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

Instructors:
Nick Weaver & John Wawrzynek
http://inst.eecs.Berkeley.edu/~cs61c/fa17

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 …

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., RISC-V)

Assembler

Machine Language Program (RISC-V)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

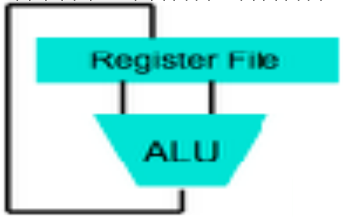
Logic Circuit Description (Circuit Schematic Diagrams)

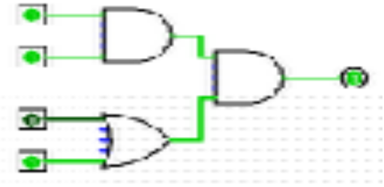
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

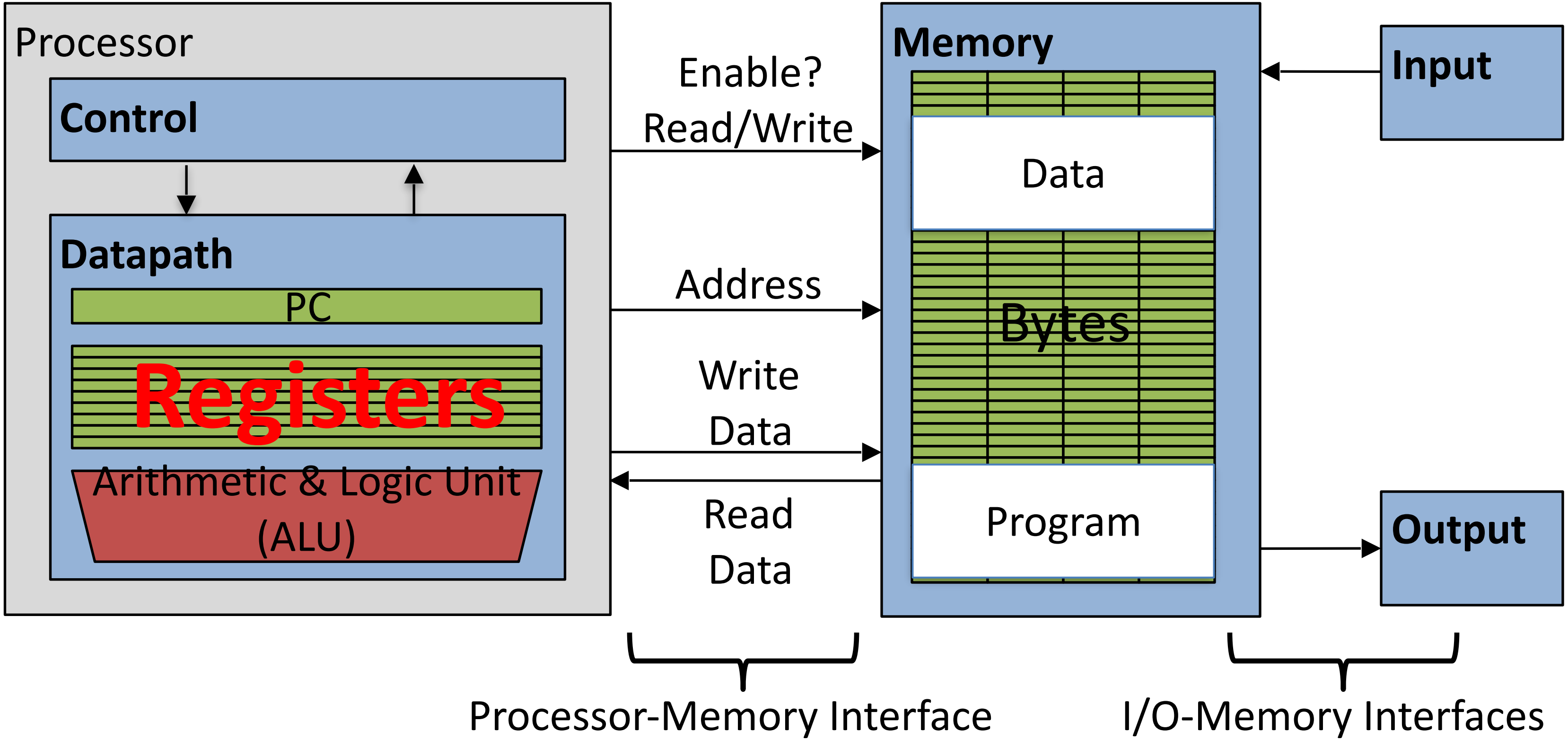




Review From Last Lecture …

- 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*

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

Logical Shifting

- Shift Left Logical: `slli x11,x12,2` # x11 = x12<<2
  - Store in x11 the value from x12 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C

Before: $0000\ 0002_{hex}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

After: $0000\ 000\underline{8}_{hex}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 10\underline{00}_{two}$

What arithmetic effect does shift left have?

- Shift Right Logical: `srli` is opposite shift; >>
  - Zero bits inserted at left of word, right bits shifted off end

Arithmetic Shifting

- *Shift right arithmetic* (**srai**) moves *n* bits to the right (insert high-order sign bit into empty bits)
- For example, if register x10 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$

- If execute sra x10, x10, 4, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

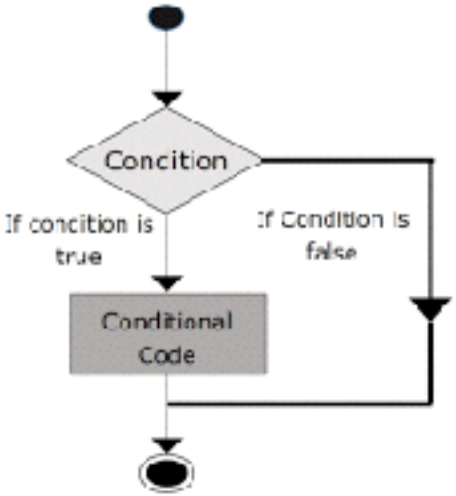
Computer Decision Making

- 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`)*

Outline

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

Example *if* Statement

- Assuming assignments below, compile *if* block

  f → `x10` g → `x11` h → `x12`

  i → `x13` j → `x14`

```
if (i == j)              bne x13,x14,skip
   f = g + h;            add x10,x11,x12
       .        skip:             .
       .                          .
       .                          .
```

Example *if-else* Statement

- Assuming assignments below, compile

  f → `x10` g → `x11` h → `x12` i → `x13` j → `x14`

```
if (i == j)                       bne x13,x14,else
   f = g + h;                     add x10,x11,x12
else                              j done
   f = g - h;             else:   sub x10,x11,x12
                          done:
```

Magnitude Compares in RISC-V

- Until now, we've only tested equalities (== and != in C);
  General programs need to test < and > as well.
- RISC-V magnitude-compare branches:

  "Branch on Less Than"

  Syntax: `blt reg1,reg2, label`

  Meaning: if (reg1 < reg2) // treat registers as signed integers
  goto label;
- "Branch on Less Than Unsigned"

  Syntax: `bltu reg1,reg2, label`

  Meaning: if (reg1 < reg2) // treat registers as unsigned integers
  goto label;

"Branch on Greater Than or Equal" (and it's unsigned version) also exists.

C Loop Mapped to RISC-V Assembly

```
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum +=  A[i];
```

```
# Assume x8 holds pointer to A
# Assign x9=A, x10=sum,
#          x11=i, x13=20
add x9, x8, x0   # x9=&A[0]
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x13,x0,20   # x13=20
Loop:
lw x12, 0(x9)    # x12=A[i]
add x10,x10,x12 # sum+=
addi x9,x9,4     # &A[i++]
addi x11,x11,1   # i++
blt x11,x13,Loop
```

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

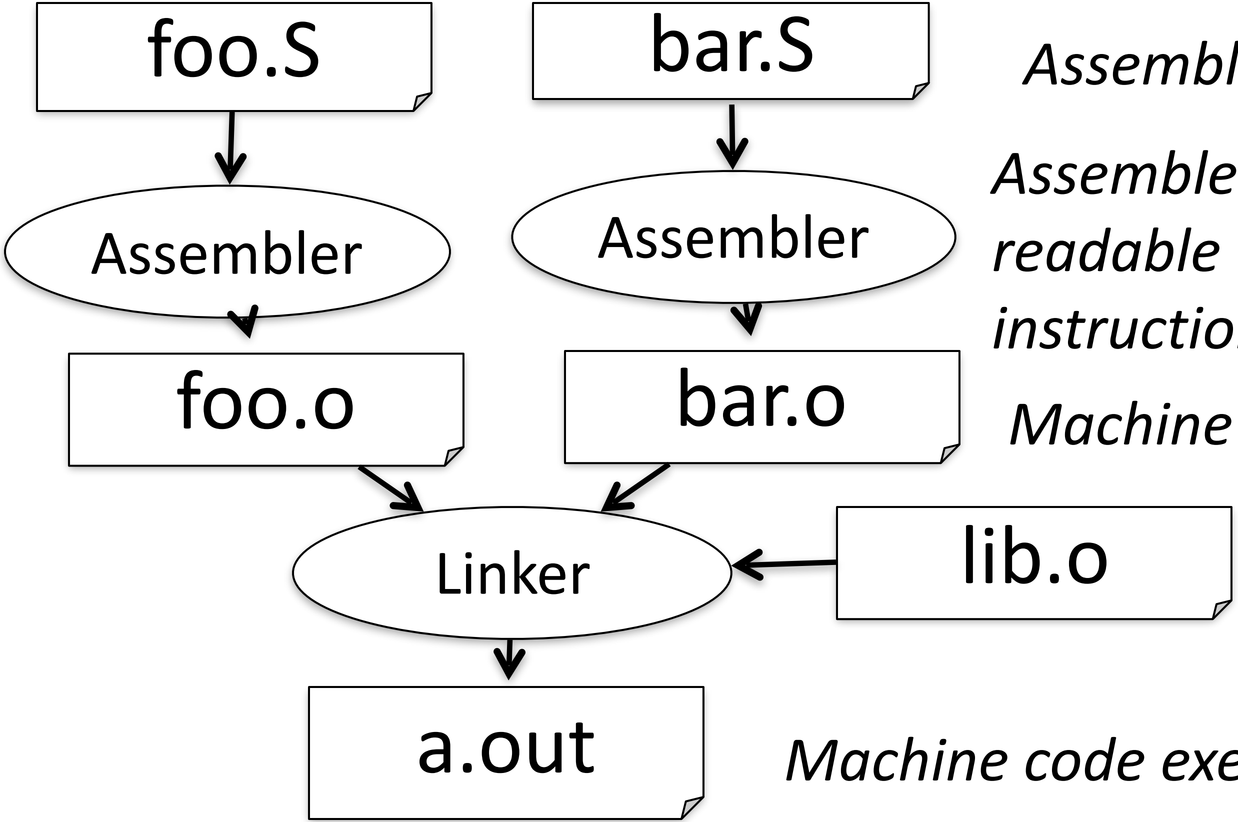
Administrivia

- The Project 1 deadline extended to Thursday, 11:59pm!
- There will be a guerrilla section Thursday 7-9PM.
- Two weeks to Midterm #1!
- Project 2-1 release later this week or early next, due 2/16.
- Project 2-2 release right after midterm and due 2/23.

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)

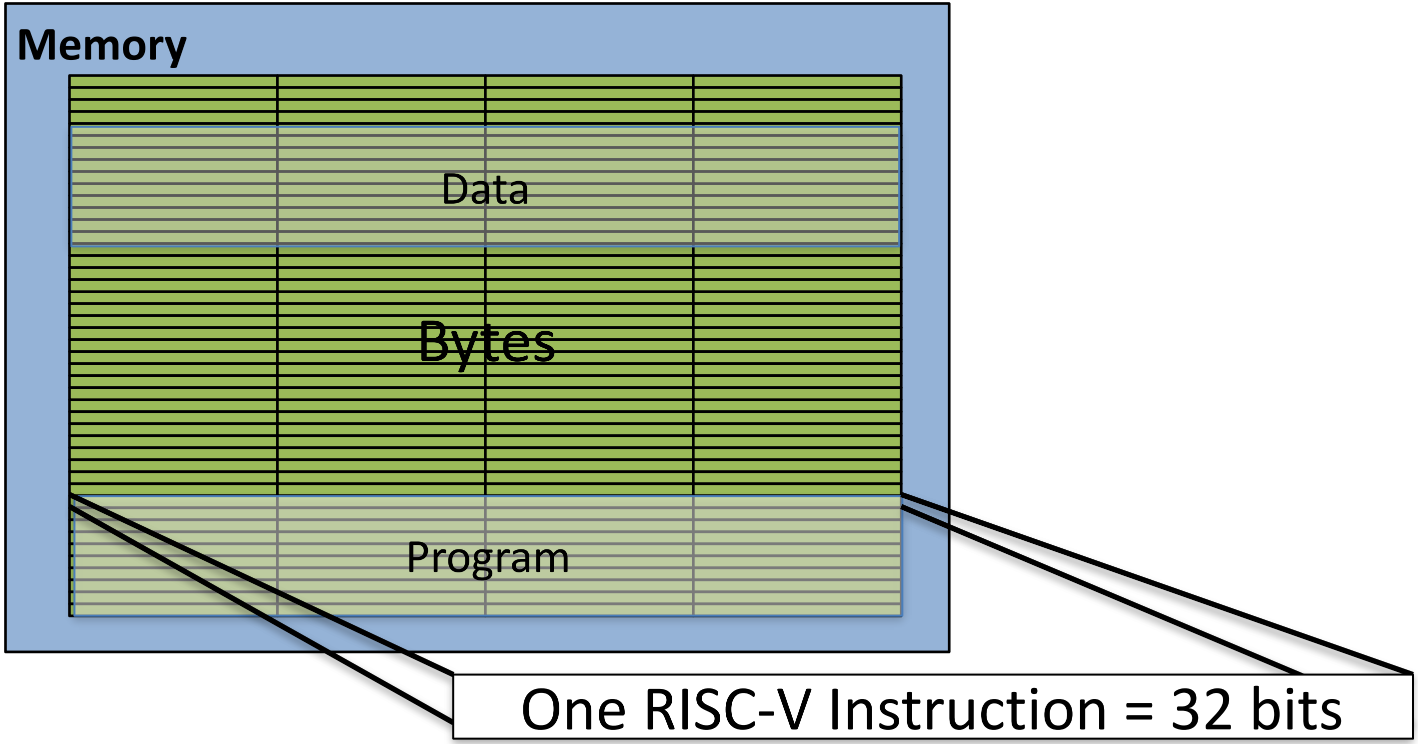
Assembler converts human-readable assembly code to instruction bit patterns

Machine code object files

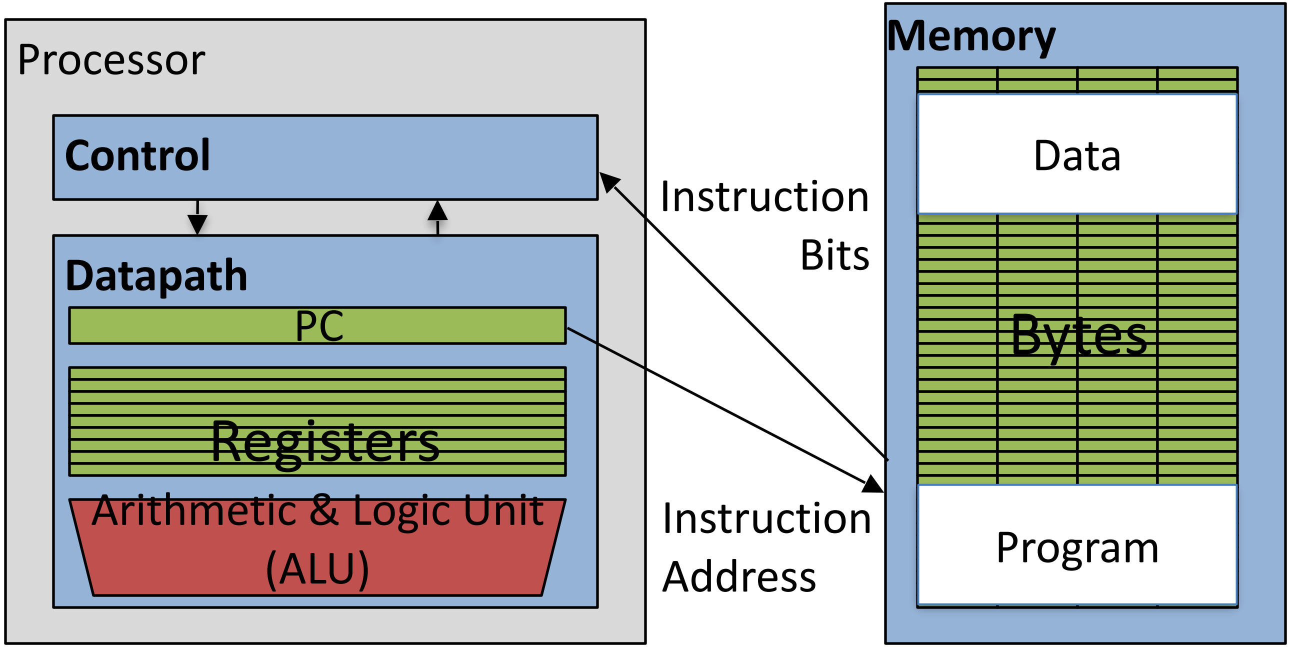
Pre-built object file libraries

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

Human-friendly symbolic names in assembly code

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

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

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

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`**)
- **`ra`**: one *return address* register for return to the point of origin (**`x1`**)

Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
 }
 int sum(int x, int y) {
  return x+y;
 }
```

RISC-V

```
address (shown in decimal)
 1000
 1004
 1008
 1012
 1016
 …
 2000
 2004
```

In RISC-V, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
 }
 int sum(int x, int y) {
  return x+y;
 }
```

RISC-V

```
address (shown in decimal)
 1000 mv a0,s0  # x = a
 1004 mv a1,s1  # y = b
 1008 addi ra,zero,1016 #ra=1016
 1012 j    sum          #jump to sum
 1016 …                 # next instruction
 …
 2000 sum: add a0,a0,a1
 2004 jr   ra      # new instr. “jump register”
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
int sum(int x, int y) {
 return x+y;
}
```

RISC-V

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
2000 sum: add a0,a0,a1
2004 jr   ra    # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi ra,zero,1016 #ra=1016
1012 j sum             #goto sum
```

- After:

```
1008 jal sum  # ra=1012,goto sum
```

- Why have a `jal`?
  - Make the common case fast: function calls very common
  - Reduce program size
  - Don't have to know where code is in memory with `jal`!
- Return from function: *jump register* instruction (`jr`)
  - Unconditional jump to address specified in register: `jr ra`
  - Assembler shorthand: `ret = jr ra`

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()

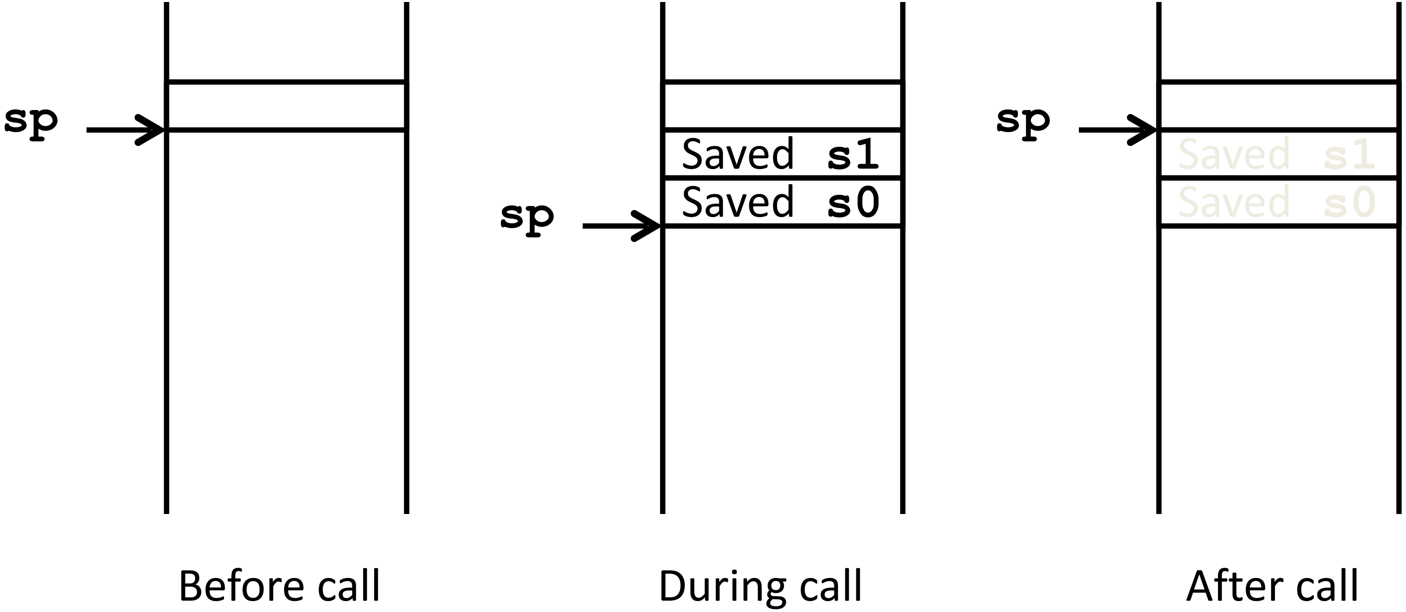
```
Leaf: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      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`**



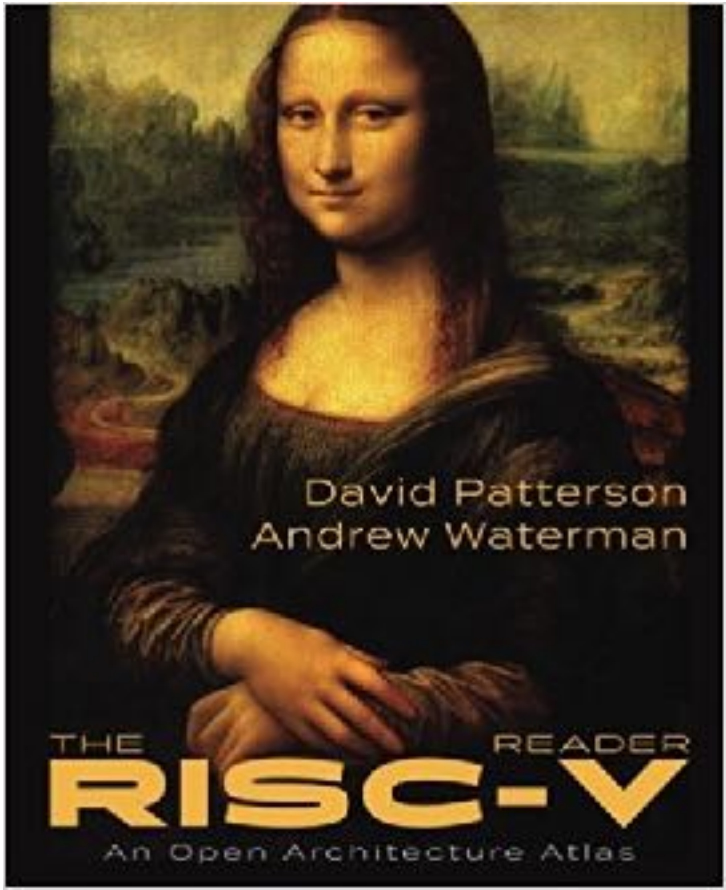
Break!

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**



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

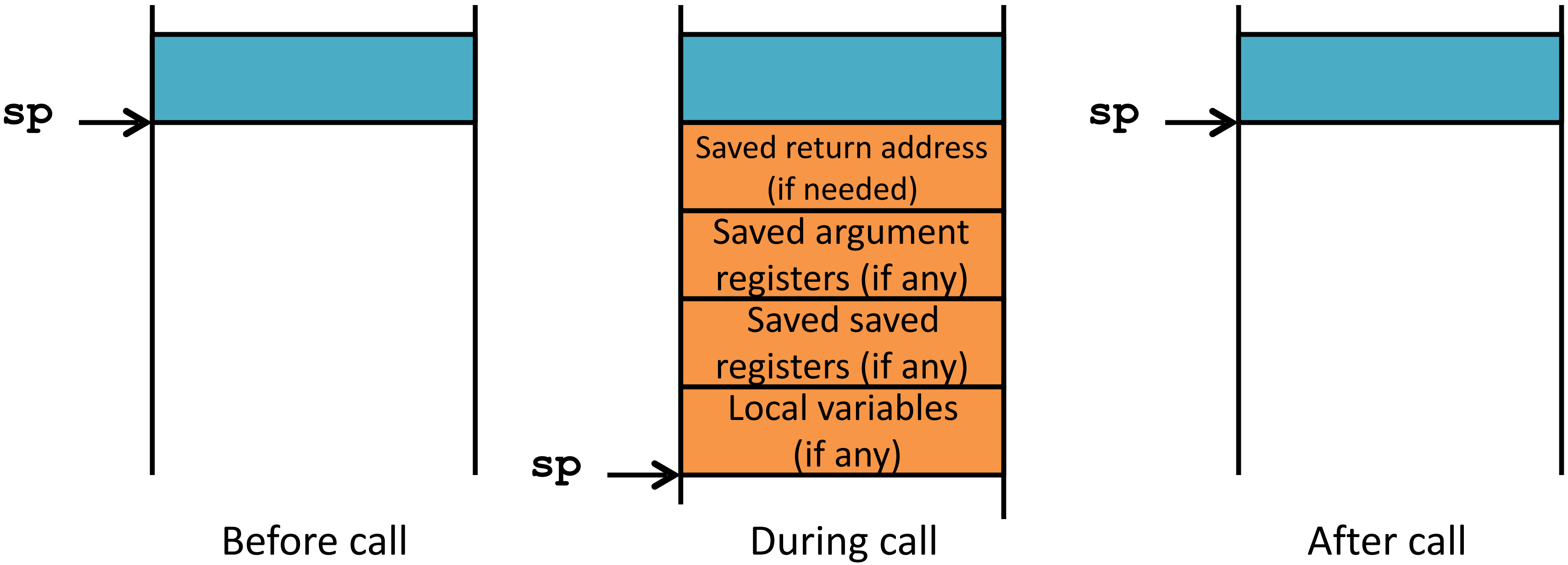
Peer Instruction

- 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



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; }
```

“push”

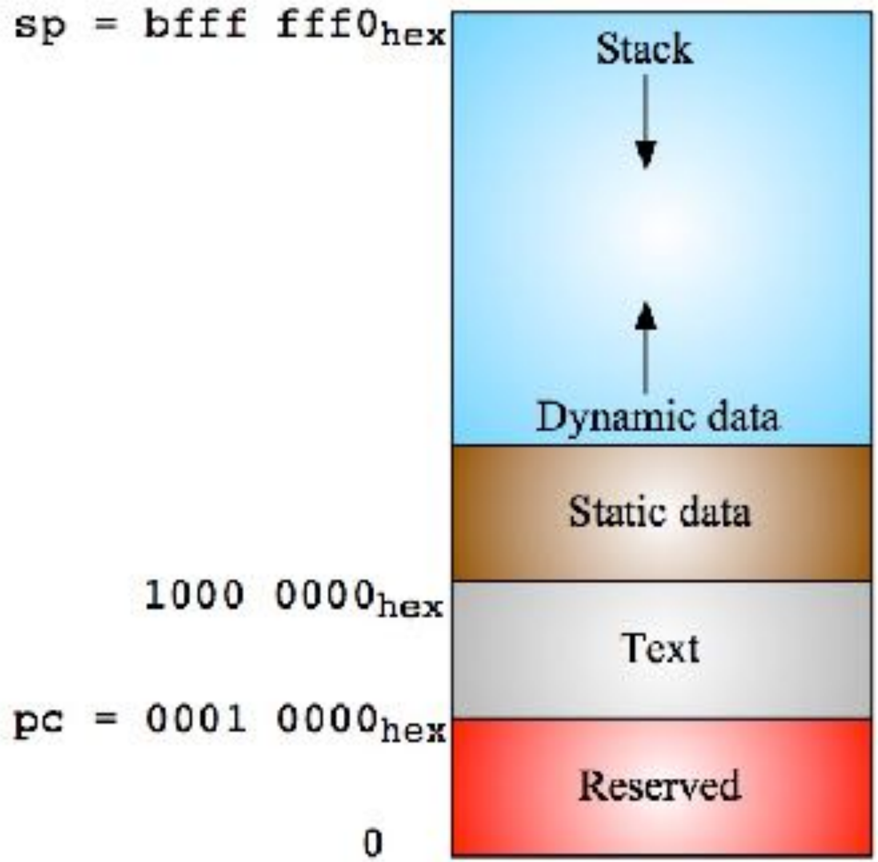
“pop”

```
sumSquare:
        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
        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`**$_{hex}$
- RV32 programs (*text segment*) in low end
  - **`0001_0000`**$_{hex}$
- *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`**$_{hex}$
- *Heap* above static for data structures that grow and shrink ; grows up to high addresses

RV32 Memory Allocation



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`**