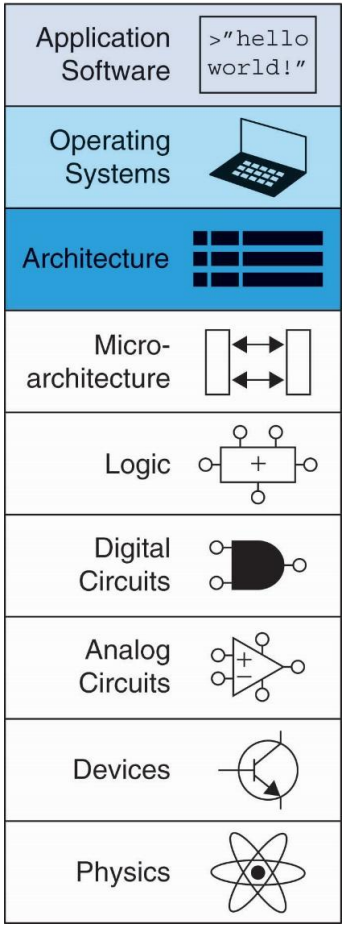
COSC175 (Systems I): Computer Organization & Design

Professor Lillian Pentecost
Fall 2024

Warm-Up October 31 (spooky!!!)

- Where we were
  - RISC-V assembly programs to make direct use of our HW architecture
- Where we are going
  - How is memory organized, and how is data managed across function calls?
- Logistics, Reminders
  - TA help 7-9PM on Sundays, Tuesdays, Thursdays in C107
  - LP Office hours M 9-10:30AM, Th 2:30-4PM
  - Weekly Exercises due Friday 5PM
  - Lab 5 Report due November 4 10PM



Programming

- **High-level languages:**
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- **High-level constructs:** loops, conditional statements, arrays, function calls
- **First, introduce instructions that support these:**
  - Logical operations
  - Shift instructions
  - Multiplication & division
  - Branches & Jumps

RISC-V Register Set

Name	Register Number	Usage
`zero`	x0	Constant value 0
`ra`	x1	Return address
`sp`	x2	Stack pointer
`gp`	x3	Global pointer
`tp`	x4	Thread pointer
`t0-2`	x5-7	Temporaries
`s0/fp`	x8	Saved register / Frame pointer
`s1`	x9	Saved register
`a0-1`	x10-11	Function arguments / return values
`a2-7`	x12-17	Function arguments
`s2-11`	x18-27	Saved registers
`t3-6`	x28-31	Temporaries

The Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

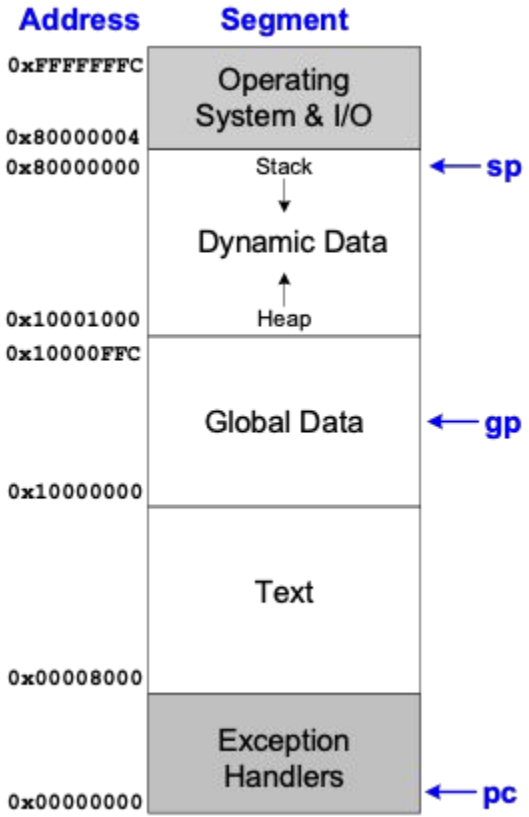
What is Stored in Memory?

- **Instructions** (also called *text*)
- **Data**
  - **Global/static:** allocated before program begins
  - **Dynamic:** allocated within program

- How **big** is memory?
  - At most 2^{32} = 4 gigabytes (4 GB)
  - From address 0x00000000 to 0xFFFFFFFF

Example RISC-V Memory Map

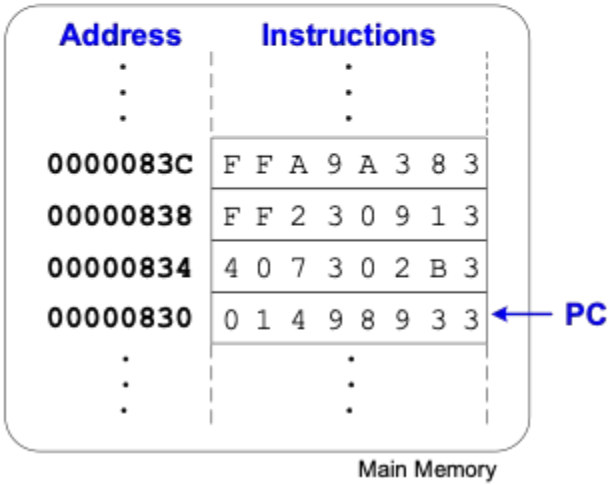
- **Instructions** (also called *text*)
- **Data**
  - **Global/static:** allocated before program begins
  - **Dynamic:** allocated within program
- Special registers to track important addresses:
  - **pc: tracks the memory address of the current instruction**
  - sp
  - gp



Example RISC-V Memory Map

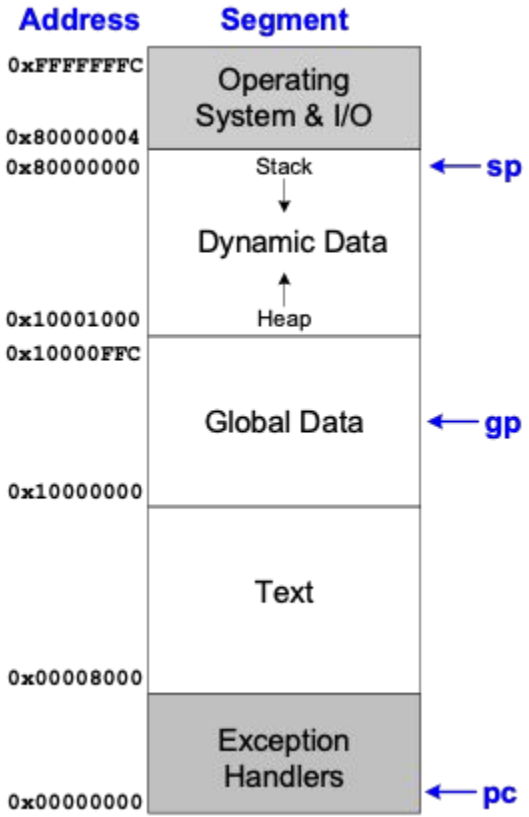
- **Instructions** (also called *text*)
- **Data**
  - **Global/static:** allocated before program begins
  - **Dynamic:** allocated within program
- Special registers to track important addresses:
  - **pc: tracks the memory address of the current instruction**
  - sp
  - gp

Assembly Code	Machine Code
`add s2, s3, s4`	`0x01498933`
`sub t0, t1, t2`	`0x407302B3`
`addi s2, t1, -14`	`0xFF230913`
`lw t2, -6(s3)`	`0xFFA9A383`



Example RISC-V Memory Map

- **Instructions** (also called *text*)
- **Data**
  - **Global/static:** allocated before program begins
  - **Dynamic:** allocated within program
- Special registers to track important addresses:
  - **pc: tracks the memory address of the current instruction**
  - sp: address of top of the stack



Example Program: C Code

```
int f, g, y; // global variables

int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b-1));
}

void main() {
  f = 2;
  g = 3;
  y = func(f,g);

  return;
}
```

Example Program: RISC-V Assembly

Address	Machine Code		RISC-V Assembly Code
10144:	ff010113	func:	addi sp,sp,-16
10148:	00112623		sw ra,12(sp)
1014c:	00812423		sw s0,8(sp)
10150:	00050413		mv s0,a0
10154:	00a58533		add a0,a1,a0
10158:	0005da63		bgez a1,1016c <func+0x28>
1015c:	00c12083		lw ra,12(sp)
10160:	00812403		lw s0,8(sp)
10164:	01010113		addi sp,sp,16
10168:	00008067		ret
1016c:	fff58593		addi a1,a1,-1
10170:	00040513		mv a0,s0
10174:	fd1ff0ef		jal ra,10144 <func>
10178:	00850533		add a0,a0,s0
1017c:	fe1ff06f		j 1015c <func+0x18>

Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Simple Function Call

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

`void` means that `simple` doesn't return a value

Simple Function Call

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  simple      # call
0x00000304         add  s0, s1, s2
...                ...

0x0000051c simple: jr   ra          # return
```

`jal simple`:

`ra` = PC + 4 (0x00000304)

jumps to `simple` label (PC = 0x0000051c)

`jr ra`:

PC = `ra` (0x00000304)

`void` means that `simple` doesn't return a value

Function Calling Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

RISC-V Function Calling Conventions

- **Call Function:** jump and link (`jal func`)
- **Return** from function: jump register (`jr ra`)
- **Arguments**: `a0` – `a7`
- **Return value**: `a0`

Input Arguments & Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

Input Arguments & Return Value

RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2  # argument 0 = 2
addi a1, zero, 3  # argument 1 = 3
addi a2, zero, 4  # argument 2 = 4
addi a3, zero, 5  # argument 3 = 5
jal  diffofsums   # call function
add  s7, a0, zero # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1   # t0 = f + g
add  t1, a2, a3   # t1 = h + i
sub  s3, t0, t1   # result = (f + g) - (h + i)
add  a0, s3, zero # put return value in a0
jr   ra           # return to caller
```

Input Arguments & Return Value

RISC-V assembly code

```
# s3 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  s3, t0, t1    # result = (f + g) - (h + i)
  add  a0, s3, zero  # put return value in a0
  jr   ra            # return to caller
```

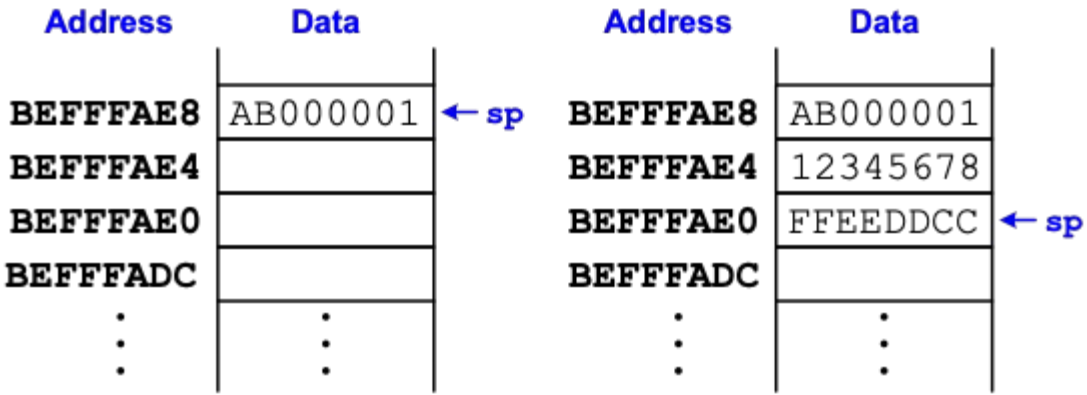
- `diffofsums` **overwrote** 3 registers: `t0`, `t1`, `s3`
- `diffofsums` can use *stack* to temporarily store registers

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands*:** uses more memory when more space needed
- ***Contracts*:** uses less memory when the space is no longer needed

The Stack

- Grows from **higher** to **lower** memory addresses
- Stack pointer: `sp` points to top of the stack



How Functions use the Stack

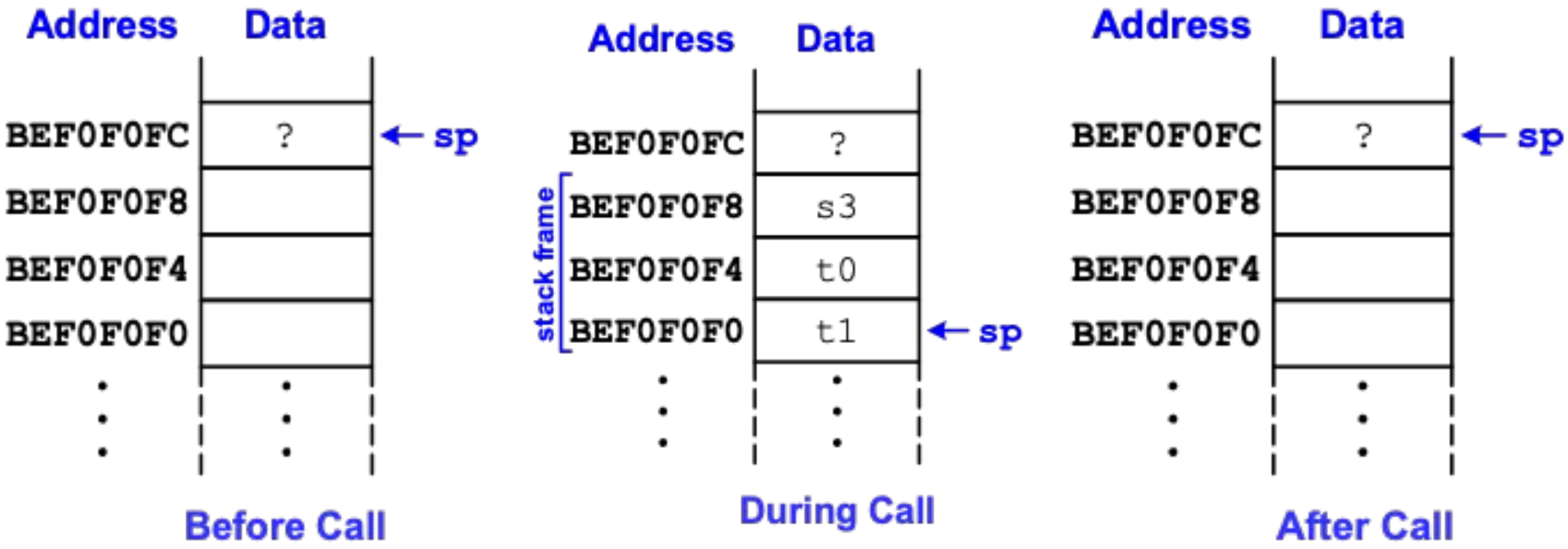
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0`, `t1`, `s3`

```
# RISC-V assembly
# s3 = result
diffofsums:
  add  t0, a0, a1   # t0 = f + g
  add  t1, a2, a3   # t1 = h + i
  sub  s3, t0, t1   # result = (f + g) - (h + i)
  add  a0, s3, zero # put return value in a0
  jr   ra           # return to caller
```

Storing Register Values on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -12       # make space on stack to
                         # store three registers
  sw   s3, 8(sp)         # save s3 on stack
  sw   t0, 4(sp)         # save t0 on stack
  sw   t1, 0(sp)         # save t1 on stack
  add  t0, a0, a1        # t0 = f + g
  add  t1, a2, a3        # t1 = h + i
  sub  s3, t0, t1        # result = (f + g) - (h + i)
  add  a0, s3, zero      # put return value in a0
  lw   s3, 8(sp)         # restore s3 from stack
  lw   t0, 4(sp)         # restore t0 from stack
  lw   t1, 0(sp)         # restore t1 from stack
  addi sp, sp, 12        # deallocate stack space
  jr   ra                # return to caller
```

The Stack During `diffofsums` Call



Preserved Registers

- What is the callee ***responsible*** for preserving / protecting the value of? AKA ***calling conventions***

Preserved *Callee-Saved*	**Nonpreserved** *Caller-Saved*
`s0-s11`	`t0-t6`
`sp`	`a0-a7`
`ra`	
stack above `sp`	stack below `sp`

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4          # make space on stack to
                           # store one register
  sw   s3, 0(sp)           # save s3 on stack
  add  t0, a0, a1          # t0 = f + g
  add  t1, a2, a3          # t1 = h + i
  sub  s3, t0, t1          # result = (f + g) - (h + i)
  add  a0, s3, zero        # put return value in a0
  lw   s3, 0(sp)           # restore $s3 from stack
  addi sp, sp, 4           # deallocate stack space
  jr   ra                  # return to caller
```

Optimized `diffofsums`

```
# what if we just avoided using a saved register, tho?

# a0 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  a0, t0, t1    # result = (f + g) - (h + i)
  jr   ra            # return to caller
```

Non-Leaf Function Calls

Non-leaf function:

a function that calls another function

```
func1:
  addi sp, sp, -4    # make space on stack
  sw   ra, 0(sp)     # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)     # restore ra from stack
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

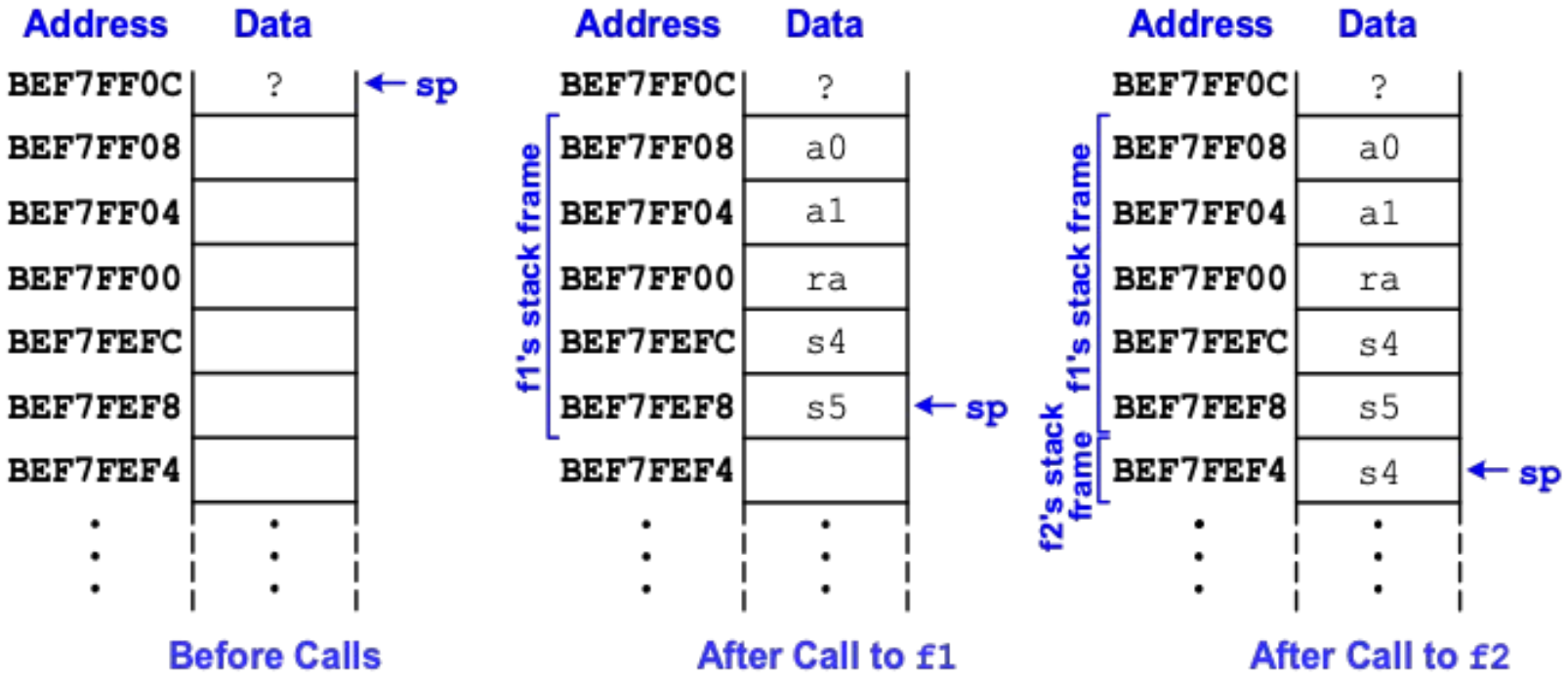
Must preserve **`ra`** before function call.

Non-Leaf Function Call Example

```
# f1 (non-leaf function) uses s4-s5 and needs a0-a1 after call to f2
f1:
  addi sp, sp, -20    # make space on stack for 5 words
  sw   a0, 16(sp)
  sw   a1, 12(sp)
  sw   ra, 8(sp)      # save ra on stack
  sw   s4, 4(sp)
  sw   s5, 0(sp)
  jal  func2
  ...
  lw   ra, 8(sp)      # restore ra (and other regs) from stack
  ...
  addi sp, sp, 20     # deallocate stack space
  jr   ra             # return to caller

# f2 (leaf function) only uses s4 and calls no functions
f2:
  addi sp, sp, -4     # make space on stack for 1 word
  sw   s4, 0(sp)
  ...
  lw   s4, 0(sp)
  addi sp, sp, 4      # deallocate stack space
  jr   ra             # return to caller
```

Stack during Function Calls



Function Call Summary

- **Caller**
  - Save any needed registers (`ra`, maybe `t0-t6/a0-a7`)
  - Put arguments in `a0-a7`
  - Call function: `jal callee`
  - Look for result in `a0`
  - Restore any saved registers

- **Callee**
  - Save registers that might be disturbed `(s0-s11)`
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`

Recursive Function Example

- Function that **calls itself**
- When converting to assembly code:
  - In the first pass, treat recursive calls as if it's calling a different function and ignore overwritten registers.
  - Then save/restore registers on stack as needed.

  - Use the textbook Code Example 6.28 as a basis for corresponding Lab 5 question to try it yourself, paying careful attention as you simulate and step through!!

Back to an Example Program: C Code

```
int f, g, y; // global variables

int func(int a, int b) {
  if (b < 0)
    return (a + b);
  else
    return(a + func(a, b-1));
}

void main() {
  f = 2;
  g = 3;
  y = func(f,g);

  return;
}
```

Example Program: RISC-V Assembly

Address	Machine Code		RISC-V Assembly Code
10144:	ff010113	func:	addi sp,sp,-16
10148:	00112623		sw ra,12(sp)
1014c:	00812423		sw s0,8(sp)
10150:	00050413		mv s0,a0
10154:	00a58533		add a0,a1,a0
10158:	0005da63		bgez a1,1016c <func+0x28>
1015c:	00c12083		lw ra,12(sp)
10160:	00812403		lw s0,8(sp)
10164:	01010113		addi sp,sp,16
10168:	00008067		ret
1016c:	fff58593		addi a1,a1,-1
10170:	00040513		mv a0,s0
10174:	fd1ff0ef		jal ra,10144 <func>
10178:	00850533		add a0,a0,s0
1017c:	fe1ff06f		j 1015c <func+0x18>

Maintain **4-word alignment** of **`sp`** (for compatibility with RV128I) though only space for 2 words needed.

Pseudoinstructions:

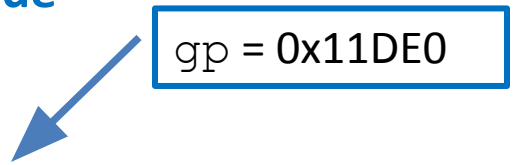
`mv`: `addi a0, s0, 0`

`ret` (return): `jr ra`

Example Program: RISC-V Assembly

Address Machine Code RISC-V Assembly Code

```
10180: ff010113 main: addi sp,sp,-16
10184: 00112623       sw   ra,12(sp)
10188: 00200713       li   a4,2
1018c: c4e1a823       sw   a4,-944(gp) # 11a30 <f>
10190: 00300713       li   a4,3
10194: c4e1aa23       sw   a4,-940(gp) # 11a34 <g>
10198: 00300593       li   a1,3
1019c: 00200513       li   a0,2
101a0: fa5ff0ef       jal  ra,10144 <func>
101a4: c4a1ac23       sw   a0,-936(gp) # 11a38 <y>
101a8: 00c12083       lw   ra,12(sp)
101ac: 01010113       addi sp,sp,16
101b0: 00008067       ret
```



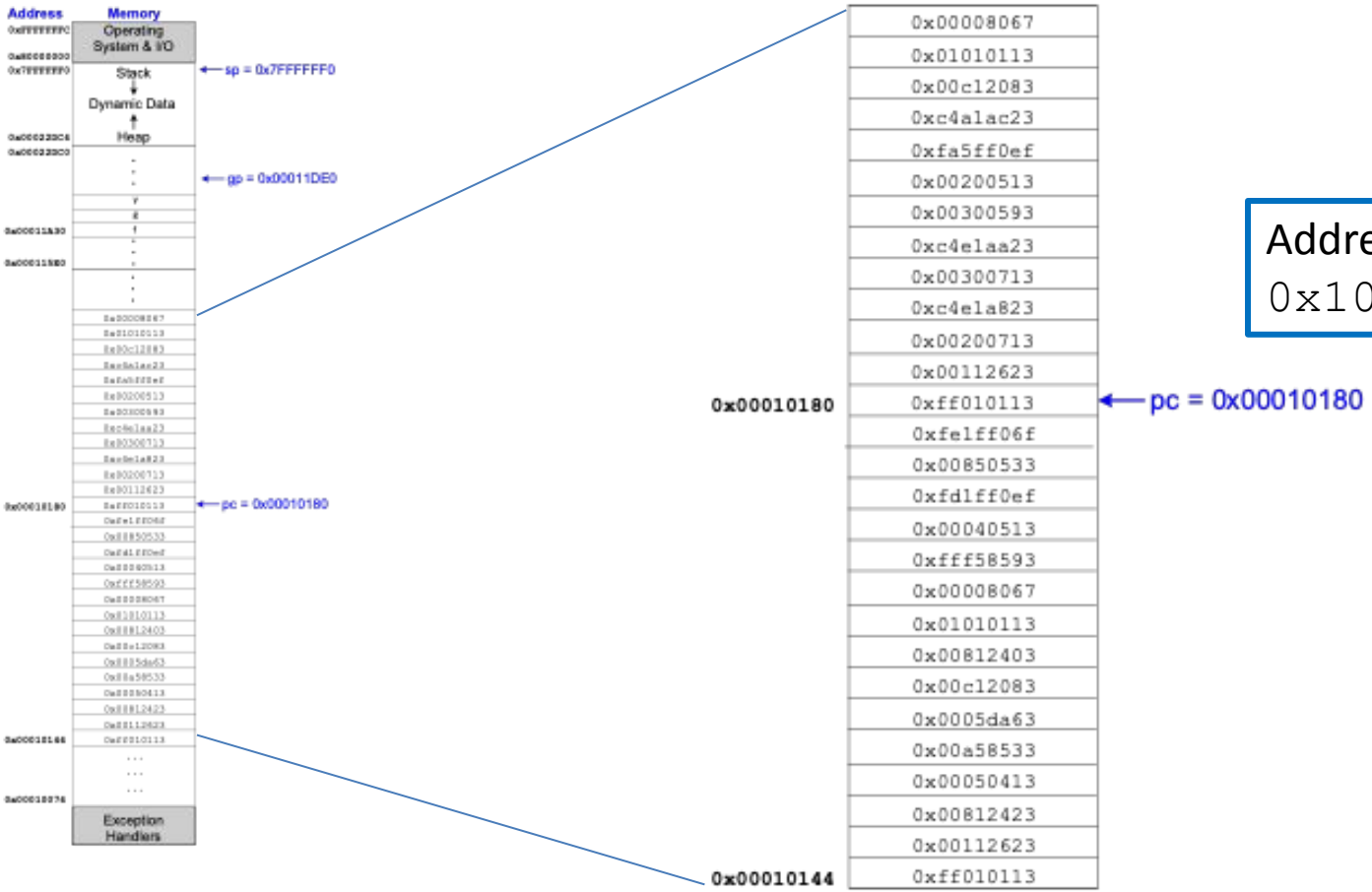
Put 2 and 3 in `f` and `g` (and argument registers) and call `func`. Then put result in `y` and return.

Example Program: Symbol Table

```
Address                    Size     Symbol Name
00010074 l     d  .text   00000000 .text
000115e0 l     d  .data   00000000 .data
00010144 g     F .text    0000003c func
00010180 g     F .text    00000034 main
00011a30 g     O .bss     00000004 f
00011a34 g     O .bss     00000004 g
00011a38 g     O .bss     00000004 y
```

- text segment: address 0x10074
- data segment: address 0x115e0
- func function: address 0x10144 (size 0x3c bytes)
- main function: address 0x10180 (size 0x34 bytes)
- f: address 0x11a30 (size 0x4 bytes)
- g: address 0x11a34 (size 0x4 bytes)
- y: address 0x11a38 (size 0x4 bytes)

Example Program in Memory



Address of `main`: `0x10180`

Wrap-Up October 31 (spooky!!)

- Coming up next!
  - You need to take time to PRACTICE
- Logistics, Reminders
  - TA help 7-9PM on Sundays, Tuesdays, Thursdays in C107
  - LP Office hours M 9-10:30AM, Th 2:30-4PM
  - Weekly Exercises due Friday 5PM
  - Lab 5 Report due November 4 10PM
- FEEDBACK
  - https://forms.gle/5Aafcm3iJthX78jx6