LSU EE 4720 **Homework 2** **Due: 13 February 2026**

Solution *Formatted 15:07, 19 February 2026*

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of MIPS, RISC-V or their assembler syntax, how a part of the problem might be solved, etc.) It is also acceptable to seek out resources for help on MIPS, RISC-V, etc. For this assignment (2026 Homework 2), please read the MIPS and RISC-V documentation *before* seeking out help. Attempt to learn by reading the indicated material.

Problem 1: Solve 2025 Final Exam Problem 5(a) and 5(b) **by hand**. Show all field values, including the `func` and `opcode` fields. You can find MIPS 32 ISA (architecture) manuals linked to the course references Web page. Solving *by hand* means to solve the problem by looking up the instructions in the respective ISA (architecture) manuals and figuring out what to put in the fields. Do not "solve" the problem by entering the code into a file and, say, using the SPIM simulator to view encoded output. Also do not "solve" the problem by prompting an LLM.

See the posted (partial) exam solution. *Why don't I just copy the solution here? Because looking at the final exam will give you an idea of what's coming next.*

There's more problems on the next page.

Problem 2: First, familiarize yourself with RISC-V by reading Chapter 1 of Volume I of the RISC-V specification, especially the Chapter 1 Introduction and Sections 1.1 and 1.3. Skip Section 1.2 unless you are comfortable with operating system and virtualization concepts. Other parts of Chapter 1 are interesting but less relevant for this problem. The questions in this assignment are about RISC-V RV32I. Also read the material in Chapter 2 relevant to this question. **Solve this question by reading the material in Chapter 2 and other parts of the RISC-V specification, not by other means.** Test questions will be written for students that have read these chapters. Chapter 2 provides the format and details of base RV32I instructions, but it does not provide opcode and related values. Those can be found in the chapter titled "RV32/64G Instruction Set Listing," which is Chapter 36 in the May 2025 version of the spec (it was Chapter 34 in 2024).

(*a*) Show the corresponding RISC-V instruction for the MIPS code below. *Hint: The mnemonic is not* `sll`*, but it's close.*

☑ Show the corresponding RISC-V instruction.

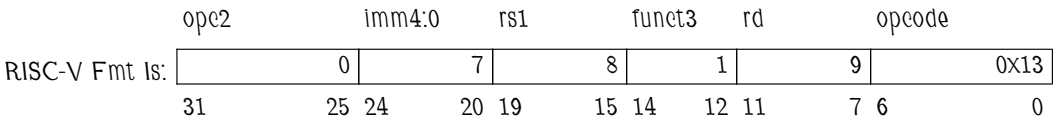
☑ Show encoding of the RISC-V instruction. ☑ Label fields, and show specific values. to the format that this particular instruction uses.

```
sll r9, r8, 7
```

The equivalent RISC-V instruction is **`slli`**. Note that in RISC-V the mnemonic for the immediate version of the shift-left instruction ends in an **`i`**, **`slli`**, whereas in MIPS the immediate version is just called **`sll`**. MIPS instruction **`sllv`** is used when the shift amount is a register, such as **`sllv r9, r8, r5`**. An interesting test question might be whether this difference (**`sll`** versus **`slli`**) is simply the result an arbitrary naming decision or whether the difference in names indicate something more. Those who solved this problem may be able to answer the question and certainly would be able to understand the answer.

The instruction and its encoding are shown below. The high seven bits are labeled **`opc2`**, though in the RISC-V standard they are called **`imm11:5`**. They are called **`opc2`** here because they are used in part as an opcode extension.

```
# SOLUTION - RISC-V
slli r9, r8, 7
```



(*b*) Convert the MIPS I code fragment below to RISC-V RV32I. Pay attention to differences between MIPS and RISC-V branch and load-upper-immediate instructions.

☑ Convert fragment from MIPS I to RISC-V RV32I. ☑ Note the difference in RISC-V branch behavior. ☑ Reduce the number of instructions where possible.

```
 bne r5, r6, SKIP
 sw r4, 5(r6)
 lui r8, 0x5678
 addi r8, r8, 0x4000
SKIP:
 add r9, r8, r10
```

Solution appears below. Branches (and jumps too) in RISC-V don't have delay slots, and so the `sw` had to be moved before the branch. Another difference is that the `lui` in RISC-V has a 20-bit immediate, so the entire constant, `0x56784`, could be loaded into `r8` with just one instruction.

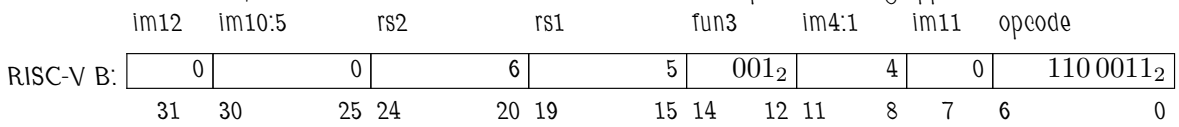
```
 sw r4, 5(r6)
 bne r5, r6, SKIP
 lui r8, 0x56784
SKIP:
 add r9, r8, r10
```

☑ Show the encoding of the RISC-V branch instruction used in the converted code fragment. Pay attention to the branch displacement (called and offset in the RISC-V documentation).

The solution appears below. In the MIPS Type I format the immediate is in bits 15:0 of the instruction, nice and simple. In the RISC-V type B format the immediate is twelve bits, but rather than placing the immediate value in twelve consecutive bits of the instruction, RISC-V breaks the immediate value into four pieces and puts them into four places. The pieces are labeled `im12`, `im11`, `im10:5`, and `im4:1`. Piece `im12` is the MSB of the immediate, piece `im4:1` is the least significant 4 bits (not counting bit zero which is not part of the instruction). Notice that the order of these pieces in the instruction is almost MSB to LSB, except that bit 11 of the immediate was plucked out of its rightful place in bit 30 of the instruction and moved to bit 7. A prank that was not noticed until it was too late? The RISC-V spec explains that it was a conscious choice to simplify the hardware by reducing the number of distinct wire connections on certain implementations.

The displacement in RISC-V starts at the branch instruction itself (in MIPS it starts at the delay slot which anyway RISC-V lacks). The value of the displacement is the number of 32-bit instructions to skip *times 2*. (Skipping one instruction means executing the instruction immediately following the branch, etc.) Or equivalently, it's the difference between the address (PC) of the branch and the address of the target, divided by 2. This method of encoding displacements cuts the range of RV32I branches in half (compared to encoding the displacement in units of instructions, as in MIPS). It was done to simplify the implementation of RISC-V RV32C (the C is for compressed, as in compressed-to-16-bits instructions). RISC-V RV32C allows for a mix of 32- and 16-bit instructions. To keep things simple the ISA was designed so that every 16-bit RV32C instruction can be converted into a (32-bit) RV32I instruction. For that to be possible RV32I branches have to be capable of computing a target that's a multiple of two bytes.

Back to the solution. The `bne` instruction skips 2 instructions, so the displacement needs to be 4. That value of 4 fits in the `im4:1` field, the other immediate fields hold zeros. The complete encoding appears below:



There's another problem on the next page.

Problem 3: In a skeptical mood I prompted ChatGPT 5.2 with: "Is the RISC zero register overrated?" It responded, in part:

```
Hot take: **the RISC zero register is neither overrated nor
sacred**its a very pragmatic hack that aged surprisingly well, but
its not universally best.

Lets unpack it like engineers, not cultists :-)

### 1. Instruction set simplification

You get a bunch of operations for free without extra opcodes:

* ‘move rd, rs‘ → ‘add rd, rs, x0‘
* ‘clear rd‘ → ‘add rd, x0, x0‘
* ‘neg rd, rs‘ → ‘sub rd, x0, rs‘
* Compare against zero without loading a literal

Thats real silicon and encoding savings, especially in early RISC
designs where every opcode bit mattered.
```

Yes, it's important to reduce the number of opcodes. But is that really a good list of pseudoinstructions that can be used by ISAs with a zero register so that they don't need to have real instructions, such as `move`, that would use a valuable opcode?

A complete transcript of the session with GPT 5.2 is at
`https://www.ece.lsu.edu/ee4720/2026/hw02-RISC-Zero-Register-Analysis.pdf`.

(*a*) In MIPS-I, which of the first three pseudoinstructions in the answer above (`move`, `clear`, `neg`), can be done *without* a zero register?

☑ If possible, show MIPS-I instructions without a zero register that can be used for `move`, `clear`, and `neg`. ☑ Don't just put a zero into some register and use that.

There are many ways to implement the first two pseudoinstructions, `move` and `clear`. For `move r1, r2` one can use `or r1, r2, r2`, or `ori r1, r2, 0`. For `clear r1` one can use `andi r1, r1, 0` or `xor r1, r1, r1`. (Note: in these examples for `clear` the source and destination register are the same for reasons which will become clear when we cover implementations. Note to self: This would make a good midterm exam question.)

There is no single MIPS-I instruction (AFAIK) that can implement `neg` without a zero register, so `neg` was the only one of the three that is a good example of why a zero register is needed. There are many ways to implement `neg` with multiple instructions, but the argument was about how with a zero register a single existing instruction can be used in place of a *single* new instruction. The assumption here is that a single instruction performing an operation is better than multiple operations performing the same operation.

(*b*) The fourth "compare-against-zero" bullet might have referred to conditional branches. Consider mnemonics `bnez` (branch not equal to zero) and `bgtz` (branch greater than zero).

☑ Are `bnez` and `bgtz` examples of operations that one gets for free in MIPS? The word *free* is in the sense used by ChatGPT 5.2 in the `You get a bunch of operations "for free"` sentence. Answer for ☑ `bnez` and for ☑ `bgtz`.

The phrase *get ... operations for free* means the operation, or pseudo-instruction, can be constructed using existing MIPS instructions using the zero register.

MIPS-I does not have a `bnez` instruction, and no other single instruction (AFAIK) can implement it without a zero register. So, with a zero register you can use `bne r1, r0, TARG` to get `bnez r1, TARG` for free (as a pseudoinstruction).

MIPS-I does have a `bgtz` instruction, so MIPS-I doesn't need to get it for free, it already has it. *Hmm, does a zero register enable a future extension of this ISA for the* `bgtz` *instruction?*