CSSE 232 – Computer Architecture I
Rose-Hulman Institute of Technology
Computer Science and Software Engineering Department

Exam 1

Name: _____ Section: 1 2 3 4 5

This exam is **closed book**. You are allowed to use the reference card from the book and one 8.5" × 11" single sided page of hand written notes. You may not use a computer, phone, etc. during the examination.

You may use a calculator on this exam.

Write all answers on these pages. Be sure to **show all work** and document your code. Do not use instructions that we have not covered (e.g. no `mul` or `div` but you can use instructions like `slli`, `srl`, etc).

RISC-V code is judged both by its correctness and its efficiency. Unless otherwise stated, you may not use RISC-V pseudoinstructions when writing RISC-V code.

For Pass/Fail problems there will be a redo opportunity for partial credit on a future date. You must submit a good faith effort to qualify for the redo opportunity.

Question	Points	Score
Problem 1	20	
Problem 2	15	
Problem 3	20	
Problem 4	25	
Problem 5	15	
Total:	95	

Problem 1. Consider the following `RISC-V` assembly code snippet disassembled from the `xv6` operating system, with partial machine language translations to the right. Note that the translation is in decimal unless specified otherwise.

```
0x0040 0000 Loop: bge  a1, a0, L8 [___]
0x0040 0004       addi s1, a1, 8
0x0040 0008       srai s1, a5, 12
0x0040 000c       addi a1, a1, 16
0x0040 0010       jal  x0, Loop   [___]
0x0040 0014  L8:  add  a1, a1, x0
0x0040 0018       jalr x0, 0(ra)
```

31	30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8	7	6 5 4 3 2 1 0
0				5		0	
8			11	0	9		0x13
			15		9		
1			-1		0		
	0		11				
							0x67

31	30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8	7	6 5 4 3 2 1 0
0	0	10	11	5	10	0	0x63
8			11	0	9		0x13
0x40C			15	5	9		0x13
16			11	0	11		0x13
1	0x3F8	1	-1		0		0x6F
0x00		0	11	0	11		0x33
0			1	0	0		0x67

Solution:

(a) (12 points) Fill in the 24 missing values in the instructions above. You may write in decimal or in hex, but must clearly indicate if it is in hex. Write any immediates for UJ and SB instructions in base 10 between the square brackets before filling the table. Do not use binary in the table.

(b) (5 points) Consider the following assembled branch instruction. Assuming this instruction is at address 0x0040 0020, what address will the branch go to when it is taken? Show your work.

```
1 111111 00110 00101 000 0000 1 1100011
```

Solution: `beq t0, t1, Loop`
$newPC = oldPC + immediate * 2$
$= 0x00400020 + 2*(-16) = 0x00400020 - 32 = 0x00400020 - 0x20 = 0x00400000$

(c) (3 points) This block of 7 instructions is replicated multiple times throughout the kernel code. Would the `bge` and `jal` instructions have the same machine translation in every replicate as the ones in the table above? Why or why not?

Solution: Yes because everything is PC-relative.

Problem 2. (15 points) Pretend you are an assembler. For each pseudo-instruction in the following table, give a **minimal** sequence of actual RISC-V instructions to accomplish the same thing. You may need to use `x31` for some of the sequences. BIG indicates an immediate value that is 32 bits and SML indicates an immediate value that fits in 12 bits. You may need to refer to specific bits of the immediate by index, e.g. SML[11].

Pseudo-instruction	Description	

lwByIndex t0, t1, t2	t1 contains a pointer to an array, and t2 an index in that array, t0 will get the data from the array at index t2 (t0 = t1[t2]).	**Solution:** slli x31, t2, 2
		add x31, x31, t1
		lw t0, 0(x31)

PUSH a0	Makes space on the stack and then pushes the data in a0 onto the stack.	**Solution:** addi sp, sp, -4 sw a0, 0(sp)
LL12 t0, 0x888	Loads the lower 12 bits of a register, filling the top 20 bits with 0s.	**Solution:** lui t0, {SML, 8b'0} srli t0, t0, 20

Problem 3. Your team is designing a RISC-V-like machine with 16-bit instructions, 12-bit addresses, and 16-bit words. Assume the machine has 64 different opcodes and has 8 registers.

(a) (5 points) Your team creates an instruction format that has an opcode, one register operand, and an immediate. Draw the instruction format for this instruction type. Label each field and show the size (in bits) of each field. Be sure to label any unused bits.

Solution: `[ imm-7b ][ rs-3b ][ op-6 ]`

(b) (5 points) If the above format is used for pc-relative branches, what is the range of branch targets? *Express your answer as the number of instructions before and after the PC where the branch can go (for example, "from PC-400 to PC+200 instructions")*

Solution: with a 7-bit immediate, we can select from $2^1 28$ instructions by shifting the immediate one bit. Signed immediates mean half are negative, and one is zero. This means, we can branch from $PC - 2^6$ instructions to $PC + 2^6 - 1$ instructions; PC-64 to PC+63.

1. Explain your addressing mode, specifically how you use the immediate field to calculate the branch target.

Solution: Something about the immediate being a signed "instruction offset" added to the PC... to make byte offset, shift one bit. `PC = imm << 1 + pc.`

(This problem continues on the next page...)

(c) (5 points) Consider the pseudo-instruction `la` that loads large immediate values *(addresses that are 12-bits in size)* into a register. How should `la` be implemented? Remember, you are the designer – you may use instructions with the format above or design new instruction formats.

Solution: Solutions vary. Beware of sign-extension/zero extension. If immediate field is six-bits and sign-extension is used, gotta deal with that. We're loading addresses, so nothing should be negative. Also NOTE: registers are 16 bits, but the immediate/address is 12 bits. One option: set/shift/ori (note, `o` below is a zero, but displayed as `o` for explanation).

```
li 0xBFC  ==   li 0b10 1111 11 1100

# -- note: set and ori zero-extend the immediate --
set rs, 0x2F  # 6 upper bits  (-> 0boooo oooo oo10 1111)
sll rs, 6     #               (-> 0boooo 1010 10oo oooo)
ori rs, 0x3C  # next 6        (-> 0boooo 1010 1011 1100)
```

(d) (5 points) Justify your design; state *both* the major advantages and the major disadvantages of your design (more than one of each).

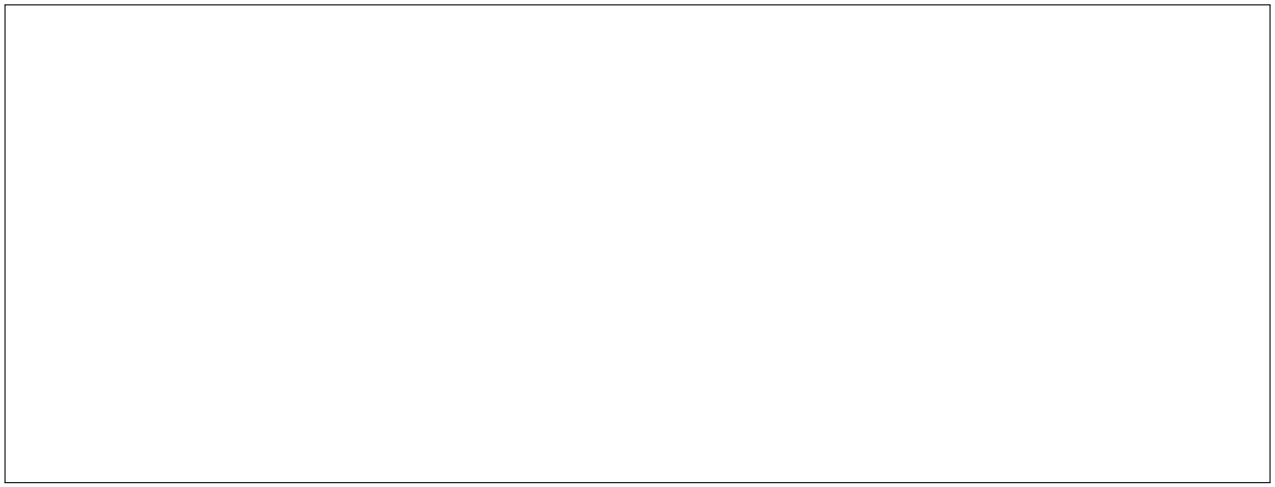
Solution: Advantages: doesn't require new format, only three instruction ops, simple.
Disadvantages: three instructions to load 12 bits. Requires use of "set" and zero-extension.

Problem 4. (25 points) Below is python code for a small program. Based on the code, answer the following questions and then complete the missing portions of the procedure below, adhering to the RISC-V procedure call conventions. Do **NOT** optimize or change the logic of this code.

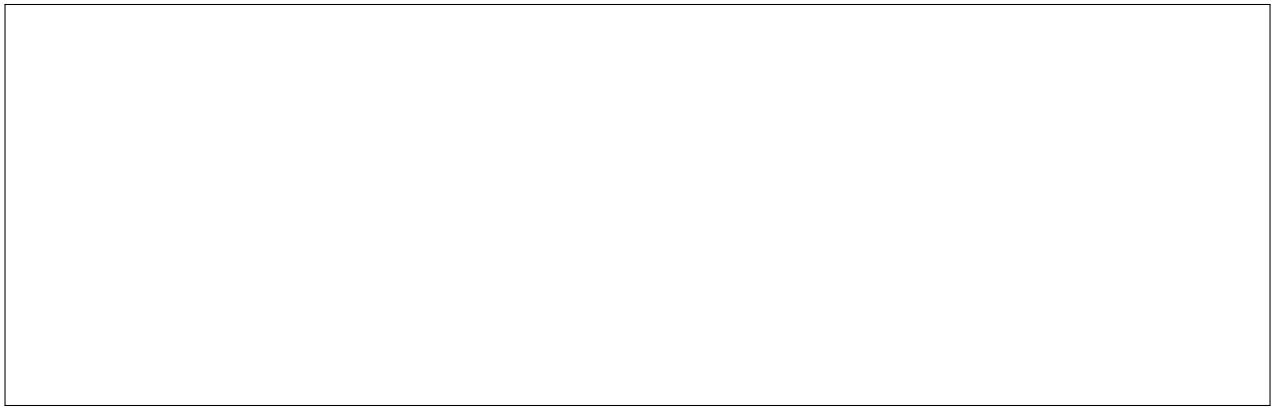
```
def calc_value(A):
    x = A + 7
    y = adjust(x)
    z = y + 3
    z = modify(z, 3)
    result = x - z
    return (result)
```

You can assume that both `adjust` and `modify` are procedures that exist (you do not need to write them), and these procedures follow the RISC-V calling conventions. Assume all local variables (e.g. `x` and `y`) are only stored in registers and not in main memory.

```
calc_value:
```



```
jal ra, adjust ;call to adjust
```



```
jal ra, modify ;call to modify
```

```
jalr x0, 0(ra)
```

Solution: this solution does not use s-registers, but that would be a viable alternative.

```
calc_value:
    addi sp, sp, -8
    sw ra, 0(sp)
    addi a0, a0, 7
    sw a0, 4(sp)
    jal ra, adjust

    addi a0, a0, 3
    addi a1, x0, 3
    jal ra, modify

    lw t0, 4(sp)
    sub a0, t0, a0
    lw ra, 0(sp)
    addi sp, sp, 8
    jalr x0, 0(ra)
```

Problem 5. You are the lead designer for a real-time 32-bits RISC-V processor. Your customer has demanded that the execution time of the benchmark program should not be longer than 5 μ seconds.

(a) (5 points) Your customer has provided you with a sample set of benchmarks that they wish to run on your processor. After some calculations, your team collects the following data from the benchmark suite. Calculate the average CPI for your processor.

Instruction Type	CPI	Count
`mem`	10	400
`branch`	3	50
`arithmetic`	7	200
`logical`	3	60
`jump`	1	40

Solution: 400 + 50 + 200 +60 + 40 = 750 inst
4000 + 150 + 1400 + 180 + 40 = 5770 cycles
5770 / 750 = 7.70 CPI

(b) (5 points) What is the minimum clock frequency (remember, $frequency = \frac{1}{cycle\ time}$) that your processor must run at in order to meet the customer's requirements?

Solution: ET = inst * CPI * cycletime
$5 * 10^{-6} = 750 * 5770/750 * F = 5770 * F$
$F = 8.67 * 10^{-4} * 10^{-6} = 8.67 * 10^{-10} = 0.867 * 10^{-9} sec/cycle$
$= 1.15 * 10^{9} cycles/sec = 1.15 GHz$

question continues on next page...

(c) (5 points) Your team noticed that a large portion of the benchmark program is made up of memory operations, and thus they have suggested adding a few complex instructions that can read and write from memory in one instruction. After some analysis, this leads to reducing the number of memory instructions by **25%**. However, memory instructions now take **12 cycles** to execute. What is the new average CPI?

Solution: 300 + 50 + 200 +60 + 40 = 650 inst
(300*12) + 150 + 1400 + 180 + 40 = 5370 cycles
5370 / 650 = 8.26 CPI

(d) (5 points) Assuming you keep the processor running at the same minimum frequency calculated earlier, will the new design still meet the clients requirements? Show your work to support your answer.

Solution: ET = inst * CPI * cycletime
$ET = 650 * 5370/650 * 0.867 * 10^{-9} = 5.37 * 10^3 * 0.867 * 10^{-9} = 4.66 * 10^{-6} sec = 4.66 \mu sec$
Yes ET still less than 5 μ sec

RISC-V Reference

Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description	Note
add	ADD	R	0110011	000	000 0000	R[rd] = R[rs1] + R[rs2]	
sub	SUB	R	0110011	000	010 0000	R[rd] = R[rs1] - R[rs2]	
xor	XOR	R	0110011	100	000 0000	R[rd] = R[rs1] ^ R[rs2]	
or	OR	R	0110011	110	000 0000	R[rd] = R[rs1] | R[rs2]	
and	AND	R	0110011	111	000 0000	R[rd] = R[rs1] & R[rs2]	
sll	Shift Left Logical	R	0110011	001	000 0000	R[rd] = R[rs1] << R[rs2]	
srl	Shift Right Logical	R	0110011	101	000 0000	R[rd] = R[rs1] >> R[rs2]	
sra	Shift Right Arith*	R	0110011	101	010 0000	R[rd] = R[rs1] >> R[rs2]	sign-extends
slt	Set Less Than	R	0110011	010	000 0000	R[rd] = (rs1 < rs2)?1:0	
addi	ADD Immediate	I	0010011	000		R[rd] = R[rs1] + SE(imm)	
xori	XOR Immediate	I	0010011	100		R[rd] = R[rs1] ^ SE(imm)	
ori	OR Immediate	I	0010011	110		R[rd] = R[rs1] | SE(imm)	
andi	AND Immediate	I	0010011	111		R[rd] = R[rs1] & SE(imm)	
slli	Shift Left Logical Imm	I	0010011	001	imm[11:5] =0x00	R[rd] = R[rs1] << imm[4:0]	
srli	Shift Right Logical Imm	I	0010011	101	imm[11:5] =0x00	R[rd] = R[rs1] >> imm[4:0]	
srai	Shift Right Arith Imm	I	0010011	101	imm[11:5] =0x20	R[rd] = R[rs1] >> imm[4:0]	sign-extends
lw	Load Word	I	0000011	010		R[rd] = M[R[rs1]+SE(imm)]	
sw	Store Word	S	0100011	010		M[R[rs1]+SE(imm)] = R[rs2]	
beq	Branch ==	SB	1100011	000		if(rs1 == rs2) PC += SE(imm) << 1	
bne	Branch !=	SB	1100011	001		if(rs1 != rs2) PC += SE(imm) << 1	
blt	Branch <	SB	1100011	100		if(rs1 < rs2) PC += SE(imm) <<1	
bge	Branch >=	SB	1100011	101		if(rs1 >= rs2) PC += SE(imm) <<1	
jal	Jump And Link	UJ	1101111			R[rd] = PC+4; PC += SE(imm) <<1	
jalr	Jump And Link Reg	I	1100111	000		R[rd] = PC+4; PC = R[rs1]+ SE(imm)	
lui	Load Upper Imm	U	0110111			R[rd] = SE(imm) << 12	
auipc	Add Upper Imm to PC	U	0010111			R[rd] = PC + (SE(imm) << 12)	
csrrw	CSR read & write	I	1110011	001		R[rd] = CSRs[csr]; CSRs[csr] = R[rs1]	
csrrs	CSR read & set	I	1110011	010		R[rd] = CSRs[csr]; CSRs[csr] = CSRs[csr] | R[rs1]	
csrrc	CSR read & clear	I	1110011	011		R[rd] = CSRs[csr]; CSRs[csr] = CSRs[csr] & ∼R[rs1]	
ecall	Environment Call	I	1110011	000	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	000	imm=0x1	Transfer control to debugger	

R = Register file access CSR = Control and Status Register
SE = Sign extend CSRs = Coprocessor registers

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	SB-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	UJ-type

Opcodes, Base conversion

Binary	Hex	Opcode
000 0000	00	
000 0001	01	
000 0010	02	
000 0011	03	lw
000 0100	04	
000 0101	05	
000 0110	06	
000 0111	07	
000 1000	08	
000 1001	09	
000 1010	0A	
000 1011	0B	
000 1100	0C	
000 1101	0D	
000 1110	0E	
000 1111	0F	
001 0000	10	
001 0001	11	
001 0010	12	
001 0011	13	I-type
001 0100	14	
001 0101	15	
001 0110	16	
001 0111	17	auipc
001 1000	18	
001 1001	19	
001 1010	1A	
001 1011	1B	
001 1100	1C	
001 1101	1D	
001 1110	1E	
001 1111	1F	

Binary	Hex	Opcode
010 0000	20	
010 0001	21	
010 0010	22	
010 0011	23	sw
010 0100	24	
010 0101	25	
010 0110	26	
010 0111	27	
010 1000	28	
010 1001	29	
010 1010	2A	
010 1011	2B	
010 1100	2C	
010 1101	2D	
010 1110	2E	
010 1111	2F	
011 0000	30	
011 0001	31	
011 0010	32	
011 0011	33	R-type
011 0100	34	
011 0101	35	
011 0110	36	
011 0111	37	lui
011 1000	38	
011 1001	39	
011 1010	3A	
011 1011	3B	
011 1100	3C	
011 1101	3D	
011 1110	3E	
011 1111	3F	

Binary	Hex	Opcode
100 0000	40	
100 0001	41	
100 0010	42	
100 0011	43	
100 0100	44	
100 0101	45	
100 0110	46	
100 0111	47	
100 1000	48	
100 1001	49	
100 1010	4A	
100 1011	4B	
100 1100	4C	
100 1101	4D	
100 1110	4E	
100 1111	4F	
101 0000	50	
101 0001	51	
101 0010	52	
101 0011	53	
101 0100	54	
101 0101	55	
101 0110	56	
101 0111	57	
101 1000	58	
101 1001	59	
101 1010	5A	
101 1011	5B	
101 1100	5C	
101 1101	5D	
101 1110	5E	
101 1111	5F	

Binary	Hex	Opcode
110 0000	60	
110 0001	61	
110 0010	62	
110 0011	63	SB-type
110 0100	64	
110 0101	65	
110 0110	66	
110 0111	67	jalr
110 1000	68	
110 1001	69	
110 1010	6A	
110 1011	6B	
110 1100	6C	
110 1101	6D	
110 1110	6E	
110 1111	6F	jal
111 0000	70	
111 0001	71	
111 0010	72	
111 0011	73	exceptions
111 0100	74	
111 0101	75	
111 0110	76	
111 0111	77	
111 1000	78	
111 1001	79	
111 1010	7A	
111 1011	7B	
111 1100	7C	
111 1101	7D	
111 1110	7E	

Registers

Register	Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x30	t3-t5	Temporaries	Caller
x31	at	Assembler Temporary	Caller

Memory Allocation