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

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

- **Compiler**

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

- **Assembler**

**Machine Language Program (RISC-V)**

**Machine Interpretation**

**Architecture Implementation**

**Logic Circuit Description (Circuit Schematic Diagrams)**

---

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

```
lw x10, 0(x12)
lw x11, 4(x12)
sw x11, 0(x12)
sw x10, 4(x12)
```

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 “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

- Processor-Memory Interface
- I/O-Memory Interfaces

- Control
- Datapath
- Registers
- Arithmetic & Logic Unit (ALU)

- Memory
  - Program
  - Bytes
  - Data

- Input
  - Enable?
  - Read/Write
  - Address
  - Write Data
  - Read Data

- Output
Example if-else Statement

• Assuming translations below, compile
  
f \rightarrow x10 \quad g \rightarrow x11 \quad h \rightarrow x12
  
i \rightarrow x13 \quad j \rightarrow x14

if (i == j)
  
f = g + h;

else
  
f = g - h;

else: bne x13, x14, Else
  
add x10, x11, x12
  
j Exit

Else: sub x10, x11, x12
  
Exit:

9/14/17
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

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

```asm
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:** \( \text{add } x_{10}, x_{11}, 4(x_{12}) \) is valid in RV32

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

**ORANGE:** \( \text{imm} \) must be multiple of 4 for \( \text{lw } x_{10}, \text{imm}(x_{10}) \) to be valid

**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)

foo.S
Assembler
foo.o
Assembler source files (text)
Assembler converts human-readable assembly code to instruction bit patterns
Machine code object files

bar.S
Assembler
bar.o
Pre-built object file libraries

Linker
lib.o
Machine code executable file

a.out
How Program is Stored

One RISC-V Instruction = 32 bits
• **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., `li rd, 13 = addi rd, x0, 13`
RISC-V Symbolic Register Names

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

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)

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

```c
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.
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: \textit{jump and link} instruction (jal)
  (really should be laj “link and jump”)
  – “link” means form an \textit{address} or \textit{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
    \begin{verbatim}
    jal FunctionLabel
    \end{verbatim}

• Return from function: \textit{jump register} instruction (jr)
  – Unconditional jump to address specified in register: \texttt{jr ra}
  – Assembler shorthand: \texttt{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

```c
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 $s_0$ and $s_1$
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)

```c
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
   - \textit{sp, gp, tp,} “saved registers” \textit{s0-s11} (\textit{s0} is also \textit{fp})

2. Not preserved across function call
   - Caller \textit{cannot} rely on values being unchanged
   - Argument/return registers \textit{a0-a7, ra,} “temporary registers” \textit{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 (tᵢ) without saving and restoring them

• YELLOW: The caller can rely on save registers (sᵢ) 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 (`ti`) without saving and restoring them
  • **YELLOW:** The caller can rely on save registers (`si`) 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

Before call

During call:
- Saved return address (if needed)
- Saved argument registers (if any)
- Saved saved registers (if any)
- Local variables (if any)

After call
Using the Stack (1/2)

• So we have a register \textit{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?

\begin{verbatim}
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
\end{verbatim}
Using the Stack (2/2)

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

**sumSquare:**

```
addi sp,sp,-8 # 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:**

```
... 9/14/17
```
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): \( \text{bfff fff0}_{\text{hex}} \)
  - Stack must be aligned on 16-byte boundary (not true in examples above)
- RV32 programs (text segment) in low end
  - \( \text{0001 0000}_{\text{hex}} \)
- static data segment (constants and other static variables) above text for static variables
  - RISC-V convention global pointer (gp) points to static
  - RV32 gp = \( \text{1000 0000}_{\text{hex}} \)
- Heap above static for data structures that grow and shrink; grows up to high addresses
RV32 Memory Allocation

\[ sp = bfff \text{fff}0_{\text{hex}} \]

\[ pc = 0001 \text{0000}_{\text{hex}} \]

Diagram showing layers:
- Stack
- Dynamic data
- Static data
- Text
- Reserved
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