Name ______________________________

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.

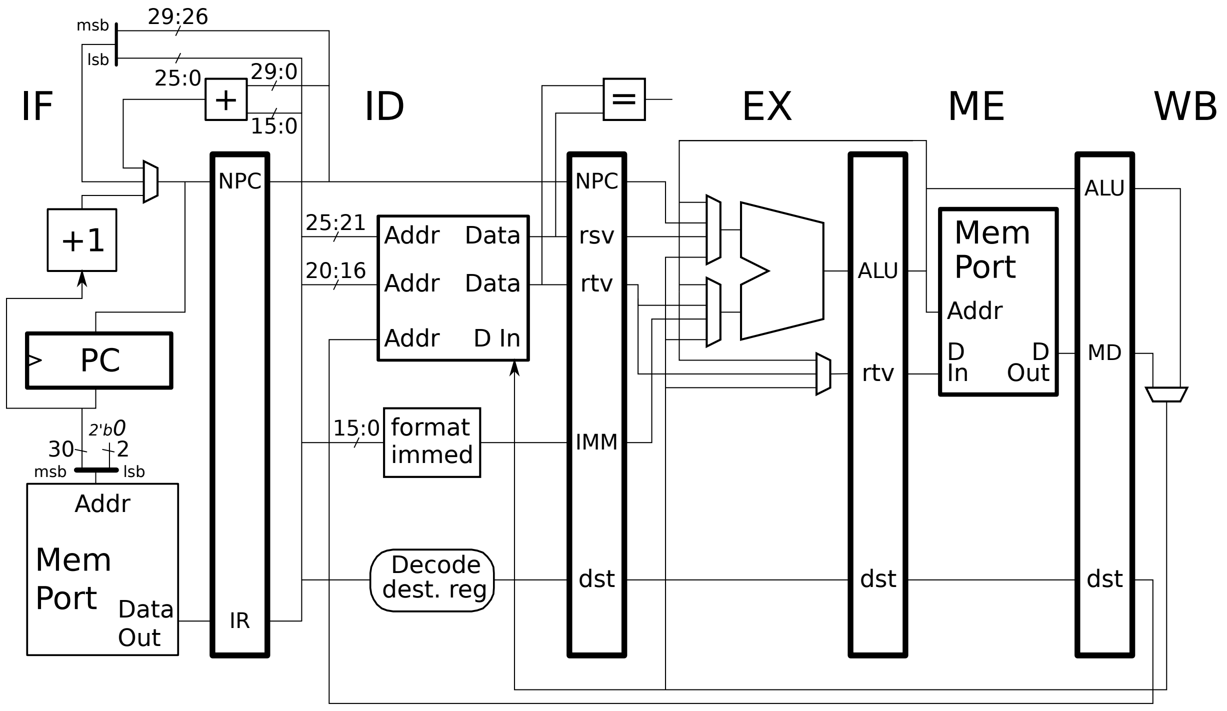
☐ Don't assume or implement new Mem Port or ALU operations.

☐ 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.


msb
29:26
lsb
25:0
29:0
15:0
IF
ID
EX
ME
WB
NPC
+1
PC
2'b0
30
2
msb
lsb
Addr
Mem
Port
Data
Out
IR
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format
immed
Decode
dest. reg
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem
Port
Addr
D
In
D
Out
ALU
MD
dst

(*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).

```
# Fragment A
addi r3, r3, 1

lbit r1, (r2..r3)

add r4, r4, r1



# Fragment B
lbit r1, (r2..r3)

addi r3, r3, 1

add r4, r4, r1



# Fragment C
lbit r1, (r2..r3)

bne r1, r0  TARG

addi r3, r3, 1



TARG:
 xor r8, r9, r10
```

(*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.

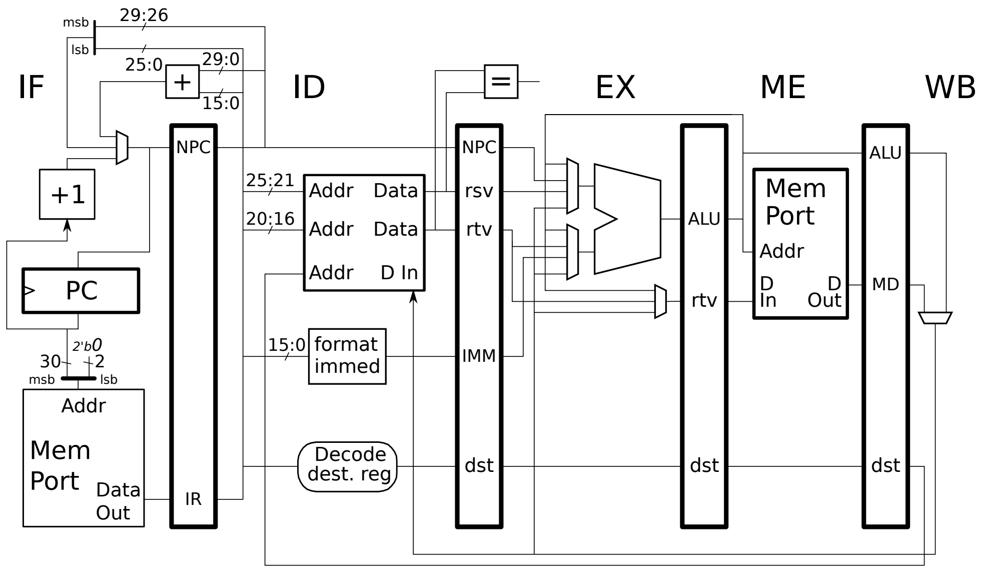
(*d*) Explain why an implementation sharing `ebit` and `lbit` hardware would execute the code fragment below slowly and describe a faster alternative.

```
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?

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.

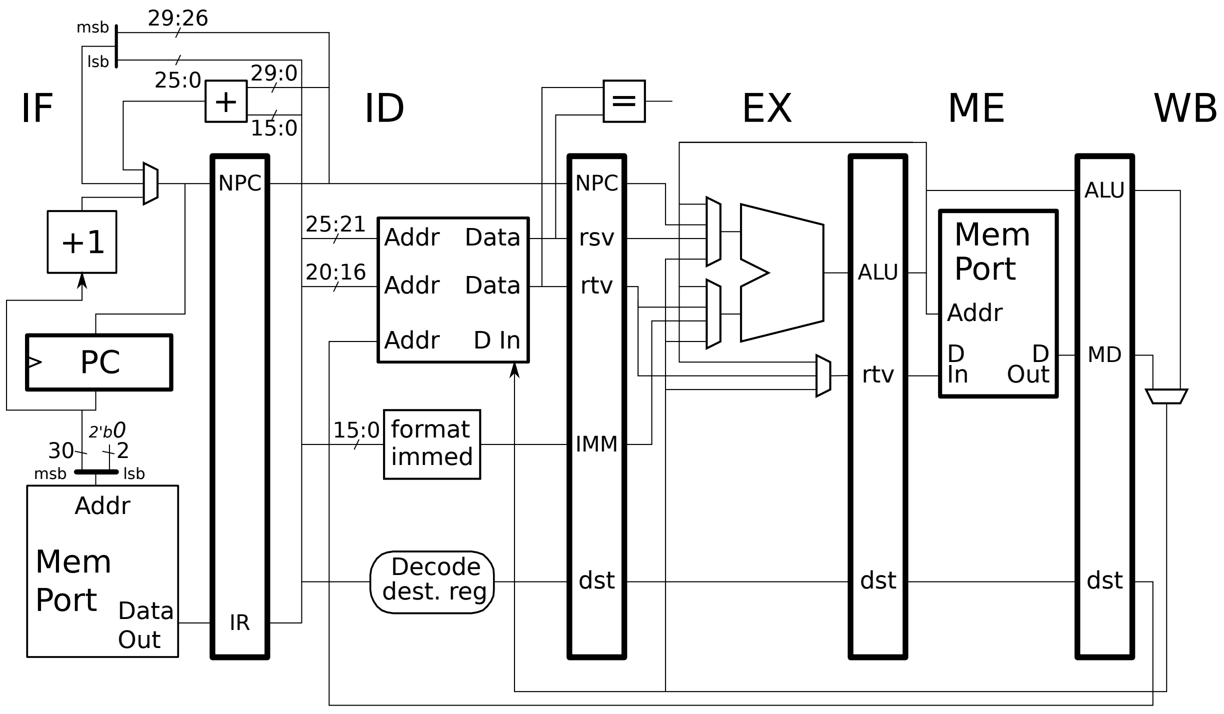
☐ Show execution.

☐ Compute IPC for a large number of iterations.

☐ Check for dependencies and available bypass paths.

```
 lw r3, 8(r3)
LOOP:
 lw r1, 4(r3)

 sw r1, 0(r3)

 bne r1, r5 LOOP

 lw r3, 8(r3)

 add r5, r3, r9
```

(*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:

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.

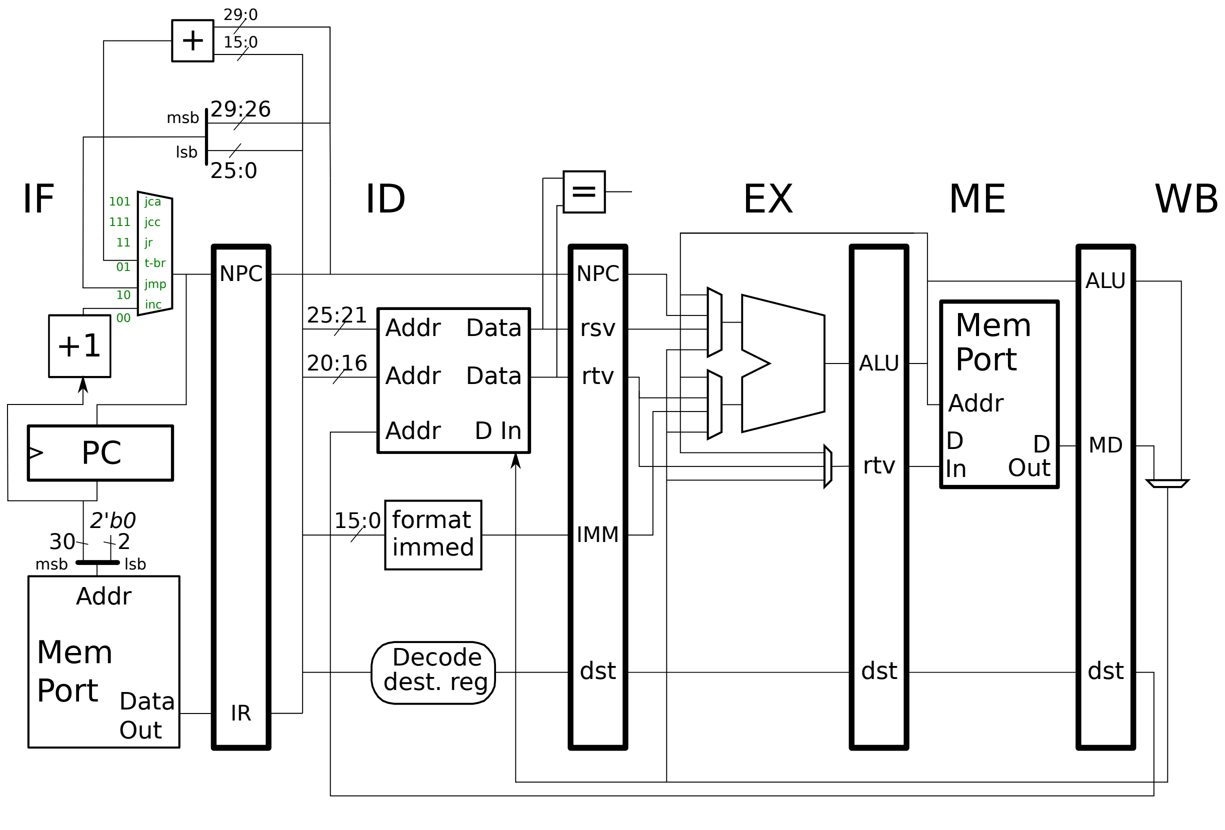
(*a*) Show the datapath changes to the MIPS pipeline on the next page needed for resolve-in-`ID` implementations of the two instructions.

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

☐ As always, pay attention to cost and performance.

(*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`?


IF
ID
EX
ME
WB
+
29:0
15:0
msb
29:26
lsb
25:0
101
111
11
01
10
00
jca
jcc
jr
t-br
jmp
inc
+1
PC
2'b0
30
2
msb
lsb
Addr
Mem Port
Data Out
NPC
IR
25:21
20:16
Addr
Data
Addr
Data
Addr
D In
15:0
format immed
Decode dest. reg
=
NPC
rsv
rtv
IMM
dst
ALU
rtv
dst
Mem Port
Addr
D In
D Out
ALU
MD
dst

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.

(*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.

(*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.

☐ Re-write the code so that it adds a 1, but does so without the `cvt.s.w` instruction.

(*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 . . .

(*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 . . .

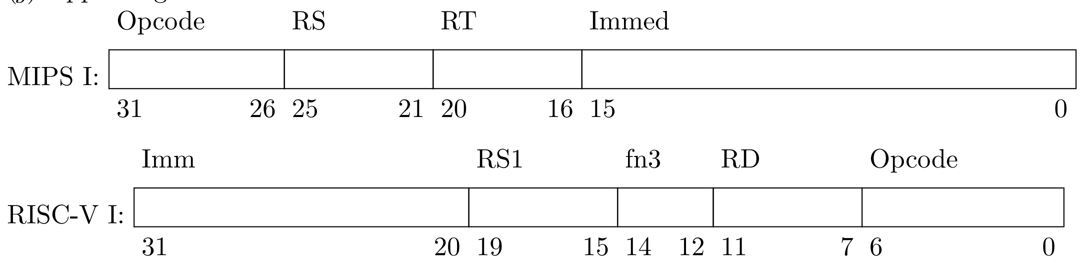
☐ Fewer of instructions help RISC-V because . . .

(*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 . . .

(*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.

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

(*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: