CMPUT 229

# Lab #6: Stack Manipulation

CMPUT 229

## Background

Register Calling Conventions

| Register | Name | Use | Saver |
|---|---|---|---|
| x0 | zero | The constant value 0 | N.A. |
| 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 register/frame pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10-x11 | a0-a1 | Function arguments/return values | Caller |
| x12-x17 | a2-a7 | Function arguments | Caller |
| x18-x27 | s2-s11 | Saved registers | Callee |
| x28-x31 | t3-t6 | Temporaries | Caller |

# Register-Saving and Register-Restoring Instructions

The register calling conventions specifies which registers a callee function needs to save if the value in the register is modified.

```
Factorial:
        li s0, 1
Loop:
        mul s0, a0, s0
        addi a0, a0, -1
        beqz a0, End
        j Loop
End:
        mv a0, s0
        jalr x0, ra, 0
```

Factorial stores to s0, a0, x0.
According to our calling convention, the callee only needs to save s0.

In this lab, these are called **register-saving instructions**

```
Factorial:
        addi sp, sp, -4
        sw s0, 0(sp)
        li s0, 1
Loop:
        mul s0, a0, s0
        addi a0, a0, -1
        beqz a0, End
        j Loop
End:
        mv a0, s0
        lw s0, 0(sp)
        addi sp, sp, 4
        jalr x0, ra, 0
```

In this lab, these are called **register-restoring instructions**

## RISC-V Instructions as Hexadecimal

```
        Factorial:
0x01000     li s0, 1
        Loop:
0x01004     mul s0, a0, s0
0x01008     addi a0, a0, -1
0x0100C     beqz a0, End
0x01010     j Loop
        End:
0x01014     mv a0, s0
0x01018     jalr x0, ra, 0
```

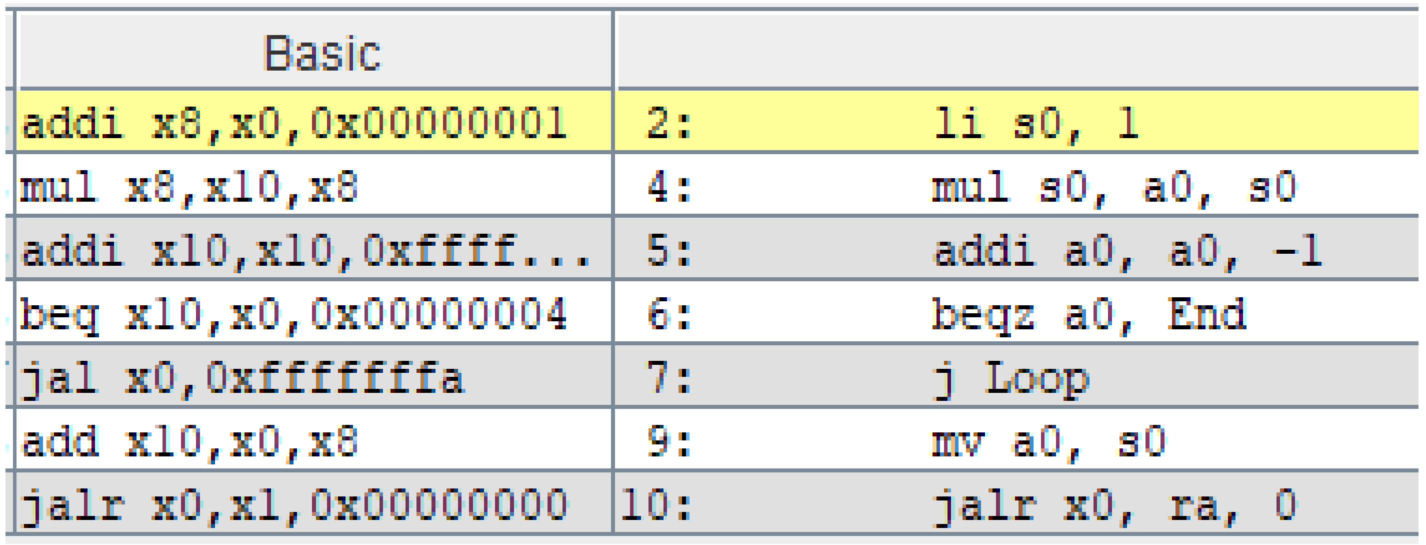
```
        Factorial:
0x01000     0x00100413
0x01004     0x02850433
0x01008     0xFFF50513
0x0100C     0x00050463
0x01010     0xFF5FF06F
0x01014     0x00800533
0x01018     0x00008067
```

Refer to the RISC-V Green Sheet and try to translate the instructions on the left to hexadecimal to confirm .

# A Note on Pseudo Instructions

When writing RISC-V code it can be helpful to use pseudo instructions that are not actually specified on your RISC-V green sheet, but are supported by RARS.

```
Factorial:
    li s0, 1
Loop:
    mul s0, a0, s0
    addi a0, a0, -1
    beqz a0, End
    j Loop
End:
    mv a0, s0
    jalr x0, ra, 0
```



| Basic | | |
|---|---|---|
| addi x8,x0,0x00000001 | 2: | li s0, 1 |
| mul x8,x10,x8 | 4: | mul s0, a0, s0 |
| addi x10,x10,0xffff... | 5: | addi a0, a0, -1 |
| beq x10,x0,0x00000004 | 6: | beqz a0, End |
| jal x0,0xfffffffa | 7: | j Loop |
| add x10,x0,x8 | 9: | mv a0, s0 |
| jalr x0,x1,0x00000000 | 10: | jalr x0, ra, 0 |

The **li s0, 1** pseudo instruction translates to **addi s0, x0, 1**

Labels don’t exist in the binary of your code. RARS uses labels like Factorial, Loop and End to simplify assembly coding. So the **beqz a0, End** pseudo instruction translates to **beq a0, x0, 4**.

Also, **j Loop** translates to **jal x0, -8**

**mv a0, s0** translates to **add a0, x0, s0**

## Input to the Lab

The input for this lab is a sequence of RISC-V instructions ending with a sentinel value (**0xFFFFFFFF**)

Factorial:

| | |
|---|---|
| **0x01000** | 0x00100413 |
| **0x01004** | 0x02850433 |
| **0x01008** | 0xFFF50513 |
| **0x0100C** | 0x00050463 |
| **0x01010** | 0xFF5FF06F |
| **0x01014** | 0x00800533 |
| **0x01018** | 0x00008067 |

Here's what the factorial function as input would look like:

[0x00100413, 0x02850433, 0xFFF50513, 0x00050463, 0xFF5FF06F, 0x00800533, 0x00008067, **0xFFFFFFFF**]

This lab will provide the address of the first instruction as input. In this example, the input to stackManipulation would be 0x01000 (a pointer to 0x00100413, the first instruction in factorial).

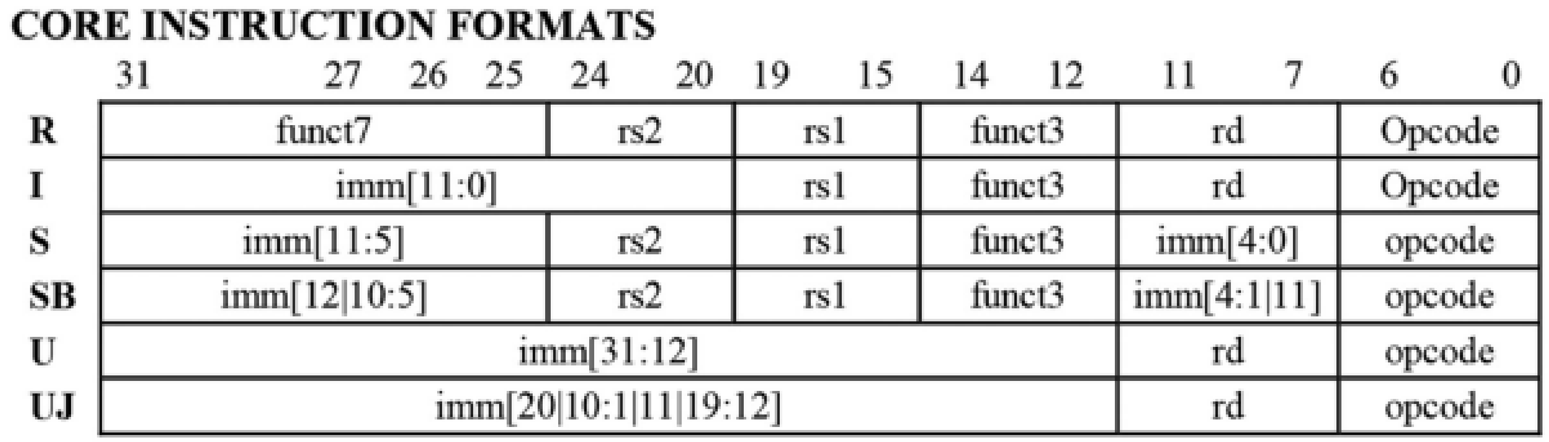
The instructions are stored in an array in memory, ending with the sentinel value (**0xFFFFFFFF)**.

A solution to this lab will parse through the instructions.

# Understanding what Hexadecimal Instructions do

Understanding the hexadecimal representation of instructions can seem intimidating. Luckily, RISC-V’s simplistic architecture makes this process more manageable.

RISC-V Instruction Formats are distinguished by their **opcode**.
    **funct3** distinguishes the instructions within that instruction format.



**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 |

For example, let’s try disassembling the first instruction (0x00100413) from the factorial function.
0x00100413 = 0000 0000 0001 0000 0**000** 0100 0**001 0011**
opcode = **001 0011,** funct3 = **000**

From the **opcode**, this is an I-type instruction. From the **funct3**, this is an addi instruction.

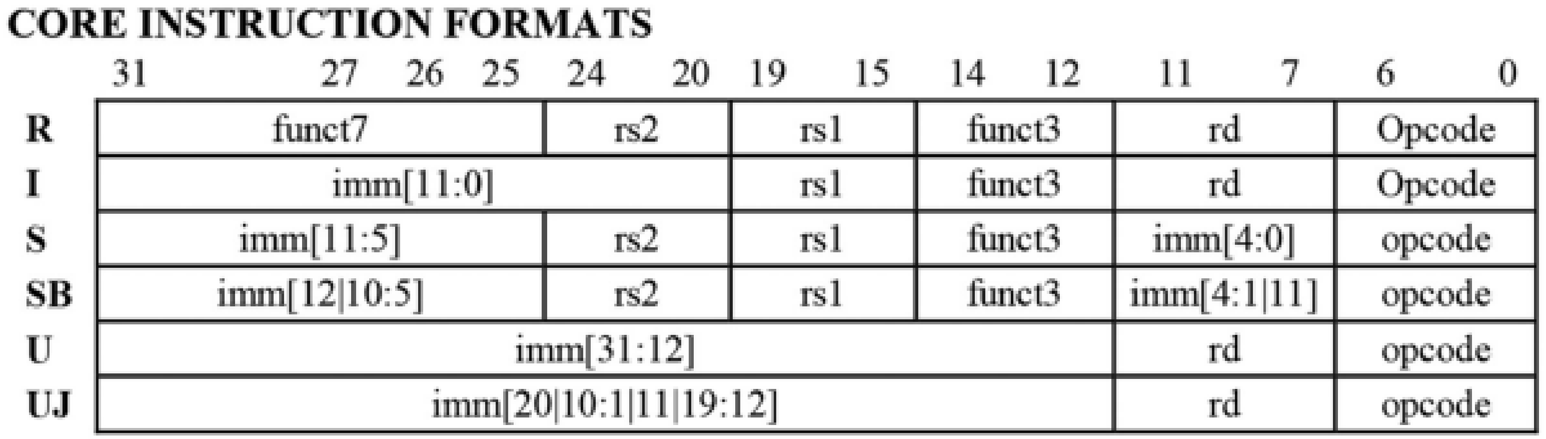
# Determining Which Instructions Write to Registers

A solution to this lab must parse through the input function and determine which registers it writes to.

RISC-V simplifies this problem through the core instruction formats.

The following instruction types all have a rd: **R, I, U, UJ.**

If an instruction is of any of the above types, then it writes to register rd.



**CORE INSTRUCTION FORMATS**

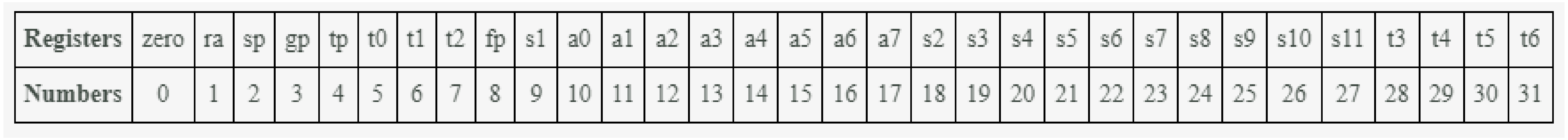
| | 31 27 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|---|---|
| **R** | funct7 | rs2 | rs1 | funct3 | rd | Opcode |
| **I** | imm[11:0] | | rs1 | funct3 | rd | Opcode |
| **S** | imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode |
| **SB** | imm[12\|10:5] | rs2 | rs1 | funct3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

The remaining instruction types do not write: **S, SB.**

**S** instructions include sb, sh, sw, sd.
**SB** instructions include beq, bne, blt, bge, bltu, bgeu.

## Register Bitmaps

A register bitmap is 32 bits where each bit represents a register. This maps bits to register numbers.



| Registers | zero | ra | sp | gp | tp | t0 | t1 | t2 | fp | s1 | a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 | s10 | s11 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Numbers | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

In this example, all bits except bit 27 are 0. In hexadecimal, this is represented as 0x0800 0000

## Using a Bitmap to Represent Register Calling Conventions

| Register | Name | Use | Saver | |
|---|---|---|---|---|
| x0 | zero | The constant value 0 | N.A. | 0 |
| x1 | ra | Return address | Caller | 0 |
| x2 | sp | Stack pointer | Callee | 1 |
| x3 | gp | Global pointer | -- | 0 |
| x4 | tp | Thread pointer | -- | 0 |
| x5-x7 | t0-t2 | Temporaries | Caller | 000 |
| x8 | s0/fp | Saved register/frame pointer | Callee | 1 |
| x9 | s1 | Saved register | Callee | 1 |
| x10-x11 | a0-a1 | Function arguments/return values | Caller | 00 |
| x12-x17 | a2-a7 | Function arguments | Caller | 000000 |
| x18-x27 | s2-s11 | Saved registers | Callee | 1111111111 |
| x28-x31 | t3-t6 | Temporaries | Caller | 0000 |

So, the register bit map representing the default RISC-V calling conventions is 0x0FFC 0304

= 0000 1111 1111 1100 0000 0011 0000 0100

= 0x0FFC 0304

CMPUT 229

## Assignment

# stackManipulation

**Description:**

This is the main function called from common.s.
Convert a RISC-V function into its stack-manipulated variation.

**Parameter:**

a0: address of the first element of an array of RISC-V instructions ending with a sentinel value (0xFFFFFFFF)

a1: register calling conventions for the RISC-V function

**Return Value:**

a0: address of the first element of a stack manipulated variation of the array of RISC-V instructions ending with a sentinel value (0xFFFFFFFF).

## findWrites

**Description:**

Find all the register writes in a RISC-V function.

**Parameter:**

a0: address of the first element of an array of RISC-V instructions ending with a sentinel value (0xFFFFFFFF)

**Return Value:**

a0: bit map of the registers written to in the RISC-V function

# findWrites Example:

| | | |
|---|---|---|
| Factorial: | | |
| li s0, 1 | store to x8 | |
| Loop: | | |
| mul s0, a0, s0 | store to x8 | bit 8 is already set |
| addi a0, a0, -1 | store to x10 | |
| beqz a0, End | doesn’t store | |
| j Loop | store to x0 | j Loop is a pseudo instruction for jal x0, -12 |
| End: | | |
| mv a0, s0 | store to x10 | bit 10 is already set |
| jalr x0, ra, 0 | store to x0 | bit 0 is already set |

Register Bit Map
0000 0000 0000 0000 0000 0101 0000 0001

**=0x0000 0501**

With the factorial function as input, findWrites should return 0x00000501 in a0.

## storeStackInstructions

**Description:**

Inserts the register-saving or register-restoring to a specified memory location. Store the sentinel value (0xFFFFFFFF) at the end of the register-saving/restoring instructions.

**Parameter:**

a0: Boolean value. If 0, store register-saving instructions. If 1, store register-restoring instructions

a1: address of the location to store register-saving/restoring instructions.

a2: bit map indicating which registers to save to the stack.

**Return Value:**

None.

# storeStackInstructions Register-Saving Example:

Let’s continue using the factorial function

```
Factorial:
	li s0, 1
Loop:
	mul s0, a0, s0
	addi a0, a0, -1
	beqz a0, End
	j Loop
End:
	mv a0, s0
	jalr x0, ra, 0
```

Inputs to storeStackInstructions:

a0: 0 (we’re storing the **register-saving instructions**)
a1: where we want to store instructions
a2: 0x0000 0100

From findWrites, we know that factorial stores to 0x0000 0501 and our default register conventions are 0x0FFC 0304.
ANDing these together we get 0x0000 0100

So, the only register that writes and we need to save is x8 (s0).

[0xFFC10113, 0x00812023, **0xFFFFFFFF]**

registerSavingInstructions:

```
	addi sp, sp, -4
	sw s0, 0(sp)
```

Remember to store the sentinel value (0xFFFFFFFF) at the end of the instructions

# storeStackInstructions Register-Restoring Example

Let's continue using the factorial function

```
Factorial:
        li s0, 1
Loop:
        mul s0, a0, s0
        addi a0, a0, -1
        beqz a0, End
        j Loop
End:
        mv a0, s0
        jalr x0, ra, 0
```

Inputs to storeStackInstructions:

a0: **1** (we're storing the **register-restoring instructions**)
a1: where we want to store instructions
a2: 0x0000 0100

[0x00012403, 0x00410113, **0xFFFFFFFF]**

```
registerRestoringInstructions:
        lw s0, 0(sp)
        addi sp, sp, 4
```

Remember to store the sentinel value (0xFFFFFFFF) at the end of the instructions

## Is that all?

Using the functions findWrites and storeStackInstructions, a naïve solution to the problem of missing register saving/restoring instructions could be implemented.

However, a solution to this lab will go further and consider the consequences of inserting instructions into already assembled code.

In this lab, we’ll focus on how inserted instructions affect jumps and accesses to the data section.

# How Inserted Instructions Affect Jumps

Inserting instructions into the binary of a function could change the behaviour of the function.
Let’s see how our factorial example is affected by inserted instructions:

```
        Factorial:
0x01000         li s0, 1
        Loop:
0x01004         mul s0, a0, s0
0x01008         addi a0, a0, -1
0x0100C         beqz a0, End (beqz a0, 4)
0x01010         j Loop (jal x0, -12)
        End:
0x01014         mv a0, s0
0x01018         jalr x0, ra, 0
```

```
0x01000  addi sp, sp, -4
0x01004  sw s0, 0(sp)
0x01008  li s0, 1
0x0100C  mul s0, a0, s0
0x01010  addi a0, a0, -1
0x01014  beqz a0, 4
0x01018  jal x0, -12
0x0101C  mv a0, s0
0x01020  lw s0, 0(sp)
0x01024  addi sp, sp, 4
0x01028  jalr x0, ra, 0
```

Notice how the address of the instructions in memory has changed. Our j Loop (pseudo instruction for jal x0, -12) has changed from **0x01010** to **0x01018**.

Luckily, jal instructions in RISC-V are relative to the current program counter. So, since the Loop instruction we want to execute is still located 12 bytes behind the j Loop instruction, the function is still correct.

# How Inserted Instructions Affect Jumps cont.

In the last slide, we found that jumps within the function body still operate the same even with our inserted instructions.

Let’s consider a variation of Factorial that calls another function to print the answer.

```
        Factorial_Print:
0x01000         li s0, 1
        Loop:
0x01004         mul s0, a0, s0
0x01008         addi a0, a0, -1
0x0100C         beqz a0, End
0x01010         j Loop
        End:
0x01014         mv a0, s0
0x01018         jal Print_Int (jal ra, 8)
0x0101C         jalr x0, ra, 0
        Print_Int:
0x01020         li a7, 1
0x01024         ecall
0x01028         jalr x0, ra, 0
```

```
0x01000    addi sp, sp, -4
0x01004    sw s0, 0(sp)
0x01008    li s0, 1
0x0100C    mul s0, a0, s0
0x01010    addi a0, a0, -1
0x01014    beqz a0, 4
0x01018    jal x0, -12
0x0101C    mv a0, s0
0x01020    jal ra, 8
0x01024    lw s0, 0(sp)
0x01028    addi sp, sp, 4
0x0102C    jalr x0, ra, 0
0x01030    li a7, 1
0x01034    ecall
0x01038    jalr x0, ra, 0
```

The jump within the function still behaves the same.

The jump outside the function **DOES NOT** behave the same.

Since the stack-restoring instructions were inserted at the end of the function, the jal offset is incorrect.

# Fixing Jumps Outside of the Function Body

With inserted instructions, jumps within the function behave the same. However, jumps outside of the function (calls to other functions) need to be corrected.

```
0x01000     addi sp, sp, -4
0x01004     sw s0, 0(sp)
0x01008     li s0, 1
0x0100C     mul s0, a0, s0
0x01010     addi a0, a0, -1
0x01014     beqz a0, 4
0x01018     jal x0, -12
0x0101C     mv a0, s0
0x01020     jal ra, 8l6
0x01024     lw s0, 0(sp)
0x01028     addi sp, sp, 4
0x0102C     jalr x0, ra, 0
0x01030     li a7, 1
0x01034     ecall
0x01038     jalr x0, ra, 0
```

To fix this forward jump outside the function body, we need to account for the inserted instructions.

Since we inserted 8 bytes at the end of the function, add 8 bytes to the jal immediate.

jal ra, 8 → jal ra, 16

Now the function behaves the same even with our compiler pass!!

If this jump was backwards, subtract the number of bytes inserted at the start of the function from the jal immediate.

# How Inserted Instructions Affect Data Accesses

Now that the solution has accounted for calls to other functions, let's also consider accesses to the data section.

Let's consider a function called *Increment* that increments a counter in memory:

```
Increment:
        la s0, counter # s0 <- pointer to counter
        lw t0, 0(s0)   # t0 <- counter
        addi t0, t0, 1 # increment counter
        sw t0, 0(s0)   # store incremented counter
        jalr x0, ra, 0 # return
```

```
Increment:
0x0040000    auipc s0, 0x0fc10
0x0040004    addi s0, s0, 0
0x0040008    lw t0, 0(s0)
0x004000C    addi t0, t0, 1
0x0040010    sw t0, 0(s0)
0x0040014    jalr x0, ra, 0
```

**la s0, counter** is a pseudo instruction

RARS translates this to the following instructions:

**auipc s0, 0x?????**
**addi s0, s0, 0x???**

auipc (add upper immediate to program counter) will do the following:

s0 <- PC + 0x**0fc10**000

Then addi will adjust the bottom 3 bytes of s0

s0 <- s0 + 0x00000**000**

To simplify things, just remember that RARS uses auipc and addi to ensure that accesses to the data section are **relative to the PC** and any location in the instructions can access any location in the data section.

# How Inserted Instructions Affect Data Accesses cont.

The load address instruction is relative to the program counter (because of the auipc instruction).
Since the compiler pass inserts register-saving instructions, the load address will occur from a different PC.

```
Increment:
0x0040000     auipc s0, 0x0fc10
0x0040004     addi s0, s0, 0
0x0040008     lw t0, 0(s0)
0x004000C     addi t0, t0, 1
0x0040010     sw t0, 0(s0)
0x0040014     jalr x0, ra, 0
```

```
Increment:
    0x00400000    addi sp, sp, -4
    0x00400004    sw s0, 0(sp)
    0x00400008    auipc s0, 0x0fc10
    0x0040000C    addi s0, s0, 0
    0x00400010    lw t0, 0(s0)
    0x00400014    addi t0, t0, 1
    0x00400018    sw t0, 0(s0)
    0x0040001C    lw s0, 0(sp)
    0x00400020    addi sp, sp, 4
    0x00400024    jalr x0, ra, 0
```

s0 <- PC + 0x0fc10000

= 0x00400008 + 0x0FC10000

s0 = 0x1001 0008

```
Counter:
      0x10010000    1
OtherData:
      0x10010004    2
      0x10010008    3
```

Since the inserted instructions affect the PC and the accesses to the data section are relative to the PC, **the access to counter is incorrect.**

# Fixing Data Accesses

From the last slide, load address instructions will behave incorrectly after inserting the stack instructions. Let's consider the same example to try to fix the load addresses.

```
Increment:
0x00400000    addi sp, sp, -4
0x00400004    sw s0, 0(sp)
0x00400008    auipc s0, 0x0fc10
0x0040000C    addi s0, s0, -8
0x00400010    lw t0, 0(s0)
0x00400014    addi t0, t0, 1
0x00400018    sw t0, 0(s0)
0x0040001C    lw s0, 0(sp)
0x00400020    addi sp, sp, 4
0x00400024    jalr x0, ra, 0
```

**s0 <- 0x1001 0008**

The inserted stack instructions changed the PC when auipc is executed. To correct this, subtract the change in the PC.

In this case, the 2 register-restoring instructions changed the PC by 8 bytes.
Let's subtract 8 bytes from the addi immediate to correct this.

addi s0, s0, 0 -> addi s0, s0, -8

```
Counter:
0x10010000    1
```

**The load address is fixed!!!**

# fixAccesses

**Description:**

Correct the accesses in a RISC-V function that may have bytes inserted at the start and end.
Adjust the immediates in jal and la instructions.

**Parameter:**

a0: address of the first element of an array of RISC-V instructions ending with a sentinel value (0xFFFFFFFF)

a1: number of bytes inserted at the start of the function.

a2: number of bytes inserted at the end of the function.

**Return Value:**

None.

# Almost There…

Lastly, consider the case that a function has multiple return statements. Where should the register-restoring instructions be inserted?

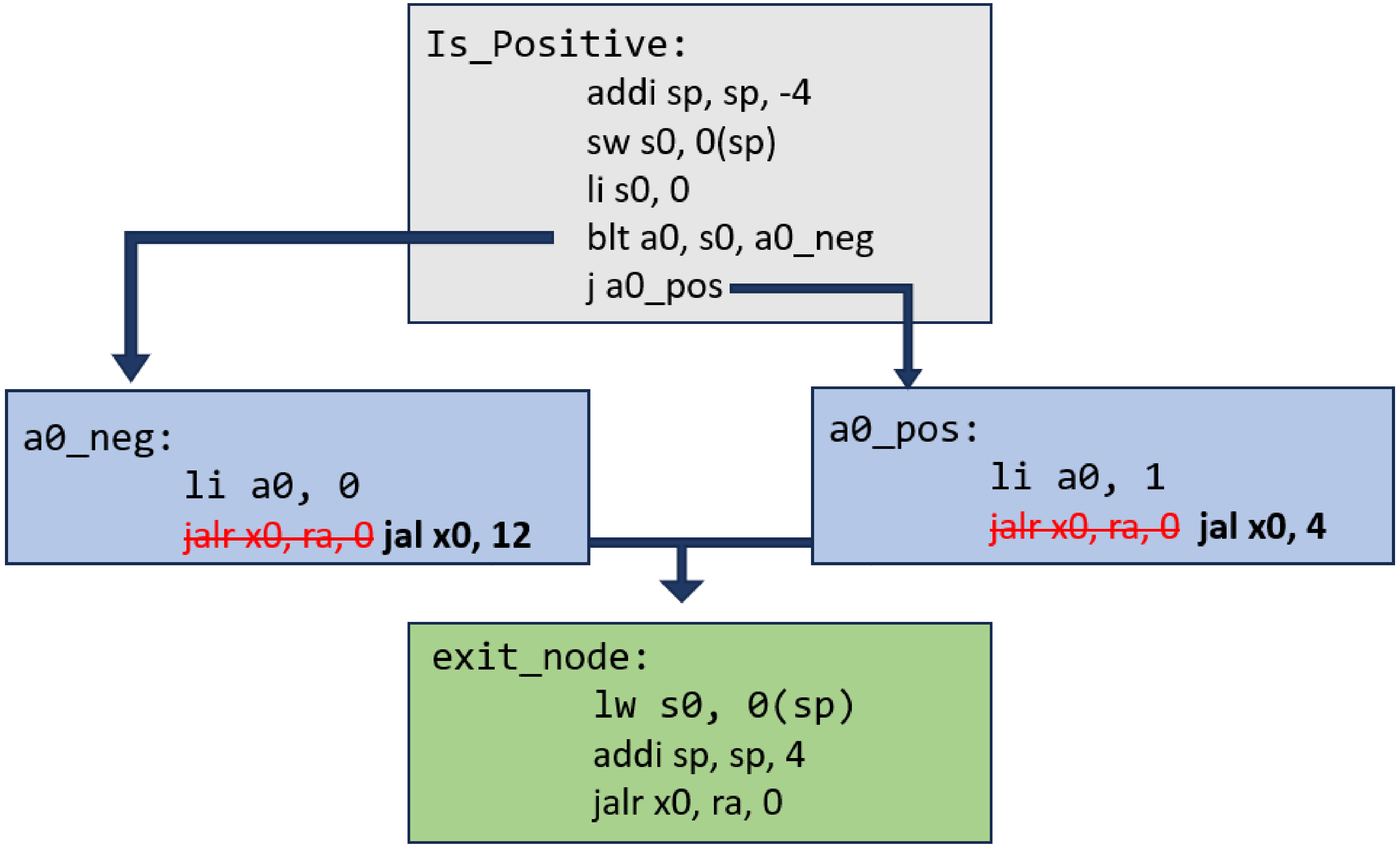
They could be inserted before every return statement.

But, this would lead to unnecessary insertions and could complicate calculating the jump immediates from the last function.

# Creating an Exit Node

Create an exit node that starts on the first register-restoring instruction ( lw x0-31, 0(sp) )

A solution should change every return instruction (even if there's just one) to jump to the exit node.



```
Is_Positive:
        addi sp, sp, -4
        sw s0, 0(sp)
        li s0, 0
        blt a0, s0, a0_neg
        j a0_pos
```

```
a0_neg:
        li a0, 0
        ~~jalr x0, ra, 0~~ jal x0, 12
```

```
a0_pos:
        li a0, 1
        ~~jalr x0, ra, 0~~ jal x0, 4
```

```
exit_node:
        lw s0, 0(sp)
        addi sp, sp, 4
        jalr x0, ra, 0
```

```
Is_Positive:
        addi sp, sp, -4
        sw s0, 0(sp)
        li s0, 0
        blt a0, s0, a0_neg
        j a0_pos
a0_neg:
        li a0, 0
        jal x0, 12
a0_pos
        li a0, 1
        jal x0, 4
exit_node:
        lw so, 0(sp)
        addi sp, sp, 4
        jalr x0, ra, 0
```

## redirectReturns

**Description:**

Converts all return statements in the function to jump to an exit node.

**Parameter:**

a0: address of the first element of an array of RISC-V instructions ending with a sentinel value (0xFFFFFFFF).

a1: pointer to exit node.

**Return Value:**

None.

# Program Flow for stackManipulation

1. Call `findWrites` to get the register bitmap of registers written to in the function.

2. Call `fixAccesses` to correct any jal or la instructions. By calling `fixAccesses` this early, it is simpler to determine whether a jal instruction jumps outside the function body.

3. Call `storeStackInstructions` to insert the register-saving instructions.

4. Copy the body of the function to space where the outputted instruction sequence will live.

5. Call `storeStackInstructions` to insert the register-restoring instructions. Remember to keep a pointer to the first instruction in the register-restoring instructions as this will be the start of the exit node.

6. Call `redirectReturns` to redirect all return statements to the exit node.

7. Insert the return statement at the end of the exit node, followed by the sentinel value (0xFFFFFFFF).

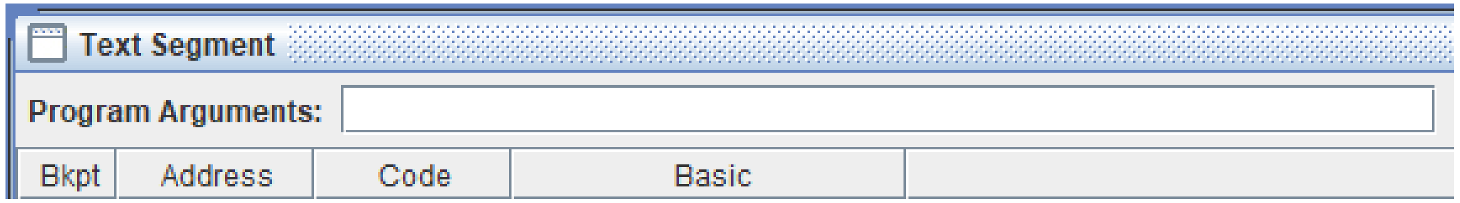
CMPUT 229

# Testing

## Program Arguments

We have provided some test inputs and outputs for you to confirm that your lab is working.

There are two program arguments for stackmanipulation.s. The first is a path to the .binary file, the second is the register saving conventions. For example, Tests/la.binary 1234FFFF



## Unit Tests

The common.s file will run unit tests on the functions in stackmanipulation.s.

The unit tests are hardcoded in the common.s file. The unit tests do not use the program arguments file at all. To see the input and expected output, check the common file.

```
1: findWrites -- [X] Great job!
2: storeStackInstructions -- [ ] Almost there!
3: fixAccesses -- [X] Great job!
4: redirectReturns -- [X] Great job!
5: stackManipulation -- [ ] Almost there!
```

## Tests Folder

In the Tests Folder, there are 4 test functions: add, la, isPos, factorial.

Each test has the following:

1. A **.s** file containing the assembly for the function
2. A **.binary** file for the function. This is what should be used in the program arguments. It was created using (rars “test.s” a dump .text Binary “test.binary”).
3. A **.correct** file. This file contains the correct output for the test and some comments on particular parts of the code.

## Creating Tests

You are encouraged to create your own tests to confirm that your lab is working.

To create your own test do the following:

1. Write the function you want to test in assembly (say my_test.s)
2. Create the binary file for your function.
   - Execute the command `rars my_test.s a dump .text Binary my_test.binary`
3. Now you can execute stackmanipulation.s with a register convention of your choice. program arguments = my_test.binary FFFF1234.

## Creating Complex Tests

If you want to test only one function from a file with many functions, you need to specify to the common file which function you want to test.

Use the dummy statement "addi x0, x0, 229" immediately before the start of the function you want to test and immediately after the end of the function.

```
… # functions not included in test

addi x0, x0, 229 # THIS TELLS THE COMMON.S FILE THAT THE
FUNCTION STARTS ON THE NEXT LINE.

factorial:
    …
    ret

addi x0, x0, 229 # THIS TELLS THE COMMON.S FILE THAT THE
FUNCTION ENDED ON THE PREVIOUS LINE.

… # more functions down here
```

Only factorial will be inputted to stackManipulation

Here is an excerpt from Tests/factorial.s which uses a complex test

CMPUT 229

# Disassembler

## What is a Disassembler?

RARS is a RISC-V Assembler that translates RISC-V Instructions to executable binary.

A Disassembler is the opposite. It converts binary to RISC-V instructions.

For this lab, we will use an Open-Source RISC-V Disassembler from: https://github.com/michaeljclark/riscv-disassembler

## How to Use the Disassembler

In the Code folder of the lab, we have created a disassembly folder.

In the disassembly folder, we have created a file called “print-instructions.c” that uses functions from the disassembler.

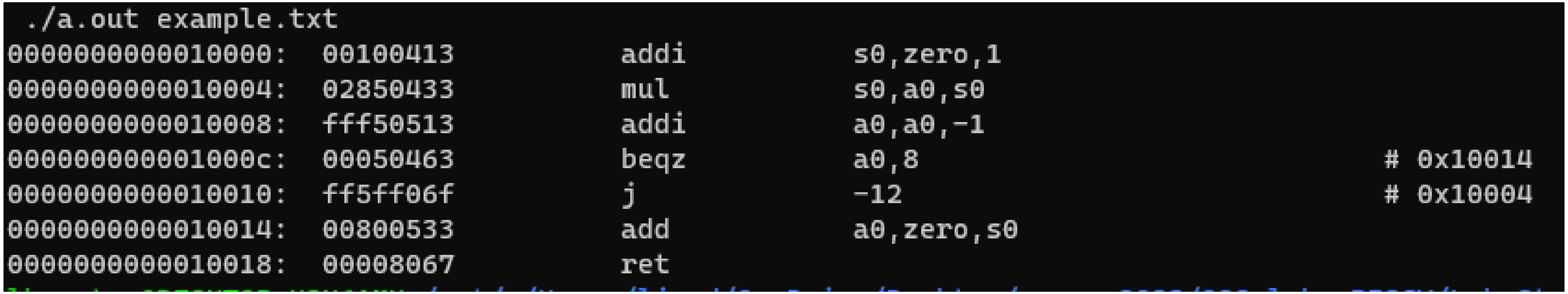
First, compiler “print-instructions.c” using a C compiler. For example, execute the command “gcc print-instructions.c” or “clang print-instructions.c”.

Next, create a text file containing hexadecimal instructions. The file should look like this:

```
0x00100413
0x02850433
0xFFF50513
0x00050463
0xFF5FF06F
0x00800533
0x00008067
```

# How to Use the Disassembler cont.

Once you have an executable for print-instructions.c (a.out) and a text file with hexadecimal instructions (example.txt), execute “./a.out example.txt”



```
 ./a.out example.txt
0000000000010000:  00100413          addi          s0,zero,1
0000000000010004:  02850433          mul           s0,a0,s0
0000000000010008:  fff50513          addi          a0,a0,-1
000000000001000c:  00050463          beqz          a0,8                        # 0x10014
0000000000010010:  ff5ff06f          j             -12                         # 0x10004
0000000000010014:  00800533          add           a0,zero,s0
0000000000010018:  00008067          ret
```

Here is the disassembled instructions from example.txt. This is a factorial function.

# Issues/Confusions with the Disassembler

While testing, we have noticed some issues with the disassembler:

- Firstly, the disassembler gets auipc immediates incorrect. Since a solution should not change auipc immediates, we recommend ignoring them in the disassembler output.
- Secondly, the disassembler translates “addi t0, t0, 0” to “mv t0, t0”. Be aware of this to avoid any confusion when working with addi instructions.
- Lastly, The disassembler will not translate the sentinel value. This is as expected.

## What to Submit?

A single file, called **stackmanipulation.s**.

Make sure the file **does not** contain a main procedure.