RISC-V Conditional Branches

- E.g., `beq x1, x2, Label`
- Branches read two registers but don’t write to a register (similar to stores)
- How to encode label, i.e., where to branch to?
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: Use the **immediate** field as a two’s-complement offset to PC
- Branches generally change the PC by a small amount
- Can specify $\pm 2^{11}$ ‘unit’ addresses from the PC
- (We will see in a bit that we can encode 12-bit offsets as immediates)

Why not use byte as a unit of offset from PC?
- Because instructions are 32-bits (4-bytes)
- We don’t branch into middle of instruction
Scaling Branch Offset

- One idea: To improve the reach of a single branch instruction, multiply the offset by four bytes before adding to PC.
- 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.
Branch Calculation

- If we don’t take the branch:
  \[ PC = PC + 4 \] (i.e., next instruction)

- If we do take the branch:
  \[ PC = PC + \text{immediate} \times 4 \]

Observations:
- \text{immediate} is number of instructions to jump (remember, specifies words) either forward (+) or backwards (−)
RISC-V Feature, $n \times 16$-bit Instructions

- Extensions to RISC-V base ISA support 16-bit compressed instructions and also variable-length instructions that are multiples of 16-bits in length.
- To enable this, RISC-V scales the branch offset by 2 bytes even when there are no 16-bit instructions.
- Reduces branch reach by half and means that $\frac{1}{2}$ of possible targets will be errors on RISC-V processors that only support 32-bit instructions (as used in this class).
- RISC-V conditional branches can only reach $\pm 2^{10} \times 32$-bit instructions on either side of PC.
### RISC-V B-Format for Branches

<table>
<thead>
<tr>
<th>31</th>
<th>30</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>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
</table>

- B-format is mostly same as S-Format, with two register sources \((rs1/\text{rs2})\) and a 12-bit immediate \(\text{imm}[12:1]\)
- But now immediate represents values -4096 to +4094 in 2-byte increments
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)
Branch Example, Determine Offset

- RISC-V Code:

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

- Branch offset =

  \[4 \times 32\text{-bit instructions} = 16 \text{ bytes}\]

- (Branch with offset of 0, branches to itself)
Branch Example, Determine Offset

- **RISC-V Code:**

  Loop: \texttt{beq x19,x10,End}
  \texttt{add x18,x18,x10}
  \texttt{addi x19,x19,-1}
  \texttt{j Loop}

  End: \# target instruction

Count instructions from branch:
1. \texttt{beq x19,x10,End}
2. \texttt{add x18,x18,x10}
3. \texttt{addi x19,x19,-1}
4. \texttt{j Loop}

<table>
<thead>
<tr>
<th>???????</th>
<th>01010</th>
<th>10011</th>
<th>000</th>
<th>??????</th>
<th>1100011</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm</td>
<td>rs2=10</td>
<td>rs1=19</td>
<td>BEQ</td>
<td>imm</td>
<td>BRANCH</td>
</tr>
</tbody>
</table>
Branch Example, Determine 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
  
  \[= 8 \times 2\]

  
<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>01000</td>
<td>01010</td>
<td>10011</td>
<td>000</td>
<td>????</td>
<td>1100011</td>
</tr>
</tbody>
</table>
### RISC-V Immediate Encoding

#### Instruction encodings, \text{inst}[31:0]

<table>
<thead>
<tr>
<th>Field</th>
<th>R-type</th>
<th>I-type</th>
<th>S-type</th>
<th>B-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td></td>
</tr>
<tr>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td></td>
</tr>
<tr>
<td>rs1</td>
<td>funct3</td>
<td>imm[4:0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct3</td>
<td>immm[4:1</td>
<td>11]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td></td>
</tr>
<tr>
<td>imm[12</td>
<td>10:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
</tr>
<tr>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit immediates produced, \text{imm}[31:0]

- **I-imm.**
  - \text{-inst[31]-}
  - \text{inst[30:25], inst[24:21], inst[20]}

- **S-imm.**
  - \text{-inst[31]-}
  - \text{inst[30:25], inst[11:8], inst[7]}

- **B-imm.**
  - \text{-inst[31]-}
  - \text{inst[7], inst[30:25], inst[11:8], 0}

---

Upper bits sign-extended from \text{inst[31]} always

Only bit 7 of instruction changes role in immediate between S and B
Branch Example, Complete Encoding

\[
\text{beq } x19, x10, \text{ offset } = 16 \text{ bytes}
\]

13-bit immediate, \( \text{imm}[12:0] \), with value 16

\[
\begin{array}{cccccccc}
0 & 000000 & 01010 & 10011 & 000 & 1000 & 0 & 1100011 \\
\end{array}
\]

\( \text{imm}[12] \) discarded, always zero

\( \text{imm}[11] \)

\( \text{imm}[10:5] \) \( rs2=10 \) \( rs1=19 \) \text{ BEQ } \text{ imm}[4:1] \text{ 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 (‘position-independent code’)

- What do we do if destination is $> 2^{10}$ instructions away from branch?
  - Other instructions save us
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 ('position-independent code')

- What do we do if destination is > $2^{10}$ instructions away from branch?
  - Other instructions save us

```assembly
beq x10, x0, far          bne x10, x0, next
# next instr    →         j   far
    next: # next instr
```
### U-Format for “Upper Immediate” Instructions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td></td>
<td>LUI</td>
</tr>
<tr>
<td>opcode</td>
<td></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
  - \texttt{lui} – Load Upper Immediate
  - \texttt{auipc} – Add Upper Immediate to PC
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 $0x{\text{DEADBEEF}}$?

\begin{align*}
lui & \ x10, \ 0x{\text{DEADB}} \quad \# \ x10 = 0x{\text{DEADB000}} \\
addi & \ x10, \ x10, \ 0xEEF \quad \# \ x10 = 0x{\text{DEADAEFF}}
\end{align*}

\textbf{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?

```
LUI x10, 0xDEADC  # x10 = 0xDEADC000

ADDI x10, x10, 0xEEF  # x10 = 0xDEADBEEF
```

Pre-increment 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:

```
li x10, 0xDEADBEEF  # Creates two instructions
```
- Adds upper immediate value to PC and places result in destination register
- Used for PC-relative addressing

Label: AUIPC x10, 0 # Puts address of label in x10
J-Format
J-Format for Jump Instructions

<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>
</table>

- **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
Uses of JAL

# JAL pseudo-instruction

j Label = jal x0, Label # Discard return address

# Call function within $2^{18}$ instructions of PC
jal ra, FuncName
JALR Instruction (I-Format)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

```
offset[11:0]  base  dest  JALR
0              0     0     0
```

- `jalr rd, rs, immediate`
  - Writes PC+4 to rd (return address)
  - Sets PC = `rs + immediate`
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
    - In contrast to branches and `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>
jalr x0, x1, <lo12bits>
“And In Conclusion...”
## Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>7</td>
<td>R-type</td>
</tr>
<tr>
<td>rs2</td>
<td>24</td>
<td>I-type</td>
</tr>
<tr>
<td>rs1</td>
<td>20</td>
<td>S-type</td>
</tr>
<tr>
<td>funct3</td>
<td>19</td>
<td>B-type</td>
</tr>
<tr>
<td>rd</td>
<td>15</td>
<td>U-type</td>
</tr>
<tr>
<td>opcode</td>
<td>31</td>
<td>J-type</td>
</tr>
</tbody>
</table>

- **R-type**
  - `imm[11:0]` rs2 rs1 funct3 rd opcode
- **I-type**
  - `imm[11:0]` rs1 funct3 rd opcode
- **S-type**
- **B-type**
- **U-type**
  - `imm[31:12]` rd opcode
- **J-type**
### Base Integer Instructions: RV32I

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shifts</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>R</td>
<td>SLL</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Logical</td>
<td>I</td>
<td>SLTI</td>
<td>rd,rs1,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>R</td>
<td>SRL</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Logical</td>
<td>I</td>
<td>SLTIU</td>
<td>rd,rs1,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>R</td>
<td>SRA</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Logical</td>
<td>I</td>
<td>SRAI</td>
<td>rd,rs1,shamt</td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD</td>
<td>ADD</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Immediate</td>
<td>ADDI</td>
<td>rd,rs1,imm</td>
<td></td>
</tr>
<tr>
<td><strong>Load Upper Imm</strong></td>
<td>U</td>
<td>LUI</td>
<td>rd,imm</td>
</tr>
<tr>
<td></td>
<td>SUB</td>
<td>rd,rs1,rs2</td>
<td></td>
</tr>
<tr>
<td>Add Upper Imm to PC</td>
<td></td>
<td>AUIPC</td>
<td>rd,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>R</td>
<td>XOR</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Logical</td>
<td>I</td>
<td>XORI</td>
<td>rd,rs1,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>R</td>
<td>OR</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Logical</td>
<td>I</td>
<td>ORI</td>
<td>rd,rs1,imm</td>
</tr>
<tr>
<td><strong>Logical</strong></td>
<td>AND</td>
<td>AND</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Immediate</td>
<td>ANDI</td>
<td>ANDI</td>
<td>rd,rs1,imm</td>
</tr>
<tr>
<td>Compare</td>
<td>Set &lt; R</td>
<td>SLT</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td>Immediate</td>
<td>SLTI</td>
<td>rd,rs1,imm</td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>Set &lt; Unsigned R</td>
<td>SLTU</td>
<td>rd,rs1,rs2</td>
</tr>
<tr>
<td></td>
<td>SETIU</td>
<td>rd,rs1,imm</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td>Branch = B</td>
<td>BEQ</td>
<td>rs1,rs2,imm</td>
</tr>
<tr>
<td></td>
<td>BNE</td>
<td>rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BLT</td>
<td>rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BGE</td>
<td>rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BLTU</td>
<td>rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BGEU</td>
<td>rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td>Jump &amp; Link</td>
<td>JAL</td>
<td>JAL</td>
<td>rd,imm</td>
</tr>
<tr>
<td></td>
<td>JALR</td>
<td>JALR</td>
<td>rd,rs1,imm</td>
</tr>
</tbody>
</table>

---

**Not in 61C**

---

**Open RISC-V Reference Card**

---

**Complete RV32I ISA**