CS 61C:
More RISC-V Instructions and How to Implement Functions
Administrivia

- Project 1 is in the books...
  - Yayyy! Success!!!

- Project 2 is being released
  - Start with lab 4 however:
    It is specifically designed to lead into the project

- Check-in #1: sign up now!
Pedagogical Notes...

- Yes we know this class is an annoying amount of work...
  - If we could we would make it 5 units... But we can't
  - And we have cut stuff from the end of the semester!

- Project 1: Learn C
  - We covered just about everything but unions in it...
  - Leads into CS162 and a whole host of (unfortunate) real world problems

- Project 2: Internalize Assembly
  - You are going to have to really get the calling convention right for it to work...
  - Leads into CS161 (stack smashing), 162 (context switching), 164 (compilers), etc etc etc
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

<table>
<thead>
<tr>
<th>Logical operations</th>
<th>C operators</th>
<th>Java operators</th>
<th>RISC-V instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit-by-bit AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td>Bit-by-bit OR</td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td>Bit-by-bit XOR</td>
<td>^</td>
<td>^</td>
<td>xor</td>
</tr>
<tr>
<td>Shift left logical</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right logical</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>srl</td>
</tr>
</tbody>
</table>
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)
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
  \texttt{beq register1,register2,L1}
  means: go to instruction labeled L1
  if (value in register1) == (value in register2)
  ….otherwise, go to next instruction
• \texttt{beq} stands for \textit{branch if equal}
• Other instruction: \texttt{bne} for \textit{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*)
Conditional Branches

• All are of the form \{comparison\} \{reg1\} \{reg2\} \{offset\}
  • If the condition is met...
    Add the offset (sign extended + left shifted by 1) to the program counter
  • We write the offset as a label in assembly...
    which the assembler than converts to the number

• Used for ifs, loops, etc...

• \texttt{beq} Branch Equal

• \texttt{bne} Branch Not Equal

• \texttt{blt} Branch Less Than (also \texttt{bltu})

• \texttt{bge} Branch Greater Then or Equal (also \texttt{bgeu})

• Note, there is no Branch Less Than or Equals...
  • Instead convert it into a branch greater than with shifted arguments
Example, a for loop

• `for(i = 0; i < 10; ++i) {
    ....
  }

• Assume i is in register x3

• `add x3 x0 x0`  # Set i to 0
  `j check`    # Jump to the check:
  `loop_start:`  # We'll see how this work soon

`loop_start:` ....

`addi x3 x3 1`
`check:`
`li x4 10`    # load the constant 10
`blt x3 x4 loop_start`
Remarks...

• The offset we jump to is generally handled by the assembler
  • It is relative to the program counter because this enables us to move blocks of code around without having to recompute the offset:
    We will see why later

• The offset is left-shifted by 1 (not 2) because RISC-V has an optional 16b instruction format
  • We don't use it but it allows for substantially smaller code in practice

• The li for setting the value 10...
  • Pseudo-instruction that the assembler will translate to \texttt{addi \ t0 \ x0 \ 10}
    (forgoing the need for the load-upper-immediate):
    But will work for long immediates by translating into a \texttt{lui} and \texttt{addi}
  • Yes, we could have moved this out of the loop (assuming we had a register we can use)...
    But don't bother prematurely optimizing code
More on unconditional branches…

• Only two actual 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 (the `next` instruction after the jump)
    • So we know where it came from
  • `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 (NO additional shift left!)
  • 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!
Notes on ordering/jump/branch prediction...
(Interesting Aside)

- We will see later when we get to pipelining that branches are an, umm, pain
  - A huge amount of effort is spent in processor design to guess predict where a branch will be taken or not:
    It is because it needs to start working on the next instructions even before it knows what the branch is going to do
- RISC-V in practice has some simple rules by default
  - If you ignore them it only impacts efficiency, **not correctness:**
    *So feel free to ignore them*
  - Unconditional (always taken): use `jal` always
  - Conditional likely to be taken (e.g. loops): jump **backwards**
  - Conditional not likely to be taken (e.g. if statements): jump **forwards**
Outline

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

foo.S → Assembler → foo.o
bar.S → Assembler → bar.o

Linker → lib.o

a.out
How Program is Stored

One RISC-V Instruction = 32 bits
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”
The "ABI" Conventions & Mnemonic Registers

• The "Application Binary Interface" defines our 'calling convention'
  • How to call other functions

• A critical portion is "what do registers mean by convention"
  • We have 32 registers, but how are they used

• Who is responsible for saving registers?
  • ABI defines a contract: When you call another function, when you call a function, that function promises *not* to overwrite certain registers

• We also have more convenient names based on this
  • So going forward, no more x3, x6... type notation
# The RISC-V Registers And Convention

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saved By Callee?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Always Zero</td>
<td>N/A</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return Address</td>
<td>No</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global Pointer</td>
<td>N/A</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread Pointer</td>
<td>N/A</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporary</td>
<td>No</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved Register/Frame Pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved Register</td>
<td>Yes</td>
</tr>
<tr>
<td>x10–x17</td>
<td>a0–7</td>
<td>Function Arguments/Return Values</td>
<td>No</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved Registers</td>
<td>Yes</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>No</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

• 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” in the ABI 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!
    • Means that the caller can not count on them not being mangled beyond recognition
  • *callee-saved*
    • The function invoked must restore them before returning (if used)
RISC-V Function Call Conventions

• Registers faster than memory, so use them
• \(a_0-a_7\) (\(x_{10}-x_{17}\)): eight argument registers to pass parameters and two return values (\(a_0-a_1\)) (caller saved)
  • Any more arguments should be passed on the stack
  • Technically we could return in \(a_2-a_7\) as well, but we're mostly dealing with C and not python or golang...
• \(ra\): one return address register for return to the point of origin (\(x_1\)) (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 sometimes use it because it makes things clearer how functions are translated.
    - It is however critically important in Intel x86 which does a lot more stack manipulations...
      So remember frame pointers when you get to CS161
- \( t0-t6 \) 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

```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`.
- 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
  - Think of it like a stack of plates in the dining commons...
    - If you had a reverse gravity field applied
      - *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 $s_0$ and $s_1$
Of course, we could optimize the function...

• This is a "leaf function": it calls no other function
  • So it could be made significantly more compact:
    We don't need to save ra and we can just use temporary & caller-saved registers only
  • So we could have just as easily used t0 and t1 instead...

```
leaf:
  add t0,a0,a1 # t0 = g + h
  add t1,a2,a3 # t1 = i + j
  sub a0,t0,t1 # return value (g + h) − (i + j)
  ret # ret is shorthand for jalr x0 ra
```
RISC-V book!

- “The RISC-V Reader”, David Patterson, Andrew Waterman
- Available from Amazon
- Print edition $19.99
- **Recommended, not required**
- Me? I’m cheap and just refer to the ISA documentation directly:
What If a Function Calls a Function?
Recursive Function Calls?

• Would clobber (overwrite) the 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 AND local variables
Optimized Function Convention

To reduce expensive loads and stores from saving (also called "spilling") and restoring registers, RISC-V function-calling convention divides registers into two categories:

1. Preserved across function call (\textit{callee} saved)
   - Caller can rely on values being unchanged
   - \texttt{sp, gp, tp, “saved registers” s0-s11 (s0 is also fp)}

2. Not preserved across function call (\textit{caller} saved)
   - Caller \textit{cannot} rely on values being unchanged, so if they want to keep them have to save them
   - Argument/return registers \texttt{a0-a7, ra, “temporary registers” t0-t6}
   - Plus two global registers (\texttt{gp, tp}) that can be read but shouldn't be changed within a function:
     - Act as pointers to shared and thread-specific global space for global variables
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**

Before call
- `sp`

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

After call
- `sp`
Using the Stack (1/2)

• So we have a register \texttt{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?

```c
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```
Using the Stack (2/2)

```c
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
mul:
    ...
```
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) : \texttt{bfff\_fff0}_{\text{hex}}
- RV32 programs (text segment) in low end
  - \texttt{0001\_0000}_{\text{hex}}
- static data segment (constants and other static variables) above text for static variables
  - RISC-V convention global pointer (\texttt{gp}) points to static
    - RV32 \texttt{gp} = \texttt{1000\_0000}_{\text{hex}}
- Heap above static for data structures that grow and shrink; grows up to high addresses
RV32 Memory Allocation

\[ \text{sp} = \text{bfff fff0}_{\text{hex}} \]

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

- We’ll need a local variable for \( c \)
  - Because we are calling other functions, let's put this in \( s0 \)

- We’ll need a local variable for \( n \)
  - Let's put this in \( s1 \)

- So let's form the “preamble” and “postamble”
  - What we always do on entering and leaving the function
  - So we need to save \( ra \), and the old versions of \( s0 \) and \( s1 \)
Preamble and Postamble

- **foo:**
  - addi sp sp -12  # Get stack space for 3 registers
  - sw s0 0(sp)    # Save s0 (it is callee saved)
  - sw s1 4(sp)    # Save s1 (it is callee saved)
  - sw ra 8(sp)    # Save ra (it will get overwritten)

  `{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...

```assembly
• blt a0 x0 foo_true    # if c < 0, jump to foo_true
  foo_false:            # this label ends up being ignored but
    # it is useful documentation
    mv s0 a0            # save c in s0
    li a0 8             # sizeof(struct node) (pseudoinst)
    jal malloc          # call malloc
    mv s1 a0            # save n in s1
    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
    j foo_done
  foo_true:             # return 0 in a0
    add a0 x0 x0
```

Again, we skipped a lot of optimization…

- On the leaf node \((c < 0)\) we didn’t need to save \(ra\) (or even \(s0\) & \(s1\) since we don't need to use them)
- We could get away with only one saved register..
  - Save \(c\) into \(s0\)
  - call \texttt{malloc}
  - save \(c\) into \(n[0]\)
  - calc \(c-1\)
  - save \(n\) in \(s0\)
  - recursive call
- But again, we don’t needlessly optimize…
And A Last Minute Aside:
The RISC-V 16b Instruction Format

- **Observation:** Many instructions in real programs either:
  - Use a very small immediate in address/immediate calculations
    - Jumps within a 2KiB address range accounts for almost all unconditional branches in a program
  - Use \( x_0 \), \( x_1 \) (\( ra \)), or \( x_2 \) (\( sp \)) as one of the registers in common usages
    - \( lw/sw \) with a word-aligned positive offset from the \( sp \) accounts for all the code to spill/restore registers
    - \( beq/bne \) is often a direct comparison to 0
  - The destination and the first source are identical
  - Uses one of the eight most-popular registers (s0, s1, a0-a5)
    - Most \( lw/sw \) operations use a word-aligned & positive offset that is less than \( 2^7 \) and use the common registers

- So there is also an optional 16b instruction encoding

- It is the reason for the 16b rather than 32b boundary for the immediates on branches
  - To support processors that implement this extension:
    Such processors can mix 16b and 32b instructions on an instruction-by-instruction basis
  - IMO, this is one of the real advantages that RISC-V offers:
    Dropping code size by 30%+ provides huge benefits, effectively equivalent to doubling the size of the instruction cache
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`