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

Instructors:
Krste Asanović and Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa17

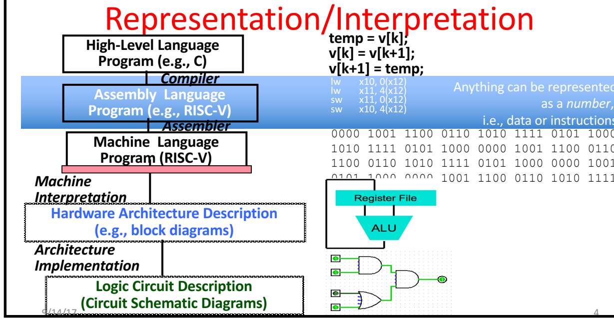
Outline

- 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 …

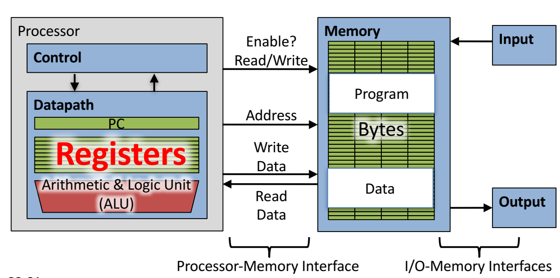
Levels of Representation/Interpretation



Review From Last Lecture …

- Computer "words" and "vocabulary" are called *instructions* and *instruction set* respectively
- 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,mul,div,and,or,sll,srl,sra`
  – `lw,sw,lb,sb` to move data to/from registers from/to memory
  – `beq, bne, j` for decision/flow control
- Simple mappings from arithmetic expressions, array access, in C to RISC-V instructions

Recap: Registers live inside the Processor



Example *if-else* Statement

- Assuming translations 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 Exit
  f = g - h;     Else: sub x10,x11,x12
                 Exit:
```


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;

C Loop Mapped to RISC-V Assembly

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

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


Peer Instruction

Which of the following is TRUE?

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

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

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

YELLOW: None of the above

Peer Instruction

Which of the following is TRUE?

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

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

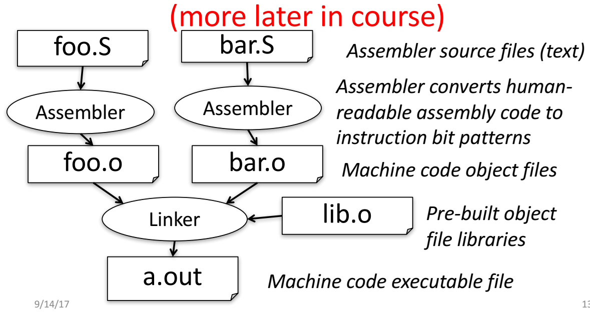
ORANGE: **`imm`** must be multiple of 4 for **`lw x10,imm(x10)`** to be valid

$\boxed{\text{YELLOW: None of the above}}$

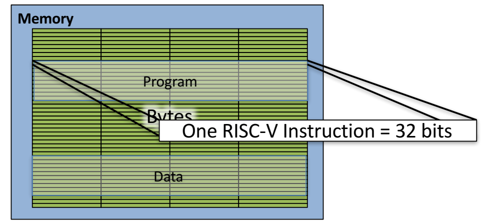
Outline

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

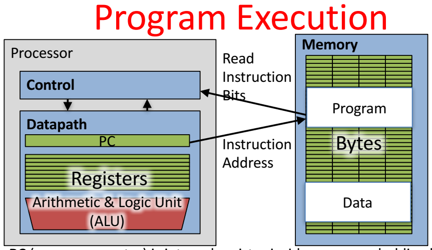
Assembler to Machine Code
(more later in course)



How Program is Stored

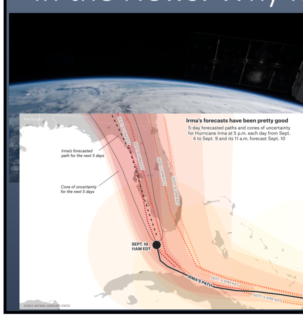


Program Execution



- **PC** (program counter) is 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)

In the News: Why fast computers matter



European Weather supercomputer ECMWF
50 tonnes
~120,000 compute cores (Intel Broadwell)
10 PetaBytes of storage
Runs Linux on each node

Break!

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.2, **`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 restore any registers you used
6. Return control to point of origin, since 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 to 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;
}
```

- 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.

RISC-V

```
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`!

RISC-V Function Call Instructions

- Invoke function: *jump and link* instruction (`jal`)
  (really should be `laj` *"link and jump"*)
  - "link" means form an *address* or *link* that points to calling site to allow function to return to proper address
  - Jumps to address and simultaneously saves the address of the following instruction in register `ra`

```
jal FunctionLabel
```

- 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`, and `f` in `s0`
- Assume need one temporary register `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, and delete
- 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`)
- Convention is grow stack down from high to low addresses
  - *Push* decrements `sp`, *Pop* increments `sp`

RISC-V Code for Leaf()

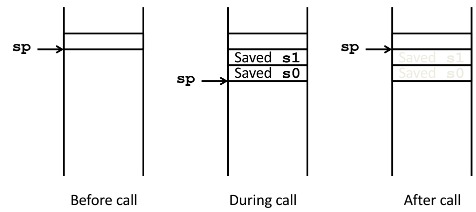
```
Leaf: addi sp,sp,-8 # adjust stack for 2 items
      sw s1, 4(sp)  # save s1 for use afterwards
      sw s0, 0(sp)  # save s0 for use afterwards

      add s0,a0,a1 # f = 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`**

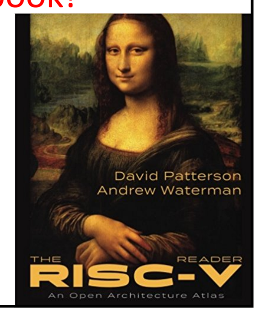


Administrivia

- HW1 is out! Get started early.
- C and Memory Management Guerrilla Session is tonight 7-9pm in 293 Cory
- Small group tutoring sessions have launched

New RISC-V book!

- "The RISC-V Reader", David Patterson, Andrew Waterman
- Available at
- `https://www.createspace.com/7439283`
- Early print edition **`$9.99`**
- Kindle edition to follow at some point
- **Recommended, not required**



Break!

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?
- RED: RISC-V uses **jal** to invoke a function and **jr** to return from a function
- GREEN: **jal** saves PC+1 in **ra**
- ORANGE: The callee can use temporary registers (**t***i*) without saving and restoring them
- YELLOW: The caller can rely on save registers (**s***i*) without fear of callee changing them

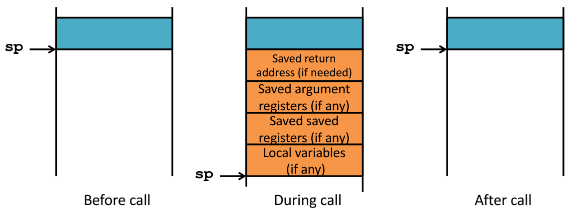
Peer Instruction

- Which statement is FALSE?
- RED: RISC-V uses **jal** to invoke a function and **jr** to return from a function
- GREEN: **jal** saves PC+1 in **ra**
- ORANGE: The callee can use temporary registers (**t***i*) without saving and restoring them
- YELLOW: 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 don't fit 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;
}
```

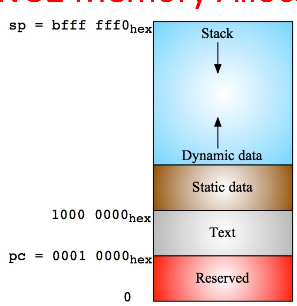

Using the Stack (2/2)

```
                 int sumSquare(int x, int y) {
sumSquare:          return mult(x,x)+ y; }
        addi sp,sp,-8 # space on stack
"push"  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`$_{hex}$
  - Stack must be aligned on 16-byte boundary (not true in examples above)
- 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

9/14/17 49