Logical Instructions
RV32 So Far…

- **Add/sub**
  - add rd, rs1, rs2
  - sub rd, rs1, rs2

- **Add immediate**
  - addi rd, rs1, imm

- **Load/store**
  - lw rd, rs1, imm
  - lb rd, rs1, imm
  - lbu rd, rs1, imm
  - sw rs1, rs2, imm
  - sb rs1, rs2, imm

- **Branching**
  - beq rs1, rs2, Label
  - bne rs1, rs2, Label
  - bge rs1, rs2, Label
  - blt rs1, rs2, Label
  - bgeu rs1, rs2, Label
  - bltu rs1, rs2, Label
  - j Label
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></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>
RISC-V Logical Instructions

- Always two variants
  - Register: \texttt{and x5, x6, x7} \# $x_5 = x_6 \& x_7$
  - Immediate: \texttt{andi x5, x6, 3} \# $x_5 = x_6 \& 3$

- Used for ‘masks’
  - \texttt{andi} with \texttt{0000 00FF}_{\text{hex}} isolates the least significant byte
  - \texttt{andi} with \texttt{FF00 0000}_{\text{hex}} isolates the most significant byte
**No NOT in RISC-V**

- There is no logical NOT in RISC-V
  - Use `xor` with `11111111_{two}`
  - Remember - simplicity...
Shift Left Logical (sll) and immediate (slli):

- **slli x11,x12,2 #x11=x12<<2**
- Store in x11 the value from x12 shifted by 2 bits to the left (they fall off end), inserting 0’s on right; << in C.
- Before: \[0000 \ 0002_{\text{hex}}\]
  \[0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0010_{\text{two}}\]
- After: \[0000 \ 0008_{\text{hex}}\]
  \[0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1000_{\text{two}}\]
- What arithmetic effect does shift left have?

- **Shift Right: srl is opposite shift; >>**
Arithmetic Shifting

- Shift right arithmetic (sra, srai) moves $n$ bits to the right (insert high-order sign bit into empty bits)
- For example, if register $x10$ contained $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{\text{two}} = -25_{\text{ten}}$
- If execute $\text{srai}\ x10, x10, 4$, result is: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$
  - Unfortunately, this is NOT same as dividing by $2^n$
    - Fails for odd negative numbers
    - C arithmetic semantics is that division should round towards 0
A Bit About Machine Program
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
Linker
a.out

bar.S
Assembler
bar.o
lib.o

foo.o
bar.o
lib.o

a.out
How Program is Stored

One RISC-V Instruction = 32 bits
Program Execution

- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates PC (default add +4 bytes to PC, to move to next sequential instruction; branches, jumps alter)

- PC (program counter) is a register internal to the processor that holds byte address of next instruction to be executed
Helpful RISC-V Assembler Features

- **Symbolic register names**
  - E.g., `a0–a7` for argument registers (`x10–x17`) for function calls
  - 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`
  - E.g., `nop = addi x0, x0, 0`
RISC-V
Function Calls
C Functions

main() {
    int i, j, k, m;
    ...
    i = mult(j, k); ...
    m = mult(i, i); ...
}

/* really dumb mult function */
int mult(int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product = product + mcand;
        mlier = mlier - 1;
    }
    return product;
}

What information must compiler/programmer keep track of?

What instructions can accomplish this?
Six Fundamental Steps in Calling a Function

1. Put **arguments** 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 **return value** in a place where calling code can access it and restore any registers you used; release local storage
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)
- Also s0–s1 (x8–x9) and s2–s11 (x18–x27): saved registers (more about those later)
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.

C

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

... `sum(a,b);` ... /* `a,b:` s0,s1 */
Instruction Support for Functions (2/4)

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

address (shown in decimal)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>mv a0,s0</td>
<td># x = a</td>
</tr>
<tr>
<td>1004</td>
<td>mv a1,s1</td>
<td># y = b</td>
</tr>
<tr>
<td>1008</td>
<td>addi ra,zero,1016</td>
<td>#ra=1016</td>
</tr>
<tr>
<td>1012</td>
<td>j sum</td>
<td>#jump to sum</td>
</tr>
<tr>
<td>1016</td>
<td>...</td>
<td># next inst.</td>
</tr>
<tr>
<td>2000</td>
<td>sum: add a0,a0,a1</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>jr ra</td>
<td>#new instr.&quot;jump reg&quot;</td>
</tr>
</tbody>
</table>

RISC-V
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 \texttt{jr} here? Why not use \texttt{j}?

• Answer: \texttt{sum} might be called by many places, so we can’t return to a fixed place. The calling proc to \texttt{sum} must be able to say “return here” somehow.

RISC-V

... sum: add a0,a0,a1
2004 \texttt{jr} ra #new instr. “jump reg”
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

  \[
  \text{jal FunctionLabel}
  \]

- **Return from function:** *jump register* instruction (**jr**)
  - Unconditional jump to address specified in register: **jr ra**
  - Assembler shorthand: **ret = jr ra**
Actually, only two instructions:

- jal rd, Label – jump-and-link
- jalr rd, rs, imm – jump-and-link register

j, jr and ret are pseudoinstructions!
- j: jal x0, Label