CS 61C:
Great Ideas in Computer Architecture
RISC-V Instruction Formats

Instructors:
Krste Asanović and Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa17
Levels of Representation/Interpretation

**High Level Language Program (e.g., C)**

- Compiler

**Assembly Language Program (e.g., RISC-V)**

- Assembler

**Machine Language Program (RISC-V)**

- Machine Interpretation
  - Hardware Architecture Description (e.g., block diagrams)
  - Architecture Implementation
  - Logic Circuit Description (Circuit Schematic Diagrams)

**Examples of Assembly Language:**

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

- lw x10, 0(x12)
- lw x11, 4(x12)
- sw x11, 0(x12)
- sw x10, 4(x12)

Anything can be represented as a number, i.e., data or instructions.

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
```

**Hardware Architecture Description (e.g., block diagrams)**

**Logic Circuit Description (Circuit Schematic Diagrams)**
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
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
  – Unconstrained use of addresses can lead to nasty bugs; avoiding errors up to you in C; limited in Java by language design
• One 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
- New machines 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 is a word
  – `lw` and `sw` both access memory one word at a time

• So how do we represent instructions?
  – Remember: Computer only understands 1s and 0s, so assembler string “`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 instructions used for RV32, RV64, RV128
Instructions as Numbers (2/2)

• One word is 32 bits, so divide instruction word into “fields”
• Each field tells processor something about instruction
• We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
  – R-format for register-register arithmetic operations
  – I-format for register-immediate arithmetic operations and loads
  – S-format for stores
  – B-format for branches (minor variant of S-format, called SB before)
  – U-format for 20-bit upper immediate instructions
  – J-format for jumps (minor variant of U-format, called UJ before)
## Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>funct7</td>
<td>Function code</td>
</tr>
<tr>
<td>30</td>
<td>rs2</td>
<td>Source register 2</td>
</tr>
<tr>
<td>29-25</td>
<td>rs1</td>
<td>Source register 1</td>
</tr>
<tr>
<td>24</td>
<td>funct3</td>
<td>Function code 3</td>
</tr>
<tr>
<td>23</td>
<td>rd</td>
<td>Destination register</td>
</tr>
<tr>
<td>19</td>
<td>opcode</td>
<td>Opcode</td>
</tr>
</tbody>
</table>

**R-type**

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

**I-type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
</table>

**S-type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>imm[12]</td>
<td>Immediate</td>
</tr>
<tr>
<td>10-5</td>
<td>imm[10:5]</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

**B-type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-12</td>
<td>imm[31:12]</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

**U-type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>imm[20]</td>
<td>Immediate</td>
</tr>
<tr>
<td>10-1</td>
<td>imm[10:1]</td>
<td>Immediate</td>
</tr>
<tr>
<td>19-12</td>
<td>imm[19:12]</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

**J-type**

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20-1</td>
<td>imm[20]</td>
<td>Immediate</td>
</tr>
<tr>
<td>10-1</td>
<td>imm[10:1]</td>
<td>Immediate</td>
</tr>
<tr>
<td>19-12</td>
<td>imm[19:12]</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

### Notes
- **funct7** and **funct3** represent specific function codes for different instruction sets.
- **rs1** and **rs2** are source registers, with **rs2** being a special register for some instructions.
- **imm** fields contain immediate values for certain operations.
- **opcode** field determines the type of instruction.
32-bit instruction word divided into six fields of varying numbers of bits each: \(7+5+5+3+5+7 = 32\)

**Examples**
- **opcode** is a 7-bit field that lives in bits 6-0 of the instruction
- **rs2** is a 5-bit field that lives in bits 24-20 of the instruction
R-Format Instructions opcode/funct fields

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>funct3</th>
<th>rd</th>
<th>7</th>
<th>6</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to $0110011_{\text{two}}$ for all R-Format register-register arithmetic 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>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

- **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
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)

9/14/17
R-Format Example

- RISC-V Assembly Instruction:
  \[ \text{add} \ x_{18},x_{19},x_{10} \]

<table>
<thead>
<tr>
<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>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccccccc}
0000000 & 01010 & 10011 & 000 & 10010 & 0110011 \\
\end{array}
\]

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>Address</th>
<th>rs2</th>
<th>rs1</th>
<th>funct7</th>
<th>funct3</th>
<th>rd</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>000</td>
<td></td>
<td>011</td>
<td>ADD</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td></td>
<td>000</td>
<td></td>
<td>011</td>
<td>SUB</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>001</td>
<td></td>
<td>011</td>
<td>SLL</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>010</td>
<td></td>
<td>011</td>
<td>SLT</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>011</td>
<td></td>
<td>011</td>
<td>SLTU</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>100</td>
<td></td>
<td>011</td>
<td>XOR</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td>011</td>
<td>SRL</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td>011</td>
<td>SRA</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>110</td>
<td></td>
<td>011</td>
<td>OR</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td>111</td>
<td></td>
<td>011</td>
<td>AND</td>
</tr>
</tbody>
</table>

Different encoding in funct7 + funct3 selects different operations.
I-Format Instructions

• What about instructions with immediates?
  – 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  – Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise

• Define new instruction format that is mostly consistent with R-format
  – Notice if instruction has immediate, then uses at most 2 registers (one source, one destination)
I-Format Instruction Layout

<table>
<thead>
<tr>
<th></th>
<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></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>12</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct3</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

- Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, **imm[11:0]**
- Remaining fields (rs1, funct3, rd, opcode) same as before
- imm[11:0] can hold values in range \([-2048_{\text{ten}}, +2047_{\text{ten}}]\]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We’ll later see how to handle immediates > 12 bits
### I-Format Example

- **RISC-V Assembly Instruction:**
  
  ```
  addi  x15, x1, -50
  ```

  **Binary Representation:**

<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>

  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
  | 11111110011110 | 00001 | 000 | 01111 | 0010011 |

  - `imm` = -50
  - `rs1` = 1
  - `ADD`
  - `rd` = 15
  - `OP-Imm`
### All RV32 I-format Arithmetic Instructions

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0010011</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
<td>rd</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).
Administrivia

• HW0 Grades were released
  – If you have an issue with them, please fill out this form by tonight:
    https://goo.gl/forms/QH44Iw746pcCxOyH2
• HW1 Part 1 Due next Monday, Part 2 Due Friday Sept 22
  – Homework-oriented office hours next Monday (check the website)
  – Homework Party on next Wednesday (6:30-10pm in 293 Cory)
  – Autograded results to be released every noon starting this Friday
• Midterm #1 in 1.5 weeks: September 26!
  – Two sided 8.5” x 11” cheat sheet + RISC-V Green Card that we give you
  – DSP students: please make sure we know about your special accommodations (contact Steven Ho the head TA if you haven’t yet)
Break!
Load Instructions are also I-Type

- The 12-bit signed immediate is added to the base address in register \texttt{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 \texttt{rd}
I-Format Load Example

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

\begin{verbatim}
<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>
\end{verbatim}

\begin{verbatim}
00000000001000 | 00010 | 010 | 01110 | 0000011
\end{verbatim}

\begin{itemize}
  \item imm=+8
  \item rs1=2
  \item LW
  \item rd=14
  \item LOAD
\end{itemize}
### All RV32 Load Instructions

<table>
<thead>
<tr>
<th><em>imm[11:0]</em></th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0000011</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>imm[11:0]</em></td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0000011</td>
<td>LH</td>
</tr>
<tr>
<td><em>imm[11:0]</em></td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0000011</td>
<td>LW</td>
</tr>
<tr>
<td><em>imm[11:0]</em></td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0000011</td>
<td>LBU</td>
</tr>
<tr>
<td><em>imm[11:0]</em></td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0000011</td>
<td>LHU</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

**funct3** field encodes size and signedness of load data
S-Format Used for Stores

<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>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

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

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

\[
\begin{array}{cccccccc}
\text{imm}[11:5] & rs2 & rs1 & \text{funct3} & \text{imm}[4:0] & \text{opcode} \\
0 & 7 & 5 & 5 & 3 & 7 \\
\text{offset}[11:5] & \text{src} & \text{base} & \text{width} & \text{offset}[4:0] & \text{STORE} \\
0000000 & 01110 & 00010 & 010 & 01000 & 0100011 \\
\end{array}
\]

- offset[11:5] = 0
- rs2 = 14
- rs1 = 2
- SW
- offset[4:0] = 8

Combined 12-bit offset = 8
All RV32 Store Instructions

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

SB
SH
SW
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 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

- **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 $\pm 2^{11}$ addresses from the PC
- **Why not use byte address offset from PC?**
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:
  - immediate is number of instructions to jump (remember, specifies words) either forward (+) or backwards (−)

9/14/17
RISC-V Feature, n×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 ½ 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 $±2^{10} \times 32$-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/rs2) and a 12-bit immediate
• 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: \texttt{beq x19,x10,End}
  add \ x18, x18, x10
  addi \ x19, x19, -1
  j \ Loop

  End: \# target instruction

• Branch offset = \(4\times32\)-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>????</td>
<td>1100011</td>
<td></td>
</tr>
</tbody>
</table>
**RISC-V Immediate Encoding**

### Instruction Encodings, \( \text{inst}[31:0] \)

| 31 | 30 | 25 24 | 21 20 | 19 | 15 14 | 12 11 | 8 7 | 6 0 | \multicolumn{2}{c}{\text{R-type}} |
|----|----|-------|------|----|-------|-------|----|----|-------|-----------------|
| funct7 | rs2 | rs1 | funct3 | rd | opcode |

| \multicolumn{5}{c}{\text{I-type}} |
| \multicolumn{5}{c}{\text{imm}[11:0]} |
| \multicolumn{5}{c}{rs1 | funct3 | rd | opcode} |

| \multicolumn{5}{c}{\text{S-type}} |
| \multicolumn{5}{c}{\text{imm}[11:5]} |
| \multicolumn{5}{c}{rs2 | rs1 | funct3 | \text{imm}[4:0] | opcode} |

| \multicolumn{5}{c}{\text{B-type}} |
| \multicolumn{5}{c}{\text{imm}[12] | \text{imm}[10:5]} |
| \multicolumn{5}{c}{rs2 | rs1 | funct3 | \text{imm}[4:1] | \text{imm}[11] | opcode} |

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

- **I-immediate**
  - \( \text{inst}[31] \)

- **S-immediate**
  - \( \text{inst}[31] \)

- **B-immediate**
  - \( \text{inst}[31] \)
  - \( \text{inst}[7] \), \( \text{inst}[30:25] \), \( \text{inst}[11:8] \), \( 0 \)

---

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, imm[12:0], with value 16

imm[0] discarded, always zero


imm[12]

imm[10:5] rs2=10

rs1=19

BEQ

imm[4:1]

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

• 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 (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`  
    # next instr  \rightarrow  
    j \quad \text{far}

  \hspace{1cm} \quad \text{next: # next instr}
Break!
U-Format for “Upper Immediate” instructions

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

\[
\text{LUI } x10, \ 0x87654 \quad \# \ x10 = 0x87654000 \\
\text{ADDI } x10, \ x10, \ 0x321 \# \ x10 = 0x87654321
\]
One Corner Case

How to set 0xDEADBEEF?

LUI x10, 0xDEADB # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF# x10 = 0xDEADAEEF

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, 0xDEAD

ADDI x10, x10, 0xEEF

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
AUIPC

- 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 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 ±2^{19} locations, 2 bytes apart
  - ±2^{18} 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
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
**JALR Instruction (I-Format)**

- 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
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>
### Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field</th>
<th>Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>31-30</td>
<td>Branch target address (R-type)</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>25-24</td>
<td>Immediate value (I-type)</td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>21-20</td>
<td>Immediate value (S-type)</td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>15-14</td>
<td>Immediate value (U-type)</td>
</tr>
<tr>
<td>rs2</td>
<td>21</td>
<td>Source register 2 (R-type)</td>
</tr>
<tr>
<td>rs1</td>
<td>19</td>
<td>Source register 1 (I-type)</td>
</tr>
<tr>
<td>funct3</td>
<td>15-14</td>
<td>Function code (R-type)</td>
</tr>
<tr>
<td>funct3</td>
<td>12-11</td>
<td>Function code (I-type)</td>
</tr>
<tr>
<td>funct3</td>
<td>12-11</td>
<td>Function code (S-type)</td>
</tr>
<tr>
<td>funct3</td>
<td>12-11</td>
<td>Function code (B-type)</td>
</tr>
<tr>
<td>funct3</td>
<td>12-11</td>
<td>Function code (U-type)</td>
</tr>
<tr>
<td>funct3</td>
<td>12-11</td>
<td>Function code (J-type)</td>
</tr>
<tr>
<td>rd</td>
<td>8-7</td>
<td>Destination register (R-type)</td>
</tr>
<tr>
<td>opcode</td>
<td>6-0</td>
<td>Opcode (R-type)</td>
</tr>
<tr>
<td>opcode</td>
<td>6-0</td>
<td>Opcode (I-type)</td>
</tr>
<tr>
<td>opcode</td>
<td>6-0</td>
<td>Opcode (S-type)</td>
</tr>
<tr>
<td>opcode</td>
<td>6-0</td>
<td>Opcode (B-type)</td>
</tr>
<tr>
<td>opcode</td>
<td>6-0</td>
<td>Opcode (U-type)</td>
</tr>
<tr>
<td>opcode</td>
<td>6-0</td>
<td>Opcode (J-type)</td>
</tr>
</tbody>
</table>
**Complete RV32I ISA**

<table>
<thead>
<tr>
<th>imm[31:12]</th>
<th>rd</th>
<th>0110111</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>0010111</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>imm[12:10:5]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[12:10:5]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[12:10:5]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
</tbody>
</table>

LUI
AUIPC
JAL
JALR
BEQ
BNE
BLT
BGE
BLTU
BGEU
LB
LH
LW
LB
LBU
LHU
SB
SH
SW
ADDI
SLTI
SLTIU
XORI
ORI
ANDI

0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI
SRLI
SRAI
SUB
SLL
SLT
SLTU
XOR
SRL
SRA
OR
AND
FENCE
FENCE.I
ECALL
EBREAK
CSRRW
CSRRS
CSRRC
CSRRWI
CSRRSI
CSRRCI

Not in CS61C