CS 61C:
Great Ideas in Computer Architecture
More Function Calls &
RISC-V Instruction Formats
Administrivia

- Project 1 in the books!
- Project 2 spec released: RISC-V Assembly Language Programming
  - Deadlines: part A - Mon 10/4, part B - Mon 10/11
- Assignments Due this Week:
  - Homework 3: 9/24 (Friday)
  - Lab 3, RISC-V Assembly Language, due 9/24
    - Reminder: lab checkoffs will end promptly at 4PM on Fridays!
- Upcoming Assignments:
  - Homework 4 released, due next Friday 10/1
Administrivia

• **Midterm Exam rescheduled:**
• Based on poll - new date/time:
• Tuesday, October 19th, 7:00 PM - 9:00 PM PT
  • reply to piazza poll on scope
  • in-person and remote option
  • more details later
Outline

- More Function Calls
- RISC-V Instruction Formats
Outline

• More Function Calls
• RISC-V Instruction Formats
Review: To call a Function

- Use `jal` instruction to transfer control to function (callee)
  - Register convention:
    - return address is saved in register `ra`
    - arguments get passed in and return values in `a0-a7`
  - Use `jalr ra` to return to caller (`ret`)
- What If a Function Calls a Function?
  - Note: this could mean a function calling itself - *recursion*.
  - 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;
}
```

- Function called `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 registers in addition to ra.

• Again, use the stack for this.

• 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
RV32 Memory Allocation

- 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
  - 0001_0000_{\text{hex}}
- \textit{static data segment} (constants and other static variables) above text for static variables
  - RISC-V convention \texttt{global pointer} (gp) points to static
  - RV32 gp = 1000_0000_{\text{hex}}
- Heap above static for data structures that grow and shrink; grows up to high addresses
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
- *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 \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)

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

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
```

"push"

"pop"
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`
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…

```
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_exit
foo_true:
    add a0,x0,x0    # return 0 in a0
```
We skipped some possible optimizations …

- 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

- For us, our version is good enough.
Outline

• More Function Calls
• RISC-V Instruction Formats
ENIAC (U.Penn., 1946)  
First Electronic General-Purpose Computer

- Blazingly fast  
  (multiply in 2.8ms!)
  - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches
Big Idea: Stored-Program Computer

- Instructions are represented as bit patterns - can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don’t have to rewire computer (days)
- Known as the “von Neumann” computers after widely distributed tech report on EDVAC project
  - Wrote-up discussions of Eckert and Mauchly
  - Anticipated earlier by Turing and Zuse

First Draft of a Report on the EDVAC
by
John von Neumann
Between the
United States Army Ordnance Department
and the
University of Pennsylvania
Moore School of Electrical Engineering
University of Pennsylvania
June 30, 1945
EDSAC (Cambridge, 1949)
First General Stored-Program Computer

- Programs held as “numbers” in memory
- 35-bit binary 2’s complement words
Consequence #1: Everything Has a Memory Address

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - Both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
- One special register keeps address of instruction being executed: “Program Counter” (PC)
  - Basically a pointer to memory
  - Intel calls it Instruction Pointer (a better name)
Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for phones and PCs, etc.
- New machines in the same family want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to "backward-compatible" instruction set evolving over time
  - Selection of Intel 8088 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today
Instructions as Numbers (1/2)

• Most data we work with is in words (32-bit chunks):
  • Each register holds a word
  • \texttt{lw} and \texttt{sw} both access memory one word at a time

• So how do we represent instructions?
  • Remember: Computer only represents 1s and 0s, so assembler string “\texttt{add x10, x11, x0}” is meaningless to hardware
  • RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also
    • Same 32-bit instruction definitions used for RV32, RV64, RV128
Instructions as Numbers (2/2)

• Divide 32-bit instruction word into “fields”
• Each field tells processor something about instruction
• We could define different set of fields for each instruction, but for hardware simplicity, group possible instructions into six basic types of instruction formats:
  • **R-format** for register-register arithmetic/logical operations
  • **I-format** for register-immediate ALU operations and loads
  • **S-format** for stores
  • **B-format** for branches
  • **U-format** for 20-bit upper immediate instructions
  • **J-format** for jumps
## Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>funct7</td>
<td></td>
<td></td>
<td>rs1</td>
<td>funct3</td>
<td></td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td>rs1</td>
<td>funct3</td>
<td></td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm[31:12]</td>
</tr>
</tbody>
</table>

- **R-type**
- **I-type**
- **S-type**
- **B-type**
- **U-type**
- **J-type**
R-Format Instruction Layout Annotation

<table>
<thead>
<tr>
<th>Field's bit positions</th>
<th>Name of field</th>
<th>Number of bits in field</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>funct7</td>
<td></td>
</tr>
<tr>
<td>25 24</td>
<td>rs2</td>
<td></td>
</tr>
<tr>
<td>20 19</td>
<td>rs1</td>
<td></td>
</tr>
<tr>
<td>15 14</td>
<td>funct3</td>
<td></td>
</tr>
<tr>
<td>12 11</td>
<td>rd</td>
<td></td>
</tr>
<tr>
<td>7 6</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- This example: 32-bit instruction word divided into six fields of differing numbers of bits each field: $7 + 5 + 5 + 3 + 5 + 7 = 32$
- In this case:
  - **opcode** is a 7-bit field that lives in bits 0-6 of the instruction
  - **rs2** is a 5-bit field that lives in bits 20-24 of the instruction
  - etc.
R-Format Instructions opcode/funct fields

- **opcode**: partially specifies which instruction it is
  - Note: This field is contains $0110011_{\text{two}}$ for all R-Format register-register arithmetic/logical instructions
- **funct7+funct3**: combined with opcode, these two fields describe what operation to perform
- Question: Why aren’t opcode and funct7 and funct3 a single 17-bit field?
  - We’ll answer this later
### R-Format Instructions register specifiers

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31)
- **rs1** (Source Register #1): specifies register containing first operand
- **rs2**: specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
R-Format Example

- RISC-V Assembly Instruction:
  \[ \text{add} \quad x18,x19,x10 \]

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

ADD  \quad rs2=10 \quad rs1=19  \quad ADD  \quad rd=18  \quad \text{Reg-Reg OP}
### All RV32 R-format instructions

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
</tr>
</tbody>
</table>

Encoding in funct7 + funct3 selects particular operation
I-Format Instructions

• What about instructions with immediates?
  • Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
  • 5-bit field only represents numbers up to the value 31: would like immediates to be much larger
• Define another instruction format that is mostly consistent with R-format
  • Note: if instruction has immediate, then uses at most 2 registers (one source, one destination)
I-Format Instruction Layout

• Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, \(\text{imm}[11:0]\)

• Remaining field format \((\text{rs1, funct3, rd, opcode})\) same as before

• \(\text{imm}[11:0]\) can hold values in range \([-2048_{\text{ten}}, +2047_{\text{ten}}]\)

• Immediate is \textit{always} sign-extended to 32-bits before use in an arithmetic/logic operation

• We’ll later see how to handle immediates > 12 bits
I-Format Example

- **RISC-V Assembly Instruction:**
  
  \[
  \text{addi} \quad x15, x1, -50
  \]

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( \text{imm} = -50 \)
- \( \text{rs1} = 1 \)
- \( \text{ADD} \)
- \( \text{rd} = 15 \)
- \( \text{OP-Imm} \)
### All RV32 I-format Arithmetic/Logical Instructions

<table>
<thead>
<tr>
<th>imm</th>
<th>funct3</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>rs1</td>
<td>010</td>
</tr>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>rs1</td>
<td>011</td>
</tr>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>rs1</td>
<td>100</td>
</tr>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>rs1</td>
<td>110</td>
</tr>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>rs1</td>
<td>111</td>
</tr>
<tr>
<td>\texttt{0000000}</td>
<td>shamt</td>
<td>rs1</td>
</tr>
<tr>
<td>\texttt{0000000}</td>
<td>shamt</td>
<td>rs1</td>
</tr>
<tr>
<td>\texttt{0100000}</td>
<td>shamt</td>
<td>rs1</td>
</tr>
</tbody>
</table>

One of the higher-order immediate bits is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI).

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions).
Load Instructions are also I-Type

\[ rd = M[rs1+imm][0:31] \]

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>op code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>width</td>
<td>dest</td>
<td>LOAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The 12-bit signed immediate is added to the base address in register \( rs1 \) to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register \( rd \)
I-Format Load Example

- RISC-V Assembly Instruction:
  \[ \text{lw} \ x14, \ 8(x2) \]

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

imm=+8 \quad rs1=2 \quad \text{LW} \quad rd=14 \quad \text{LOAD}
All RV32 Load Instructions

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0000011</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0000011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0000011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0000011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0000011</td>
</tr>
</tbody>
</table>

- LBU is “load unsigned byte”
- LH is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- LHU is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
- There is no LWU in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register
S-Format Used for Stores

- Store needs to read two registers, \texttt{rs1} for base memory address, and \texttt{rs2} for data to be stored, as well as need immediate offset!
- Can’t have both \texttt{rs2} and immediate in same place as other instructions!
- Note that stores don’t write a value to the register file, \textit{no rd}!
- RISC-V design decision is move low 5 bits of immediate to where \texttt{rd} field was in other instructions – keep \texttt{rs1/rs2} fields in same place.
S-Format Example

- RISC-V Assembly Instruction:

\texttt{sw x14, 8(x2)}

\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{imm[11:5]} & \text{rs2} & \text{rs1} & \text{funct3} & \text{imm[4:0]} & \text{opcode} \\ \hline
7 & 5 & 5 & 3 & 5 & 7 \\ \hline
\end{array}

\text{offset[11:5]} \quad \text{src} \quad \text{base} \quad \text{width} \quad \text{offset[4:0]} \quad \text{STORE}

\begin{array}{cccccccc}
0000000 & 01110 & 00010 & 010 & 01000 & 0100011 \\
\end{array}

\begin{align*}
\text{offset[11:5]} \quad \text{rs2}=14 \quad \text{rs1}=2 \quad \text{SW} \quad \text{offset[4:0]} \quad \text{STORE} \\
=0 \quad =8 \\
\end{align*}

\begin{array}{|c|c|}
\hline
00000000 & 01000 \\
\hline
\end{array}

combined 12-bit offset = 8
## All RV32 Store Instructions

|-----------|-----|-----|-----|---------|---------|

The `funct3` field encodes size.
### RISC-V Conditional Branches

- E.g., `BEQ x1, x2, Label`
- Branches read two registers but don’t write a register (similar to stores)
- How to encode the label, i.e., where to branch to?
- We use an immediate to encode *PC relative offset*
  - If we don’t take the branch:
    \[ \text{PC} = \text{PC} + 4 \] (i.e., next instruction)
  - If we do take the branch:
    \[ \text{PC} = \text{PC} + \text{immediate} \]
Branching Instruction Usage

- Branches typically used for loops (if-else, while, for)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- Recall: Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)
PC-Relative Addressing

- **PC-Relative Addressing:** Use the immediate field as a two’s-complement offset relative to PC
  - Branches generally change the PC by a small amount
  - With the 12-bit immediate, could specify $\pm 2^{11}$ byte address offset from the PC
- To improve the reach of a single branch instruction, *in principle*, could multiply the byte immediate by 4 before adding to PC (*instructions are 4 bytes and word aligned*).
- This would allow one branch instruction to reach $\pm 2^{11} \times 32$-bit instructions either side of PC
  - Four times greater reach than using byte offset
  - However …
RISC-V Feature, $n \times 16$-bit instructions

- However, extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 2-Bytes in length.
- To enable this, RISC-V always scales the branch immediate by 2 bytes - even when there are no 16-bit instructions.
- This means for us,
  1. the low bit of the stored immediate value will always be 0)
  2. The immediate is left-shifted by 1 before adding to PC
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$-bit instructions either side of PC.
RISC-V B-Format for Branches

- B-format is mostly same as S-Format, with two register sources (rs1/r2s) and a 12-bit immediate
- But now immediate represents the branch offset in units of half-words. To convert to units of Bytes, left-shift by 1.
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
  - Thus the \texttt{imm[12:1]} in the total encoding, compared with \texttt{imm[11:0]} in the I-type encodings
Branch Example, determine offset

- **RISC-V Code:**
  ```riscv
  Loop:   beq  x19, x10, End
          add  x18, x18, x10
          addi x19, x19, -1
          j    Loop
  End:    # target instruction
  ```

- **Branch offset** = 4 × 32-bit instructions = 16 bytes
- *(Branch with offset of 0, branches to itself)*
Branch Example, encode offset

- RISC-V Code:

  Loop:  
  ```
  beq   x19, x10, End
  add   x18, x18, x10
  addi  x19, x19, -1
  j     Loop
  ```

  End:  # target instruction

  Offset = 16 bytes = 8x2

<table>
<thead>
<tr>
<th>imm</th>
<th>rs2=10</th>
<th>rs1=19</th>
<th>BEQ</th>
<th>imm</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>01010</td>
<td>10011</td>
<td>000</td>
<td>1100011</td>
<td>1100011</td>
<td></td>
</tr>
</tbody>
</table>
Branch Example, complete encoding

```
beq   x19, x10, offset = 16 bytes
```

13-bit immediate, imm[12:0], with value 16

```
0 000000 01010 10011 000 1000 0 1100011
```


imm[10:5] rs2=10  rs1=19  BEQ  BRANCH
## All RISC-V Branch Instructions

|--------------|------|------|-----|-------------|---------|

BEQ, BNE, BLT, BGE, BLTU, BGEU
Questions on PC-addressing

• Does the value in branch immediate field change if we move the code?
  • If moving individual lines of code, then yes
  • If moving all of code, then no (because PC-relative offsets)

• What do we do if destination is > 2^{10} instructions away from branch?
  • Other instructions save us
    • `beq x10,x0,far`  \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{beq x10,x0,far} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
    \[ \text{bne x10,x0,next} \]
    \[ \text{# next instr} \rightarrow \text{bne x10,x0,next} \]
## U-Format for “Upper Immediate” instructions

<table>
<thead>
<tr>
<th>imm[31:12]</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>U-immediate[31:12]</td>
<td>dest</td>
<td>LUI</td>
</tr>
<tr>
<td>U-immediate[31:12]</td>
<td>dest</td>
<td>AUIPC</td>
</tr>
</tbody>
</table>

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, rd
- Used for two instructions
  - LUI – Load Upper Immediate, rd = imm << 12
  - AUIPC – Add Upper Immediate to PC, rd = PC + (imm << 12)
LUI to create long immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

LUI x10, 0x87654  # x10 = 0x87654000
ADDI x10, x10, 0x321 # x10 = 0x87654321
One Corner Case

How to set 0xDEADBEEF?

```
LUI x10, 0xDEADB  # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF# x10 = 0xDEAD
```

ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract -1 from upper 20 bits
Solution

How to set 0xDEADBEEF?

\[
\text{LUI } x10, \ 0x\text{DEADC} \quad \# \ x10 = 0x\text{DEADC000}
\]
\[
\text{ADDI } x10, \ x10, \ 0xEEF \# \ x10 = 0xDEADBEEF
\]

Pre-increment the value placed in upper 20 bits, if sign bit will be set on immediate in lower 12 bits.

Assembler pseudo-op handles all of this:

\[
\text{li } x10, \ 0xDEADBEEF \ # \text{ Creates two instructions}
\]
# Uses of JAL

# j pseudo-instruction
j Label = jal x0, Label # Discard return address

# Call function within $2^{18}$ instructions of PC
jal ra, FuncName
J-Format for Jump Instructions

- JAL saves PC+4 in register rd (the return address)
  - Assembler “j” jump is pseudo-instruction, uses JAL but sets rd=x0 to discard return address
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
  - $\pm 2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

offset[20:1] dest JAL
Uses of JALR

# ret and jr psuedo-instructions
ret = jr ra = jalr x0, ra, 0

# Call function at any 32-bit absolute address
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>

# Jump PC-relative with 32-bit offset
auipc x1, <hi20bits>  # Adds upper immediate value to
                      # and places result in x1
jalr x0, x1, <lo12bits>  # Same sign extension trick needed
                          # as LUI
JALR Instruction (I-Format)

- **JALR** \( r_d \), \( r_s \), immediate
  - Writes PC+4 to \( r_d \) (return address)
  - Sets PC = \( r_s \) + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes

### Table

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>20 19</th>
<th>15 14</th>
<th>12</th>
<th>11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>12</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>5</td>
<td>base</td>
<td>0</td>
<td>dest</td>
<td>JALR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>[31:30]</td>
<td>Instruction function</td>
</tr>
<tr>
<td>rs2</td>
<td>[25:24]</td>
<td>Source register 2</td>
</tr>
<tr>
<td>rs1</td>
<td>[21:20]</td>
<td>Source register 1</td>
</tr>
<tr>
<td>funct3</td>
<td>[19:15]</td>
<td>Function code</td>
</tr>
<tr>
<td>opcode</td>
<td>[8:0]</td>
<td>Instruction code</td>
</tr>
</tbody>
</table>

### R-type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>[31:24]</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

### I-type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1</td>
<td>[31:24]</td>
<td>Source register 1</td>
</tr>
<tr>
<td>funct3</td>
<td>[23:19]</td>
<td>Function code</td>
</tr>
<tr>
<td>rd</td>
<td>[18:12]</td>
<td>Destination register</td>
</tr>
<tr>
<td>opcode</td>
<td>[11:0]</td>
<td>Instruction code</td>
</tr>
</tbody>
</table>

### S-type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs2</td>
<td>[31:25]</td>
<td>Source register 2</td>
</tr>
<tr>
<td>rs1</td>
<td>[24:20]</td>
<td>Source register 1</td>
</tr>
<tr>
<td>funct3</td>
<td>[19:15]</td>
<td>Function code</td>
</tr>
<tr>
<td>imm[4:0]</td>
<td>[14:12]</td>
<td>Immediate</td>
</tr>
<tr>
<td>opcode</td>
<td>[11:0]</td>
<td>Instruction code</td>
</tr>
</tbody>
</table>

### B-type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs2</td>
<td>[31:25]</td>
<td>Source register 2</td>
</tr>
<tr>
<td>rs1</td>
<td>[24:20]</td>
<td>Source register 1</td>
</tr>
<tr>
<td>funct3</td>
<td>[19:15]</td>
<td>Function code</td>
</tr>
<tr>
<td>imm[4:1]</td>
<td>[14:13]</td>
<td>Immediate</td>
</tr>
<tr>
<td>opcode</td>
<td>[11:0]</td>
<td>Instruction code</td>
</tr>
</tbody>
</table>

### U-type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>[31:12]</td>
<td>Immediate</td>
</tr>
<tr>
<td>rd</td>
<td>[11:0]</td>
<td>Destination register</td>
</tr>
<tr>
<td>opcode</td>
<td>[11:0]</td>
<td>Instruction code</td>
</tr>
</tbody>
</table>

### J-type

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[20]</td>
<td>[20]</td>
<td>Immediate</td>
</tr>
<tr>
<td>imm[10:1]</td>
<td>[19:11]</td>
<td>Immediate</td>
</tr>
<tr>
<td>imm[19:12]</td>
<td>[18:0]</td>
<td>Immediate</td>
</tr>
<tr>
<td>rd</td>
<td>[11:0]</td>
<td>Destination register</td>
</tr>
<tr>
<td>opcode</td>
<td>[11:0]</td>
<td>Instruction code</td>
</tr>
</tbody>
</table>
## Complete RV32I ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Opcode</th>
<th>Funct</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUI</td>
<td>000000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>AUIPC</td>
<td>000000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>JAL</td>
<td>010000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>JALR</td>
<td>010000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>BEQ</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>000</td>
</tr>
<tr>
<td>BNE</td>
<td>010000</td>
<td>rs2</td>
<td>rsl</td>
<td>000</td>
</tr>
<tr>
<td>BLT</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>010</td>
</tr>
<tr>
<td>BGE</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>011</td>
</tr>
<tr>
<td>BLTU</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>101</td>
</tr>
<tr>
<td>BGEU</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>111</td>
</tr>
<tr>
<td>LB</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>101</td>
</tr>
<tr>
<td>LH</td>
<td>010000</td>
<td>rs2</td>
<td>rsl</td>
<td>111</td>
</tr>
<tr>
<td>LW</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>001</td>
</tr>
<tr>
<td>LBU</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>010</td>
</tr>
<tr>
<td>LHU</td>
<td>000000</td>
<td>rs2</td>
<td>rsl</td>
<td>101</td>
</tr>
<tr>
<td>SB</td>
<td>000000</td>
<td>pred</td>
<td>succ</td>
<td>000</td>
</tr>
<tr>
<td>SH</td>
<td>000000</td>
<td>pred</td>
<td>succ</td>
<td>000</td>
</tr>
<tr>
<td>SW</td>
<td>000000</td>
<td>pred</td>
<td>succ</td>
<td>000</td>
</tr>
<tr>
<td>ADDI</td>
<td>csr</td>
<td>rsl</td>
<td>011</td>
<td>rd</td>
</tr>
<tr>
<td>SLTI</td>
<td>csr</td>
<td>rsl</td>
<td>011</td>
<td>rd</td>
</tr>
<tr>
<td>SLLTU</td>
<td>csr</td>
<td>rsl</td>
<td>101</td>
<td>rd</td>
</tr>
<tr>
<td>XOIR</td>
<td>csr</td>
<td>zimm</td>
<td>101</td>
<td>rd</td>
</tr>
<tr>
<td>ORI</td>
<td>csr</td>
<td>zimm</td>
<td>110</td>
<td>rd</td>
</tr>
<tr>
<td>ANDI</td>
<td>csr</td>
<td>zimm</td>
<td>111</td>
<td>rd</td>
</tr>
</tbody>
</table>

**Not in CS61C**

---

**62**

---

**Computer Science 61C Fall 2021**

Wawrzynek and Weaver