CS 61C:
Great Ideas in Computer Architecture
More RISC-V Instructions and *How to Implement Functions*

Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

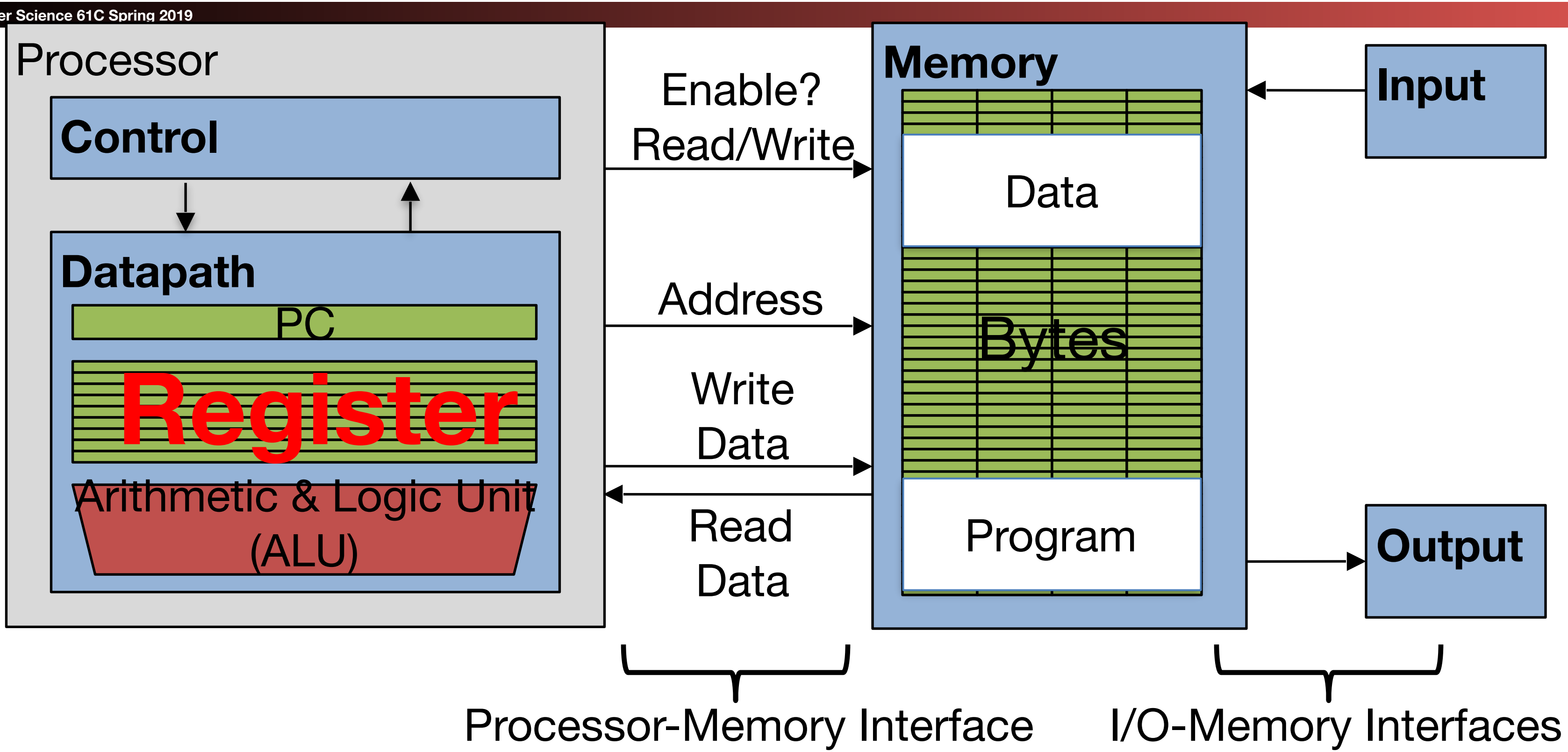
Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

Review From Last Lecture …

- Computer's native operations called instructions. The instruction set defines all the valid instructions.
- RISC-V is example RISC instruction set - used in CS61C
  - Lecture/problems use 32-bit RV32 ISA, book uses 64-bit RV64 ISA
- Rigid format: one operation, two source operands, one destination
  - **`add,sub`**
  - **`lw,sw,lb,sb`** to move data to/from registers from/to memory
- Simple mappings from arithmetic expressions, array access, in C to RISC-V instructions

Recap: Registers live inside the Processor



RISC-V Logical Instructions

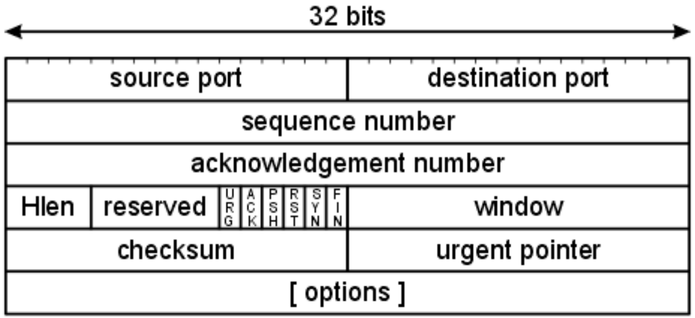
- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called logical operations

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	**and**
Bit-by-bit OR	|	|	**or**
Bit-by-bit XOR	^	^	**xor**
Shift left logical	<<	<<	**sll**
Shift right logical	>>	>>	**srl**

Why Shifts and Logical Operations? "Bit Twiddling…"

- Often have to pack/unpack fields
- EG, in c:
  - int *packet
    packet[0] = sport << 16 | dport
- Becomes (packet in x1, sport in x2, dport in x3)
  - **`slli x4, x2, 16`**
  - **`or x4, x4, x3`**
  - **`sw x4, 0(x1)`**

TCP header format



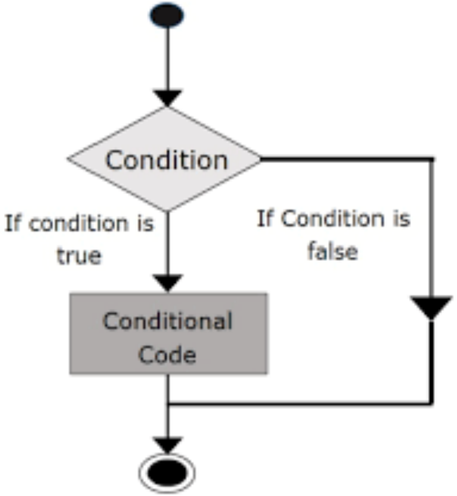
Computer Decision Making

- Based on computation, do something different
- Normal operation on CPU is to execute instructions in sequence
- Need special instructions for programming languages: *if*-statement

- RISC-V: *if*-statement instruction is

  `beq register1,register2,L1`

  means: go to instruction labeled L1
  if (value in register1) == (value in register2)
  ....otherwise, go to next instruction
- **`beq`** stands for *branch if equal*
- Other instruction: **`bne`** for *branch if not equal*



Types of Branches

- **Branch** – change of control flow
- **Conditional Branch** – change control flow depending on outcome of comparison
  - branch *if* equal (**`beq`**) or branch *if not* equal (**`bne`**)
  - Also branch if less than (**`blt`**) and branch if greater than or equal (**`bge`**)
- **Unconditional Branch** – always branch
  - a RISC-V instruction for this*: jump (`j`)*

More on unconditional branches…

- Only two instructions
  - `jal rd offset`
  - `jalr rd rs (offset)`
- Jump And Link
  - Add the immediate value to the current address in the program (the “Program Counter”), go to that location
    - The offset is 20 bits, sign extended and left-shifted ***one (not two)***
  - At the same time, store into `rd` the value of PC+4
  - `j offset` == `jal x0 offset` (yes, jump is a pseudo-instruction in RISC-V)
- Two uses:
  - Unconditional jumps in loops and the like
  - Calling other functions

Jump and Link Register

- The same except the destination
  - Instead of PC + immediate it is **`rs`** + immediate
    - Same immediate format as I-type: 12 bits, sign extended
- Again, if you don't want to record where you jump to…
  - **`jr rs`** == **`jalr x0 rs`**
- Two main uses
  - Returning from functions (which were called using Jump and Link)
  - Calling pointers to function
  - We will see how soon!

Peer Instruction

Which of the following is TRUE?

A: `add x10,x11,4(x12)` is valid in RV32

B: can byte address 8GB of memory with an RV32 word

C: `imm` must be multiple of 4 for `lw x10,imm(x10)` to be valid

D: None of the above

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion …

Peer Instruction

Which of the following is TRUE?

A: **`add x10,x11,4(x12)`** is valid in RV32

B: can byte address 8GB of memory with an RV32 word

C: **`imm`** must be multiple of 4 for **`lw x10,imm(x10)`** to be valid

D: None of the above

Peer Instruction

Which of the following is TRUE?

A: `add x10,x11,4(x12)` is valid in RV32

B: can byte address 8GB of memory with an RV32 word

C: `imm` must be multiple of 4 for `lw x10,imm(x10)` to be valid

D: None of the above

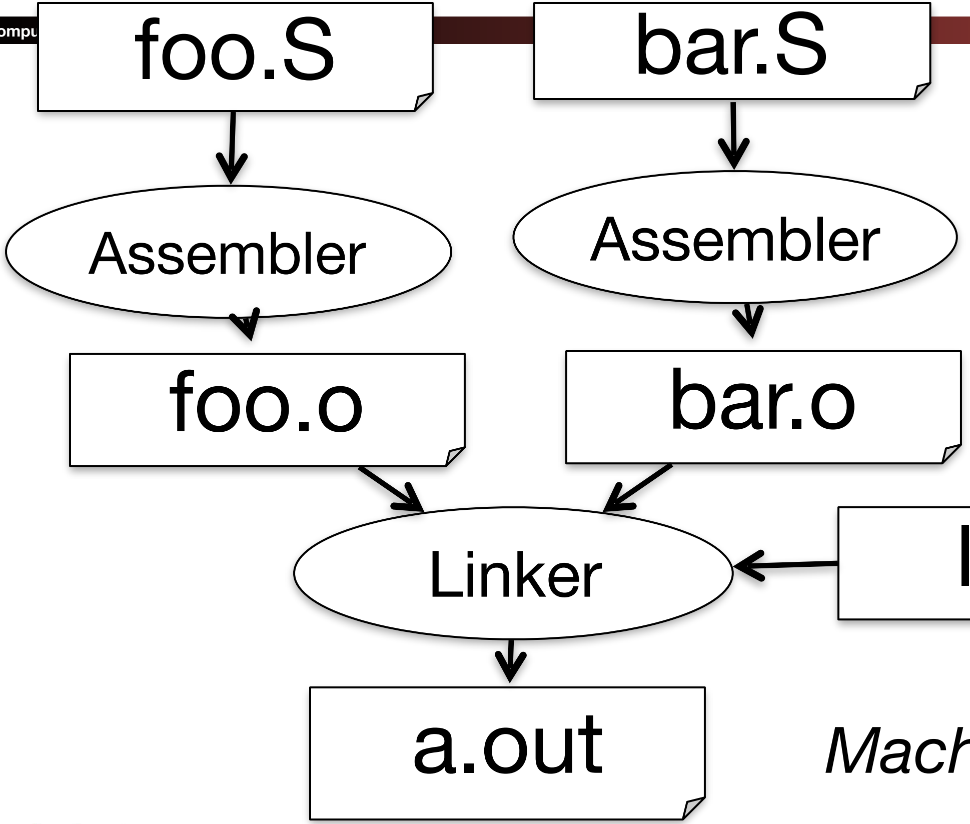
Administrivia

- Project 1-2 will be released Saturday…
  - Project PARRRTAYYY date/time TBD
- CSM signups are now available
- Reminder: Lab Load Balancing…
- Reminder: Slip days!
  - If you're sick, or your grandma died, or heck, you just feel like bleh and not doing it, you can just use your slip days, no need to ask for extensions.

Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

Assembler to Machine Code (more later in course)



Assembler source files (text)

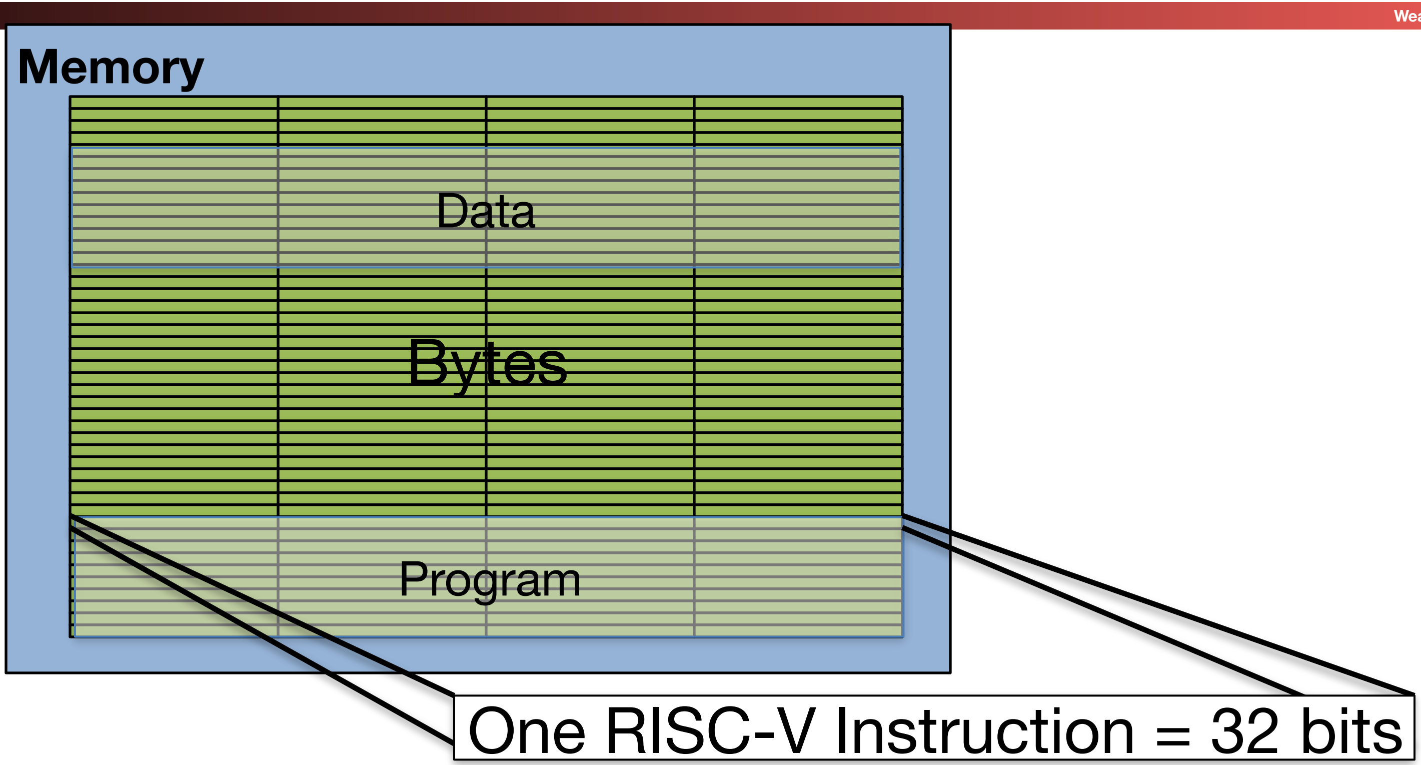
Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

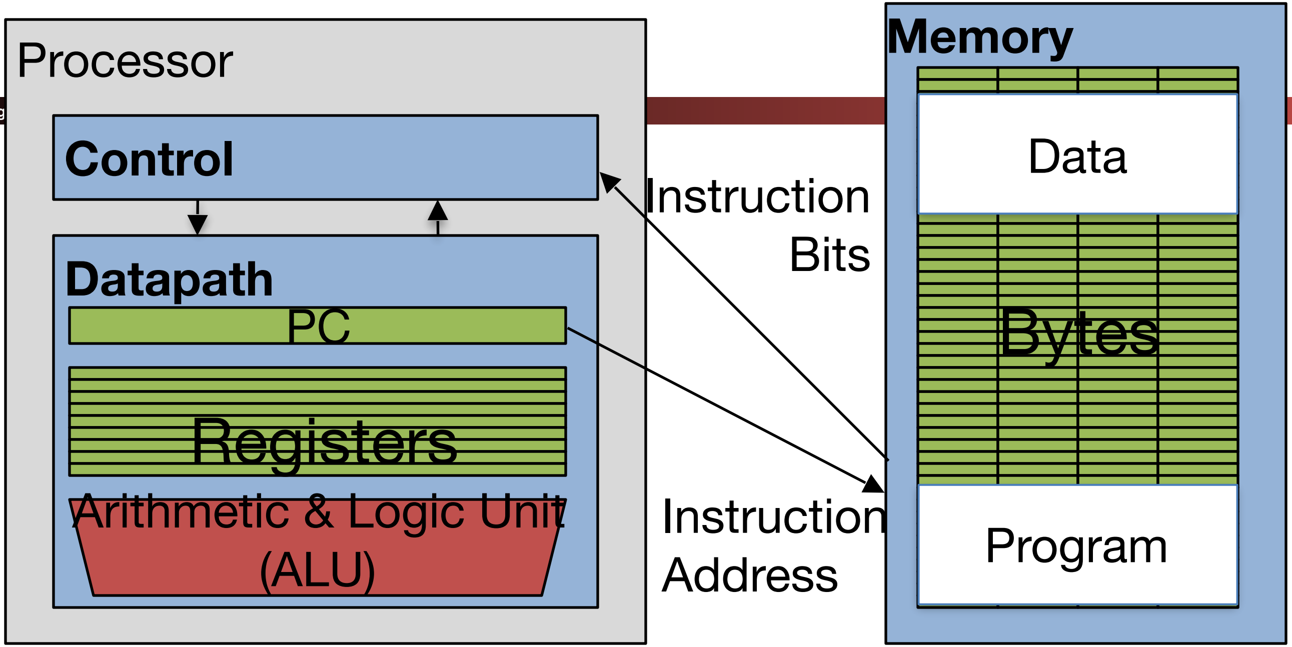
Pre-built object file libraries

Machine code executable file

How Program is Stored



Program Execution



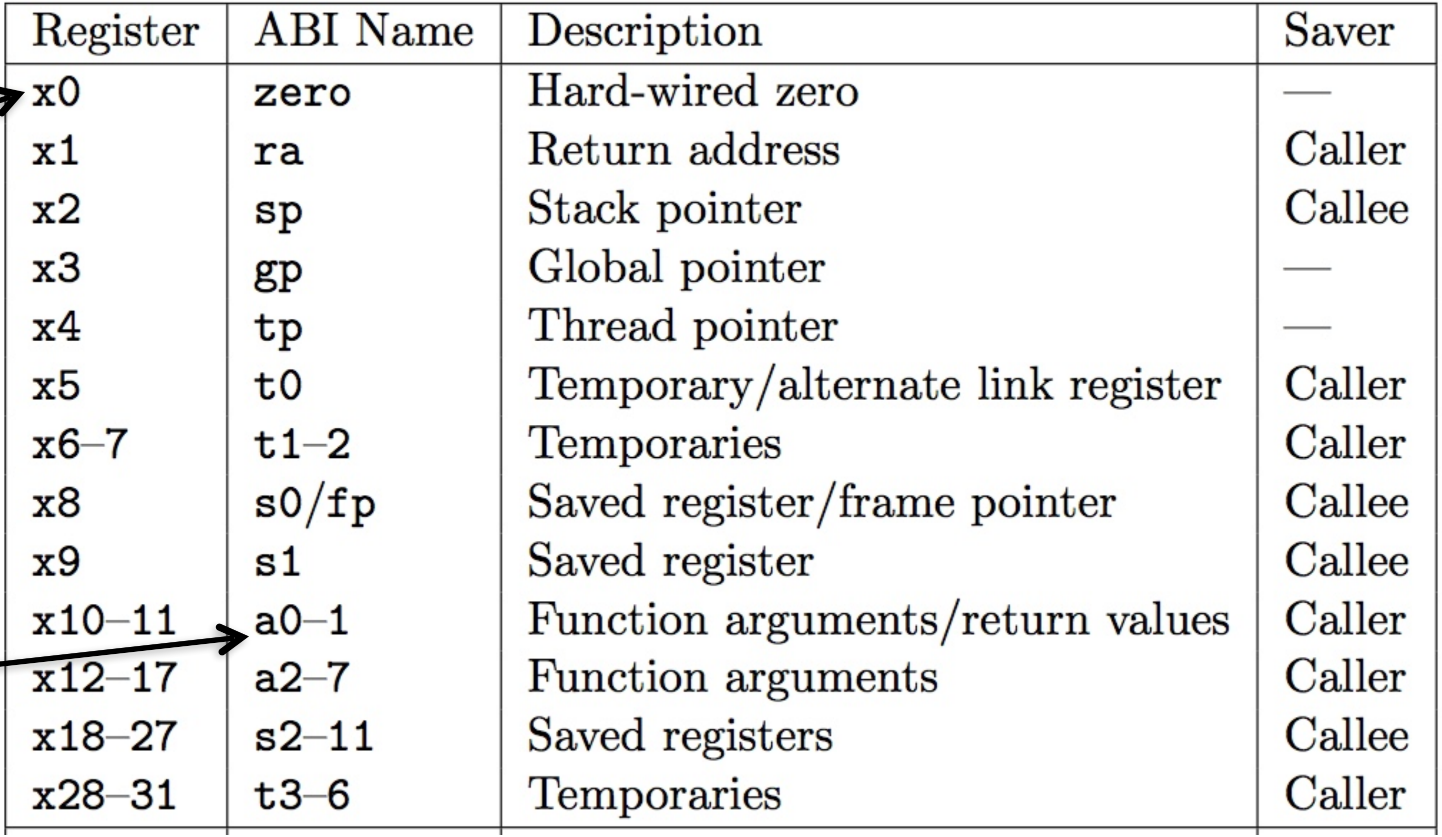
- **PC** (program counter) is special internal register inside processor holding byte address of next instruction to be executed
- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is add +4 bytes to PC, to move to next sequential instruction)

Helpful RISC-V Assembler Features

- Symbolic register names
  - E.g., **`a0-a7`** for argument registers (**`x10-x17`**)
  - E.g., **`zero`** for **`x0`**
- Pseudo-instructions
  - Shorthand syntax for common assembly idioms
  - E.g., “**`mv rd, rs`**” = “**`addi rd, rs, 0`**”
  - E.g., “**`li rd, 13`**” = “**`addi rd, x0, 13`**”

RISC-V Symbolic Register Names

Numbers hardware understands



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Human-friendly symbolic names in assembly code

Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

Six Fundamental Steps in Calling a Function

- Put parameters in a place where function can access them
- Transfer control to function
- Acquire (local) storage resources needed for function
- Perform desired task of the function
- Put result value in a place where calling code can access it and maybe restore any registers you used
- Return control to point of origin.
  - (Note: a function can be called from several points in a program, including from itself.)

The Calling Convention: A Contract Between Functions…

- The “Calling Convention” is the format/usage of registers in a way between the function ***caller*** and function ***callee***, if all functions implement it, everything works out
    - It is effectively a contract between functions
- Registers are two types
    - ***caller-saved***
        - The function invoked (the callee) can do whatever it wants to them!
    - ***callee-saved***
        - The function invoked must restore them before returning (if used)

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- **`a0–a7`** (**`x10–x17`**): eight *argument* registers to pass parameters and two return values (**`a0–a1`**) (***caller saved***)
  - Any more arguments should be passed on the stack
- **`ra`**: one *return address* register for return to the point of origin (**`x1`**) (***caller saved***)
- **`sp`**: pointer to the bottom of the stack (***callee saved***)

More Conventions

- **`s0-s11`** Saved registers: Preserved across function calls
- **`fp`** Frame Pointer: Pointer to the top of the call frame
  - Also is **`s0`**, the first saved register, callee saved
  - Frame pointer can often be omitted by the compiler, but we will use it because it makes things clearer how functions are translated.
- **`t0-t6`** Temporaries: Caller saved

Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

Example

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

- Parameter variables **`g`**, **`h`**, **`i`**, and **`j`** in argument registers **`a0`**, **`a1`**, **`a2`**, and **`a3`**.
- Assume we compute **`f`** by using **`s0`** and **`s1`**

Where Are Old Register Values Saved to Restore Them After Function Call?

- Need a place to save old values before call function, restore them when return
- Ideal is *stack*: last-in-first-out queue (e.g., stack of plates)
  - Push: placing data onto stack
  - Pop: removing data from stack
- Stack in memory, so need register to point to it
- **`sp`** is the *stack pointer* in RISC-V (**`x2`**)
- **`sp`** always points to the last used place on the stack
- Convention is grow stack down from high to low addresses
  - *Push* decrements **`sp`**, *Pop* increments **`sp`**

RISC-V Code for Leaf()

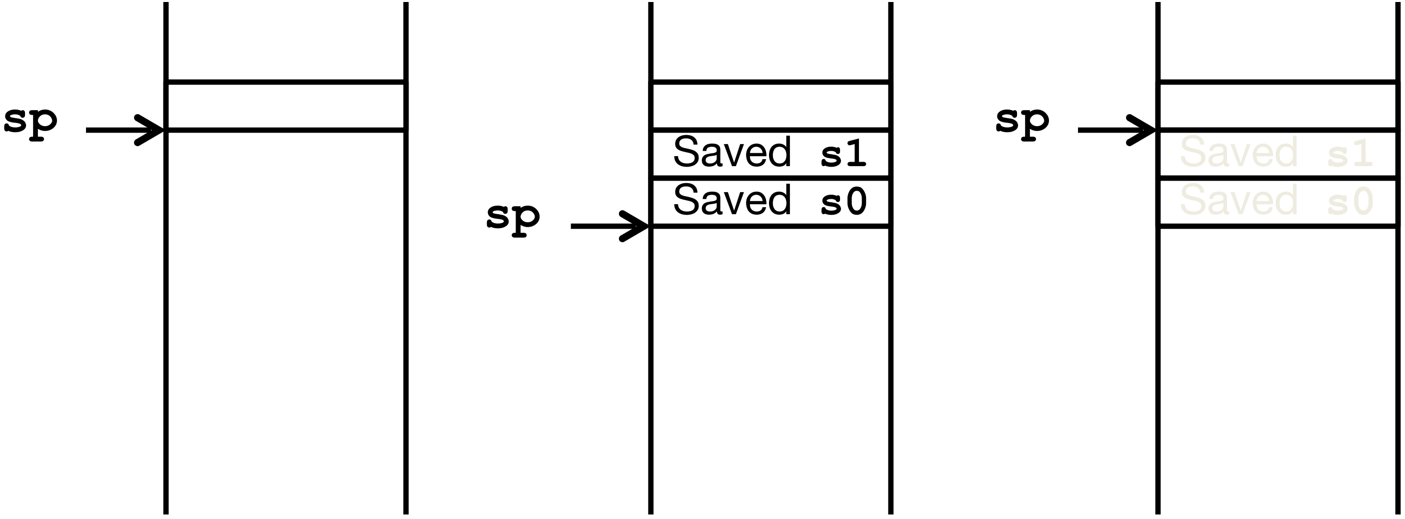
```
Leaf: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

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

      lw s0, 0(sp) # restore register s0 for caller
      lw s1, 4(sp) # restore register s1 for caller
      addi sp,sp,8 # adjust stack to delete 2 items
      jr ra        # jump back to calling routine
```

Stack Before, During, After Function

- Need to save old values of **`s0`** and **`s1`**



Before call During call After call

Of course, we could optimize the function…

- We could have just as easily used **`t0`** and t1 instead…

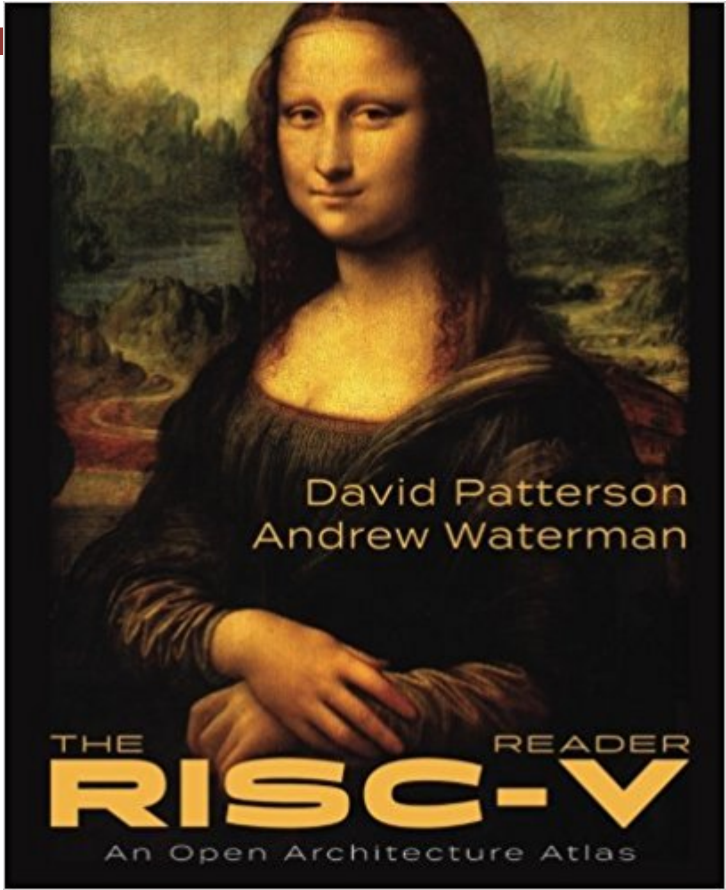
```
leaf:
   add s0,a0,a1 # s0 = g + h
   add s1,a2,a3 # s1 = i + j
   sub a0,s0,s1 # return value (g + h) - (i + j)
   ret # ret is shorthand for jalr x0 ra
```

New RISC-V book!

- “The RISC-V Reader”, David Patterson, Andrew Waterman

- Available from Amazon
- Print edition **`$19.99`**
- Kindle edition to follow at some point

- **Recommended, not required**
- Me? I’m cheap and just refer to the ISA documentation directly:
    - https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf



What If a Function Calls a Function? Recursive Function Calls?

- Would clobber values in **`a0-a7`** and **`ra`**
- What is the solution?

Nested Procedures (1/2)

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

- Something called **sumSquare**, now **sumSquare** is calling **mult**
- So there's a value in **ra** that **sumSquare** wants to jump back to, but this will be overwritten by the call to **mult**

Need to save **sumSquare** return address before call to **mult**

Nested Procedures (2/2)

- In general, may need to save some other info in addition to **`ra`**.
- When a C program is run, there are three important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes - e.g., C globals
  - Heap: Variables declared dynamically via **`malloc`**
  - Stack: Space to be used by procedure during execution; this is where we can save register values

Optimized Function Convention

To reduce expensive loads and stores from spilling and restoring registers, RISC-V function-calling convention divides registers into two categories:

1. Preserved across function call
   - Caller can rely on values being unchanged
   - **`sp`**, **`gp`**, **`tp`**, "saved registers" **`s0- s11`** (**`s0`** is also **`fp`**)
2. Not preserved across function call
   - Caller *cannot* rely on values being unchanged
   - Argument/return registers **`a0-a7,ra`**, "temporary registers" **`t0-t6`**

Peer Instruction

- Which statement is FALSE?
- A: RISC-V uses **`jal`** to invoke a function and **`jr`** to return from a function
- B: **`jal`** saves PC+1 in **`ra`**
- C: The callee can use temporary registers (**`t`***i*) without saving and restoring them
- D: The caller can rely on save registers (**`s`***i*) without fear of callee changing them

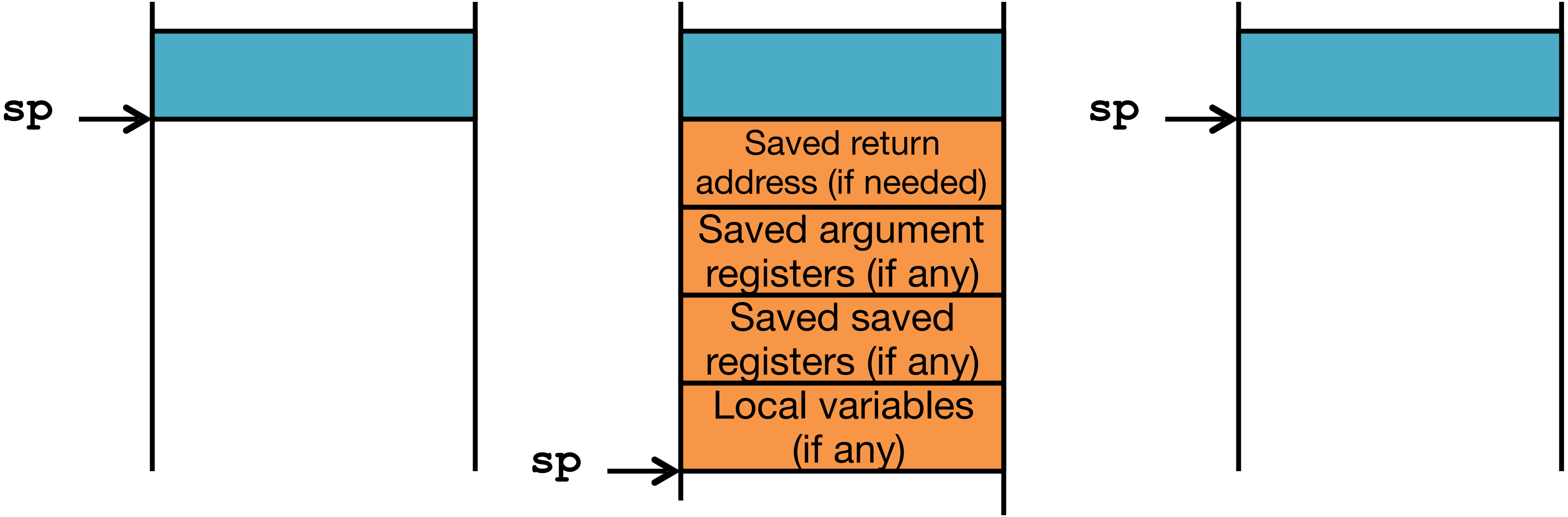
Peer Instruction

- Which statement is FALSE?
- A: RISC-V uses **`jal`** to invoke a function and **`jr`** to return from a function
- B: **`jal`** saves PC+1 in **`ra`**
- C: The callee can use temporary registers (**`t`***i*) without saving and restoring them
- D: The caller can rely on save registers (**`s`***i*) without fear of callee changing them

Allocating Space on Stack

- C has two storage classes: automatic and static
  - *Automatic* variables are local to function and discarded when function exits
  - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that aren't in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables

Stack Before, During, After Function


sp
Saved return address (if needed)
Saved argument registers (if any)
Saved saved registers (if any)
Local variables (if any)
sp
sp

Before call

During call

After call

Using the Stack (1/2)

- So we have a register **sp** which always points to the last used space in the stack
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info
- So, how do we compile this?

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

Using the Stack (2/2)

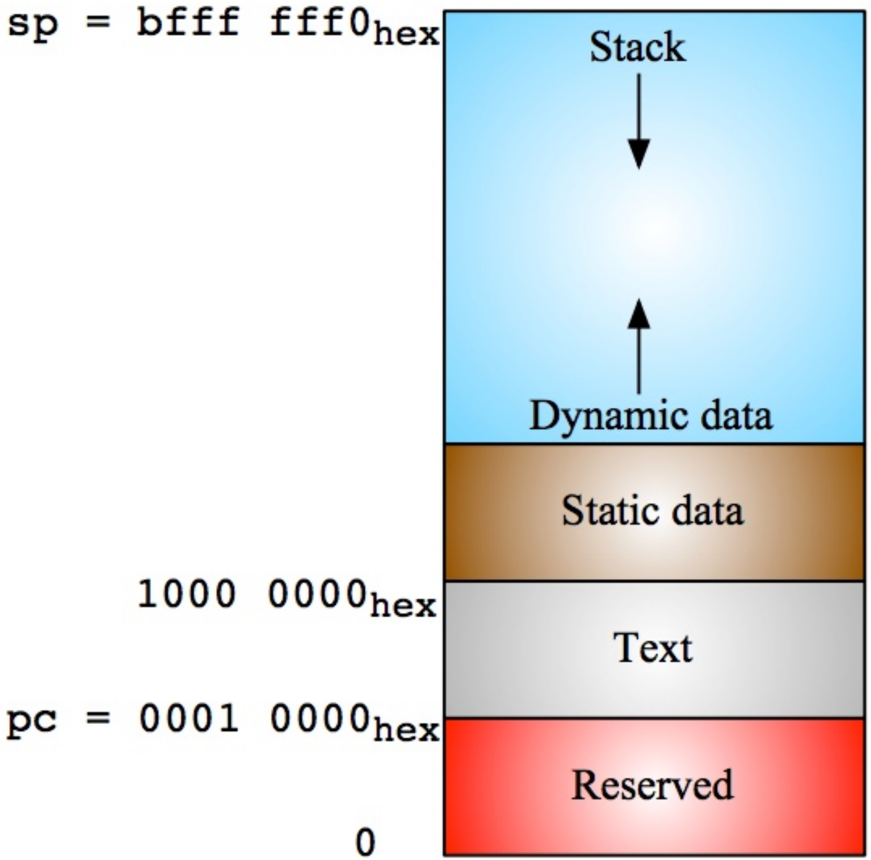
```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
        sumSquare:
“push”      addi sp,sp,-8   # reserve space on stack
            sw ra, 4(sp)    # save ret addr
            sw a1, 0(sp)    # save y
            mv a1,a0        # mult(x,x)
            jal mult        # call mult
            lw a1, 0(sp)    # restore y
            add a0,a0,a1    # mult()+y
            lw ra, 4(sp)    # get ret addr
“pop”       addi sp,sp,8    # restore stack
            jr ra
        mult: ...
```

Where is the Stack in Memory?

- RV32 convention (RV64 and RV128 have different memory layouts)
- Stack starts in high memory and grows down
  - Hexadecimal (base 16) : `bfff_fff0`$_{\text{hex}}$
- RV32 programs (*text segment*) in low end
  - `0001_0000`$_{\text{hex}}$
- *static data segment* (constants and other static variables) above text for static variables
  - RISC-V convention *global pointer* (`gp`) points to static
  - RV32 `gp` = `1000_0000`$_{\text{hex}}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

RV32 Memory Allocation



A Richer Translation Example...

- `struct node {unsigned char c, struct node *next};`
  - c will be at 0, next will be at 4 because of alignment
  - sizeof(struct node) == 8

-
  ```
  struct node * foo(char c){
      struct node *n
      if(c < 0) return 0;
      n = malloc(sizeof(struct node));
      n->next = foo(c - 1);
      n->c = c;
      return n;
  }
  ```

So What Will We Need?

- We’ll need to save **`ra`**
  - Because we are calling other function
- We’ll need a local variable for c
  - Because we are calling other functions
  - Lets put this in **`s0`**
- We’ll need a local variable for n
  - Lets put this in **`s1`**
- So lets form the “preamble” and “postamble”
  - What we always do on entering and leaving the function

Preamble and Postamble

- `foo:`

```
  addi sp sp -12  # Get stack space for 3 registers
  sw s0 0(sp)     # Save s0
  sw s1 4(sp)     # Save s1
  sw ra 8(sp)     # Save ra

{body goes here}  # whole function stuff…

foo_exit:         # Assume return value already in a0
  lw s0 0(sp)     # Restore Registers
  lw s1 4(sp)
  lw ra 8(sp)
  add sp sp 12    # Restore stack pointer
  ret             # aka.. jalr x0 ra
```

And now the body…

```
 bge a0 x0 foo_false   # if c < 0,
 add a0 x0 x0          # return 0 in a0
 j foo_done
foo_false:
 mv s0 a0              # save c
 li a0 8               # sizeof(struct node) (pseudoinst)
 jal malloc            # call malloc
 mv s1 a0              # save n
 addi a0 s0 -1         # c-1 in a0
 jal foo               # call foo recursively
 sw a0 4(s1)           # write the return value into n->next
 sb s0 0(s1)           # write c into n->c (just a byte)
 mv a0 s1              # return n in a0
```

Again, we skipped a lot of optimization…

- On the leaf node (c < 0) we didn’t need to save ra (or even the s0 & s1)
- We could get away with only one saved register..
  - Save c into s0
  - call malloc
  - save c into n
  - calc c-1
  - save n in s0
- But again, we don’t needlessly optimize…

Outline

- RISC-V ISA and C-to-RISC-V Review
- Program Execution Overview
- Function Call
- Function Call Example
- And in Conclusion …

And in Conclusion …

- Functions called with `jal`, return with `jr ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…

  Arithmetic: `add, addi, sub`

  Memory: `lw, sw, lb, lbu, sb`

  Decision: `beq, bne, blt, bge`

  Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!
  - **a0-a7** for function arguments, **a0-a1** for return values
  - **sp**, stack pointer, **ra** return address
  - **s0-s11** saved registers
  - **t0-t6** temporaries
  - **zero**