CS 211 S25 (Sections 5–8): **Final Exam Practice**

Prof. Minesh Patel

Friday, 5 May

Please sign your name and NetID below, or your work may not be graded.

I certify that the answers to this exam represent my own work and follow the RU academic integrity policies (`cs.rutgers.edu/academics/undergraduate/academic-integrity-policy`*).*

PRINT Your Name:

Your NetID (e.g., mp2099)

Instructions

- Each question is marked with its point value, totaling 400 across the exam. Choose wisely.
- Write your **final answer** inside the answer box; your work outside it.
- The Documentation appendix provides any documentation you may need (e.g., ASCII table).
- Please wait at your desk when you're finished: we will collect your exams.

Default Assumptions

Unless otherwise stated, assume:

- All provided code is syntactically correct.
- All provided code uses correct header file includes (omitted for brevity).
- All machines are 64-bit little-endian, just like your `ilab` environment.
- `sizeof(char) == 1`, `sizeof(short) == 2`, `sizeof(int) == 4`, `sizeof(long) == 8`
- Floating point numbers use the standard IEEE 754 32-bit representation we learned in class.
- All multiple select questions have partial credit with a hidden partial penalty for incorrect selections.

Exam Policies

- This is a closed book, closed notes exam.
- No electronic devices are permitted. Please turn them off: accessing any prohibited materials during the exam will lead to a score of 0.

Q1 [0 Points] True or False

Circle True (T) or False (F) for each of the following questions.

T [F] I forgot to write down my name and ID number.

T [F] Three AND gates can be used to express a XOR operation.

[T] F 32-bit IEEE 754 floating-point numbers can precisely represent all 16-bit unsigned integers.

T [F] The target of a RISC-V branch instruction can be located at most $\pm$ 2 MiB away from the branch instruction.

T [F] UTF-8 characters are each 1-byte wide.

[T] F The ISA defines the interface between the software and the hardware.

T [F] Larger caches generally require less time to access.

[T] F C's `goto` statements are comparable to unconditional jump operations in assembly.

[T] F Labels in C identify specific positions in the code.

Q2 [0 Points] Boolean Algebra Simplification

Fully simplify the given boolean algebra expressions:

(a) $AB + A'B + AB'$

$B + AB'$ or $A + B$

(b) $(A + B + C)(A' + B)(B' + C')$

$BC' + A'B'C$

Q3 [0 Points] RISC-V Array

Suppose we run the following RISC-V assembly program:

```
.globl arr_func
arr_func:
    mv   a2, x0
    mv   a3, x0
    mv   a4, x0
    mv   a5, x0
    mv   a6, x0
loop:
    beq  a3, a1, done
    add  a4, a3, a3
    add  a4, a4, a4
    add  a5, a0, a4
    lw   a6, 0(a5)
    blt  a6, x0, skip
    add  a2, a2, a6
skip:
    addi a3, a3, 1
    j    loop
done:
    mv   a0, a2
    ret
```

If this function was called so that the first input was the memory address of an `int32_t` array [-2, 3, 7, -1, 2], and the second input was 5 (the length of the array), what does `arr_func` return?

Answer: 12

Q4 [0 Points] Cache Reasoning

What is the general motivation behind using caches?

- Creating the illusion that memory is both large and fast (by storing the most frequently used information where it can be accessed at higher speeds).
- Accessing main memory is significantly slower than accessing caches.

Q5 [0 Points] Struct Types

Kevin is writing a function to insert a new node at the head of a singly-linked list. Help him figure out how to update the head pointer.

```
1  #include <stdlib.h>
2
3  struct node
4  {
5      void *data;
6      struct node *next;
7  };
8
9  void insert_at_head(struct node **head)
10 {
11     struct node *new_head = (struct node *)malloc(sizeof(struct node));
12     new_head->data = NULL;
13     new_head->next = *head;
14     *head = new_head;
15 }
16
17 int main(int argc, char **argv)
18 {
19     struct node head = {NULL, NULL};
20     struct node *list = &head;
21     insert_at_head(&list);
22     return EXIT_SUCCESS;
23 }
```

Q6 [0 Points] MUX Implementation

Implement the following equation using only 2:1 MUXes. Each mux input may only be (i) a single variable (or its negation, e.g., a'); (ii) hardwired 0 or 1; (iii) or the output of another MUX. *Hint: use the MUX truth table to figure out how to perform AND and OR operations on its inputs.*

$$(a + b)cd$$

3 MUXes. We can rewrite this equation as $(ab' + 1b)(0d' + cd) = XY$, where $X = (ab' + 1b)$ and $Y = (0d' + cd)$. Then, the final mux is $0Y' + XY$.

Q7 [0 Points] Sign Extension

All of the following are well-defined typecasts in C because the destination type can correctly represent the source type.

Give the **hexadecimal representation** of the **entire destination object**, including leading zeroes. For example, a `uint16_t` storing the value 0 has representation `0x0000`.

- `int32_t a = (int8_t)-128; // a=0x` ffffff80
- `uint16_t a = (int8_t)13; // a=0x` 000d
- `int16_t a = (uint8_t)127; // a=0x` 007f

Documentation

Selected Boolean Identities

Name	Identity	
Commutative (AND)	$a \cdot b$	$b \cdot a$
Commutative (OR)	$a + b$	$b + a$
Associative (AND)	$a \cdot (b \cdot c)$	$(a \cdot b) \cdot c$
Associative (OR)	$a + (b + c)$	$(a + b) + c$
Distributive (AND)	$a \cdot (b + c)$	$(a \cdot b) + (a \cdot c)$
Distributive (OR)	$a + (b \cdot c)$	$(a + b) \cdot (a + c)$
DeMorgan's (AND)	$(a \cdot b)'$	$a' + b'$
DeMorgan's (OR)	$(a + b)'$	$a' \cdot b'$
Absorption	$a + (a \cdot b)$	a
Absorption	$a \cdot (a + b)$	a
Redundancy	$a \cdot b' + b$	$a + b$
Redundancy	$(a + b) \cdot (a + b')$	a
Redundancy	$(a \cdot b) + (a \cdot b')$	a
Redundancy	$(a + b) \cdot (a' + b)$	b
Consensus	$a \cdot b + a' \cdot c + b \cdot c$	$a \cdot b + a' \cdot c$
Covering	$a + a' \cdot b$	$a + b$

RISC-V Reference Sheet

1. RV32I and RV64I Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith	R	0110011	0x5	0x20	rd = rs1 >> rs2	sext
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	signed
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	unsigned
addi	ADD (Immediate)	I	0010011	0x0		rd = rs1 + imm	
xori	XOR (Immediate)	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR (Immediate)	I	0010011	0x6		rd = rs1 | imm	
andi	AND (Immediate)	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00	rd = rs1 >> imm[0:4]	zext
srai	Shift Right Arith Imm	I	0010011	0x5	imm[11:5]=0x20	rd = rs1 >> imm[0:4]	sext
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	signed
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	unsigned
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	sext
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	sext
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	sext
ld	Load Double Word	I	0000011	0x3		rd = M[rs1+imm][0:63]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zext
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zext
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]	
sd	Store Double Word	S	0100011	0x3		M[rs1+imm][0:63] = rs2[0:63]	
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	signed
bge	Branch $\geq$	B	1100011	0x5		if(rs1 >= rs2) PC += imm	signed
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	unsigned
bgeu	Branch $\geq$ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	unsigned
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)	

Note: `w` variants (e.g., addiw) operate on 32-bit quantities (not listed here).

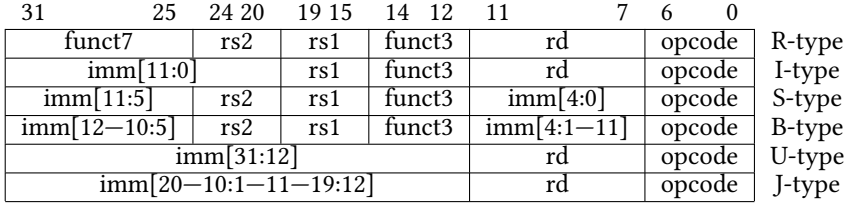
2. RISC-V Pseudoinstructions

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine

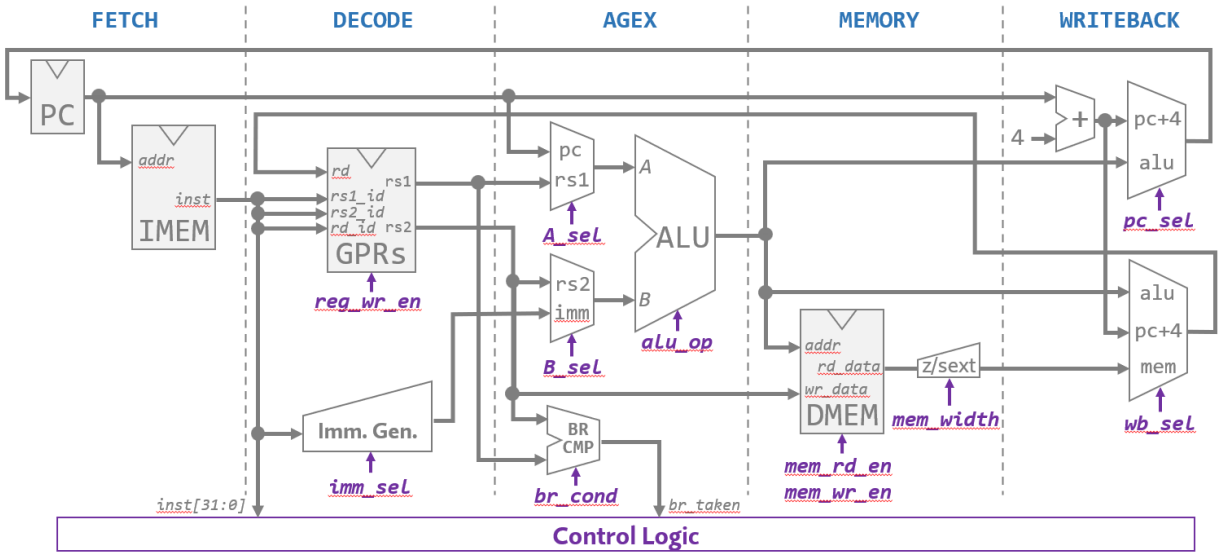
3. General-Purpose Registers

Register	ABI 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-x31`	`t3-t6`	Temporaries	Caller

4. RISC-V Instruction Encodings



Single-Cycle RV64I CPU Datapath



`memset` from `<string.h>`

```
1  void *memset( void *dest, int ch, size_t count );
```

Description:
Copies the value `(unsigned char)ch` into each of the first count characters of the object pointed to by `dest`. The behavior is undefined if access occurs beyond the end of the `dest` array. The behavior is undefined if `dest` is a `NULL` pointer.
Parameters:

- `dest`: pointer to the object to fill
- `ch`: fill byte
- `count`: number of bytes to fill
- `destsz`: size of the dest array

Return Value:
A copy of `dest`

`memcpy` from `<string.h>`

```
1  void *memcpy(void * dest, const void * source, size_t count);
```

Description:
Copies the values of `count` bytes from the location pointed to by `source` directly to the memory block pointed to by `dest`. To avoid overflows, the size of the arrays pointed to by both the `dest` and `source` parameters, shall be at least `count` bytes, and should not overlap (for overlapping memory blocks, memmove is a safer approach).
Parameters:

- `dest`: pointer to the object to copy to
- `src`: pointer to the object to copy from
- `count`: number of bytes to copy

Return Value:
A copy of `dest`

Relevant ASCII Codepoints

Hex Value	0x61	0x62	0x63	0x64	0x65	0x66	0x67	0x68	0x69	0x6a	0x6b	0x6c	0x6d
Character	a	b	c	d	e	f	g	h	i	j	k	l	m

Hex Value	0x6e	0x6f	0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7a
Character	n	o	p	q	r	s	t	u	v	w	x	y	z