UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Fall 2019

Midterm 1

Name: _____

Instructions:

1. This exam is open-note, open-book. Calculators are allowed.
2. No devices that can communicate (laptops, phones, or PDAs) allowed, with the exception of a device that can read your e-book textbook, on which you must turn off your wifi and/or cellular connections. Other than "find", no typing. Turn off all phones, pagers, and alarms.
3. *You may not share any materials with anyone else, including books, notes, lab reports, datasheets, calculators, and most importantly, answers!*
4. This exam is designed to be a 60 minute exam, but you have 110 minutes to complete it. (Roughly) one point is budgeted per minute. Use your time wisely!

Excerpts from the UC Davis Code of Academic Conduct:

1. Each student should act with personal honesty at all times.
2. Each student should act with fairness to others in the class. That means, for example, that when taking an examination, students should not seek an unfair advantage over classmates through cheating or other dishonest behavior.
3. Students should take group as well as individual responsibility for honorable behavior.

I understand the honor code and agree to be bound by it.

Signature: _____

Page:	2	3	4	5	6	9	Total
Points:	8	12	12	8	10	15	65
Score:							

All instructions in this exam, unless otherwise noted, are RISC-V instructions. Recall that if the instruction writes to a register, that register appears first in the instruction. Unless otherwise indicated, you may not use any RISC-V pseudoinstructions on this exam, only actual RISC-V instructions. Feel free to detach the two RISC-V reference pages and the adder reference page at the end of the exam, but if you do, please take them with you and recycle them (don't try to reattach them or turn them in).

1. Many machine learning applications do not require the precision or range of single-precision (32b) floating point. Thus Google has introduced a new 16-bit floating-point format, `bfloat16`, which is structured in a similar way to the IEEE 754 formats we discussed in class. `bfloat16` has one sign bit, an 8-bit exponent, and a 7-bit significand (the bits after the 1.).

   (a) (5 points) **What is the smallest normal positive number that can be represented as a `bfloat16`?** ("Normal" here means "ignore subnormal numbers".) Express your answer as a normalized number $\pm a.b \times 2^c$, where you specify the sign, a, b, and c. You should clearly show your work if you want partial credit.

   (b) NVIDIA hardware also supports a 16-bit floating-point format, `half`. It has 1 sign bit, 5 exponent bits, and 11 significand bits.

   Answer each of the following questions with `bfloat16` or `half`.

   i. (1 point) The format with the largest representable value is (`bfloat1` or `half`):

   ii. (1 point) The format with a representable value closest to zero is (`bfloat1` or `half`):

   iii. (1 point) The format with the smallest range between two adjacent values when the bias-adjusted exponent is zero (meaning the normalized number is $1.x \times 2^0$) (where, for these two adjacent values, the floating-point representation is identical except for the least significant bit of the significand) is (`bfloat1` or `half`):

2. Short code sequences.

   (a) (6 points) MIPS has the instruction `nor`. RISC-V lacks this instruction. **Implement `nor x2, x3, x4` in as few RISC-V instructions as possible.** You may not make any changes to any register except `x2`.

   (b) (6 points) RISC-V also lacks the instruction `sgtz`, which sets its output register to 1 if the input register is larger than zero and to 0 otherwise. **Implement `sgtz x2, x3` in as few RISC-V instructions as possible.** You may not make any changes to any register except `x2`.

(c) (12 points) RISC-V has a floating-point instruction called `fsgnj.d rd, rs1, rs2`, "floating-point sign injection". It returns a number whose magnitude is the same as its `rs1` argument but whose sign is the same as its `rs2` argument. Note it works on double-precision (64b) data.

Implement `fsgnj.d x2, x3, x4` in as few *integer* RISC-V instructions as possible. *Your inputs are 64b floating-point numbers stored in 64b integer registers and your output is also a 64b floating-point number in a 64b integer register. Do* ***not*** *use floating-point registers or instructions to solve this problem.* You may use `x10`, `x11`, etc. as temporary registers if needed. For this part only, you may use the pseudoinstruction `li` (load immediate), which will count as one instruction.

To aid grading (to help give you partial credit), describing your approach in a sentence or two will be helpful.

3. (8 points) Let's consider adding a reciprocal operation to RISC-V. Your engineers have determined that the primary use for a reciprocal operation is to normalize 3-component vectors:

$$x_{\text{new}} = \frac{x}{\sqrt{x^2 + y^2 + z^2}},$$

with y_{new} and z_{new} defined analogously. Note we could take the reciprocal of the square root once and then perform multiply instructions to achieve the same results. To compute x_{new}, y_{new}, and z_{new}, we could replace divide operations with reciprocal and multiply instructions.

If we assume that add instructions take 1 cycle, multiply operations take 4 cycles, and both square-root and divide operations take 20 cycles, **what is the maximum number of cycles for the reciprocal operation so that 3-component vector normalization is at least as fast using reciprocal and multiply as it was using division?**

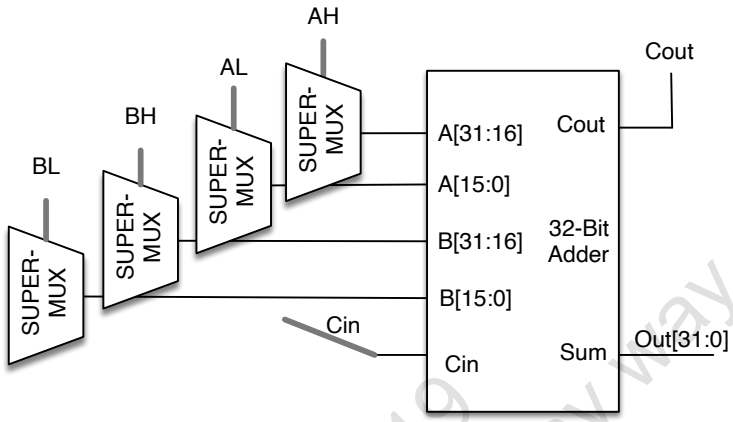
4. You have simplified the design of your new RISC-Y processor to the point where you only have two classes of instructions, A-type (which takes 1 cycle to finish) and B-type (which takes 3). You compile your company's most important benchmark on your internal compiler and find it has a CPI of 2.8.

   The Team Rocket compiler company visits your company and boasts they can achieve twice the MIPS (millions of instructions per second) on your benchmark on your RISC-Y processor. You test their compiler: it's true, they can. However, the runtime of their compiled code is the same as yours.

   (a) (7 points) What is the instruction mix (what percentage of A-type and what percentage of B-type) of the compiled code from the Team Rocket compiler?

   (b) (3 points) If Team Rocket's code gets twice as many MIPS, **why does it run at the same speed as your internal compiler's?** Be precise as to why and quantify your answer.

5. In this problem we design an integer ALU that is inspired by the POWER instruction set. For the purposes of this problem, assume that POWER supports 32-bit integers, that it has an R-type instruction format that takes two input registers and one output register, and an I-type instruction format that takes one input register, one output register, and a 16-bit immediate.



Our base hardware is a 32-bit adder that has five inputs. Four of the inputs (`AH`, `AL`, `BH`, and `BL`) specify the A and B inputs to the adder, each divided into two 16-bit chunks (the 16b high and low halves of the 32b input words to the adder). The fifth input (`Cin`) is a one-bit carry in.

Your job is to select what goes into these five inputs. For the first four inputs, you must set the control signals on a SUPERMUX, which is a 16-bit wide mux that can select from 14 16-bit inputs:

- `A[31:16]`
- `A[15:0]`
- `B[31:16]`
- `B[15:0]`
- `~A[31:16]` (invert the bits of `A[31:16]`)
- `~A[15:0]`
- `~B[31:16]`
- `~B[15:0]`
- `imm[15:0]` (the immediate from I-type instructions)
- `A[31] x 16` (bit `A[31]` copied to all 16 bits)
- `B[31] x 16` (bit `B[31]` copied to all 16 bits)
- `imm[15] x 16` (bit `imm[15]` copied to all 16 bits)
- `0 x 16` (16 zeroes)
- `1 x 16` (16 ones)

For the last control input (`Cin`), you can set it to `0` or `1`.

As an example, if I asked you to configure this ALU to compute a standard 32-bit add, you would set the following:

- `AH` = `A[31:16]`
- `AL` = `A[15:0]`
- `BH` = `B[31:16]`
- `BL` = `B[15:0]`
- `Cin` = `0`

This information is replicated at the end of the exam. Feel free to detach it and use it to answer the following questions. There are no questions on this page.

For each of the following instructions, fill in how you would set the five control signals.

(a) (3 points) `SUB`: a 32-bit subtract, `A-B`.

(b) (3 points) `LUI`: load the 16-bit immediate into the upper 16 bits of the output, setting the other output bits to zero

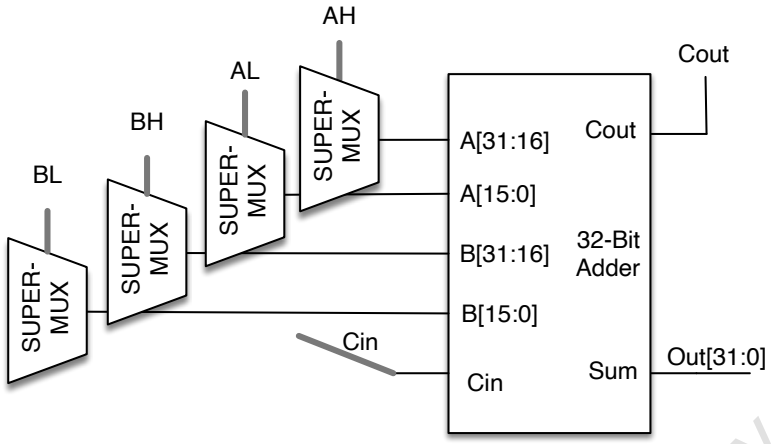
(c) (3 points) `ADDIS`: "add immediate shifted"; add input `A` to the (immediate left-shifted by 16 bits)

(d) (3 points) `DBL`: "multiply by 2"; return `A`×2

(e) (3 points) In English, what does the instruction with the following control settings do?

- `AH = 1 x 16`
- `AL = 1 x 16`
- `BH = B[15:0]`
- `BL = B[31:16]`
- `Cin = 1`

There are no questions on this page. This page is for reference only. Feel free to detach it.



SUPERMUX inputs:

- `A[31:16]`
- `A[15:0]`
- `B[31:16]`
- `B[15:0]`
- `~A[31:16]` (invert the bits of `A[31:16]`)
- `~A[15:0]`
- `~B[31:16]`
- `~B[15:0]`
- `imm[15:0]` (the immediate from I-type instructions)
- `A[31] x 16` (bit `A[31]` copied to all 16 bits)
- `B[31] x 16` (bit `B[31]` copied to all 16 bits)
- `imm[15] x 16` (bit `imm[15]` copied to all 16 bits)
- `0 x 16` (16 zeroes)
- `1 x 16` (16 ones)

For the last control input (`Cin`), you can set it to `0` or `1`.

As an example, if I asked you to configure this ALU to compute a standard 32-bit add, you would set the following: `AH` = `A[31:16]`; `AL` = `A[15:0]`; `BH` = `B[31:16]`; `BL` = `B[15:0]`; `Cin` = `0`.

RISC-V Reference

James Zhu <jameszhu@berkeley.edu>

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
add	ADD	R	0000011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0000011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0000011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0000011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0000011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0000011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0000011	0x2	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0000011	0x3	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2		rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3		rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0	0x00	rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x0	0x00	rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x0	0x00	rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x0	0x00	rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	0x00	rd = rs1 << imm	
srli	Shift Right Logical Imm	I	0010011	0x1	0x00	rd = rs1 >> imm	
srai	Shift Right Arith Imm	I	0010011	0x3	0x20	rd = rs1 >> imm	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch $\leq$	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch $\geq$ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	0x00	Transfer control to OS	imm: 0x000
ebreak	Environment Break	I	1110011	0x0	0x00	Transfer control to debugger	imm: 0x001

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	F3	F7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

Inst	Name	FMT	Opcode	F3	F5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	F3	F5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Fall 2019

Midterm 1 Solutions

All instructions in this exam, unless otherwise noted, are RISC-V instructions. Recall that if the instruction writes to a register, that register appears first in the instruction. Unless otherwise indicated, you may not use any RISC-V pseudoinstructions on this exam, only actual RISC-V instructions. Feel free to detach the two RISC-V reference pages and the adder reference page at the end of the exam, but if you do, please take them with you and recycle them (don't try to reattach them or turn them in).

1. Many machine learning applications do not require the precision or range of single-precision (32b) floating point. Thus Google has introduced a new 16-bit floating-point format, `bfloat16`, which is structured in a similar way to the IEEE 754 formats we discussed in class. `bfloat16` has one sign bit, an 8-bit exponent, and a 7-bit significand (the bits after the 1.).

   (a) (5 points) **What is the smallest normal positive number that can be represented as a `bfloat16`?** ("Normal" here means "ignore subnormal numbers".) Express your answer as a normalized number $\pm a.b \times 2^c$, where you specify the sign, a, b, and c. You should clearly show your work if you want partial credit.

   > **Solution:** A positive number has sign bit 0.
   > The smallest normalized exponent is 1. We need to determine the bias. Since the exponent has 8 bits, exactly the same as single-precision floating point, it has the same encoding and bias (127).
   > The smallest significand is 0 (so the fraction is 1.0).
   > Thus the smallest normal positive number is $1.0 \times 2^{1-127} = 1.0 \times 2^{-126}$.

   (b) NVIDIA hardware also supports a 16-bit floating-point format, `half`. It has 1 sign bit, 5 exponent bits, and 11 significand bits.

   Answer each of the following questions with `bfloat16` or `half`.

   i. (1 point) The format with the largest representable value is (`bfloat1` or `half`):

   > **Solution:** `bfloat16`, which has more exponent bits.

   ii. (1 point) The format with a representable value closest to zero is (`bfloat1` or `half`):

Solution: `bfloat16`, which has more exponent bits. This is true whether subnormals are or are not allowed.

iii. (1 point) The format with the smallest range between two adjacent values when the bias-adjusted exponent is zero (meaning the normalized number is $1.x \times 2^0$) (where, for these two adjacent values, the floating-point representation is identical except for the least significant bit of the significand) is (`bfloat1` or `half`):

Solution: `half`, which has more significand bits and thus more precision.

2. Short code sequences.

(a) (6 points) MIPS has the instruction `nor`. RISC-V lacks this instruction. **Implement `nor x2, x3, x4` in as few RISC-V instructions as possible.** You may not make any changes to any register except `x2`.

Solution: `nor` is just `or` followed by `not`, but there is no `not`; but `xori` with all 1s as the immediate implements `not`.

```
or x2, x3, x4
xori x2, x2, 0xfff # alternatively, xori x2, x2, -1
```

Grading: Correct but not minimal: 4 points; no more than 2 points if incorrect.

(b) (6 points) RISC-V also lacks the instruction `sgtz`, which sets its output register to 1 if the input register is larger than zero and to 0 otherwise. **Implement `sgtz x2, x3` in as few RISC-V instructions as possible.** You may not make any changes to any register except `x2`.

Solution: This is a RISC-V pseudoinstruction and is implemented as `slt x2, x0, x3`.

Grading: Correct but not minimal: 4 points; no more than 2 points if incorrect.

(c) (12 points) RISC-V has a floating-point instruction called `fsgnj.d rd, rs1, rs2`, "floating-point sign injection". It returns a number whose magnitude is the same as its `rs1` argument but whose sign is the same as its `rs2` argument. Note it works on double-precision (64b) data.

Implement `fsgnj.d x2, x3, x4` in as few *integer* RISC-V instructions as possible. *Your inputs are 64b floating-point numbers stored in 64b integer registers and your output is also a 64b floating-point number in a 64b integer register. Do* ***not*** *use floating-point registers or instructions to solve this problem.* You may use `x10`, `x11`, etc. as temporary registers if needed. For this part only, you may use the pseudoinstruction `li` (load immediate), which will count as one instruction.

To aid grading (to help give you partial credit), describing your approach in a sentence or two will be helpful.

Solution: This question is essentially "set bit 63 (the sign bit) of `x3` to `x4`'s sign bit".

```
li x10, 0x8000000000000000 # 1 in the sign bit, otherwise 0
xori x11, x10, 0xfff # x11 now has 0x7fffffffffffffff
and x10, x4, x10 # x10 is x4's sign bit
and x11, x3, x11 # x11 is x3's exponent/significand
or x2, x10, x11 # x2 combines x4's sign bit and x3's e/s
```

3. (8 points) Let's consider adding a reciprocal operation to RISC-V. Your engineers have determined that the primary use for a reciprocal operation is to normalize 3-component vectors:

$$x_{\text{new}} = \frac{x}{\sqrt{x^2 + y^2 + z^2}},$$

with y_{new} and z_{new} defined analogously. Note we could take the reciprocal of the square root once and then perform multiply instructions to achieve the same results. To compute x_{new}, y_{new}, and z_{new}, we could replace divide operations with reciprocal and multiply instructions.

If we assume that add instructions take 1 cycle, multiply operations take 4 cycles, and both square-root and divide operations take 20 cycles, **what is the maximum number of cycles for the reciprocal operation so that 3-component vector normalization is at least as fast using reciprocal and multiply as it was using division?**

Solution: First, both implementations must compute the square root, so we can leave that out of the analysis.

Using divide, our instruction sequence is three divisions, one for each of x_{new}, y_{new}, and z_{new}. This would take 60 cycles.

Using reciprocal and multiply, our instruction sequence is one reciprocal and three multiplies. If our three multiplies take 4 cycles each, that allows the reciprocal to be at most **48 cycles** to deliver equivalent performance to the divide solution.

4. You have simplified the design of your new RISC-Y processor to the point where you only have two classes of instructions, A-type (which takes 1 cycle to finish) and B-type (which takes 3). You compile your company's most important benchmark on your internal compiler and find it has a CPI of 2.8.

   The Team Rocket compiler company visits your company and boasts they can achieve twice the MIPS (millions of instructions per second) on your benchmark on your RISC-Y

processor. You test their compiler: it's true, they can. However, the runtime of their compiled code is the same as yours.

(a) (7 points) What is the instruction mix (what percentage of A-type and what percentage of B-type) of the compiled code from the Team Rocket compiler?

Solution:

$$\text{MIPS} = \frac{\text{instructions}}{\text{s}} = \frac{\text{instructions}}{\text{cycle}} \cdot \frac{\text{cycles}}{\text{s}}$$

Since the same hardware is used for both your compiler and Team Rocket's compiler, if they get twice as many MIPS, it means their CPI is twice as good (1.4, vs. the original 2.8). Thus their instruction mix must be **0.8 A-type and 0.2 B-type** ($0.8 \cdot 1 + 0.2 \cdot 3 = 1.4$).

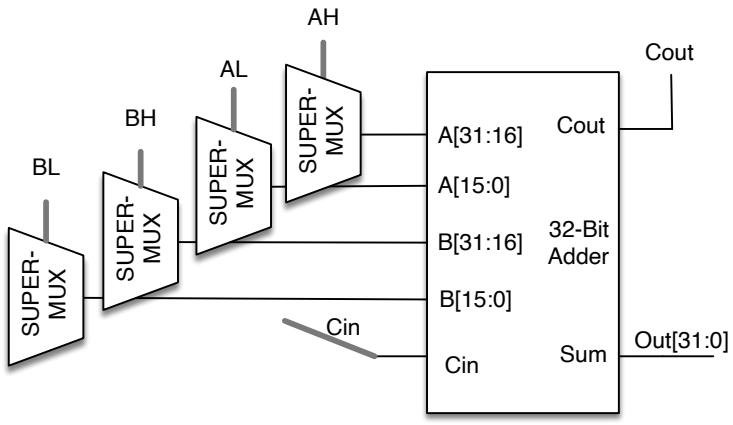
(b) (3 points) If Team Rocket's code gets twice as many MIPS, **why does it run at the same speed as your internal compiler's?** Be precise as to why and quantify your answer.

Solution:

$$\text{runtime} = \frac{\text{s}}{\text{program}} = \frac{\text{s}}{\text{cycle}} \cdot \frac{\text{cycles}}{\text{instruction}} \cdot \frac{\text{instructions}}{\text{program}}$$

Runtime ($\frac{\text{s}}{\text{program}}$) is the same for both compilers's code.
$\frac{\text{s}}{\text{cycle}}$ is the same because it's the same hardware.
$\frac{\text{cycles}}{\text{instruction}}$ is twice as good for the Team Rocket code.
Thus Team Rocket's code must have **twice as many instructions** in the program ($\frac{\text{instructions}}{\text{program}}$) as your internal compiler's.

5. In this problem we design an integer ALU that is inspired by the POWER instruction set. For the purposes of this problem, assume that POWER supports 32-bit integers, that it has an R-type instruction format that takes two input registers and one output register, and an I-type instruction format that takes one input register, one output register, and a 16-bit immediate.



Our base hardware is a 32-bit adder that has five inputs. Four of the inputs (`AH`, `AL`, `BH`, and `BL`) specify the A and B inputs to the adder, each divided into two 16-bit chunks (the 16b high and low halves of the 32b input words to the adder). The fifth input (`Cin`) is a one-bit carry in.

Your job is to select what goes into these five inputs. For the first four inputs, you must set the control signals on a SUPERMUX, which is a 16-bit wide mux that can select from 14 16-bit inputs:

- `A[31:16]`
- `A[15:0]`
- `B[31:16]`
- `B[15:0]`
- `~A[31:16]` (invert the bits of `A[31:16]`)
- `~A[15:0]`
- `~B[31:16]`
- `~B[15:0]`
- `imm[15:0]` (the immediate from I-type instructions)
- `A[31] x 16` (bit `A[31]` copied to all 16 bits)
- `B[31] x 16` (bit `B[31]` copied to all 16 bits)
- `imm[15] x 16` (bit `imm[15]` copied to all 16 bits)
- `0 x 16` (16 zeroes)
- `1 x 16` (16 ones)

For the last control input (`Cin`), you can set it to `0` or `1`.

As an example, if I asked you to configure this ALU to compute a standard 32-bit add, you would set the following:

- `AH` = `A[31:16]`

- `AL` = `A[15:0]`
- `BH` = `B[31:16]`
- `BL` = `B[15:0]`
- `Cin` = `0`

For each of the following instructions, fill in how you would set the five control signals.

(a) (3 points) `SUB`: a 32-bit subtract, `A-B`.

Solution:

- `AH` = `A[31:16]`
- `AL` = `A[15:0]`
- `BH` = `~B[31:16]`
- `BL` = `~B[15:0]`
- `Cin` = `1`

Grading for this and the next three: All correct: 3 points; 3–4 correct: 2 points; 1–2 correct: 1 point.

(b) (3 points) `LUI`: load the 16-bit immediate into the upper 16 bits of the output, setting the other output bits to zero

Solution:

- `AH` = `imm[15:0]`
- `AL` = `0 x 16`
- `BH` = `0 x 16`
- `BL` = `0 x 16`
- `Cin` = `0`

(c) (3 points) `ADDIS`: "add immediate shifted"; add input `A` to the (immediate left-shifted by 16 bits)

Solution:

- `AH` = `A[31:16]`
- `AL` = `A[15:0]`

- BH = imm[15:0]
- BL = 0 x 16
- Cin = 0

(d) (3 points) DBL: "multiply by 2"; return A×2

Solution:

- AH = A[31:16]
- AL = A[15:0]
- BH = A[31:16]
- BL = A[15:0]
- Cin = 0

(e) (3 points) In English, what does the instruction with the following control settings do?

- AH = 1 x 16
- AL = 1 x 16
- BH = B[15:0]
- BL = B[31:16]
- Cin = 1

Solution: Swap B's high halfword (upper 16 bits) with B's low halfword (lower 16 bits). (The result is B[15:0] :: B[31:0].) Note that the settings for A correspond to -1 and the carry in is +1, so these all cancel during addition.

UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Fall 2019

Midterm 2

Name: _____

Instructions:

1. This exam is open-note, open-book. Calculators are allowed.
2. No devices that can communicate (laptops, phones, or PDAs) allowed, with the exception of a device that can read your e-book textbook, on which you must turn off your wifi and/or cellular connections. Other than "find", no typing. Turn off all phones, pagers, and alarms.
3. *You may not share any materials with anyone else, including books, notes, lab reports, datasheets, calculators, and most importantly, answers!*
4. This exam is designed to be a 60 minute exam, but you have 110 minutes to complete it. (Roughly) one point is budgeted per minute. Use your time wisely!

Excerpts from the UC Davis Code of Academic Conduct:

1. Each student should act with personal honesty at all times.
2. Each student should act with fairness to others in the class. That means, for example, that when taking an examination, students should not seek an unfair advantage over classmates through cheating or other dishonest behavior.
3. Students should take group as well as individual responsibility for honorable behavior.

I understand the honor code and agree to be bound by it.

Signature: _____

Page:	2	3	5	6	7	8	9	11	Total
Points:	5	5	9	4	10	5	12	12	62
Score:									

All instructions in this exam, unless otherwise noted, are RISC-V instructions. Recall that if the instruction writes to a register, that register appears first in the instruction. Unless otherwise indicated, you may not use any RISC-V pseudoinstructions on this exam, only actual RISC-V instructions. Feel free to detach the two RISC-V reference pages and the datapath at the end of the exam, but if you do, please take them with you and recycle them (don't try to reattach them or turn them in).

1. In this problem we look at the operation "absolute value of difference" (`ABSDIFF`), which takes two signed integer arguments and returns the signed absolute value of the difference between them (note the sign of the result better not be negative). In C, this looks like:

```
int abs(int x, int y) {
    int z = x - y;
    if (z >= 0) return z;
    return -z;
}
```

A straightforward implementation of the above code requires control-flow instructions (branches and/or jumps). In this problem we consider non-RISC-V instructions that might allow us to implement this without control-flow instructions. In this problem ignore the fact that a twos-complement representation is unbalanced (`-MININT` cannot be represented).

(a) (5 points) Using a minimal sequence of RISC-V instructions, **implement `ABSDIFF x1, x2, x3` without control-flow instructions**. `ABSDIFF x1, x2, x3` means `x1 = ABS(x2 - x3)`. You may use any RISC-V registers, but do not overwrite `x2` or `x3`. In this part, **you must use one or more conditional move instructions** (which, as we discussed in class, are part of many instruction sets but not RISC-V), and specifically any or all of three different conditional move instructions:

- `CMOVGTZ x1, x2, x3 # if (x3 > 0) x1 = x2`
- `CMOVLTZ x1, x2, x3 # if (x3 < 0) x1 = x2`
- `CMOVEQZ x1, x2, x3 # if (x3 == 0) x1 = x2`

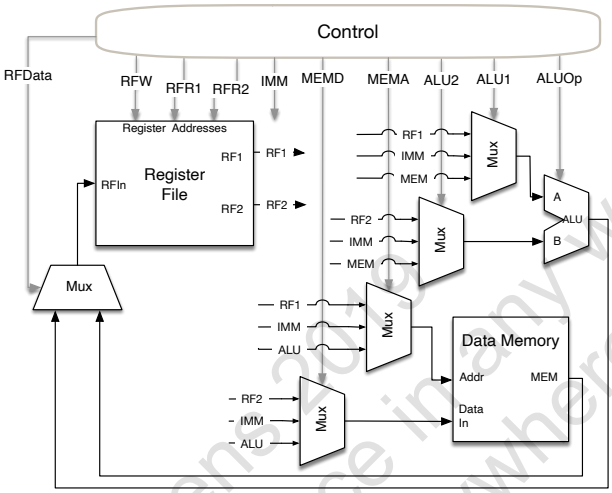
(b) (5 points) Using a minimal sequence of RISC-V instructions, **implement `ABSDIFF x1, x2, x3` without control-flow instructions**. You may use any RISC-V registers, but do not overwrite `x2` or `x3`. In this part, **you must use either or both of the following saturating-subtract instructions**:

- `SUBSAT x1, x2, x3` (subtraction of `x2-x3`, treating both operands as signed numbers and producing a signed result, but rather than overflowing, saturating to the largest/smallest signed value)
- `SUBUSAT x1, x2, x3` (subtraction of `x2-x3`, treating both operands as unsigned numbers and producing an unsigned result, but rather than overflowing, saturating to the largest/smallest unsigned value)

Recall that the actual computation of subtraction of signed and unsigned numbers generates exactly the same bit result; it's only the saturation operation that differs between these two operations. This problem is fairly tricky.

2. In this problem we will look at the single-cycle datapath of the "Bitner" processor, only focusing on the compute part of the Bitner pipeline. In this datapath, the ALU can perform one arithmetic operation every clock cycle and the data memory can perform one memory operation every clock cycle. Note the output of the ALU is an input to the data memory and the output of the data memory is an input to the ALU. Assume Bitner's instruction format contains an opcode, three registers, and one immediate.

   At the end of this exam is a larger, detachable copy of this datapath.



Control points:

- `RFR1`: register number of the first register to be read from the register file
- `RFR2`: register number of the second register to be read from the register file
- `IMM`: the immediate specified by the instruction
- `ALU1`: controls the source of data into the ALU's input 1 (choices: `RF1`, `IMM` `MEM`)
- `ALU2`: controls the source of data into the ALU's input 2 (choices: `RF2`, `IMM`, `MEM`)
- `MEMA`: controls the source of data into the memory's address input (choices: `RF1`, `IMM`, `ALU`)
- `MEMD`: controls the source of data into the memory's data input (choices: `RF2`, `IMM`, `ALU`)
- `ALUOP`: controls the operation performed by the ALU (choices: `ADD`)
- `RFW`: register number of the register to be written to the register file
- `RFData`: controls the source of data to be written into register `RFW` in the register file (choices: `ALU` or `MEM`)

In this problem we ask you to "set the control points" for this datapath. As an example, the instruction `ADD x1, x2, x3` would have the following settings: `RFR1 = 2; RFR2 = 3; IMM = X; ALU1 = RF1; ALU2 = RF2; MEMA = X; MEMD = X; ALUOP = ADD; RFW = 1; RFData = ALU`, where `X` is a don't-care.

(a) (5 points) Which of these signals should come directly from **bits in the 32-bit instruction** (a bit or bits in the instruction are routed as control signals directly into the datapath without any intermediate logic) and which should be generated by **additional logic derived from the 32-bit instruction**? Answer either "bits" or "logic".

- _____ RFR1
- _____ RFR2
- _____ IMM
- _____ ALU1
- _____ ALU2
- _____ MEMA
- _____ MEMD
- _____ ALUOP
- _____ RFW
- _____ RFData

(b) (4 points) What is the primary reason why this particular datapath would be a challenge to pipeline with good performance?

(c) Consider the following C program. This program fetches the value in `array` from memory, adds 42 to it, stores the new value back to memory, and increments `array` to point to the next element in the array.

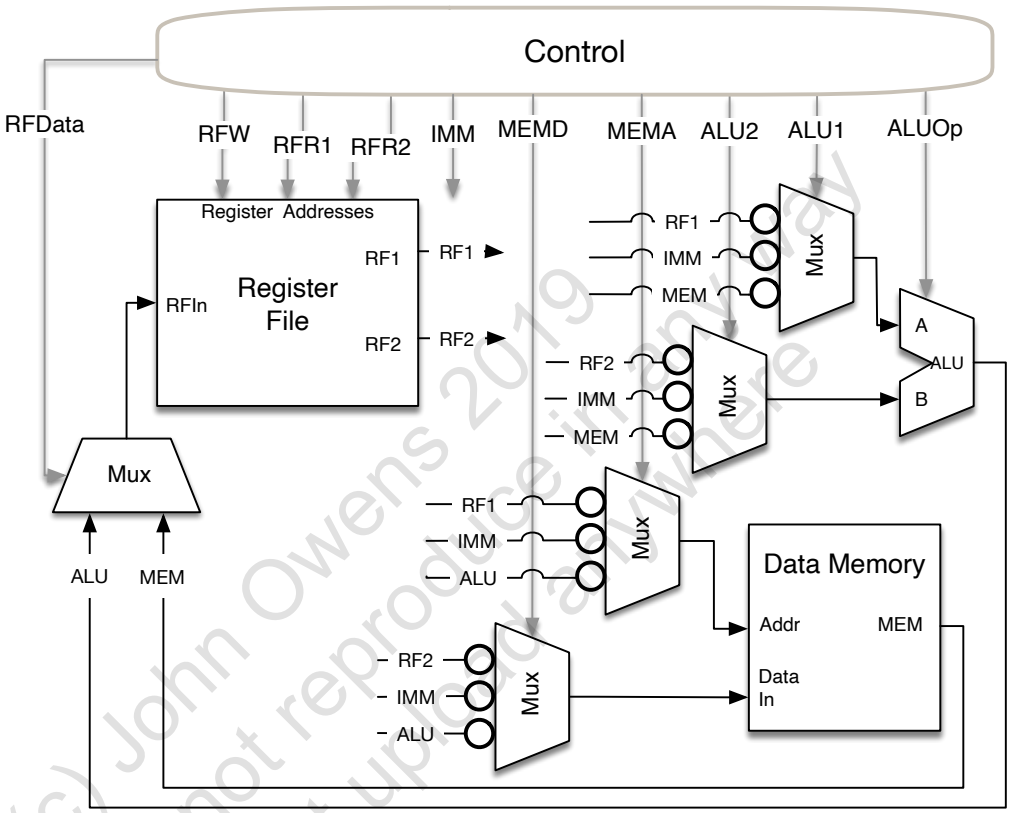
```
int * array;     // assume 64-bit integers
while (1) {
  *array++ += 42;
}
```

Assume that `array` is stored in `x2`.

i. (4 points) **Write the *body* of this loop** (corresponding to the single line of C code between the braces) **as a minimal sequence of instructions in RISC-V assembly.** You can use `x3` as a temporary. Your solution should have 4 RISC-V instructions. Do not write any instructions that relate to the loop itself (you should have no branch/jump instructions).

ii. (10 points) **Write the *body* of this loop** (corresponding to the single line of C code between the braces) **as a minimal sequence of instructions on Bitner**. Express each instruction as a setting of control points; by this we mean to specify how you would set each of the 10 "control points" listed above. If a control point is a don't-care, mark it as `X`. You can use `x3` as a temporary. Do not write any instructions that relate to the loop itself (you should have no branch/jump instructions). Hint: Because this datapath is potentially more capable than the RISC-V datapath we discussed in class, it may be possible to accomplish more in a single instruction than you could in a RISC-V datapath.

(d) (5 points) On the following diagram, note 12 circles on the inputs to the 4 muxes. **Put an X through each circle whose input is not required to run RISC-V instructions. For inputs that are required to run RISC-V instructions, do nothing.** For instance, if the top input of the ALU does not require a RF1 input to run RISC-V instructions, put an X in the topmost circle.



(e) Ben Bitdiddle proposes new instructions for Bitner. Ben hopes each of these instructions can be implemented as single Bitner instructions (not sequences of Bitner instructions). For each of Ben's proposals:

- If it is possible to implement Ben's proposal as a single instruction with the current datapath, set the control points that implement the instruction on this datapath; OR
- If it is *not* possible to implement Ben's proposal as a single instruction with the current datapath, describe the changes needed to the datapath (as English sentences).

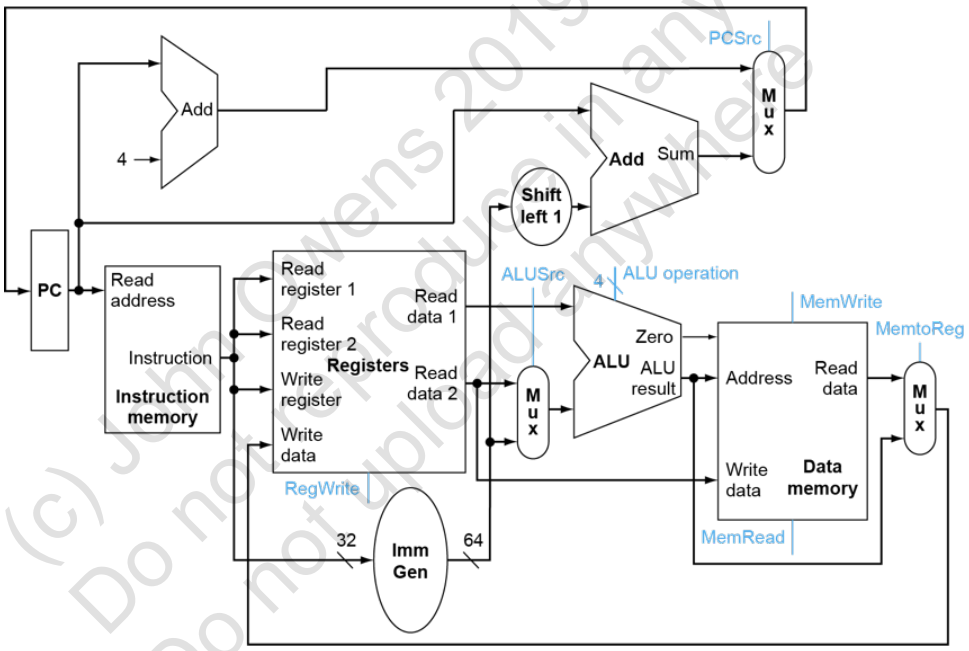
i. (4 points) `LDADDI xdest1, xdest2, xsrc, imm`: Perform two operations in parallel: (1) Register `xdest1` is set to `xsrc + imm`; register `xdest2` is set to the memory contents of the address stored in `xsrc`.

ii. (4 points) `MEMCPY xdest, xsrc`: Copies the memory word stored at the address specified in `xsrc` into the memory location specified by the address in `xdest`.

iii. (4 points) `LD2 xdest, imm(xbase1+xbase2)`: Same as `LD` but instead of the load address as `xbase+imm`, it now supports `xbase1+xbase2+imm`.

3. Alyssa P. Hacker was looking at the class datapath (below) and identified some possible inefficiencies. Specifically, she was looking at the following control signals:

   - `RegWrite`: 1 only if data is written into the register file
   - `ALUSrc`: the second input of the ALU is from the register file (0) or from the immediate (1)
   - `PCSrc`: the next PC comes from PC+4 (0) or from a computed branch/jump target (1)
   - `MemWrite`: 1 only if the write data input to the memory will be written to the address input
   - `MemRead`: 1 only if the data memory is outputting the contents of the memory address specified by its address input
   - `MemtoReg`: the data to write into the register file comes from the ALU (0) or from the data memory (1)



She proposed the following design simplifications. For each proposed design change, **indicate whether her proposal could work properly *OR*, if it will not, an instruction that will not work correctly and why**. For the purposes of this question, only consider the following instructions (the ones we covered in Lecture 2): `ADD`, `ADDI`, `AND`, `ANDI`, `BEQ`, `BGE`, `BLT`, `BNE`, `JAL`, `JALR`, `LD`, `LUI`, `OR`, `ORI`, `SD`, `SLLI`, `SRLI`, `SUB`, `XOR`, `XORI`.

(a) (4 points) Combine `MemWrite` and `MemRead` into one signal `MemOp`, where `MemOp` is set to 1 only on store instructions, and `MemWrite <- MemOp` and `MemRead <- not(MemOp)`. Assume that the memory system can read from any arbitrary memory address without any side effects (no exceptions/page faults/etc.) and that there are no performance consequences if we do this.

(b) (4 points) Remove `RegWrite`; instead set `RegWrite` if `PCSrc` is 0.

(c) (4 points) Remove `ALUSrc`; instead set `ALUSrc` if either `MemWrite` or `MemRead` are 1.

RISC-V Reference

James Zhu <jameszhu@berkeley.edu>

RISC-V Instruction Set

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
add	ADD	R	0000011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0000011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0000011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0000011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0000011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0000011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0000011	0x2	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0000011	0x3	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2		rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3		rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0	0x00	rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x0	0x00	rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x0	0x00	rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x0	0x00	rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	0x00	rd = rs1 << imm	
srli	Shift Right Logical Imm	I	0010011	0x1	0x00	rd = rs1 >> imm	
srai	Shift Right Arith Imm	I	0010011	0x3	0x20	rd = rs1 >> imm	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch $\leq$	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch $\geq$ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	0x00	Transfer control to OS	imm: 0x000
ebreak	Environment Break	I	1110011	0x0	0x00	Transfer control to debugger	imm: 0x001

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	F3	F7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

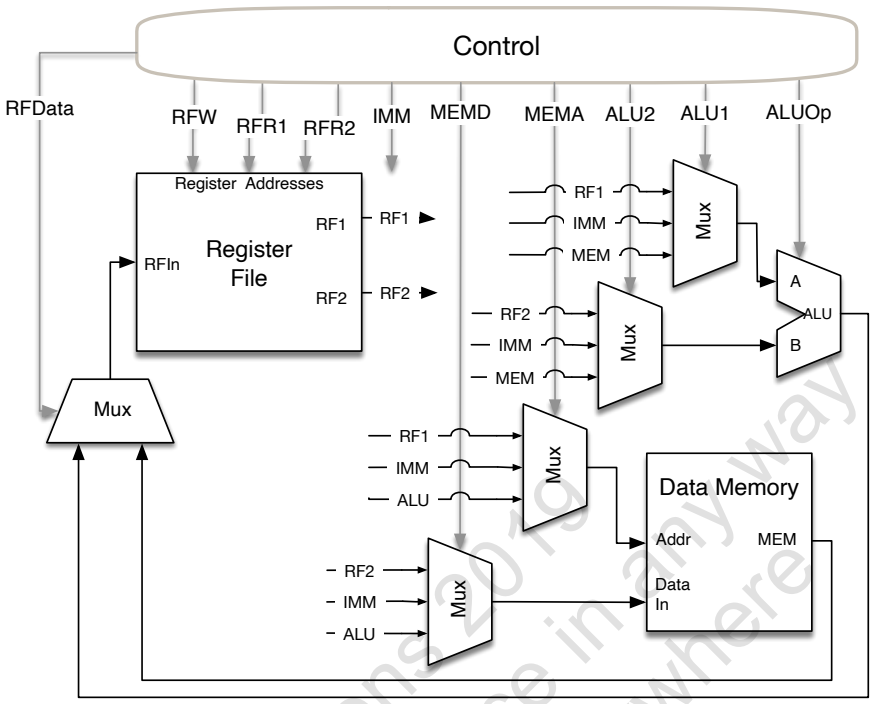
31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7

Inst	Name	FMT	Opcode	F3	F5	Description (C)
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] | rs2; M[rs1] = rd
amoxor.w	Atomix XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	F3	F5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = *((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

This is identical to the datapath and control points presented earlier in the exam. Feel free to detach this for reference and don't turn it in.



Control points:

- `RFR1`: register number of the first register to be read from the register file
- `RFR2`: register number of the second register to be read from the register file
- `IMM`: the immediate specified by the instruction
- `ALU1`: controls the source of data into the ALU's input 1 (choices: `RF1`, `IMM` `MEM`)
- `ALU2`: controls the source of data into the ALU's input 2 (choices: `RF2`, `IMM`, `MEM`)
- `MEMA`: controls the source of data into the memory's address input (choices: `RF1`, `IMM`, `ALU`)
- `MEMD`: controls the source of data into the memory's data input (choices: `RF2`, `IMM`, `ALU`)
- `ALUOP`: controls the operation performed by the ALU (choices: `ADD`)
- `RFW`: register number of the register to be written to the register file
- `RFData`: controls the source of data to be written into register `RFW` in the register file (choices: `ALU` or `MEM`)

UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Fall 2019

Midterm 2 Solutions

All instructions in this exam, unless otherwise noted, are RISC-V instructions. Recall that if the instruction writes to a register, that register appears first in the instruction. Unless otherwise indicated, you may not use any RISC-V pseudoinstructions on this exam, only actual RISC-V instructions. Feel free to detach the two RISC-V reference pages and the datapath at the end of the exam, but if you do, please take them with you and recycle them (don't try to reattach them or turn them in).

1. In this problem we look at the operation "absolute value of difference" (`ABSDIFF`), which takes two signed integer arguments and returns the signed absolute value of the difference between them (note the sign of the result better not be negative). In C, this looks like:

```
int abs(int x, int y) {
    int z = x - y;
    if (z >= 0) return z;
    return -z;
}
```

A straightforward implementation of the above code requires control-flow instructions (branches and/or jumps). In this problem we consider non-RISC-V instructions that might allow us to implement this without control-flow instructions. In this problem ignore the fact that a twos-complement representation is unbalanced (`-MININT` cannot be represented).

(a) (5 points) Using a minimal sequence of RISC-V instructions, **implement `ABSDIFF x1, x2, x3` without control-flow instructions**. `ABSDIFF x1, x2, x3` means `x1 = ABS(x2 - x3)`. You may use any RISC-V registers, but do not overwrite `x2` or `x3`. In this part, **you must use one or more conditional move instructions** (which, as we discussed in class, are part of many instruction sets but not RISC-V), and specifically any or all of three different conditional move instructions:

- `CMOVGTZ x1, x2, x3 # if (x3 > 0) x1 = x2`
- `CMOVLTZ x1, x2, x3 # if (x3 < 0) x1 = x2`
- `CMOVEQZ x1, x2, x3 # if (x3 == 0) x1 = x2`

Solution:

```
SUB x4, x1, x2
CMOVGTZ x1, x4, x4 # x1 now has the correct answer
                   # if the diff was positive
SUB x4, x0, x4     # flip the sign of x4
CMOVGTZ x1, x4, x4 # x1 now also has the correct answer
                   # if the diff was negative
CMOVEQZ x1, x4, x4 # better cover the 0 case too
```

(b) (5 points) Using a minimal sequence of RISC-V instructions, **implement `ABSDIFF x1, x2, x3` without control-flow instructions**. You may use any RISC-V registers, but do not overwrite `x2` or `x3`. In this part, **you must use either or both of the following saturating-subtract instructions**:

- `SUBSAT x1, x2, x3` (subtraction of `x2-x3`, treating both operands as signed numbers and producing a signed result, but rather than overflowing, saturating to the largest/smallest signed value)
- `SUBUSAT x1, x2, x3` (subtraction of `x2-x3`, treating both operands as unsigned numbers and producing an unsigned result, but rather than overflowing, saturating to the largest/smallest unsigned value)

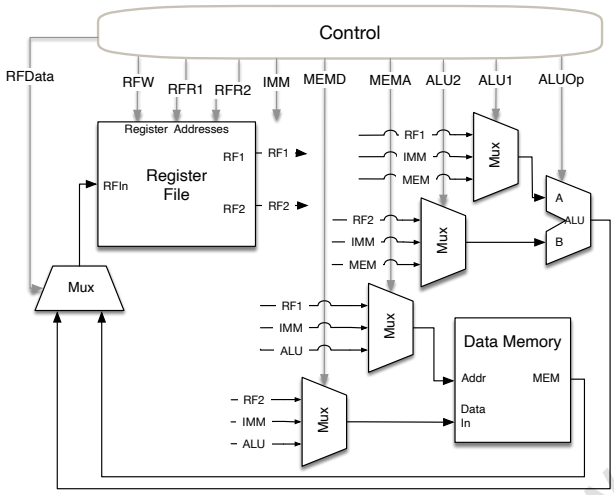
Recall that the actual computation of subtraction of signed and unsigned numbers generates exactly the same bit result; it's only the saturation operation that differs between these two operations. This problem is fairly tricky.

Solution: The key realization here is that unsigned saturated subtract of $a - b$ is equivalent to $\max(a - b, 0)$.

```
SUBUSAT x1, x2, x3  \# x1 = max(x2-x3,0)
SUBUSAT x4, x3, x2  \# x4 = max(x3-x2,0); either x1 or x4 is now 0
OR      x1, x1, x4  \# choose whichever of x1 or x4 is not 0
```


2. In this problem we will look at the single-cycle datapath of the "Bitner" processor, only focusing on the compute part of the Bitner pipeline. In this datapath, the ALU can perform one arithmetic operation every clock cycle and the data memory can perform one memory operation every clock cycle. Note the output of the ALU is an input to the data memory and the output of the data memory is an input to the ALU. Assume Bitner's instruction format contains an opcode, three registers, and one immediate.

   At the end of this exam is a larger, detachable copy of this datapath.



Control points:

- `RFR1`: register number of the first register to be read from the register file
- `RFR2`: register number of the second register to be read from the register file
- `IMM`: the immediate specified by the instruction
- `ALU1`: controls the source of data into the ALU's input 1 (choices: `RF1`, `IMM` `MEM`)
- `ALU2`: controls the source of data into the ALU's input 2 (choices: `RF2`, `IMM`, `MEM`)
- `MEMA`: controls the source of data into the memory's address input (choices: `RF1`, `IMM`, `ALU`)
- `MEMD`: controls the source of data into the memory's data input (choices: `RF2`, `IMM`, `ALU`)
- `ALUOP`: controls the operation performed by the ALU (choices: `ADD`)
- `RFW`: register number of the register to be written to the register file
- `RFData`: controls the source of data to be written into register `RFW` in the register file (choices: `ALU` or `MEM`)

In this problem we ask you to "set the control points" for this datapath. As an example, the instruction `ADD x1, x2, x3` would have the following settings: `RFR1 = 2; RFR2 = 3; IMM = X; ALU1 = RF1; ALU2 = RF2; MEMA = X; MEMD = X; ALUOP = ADD; RFW = 1; RFData = ALU`, where `X` is a don't-care.

(a) (5 points) Which of these signals should come directly from **bits in the 32-bit instruction** (a bit or bits in the instruction are routed as control signals directly into the datapath without any intermediate logic) and which should be generated by **additional logic derived from the 32-bit instruction**? Answer either "bits" or "logic".

- ____**bits**____ `RFR1`

- ____bits____ RFR2
- ____bits____ IMM
- ____logic____ ALU1
- ____logic____ ALU2
- ____logic____ MEMA
- ____logic____ MEMD
- ____logic____ ALUOP
- ____bits____ RFW
- ____logic____ RFData

(b) (4 points) What is the primary reason why this particular datapath would be a challenge to pipeline with good performance?

Solution: Because the data memory output is an input to the ALU and the ALU output is an input to the memory, we'd have two choices:

- We could put them both into the same pipeline stage in which case that stage would be very long and yield poor performance
- We could put them into separate pipeline stages but then would have to choose which came first, and we would have a structural hazard every time they were used in the opposite order

(c) Consider the following C program. This program fetches the value in `array` from memory, adds 42 to it, stores the new value back to memory, and increments `array` to point to the next element in the array.

```
int * array;     // assume 64-bit integers
while (1) {
  *array++ += 42;
}
```

Assume that `array` is stored in `x2`.

i. (4 points) **Write the *body* of this loop** (corresponding to the single line of C code between the braces) **as a minimal sequence of instructions in RISC-V assembly.** You can use `x3` as a temporary. Your solution should have 4 RISC-V instructions. Do not write any instructions that relate to the loop itself (you should have no branch/jump instructions).

Solution:

```
ld x3, 0(x2)
addi x3, x3, 42
sd x3, 0(x2)
addi x2, x2, 8
```

ii. (10 points) **Write the *body* of this loop** (corresponding to the single line of C code between the braces) **as a minimal sequence of instructions on Bitner**. Express each instruction as a setting of control points; by this we mean to specify how you would set each of the 10 "control points" listed above. If a control point is a don't-care, mark it as `X`. You can use `x3` as a temporary. Do not write any instructions that relate to the loop itself (you should have no branch/jump instructions). Hint: Because this datapath is potentially more capable than the RISC-V datapath we discussed in class, it may be possible to accomplish more in a single instruction than you could in a RISC-V datapath.

Solution: We can collapse the four RISC-V instructions above into two Bitner instructions:

- Combine the load and the first add-immediate (+42) into one instruction; the output of the load is the input to an add, and the output of the add goes back into the register file.

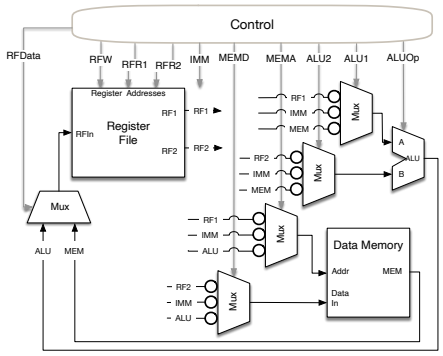
```
RFR1 = 2
RFR2 = X
IMM = 42
RFW = 3
RFData = ALU
ALU1 = MEM
ALU2 = IMM (42)
MEMA = RF1
MEMD = X
ALUOP = ADD
```

- Then combine the store and the second add-immediate (+8) into one instruction that run in parallel. Only the add-immediate writes back.

```
RFR1 = 2
RFR2 = 3
IMM = 8
RFW = 2
RFData = ALU
ALU1 = RF1
ALU2 = IMM (8)
MEMA = RF1
MEMD = RF2
ALUOP = ADD
```


(d) (5 points) On the following diagram, note 12 circles on the inputs to the 4 muxes. **Put an X through each circle whose input is not required to run RISC-V**

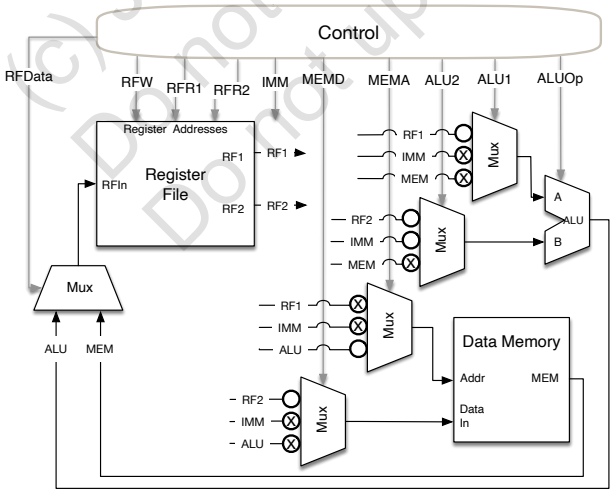
instructions. For inputs that are required to run RISC-V instructions, do nothing. For instance, if the top input of the ALU does not require a RF1 input to run RISC-V instructions, put an X in the topmost circle.



Solution: To run only RISC-V instructions, we do **not** need the following connections:

- ALU A: `IMM`, `MEM`
- ALU B: `MEM`
- Memory address: `RF1`, `IMM`
- Memory data: `IMM`, `ALU`

It's arguable that ALU A and ALU B could switch their answers, although that may lead to complications with where I-type instructions get their RF input.



(e) Ben Bitdiddle proposes new instructions for Bitner. Ben hopes each of these instructions can be implemented as single Bitner instructions (not sequences of Bitner instructions). For each of Ben's proposals:

- If it is possible to implement Ben's proposal as a single instruction with the current datapath, set the control points that implement the instruction on this datapath; OR
- If it is *not* possible to implement Ben's proposal as a single instruction with the current datapath, describe the changes needed to the datapath (as English sentences).

i. (4 points) `LDADDI xdest1, xdest2, xsrc, imm`: Perform two operations in parallel: (1) Register `xdest1` is set to `xsrc + imm`; register `xdest2` is set to the memory contents of the address stored in `xsrc`.

> **Solution:** The current datapath does not support this; it requires writing back to two different registers. We would have to add a second write port to the register file and make sure the outputs of the ALU and the memory could both be routed into the two write ports.

ii. (4 points) `MEMCPY xdest, xsrc`: Copies the memory word stored at the address specified in `xsrc` into the memory location specified by the address in `xdest`.

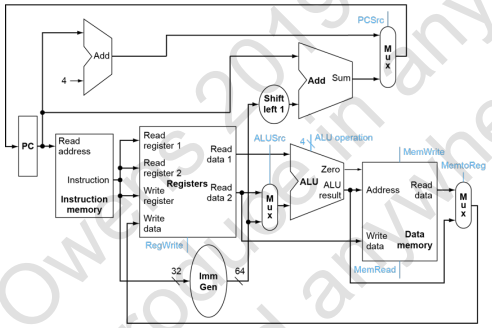
> **Solution:** The current datapath does not support this; it requires two memory operations and thus two data memories. There are multiple places where a second memory could be placed, including:
>
> - In parallel with the current data memory (rather than having one ALU and one memory in parallel, we'd have one ALU and two memories in parallel). The output of the first memory (load) must be able to be routed into the data input of the second memory (store).
> - In series after the current data memory (the output of the first memory can be routed into the data input of the second memory).

iii. (4 points) `LD2 xdest, imm(xbase1+xbase2)`: Same as `LD` but instead of the load address as `xbase+imm`, it now supports `xbase1+xbase2+imm`.

> **Solution:** The current datapath does not support this; it does not allow two additions. One way to address this is to add an adder to the address input of the data memory that allows a second addition. One input to that adder should be the output of the ALU; the second input could be either read register or the immediate (the datapath allows any of those three to work).

3. Alyssa P. Hacker was looking at the class datapath (below) and identified some possible inefficiencies. Specifically, she was looking at the following control signals:

   - `RegWrite`: 1 only if data is written into the register file
   - `ALUSrc`: the second input of the ALU is from the register file (0) or from the immediate (1)
   - `PCSrc`: the next PC comes from PC+4 (0) or from a computed branch/jump target (1)
   - `MemWrite`: 1 only if the write data input to the memory will be written to the address input
   - `MemRead`: 1 only if the data memory is outputting the contents of the memory address specified by its address input
   - `MemtoReg`: the data to write into the register file comes from the ALU (0) or from the data memory (1)



She proposed the following design simplifications. For each proposed design change, **indicate whether her proposal could work properly *OR*, if it will not, an instruction that will not work correctly and why**. For the purposes of this question, only consider the following instructions (the ones we covered in Lecture 2): `ADD`, `ADDI`, `AND`, `ANDI`, `BEQ`, `BGE`, `BLT`, `BNE`, `JAL`, `JALR`, `LD`, `LUI`, `OR`, `ORI`, `SD`, `SLLI`, `SRLI`, `SUB`, `XOR`, `XORI`.

(a) (4 points) Combine `MemWrite` and `MemRead` into one signal `MemOp`, where `MemOp` is set to 1 only on store instructions, and `MemWrite <- MemOp` and `MemRead <- not(MemOp)`. Assume that the memory system can read from any arbitrary memory address without any side effects (no exceptions/page faults/etc.) and that there are no performance consequences if we do this.

> **Solution:** This should work properly. The reasons to have two signals are to allow both of them to be true at a time (and we have no instructions that do both a read and a write) or both of them to be false (which is probably true on non-memory instructions, but the `MemtoReg` mux handles this case). Note, however, that we don't generally want to dispatch memory reads that aren't

from a load instruction; we may access addresses that we're not allowed to access (e.g., segmentation faults), we'd probably pollute the cache, and we'd burden the memory system with requests that we don't need so there would be performance consequences.

(b) (4 points) Remove `RegWrite`; instead set `RegWrite` if `PCSrc` is 0.

Solution: This doesn't work. Alyssa's logic here is that non-branch-or-jump instructions always write back to the register file. This is not true for store instructions (`SD`).

(c) (4 points) Remove `ALUSrc`; instead set `ALUSrc` if either `MemWrite` or `MemRead` are 1.

Solution: This doesn't work. Alyssa's logic here is that we use the immediate input to the ALU only on load and store instructions, but we also use it on ALU instructions that take an immediate (`ADDI`, `ANDI`, `ORI`, etc.).

UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Fall 2019

Final Examination

Name: _____

Instructions:

1. This exam is open-note, open-book. Calculators are allowed.
2. No devices that can communicate (laptops, phones, or PDAs) allowed, with the exception of a device that can read your e-book textbook, on which you must turn off your wifi and/or cellular connections. Other than "find", no typing. Turn off all phones, pagers, and alarms.
3. *You may not share any materials with anyone else, including books, notes, lab reports, datasheets, calculators, and most importantly, answers!*
4. This exam is designed to be a 60 minute exam, but you have 120 minutes to complete it. (Roughly) one point is budgeted per minute. Use your time wisely!

Excerpts from the UC Davis Code of Academic Conduct:

1. Each student should act with personal honesty at all times.
2. Each student should act with fairness to others in the class. That means, for example, that when taking an examination, students should not seek an unfair advantage over classmates through cheating or other dishonest behavior.
3. Students should take group as well as individual responsibility for honorable behavior.

I understand the honor code and agree to be bound by it.

Signature: _____

Page:	2	3	4	5	6	7	8	9	Total
Points:	4	10	10	9	7	3	12	6	61
Score:									

All instructions in this exam, unless otherwise noted, are RISC-V instructions. Recall that if the instruction writes to a register, that register appears first in the instruction. Unless otherwise indicated, you may not use any RISC-V pseudoinstructions on this exam, only actual RISC-V instructions. Feel free to detach the two RISC-V reference pages and the memory system diagram at the end of the exam, but if you do, please take them with you and recycle them (don't try to reattach them or turn them in).

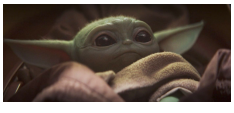
1. We discussed dense matrix-dense matrix multiplication in class. To recap, consider A, B, and C as $n \times n$ matrices; we wish to compute $C = C + A \times B$. This can be computed by

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j) // this is normal (scalar) + and *
```


We begin by computing the *arithmetic intensity* of this code in operations per memory word, assuming no caches, as a function of n. (Recall that the arithmetic intensity is the number of arithmetic operations per word of memory bandwidth.) Assume n is large; you can approximate any answers given this assumption. Assume each addition or multiplication counts as one operation. Assume all of A, B, and C are stored in main memory (DRAM) and each read from/write to main memory counts as one memory word.

The inner line of code runs n^3 times. It has 2 arithmetic operations, 3 loads, and 1 write. Thus the arithmetic intensity is $\frac{2n^3}{(3+1)n^3} = 1/2$.

(a) (4 points) Consider two different implementations of the same task *with the same runtime*. Baby Yoda's has high arithmetic intensity and is limited by arithmetic capability. The Mandalorian's has low arithmetic intensity and is limited by memory bandwidth. Even though both have the same runtime, we prefer Baby Yoda's implementation with higher arithmetic intensity because of how we expect future machines to behave. In terms of technology trends, **why do we prefer implementations with more arithmetic intensity?**

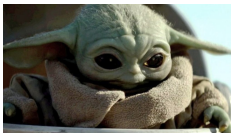


(b) (4 points) Assume we have a very large cache so all memory loads only read a reference the first time they see that reference and (because they are captured in the cache) all memory writes occur only at the end of the program. Reads from/writes to cache are free (do not count in terms of arithmetic intensity). **What is the arithmetic intensity now?**

(c) (6 points) Now consider a more realistic cache that is not large enough to hold the entire input and output. This cache can hold two matrix rows or columns plus one output element. Assume there are no cache conflicts; any two rows/columns plus one output element can fit in the cache. The Mandalorian proposes using the cache in the following way:

```
for i = 1 to n
  // read row i of A into cache
  for j = 1 to n
    // read row j of B into cache
    // read C(i,j) into cache
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j) // this is normal (scalar) + and *
    // after k loop is complete, write C(i,j) back to memory
```

What is the arithmetic intensity now? It is OK to ignore lower-order terms in this analysis.



(d) (10 points) Finally, Baby Yoda proposes to partition the matrix in a different way. Instead of reading 1-dimensional rows or columns from A and B, Baby Yoda arranges those matrices as $b \times b$ subblocks. (For instance, a 128×128 matrix may be structured as a 4×4 grid of 32×32 subblocks.) The cache can hold any two input blocks plus one output block at any time.

Now our matrix multiply looks like this (note our operations are now on blocks at a time instead of rows or columns at a time). The arithmetic, while factored differently, is exactly the same as the previous case.

```
// note i, j, and k are now block indexes, not element indexes
for i = 1 to n/b
  for j = 1 to n/b
    // read block C(i,j) into cache
    for k = 1 to n/b
      // read block A(i,k) into cache
      // read block B(k,j) into cache
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
      // Here + indicates a matrix sum of blocks (block + block)
      // Here * indicates a matrix multiply of blocks (block * block)
    // after k loop is complete, write block C(i,j) back to memory
```

What is the arithmetic intensity now? It is again OK to ignore lower-order terms in this analysis.

2. In the 2-page RISC-V Reference Card at the end of the exam, at the bottom right of the first page, you will see the "16-bit (RVC) Instruction Formats". RVC stands for "RISC-V Compressed".

   (a) (5 points) Note CR-type instructions have a `funct4` entry as part of their opcodes instead of a `funct3` (as in all the other instructions). Assume that the opcode combinations `(inst[1:0] == 01 OR inst[1:0] == 10) AND inst[15:13] == 100` are used to encode CR-type instructions. Given the instruction formats listed in the reference page, **how many different instructions can RVC support?** Note not all possible RVC instructions are specified on the reference card.

   (b) In practice, these 16-bit RVC instructions are translated to 32-bit full ("RVI") RISC-V instructions. This process is summarized in the table on the reference card called "Optional Compressed (16-bit) Instruction Extension: RVC", but we will highlight the important parts here directly. In this part, you may need to consult the table "16-bit (RVC) instruction formats".

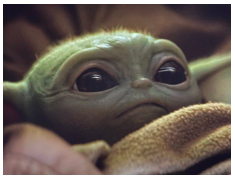
      i. (2 points) The RVC instruction `C.ADD rd, rs1` translates to the RVI instruction `ADD rd, rd, rs1`. Both instructions run a full 64b add. However, the `C.ADD` operation has a significant restriction compared to the full `ADD` instruction. **What is that restriction? Be specific.**

      ii. (2 points) Some RVC instructions, like `C.BEQZ`, have a register argument with a prime (apostrophe) after the register name (`C.BEQZ rs1', imm` → `BEQ rs1', x0, imm`). **What is the restriction indicated by the prime?**

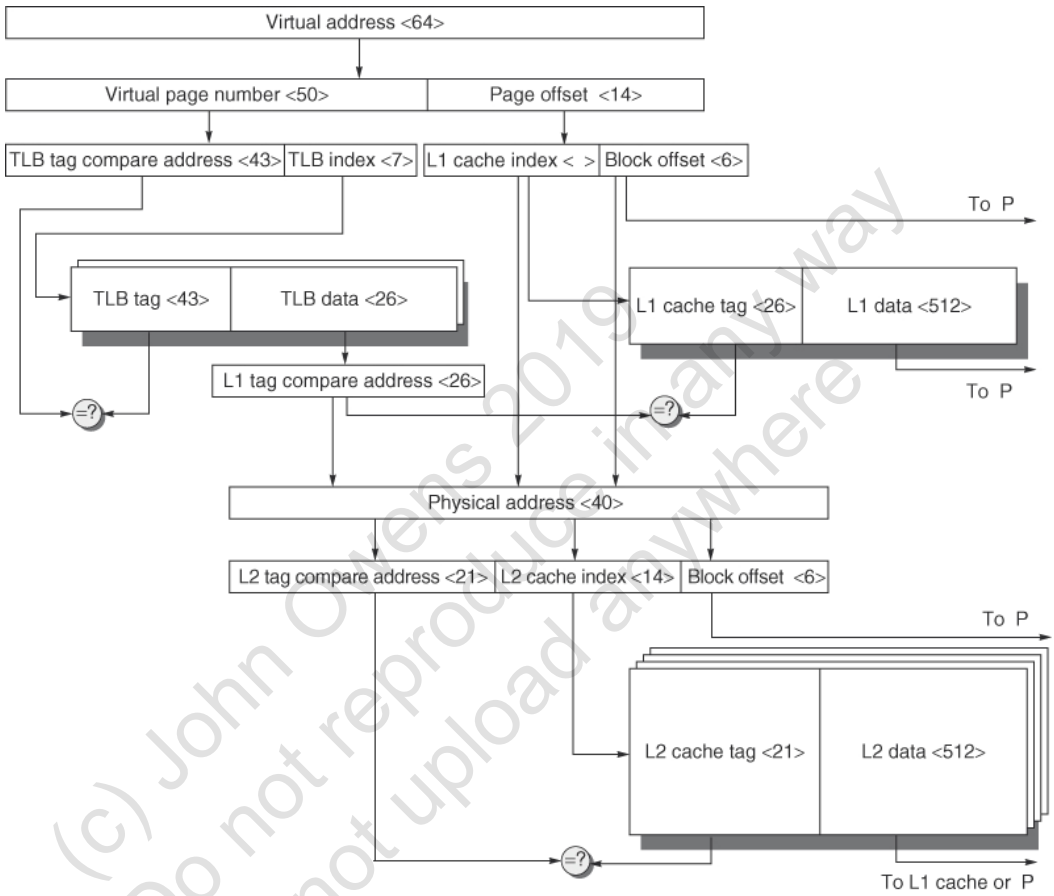
iii. (2 points) All RVC memory-access instructions multiply their immediates by a power-of-two constant (e.g., `C.LW rd', rs1', imm` → `LW rd', rs1', imm*4`). Because of this multiplication, **what can a RVI memory-access instruction do that its partner RVC instruction cannot?** (The answer is not "allows byte addressing"; both RVC and RVI memory-access instructions are byte-addressed.)

iv. (2 points) The instruction `C.ADDI16SP` is primarily intended to manipulate the _____. (Fill in the previous blank with one word.)

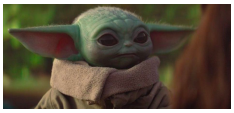
(c) (3 points) The new BabyYoda compiler inputs a RVI (32-bit) RISC-V instruction sequence and transforms some RVI instructions into their RVC (16-bit) equivalents. When running the resulting code, compared to the RVI implementation, **would you expect the instruction cache hit rate would increase, decrease, or stay the same?** Justify your answer.



3. While sipping his soup, Baby Yoda uses the Force to derive the following characteristics of his laptop's memory system, which is byte-addressed and contains a virtual memory subsystem and L1 and L2 caches. A copy of this diagram is attached at the end of the exam, which you can detach. In this diagram, `<n>` means a signal of `<n>` bits wide.



(a) (3 points) What is the page size?

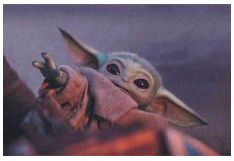


(b) (3 points) The TLB has 256 entries in it. **What is the associativity of the TLB** (e.g., 1-way set associative = direct-mapped, 2-way set associative, 4-way, fully associative, etc.)?

(c) (3 points) How large is the L1 cache (data only)?

(d) (3 points) Assuming the L2 cache stores 2^{22} B = 4 MiB of data, what is its associativity?

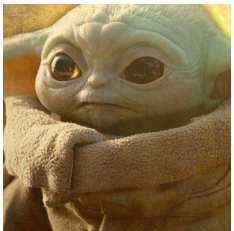
(e) (3 points) What is the motivation for making replacement algorithms for hardware caches less complicated than replacement algorithms for pages?



4. Assume that you have 2 64-bit double-precision floating point numbers stored in (integer) registers `x2` and `x3`. You run the instruction `XOR x1, x2, x3`. In this question "normal number" means "not zero, not infinity, not denormal, not NaN".

   (a) (3 points) If you know that `x2` is a positive normal number and `x3` is a negative normal number, **what bits in `x1` do you know for certain?** (Your answer should be of the form "Bit 42 will be a 1 and bit 24 will be a 0".) Bit 0 is the least significant bit (LSB) and bit 63 is the most significant bit (MSB).

   (b) (3 points) If you know that `x2` is a negative denormal number and `x3` is negative infinity, **what bits in `x1` do you know for certain?** (Your answer should be of the form "Bit 42 will be a 1 and bit 24 will be a 0".)



Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

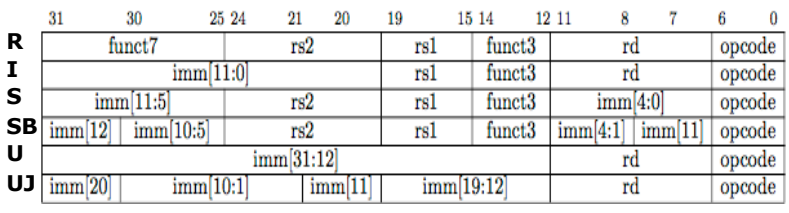
RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

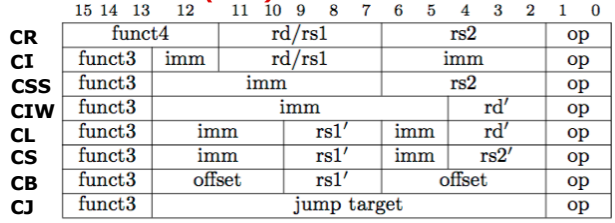
Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats



16-bit (RVC) Instruction Formats



RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

<table>
<tr><th colspan="5">Optional Multiply-Divide Instruction Extension: RVM</th></tr>
<tr><th>Category</th><th>Name</th><th>Fmt</th><th>RV32M (Multiply-Divide)</th><th>+RV{64,128}</th></tr>
<tr><td>Multiply</td><td>MULtiply</td><td>R</td><td>MUL rd,rs1,rs2</td><td>MUL{W|D} rd,rs1,rs2</td></tr>
<tr><td></td><td>MULtiply upper Half</td><td>R</td><td>MULH rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>MULtiply Half Sign/Uns</td><td>R</td><td>MULHSU rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>MULtiply upper Half Uns</td><td>R</td><td>MULHU rd,rs1,rs2</td><td></td></tr>
<tr><td>Divide</td><td>DIVide</td><td>R</td><td>DIV rd,rs1,rs2</td><td>DIV{W|D} rd,rs1,rs2</td></tr>
<tr><td></td><td>DIVide Unsigned</td><td>R</td><td>DIVU rd,rs1,rs2</td><td></td></tr>
<tr><td>Remainder</td><td>REMainder</td><td>R</td><td>REM rd,rs1,rs2</td><td>REM{W|D} rd,rs1,rs2</td></tr>
<tr><td></td><td>REMainder Unsigned</td><td>R</td><td>REMU rd,rs1,rs2</td><td>REMU{W|D} rd,rs1,rs2</td></tr>
</table>

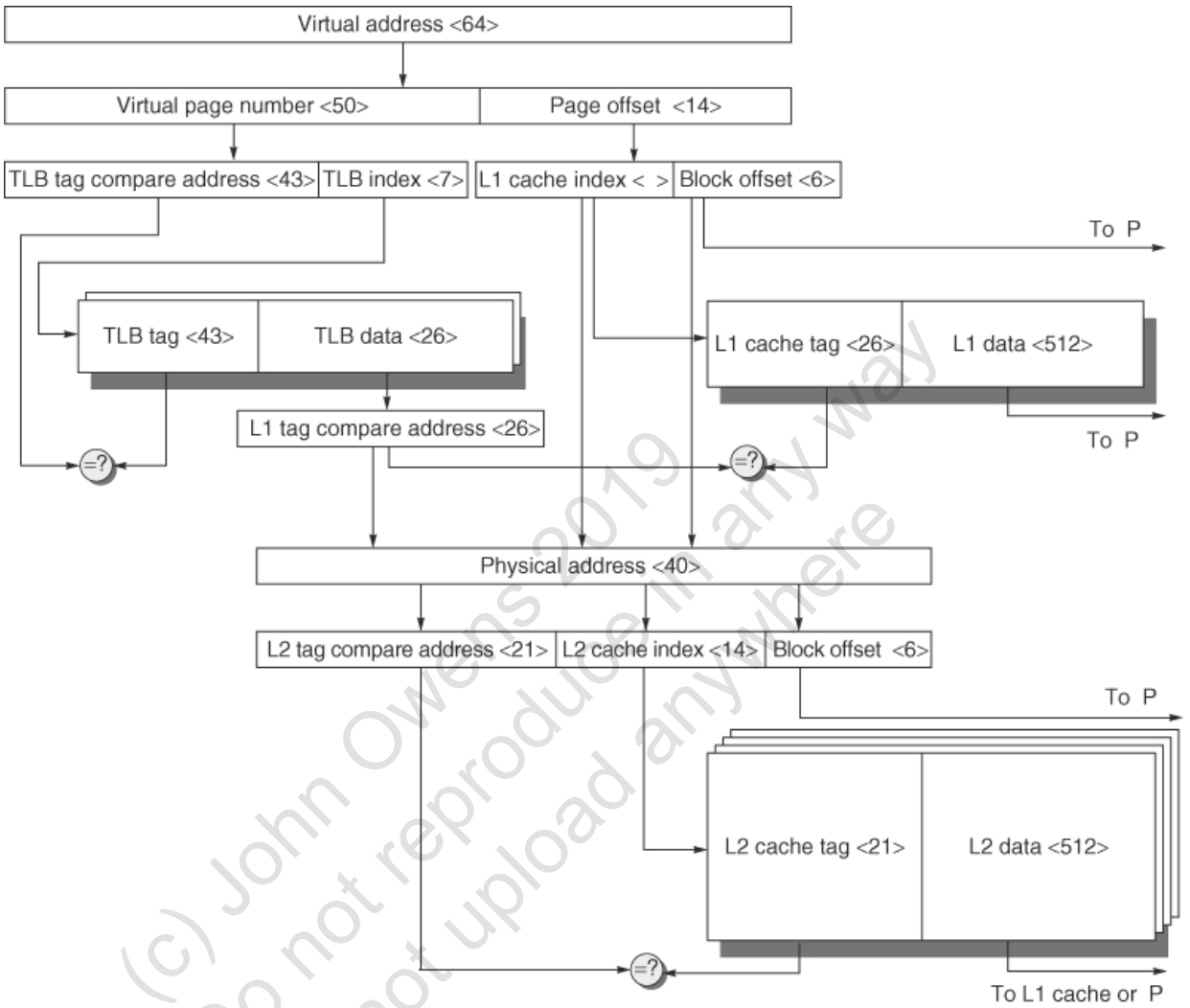
<table>
<tr><th colspan="5">Optional Atomic Instruction Extension: RVA</th></tr>
<tr><th>Category</th><th>Name</th><th>Fmt</th><th>RV32A (Atomic)</th><th>+RV{64,128}</th></tr>
<tr><td>Load</td><td>Load Reserved</td><td>R</td><td>LR.W rd,rs1</td><td>LR.{D|Q} rd,rs1</td></tr>
<tr><td>Store</td><td>Store Conditional</td><td>R</td><td>SC.W rd,rs1,rs2</td><td>SC.{D|Q} rd,rs1,rs2</td></tr>
<tr><td>Swap</td><td>SWAP</td><td>R</td><td>AMOSWAP.W rd,rs1,rs2</td><td>AMOSWAP.{D|Q} rd,rs1,rs2</td></tr>
<tr><td>Add</td><td>ADD</td><td>R</td><td>AMOADD.W rd,rs1,rs2</td><td>AMOADD.{D|Q} rd,rs1,rs2</td></tr>
<tr><td>Logical</td><td>XOR</td><td>R</td><td>AMOXOR.W rd,rs1,rs2</td><td>AMOXOR.{D|Q} rd,rs1,rs2</td></tr>
<tr><td></td><td>AND</td><td>R</td><td>AMOAND.W rd,rs1,rs2</td><td>AMOAND.{D|Q} rd,rs1,rs2</td></tr>
<tr><td></td><td>OR</td><td>R</td><td>AMOOR.W rd,rs1,rs2</td><td>AMOOR.{D|Q} rd,rs1,rs2</td></tr>
<tr><td>Min/Max</td><td>MINimum</td><td>R</td><td>AMOMIN.W rd,rs1,rs2</td><td>AMOMIN.{D|Q} rd,rs1,rs2</td></tr>
<tr><td></td><td>MAXimum</td><td>R</td><td>AMOMAX.W rd,rs1,rs2</td><td>AMOMAX.{D|Q} rd,rs1,rs2</td></tr>
<tr><td></td><td>MINimum Unsigned</td><td>R</td><td>AMOMINU.W rd,rs1,rs2</td><td>AMOMINU.{D|Q} rd,rs1,rs2</td></tr>
<tr><td></td><td>MAXimum Unsigned</td><td>R</td><td>AMOMAXU.W rd,rs1,rs2</td><td>AMOMAXU.{D|Q} rd,rs1,rs2</td></tr>
</table>

<table>
<tr><th colspan="5">Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ</th></tr>
<tr><th>Category</th><th>Name</th><th>Fmt</th><th>RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)</th><th>+RV{64,128}</th></tr>
<tr><td>Move</td><td>Move from Integer</td><td>R</td><td>FMV.{H|S}.X rd,rs1</td><td>FMV.{D|Q}.X rd,rs1</td></tr>
<tr><td></td><td>Move to Integer</td><td>R</td><td>FMV.X.{H|S} rd,rs1</td><td>FMV.X.{D|Q} rd,rs1</td></tr>
<tr><td>Convert</td><td>Convert from Int</td><td>R</td><td>FCVT.{H|S|D|Q}.W rd,rs1</td><td>FCVT.{H|S|D|Q}.{L|T} rd,rs1</td></tr>
<tr><td></td><td>Convert from Int Unsigned</td><td>R</td><td>FCVT.{H|S|D|Q}.WU rd,rs1</td><td>FCVT.{H|S|D|Q}.{L|T}U rd,rs1</td></tr>
<tr><td></td><td>Convert to Int</td><td>R</td><td>FCVT.W.{H|S|D|Q} rd,rs1</td><td>FCVT.{L|T}.{H|S|D|Q} rd,rs1</td></tr>
<tr><td></td><td>Convert to Int Unsigned</td><td>R</td><td>FCVT.WU.{H|S|D|Q} rd,rs1</td><td>FCVT.{L|T}U.{H|S|D|Q} rd,rs1</td></tr>
<tr><td>Load</td><td>Load</td><td>I</td><td>FL{W,D,Q} rd,rs1,imm</td><td></td></tr>
<tr><td>Store</td><td>Store</td><td>S</td><td>FS{W,D,Q} rs1,rs2,imm</td><td></td></tr>
<tr><td>Arithmetic</td><td>ADD</td><td>R</td><td>FADD.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>SUBtract</td><td>R</td><td>FSUB.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>MULtiply</td><td>R</td><td>FMUL.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>DIVide</td><td>R</td><td>FDIV.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>SQuare RooT</td><td>R</td><td>FSQRT.{S|D|Q} rd,rs1</td><td></td></tr>
<tr><td>Mul-Add</td><td>Multiply-ADD</td><td>R</td><td>FMADD.{S|D|Q} rd,rs1,rs2,rs3</td><td></td></tr>
<tr><td></td><td>Multiply-SUBtract</td><td>R</td><td>FMSUB.{S|D|Q} rd,rs1,rs2,rs3</td><td></td></tr>
<tr><td></td><td>Negative Multiply-SUBtract</td><td>R</td><td>FNMSUB.{S|D|Q} rd,rs1,rs2,rs3</td><td></td></tr>
<tr><td></td><td>Negative Multiply-ADD</td><td>R</td><td>FNMADD.{S|D|Q} rd,rs1,rs2,rs3</td><td></td></tr>
<tr><td>Sign Inject</td><td>SiGN source</td><td>R</td><td>FSGNJ.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>Negative SiGN source</td><td>R</td><td>FSGNJN.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>Xor SiGN source</td><td>R</td><td>FSGNJX.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td>Min/Max</td><td>MINimum</td><td>R</td><td>FMIN.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>MAXimum</td><td>R</td><td>FMAX.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td>Compare</td><td>Compare Float =</td><td>R</td><td>FEQ.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>Compare Float <</td><td>R</td><td>FLT.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td></td><td>Compare Float ≤</td><td>R</td><td>FLE.{S|D|Q} rd,rs1,rs2</td><td></td></tr>
<tr><td>Categorization</td><td>Classify Type</td><td>R</td><td>FCLASS.{S|D|Q} rd,rs1</td><td></td></tr>
<tr><td>Configuration</td><td>Read Status</td><td>R</td><td>FRCSR rd</td><td></td></tr>
<tr><td></td><td>Read Rounding Mode</td><td>R</td><td>FRRM rd</td><td></td></tr>
<tr><td></td><td>Read Flags</td><td>R</td><td>FRFLAGS rd</td><td></td></tr>
<tr><td></td><td>Swap Status Reg</td><td>R</td><td>FSCSR rd,rs1</td><td></td></tr>
<tr><td></td><td>Swap Rounding Mode</td><td>R</td><td>FSRM rd,rs1</td><td></td></tr>
<tr><td></td><td>Swap Flags</td><td>R</td><td>FSFLAGS rd,rs1</td><td></td></tr>
<tr><td></td><td>Swap Rounding Mode Imm</td><td>I</td><td>FSRMI rd,imm</td><td></td></tr>
<tr><td></td><td>Swap Flags Imm</td><td>I</td><td>FSFLAGSI rd,imm</td><td></td></tr>
</table>

<table>
<tr><th colspan="4">RISC-V Calling Convention</th></tr>
<tr><th>Register</th><th>ABI Name</th><th>Saver</th><th>Description</th></tr>
<tr><td>x0</td><td>zero</td><td>---</td><td>Hard-wired zero</td></tr>
<tr><td>x1</td><td>ra</td><td>Caller</td><td>Return address</td></tr>
<tr><td>x2</td><td>sp</td><td>Callee</td><td>Stack pointer</td></tr>
<tr><td>x3</td><td>gp</td><td>---</td><td>Global pointer</td></tr>
<tr><td>x4</td><td>tp</td><td>---</td><td>Thread pointer</td></tr>
<tr><td>x5-7</td><td>t0-2</td><td>Caller</td><td>Temporaries</td></tr>
<tr><td>x8</td><td>s0/fp</td><td>Callee</td><td>Saved register/frame pointer</td></tr>
<tr><td>x9</td><td>s1</td><td>Callee</td><td>Saved register</td></tr>
<tr><td>x10-11</td><td>a0-1</td><td>Caller</td><td>Function arguments/return values</td></tr>
<tr><td>x12-17</td><td>a2-7</td><td>Caller</td><td>Function arguments</td></tr>
<tr><td>x18-27</td><td>s2-11</td><td>Callee</td><td>Saved registers</td></tr>
<tr><td>x28-31</td><td>t3-t6</td><td>Caller</td><td>Temporaries</td></tr>
<tr><td>f0-7</td><td>ft0-7</td><td>Caller</td><td>FP temporaries</td></tr>
<tr><td>f8-9</td><td>fs0-1</td><td>Callee</td><td>FP saved registers</td></tr>
<tr><td>f10-11</td><td>fa0-1</td><td>Caller</td><td>FP arguments/return values</td></tr>
<tr><td>f12-17</td><td>fa2-7</td><td>Caller</td><td>FP arguments</td></tr>
<tr><td>f18-27</td><td>fs2-11</td><td>Callee</td><td>FP saved registers</td></tr>
<tr><td>f28-31</td><td>ft8-11</td><td>Caller</td><td>FP temporaries</td></tr>
</table>

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*

Diagram from Problem 3.



UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Fall 2019

Final Examination Solutions

All instructions in this exam, unless otherwise noted, are RISC-V instructions. Recall that if the instruction writes to a register, that register appears first in the instruction. Unless otherwise indicated, you may not use any RISC-V pseudoinstructions on this exam, only actual RISC-V instructions. Feel free to detach the two RISC-V reference pages and the memory system diagram at the end of the exam, but if you do, please take them with you and recycle them (don't try to reattach them or turn them in).

1. We discussed dense matrix-dense matrix multiplication in class. To recap, consider A, B, and C as $n \times n$ matrices; we wish to compute $C = C + A \times B$. This can be computed by

```
1 for i = 1 to n
2   for j = 1 to n
3     for k = 1 to n
4       C(i,j) = C(i,j) + A(i,k) * B(k,j) // this is normal (scalar) + and *
```


We begin by computing the *arithmetic intensity* of this code in operations per memory word, assuming no caches, as a function of n. (Recall that the arithmetic intensity is the number of arithmetic operations per word of memory bandwidth.) Assume n is large; you can approximate any answers given this assumption. Assume each addition or multiplication counts as one operation. Assume all of A, B, and C are stored in main memory (DRAM) and each read from/write to main memory counts as one memory word.

The inner line of code runs n^3 times. It has 2 arithmetic operations, 3 loads, and 1 write. Thus the arithmetic intensity is $\frac{2n^3}{(3+1)n^3} = 1/2$.

(a) (4 points) Consider two different implementations of the same task *with the same runtime*. Baby Yoda's has high arithmetic intensity and is limited by arithmetic capability. The Mandalorian's has low arithmetic intensity and is limited by memory bandwidth. Even though both have the same runtime, we prefer Baby Yoda's implementation with higher arithmetic intensity because of how we expect future machines to behave. In terms of technology trends, **why do we prefer implementations with more arithmetic intensity?**

Solution: Historically, arithmetic capability increases more quickly than memory bandwidth (this is the "memory wall"). A future machine will hopefully continue to increase both arithmetic capability and memory bandwidth, but arithmetic capability will likely increase more, so Baby Yoda's implementation will run faster on future machines than the Mandalorian's.

(b) (4 points) Assume we have a very large cache so all memory loads only read a reference the first time they see that reference and (because they are captured in the cache) all memory writes occur only at the end of the program. Reads from/writes to cache are free (do not count in terms of arithmetic intensity). **What is the arithmetic intensity now?**

Solution: We still do $2n^3$ operations. But now we read each element of A, B, and C once and write each element of C once. That's a total of $4n^2$ memory operations. So the arithmetic intensity is $\frac{2n^3}{4n^2} = n/2$.

(c) (6 points) Now consider a more realistic cache that is not large enough to hold the entire input and output. This cache can hold two matrix rows or columns plus one output element. Assume there are no cache conflicts; any two rows/columns plus one output element can fit in the cache. The Mandalorian proposes using the cache in the following way:

```
for i = 1 to n
  // read row i of A into cache
  for j = 1 to n
    // read row j of B into cache
    // read C(i,j) into cache
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j) // this is normal (scalar) + and *
    // after k loop is complete, write C(i,j) back to memory
```

What is the arithmetic intensity now? It is OK to ignore lower-order terms in this analysis.

Solution: Again we have $2n^3$ operations. The primary memory cost is reading each column of B n times (n^2 column reads $\times$ n elements per column $= n^3$ memory accesses). Thus the arithmetic intensity is $\frac{2n^3}{n^3} = 2$.

More precisely, we do the following memory operations:

- read each column of B n times (n^3 memory loads)

- read each column of A once for each i (n^2 memory loads)
- read and write each element of C once ($2n^2$ memory loads)

for a total of $= n^3 + 3n^2$ memory operations. The arithmetic intensity is thus $\frac{2n^3}{n^3+3n^2}$, roughly 2 for large n.

(d) (10 points) Finally, Baby Yoda proposes to partition the matrix in a different way. Instead of reading 1-dimensional rows or columns from A and B, Baby Yoda arranges those matrices as $b \times b$ subblocks. (For instance, a 128×128 matrix may be structured as a 4×4 grid of 32×32 subblocks.) The cache can hold any two input blocks plus one output block at any time.

Now our matrix multiply looks like this (note our operations are now on blocks at a time instead of rows or columns at a time). The arithmetic, while factored differently, is exactly the same as the previous case.

```
// note i, j, and k are now block indexes, not element indexes
for i = 1 to n/b
  for j = 1 to n/b
    // read block C(i,j) into cache
    for k = 1 to n/b
      // read block A(i,k) into cache
      // read block B(k,j) into cache
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
      // Here + indicates a matrix sum of blocks (block + block)
      // Here * indicates a matrix multiply of blocks (block * block)
    // after k loop is complete, write block C(i,j) back to memory
```

What is the arithmetic intensity now? It is again OK to ignore lower-order terms in this analysis.

Solution: Again we have $2n^3$ operations.

For both A and B, we read $(n/b)^3$ blocks of size b^2. That's $2n^3/b$ reads.

We also read and write each block of C once (n^2 reads, n^2 writes). (For large n, you can ignore this term.)

The arithmetic intensity is thus $\frac{2n^3}{2n^3/b+2n^2}$, roughly b for large n. This is substantially better than the previous cached version (2). It suggests larger b gives better arithmetic intensity. This is true, but it also means the cache must be large enough to hold three blocks of size b.

2. In the 2-page RISC-V Reference Card at the end of the exam, at the bottom right of the first page, you will see the "16-bit (RVC) Instruction Formats". RVC stands for "RISC-V Compressed".

   (a) (5 points) Note CR-type instructions have a `funct4` entry as part of their opcodes instead of a `funct3` (as in all the other instructions). Assume that the opcode combinations `(inst[1:0] == 01 OR inst[1:0] == 10) AND inst[15:13] == 100` are used to encode CR-type instructions. Given the instruction formats listed in the reference page, **how many different instructions can RVC support?** Note not all possible RVC instructions are specified on the reference card.

   > **Solution:** Most instructions have 2 opcode bits and 3 funct bits. This implies 5 opcode bits total, so $2^5 = 32$ instructions can be supported in this way.
   >
   > However, CR-type instructions get an extra opcode bit `instr[12]`. The CR instructions take 2 of the 32 instruction slots available to a 5-bit opcode. Since CR-type instructions have an additional opcode bit, they can fit 4 instructions into those 2 instruction slots.
   >
   > Thus RVC, given these assumptions, supports 34 instructions.

   (b) In practice, these 16-bit RVC instructions are translated to 32-bit full ("RVI") RISC-V instructions. This process is summarized in the table on the reference card called "Optional Compressed (16-bit) Instruction Extension: RVC", but we will highlight the important parts here directly. In this part, you may need to consult the table "16-bit (RVC) instruction formats".

      i. (2 points) The RVC instruction `C.ADD rd, rs1` translates to the RVI instruction `ADD rd, rd, rs1`. Both instructions run a full 64b add. However, the `C.ADD` operation has a significant restriction compared to the full `ADD` instruction. **What is that restriction? Be specific.**

      > **Solution:** `C.ADD` only takes 2 register arguments, so the destination register must also be a source register. `ADD` takes 3 register arguments, so the two source registers can be distinct from the destination register.

      ii. (2 points) Some RVC instructions, like `C.BEQZ`, have a register argument with a prime (apostrophe) after the register name (`C.BEQZ rs1', imm` → `BEQ rs1', x0, imm`). **What is the restriction indicated by the prime?**

      > **Solution:** Registers indicated with a prime are only specified with 3 bits in the instruction encoding, instead of RISC-V's 5 bits. Thus these instructions can only access a subset of registers (this is not important, but it's `x8`–`x15`).

      iii. (2 points) All RVC memory-access instructions multiply their immediates by a

power-of-two constant (e.g., `C.LW rd', rs1', imm` → `LW rd', rs1', imm*4`). Because of this multiplication, **what can a RVI memory-access instruction do that its partner RVC instruction cannot?** (The answer is not "allows byte addressing"; both RVC and RVI memory-access instructions are byte-addressed.)

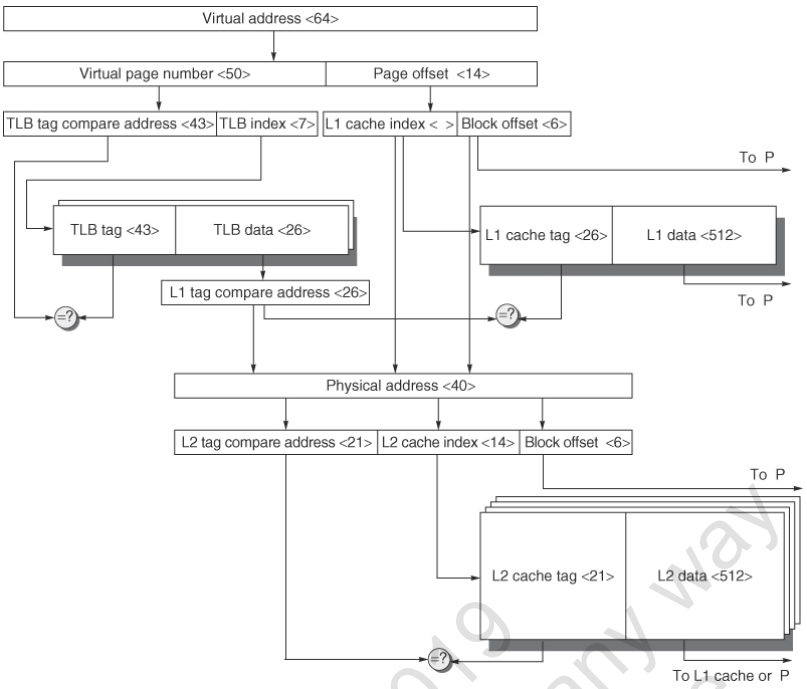
> **Solution:** RVI instructions support misaligned memory operations where the n-byte memory address is not aligned to an n-byte boundary. RVC instructions do not allow misaligned memory addresses.

iv. (2 points) The instruction `C.ADDI16SP` is primarily intended to manipulate the ___**stack**___. (Fill in the previous blank with one word.)

(c) (3 points) The new BabyYoda compiler inputs a RVI (32-bit) RISC-V instruction sequence and transforms some RVI instructions into their RVC (16-bit) equivalents. When running the resulting code, compared to the RVI implementation, **would you expect the instruction cache hit rate would increase, decrease, or stay the same?** Justify your answer.

> **Solution:** The I-cache hit rate should increase. The instruction stream is identical in terms of operations but the instruction stream with RVC instructions takes fewer bytes, takes up less space in the cache, can thus store more instructions than the RVI-only stream, and hence will have a higher hit rate.

3. While sipping his soup, Baby Yoda uses the Force to derive the following characteristics of his laptop's memory system, which is byte-addressed and contains a virtual memory subsystem and L1 and L2 caches. A copy of this diagram is attached at the end of the exam, which you can detach. In this diagram, `<n>` means a signal of `<n>` bits wide.



(a) (3 points) What is the page size?

Solution: The page offset is 14 bits, so the page size is $2^{14} = 16\,\text{kB}$.

(b) (3 points) The TLB has 256 entries in it. **What is the associativity of the TLB** (e.g., 1-way set associative = direct-mapped, 2-way set associative, 4-way, fully associative, etc.)?

Solution: The TLB index has 7 bits, so there are $2^7 = 128$ sets. That's 2 entries per set; the TLB is 2-way set associative.

(c) (3 points) How large is the L1 cache (data only)?

Solution: The cache index is 8 bits, so there are $2^8 = 256$ entries in the cache. Each entry is 512 bits (64 bytes), also shown by a 6-bit block offset.
$256\,\text{entries} \times 64\,\text{bytes}$ is 16 kB.

(d) (3 points) Assuming the L2 cache stores $2^{22}\,\text{B} = 4\,\text{MiB}$ of data, what is its associativity?

Solution: The data per entry is 512 bits = 64 B. Thus there are $4\,\text{MiB}/64\,\text{B} = 64\,\text{k} = 2^{16}$ entries. The cache index has only 14 bits, so that implies 4 entries per set, making the cache 4-way set associative.

(e) (3 points) What is the motivation for making replacement algorithms for hardware caches less complicated than replacement algorithms for pages?

Solution: Either of a couple of general reasons are good answers here:

- Cache replacement algorithms have to run quickly (and hence can't be as complicated) because processing time for the algorithm (thus hit/miss times) must be low.
- Page replacement algorithms must be as accurate as possible (reducing the miss rate) because the miss penalty when pages are not in memory and have to be paged from disk is very long.

4. Assume that you have 2 64-bit double-precision floating point numbers stored in (integer) registers `x2` and `x3`. You run the instruction `XOR x1, x2, x3`. In this question "normal number" means "not zero, not infinity, not denormal, not NaN".

   (a) (3 points) If you know that `x2` is a positive normal number and `x3` is a negative normal number, **what bits in `x1` do you know for certain?** (Your answer should be of the form "Bit 42 will be a 1 and bit 24 will be a 0".) Bit 0 is the least significant bit (LSB) and bit 63 is the most significant bit (MSB).

   Solution: Bit 63 will be 1, because you know the two input sign bits will be different.

   (b) (3 points) If you know that `x2` is a negative denormal number and `x3` is negative infinity, **what bits in `x1` do you know for certain?** (Your answer should be of the form "Bit 42 will be a 1 and bit 24 will be a 0".)

   Solution: Bit 63 will be 0, because both input sign bits are the same.
   Bits 62–52 inclusive will all be 1, because `x2`'s exponent is all zeroes and `x3`'s exponent is all ones.