Great Ideas in Computer Architecture

More RISC-V, RISC-V Functions

Instructor: Sean Farhat

Review of Last Lecture (1/2)

- RISC Design Principles
  - Smaller is faster: 32 registers, fewer instructions
  - Keep it simple: rigid syntax
- RISC-V Registers: `s0-s11`, `t0-t6`, `x0`
  - No data types, just raw bits, operations determine how they are interpreted
- Memory is byte-addressed
  - no types → no automatic pointer arithmetic

Review of Last Lecture (2/2)

- RISC-V Instructions
  - Arithmetic: `add, sub, addi, mult, div`
  - Data Transfer: `lw`, `sw`, `lb, sb, lbu`
  - Branching: `beq, bne, bge, blt, jal, j, jalr, jr;`
  - Bitwise: `and, or, xor, andi, ori, xori`
  - Shifting: `sll, srl, sra, slli, srli, srai`

`i` = "immediate" (constant integer)

RISC-V Agenda

- **Sign Extension Practice**
- Pseudo-Instructions
- C to RISC-V Practice
- Functions in Assembly
- Function Calling Conventions

Sign in Two's Complement

- How do we know if a binary two's complement number is negative?

Sign in Two's Complement

- How do we know if a binary two's complement number is negative?

Binary:	0b10000010	0b01111111	0b11110000
Hex:	0x82	0x7F	0xF0

Sign in Two's Complement

- How do we know if a binary two's complement number is negative?
  - Look at the most significant bit!

Binary:	0b10000010	0b01111111	0b11110000
Hex:	0x82	0x7F	0xF0
	Negative	Positive	Negative

Sign Extension

- If we want to take an 8-bit two's complement number and make it a 9-bit number, how would we do so?

0b0000 0010 (+2) -> 0b0 0000 0010 (2)

0b1111 1110 (-2) -> 0b1 1111 1110 (-2)

We replicate the most significant bit!

Arithmetic Sign Extension

When doing math, immediate values are sign extended

addi t0, x0,-1 == addi t0, x0, 0xFFF

t0 -> [-1] -> [0xFFFFFFFF]

addi t0, x0, 0x0FF   t0 -> [0x000000FF]

addi t0, x0, 0xF77   t0 -> [0xFFFFFF77]

Why are we only using 12 bits for the immediate in these instructions? Find out next lecture!

Loading Sign Extension

- For assembly, this happens when we pull data out of memory
- Byte in memory:

  0b1111 1110 (-2)
- load byte -> Register contents:

  0b XXXX XXXX XXXX XXXX XXXX XXXX 1111 1110

  What do we do with the X values?

Loading Sign Extension

- For assembly, this happens when we pull data out of memory
- Byte in memory:
  0b1111 1110 (-2)
- load byte -> Register contents:
  0b **1111 1111 1111 1111 1111 1111** 1111 1110

  What do we do with the X values? **Sign extend!**

Loading Sign Extension

Normal (signed) loads sign extend the most significant bit

Memory: 0b1000 1111

Load Byte -> $\boxed{\text{0b1111 1111 1111 1111 1111 1111 1000 1111}}$

Memory: 0b0000 1111

Load Byte -> $\boxed{\text{0b0000 0000 0000 0000 0000 0000 0000 1111}}$

Loading Sign Extension

Offset loads also sign extend:

Memory = [0x00008011] (address in s0)

Assume system is little endian

lb t0, 0(s0) -> loading 0b00010001

0b0000 0000 0000 0000 0000 0000 0001 0001

lb t0, 1(s0) -> loading 0b10000000

0b1111 1111 1111 1111 1111 1111 1000 0000

Loading Sign Extension

Unsigned loads do not sign extend, but rather fill with zeros:

Memory = [0x00008011] (address in s0)

Assume system is little endian

lbu t0, 1(s0) -> loading 0b10000000

0b0000 0000 0000 0000 0000 0000 1000 0000

RISC-V Agenda

- Sign Extension Practice
- **Pseudo-Instructions**
- C to RISC-V Practice
- Functions in Assembly
- Function Calling Conventions

Assembly Instructions

- A low-level programming language where the program instructions match a particular architecture's operations
  - Code can be compiled into different assembly languages, but an assembly language can only run on hardware that supports it
- But sometimes, for the programmer's benefit, it's useful to have additional instructions that aren't really implemented by the hardware
  - Instead translated into real instructions
- Example: `mv   dst,reg1`
  translates into `addi dst,reg1,0`

More Pseudo-Instructions

- Load Immediate (`li`)
  - `li dst, imm`
  - Loads 32-bit immediate into `dst`
  - utilizes: `addi, lui`
- Load Address (`la`)
  - `la dst, label`
  - Loads address of specified label into `dst`
  - translates to: `auipc dst, <offset to label>`
- No Operation (nop)
  - `nop`
  - Do nothing
  - translates to: `addi x0, x0, 0`

Pseudo-Instructions are useful

- Even the `j` instruction is actually a pseudo-Instruction
  - We will see what this converts to later this lecture
- Pseudo-Instructions are core to writing RISC assembly code and you will see them in any RISC assembly code you read

Full list of RISC-V supported pseudo instructions is on the greensheet

RISC-V Agenda

- Sign Extension Practice
- Pseudo-Instructions
- **C to RISC-V Practice**
- Functions in Assembly
- Function Calling Conventions

C to RISC-V Practice

- Let's put our all of our new RISC-V knowledge to use in an example: "Fast String Copy"
- C code is as follows:

```
/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != '\0') ;
```

- What do we know about its structure?
  - Single `while` loop
  - Exit condition is an equality test

C to RISC-V Practice

- Start with code skeleton:

```
# copy String p to q
# p→s0, q→s1 (char* pointers)
Loop:                           # t0 = *p
                                # *q = t0
                                # p = p + 1
                                # q = q + 1
                                # if *p==0, go to Exit
      j Loop                    # go to Loop
Exit:
```

C to RISC-V Practice

- Finished code:

```
# copy String p to q
# p→s0, q→s1 (char* pointers)
Loop: lb   t0,0(s0)       # t0 = *p
      sb   t0,0(s1)       # *q = t0
      addi s0,s0,1        # p = p + 1
      addi s1,s1,1        # q = q + 1
      beq  t0,x0,Exit     # if *p==0, go to Exit
      j Loop              # go to Loop
Exit: # N chars in p => N*6 instructions
```

C to RISC-V Practice

- Finished code:

What if lb sign extends?

```
# copy String p to q
# p→s0, q→s1 (char* pointers)
Loop: lb   t0,0(s0)
      sb   t0,0(s1)
      addi s0,s0,1
      addi s1,s1,1
      beq  t0,x0,Exit
      j Loop
Exit: # N chars in p => N*6 instructions
```

Not a problem because sb only writes a single byte.

(The sign extension is ignored)

C to RISC-V Practice

- Alternate code using bne:

```
# copy String p to q
# p→s0, q→s1 (char* pointers)
Loop: lb   t0,0(s0)          # t0 = *p
      sb   t0,0(s1)          # *q = t0
      addi s0,s0,1           # p = p + 1
      addi s1,s1,1           # q = q + 1
      bne  t0,x0,Loop        # if *p==0, go to Loop
Exit: # N chars in p => N*5 instructions
```

Question: What C code properly fills in the following blank?

```
do {i--; } while(_______________);
```

```
Loop:                  # i→s0, j→s1
addi  s0,s0,-1         # i = i - 1
slti  t0,s1,2          # t0 = (j < 2)
bne   t0,x0 Loop         # goto Loop if t0!=0
slt   t0,s1,s0         # t0 = (j < i)
bne   t0,x0 ,Loop      # goto Loop if t0!=0
```

(A) `j < 2 || j < i`

(B) `j ≥ 2 && j < i`

(C) `j < 2 || j ≥ i`

(D) `j < 2 && j ≥ i`

Question: What C code properly fills in the following blank?

```
do {i--; } while(______________);
```

```
Loop:                   # i→s0, j→s1
addi   s0,s0,-1         # i = i - 1
slti   t0,s1,2          # t0 = (j < 2)
bne    t0,x0 Loop         # goto Loop if t0!=0
slt    t0,s1,s0         # t0 = (j < i)
bne    t0,x0 ,Loop      # goto Loop if t0!=0
```

(A) `j < 2 || j < i`

(B) `j ≥ 2 && j < i`

(C) `j < 2 || j ≥ i`

(D) `j < 2 && j ≥ i`

RISC-V Agenda

- Sign Extension Practice
- Pseudo-Instructions
- C to RISC-V Practice
- **Functions in Assembly**
- Function Calling Conventions

Six Steps of Calling a Function

1. Put *arguments* in a place where the function can access them
2. Transfer control to the function
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place and "cleans up"
6. Control is returned to you

1 and 5: Where should we put the arguments and return values?

- Registers way faster than memory, so use them whenever possible

- `a0-a7`: eight *argument* registers to pass parameters
- `a0-a1`: two *argument* registers also used to return values
  - Order of arguments matters
  - If need extra space, use memory (the stack!)

Example: function in assembly

```
void main(void){
      a = 3;
      b = a+1;
      a = add(a, b);
   ...
}

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

```
main:
      addi a0, x0, 3
      addi a1, a0, 1
      jal ra, add
      ...

add:
      add a0, a0, a1
      jr ra
```

More Registers

- `a0-a7`: eight *argument* registers to pass parameters
- `a0-a1`: two registers to return values
- `sp`: “stack pointer”
  - Holds the current memory address of the “bottom” of the stack

2 and 6: How do we Transfer Control?

- Jump (`j`)
  - `j label`
- Jump and Link (`jal`)
  - `jal dst label`
- Jump and Link Register (`jalr`)
  - `jalr dst src imm`

Used to invoke a function

- "and Link": Saves the location of instruction in a register before jumping
- Jump Register (`jr`)
  - `jr src`

Used to return from a function (src = ra)

- `ra` = *return address* register, used to save <u>where a function is called from</u> so we can get back

Function Call Example

```
... sum(a,b); ...          /* a→s0,b→s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

RISC-V

address (decimal)

```
1000  addi  a0,s0,0        # x = a
1004  addi  a1,s1,0        # y = b
1008  addi  ra,x0,1016     # ra=1016
1012  j     sum            # jump to sum
1016
...
2000  sum: add a0,a0,a1
2004  jr    ra              # return
```

Would we know this before compiling?

Otherwise we don't know where we came from

Function Call Example

```
... sum(a,b); ...            /* a→s0,b→s1 */

int sum(int x, int y) {
  return x+y;
}
```

C

RISC-V

address (decimal)

```
1000  addi a0,s0,0       # x = a
1004  addi a1,s1,0       # y = b
1008  jal  ra sum        # ra=1012, goto sum
1012
...

2000  sum: add v0,a0,a1
2004  jr   ra             # return
```

J is a pseudo-instruction explained

- `jal` syntax: `jal dst label`
- You supply the register used to link
  - When calling a function you use ra
- What happens if you specify x0?
  - `jal x0 label`
  - `x0` always contains 0, so attempts to write to it do nothing
  - So jal x0 label is just jumping without linking
- `j label` is a pseudo-instruction for `jal x0 label`
  - Similarly `jr` is a pseudo-instruction for `jalr` following the same idea

Review Question

`ret` is a pseudocode instruction that can be used to return from a function. Which real instruction(s) would you use to create ret?

Description: PC `= R[1]`

[A] `jal x0,ra`

[B] `beq x0,x0,ra`

[C] `jalr x0,ra,0`

[D] `j ra`

[E] `jalr ra,ra,0`

Review Question

`ret` is a pseudocode instruction that can be used to return from a function. Which real instruction(s) would you use to create ret?

Description: PC `= R[1]`

[A] Invalid Syntax

[B] Invalid Syntax

`[c] jalr x0,ra,0`

[D] Invalid Syntax

[E] Would return properly though it would overwrite ra after doing so

3: Local storage for variables

- Stack pointer (sp) holds the address of the bottom of the stack
  - Decrement it (recall stack grows downwards)
  - Then use store word to write to a variable
  - To “clean up”, just increment the stack pointer

```
# store t0 to the stack
addi sp, sp, -4
sw   t0, 0(sp)
```

RISC-V Agenda

- Sign Extension Practice
- Pseudo-Instructions
- C to RISC-V Practice
- Functions in Assembly
- **Function Calling Conventions**

Six Steps of Calling a Function

✓ Put *arguments* in a place where the function can access them

✓ Transfer control to the function

✓ The function will acquire any (local) storage resources it needs

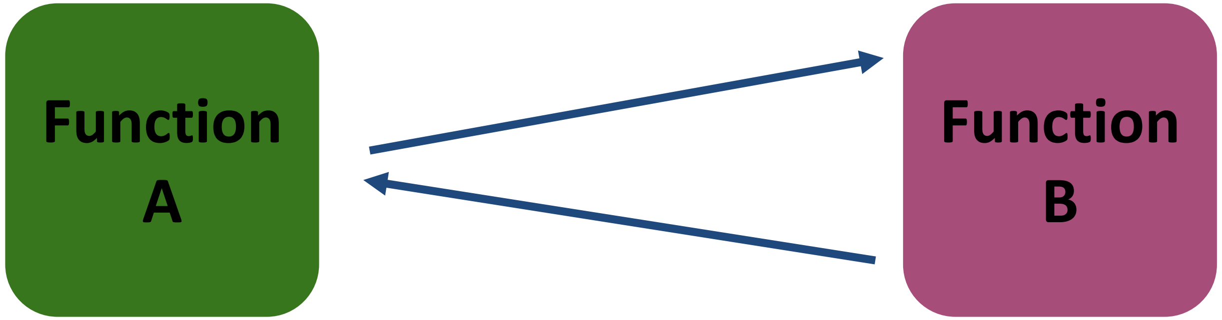
4. The function performs its desired task

✓ The function puts *return value* in an accessible place and "cleans up"

✓ Control is returned to you

Which registers can we use?

- Problem: how does the function know which registers are safe to use?



Function A may have been using t0 when it called Function B!

Example: sumSquare

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y; }
```

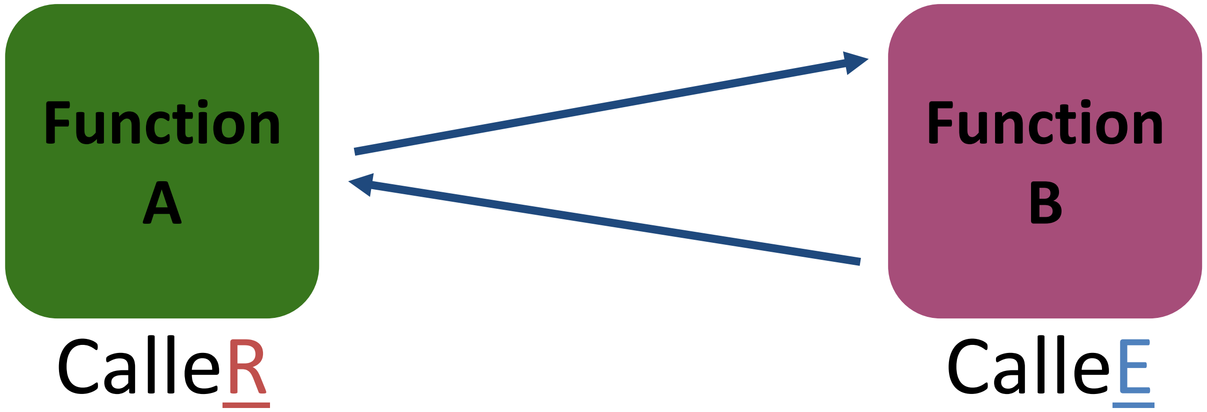
- What do we need to save?
  - Call to `mult` will overwrite `ra`, so save it
  - Reusing `a1` to pass 2nd argument to `mult`, but need current value (`y`) later, so save `a1`

Calling Conventions

- **CalleR**: the calling function
- **CalleE**: the function being called

- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may have changed

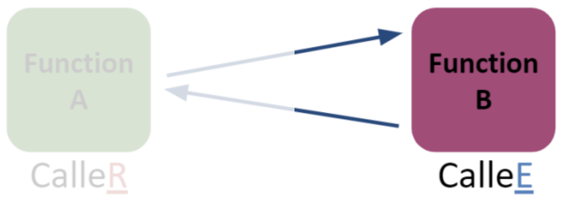
Caller and Callee




```
void functionA(void) {
     // do stuff
     functionB(void);
     // do more stuff
    return;}
```

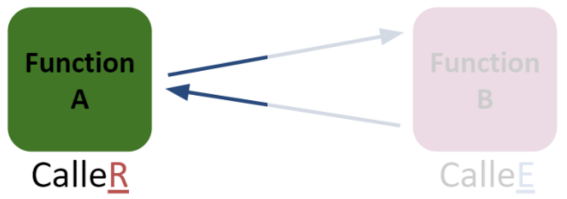
Saved Registers (Callee Saved)

- These registers are expected to be the same before and after a function call
  - If calleE uses them, it must restore values before returning
  - This means save the old values, use the registers, then reload the old values back into the registers
- `s0-s11` (*saved* registers)
- `sp` (stack pointer)
  - If not in same place, the caller won't be able to properly access its own stack variables



Volatile Registers (Caller Saved)

- These registers can be freely changed by the calleE
  - If calleR needs them, it must save those values before making a procedure call
- `t0-t6` (*temporary* registers)
- `a0-a7` (return address and arguments)
- `ra` (return address)
  - These will change if calleE invokes another function (nested function means calleE is also a calleR)



Register Conventions

Each register is one of two types:

- Caller saved
  - The callee function can use them freely
    (if needed, the caller had to save them before invoking and will restore them afterwards)
- Callee saved
  - The callee function must save them before modifying them, and restore them before returning
    (avoid using them at all, and no need to save)

This is a contract agreed upon by all functions

Calling Convention on Greencard

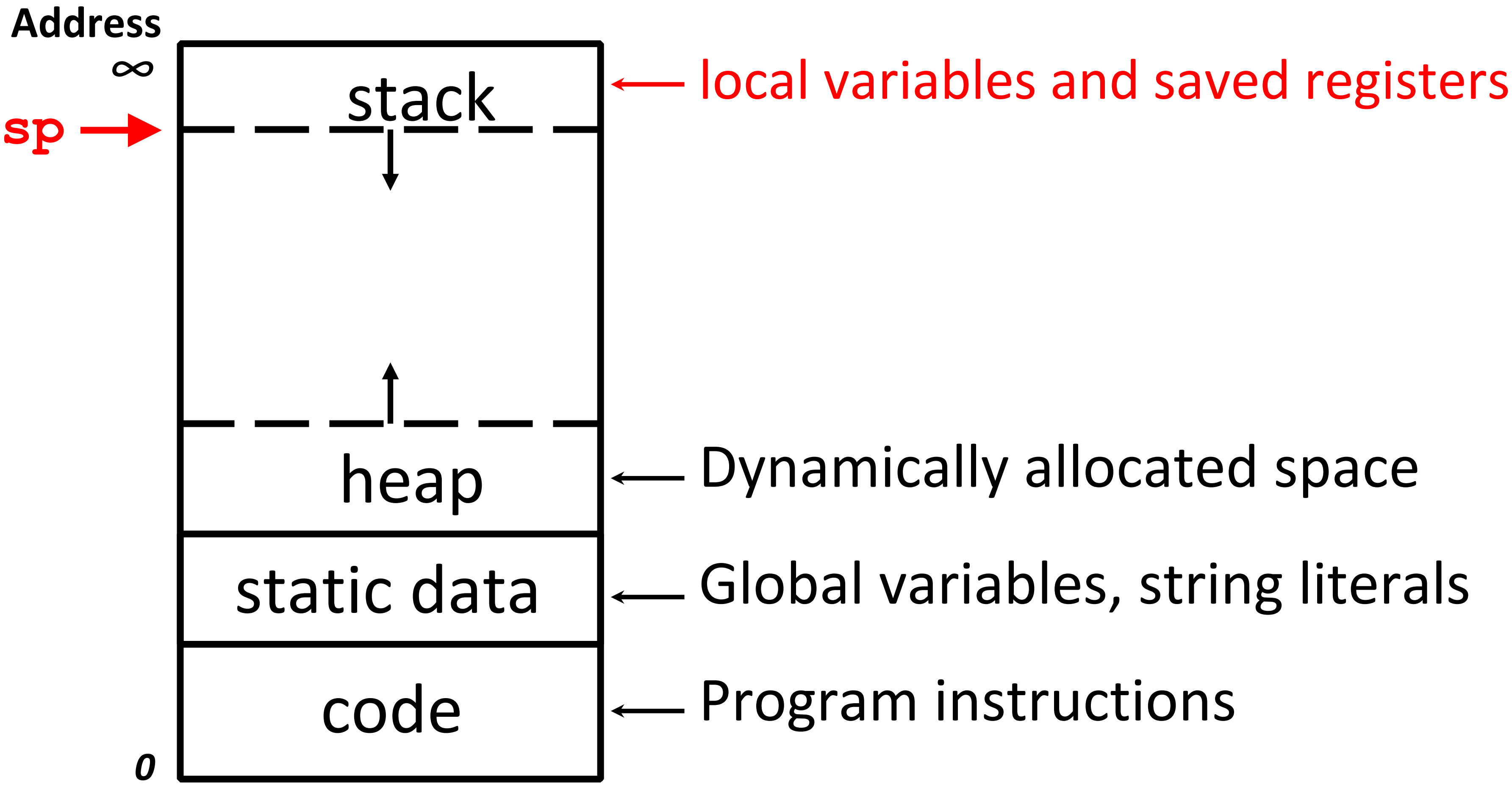
REGISTER NAME, USE, CALLING CONVENTION

④

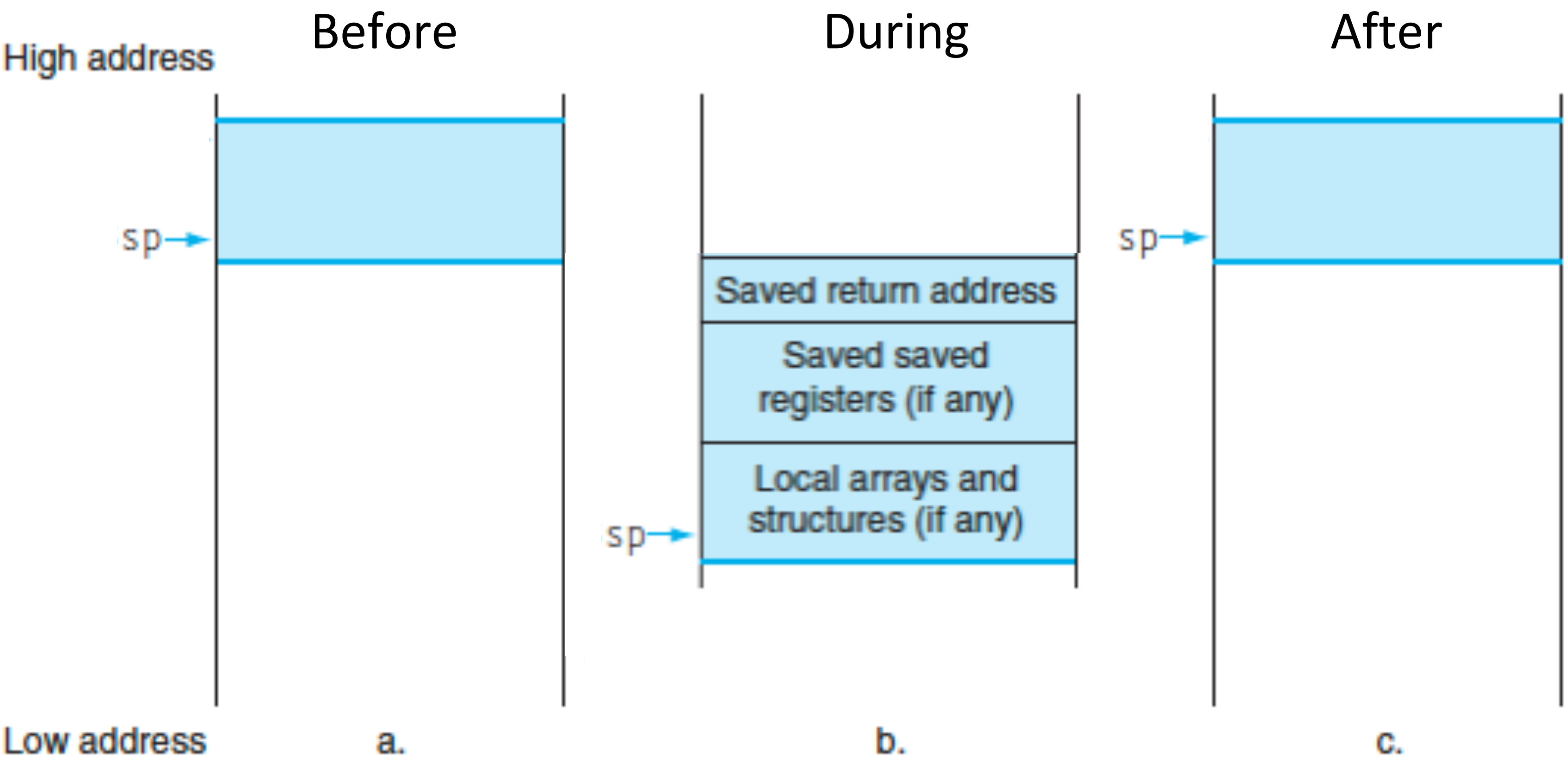
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

gp and tp are special registers we won't worry about in this class

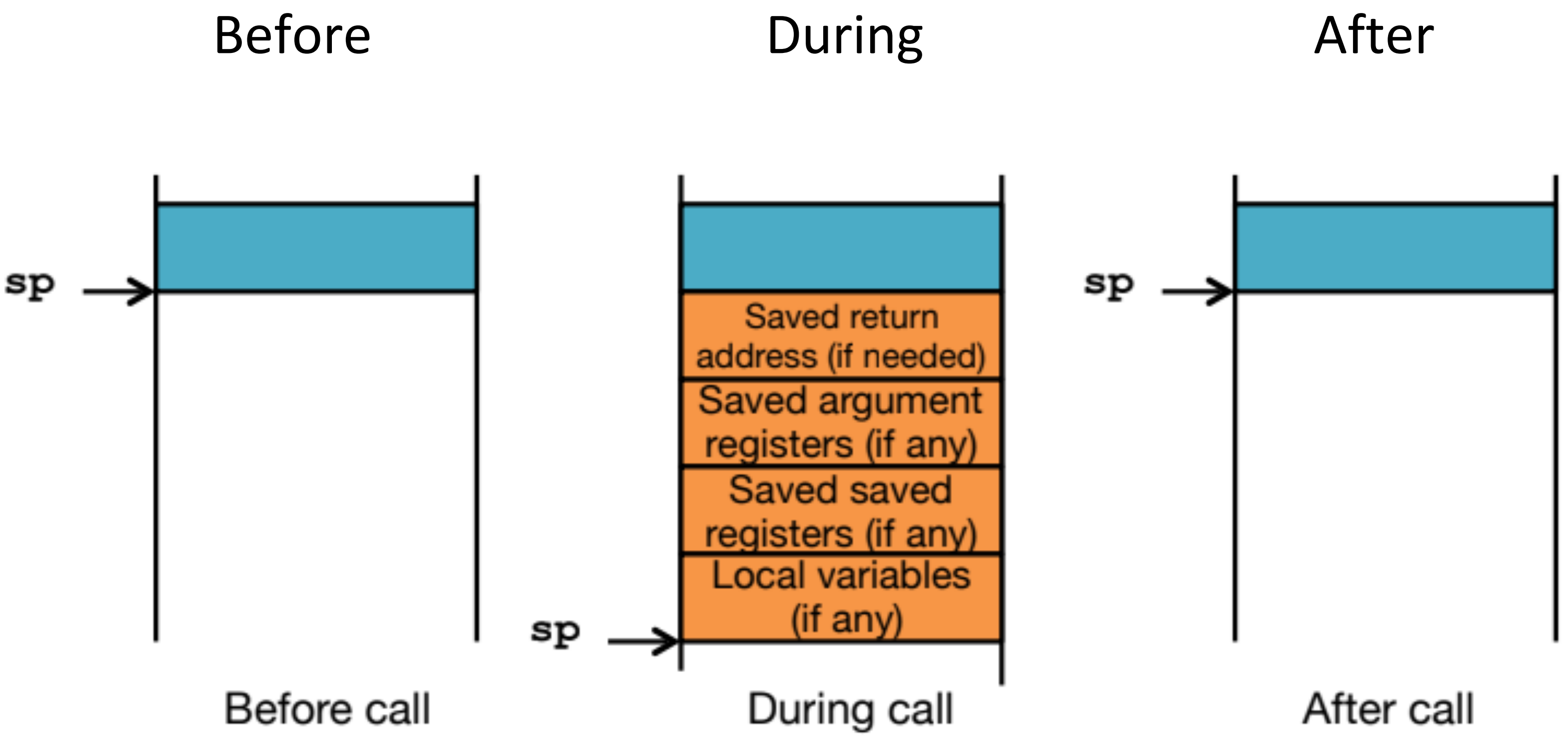
How do we save registers? The stack!



Stack Before, During, After Call



Stack Before, During, After Call



```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
sumSquare:
          addi sp,sp,-8      # make space on stack
"push"    sw ra, 4(sp)       # save ret addr
          sw a1, 0(sp)       # save y
          add a1,a0,x0       # set 2nd mult arg
          jal mult           # call mult
          lw a1, 0(sp)       # restore y
"pop"     add a0,a0,a1       # ret val = mult(x,x)+y
          lw ra, 4(sp)       # get ret addr
          addi sp,sp,8       # restore stack
          jr ra
mult:     ...
```

Basic Structure of a Function

Prologue

```
func_label:
addi sp,sp, -framesize
sw ra, <framesize-4>(sp)
#store other callee saved registers
#save other regs if need be
```

Body **(call other functions…)**

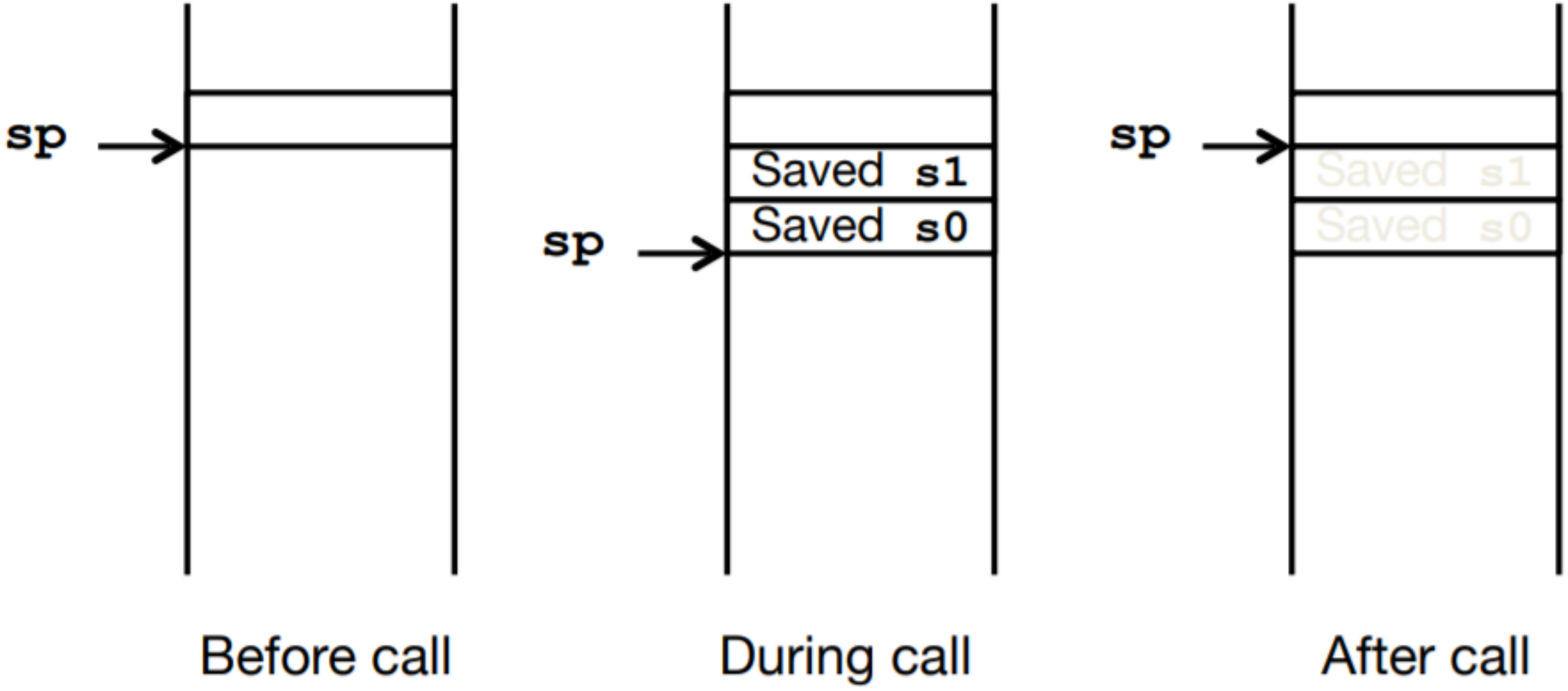
```
...
```

Epilogue

```
#restore other regs if need be
#restore other callee saved registers
lw ra, <framesize-4>(sp)
addi sp,sp, framesize
jr ra
```

Stack during function execution

- Need to save old values of s0 and s1



Example: Using Saved Registers

```
myFunc: # Uses s0 and s1
   addi     sp,sp,-12   # This is the Prologue
   sw       ra,8(sp)    # Save saved registers
   sw       s0,4(sp)
   sw       s1,0(sp)
   ...                  # Do stuff with s0 and s1
   jal      func1       # func1 and func2 will abide by convention,
   ...                  #   so we don't care if they use s0 or s1, we can
   jal      func2       #   use them normally
   ...                  # Do stuff with s0 and s1
   lw       s1,0(sp)    # This is the Epilogue
   lw       s0,4(sp)    # Restore saved registers
   lw       ra,8(sp)
   addi     sp,sp,12
   jr       ra          # return
```

Example: Using Volatile Registers

```
myFunc: # Uses t0
   addi      sp,sp,-4     # This is the Prologue
   sw        ra,0(sp)     # Save saved registers
   ...                    # Do stuff with t0
   addi      sp,sp,-4     # Save volatile registers
   sw        t0,0(sp)     #   before calling a function
   jal       func1        # Function may change t0
   lw        t0,0(sp)     # Restore volatile registers
   addi      sp,sp,4      #   before you use them again
   ...                    # Do stuff with t0
   lw        ra,0(sp)   # This is the Epilogue
   addi      sp,sp,4    # Restore saved registers
   jr        ra         # return
```

Register Conventions Summary

- One more time for luck:
  - CalleR must save any volatile registers it is using onto the stack before making a procedure call
  - CalleR can trust saved registers to maintain values
  - CalleE must "save" any saved registers it intends to use by putting them on the stack before overwriting their values
- Notes:
  - CalleR and calleE only need to save the appropriate registers *they are using* (not all!)
  - Don't forget to restore the values later

RISC-V Agenda

- Sign Extension Practice
- Pseudo-Instructions
- C to RISC-V Practice
- Functions in Assembly
- Function Calling Conventions
- **Summary**

Example function with calling convention

```c
int Leaf(int g, int h,
        int i, int j) {

    int f;
    f = (g+h)-(i+j);
    return f;
}
```

```
Leaf:
addi sp,sp,-8
# allocate stack
sw s1,4(sp)         # save s1
sw s0,0(sp)         # save s0

add s0,a0,a1        # s0 = g+h
add s1,a2,a3        # s1 = i+j
sub a0,s0,s1
# return value a0 = s0-s1

lw s0,0(sp)         # restore s0
lw s1,4(sp)         # restore s1
addi sp,sp,8        # free stack
jr ra               # return
```

Choosing Your Registers

- Minimize register footprint
  - Optimize to reduce number of registers you need to save by choosing which registers to use in a function
  - Only save when you absolutely have to
- Function does NOT call another function
  - Use only `t0-t6` and there is nothing to save!
- Function calls other function(s)
  - Values you need throughout go in `s0-s11`, others go in `t0-t6`
  - At each function call, check number arguments and return values for whether you or not you need to save

Different register choices could reduce effort

```
int Leaf(int g, int h,
         int i, int j) {

      int f;
      f = (g+h)-(i+j);
      return f;
}
```

```
Leaf:
# nothing to save on stack

add t0,a0,a1        # t0 = g+h
add t1,a2,a3        # t1 = i+j
sub a0,t0,t1
# return value a0 = t0-t1
# nothing to restore from stack
jr ra               # return
```

Be lazy! Use register choices that minimize saving to the stack.
It makes your program faster too...

Summary (1/2)

- Pseudo-instructions
- Functions is assembly
  - Six steps of calling a function
    1. Place arguments
    2. Jump to function
    3. Create local storage (Prologue)
    4. Perform desired task
    5. Place return value and clean up storage (Epilogue)
    6. Jump back to caller

Summary (2/2)

- Calling conventions
  - Need a method for knowing which registers can be trusted across function calls
  - Caller-saved registers (Volatile Registers)
    - Saved by caller if needed
    - Free to use by callee
  - Callee-saved registers (Saved Registers)
    - Saved by callee if needed
    - Safe across function calls for caller