหตุผลครบ

ส่วนที่ 4 · แผ่นสุดท้ายก่อนเข้าห้องสอบ

Σ สูตรทั้งหมดในกล่องเดียว
ความจุ = 2^(บิต address) × (ไบต์ต่อ 1 ช่อง) · **ขนาดกล่องในรูป** = 2^(จำนวนบิตสาย Addr)
Branch imm = (addr(label) − addr(branch) − 4) ÷ 4 | **ปลายทาง** = addr(branch) + 4 + imm × 4
Jump field = addr(target) ÷ 4 (26 บิต) | **ปลายทาง** = (PC+4)[31:28] || field || 00
offset ของ A[k] = k × sizeof | **2's comp** = 0x10000 − |n| (16 บิต)
Big Endian word = byte[a]·byte[a+1]·byte[a+2]·byte[a+3] เรียงซ้าย→ขวา
Bandwidth = ความถี่ × (data bus ÷ 8) | **รอบส่งข้อมูล** = ⌈N ÷ M⌉

คำศัพท์	ความหมายสั้นที่ใช้ตอบได้เลย
Volatile / Non-volatile	ข้อมูลหาย / ไม่หายเมื่อดับไฟ (RAM / ROM · Main Mem / Boot Mem)
Tri-state / Hi-Z	เอาต์พุต 3 สถานะ (0, 1, ลอย) / สถานะไม่ขับสัญญาณ ปล่อยให้คนอื่นใช้ bus
Chip Select (CS)	ขาที่บอกว่าชิปถูกเลือก · วงกลม ○ = active-LOW
Decoder	แปลง n บิต → เลือกเอาต์พุต 1 ใน 2^n เส้น
Bus master / Arbiter	ตัวที่ควบคุม bus ในจังหวะนั้น / ตัวตัดสินว่าใครได้เป็น master
Bus contention	หลายตัวขับ bus พร้อมกัน → ค่าไม่นิยาม + กระแสลัดวงจร
Memory-mapped I/O	I/O ใช้ address ปนกับหน่วยความจำ ⇒ ติดต่อด้วย `lw/sw`
Isolated I/O	แยกพื้นที่ I/O ออกจาก memory ต้องใช้คำสั่งเฉพาะ (`in/out`)
ISR / Interrupt vector	โปรแกรมตอบ interrupt / address ที่ CPU กระโดดไปเมื่อเกิด interrupt
Polling	CPU วนอ่าน status register เอง (busy-waiting)
DMA	ย้ายข้อมูลระหว่าง I/O กับหน่วยความจำเองโดยไม่ผ่าน CPU
Bootstrapping	โปรแกรมชุดแรกหลังเปิดเครื่อง อยู่ใน non-volatile memory
Relocatable	ย้ายโค้ดไปที่อื่นแล้วยังทำงานได้ (เพราะ branch ใช้ offset สัมพัทธ์)
Pseudo-instruction	คำสั่งที่ assembler แปลงเป็นคำสั่งจริง — ฮาร์ดแวร์ไม่มี (`li, move, blt`)
Sign / Zero extension	ขยาย 16→32 บิต โดยก๊อปบิตเครื่องหมาย / เติม 0 (`addi` ใช้ sign · `andi/ori` ใช้ zero)

⚠ 12 กับดักที่ทำให้เสียคะแนนมากที่สุด
① ข้อ "อธิบาย" ตอบคำเดียวไม่มีเหตุผล ② ตอบค่า data bus เป็นตัวเลขทั้งที่โจทย์ไม่ได้ให้เนื้อข้อมูล ③ คิดว่า arbiter เขียน bus ได้ตลอด ④ คิดว่าอ่าน bus ต้องขอสิทธิ ⑤ ช่อง (d) ของ interrupt ตอบ +4 ทั้งที่โจทย์ว่า "ก่อนจะโหลด" ⑥ `lw/sw` สลับ rs กับ rt ⑦ ลืม −4 หรือลืมหาร 4 ในการคิด branch offset ⑧ ลืมหาร 4 ใน `j` ⑨ ใช้ `li/move/blt/bgt` ⑩ อ่าน Big Endian กลับด้าน ⑪ จำ address map ของรูป 3 กับรูป 4 สลับกัน ⑫ ลืมคูณ index ด้วย 4 สำหรับ array int

★ 3 นาทีแรกในห้องสอบ
① เปิดดูรูปวงจรทุกรูป แล้ว **ร่าง address map ลงกระดาษทดทันที** (ใช้แม่แบบข้อ 1.17)
② เขียนตาราง register (`$s0` =16 … `$t0` =8) และ op ที่ต้องใช้ไว้มุมกระดาษ
③ ค่อยเริ่มตอบข้อ 5 → 6 → 1 → 2 → 3 → 4

1.17 · แม่แบบเปล่า — ใช้ร่าง address map ในห้องสอบ

ใช้ยังไง
พอเห็นรูปวงจร ให้วาดตารางนี้ลงกระดาษทดก่อนตอบข้อไหนก็ตาม แล้วค่อยตอบทุกข้อย่อยจากตารางเดียวนี้ — เร็วกว่าและไม่พลาด

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

ช่องจดค่า register ตอนไล่โปรแกรม

addr	machine code	assembly	$ ที่เปลี่ยน	แตะ I/O?