# Hardware Design: Tutorial 2 (ISA II) Solutions

**Ali Alsarraf** (TA)
**Haiyu Mao** (Module Lead)

30th Jan 2026

## Question 1: Subtraction — MIPS vs LC-3 (assembly and ALU)

You are given:

- MIPS registers: `$s0=b`, `$s1=c`, `$s2=d`
- LC-3 registers: `R0=b`, `R1=c`, `R3=d`

**(a)** Write **MIPS** assembly to compute

$$a = b + c - d$$

Store the result in `$s3`.

**(b)** Write **LC-3** assembly to compute the same expression, storing the result in `R6`.

**(c)** Explain briefly how subtraction is implemented in LC-3 using **two's complement**.

**(d)** At the hardware level (ALU), is subtract a fundamentally different operation from add? Explain briefly.

**(e)** Compute **two** MIPS solutions for

$$a = b - 3$$

One using `subi` (if available) and one using `addi`. Is `subi` necessary?

**Solution**

**(a) MIPS** (use `add` then `sub`)

```
add  $t0, $s0, $s1      # t0 = b + c
sub  $s3, $t0, $s2      # s3 = (b + c) - d
```

So $\$s3 = b + c - d$.

**(b) LC-3** (LC-3 has no `SUB`, so compute **-d** and add it)

```
ADD R2, R0, R1          ; R2 = b + c
NOT R4, R3              ; R4 = ~d
ADD R5, R4, #1          ; R5 = ~d + 1 = -d
ADD R6, R2, R5          ; R6 = (b + c) + (-d) = b + c - d
```

So $R6 = b + c - d$.

**(c) Two's complement idea**

$$x - y = x + (\sim y + 1)$$

LC-3 implements subtraction by doing `NOT` **then** `ADD #1` to form **-y**, then `ADD`.

**(d) ALU viewpoint** Subtract is typically implemented using the **same adder**:

$$A - B = A + (\sim B) + 1$$

Hardware often inverts $B$ (controlled by a SUB signal) and sets carry-in to 1.

**(e) Is `subi` necessary? Not necessary.** Subtraction by a constant is addition with a negative immediate.

If `subi` exists:

```
subi $s1, $s0, 3        # s1 = b - 3
```

Always valid:

```
addi $s1, $s0, -3       # s1 = b + (-3) = b - 3
```

So `addi` with a negative immediate is sufficient.

## Question 2: LC-3 — LEA vs LDI

Assume:

- The instruction is at address `x4018`.
- The incremented PC during execution is `PC+ = x4019`.

Memory contents:

- `M[x4016] = x3100`
- `M[x3100] = x00A5`

**(a)** What value ends up in `R5` after each instruction?

**(i)** `LEA R5, #-3`

**(ii)** `LDI R5, #-3`

**(b)** Which instruction(s) access memory to obtain the value loaded into `R5`? How many memory reads occur?

**Solution**

First compute the effective address using **PC+** and the offset:

$$\textbf{PC+ = x4019}, \quad \textbf{\#-3} \Rightarrow x4019 + (-3) = x4016$$

**(a)(i) LEA** loads the **address** (no memory read for the loaded value):

$$R5 \leftarrow x4016$$

**(a)(ii) LDI** is **indirect**: it reads memory twice.

$$A = x4016, \quad P = M[A] = M[x4016] = x3100, \quad R5 = M[P] = M[x3100] = x00A5$$

**(b) Memory reads**

- **LEA:** 0 reads to obtain the loaded value.
- **LDI:** 2 reads (one to get the pointer, one to get the final value).

## Question 3: MIPS — Loading a full 32-bit constant

You want:

$$\$s0 \leftarrow 0x6D5E4F3C$$

**(a)** Why can't this be done with a single `addi`?
**(b)** Write a correct MIPS sequence to load this constant into `$s0`.

**Solution**

**(a)** `addi` has a **16-bit immediate** (sign-extended), so it cannot directly encode an arbitrary **32-bit** constant.

**(b)** Standard approach: load upper 16 bits, then fill lower 16 bits.

```
lui  $s0, 0x6D5E       # $s0 = 0x6D5E0000
ori  $s0, $s0, 0x4F3C  # $s0 = 0x6D5E4F3C
```

So $\$s0 = 0x6D5E4F3C$.

## Question 4: Branching — BEQ (MIPS) vs BRz (LC-3)

Assume:

- MIPS: `$s0 = 12`, `$s1 = 12`
- LC-3: `R0 = 12`, `R1 = 12`

You want: "If the two registers are equal, branch to label `TARGET`."

**(a)** Write the MIPS instruction.
**(b)** Write an LC-3 instruction sequence that achieves the same effect using `BRz` only.
**(c)** Explain one key ISA difference between how MIPS and LC-3 decide whether to branch.

**Solution**

**(a) MIPS** compares two registers inside the branch instruction:
`beq $s0, $s1, TARGET`
**(b) LC-3** branches using **condition codes** (N/Z/P). To branch on equality, compute

$$R0 - R1$$

and branch if the result is zero:

```
NOT R2, R1
ADD R3, R2, #1      ; R3 = -R1
ADD R4, R3, R0      ; R4 = R0 - R1
BRz TARGET          ; branch if equal
```

**(c) Key ISA difference**

- **MIPS: comparison is part of the branch instruction** (e.g., `beq rs, rt, label`).
- **LC-3:** branch decision depends on **condition codes** set by a *previous* ALU-style instruction.

So LC-3 often needs an explicit "compute then branch" sequence.

## Question 5: Branch offset in MIPS

You see this code with addresses:

```
0xA4:  beq  $t0, $0, ELSE
0xA8:  addi $v0, $0, 1
0xAC:  addi $sp, $sp, 8
0xB0:  jr   $ra
0xB4:  ELSE: addi $a0, $a0, -1
```

What is the 16-bit immediate (offset) field inside the `beq`?

**Solution**

Branch offsets are relative to `PC+4`.
Branch is at `0xA4`, so:

$$\textbf{PC+4 = 0xA8}$$

Target `ELSE` is at `0xB4`.
Byte distance:

$$0xB4 - 0xA8 = 0x0C = 12$$

Convert bytes to instructions (divide by 4):

$$12/4 = 3$$

So the immediate field is 3.

## Question 6: LC-3 addressing-mode trace

Assume:

- PC initially points to `x30F6`
- Instruction at `x30F6` is `LEA R1, #-3`

**(a)** What value is written to `R1`?
**(b)** What is the PC after executing the instruction?

**Solution**

From LEA semantics:

$$R1 \leftarrow PC^{+} + \text{sext(offset)}$$

and the PC increments during fetch.

PC starts at **x30F6**.
After fetch, $PC^{+}$ **= x30F7**.
**Offset is -3, so:**

$$R1 = x30F7 - 3 = x30F4$$

**PC after executing the instruction is the next instruction address:**

$$PC = x30F7$$