LSU EE 4720 **Homework 4** **Due: 19 Mar 2025 at 09:30 CDT**

Solution *Formatted 17:23, 19 March 2025*

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 or assembler syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out assembly language resources for help on MIPS, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Student Expectations

Some of the problems require thought, and students are expected to persevere until they find a solution. It is each student's duty to him or herself to resolve frustrations and roadblocks quickly, hopefully helped along by the satisfaction of making progress. There are plenty of old problems and solutions to look at. One way to resolve issues is to ask Dr. Koppelman or others for help.

Resources

See old homework and exams. There are a few questions about VAX in past assignments. There are question about RISC-V in many of the more recent assignments.

Problem 1: Remember that VAX is one of the few examples of a good CISC ISA. CISC ISAs are not considered suitable for current implementation technology, but those who do not learn by history are doomed to repeat it, so look over the summary of the VAX instruction set which can be found in Chapter 2 of the VAX 11/780 Architecture Handbook Volume 1, 1977-78. Focus on Section 2.4, which summarizes the instruction set. Consider item 5 in that section, which starts "Instructions provided specifically for high-level language constructs." Three examples of such instructions are given, `ACB`, `CALLS`, and `CASE`. As guided by the check boxes below, explain how a register-only version of suitable each instruction is for implementation in a RISC ISA. The instruction descriptions in the architecture handbook use metasyntactic symbols `rx`, `mx`, and `wx` to sources and destinations. (In MIPS `rs`, `rt`, and `rd` are metasyntactic symbols.) Symbol `rx` is used for a read (source) operand (signified by the `r`) that can come from a register, immediate, or memory (signified by the `x`). Similarly the `w` in `wx` signifies an argument that is written (a destination), and the `m` in `mx` signifies an argument that is read and then written. The questions below ask about hypothetical *register-only* versions of these instructions in which arguments `rx`, `mx`, and `wx` refer only to register arguments.

The instructions are explained in the architecture manual, but feel free to seek out other references. The description of `ACB` is fairly straightforward. The `CALLS` instruction is clear but may be difficult to understand for those who are less familiar with bit masks or bit vectors. In addition to the Architecture Handbook, see VAX MACRO and Instruction Set Reference Manual for a description of the `CASE` instruction and an example of its use. Note that for `CASES` the table (`displ`) is in memory immediately after the instruction. The operation performed by the `CASE` instruction is similar to the MIPS assembly code for the dense switch statement presented in the class control flow demo code. Of course, `CASE` does most of that with one instruction.

☑ A register-and-displacement-operand-only version of the `ACB` instruction ◯ *is definitely not suitable for a RISC ISA*, ⊗ *arguably possible for a RISC ISA*, ◯ *fits well into a RISC ISA*.

☑ Explain. In your explanation consider how easy it would be to ☑ encode in a RISC ISA (allow some flexibility) and how easy it would be ☑ to implement in a five-stage pipeline.

Page 8-10 of the Architecture Handbook describes `ACB` as taking four operands, a *limit*, *add* (increment amount), an *index*, and a *displacement*. To execute the branch the hardware computes index+add compares, the sum to limit, and branches to PC + displacement if the sum is greater than limit (if add is positive) or less than limit (if add is negative).

Encoding all of these operands would be possible, but not easy because there would not be much room for an immediate and three register fields in a 32-bit instruction. In MIPS one could use `sa` and `func` for the displacement (which would be 11 bits), but that would require a new opcode. (Less radical type-R instructions use the `func` field as an opcode extension.) Another possibility is to consider a variation without the add (increment) field, and instead always just add one. Or, one could dispense with the limit field, and instead take the branch if the result were positive (and so add [the increment] would have to be negative).

Resolving the branch requires both an addition and a comparison. If using the five-stage MIPS pipeline the comparison would have to be after `EX`, in `ME`. Without branch prediction there would be a two or three instruction penalty (depending on how long the comparison takes). This would not be a problem with branch prediction. The extra comparison unit adds to cost, as would the need to carry the branch target to the `ME` or `WB` stage. So it's doable, but it would add significantly to cost. If the comparison were done *before* the addition then it would be possible to resolve the branch in `ID` so this would be much easier to add to a RISC ISA because it could use the some comparison unit used by existing branch instructions. In MIPS only equality could be tested without requiring a new comparison unit, but other RISC ISAs do allow magnitude-comparison jumps so that a new comparison unit would not need to be added.

☑ The `CALLS` instruction ⊗ *is definitely not suitable for a RISC ISA*, ◯ *arguably possible for a RISC ISA*, ◯ *fits well into a RISC ISA*.

☑ Explain. In your explanation consider how easy it would be to ☑ encode in a RISC ISA (allow some flexibility) and how easy it would be ☑ to implement in a five-stage pipeline.

The `CALLS` instruction extends the stack (an area of memory used for storing register values, local variables, and other information associated with a called procedure), and the writes the stack with several register values, including a return address and caller-save registers. The list of the caller save register is specified in a bit mask placed in the first part of the called procedure.

Encoding `calls` is not a problem because it has two arguments, *numarg* specifies the number of parameters in the called function, and *dst* specifies the address of the target. A register or small immediate could hold *numarg*, and a larger immediate could hold *dst* as a displacement from PC, for example.

Implementing the instruction is definitely a problem because the hardware must first load the *entry mask* at the beginning of an instruction, then store the value of those registers specified in the entry mask, in addition to always-save values such as the return address. In a pipelined implementation there is one memory port for loads and stores, and that is in ME (using MIPS stage names). An instruction only gets to use ME for one cycle, so it could not perform the load and stores that are needed. So that rules it out. Do not expect to convince management otherwise.

Those who nevertheless want to make a case for such an instruction, read on. The only way to implement this in what before this instruction was a typical pipelined RISC implementation would be for the `CALLS` instruction to "take over" the pipeline and operate it as a CISC implementation would, meaning it would execute over multiple steps, using the ALU and memory port multiple times as directed by either really complex control logic or a smaller computer, called a *microprogrammed control unit*. As most have probably guessed, CISC implementations use microprogrammed control units.

☑ A register-operand-only version of the `CASE` instruction ◯ *is definitely not suitable for a RISC ISA*, ⊗ *arguably possible for a RISC ISA*, ◯ *fits well into a RISC ISA*.

☑ Explain. In your explanation consider how easy it would be to ☑ encode in a RISC ISA (allow some flexibility) and how easy it would be ☑ to implement in a five-stage pipeline.

The `CASE` instruction has three operands, *selector* (s), *base* (b), and *limit* (l). Also, immediately following each `CASE` instruction is a table of memory addresses. If $s < l$ execution continues after the table. Otherwise execution jumps to $\mathrm{PC} + M[\mathrm{PC} + 2(s - b)]$, where $M[a]$ is the two bytes of memory starting at address a.

Because it has three source operands, it is easy enough to encode in a RISC ISA(with the source operands all being registers).

An implementation would need to compute address $\mathrm{PC} + 2(s - b)$. For $b = 0$ this would be little different than computing branch target. (For MIPS multiply by 4 instead of 2, but for RISC-V branch displacements are in units of half-instructions.) To tack this on to a RISC implementation one might design an ALU that can compute $\mathrm{PC} + 2(s - b)$ in one cycle. That's not impossible but it is doable. Expect an argument from management. An alternative would be to add a second `EX` stage. If this is a five-stage pipeline before the change then getting that additional stage approved just for this would be very difficult. Perhaps the easiest thing to do is to implement a version of `CASE` that lacks a base argument, meaning that the `EX` stage would just have to compute $\mathrm{PC} + 2s$. This is almost like a MIPS branch, except that in a branch the s would be the immediate value.

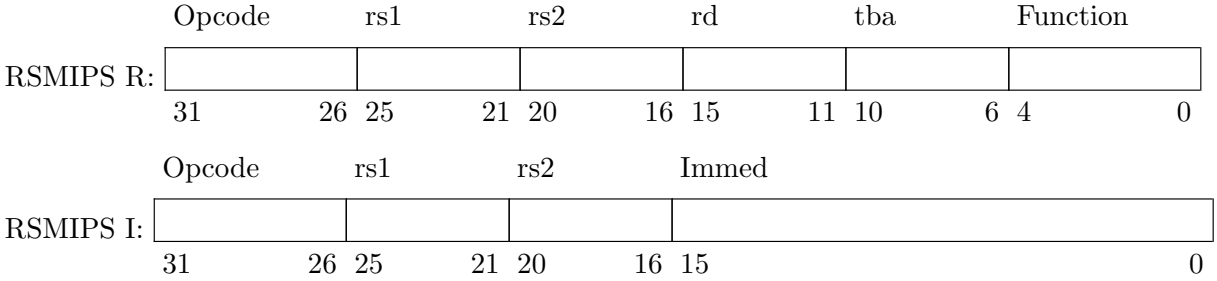
In `EX` the value of $\mathrm{PC} + 2(s - b)$ (or $\mathrm{PC} + 2s$) is connected to the `Addr` input of the memory port and a read operation is performed, reading the address to jump to. In `WB` the value loaded from memory *is connected to the* PC (rather than being written to memory). This last part, writing the PC when an instruction is in `WB` rather than `ID`, also might make RISC purists defensive. There is the cost of another input to the multiplexor feeding PC, and also the large penalty. Assuming there is a delay slot, the penalty would be three cycles. In the example below the first six elements of the dependency table (two bytes each) are fetched as though they are instructions. They are squashed in cycle 4. The control logic could also have stalled fetch in cycle 2, since by then it had seen the `CASE` pass through `ID` a cycle earlier.

```
# Cycle           0  1  2  3  4  5  6  7  8  9
 CASE r1, r2, r3  IF ID EX ME WB
 nop                 IF ID EX ME WB
 dep[0] dep[1]          IF ID EXx
 dep[2] dep[3]             IF IDx
 dep[4] dep[5]                IFx

casex:
 sw r1, 2(re)                      IF ID EX ME WB
# Cycle           0  1  2  3  4  5  6  7  8  9
```

So this is doable, especially without the base. Appearing below is what MIPS code might be used without a **CASE** instruction, (based on the course example for coding a dense switch statement). With a **CASE** instruction the correct target is reached in 5 cycles, without such an instruction it takes 7 cycles, and that's for a base of zero and without checking whether `t1` is out of range.

```
        .data
 ## Dispatch table, holding address of case statements.
DTABLE:
        .word CASE0
        .word CASE1
        .word CASE2
        .word CASE3
        .word CASE4
        .word CASE5
        .text

#       Cycle                     0  1  2  3  4  5  6  7  8  9  10 11
        lui $t5, hi(DTABLE)       IF ID EX ME WB
        ori $t5, $t5, lo(DTABLE)     IF ID EX ME WB
        sll $t6, $t1, 2                 IF ID EX ME WB
        add $t6, $t6, $t5                  IF ID EX ME WB
        lw $t7, 0($t6)                        IF ID EX ME WB
        jr $t7                                   IF ID EX ME WB
        nop                                         IF ID EX ME WB
....
CASEx:
        xor $s1, $s2, $s3                              IF ID EX ME WB
#       Cycle                     0  1  2  3  4  5  6  7  8  9  10 11
```

Problem 2: RSMIPS is a hypothetical ISA with similarities to MIPS. Appearing below is RSMIPS' instruction format R, which is identical to MIPS' format R (except for the names of the source fields). Unlike MIPS, in RSMIPS all instructions that write a result to a register use the `rd` field for the register number (and the `rd` field is always in bits 15:11). Yes, RSMIPS is Real Strict about source and destination register fields, hence the name. Also notice that different from MIPS the RSMIPS source fields are named `rs1` and `rs2`. Remember that in MIPS, `rt` can be used as either a source or destination, depending on the instruction.

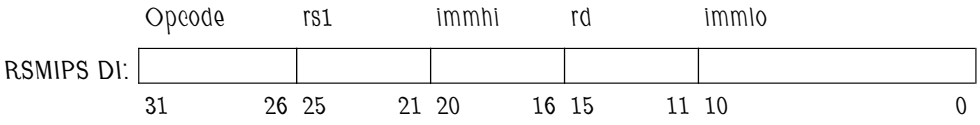


Because of this Real Strict provision, something like MIPS' format I can't be used for instructions such as `addi` and `lw`, but format I can be used for instructions such as `sw` and `beq`.

(*a*) In RSMIPS *format DI* is used for immediate instructions that write a result. Show a possible format DI. This is easy for those that understand what an instruction format is. (Note that RISC-V also follows this Real Strict philosophy, but the answer to this question is not an exact copy of a RISC-V instruction format.)

☑ Show a possible format DI.

Solution appears below. There are two new fields, `immhi` and `immlo`, both are used for the 16-bit immediate.



(*b*) Convert the MIPS implementation below into an RSMIPS that works with format DI, format I, and format R RSMIPS instructions as requested in the checkbox items below. The illustration in SVG format can be found at `https://www.ece.lsu.edu/ee4720/2025/hw04-rsmips.svg`. It can be modified with your favorite SVG editor, even if it's not Inkscape.

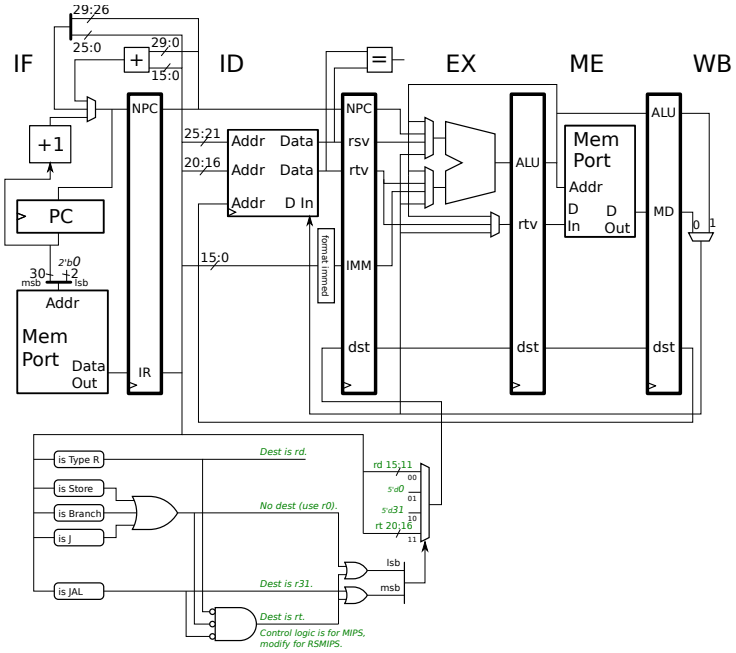
☑ Modify the control logic to extract the correct destination register.

☑ Modify the datapath and control logic to provide the correct immediate.

☑ Be sure that the logic works with RSMIPS' format I, DI, and R instructions.

The solution appears on the lower part of the next page. The low 11 bits of the **format immed** input are always connected to bits 10:0 of the instruction. For the remaining bits of the **format immed** input a multiplexor selects either bits 20:16 (for format DI instructions) or bits 15:11 (for format I instructions). The **No dest** logic (finishing with the big OR gate) detects format I (and harmlessly format J), and is used as a select signal for the new blue **format immed** mux.

Since the destination register now never comes from the (now non-existent) **rt** field, that input to the **dst** mux was removed, as was the control logic selecting the **rt** input. This simplifies the remaining control logic: the is Type R is no longer needed and the two-input OR gates are now just wire.



Original above, solution below.