CS 61C: More RISC-V Instructions and How to Implement Functions
Review: What can be stored in a register?

- Registers are 32 bits
- Registers can hold any value
  - A pointer to the beginning of an array
  - A pointer to a string
  - An integer value
  - etc
Instructions We Have Learned So Far

- Addition/subtraction
  - add
  - sub
- Adding constants
  - addi
- Memory access
  - lw
  - lb
  - sw
  - sb
- Logical
  - and
  - or
  - xor
  - sll
  - slli
  - sra
  - srai
- Conditional Branching...
Conditional Branches Summary

- Used for ifs, loops, etc...
- Format: `{comparison} {reg1} {reg2} {label}
  - `beq`
  - `bne`
  - `blt, bltu`
  - `bge, bgeu`
- No “branch-less-than-or-equals” and no “branch-greater-than” ..
  - Instead convert to others by swapped arguments
    - A > B is equivalent to B < A
    - A <= B is equivalent to B >= A
Unconditional Branches

- Jump
  - j label
  - Always jump to the code located at label
If-Else Statement

```
if (a == b)
  e = c + d;
else
  e = c - d;
```

```
x10 = a
x11 = b
x12 = c
x13 = d
x14 = e
```

```
bne x10,x11,else
  add x14,x12,x13
  j done
else:
  sub x14,x12,x13
  done:
```
Loop Example

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

Assume x8 holds the address of the array

```
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: bge x11,x13,Done
lw x12,0(x9)  # x12=A[i]
add x10,x10,x12 # sum+=A[i]
addi x9,x9,4   # x9=&A[i+1]
addi x11,x11,1 # i++
j Loop
Done:
```
Program Counter

- Program Counter (PC) is a register that holds the memory address of the instruction being executed.

```assembly
if (a == b)
    e = c + d;
else
    e = c - d;
```

```
x10 = a
x11 = b
x12 = c
x13 = d
x14 = e
```

PC

```
bne x10,x11,else
add x14,x12,x13
j done
```

```
else:
sub x14,x12,x13
```

done:
Incrementing PC

• RV32 instructions are 32 bits = 4 bytes
• When we want to move to the next instruction, the processor increments PC by 4 bytes

if (a == b)  
e = c + d;
else  
e = c - d;

\[
x_{10} = a \\
x_{11} = b \\
x_{12} = c \\
x_{13} = d \\
x_{14} = e
\]

PC  \rightarrow   
\text{bne } x_{10},x_{11},\text{else} \\
\text{add } x_{14},x_{12},x_{13} \\
\text{j } \text{done} \\
\text{else: } \text{sub } x_{14},x_{12},x_{13} \\
\text{done:}
What if we want PC to execute a function at a different location?

- Jump Instructions
  - We already saw `j label`

- When we jump to a function, we need to know where to return when we are finished with the function call

- Jump instructions need to do two things
  1. Store the return address
  2. Update the value of the PC
The label that we want to jump to gets translated by the assembler to a 20-bit offset.

- We’ll learn about why it’s 20 bits on Thursday.
Jump Example

caller:

PC # do some stuff
jal x1, callee # do some more stuff

callee:

# do some stuff
# how to return?
Return Address Register

• We can choose for any register to hold the return address
• Standard convention
  • Designate register x1 to hold the return address
  • x1 has an alternate name = ra

jal ra, L1
Return Address Register

- When we jump to a function, we need a return address
- When we jump because of a loop or branch, we don’t need a return address
- To avoid saving the return address, we can specify x0 as the destination register

jal x0, L1
Recall: Pseudo Instructions

- Instructions that are available for the programmer’s use but are not implemented in the ISA
- These instructions are translated by the assembler to real RISC-V instructions
- Example
  - RISC-V ISA doesn’t define bgt to avoid redundancy; however there is a bgt pseudo instruction
  - `bgt x2 x3 foo` → `blt x3 x2 foo`
Jump Pseudo Instruction

```
jal x0, L1  \rightarrow  j L1
```

\[ PC = PC + \text{offset} \]

Return address not saved
Recall: Jump Example

\[
\text{if} \ (a == b) \\
\quad e = c + d; \\
\text{else} \\
\quad e = c - d;
\]

\[
\text{bne } x10, x11, \text{else} \\
\text{add } x14, x12, x13 \\
\text{j done} \\
\text{else: sub } x14, x12, x13 \\
\text{done:}
\]

\[
x10 = a \\
x11 = b \\
x12 = c \\
x13 = d \\
x14 = e
\]
JALR

• With only a 20-bit offset, we cannot jump to everywhere in memory, so we have another instruction:

\[
\text{jalr } rd, rs, \text{ imm}
\]

- \( rd = \text{return address} \)
- \( \text{PC} = [rs] + \text{imm} \)

- \( rd = \text{register where the return address will be stored} \)
- \( \text{Immediate value to be added to the base register} \)
JALR

• When we want to return from a function
  • Our return address is going to be stored in a register
  • We don’t need to save another return address

jalr rd, rs, imm

jalr x0, ra, 0

rd = destination register
rs = source register
Jump Example

caller:

# do some stuff
jal ra, callee
# do some more stuff

callee:

# do some stuff
jalr x0, ra, 0
Jump Register Pseudo Instruction

jalr x0, rs, 0 → jr rs
PC = [rs]  (rs = source register)
Return address not saved

jalr x0, ra, 0 → jr ra → ret
PC = [ra]
Return Address not saved
Jump Summary

• Jump-and-link
  • jal rd, label
    • jal x0, label -> j label

• Jump-and-link-register
  • jalr rd, rs, imm (rs = source register)
    • jalr x0, rs, 0 -> jr rs
    • jalr x0, ra, 0 -> jr ra -> ret
Jump Example

caller:

PC

# do some stuff
jal ra, callee
# do some more stuff

callee:

# do some stuff
ret
Saving Registers

• When we call another function, what happens to the values that are stored in the registers?
  • The other function needs to use those registers for its computations, so it might overwrite our values

• How to prevent this?
  • One option: We can save all of the registers we are using before we call a function and then restore the values

• Where can we save these values?
  • The stack
Allocating Space on Stack

- C has two storage classes: automatic and static
  - **Automatic** variables are local to a 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
RV32 Memory Allocation

- Stack
- Dynamic data
- Static data
- Text
- Reserved
Stack Pointer (SP)

- A register that holds the memory address of the location of the last item placed on the stack (x2)
Stack Pointer

```
\text{bfff fff0}_{\text{hex}}
```

- Stack
- Dynamic data
- Static data
- Text
- Reserved

SP
Stack Frame
Stack Pointer

- When you place an item on the stack, you decrement the stack pointer
  - PUSH
- When you take an item off the stack, you increment the stack pointer
  - POP
How to move the stack pointer?

- Making room to store something
  - Decrement the stack by x bytes
  - `addi sp, sp, -x`

- Removing something from the stack
  - Increment the stack by x bytes
  - `addi sp, sp, x`
How to Store a Value on the Stack

• If register x5 contains the data that we want to store on the stack

```
addi sp, sp, -4
sw x5, 0(sp)
```
When we call another function, what happens to the values that are stored in the registers?

- The other function needs to use those registers for its computations, so it might overwrite our values.

How to prevent this?

- One option: We can save all of the registers we are using before we call a function and then restore the values.

Where can we save these values?

- The stack.
Saving Registers

- We can save all of our registers before we call a function
  - All registers would be saved by the caller
- Another thing we can do is save all the registers before we use them
  - All registers would be saved by the callee
- Need to standardize how we do this
  - Meet somewhere in the middle, I’ll save some and you save some
  - The registers that are saved by the caller and callee are specified by the calling convention
Calling Convention

- Temporary registers
  - Saved by caller
- Saved Registers
  - Saved by callee
## Calling Convention

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Description</th>
<th>Saved by</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>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack Pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8–x9</td>
<td>s0–s1</td>
<td>Saved Register</td>
<td>Callee</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>
Argument Registers

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

arguments

return value
Argument and return registers

- Our functions need to have a place where they can expect the arguments and return values to be.
- We will set aside registers x10-x17 to be argument registers.
  - New names => a0-a7
  - a0 and a1 will also serve as return value registers.
- If the caller has some temporary values in the registers that it wants to use after making a function call, it must save those values.
# Calling Convention

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Description</th>
<th>Saved by</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>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>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>Caller</td>
</tr>
<tr>
<td>x8–x9</td>
<td>s0–s1</td>
<td>Saved Register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10–x17</td>
<td>a0–7</td>
<td>Function Arguments/Return Values</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>
Example

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

add t0, a0, a1  # t0 = g + h
add t1, a2, a3  # t1 = i + j
sub a0, t0, t1  # f = (g + h) - (i + j)

jr ra  # return to calling function
Example

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

```assembly
addi sp, sp, -8     # adjust stack to store 2 items
sw s1, 4(sp)       # save s1 because we are overwriting it
sw s0, 0(sp)       # save s0 because we are overwriting it

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

lw s0, 0(sp)       # restore s0
lw s1, 4(sp)       # restore s1
addi sp, sp, 8     # adjust stack to delete 2 items
jr ra              # return to calling function
```
Stack Before, During, After Function

- Need to save old values of $s0$ and $s1$

Before call

During call: Saved $s1$
Saved $s0$

After call: Saved $s1$
Saved $s0$
Example

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

int foo(int x) {
    // do stuff
    int x = bar(g, h, i, j);
    return x * 2;
}

int main() {
    // do stuff
    foo(x);
    // do stuff
```
Example

```c
int foo(int g, int h, int i, int j) {
   // do stuff
   int x = bar(g, h, i, j)
   return x * 2;
}
```

In `foo`, `g`, `h`, `i`, and `j` are in `s0-s3`

---

```assembly
# do stuff (code omitted)
# save ra
addi sp, sp, -4
sw ra, 0(sp)
# set up argument registers
add a0, s0, x0
add a1, s1, x0
add a2, s2, x0
add a3, s3, x0
jal bar
# restore ra
lw ra, 0(sp)
addi sp, sp, 4
slli a0, a0, 1
jr ra
```

Set up for function call (Prologue)

Tear down from function call (Epilogue)
Six Fundamental Steps in Calling a Function

1. Put parameters in a place where function can access them
   • put parameters in argument registers

2. Transfer control to function
   • With a jump instruction

3. Acquire (local) storage resources needed for function
   • make room for local variables on stack

4. Perform desired task of the function

5. Put result value in a place where calling code can access it
   • a0-a1 register

6. Return control to point of origin
   • ret
More Instructions!

- See the RISC-V Green Card
  - [https://inst.eecs.berkeley.edu/%7Ecs61c/resources/riscvcard.pdf](https://inst.eecs.berkeley.edu/%7Ecs61c/resources/riscvcard.pdf)