

CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 7: RISC-V Procedures

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

Announcements

- Please send us DSP letters as soon as possible!
- Labs 1 and 2: due Thursday (today), June 29th, 11:59 PM PT
- Homework 2 released! due Wednesday, July 5th, 11:59 PM PT
 - Short, only lecture 5 (FP) concepts
- Project 1: Due Friday (tomorrow), June 30th, 11:59 PM PT

Last Time

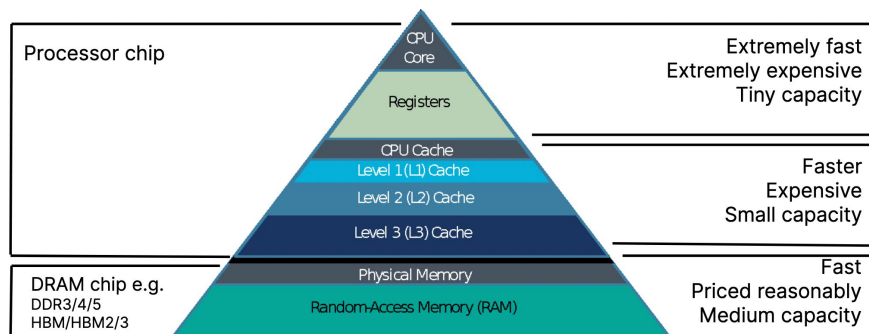
- Intro to Assembly Languages
- RISC-V Registers
- RISC-V Instructions and Instruction Types

Agenda

- Lecture 6 (Thursday)
 - Intro to Assembly Languages and RISC-V
- Lecture 7 (Today)
 - Translating C concepts into RISC-V
 - Variables
 - Arrays
 - If-Else
 - Loops
 - Functions
 - Calling Convention
- Lecture 8 (Next Wednesday)
 - Translating RISC-V to binary

Recall: RISC-V Overview

- RISC-V has two main components: the CPU, and memory
- The CPU has access to 32 registers, and generally performs operations on those registers directly.
- Memory acts as a large array of space to store data, and is accessed through load/store instructions
- Apart from x0, registers are identical in behavior, and are only differentiated by their usage. Each register has a name that is used to indicate its intended usage.
- RISC-V is designed such that any C program should be translatable into RISC-V. As such, it has analogues for C syntax.



Recall: Example from last lecture

```
addi x11,x0,0x3F5
```

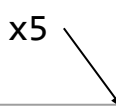
```
sw x11,0(x5)
```

```
lb x12,1(x5)
```

1) x11 → 0x0000003F5

x5

2)



Addr	0	1	2	3
Data	0xF5	<u>0x03</u>	0x00	0x00

RISC-V generally
little-endian

3) x12 → 0x000000003

General Structure of this Lecture

- Each section will have several slides:
- A snippet of C code to translate into RISC-V
- Starter RISC-V code to copy-paste directly into Venus (Note: Don't worry if the syntax there doesn't make sense)
- Lecture only: Venus demonstration
 - If reviewing this lecture, it is recommended to try working on this problem yourself before continuing
- One example of a viable solution
- Discussion on general strategy and limitations

Notes on the following demonstrations

- As noted earlier, each register has an additional name which reflects its usage.
- For the next few demonstrations, it is unnecessary to go over the entire name set. We will instead use the following register names for readability:
 - **x8=s0**: We will set s0 to point to the start of a malloc'ed buffer of length 64 KiB (so we can play around with memory).
 - **x10=a0**: At the end of each code snippet, we will print out a number (**result**) to confirm the code worked as intended. This output should be put in a0.
 - **x5=t0, x6=t1, x7=t2**: General-purpose registers

Translating C concepts into RISC-V: Variables

Variables: C code

```
int a = 6;  
int b = 3;  
int result = (1<<a)-b;  
  
printf("%d\n", result); // Should output 61
```

Variables: RISC-V Starter (Copy and Paste to try it out!)

```
#Sets up register s0 to point to an allocated buffer of 64 KiB
li a0 9
li a1 65536
ecall
mv s0 a0
#End setup

#TODO

#Print out the value stored in a0 as an integer, then exit.
#This should output 61
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit
```

Variables Solution 1: Use registers

a: stored in t0

b: stored in t1

addi t0 x0 6 # int a = 6

addi t1 x0 3 # int b = 3

addi a0 x0 1 # Output = 1

sll a0 a0 t0 # Output = 1 << a

sub a0 a0 t1 # Output = (1<<a)-b

Variables Solution 1: Use registers: Analysis

- Pro: Generally fast to use, makes more understandable code
- Con: Doesn't really work well for multi-word objects like arrays or structs
- Con: Limited to only a few variables (can't assign 50 variables at once)

Variables: RISC-V Starter

```
#Sets up register s0 to point to an allocated buffer of 64 KiB
li a0 9
li a1 65536
ecall
mv s0 a0
#End setup
```

#TODO. Ensure that we can store arbitrarily many variables

```
#Print out the value stored in a0 as an integer, then exit.
#This should output 61
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit
```

Variables: Solution 2: Use memory

a: Stored at 0(s0)

b: Stored at 4(s0)

addi t0 x0 6 # Set temp register to 6

sw t0 0(s0) # int a = 6

addi t0 x0 3 # Set temp register to 3

sw t0 4(s0) # int b = 3

addi a0 x0 1 # Output = 1

lw t0 0(s0) # Load a to temp

sll a0 a0 t0 # Output = 1 << a

lw t0 4(s0) # Load b to temp

sub a0 a0 t0 # Output = (1<<a)-b

Variables: Solution 2: Use memory: Analysis

- Pro: Allows for arbitrarily many variables in a program (as long as you have memory)
- Pro: Generalizes well to arrays and structs
- Pro: Allows use of the address operator on the variable
- Con: For a human programmer, need to keep track of offsets
- Con: Significantly slower to load/store to main memory

Variables: Register storage vs Memory storage in the real world

- When concerned with performance, we try to minimize the number of loads/stores in our code.
 - For variables that are used often, like loop variables, try to use registers.
 - For variables that are large, we end up forced to use memory.
- When writing C code, the compiler is ultimately responsible for deciding which variables go in registers.
- The **register** keyword can be used to *suggest* that a variable gets stored in a register instead of in memory.
 - Ex. `"register int i;"` declares a variable `i` as an integer and assigns it a register if possible.
 - Most modern compilers are good enough that you probably won't gain much benefit out of using this.
 - C++ actually ignores the register keyword entirely.

Translating C concepts into RISC-V: Arrays

Arrays: C code

```
int i[5] = {0,1,2,3,4};
```

```
int b = 3;
```

```
int result = i[b];
```

```
printf("%d\n", result); //Should output 4
```

Arrays: RISC-V Starter

```
#Sets up register s0 to point to an allocated buffer of 64 KiB
li a0 9
li a1 65536
ecall
mv s0 a0
#End setup

#TODO

#Print out the value stored in a0 as an integer, then exit.
#This should output 4
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit
```

Arrays: Solution

#i: Stored at 0(s0) to 20(s0)

#b: Stored in t1

```
#int i[] = {0,1,2,3,4}
```

```
addi t0 x0 0      # Set temp register to 0
```

```
sw t0 0(s0)       # i[0] = 0
```

```
addi t0 x0 1      # Set temp register to 1
```

```
sw t0 4(s0)       # i[1] = 1
```

```
addi t0 x0 2      # Set temp register to 2
```

```
sw t0 8(s0)       # i[2] = 2
```

```
addi t0 x0 3      # Set temp register to 3
```

```
sw t0 12(s0)      # i[3] = 3
```

```
addi t0 x0 4      # Set temp register to 4
```

```
sw t0 16(s0)      # i[4] = 4
```

```
addi t1 x0 3      # int b = 3
```

```
slli t0 t1 2      # sizeof(int) == 4, so multiply b by 4 to get the offset in bytes
```

```
add t0 s0 t0       # temp = (start of i array) + (offset from i)
```

```
lw a0 0(t0)       # Load i[b] to output
```

Arrays: Analysis

- If dealing with a fixed index (e.g. `arr[3]` or `arr[1]`)
 - We can just do the math when writing the program and pick the offset manually.
 - Example: `sw t0 12(s0) # i[3] = 4`
- If dealing with a variable index (e.g. `arr[i]` or `arr[foo(x)]`)
 - We need to evaluate the index first (e.g. find the value of `i` or `foo(x)`)
 - Then, we multiply the index by `sizeof(type)`.
- You can't provide three registers to a load/store
 - Example: `lw t0 t1(s0)` is not valid
 - Instead, we need to add `t1` and `s0` together first, put the result somewhere (e.g. `t2`), then do `lw t0 0(t2)`

Translating C concepts into RISC-V: If-Else

If-Else: C code

```
int a = 1; // Alternate: 0
int b;
if(a == 1) {
    b = 61;
}
else {
    b = 100;
}
int result = b;

printf("%d\n", result); // Should output 61; alternate 100
```


If-Else: RISC-V Starter

```
#Sets up register s0 to point to an allocated buffer of 64 KiB
li a0 9
li a1 65536
ecall
mv s0 a0
#End setup
```

```
#a: x5
#b: x6
```

```
addi x5 x0 1 #alt: 0
#TODO
```

```
#Print out the value stored in a0 as an integer, then exit.
#This should output 61/100
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit
```

If-Else: Solution

a: t0

b: t1

addi t0 x0 1 # int a = 1

addi t2 x0 1 # temp = 1

bne t0 t2 False # if temp != a goto False

True: addi t1 x0 61 # True case: b = 61

j End # Jump to end so we don't go through False

False: addi t1 x0 100 # False case: b=100

End: mv a0 t1 # Set output

If-Else: Analysis

- Recall: Branches simply go to the next line if the condition isn't met.
 - It's common to negate the condition of the if clause.
 - RISC-V branch philosophy: Skip this code if the condition is false.
 - C if statement philosophy: Run this code if the condition is true.
- Labels identify specific lines
 - Labels don't actually "split" the program in any way.
 - The "True" label was unneeded because it was never referenced, but was added for readability.
 - This also allows you to save a few j instructions (note that the false clause doesn't "j End" because End is written immediately after the false clause).

```
True: addi t1 x0 61
      j End
False: addi t1 x0 100
      End: mv a0 t1
```

Translating C concepts into RISC-V: Loops

Loops: C code

```
int a[100];  
int b = 0;  
for(int i = 0; i < 100; i++) {  
    a[i] = i;  
    b+= i;  
}  
int result = b;  
  
printf("%d\n", result); //Should output 4950
```

Loops: RISC-V Starter

```
#Sets up register s0 to point to an allocated buffer of 64 KiB
li a0 9
li a1 65536
ecall
mv s0 a0
#End setup
```

```
#a: 0(s0) to 400(s0)
#b: t0
#i: t1
```

```
#TODO
```

```
#Print out the value stored in a0 as an integer, then exit.
#This should output 4950
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit
```

Loops: Solution

```
# a: 0(s0) to 400(s0)
# b: t0
# i: t1
```

```
      addi t0 x0 0      # int b = 0
      addi t1 x0 0      # int i = 0
Loop: slli t2 t1 2      # temp = i*sizeof(int)
      add t2 t2 s0      # temp = a+(offset)
      sw t1 0(t2)       # Store i at a[i]
      add t0 t0 t1      # b += i
      addi t1 t1 1      # i++
      addi t2 x0 100    # temp = 100
      blt t1 t2 Loop    # If i<100, return to the start of loop
      mv a0 t0          # Output=b
```

Loops: General Pattern

```
# a: 0(s0) to 400(s0)
# b: t0
# i: t1
```

(1) Create a label

Loop:

```
addi t0 x0 0      # int b = 0
addi t1 x0 0      # int i = 0
slli t2 t1 2      # temp = i * sizeof(int)
add t2 t2 s0      # temp = a + (offset)
sw t1 0(t2)       # store i at a[i]
add t0 t0 t1      # b += i
addi t1 t1 1      # i++
addi t2 x0 100    # temp = 100
blt t1 t2 Loop    # If i < 100, return to the start of loop
mv a0 t0          # Output = b
```

(2) Set exit condition

(3) Increment variable
which exit condition is
based on

Note: These parts
might not occur in
this order.

Loops: Analysis

- Often accomplished by either:
 - A branch at the start and a jump at the end of the loop, or
 - A branch at the end of the loop
- Note that the code as written only works because we will have at least one iteration of the loop.
 - If we set the bound to 0 instead, the C code wouldn't run at all, while this RISC-V code would still do a single iteration.
- How would we write a loop that can terminate without a single iteration?

```
addi t0 x0 0
addi t1 x0 0
Loop: slli t2 t1 2
      add t2 t2 s0
      sw t1 0(t2)
      add t0 t0 t1
      addi t1 t1 1
      addi t2 x0 100
      blt t1 t2 Loop
      mv a0 t0
```

Translating C concepts into RISC-V: Functions

Recall: jal and jalr behavior (Why we link)

- Ex. `jal x1 Label` (Jump and Link)
- Meaning: Set x1 to PC+4, then jump to Label
- Links allow us to “return” to where we were before we jumped
- Often used to mimic the behavior of functions:

```
main:  addi x10 x0 5
       jal x1 foo
       ...
foo:   ...
       jr x1
```

1. We jump to foo, and store the address of the next line into x1
2. Later, foo runs `jr x1`, returning to the point after the “function call.”

- This lets foo act as a function (though there are critical differences)
- x1 is named **ra**, for “return address”

Functions: C code

```
int foo(int x) {  
    return x+3;  
}
```

```
int result = foo(7);
```

```
printf("%d\n", result); // Should output 10
```

Functions: RISC-V Starter

```
#Sets up register s0 to point to an allocated buffer of 64 KiB
li a0 9
li a1 65536
ecall
mv s0 a0
#End setup
```

```
#a: a0
```

```
#TODO
```

```
#Print out the value stored in a0 as an integer, then exit.
#This should output 10
mv a1 a0
li a0 1
ecall
li a0 10
ecall
#End print and exit
```

```
foo: #TODO
```

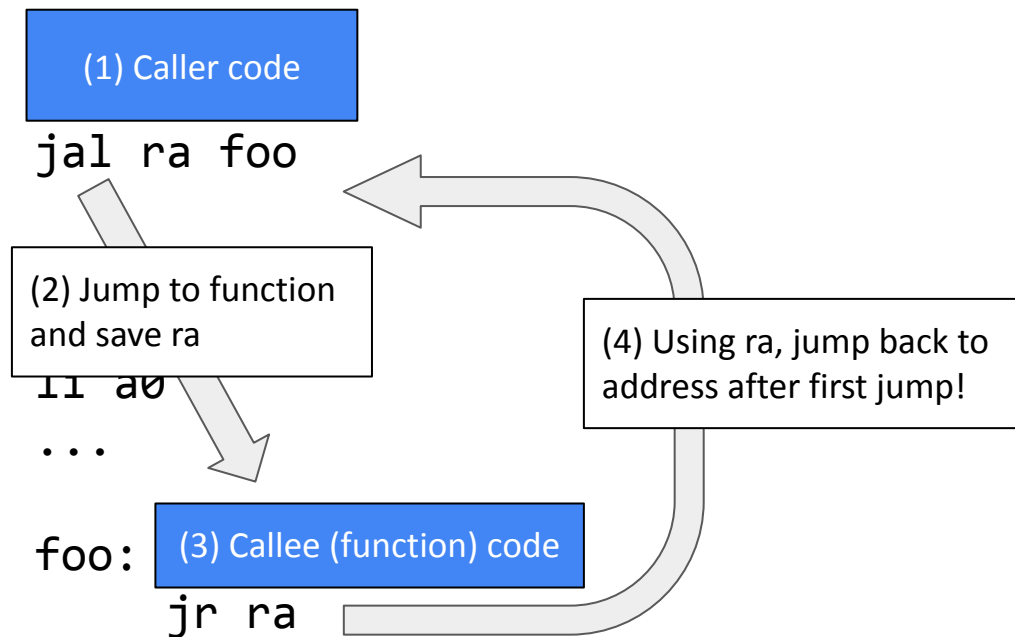
Functions: Solution

```
# result: a0
addi a0 x0 7    # Set argument for foo to 7
jal ra foo      # Jump to the start of foo, and
                 # set x1=ra to point to the line of code after this

mv a1 a0        # Code to print a0
li a0 1
...

foo: addi a0 a0 3 # return x+3
      jr ra      # Jump to the line pointed to by ra
```

Functions: General Pattern



Functions: Analysis

- Functions are effectively the same as labels
 - RISC-V makes no distinction between labels for functions and labels for loops.
- The link from a jal is designed specifically to allow you to return with a corresponding jr. This link is referred to as the "**return address**".
- In order for a function to "work", we have to specify:
 - Which registers contain the **function arguments**
 - Which register contains the **return address**
 - Which register will contain the **return value**
 - Which registers (if any) **can potentially be modified** by our function.
- We COULD define these separately for every single function, and keep track of this separately for every function we call.
 - Or we could set up some **conventions** to apply to all functions and save some headache

RISC-V Calling Convention

Registers

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0	zero	x8		x16		x24	
x1		x9		x17		x25	
x2		x10		x18		x26	
x3		x11		x19		x27	
x4		x12		x20		x28	
x5		x13		x21		x29	
x6		x14		x22		x30	
x7		x15		x23		x31	

Registers: Argument Registers

- Registers **x10-x17** are defined as **argument registers** (named **a0-a7**)
- A function expects its arguments to be stored in these registers
 - The function signature defines which registers contain which arguments
- **a0** (and less commonly **a1**) are also defined as the return values of any function
 - In other words, all functions return their result through **a0**

Registers

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0	zero	x8		x16	a6	x24	
x1		x9		x17	a7	x25	
x2		x10	a0	x18		x26	
x3		x11	a1	x19		x27	
x4		x12	a2	x20		x28	
x5		x13	a3	x21		x29	
x6		x14	a4	x22		x30	
x7		x15	a5	x23		x31	

Registers: Return Address

- Register **x1** is defined to store the **return address** (named **ra**)
- A function expects to receive the expected return address in the ra register
 - We almost always call jal or jalr using the ra register
- Pseudoinstructions exist with this in mind:
 - `jal Label` → `jal ra Label`
 - `jalr <reg>` → `jalr ra <reg> 0`
 - `ret` → `jr ra`

Registers

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0	zero	x8		x16	a6	x24	
x1	ra	x9		x17	a7	x25	
x2		x10	a0	x18		x26	
x3		x11	a1	x19		x27	
x4		x12	a2	x20		x28	
x5		x13	a3	x21		x29	
x6		x14	a4	x22		x30	
x7		x15	a5	x23		x31	

Registers: Saved and Temporary Registers

- Most of the remaining registers are divided into s registers (**s0-s11**) and t registers (**t0-t6**)
- **t** registers are **temporary**: Functions can change their values.
 - Functions do not need to set them back to their original values.
 - Use when doing intermediate calculations, or when doing work with no function calls in the middle.
- **s** registers are **saved**: Functions can use them, only if they can reset them to their original values.
 - Use when a value needs to persist through function calls, but still gets used often enough that you want it in a register (as opposed to memory)

Example: Saved and Temporary Registers

```
addi t0 x0 5      #t0 = 5
```

```
addi s0 x0 10     #s0 = 10
```

```
jal foo
```

#At this point, t0 may be any number, while s0 must be 10.

Registers: Saved and Temporary Registers

- t registers are considered temporary: Functions are allowed to change their values however they want, and do not need to set them back to their original values.
- s registers are saved: Functions may use them only if they can reset them to their original values.
- **a** registers are also considered temporary, so in a pinch, we can use a registers as additional temporaries.
 - It's generally rare to use that many temporaries, though.
- Rule of defensive programming
 - Assume that when you call a function, all the t registers WILL change to a completely random value
 - Assume that when you write a function, it WILL be called by a function that uses all s registers somehow.

Registers

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0	zero	x8	s0	x16	a6	x24	s8
x1	ra	x9	s1	x17	a7	x25	s9
x2		x10	a0	x18	s2	x26	s10
x3		x11	a1	x19	s3	x27	s11
x4		x12	a2	x20	s4	x28	t3
x5	t0	x13	a3	x21	s5	x29	t4
x6	t1	x14	a4	x22	s6	x30	t5
x7	t2	x15	a5	x23	s7	x31	t6

Stack

- We've found several things that need to be saved when we call a function.
 - **Local variables** that don't fit in any of our registers.
 - The old value of any **s register** that we want to modify (so we can restore it before the function ends)
 - The **ra** if we want to call another function in our current function (since calling jal will overwrite ra, and we need the old ra to figure out where to go when our function returns)
- All of these can get stored in the stack
- Register x2 is the **stack pointer (sp)**, and is defined to point to the bottom of the stack.
 - Note: Registers x3 and x4 are the global pointer and thread pointer, respectively (**gp** and **tp**), used for static variables and multithreading. However, their exact use is out of scope for this class.

Registers

Reg	Name	Reg	Name	Reg	Name	Reg	Name
x0	zero	x8	s0	x16	a6	x24	s8
x1	ra	x9	s1	x17	a7	x25	s9
x2	sp	x10	a0	x18	s2	x26	s10
x3	gp	x11	a1	x19	s3	x27	s11
x4	tp	x12	a2	x20	s4	x28	t3
x5	t0	x13	a3	x21	s5	x29	t4
x6	t1	x14	a4	x22	s6	x30	t5
x7	t2	x15	a5	x23	s7	x31	t6

Stack

- A function should restore the stack before returning. To be precise:
 - Data **above** `sp` at function start is considered **immutable**, and must not be modified by the function.
 - Data **below** the `sp` at function start is **mutable**; the function may modify anything in that area.
 - The **value of `sp`** must be **restored** by the end of the function. Usually done by subtracting, then later adding the same amount from the `sp` (ex. `addi sp sp -8` at the beginning of a function, `addi sp sp 8` at the end)

Stack Example

PROLOGUE:

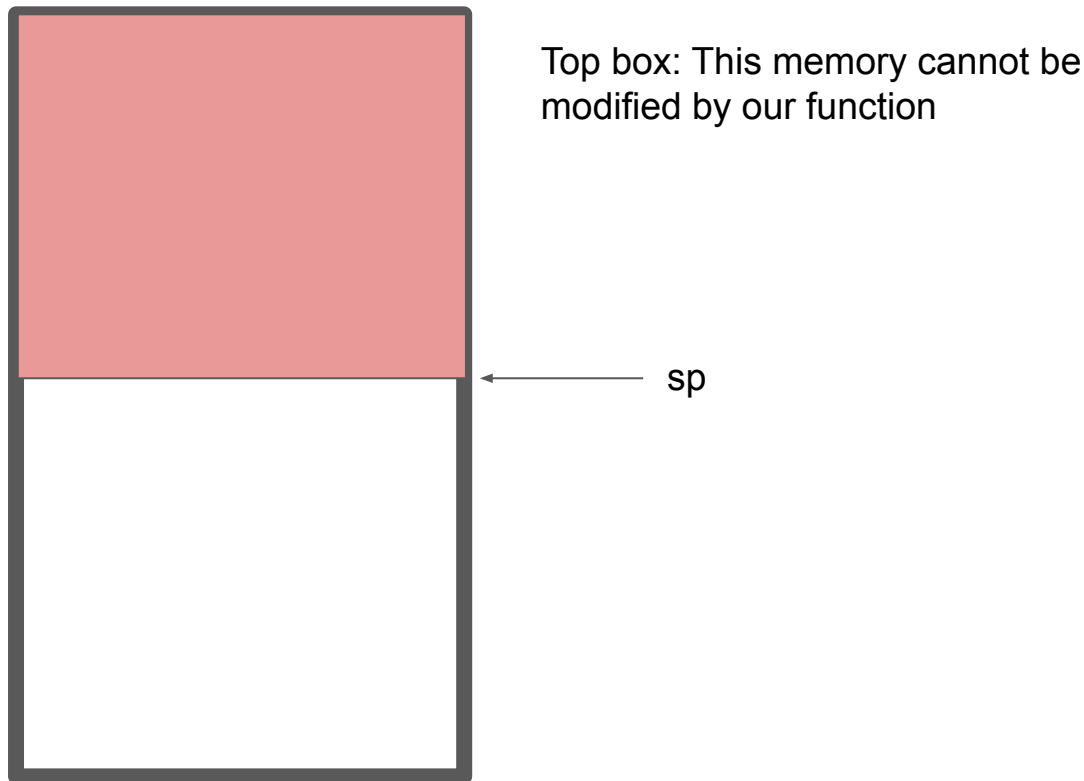
- 1: **The function is called**
- 2: Move the stack pointer down
- 3: Save any data we need to

BODY:

- 4: Run the function (which may call further functions)

EPILOGUE

- 5: Restore any registers we need to
- 6: Move the stack pointer back up
- 7: Return from the function



Stack Example

PROLOGUE:

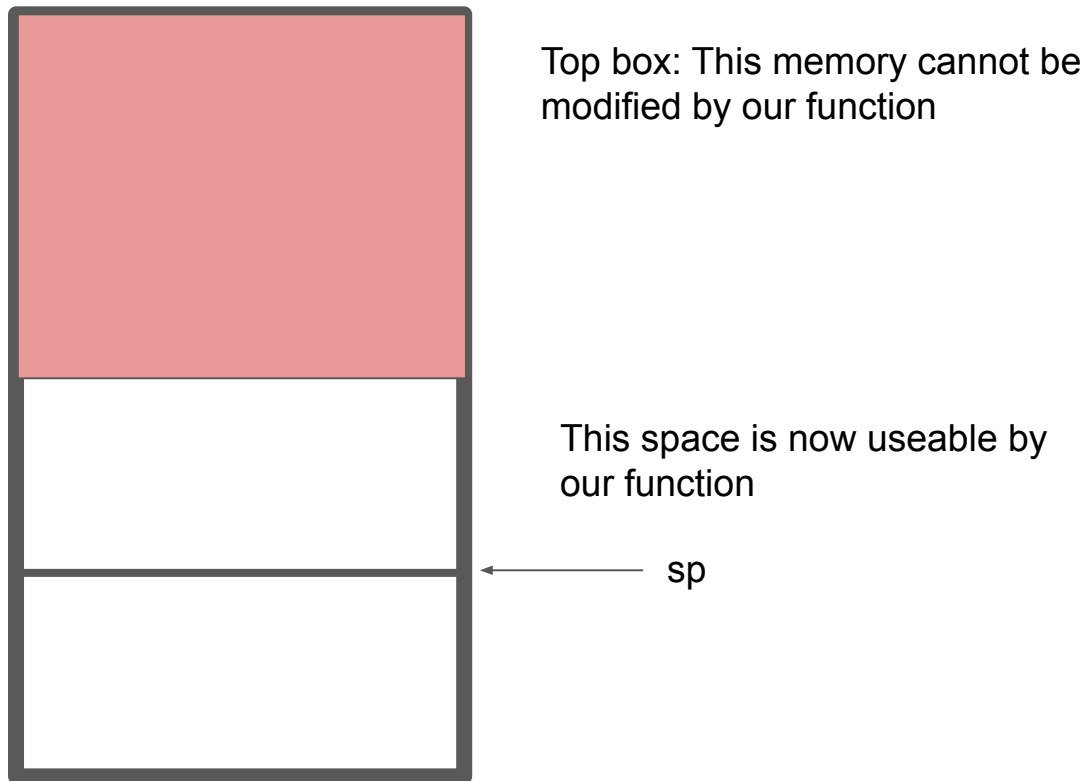
- 1: The function is called
- 2: Move the stack pointer down**
- 3: Save any data we need to

BODY:

- 4: Run the function (which may call further functions)

EPILOGUE

- 5: Restore any registers we need to
- 6: Move the stack pointer back up
- 7: Return from the function



Stack Example

PROLOGUE:

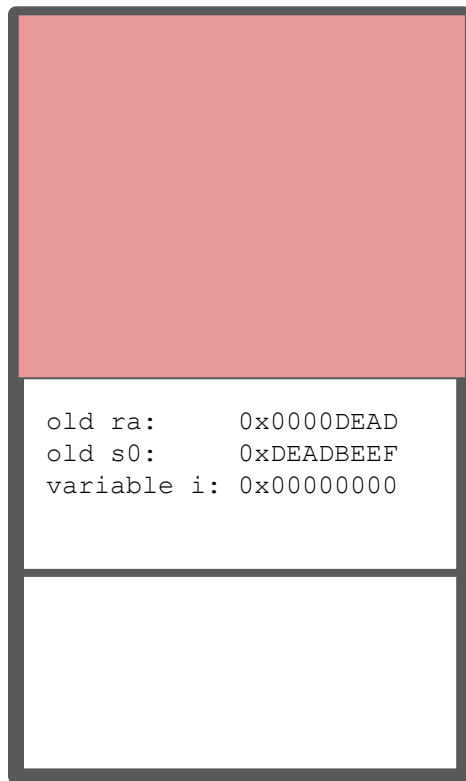
- 1: The function is called
- 2: Move the stack pointer down
- 3: Save any data we need to**

BODY:

- 4: Run the function (which may call further functions)

EPILOGUE

- 5: Restore any registers we need to
- 6: Move the stack pointer back up
- 7: Return from the function



Top box: This memory cannot be modified by our function

This space is now useable by our function

sp

Stack Example

PROLOGUE:

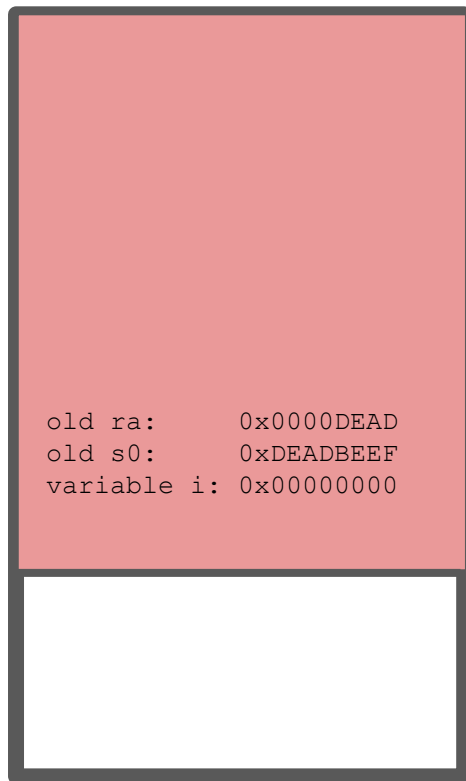
- 1: The function is called
- 2: Move the stack pointer down
- 3: Save any data we need to

BODY:

- 4: Run the function (which may call further functions)**

EPILOGUE

- 5: Restore any registers we need to
- 6: Move the stack pointer back up
- 7: Return from the function



Top box: This memory cannot be modified by our function

If we call another function then that function is not allowed to modify this block of memory

sp

Stack Example

PROLOGUE:

- 1: The function is called
- 2: Move the stack pointer down
- 3: Save any data we need to

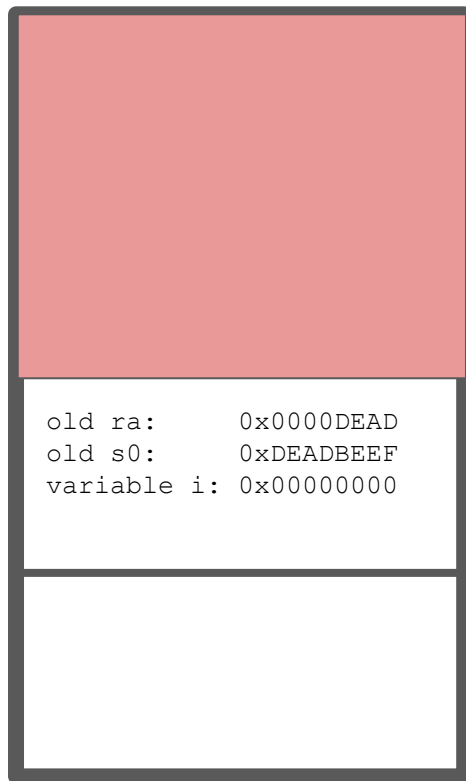
BODY:

- 4: Run the function (which may call further functions)

EPILOGUE

- 5: Restore any registers we need to**

- 6: Move the stack pointer back up
- 7: Return from the function



Top box: This memory cannot be modified by our function

At the end of the function, any data we have here should only have been changed by our code

sp

Stack Example

PROLOGUE:

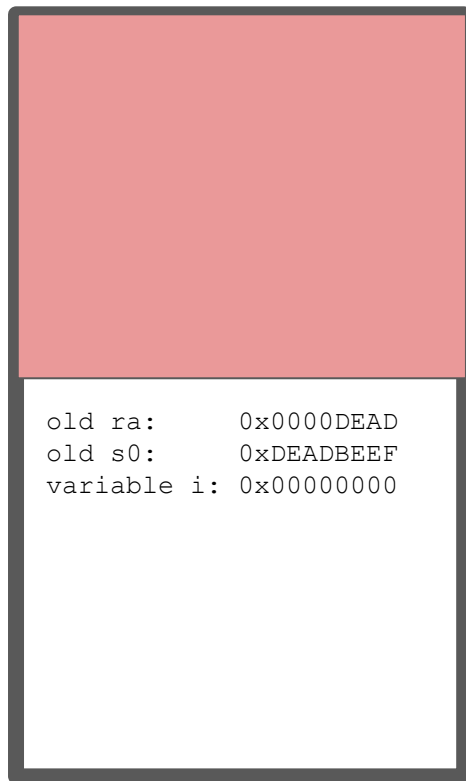
- 1: The function is called
- 2: Move the stack pointer down
- 3: Save any data we need to

BODY:

- 4: Run the function (which may call further functions)

EPILOGUE

- 5: Restore any registers we need to
- 6: Move the stack pointer back up**
- 7: Return from the function



Top box: This memory cannot be modified by our function

We move the sp back up. The data we used is still there, but will probably be overwritten by later function calls.

Stack Example

PROLOGUE:

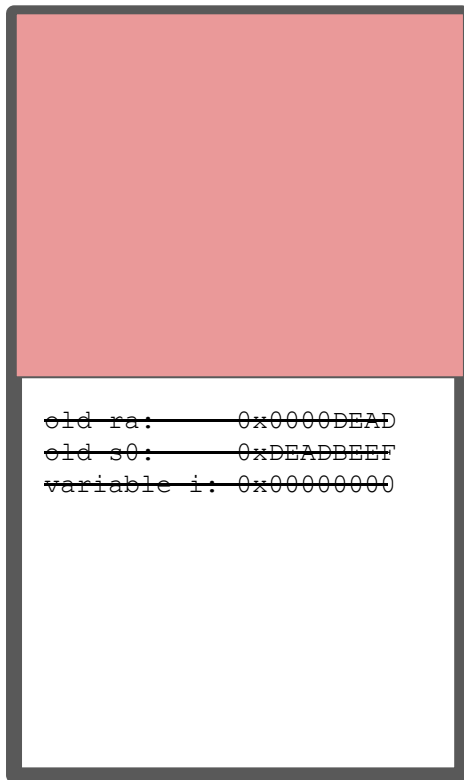
- 1: The function is called
- 2: Move the stack pointer down
- 3: Save any data we need to

BODY:

- 4: Run the function (which may call further functions)

EPILOGUE

- 5: Restore any registers we need to
- 6: Move the stack pointer back up
- 7: Return from the function**



Top box: This memory cannot be modified by our function

Stack is restored to the same state it was in before the function was called.

sp

Effectively garbage down here.

Recursive functions: C code

```
int fact(int x) {  
    if(x == 0) return 1;  
    int y = fact(x-1);  
    return x * y;  
}  
int result = fact(7);  
  
printf("%d\n", result); //Should output 5040
```

Recursive functions: RISC-V Starter

```
addi a0 x0 7  
jal fact
```

```
#Print out the value stored in a0 as an integer, then exit.
```

```
#This should output 5040
```

```
mv a1 a0
```

```
li a0 1
```

```
ecall
```

```
li a0 10
```

```
ecall
```

```
#End print and exit
```

```
fact: #TODO
```

Recursive functions: Solution

```
fact:
#x: s0
#y: a0

beq a0 x0 TailCase      # If in the tail case, handle separately
addi sp sp -8           # Begin Prologue: set aside space for 8 bytes of data
sw s0 0(sp)             # Store s0 (since we plan on using it)
sw ra 4(sp)             # Store ra (since we plan on calling another function)

mv s0 a0                # x = input (Save in s register so we keep it through the recursive call)
addi a0 a0 -1           # Set argument to x-1
jal fact                # Recursively call fact. Expect return value in y=a0
mul a0 a0 s0            # Output = x * y

lw s0 0(sp)            # Begin Epilogue: Restore old value of s0
lw ra 4(sp)            # Restore old value of ra
addi sp sp 8           # Restore stack pointer to original value
jr ra                 # Return

TailCase:
addi a0 x0 1           # In tail case, set a0 to 1, and return
jr ra
```

Recursive functions: Analysis

- As long as you follow calling convention (abbreviated CC), it is possible to make any C function.
- If you fail to follow CC, then that leads to extremely difficult bugs, because the underlying assumptions we make about how function calls work are now broken
- If your function has multiple return commands (such as through branching code), make sure that the stack gets restored regardless of which branch gets taken.
 - An easy way to ensure this is to have an epilogue label that you always jump to – see examples of this in discussion!

Summary

- C code can be systematically translated to RISC-V
 - Use these principles to help you write assembly code
- Calling convention determines how each register should be used
 - Allows us to write RISC-V code without having to track what's in every single register
- Next Tuesday is July 4! No class Mon/Tues.
- **Next Wednesday:** Lecture 8, RISC-V Instruction Format
 - How instructions are encoded as bits that can be understood by circuits