Name Solution

Computer Architecture
LSU EE 4720
Midterm Solve-Home Examination

Friday, 26 March 2021 to Monday, 29 March 2021 16:00 CDT

Work on this exam alone. Regular class resources, such as notes, papers, documentation, and code, can be used to find solutions. Do not discuss this exam with classmates or anyone else, except questions or concerns about problems should be directed to Dr. Koppelman.

Problem 1		(30 pts)
Problem 2		(15 pts)
Problem 3		(15 pts)
Problem 4		(40 pts)
Exam Total		(100 pts)

☣ $V(\,[\,\mathrm{mRNA}\,|\,\mathrm{aV}\,]\,) \wedge r \geq 2\,\mathrm{m} \quad \Rightarrow \quad R_e < 1$

Good Luck!

Problem 1: [30 pts] One instruction that might have come in handy for Homework 2 is the proposed `lbit`, load bit, instruction. Consider `lbit r1, (r2..r3)`. This instruction will load a single bit from memory into `r1`. Register `r2` holds a base address and `r3` holds a bit offset. The bit offset is relative to the most-significant bit of the byte at address `r2`. So if `r3` is zero the MSB is loaded into `r1`. If `r3` is 7 the LSB of the byte at `r2` is loaded into `r1`, if `r3` is 8 the MSB of the byte at `r2+1` is loaded into `r1`, etc. (As with Homework 1 and 2, bit ordering is big-endian.) To help understanding `lbit` there are two code fragments below. They do the same thing, the first uses `lbit`, the second uses existing MIPS instructions.

```
# Proposed Instruction
lbit r1, (r2..r3)

# Equivalent MIPS Code
sra r9, r3, 3
add r9, r2, r9
lbu r1, 0(r9)
sll r1, r1, 24
andi r9, r3, 0x7
sllv r1, r1, r9
srl r1, r1, 31
```

(*a*) Modify the illustrated MIPS implementation so that it implements `lbit`, omitting control logic. Assume that the memory port will be set to perform a read byte unsigned operation (the same operation as would be performed for the `lbu` instruction) and the ALU will be set to perform an add operation. (That is, don't assume or try to add new operations for the memory port nor for the ALU.) The modifications should provide the appropriate address to the memory port and should place the appropriate bit in the destination register.

As always, assume that the critical path is through the memory port. For this problem it is okay to put additional non-control logic in the `WB` stage.

☑ Add logic to compute the correct load address.

☑ Add logic to extract the needed bit.

☑ There is no need to show control logic.

☑ Don't assume or implement new Mem Port or ALU operations.

☑ It's okay to add logic to the `WB` stage.

☑ Pay attention to performance.

☑ Pay attention to cost. ☑ Do not show functional units that are more complicated than necessary. ☑ Use existing pipeline latches and other data carrying paths when possible.

☑ As always, do not break other instructions.

The solution appears on the next page.

The solution to parts a, b, and c appears below in blue and part-c-specific (`ebit`) changes appear in green.

For `lbit` assume that control logic is set initially for an `lbu` instruction. To actually implement `lbit` some control logic would need to be changed, but that's not part of this problem. In the EX stage we need to change how the address is computed. An `lbu` would compute `rsv+IMM`, but for `lbit` we need `rsv+rtv/8`. The added logic computes `rtv/8` and connects the value to a new input on the lower ALU mux. Note that `rtv/8` is computed by effectively shifting to the right by 3 bits while sign extending. (With sign extension `rtv` can be negative.) Note that the shifting itself uses no hardware, it just relabels bit positions. The only added hardware in EX is the new mux input.

Note that the value of `rtv` is taken from the output of the mux used by store instructions. By doing so we can get bypassed values without having to add new hardware. Bypassed values are needed for part b.

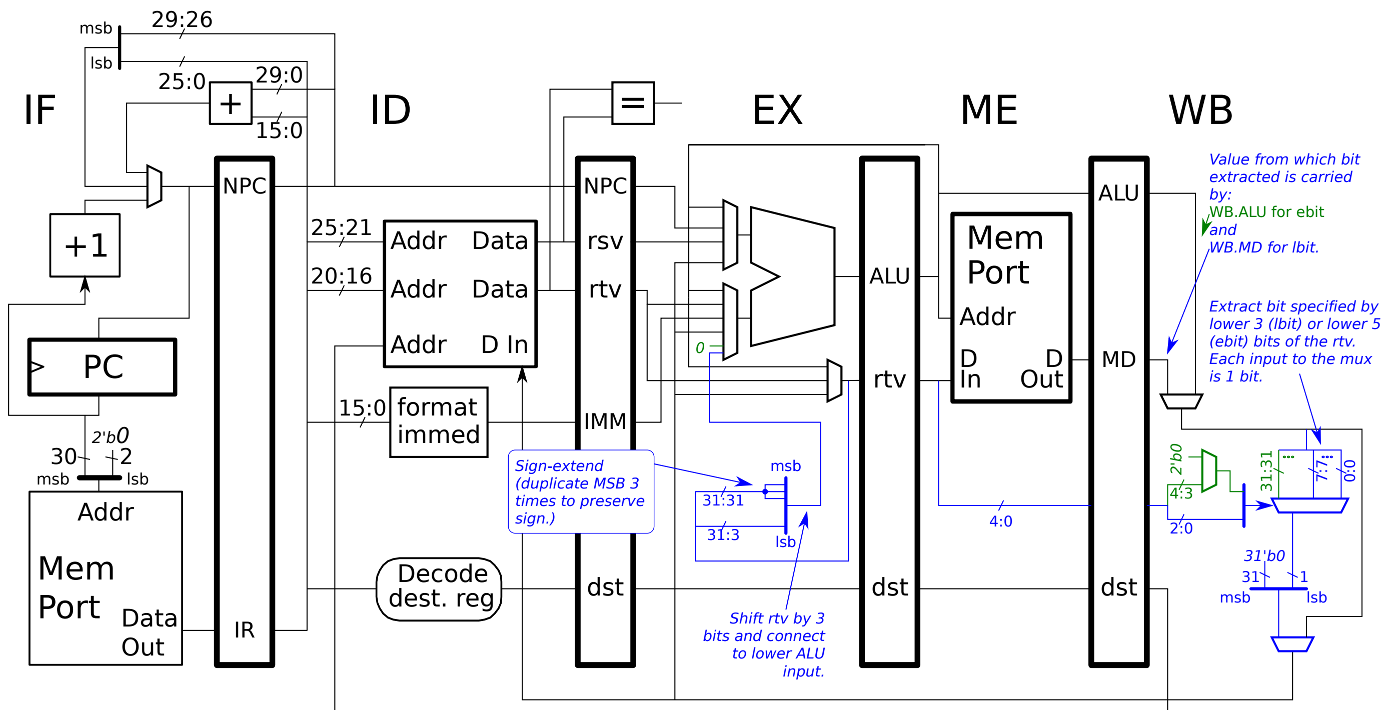
In WB a mux is used to extract the needed bit, it will be called the *bit-extract mux* in this discussion. (That's the mux with more than two inputs.) For `lbit` the mux needs 8 inputs, each of one bit, connected to the 8 LSB of `WB.MD`. The hardware shown, rather than connecting directly to `WB.MD`, connects to the mux that selects either `WB.MD` or `WB.ALU`. This is needed for part c. Also, for part c the bit extract mux has 32 inputs.

The select signal for the bit-extract mux is taken from the LSB of `rtv`. In the existing hardware `rtv` only makes it as far as ME. So for this problem a new pipeline latch is added carrying the 5 LSB of `rtv` to WB. For part a only the 3 LSB are used, for part c all 5 bits are used.

The output of the bit-extract mux is just 1 bit. Thirty-one zeros are appended to make a 32-bit quantity.

For part c, in which `ebit` is implemented, there are several differences with the `lbit` implementation. First, we need to deliver the `rsv` to WB. To do so without requiring a new ALU operation, a new zero input to the lower mux has been added. The ALU will perform an add operation and use the zero input, so the output of the ALU will be `rsv`. In WB the `rsv` is in `WB.ALU`, and as mentioned earlier, it has a path to the bit extract mux. The select signal needs to be 5 bits for `ebit`, so a mux is used to set the 2 MSB of the bit-extract mux's select signal: to zero for `lbit` (valid values are 7-0) and for `ebit` use bits `4:3` of `rtv` so we can get a range of 31-0 at the select signal.

Common Mistakes: One common mistake was to use the equivalent MIPS code as a blueprint for the hardware. That equivalent code was provided to be clear on *what* the `lbit` instruction should do, not *how* it should do it. So a solution with a box for each instruction in the equivalent code, such as a four shift-unit boxes for `sra`, `sll`, `sllv`, and `srl` would be very wasteful.



(*b*) Show the execution of the code fragments below on your implementation. Add reasonable bypass paths to eliminate stalls.

☑ Add reasonable bypass paths to avoid stalls that would be suffered by the code below.

☑ Show execution of each code fragment (with reasonable bypass paths).

The executions appear below, with highlighting used to emphasize registers carrying dependencies.

```
## SOLUTION
# Fragment A        0  1  2  3  4  5  6  7
addi R3, r3, 1      IF ID EX ME WB
lbit R1, (r2..R3)      IF ID EX ME WB
add r4, r4, R1            IF ID -> EX ME WB

## SOLUTION
# Fragment B        0  1  2  3  4  5  6
lbit R1, (r2..r3)   IF ID EX ME WB
addi r3, r3, 1         IF ID EX ME WB
add r4, r4, R1            IF ID EX ME WB

## SOLUTION
# Fragment C        0  1  2  3  4  5  6  7  8  9
lbit R1, (r2..r3)   IF ID EX ME WB
bne R1, r0  TARG       IF ID ----> EX ME WB
addi r3, r3, 1            IF ----> ID EX ME WB

TARG:
xor r8, r9, r10                     IF ID EX ME WB
# Cycle:            0  1  2  3  4  5  6  7  8  9
```

(*c*) Consider another instruction `ebit`, extract bit. Consider `ebit r1, r2, r3`. This instruction extracts the bit at position `r3` from `r2` and writes it to `r1`. The MSB is at position 0. The bit position is in the least significant five bits of `r3`, other bits of `r3` are ignored.

Both `lbit` and `ebit` extract a bit from a value, so it is possible to use some of the hardware for `lbit` to implement `ebit`. One difference is that `lbit` extracts an 8-bit quantity while `ebit` extracts a bit from a 32-bit quantity. If the hardware were shared, the `lbit` hardware would have to be upgraded to handle 32-bit values.

Ignoring whether such sharing really is a good idea, modify the implementation of `lbit` so that it could implement `ebit` using hardware shared with `lbit`.

☑ Modify MIPS hardware to implement `ebit` using hardware shared with `lbit`.

☑ No need to show control logic.

The solution to this part is shown and discussed several pages back on the page showing the hardware changes.

(*d*) Explain why an implementation sharing `ebit` and `lbit` hardware would execute the code fragment below slowly and describe a faster alternative.

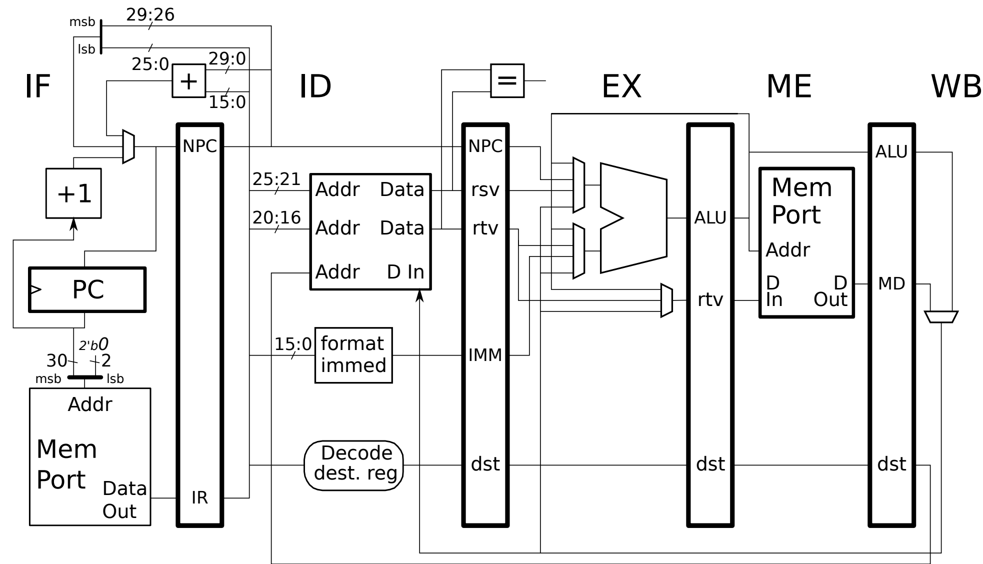
```
ebit r1, r2, r3
add r4, r4, r1
```

☑ Why does the shared hardware implementation slow code below?

☑ Why is an implementation of `ebit` that is similar to other computation instructions faster?

It is slower because the result is computed in `WB`. This will force a dependent instruction, such as the `add` in the example above to stall one cycle. If a bit-extraction mux for `ebit` had been placed in `EX` then it would be possible to bypass in the example above without a stall.

Problem 2: [15 pts] Consider the pointer-chasing loop below. Assume that the loop executes many iterations on the illustrated hardware.



(*a*) Show an execution of the loop below for enough iterations—at least two—to compute the IPC (inverse of CPI). The IPC is the number of executed instructions divided by the number of cycles. Compute it for a very large number of iterations.

☑ Show execution.

☑ Compute IPC for a large number of iterations.

☑ Check for dependencies and available bypass paths.

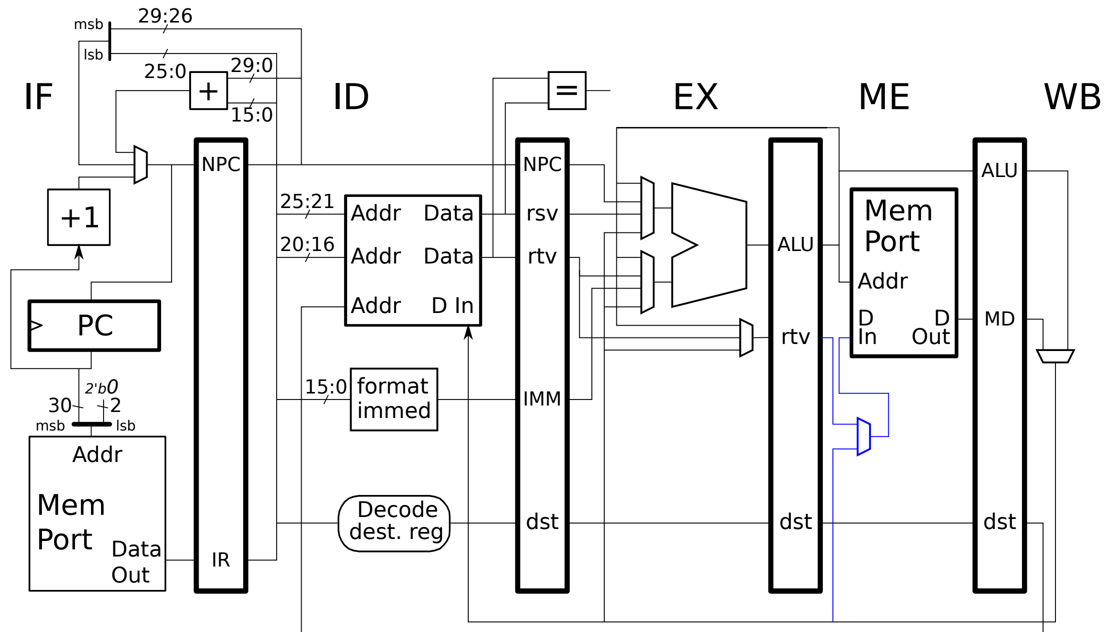
The execution appears below. The first iteration starts in cycle 1, the second starts in cycle 7, so the time for an iteration is $7 - 1 = 6$ cycles. (The start time is when the first instruction is in `IF`.) Since there are 4 instructions in the loop body the IPC is $\frac{4}{6} = \frac{2}{3}$ insn/cycle.

Common Mistakes: One common mistake is forgetting the delay slot instruction (`lw r3`).

```
 ## SOLUTION
 lw r3, 8(r3)      IF ID EX ME WB                              BEFORE LOOP
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 4(r3)         IF ID -> EX ME WB                        FIRST ITERATION
 sw r1, 0(r3)            IF -> ID -> EX ME WB
 bne r1, r5 LOOP               IF -> ID EX ME WB
 lw r3, 8(r3)                        IF ID EX ME WB
LOOP:  # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 4(r3)                           IF ID -> EX ME WB      SECOND ITERATION
 sw r1, 0(r3)                              IF -> ID -> EX ME WB
 bne r1, r5 LOOP                                 IF -> ID EX ME WB
 lw r3, 8(r3)                                          IF ID EX ME WB
       # Cycle     0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16

 add r5, r3, r9                                                AFTER LOOP
```

(*b*) If the previous part were solved correctly, then there should be two stalls per iteration. One stall could be eliminated by a bypass path, but the other could not (without increasing critical path). For each stall in your execution (even if there are more or less than two) show a reasonable bypass path that would avoid the stall or else explain why such a bypass is not reasonable.



☑ Show reasonable bypass paths needed to avoid stalls on your code.

☑ For each stall that could not be eliminated with a bypass path, explain why:

As stated in the problem, the code suffers two stalls. The first stall is due to a `lw/lw` dependence. On the execution above it ends in cycle 3 in the first iteration and 9 in the second iteration. (A stall ends at the arrow head in the diagram.)

The second stall is due to the `lw/sw` dependence. That ends at cycle 5 in the first iteration.

To examine which bypass paths are possible consider the stall-less, hypothetical, **not necessarily correct** execution below.

```
## WARNING: HYPOTHETICAL EXECUTION WITHOUT STALLS. WILL NOT RUN CORRECTLY.
 lw r3, 8(r3)        IF ID EX ME WB                                   BEFORE LOOP
LOOP:   # Cycle      0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
 lw r1, 4(r3)           IF ID EX ME WB
 sw r1, 0(r3)              IF ID EX ME WB
```

Consider first the `lw r3` / `lw r1` stall. In the hypothetical stall-less execution above the `lw r1` needs the value of `r3` in the beginning of cycle 3 when it is in EX. But at that time `lw r3` is just starting ME and so the value has not been loaded. So there is no reasonable way to bypass the value.

Next, consider the `lw/sw` dependence, carried by `r1`. The `sw` needs `r1` at the beginning of its time in the ME stage in cycle 5. At that time the `lw` is in WB, and so it is possible to bypass it. That bypass path is shown in blue. It has been pointed out that the Mem Port is assumed to be on the critical path. But the critical path is from the `Addr` input to the `D Out` output. We can assume that there is some slack on the `D In` input, meaning that it can be delayed for a short time, which is why the mux is not a problem. It would be a problem to put a mux before the Mem Port `Addr` input or after the Mem Port `D Out` output.

Problem 3: [15 pts] Appearing below are two candidate MIPS instructions, `jca`, *jump case add*, and `jcc`, *jump case concatenate*, that can be used to implement C-style `switch` statements. The instructions are designed for case statements that each consist of up to eight instructions. In both instructions the `rs` register (register `r1` in the examples) holds the address of case statement zero. Case statement 1 starts at address `r1+32`, case statement 2 starts at address `r1+32*2`, etc. The `rt` register (`r2` in the examples) holds the number of the case statement to jump to, so the address to jump to is `r1+32*r2`. The only difference between the two instructions is that in `jca` the value of `r1` must be a multiple of 4 (since instruction addresses are aligned) while in `jcc` the value of `r1` must be a multiple of 4096 (the 12 least-significant bits must be zero) and `r2` must be less than 128. Like other MIPS control transfers, both instructions have a 1-instruction delay slot. Note that `jca r1, r0` is equivalent to `jr r1`.

The code below shows the use of `jca` and an equivalent code fragment that uses only existing MIPS instructions.

```
# Candidate Instruction
jca r1, r2   # Jump to r1 + r2 * 32
nop

# Another Candidate Instruction
jcc r1, r2   # Jump to { r1[31:12], r2[6:0], 5'b0 }
nop

# Equivalent code to jca (and partly jcc) using existing MIPS instructions.
sll r9, r2, 5
add r9, r9, r1
jr r9
nop
```

A resolve-in-`ID` implementation of `jcc` can be designed at low cost and with no risk of lengthening the critical path. In contrast, a resolve-in-`ID` implementation of `jca` would add to cost and risk critical path impact.

(*a*) Show the datapath changes to the MIPS pipeline on the next page needed for resolve-in-`ID` implementations of the two instructions.

☑ Show datapath changes (not control logic) for resolve-in-`ID` implementation of ☑ `jca` and ☑ `jcc`.

☑ As always, pay attention to cost and performance.

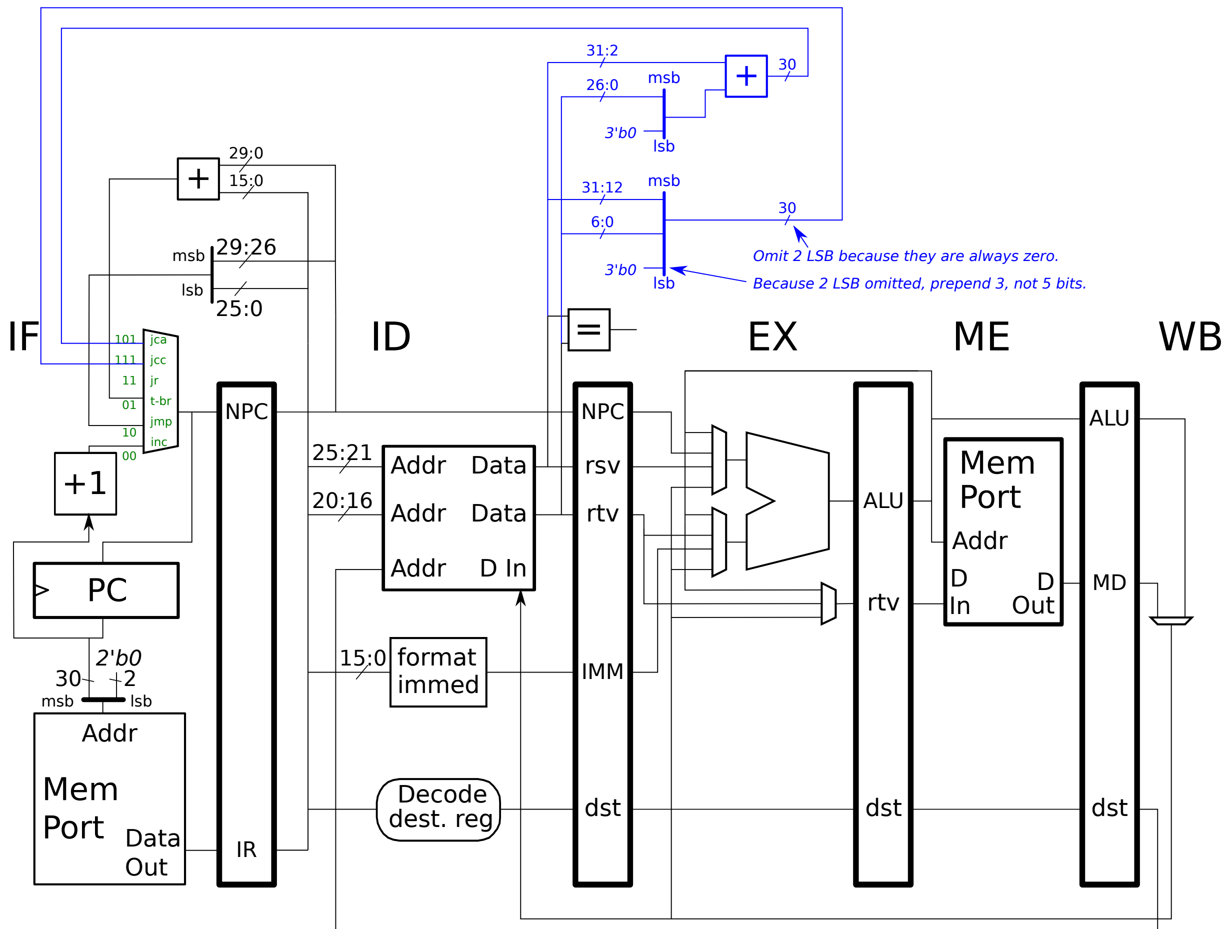
The solution and a discussion of the solution appears on the next page.

(*b*) Explain why computing a branch target, which is done using an adder, has no critical path impact while there is critical path impact for `jca`.

☑ Why can a branch safely use an adder in `ID`, but not `jca`?

Because the inputs to the adder for the branch are available at the beginning of the clock cycle, in contrast the inputs to the adder for `jca` are available later, after they are retrieved from the register file. Because of this one cannot automatically assume that there is enough time for the `jca` adder to compute its sum.

Solution to part a appears below in blue. Note that since the PC holds only 30 bits, it is not necessary to compute the two LSB of the target. (They would always be zero anyway.) For that reason 3 bits, instead of 5 bits, are prepended to the `rt` register value.



Problem 4: [40 pts] Answer each question below.

(*a*) MIPS branches have one delay slot. That enables five-stage scalar MIPS implementations to fetch the delay-slot instruction while resolving the branch. So, is the delay slot a feature of the ISA or a feature of the implementation?

☑ Is a delay slot an ISA feature or an implementation feature? ☑ Explain.

It's an ISA feature because it describes what instructions do. That is, the feature says that the instruction after a branch executes whether or not the branch is taken. Because of this ISA feature some implementations, including our five-stage MIPS pipeline, execute branches with zero penalty. But for others, such as superscalar pipelines (which will be covered soon), a delay slot adds to complexity and provides little benefit. Because it is an ISA feature, a superscalar implementation must execute the one delay slot instruction, even if a different number of delay slots would have made more sense.

(*b*) There are 32 MIPS integer (general-purpose) registers, usually called `r0` to `r31`. But these registers are also given names, which are shown in the table below. Suppose we wanted to rearrange the names. For example, suppose we wanted to name register `r16 t8` (instead of name `r24 t8`) and make `r24` the new `k0`. Which registers could we rearrange without changing the ISA? It must be possible to use the registers for the purpose suggested by their names after rearranging.

```
Names      Numbers Suggested Usage
$zero:     0       The constant zero.
$at:       1       Reserved for assembler.
$v0-$v1:   2-3     Return value
$a0-$a3:   4-7     Argument
$t0-$t7:   8-15    Temporary (Not preserved by callee.)
$s0-$s7:  16-23    Saved by callee.
$t8-$t9:  24-25    Temporary (Not preserved by callee.)
$k0-$k1:  26-27    Reserved for kernel (operating system).
$gp       28       Global Pointer
$sp       29       Stack Pointer
$fp       30       Frame Pointer
$ra:      31       Return address.
```

☑ Which register numbers can get new names without having to change the ISA? ☑ Explain.

The ISA dictates special behavior for `r0` (it's always zero) and for `r31` (the `jal` instruction writes it with return address). There is no special behavior for the other registers so they can be renamed. That is, there's no problem with making `r1-r4` the argument registers and `r5-r6` the return value registers.

Common Mistake: Some incorrectly supposed that the important-sounding registers such as `sp` and `at` could not be renamed. When it comes to the ABI (which includes rules for how registers are used when making procedure calls) `sp` is important, but no more important than `t0` or `s0`. But there is nothing about the MIPS ISA that makes `r29` a good choice for the stack pointer (`sp`), any other register except `r0` and `r31` could be chosen.

(*c*) The code fragment below adds 1 to the floating-point value in register `f2` and puts the sum in `f3`.

```
addi $t1, $0, 1     # Integer 1
mtc1 $t1, $f1
cvt.s.w $f1, $f1
add.s $f3, $f1, $f2
```

☑ Explain what the `cvt.s.w` instruction does in the code above.

The instruction converts the value in the `fs` register (`f1` in the example) from a signed integer to a single-precision floating-point value and writes the result in the `fd` register (also `f1` in the example, but it could be different than the source).

☑ Re-write the code so that it adds a 1, but does so without a `cvt` instruction. *Note: The original exam used the wording "without a* `cvt.s.w` *instruction.".*

```
## SOLUTION
lui $t1, 0x3f80    # Note: An IEEE 754 Single 1.0 = 0x3f800000
mtc1 $t1, $f1
add.s $f3, $f1, $f2
```

(*d*) In early RISC ISAs, including MIPS I, floating-point registers were 32 bits and yet many of these ISAs had double-precision (64-bit) floating-point instructions. Where do these instructions find their 64-bit operands?

☑ MIPS-I gets 64-bit floating-point operands ...

... from a pair of registers consecutive. The instruction specifies the even-numbered register of the pair. Consider `add.d f0, f2, f8`. The first source is in registers `f2` and `f3`, the second source is in `f8` and `f9`. Registers `f0` and `f1` are written with the result.

(*e*) ARM A64 and RISC-V RV64 are both late RISC ISAs. But ARM A64 has many more instructions than RISC-V. How does having more instructions help A64 and fewer instructions help RISC-V?

☑ Lots of instructions help A64 because ...

They enable programs to be written with fewer instructions. This reduces program size and presumably reduces execution time and execution energy. (Certainly reduced execution time and energy were a goal, since A64 is successful we presume that these goals have been reached in many A64 implementations.)

☑ Fewer instructions help RISC-V because ...

... it was designed for teaching research purposes and having fewer instructions would reduce the difficulty of carrying out investigations. (Though intended for teaching and research an important design goal is that it be complete enough to be used in practical applications.)

Some pointed out that RISC-V could have fewer instructions in its base version, say RV64I, because its modular design enabled additional instructions to be added only by those who needed them. Even so, RISC-V is still much simpler than A64. Bit extraction instructions are still in draft form, and there is nothing like A64's scaled indexing. Though not 100% correct, an answer along these lines would get full credit.

Some pointed out that RISC-V was intended for embedded applications where low hardware cost is important and easier to achieve with fewer instructions. That's not a bad argument, but it ignores several contributors to cost: the core itself (such as our pipeline), program storage, and execution energy. With a smaller instruction set comes lower-cost cores. But programs will be longer, requiring more storage and potentially more energy executing them. That's only a problem if the code for the embedded application is beyond a certain size and if energy is an issue.

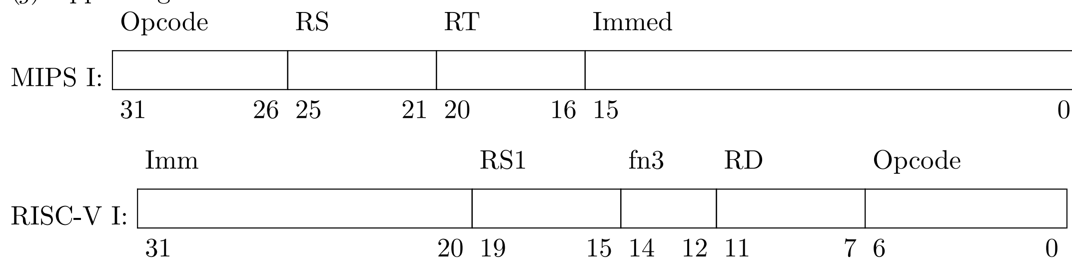
(*f*) Explain the problem with this statement:

Implementations of CISC ISAs were slow because of complex instructions. Only later did computer engineers discover that with simpler RISC ISAs implementations could be made faster.

☑ The statement is misleading or incorrect because ...

... it ignores that the implementation technology used when CISC ISAs were developed was much more costly, so pipelined designs (for which RISC ISAs were intended) were out of the question and memory could not be wasted. For that reason an implementation of a RISC ISA would be slower than an implementation of a CISC ISA back then. Those complex instructions helped reduce redundant work (such as reading and writing registers or memory). In a non-pipelined implementation complex instructions are not particularly difficult to implement. For example, and arithmetic instruction could retrieve its operands from memory, compute arithmetic on them, and store them back in memory. There's no problem with complex memory addressing modes, because a single ALU could be used as many times as needed.

(*g*) Appearing below are MIPS and RISC ISAs' immediate formats.



☑ What advantage does the MIPS format have?

☑ Show an example of a MIPS instruction that could not be encoded in the RISC-V format.

The MIPS format has a larger immediate field and so it can be used with larger immediate values. For example, the MIPS instruction `addi r1, r2, 0x1234` has no RISC-V counterpart. That is `addi a1, a2, 0x1234` is not a RISC-V instruction because the immediate would not fit.

☑ What advantage does the RISC-V format have? (Another question on this exam implies this advantage is wasted.)

There are more opcode bits in the RISC-V format, a total of 10 and so based on this format there can be more format-I opcodes than in MIPS.

(*h*) Compilers optimize by scheduling (rearranging) instructions to avoid stalls due to true dependencies. In that case, why do we need to have bypass paths?

☑ Bypass paths are needed despite optimizations because:

Because it is not always possible to find an instruction to put between a dependent pair to avoid a stall. This is particularly difficult in code with frequent branches. Suppose, for example, one out of six instructions were a branch. Then scheduling could only easily be done with about five instructions, making it difficult to find instructions to move.