MT Review Session 2: RISC-V

TAs: Nithila Poongovan

Credit to: Nikhil Kandkur, Sonika Vuyyuru, Esme Cohen, Nitin Nazeer, Caroline Liu

Midterm PSA

Please do not include anything typed/printed on your cheatsheet!
We will be confiscating cheat sheets that do not meet this requirement…

Introduction to RISC-V

- RISC-V is an assembly language
  - Programs are translated to assembly via the compiler
- RISC-V ISA has 32 registers, including a program counter
  - x0: always stores 0
  - t-registers: temporary registers
  - a-registers: argument registers
  - s-registers: save registers
  - sp: stores the stack pointer
  - ra: stores the return address of the caller

High-level language (C)	int x = 5;
Low-level language (RISC-V)	addi a0, x0, 5
Machine code	10100000000010100010011

Learn to use the CS61C Reference sheet!

#	Name	Description	#	Name	Desc
x0	zero	Constant 0	x16	a6	*Args*
x1	ra	*Return Address*	x17	a7	
x2	sp	Stack Pointer	x18	s2	*Saved Registers*
x3	gp	Global Pointer	x19	s3	
x4	tp	Thread Pointer	x20	s4	
x5	t0	*Temporary Registers*	x21	s5	
x6	t1		x22	s6	
x7	t2		x23	s7	
x8	s0	Saved Registers	x24	s8	
x9	s1		x25	s9	
x10	a0	*Function Arguments or Return Values*	x26	s10	
x11	a1		x27	s11	
x12	a2	*Function Arguments*	x28	t3	*Temporaries*
x13	a3		x29	t4	
x14	a4		x30	t5	
x15	a5		x31	t6	
Caller saved registers					
Callee saved registers (except x0, gp, tp)					

CORE INSTRUCTION FORMATS

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

Instruction	Name	Description	Type	Opcode	Funct3	Funct7
`add rd rs1 rs2`	ADD	`rd = rs1 + rs2`	R	`011 0011`	`000`	`000 0000`

Calling Convention

By convention, registers are classified as one of …

- ***caller-saved***
  - The function invoked (the callee) can do whatever it wants to them!
  - Means that the caller can not count on their contents not being destroyed
- ***callee-saved***
  - The function invoked must restore them before returning (if used)

JUMP!

IMPORTANT: the immediate in any jump AND branch instruction is NOT the address of the instruction it is moving to, but THE OFFSET to that instruction

	Label provided	Register provides address to jump to
Do not return after call	j [label]	jr [reg]
Return after call (LINK)	jal [rd] [label]	jalr [rd] [reg] [imm]

Warm-Up Exam Questions…

Q3.3 (6 points) Write a sequence of at most two instructions or pseudoinstructions that are equivalent to the `j loop` instruction.

You must use a `jalr` instruction or `jalr` pseudoinstruction in at least one of the blanks. You may not use a `jal` instruction, branch instruction, or `jal` pseudoinstruction in any of the blanks.

1 _____

2 _____

SU22 Final Q3.3

SU23 Midterm Q4.1

Q4.1 (2 points) We want to create a pseudoinstruction to check whether a number is odd or not. This instruction, written `is_odd rd rs1`, will put the value 1 in `rd` if the value in `rs1` is odd, and the value 0 otherwise. What is the RISC-V instruction that `is_odd rd rs1` would translate to? You may only use one instruction, and you may not use any pseudoinstructions.

Note: Your solution may include `rd` and `rs1`.

Warm-Up Exam Questions…

SU22 Final Q3.3

Q3.3 (6 points) Write a sequence of at most two instructions or pseudoinstructions that are equivalent to the `j loop` instruction.

You must use a `jalr` instruction or `jalr` pseudoinstruction in at least one of the blanks. You may not use a `jal` instruction, branch instruction, or `jal` pseudoinstruction in any of the blanks.

1 `la t0, loop`

2 `jalr x0 t0 0`

SU23 Midterm Q4.1

Q4.1 (2 points) We want to create a pseudoinstruction to check whether a number is odd or not. This instruction, written `is_odd rd rs1`, will put the value 1 in `rd` if the value in `rs1` is odd, and the value 0 otherwise. What is the RISC-V instruction that `is_odd rd rs1` would translate to? You may only use one instruction, and you may not use any pseudoinstructions.

Note: Your solution may include `rd` and `rs1`.

`andi rd rs1 1`

Exam question: Summer 2019 Midterm 1 Q5 (modified)

Implement strncpy in RISC-V.

```
char* strncpy(char* destination, char* source, unsigned int n);
```

`strncpy` takes in two `char*` arguments and copies up to the first n characters from source into destination. **If it reaches a null terminator, then it copies that value into destination and stops copying in characters.** If there is no null terminator among the first n characters of source, the string placed in destination will not be null-terminated.

`strncpy` returns a pointer to the `destination` string.

Assume a0 = destination, a1 = sources, a2 = n

TIP: come up with pseudocode first!

Exam question: Summer 2019 Midterm 1 Q5 (modified)

Pseudocode:

```
i = 0
while (i != n):
        copy_one_char_from_source_to_dest
        if source_is_null_terminator:
                return dest
        increment source pointer
return dest
```

Summer 2019 Midterm 1 Q5 (modified)

Pseudocode:

```
i = 0
while (i != n):
        copy_one_char_from_source_to_dest
        if source_is_null_terminator:
                return dest
        increment source pointer
return dest
```

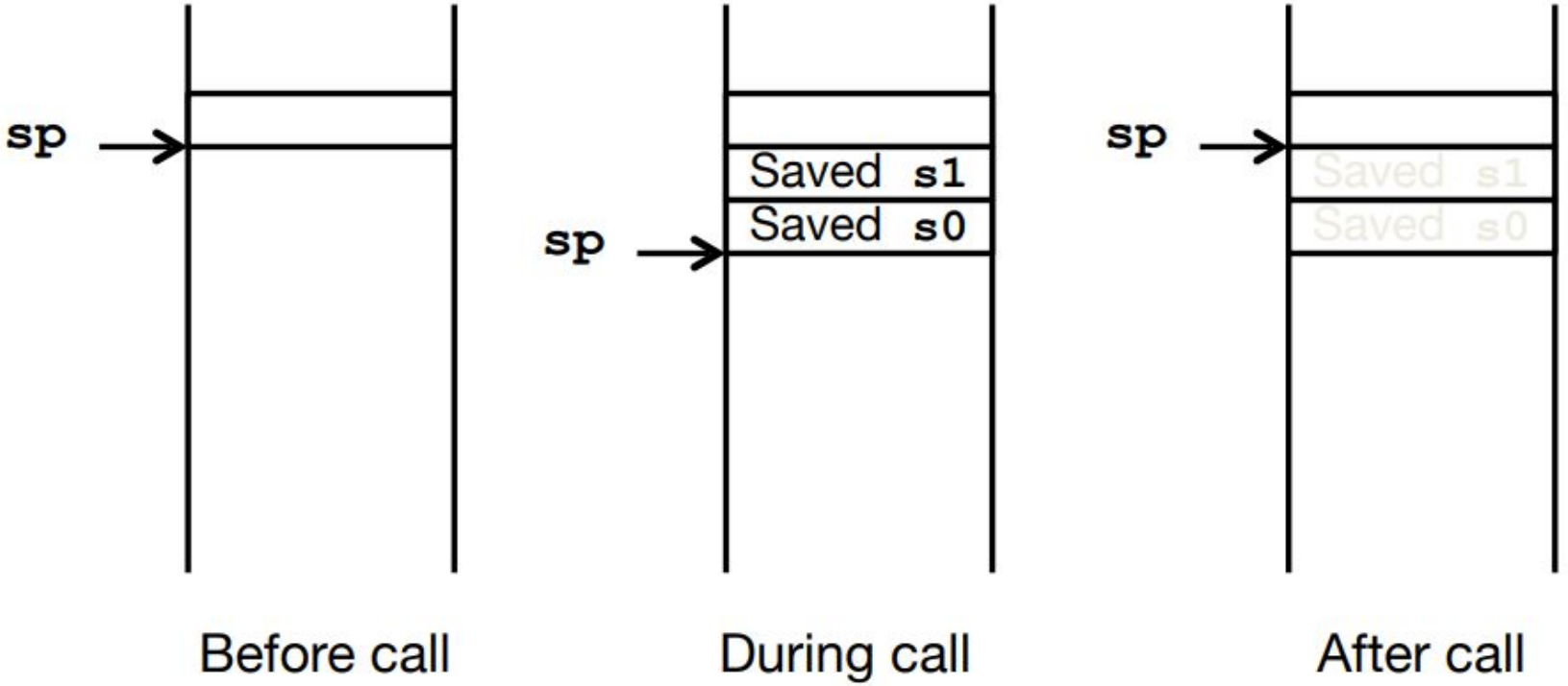
RISC-V Code:

```
# a0 = dest, a1 = source, a2 = n
strncpy:
    li t0, 0                # t0 = counter = 0
    mv t2, a0               # t2 = storing ptr to dest start
loop:
    beq t0, a2, end         # if counter = n, end
    lb t1, 0(a1)            # char = source[a1]
    sb t1, 0(a0)            # dest[a0] = char
    beq t1, x0, end         # if char == '/0', end
    addi t0, t0, 1          # counter += 1
    addi a1, a1, 1          # source += 1
    addi a0, a0, 1          # dest += 1
    j loop                  # repeat loop
end:
    mv a0, t2               # t2 contains the original dest
    ret
```

General Calling Convention strategy

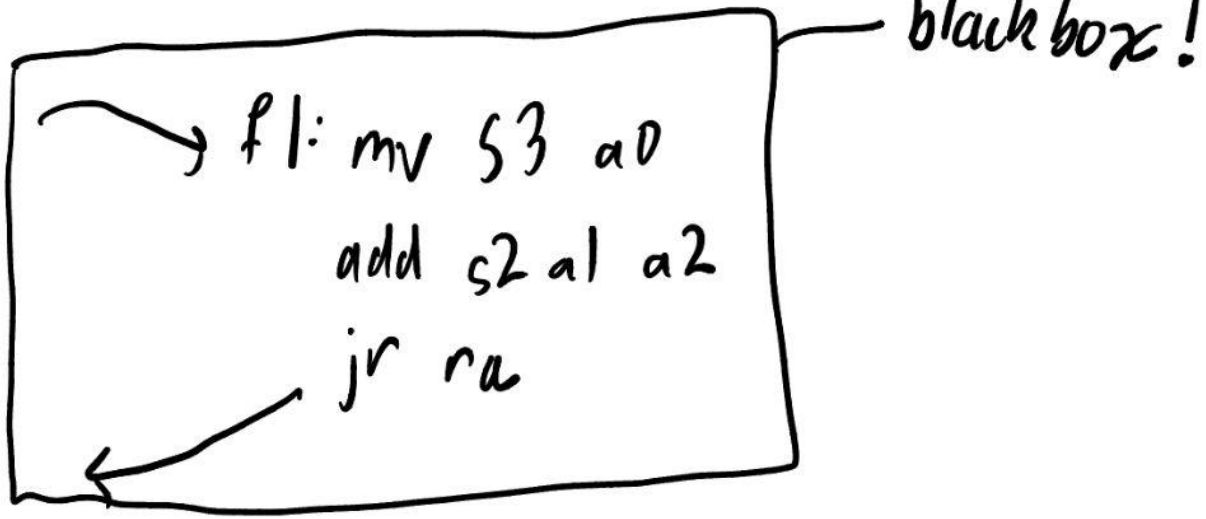
1. Put parameters in a place where function can access them
   - Put parameters in argument registers
2. Transfer control to function
   - Jump instruction
3. Acquire (local) storage resources needed for function
   - Make room for local variables on stack
4. Perform desired task of the function
5. Put result value in a place where caller code can access it
   - a0-a1 register
6. Return control to point of origin
   - `Ret = jr ra = jalr x0 ra 0`

Storing on the stack: Visual



# assume a0=2
a1=3

```
main:
    addi s2 x0 1
    mul s3 a0 a1
    jal ra f1
```




```
    mv a0 s3
    slli s2 s2 2
```

Exam question: Summer 2018 Midterm 1 Q3

1. Fill in the following RISC-V code so that it properly follows convention. Assume that all labels not currently in the code are external functions. You may not need all the lines provided.

```
Pro:
      _____________________________
      _____________________________
      _____________________________
      _____________________________
      _____________________________
      _____________________________
      _____________________________
Body:
      mv s1 a0
      jal ra foo
      mv s2 a0
      addi a0 x0 6
Loop:
      beq a0 x0 Epi
      addi a0 a0 -1
      mv s3 a0
      jal ra foo
      addi s2 s2 a0
      mv a0 s3
      j Loop
Epi:
      _____________________________
      _____________________________
      _____________________________
      _____________________________
      _____________________________
      _____________________________
```

Exam question: Summer 2018 Midterm 1 Q3

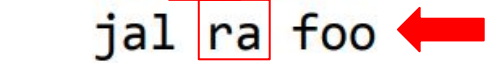
```
Pro:
      addi  sp sp -16
      sw    ra 0(sp)
      sw    s1 4(sp)
      sw    s2 8(sp)
      sw    s3 12(sp)
```

```
Body:
      mv s1 a0
      jal ra foo
      mv s2 a0
      addi a0 x0 6
Loop:
      beq a0 x0 Epi
      addi a0 a0 -1
      mv s3 a0
      jal ra foo
      addi s2 s2 a0
      mv a0 s3
      j Loop
```

Exam question: Summer 2018 Midterm 1 Q3

```
Pro:
        addi  sp sp -16
        sw    ra 0(sp)
        sw    s1 4(sp)
        sw    s2 8(sp)
        sw    s3 12(sp)
Epi:
        lw    s3 12(sp)
        lw    s2 8(sp)
        lw    s1 4(sp)
        lw    ra 0(sp)
        addi sp sp 16
        jr ra
```

```
Body:
        mv s1 a0
        jal ra foo
        mv s2 a0
        addi a0 x0 6
Loop:
        beq a0 x0 Epi
        addi a0 a0 -1
        mv s3 a0
        jal ra foo
        addi s2 s2 a0
        mv a0 s3
        j Loop
```



RISC-V Instruction Formats

- To convert a RISC-V instruction to binary, we first need to know what instruction format it belongs to
  - R-format: belongs to general arithmetic instructions with only (R)egisters
  - I-format: belongs to instructions that use an (I)mmediate
    - Load instructions and JALR are I instructions!!
    - Beware of I and I* instructions (I* contains slli, srli, srai)
  - S-format: belongs to (s)ave instructions
  - U-format: belong to instructions that use (u)pper immediates
  - B-format: belong to instructions that (b)ranch
  - J-format: belongs to instructions that have unconditional (j)umps!

EXAM QUESTION: Summer 2018 MT1 Q5

You are given the following RISC-V code:

```
Loop:   andi  t2 t1 1
        srli  t3 t1 1
        bltu  t1 a0 Loop
        jalr s0 s1 MAX_POS_IMM
        ...
```

1) What is the value of the **byte offset** that would be stored in the immediate field of the `bltu` instruction?

_____

2) What is the binary encoding of the `bltu` instruction? Feel free to use the following space for scratch work—it will not be graded. Put your final answer in hexadecimal.

31 0

0x_____

Exam question: Summer 2018 Midterm 1 Q5

You are given the following RISC-V code:

```
Loop:   andi  t2 t1 1
        srli  t3 t1 1
        bltu  t1 a0 Loop
        jalr s0 s1 MAX_POS_IMM
        ...
```

1) What is the value of the **byte offset** that would be stored in the immediate field of the `bltu` instruction?

Loop label is 2 instructions above
-32 bits x 2 = -64 bits = **-8 bytes**

2) What is the binary encoding of the `bltu` instruction? Feel free to use the following space for scratch work—it will not be graded. Put your final answer in hexadecimal.

31 0

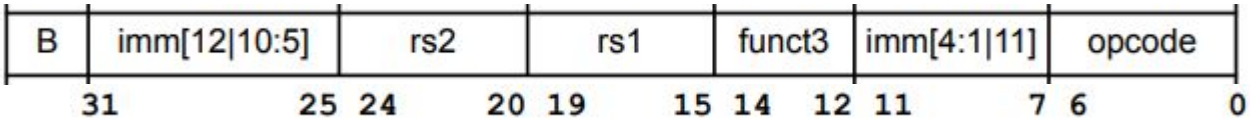
0x_____

Exam question: Summer 2018 Midterm 1 Q5

2) What is the binary encoding of the `bltu` instruction? Feel free to use the following space for scratch work—it will not be graded. Put your final answer in hexadecimal.

```
bltu t1 a0 Loop
t1 = x6 = 0b00110
a0 = x10 = 0b01010
opcode = 110 0011
funct3 = 110
imm = -8 = 0b1 1111 1111 1000

[1][111111] | 01010 | 00110 | 110 | [1100][1] | 1100011
→ 0b 1111 1110 1010 0011 0110 1100 1110 0011
→ 0x  F    E    A    3    6    C    E    3
```



`bltu rs1 rs2 label`	Branch if Less Than (Unsigned)	`PC = PC + offset`	B	`110 0011` `110`

Exam question: Summer 2020 Midterm 1 4c

(a) Leave your answers fully simplified as integers. **Do not leave powers of 2 in your answer!** Feel free to use a calculator to simplify your answer.

You want to build a mini RISC-V instruction architecture that only supports 16 registers, which allows the length of the register fields to be shortened. Assuming that you use the extra bits to extend the immediate field, what is the range of half-word instructions that can be reached using a branch instruction in this new format? [`<lower bound>`, `<upper bound>`]

i. (0.75 pt) `<lower bound>`

ii. (0.75 pt) `<upper bound>`

Exam question: Summer 2020 Midterm 1 4c

i. (0.75 pt) `<lower bound>`

-8192

$[-2^{13}, 2^{13} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

14 bits + 1 implicit 0 -> range = [-2^14, 2^14 - 2 (since imm[0] is set to 0)] -> divide each bound by 2 since half-word (2-byte) instructions -> [-2^13, 2^13 - 2]

ii. (0.75 pt) `<upper bound>`

8191

$[-2^{13}, 2^{13} - 1]$ Since we now only need 4 bits for the register fields, for Branch instructions, the immediate field will have 14 bits.

Tricky Exam Q: Fall 19 Final Q4a,b

Q4) Felix Unger must have written this RISC-V code! **(30 pts = 3*10)**

```
mystery:
      la t6, loop
loop: addi x0, x0, 0        ### nop
      lw   t5, 0(t6)
      addi t5, t5, 0x80
      sw   t5, 0(t6)
      addi a0, a0, -1
      bnez a0, loop
      ret
```

You are given the code above, and told that you can read and write to any word of memory without error. The function **`mystery`** lives somewhere in memory, but *not* at address **`0x0`**. Your system has no caches.

a) At a functional level, in seven words or fewer, what does **`mystery(x)`** do when **`x < 10`**?

_____ _____ _____ _____ _____ _____ _____

b) One by one, what are the values of **`a0`** that **`bnez`** sees with **`mystery(13)`** at every iteration? We've done the first few for you. List no more than 13; if it sees fewer than 13, write N/A for the rest.

12, 11, _____, _____, _____, _____, _____, _____, _____, _____, _____, _____, _____

Keep in mind:

- What `la` does
- a0 = x10

Tricky Exam Q: Fall 19 Final Q4a,b

```
mystery:
        la t6, loop
loop:   addi x0, x0, 0
        lw   t5, 0(t6)
        addi t5, t5, 0x80
        sw   t5, 0(t6)
        addi a0, a0, -1
        bnez a0, loop
        ret
```

Loads address of first instruction of *loop* in t6

nop

Loads 4 bytes at t6 -> loads first *loop* instruction

Adds 0x80 to instruction -> effectively increases rd by 1

Saves modified instruction at address

Decrements a0 = x10 by 1

If a0 is not 0, then go back to loop

Jump to address stored in `ra`

Tricky Exam Q: Fall 19 Final Q4a,b

```
mystery:
        la t6, loop
loop:   addi x0, x0, 0          ### nop
        lw   t5, 0(t6)
        addi t5, t5, 0x80
        sw   t5, 0(t6)
        addi a0, a0, -1
        bnez a0, loop
        ret
```

Loads 4 bytes at t6 -> loads first *loop* instruction

Adds 0x80 to instruction -> effectively increases rd by 1

Saves modified instruction at address

Decrements a0 = x10 by 1

If a0 is not 0, then go back to loop

Jump to address stored in `ra`

Answer to a) **Resets all registers at index 0 to a0 -1 to 0**

Answer to 4b

We're merrily rolling along, resetting all the registers, when we reset x10 = a0! But then "addi a0,a0,-1" makes it -1 so it actually never hits the stopping "branch equal to zero" case then! So the bnez sees -1, then -2, then -3 as the resetter continues along its merry way.

2nd

Reset register # on "nop" line	a0 before addi line	bnez sees a0 value
0	13	12
1	12	11
2	11	10
3	10	9
4	9	8
5	8	7
6	7	6
7	6	5
8	5	4
9	4	3
10	0	-1 (or 2^{32} - 1)
11	-1	-2 (or 2^{32} - 2)
12	-2	-3 (or 2^{32} - 3)

Set "nop" instruction to addi x10 x0 0 -> **a0 resets to 0!!**