CS61C
Great Ideas in Computer Architecture (a.k.a. Machine Structures)

RISC-V Instruction Representation
Great Idea #1: Abstraction (Levels of Representation/Interpretation)

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

\[
\text{temp} = v[k]; \\
v[k] = v[k+1]; \\
v[k+1] = \text{temp};
\]

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

\[
\begin{align*}
\text{lw} & \ x3, \ 0(x10) \\
\text{lw} & \ x4, \ 4(x10) \\
\text{sw} & \ x4, \ 0(x10) \\
\text{sw} & \ x3, \ 4(x10)
\end{align*}
\]

Machine Language Program (RISC-V)

\[
\begin{align*}
1000 & \ 1101 \ 1110 \ 0010 \ 0000 \ 0000 \ 0000 \ 0000 \\
1000 & \ 1110 \ 0001 \ 0000 \ 0000 \ 0000 \ 0000 \ 0100 \\
1010 & \ 1110 \ 0001 \ 0010 \ 0000 \ 0000 \ 0000 \ 0000 \\
1010 & \ 1101 \ 1110 \ 0010 \ 0000 \ 0000 \ 0000 \ 0100
\end{align*}
\]

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

\[
\text{Out} = AB + CD
\]
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 Electronic 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 (IP)
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
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
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)
- U-format for 20-bit upper immediate instructions
- J-format for jumps (minor variant of U-format)
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>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>

- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to $0110011_2$ for all R-Format register-register arithmetic instructions

- **funct7 + funct3**: combined with opcode, these two fields describe what operation to perform

- **Question**: You have been professing simplicity, so why aren’t opcode and funct7 and funct3 a single 17-bit field?
  - We’ll answer this later
R-Format Instructions Register Specifiers

- **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)
R-Format Example

- RISC-V Assembly Instruction:
  
  ```
  add x18, x19, x10
  ```

<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>01010</td>
<td>10011</td>
<td>000</td>
<td>10010</td>
<td>0110011</td>
</tr>
</tbody>
</table>

  add rs2=10 rs1=19
  add rd=18 Reg-Reg OP
Your Turn

- What is correct encoding of `add x4, x3, x2`?

1) 4021 8233_{hex}
2) 0021 82b3_{hex}
3) 4021 82b3_{hex}
4) 0021 8233_{hex}
5) 0021 8234_{hex}

<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>00000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Garcia, Nikolić
## All RV32 R-format Instructions

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

Different encoding in funct7 + funct3 selects different operations.

Can you spot two new instructions?
I-Format
Layout
I-Format Instructions

- What about instructions with immediates?
  - Compare:
    - `add  rd, rs1, rs2`
    - `addi  rd, rs1, imm`
  - 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)
- 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_{ten}, +2047_{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:**
  
  ```assembly
  addi x15, x1, -50
  ```

- **Binary Representation:**

<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>

- **Binary Code:**

  111111001110 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>0000000</td>
<td>shamt</td>
<td>001</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>0000000</td>
<td>shamt</td>
<td>101</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>0100000</td>
<td>shamt</td>
<td>101</td>
<td>rd</td>
<td>0010011</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).

Garcia, Nikolić

Berkeley UNIVERSITY OF CALIFORNIA RISC-V (21)
RISC-V Loads
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:**
  \[ \text{lw} \ x14, \ 8(\text{x2}) \]

<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>
<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>

- \[ \text{imm} = +8 \]
- \[ \text{rs1} = 2 \]
- \[ \text{lw} \]
- \[ \text{rd} = 14 \]

\[(\text{load word})\]
### All RV32 Load Instructions

<table>
<thead>
<tr>
<th></th>
<th>rs1</th>
<th>imm[11:0]</th>
<th>rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lb</td>
<td>000</td>
<td></td>
<td>0000011</td>
<td>lb</td>
</tr>
<tr>
<td>lh</td>
<td>001</td>
<td></td>
<td>0000011</td>
<td>lh</td>
</tr>
<tr>
<td>lw</td>
<td>010</td>
<td></td>
<td>0000011</td>
<td>lw</td>
</tr>
<tr>
<td>lbu</td>
<td>100</td>
<td></td>
<td>0000011</td>
<td>lbu</td>
</tr>
<tr>
<td>lhu</td>
<td>101</td>
<td></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
S-Format Used for Stores

- Store needs to read two registers, \( rs1 \) for base memory address, and \( rs2 \) for data to be stored, as well 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 to 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:
  \[
  \text{sw \ x14, \ 8(x2)}
  \]

<table>
<thead>
<tr>
<th>Offset</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>imm[4:0]</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0000011</td>
</tr>
</tbody>
</table>

Combining the 12-bit offset:

\[
\text{offset[11:5] = 0, \ rs2 = 14, \ rs1 = 2, \ offset[4:0] = 8, \ STORE}
\]

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

- Store byte, halfword, word

<table>
<thead>
<tr>
<th>Imm[11:5]</th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>imm[4:0]</th>
<th>0100011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>imm[4:0]</td>
<td>0100011</td>
</tr>
<tr>
<td>Imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>imm[4:0]</td>
<td>0100011</td>
</tr>
</tbody>
</table>

**sb**

**sh**

**sw**

**width**
B-Format Layout
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**

- **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 (−)
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 $\pm 2^{10} \times 32$-bit instructions on either side of PC.
### RISC-V B-Format for Branches

<table>
<thead>
<tr>
<th>B-Format Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>offset[12</td>
</tr>
<tr>
<td>rs2</td>
</tr>
</tbody>
</table>

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate 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:  beq x19, x10, End
         add x18, x18, x10
         addi x19, x19, -1
         j   Loop
  End:   # target instruction
  ```

  **Count instructions from branch:**
  1  2  3  4

<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></td>
<td></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 x 2

  Offset Table:

<table>
<thead>
<tr>
<th>Value</th>
<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>0000000</td>
<td>01010</td>
<td>10011</td>
<td>000</td>
<td>1100011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### RISC-V Immediate Encoding

**Instruction encodings, \( \text{inst}[31:0] \)**

<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>
<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>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{imm}[11:5] )</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>( \text{imm}[4:0] )</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{imm}[12</td>
<td>10:5] )</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>( \text{imm}[4:1</td>
<td>11] )</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{inst}[31] )</td>
<td>( \text{inst}[30:25] )</td>
<td>( \text{inst}[11:8] )</td>
<td>( \text{inst}[7] )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{inst}[31] )</td>
<td>( \text{inst}[7] )</td>
<td>( \text{inst}[30:25] )</td>
<td>( \text{inst}[11:8] )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Only bit 7 of instruction changes role in immediate between S and B.

---

Garcia, Nikolić

Berkeley

University of California

RISC-V (41)
Beq x19, x10, offset = 16 bytes

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

\text{imm}[0] discarded, always zero

\text{imm}[12]

\text{imm}[10:5] \text{rs2} = 10 \quad \text{rs1} = 19 \quad \text{BEQ \ imm}[4:1] \quad \text{BRANCH}
### All RISC-V Branch Instructions

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

Key:
- `beq` (Branch Equal)
- `bne` (Branch Not Equal)
- `blt` (Branch Less Than)
- `bge` (Branch Greater Equal)
- `bltu` (Branch Less Than Unsigned)
- `bgeu` (Branch Greater Equal Unsigned)
Long
Immediates
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

```
beq x10, x0, far
# next instr
j   far
```

```
bne x10, x0, next
next: # next instr
```
U-Format for “Upper Immediate” Instructions

- 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`).

```
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 = 0xDEADAEFF
```

**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 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
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)

**Opcode:** imm[11:0] rsl func3 rd opcode

<table>
<thead>
<tr>
<th>31</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>imm[11:0]</td>
<td>rsl</td>
<td>func3</td>
<td>rd</td>
<td>opcode</td>
<td>dest</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>JALR</td>
</tr>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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
# ret and jr pseudo-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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>7</td>
<td>Function 7</td>
</tr>
<tr>
<td>rs2</td>
<td>25</td>
<td>Register 2</td>
</tr>
<tr>
<td>rs1</td>
<td>24</td>
<td>Register 1</td>
</tr>
<tr>
<td>funct3</td>
<td>21</td>
<td>Function 3</td>
</tr>
<tr>
<td>rd</td>
<td>20</td>
<td>Destination Register</td>
</tr>
<tr>
<td>opcode</td>
<td>19</td>
<td>Opcode</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>18</td>
<td>Immediate (I-type)</td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>12</td>
<td>Immediate (S-type)</td>
</tr>
<tr>
<td>imm[12</td>
<td>10:5]</td>
<td>6</td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>20</td>
<td>Immediate (U-type)</td>
</tr>
<tr>
<td>imm[20</td>
<td>10:1</td>
<td>11]</td>
</tr>
</tbody>
</table>

#### Instruction Types
- **R-type:** 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
- **I-type:** 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
- **S-type:** 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
- **B-type:** 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
- **U-type:** 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
- **J-type:** 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
### RV32I ISA Reference Card

#### 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>Shifts</td>
<td>Shift Left Logical R</td>
<td>SLL</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>Shift Left Log. Imm. I</td>
<td>SLIU</td>
<td>rd,r1,shamt</td>
</tr>
<tr>
<td></td>
<td>Shift Right Logical R</td>
<td>SRL</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>Shift Right Log. Imm. I</td>
<td>SRLI</td>
<td>rd,r1,shamt</td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic R</td>
<td>SRA</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>Shift Right Arith. Imm. I</td>
<td>SRAI</td>
<td>rd,r1,shamt</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADD R</td>
<td>ADD</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>ADD Immediate I</td>
<td>ADDI</td>
<td>rd,r1,imm</td>
</tr>
<tr>
<td></td>
<td>SUBtract R</td>
<td>SUB</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td>Load Upper Imm</td>
<td>U</td>
<td>LUI</td>
<td>rd,imm</td>
</tr>
<tr>
<td>Add Upper Imm to PC</td>
<td>R</td>
<td>AUIPC</td>
<td>rd,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>XOR I</td>
<td>XOR</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>XOR Immediate I</td>
<td>XORI</td>
<td>rd,r1,imm</td>
</tr>
<tr>
<td></td>
<td>OR I</td>
<td>OR</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>OR Immediate I</td>
<td>ORI</td>
<td>rd,r1,imm</td>
</tr>
<tr>
<td></td>
<td>AND R</td>
<td>AND</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>AND Immediate I</td>
<td>ANDI</td>
<td>rd,r1,imm</td>
</tr>
<tr>
<td>Compare</td>
<td>Set &lt; R</td>
<td>SLT</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>Set &lt; Immediate I</td>
<td>SLTI</td>
<td>rd,r1,imm</td>
</tr>
<tr>
<td></td>
<td>Set &lt; Unsigned R</td>
<td>SLTU</td>
<td>rd,r1,r2</td>
</tr>
<tr>
<td></td>
<td>Set &lt; Imm Unsigned R</td>
<td>SLTIU</td>
<td>rd,r1,imm</td>
</tr>
<tr>
<td>Branches</td>
<td>Branch = B</td>
<td>BEQ</td>
<td>rs1,rs2,imm</td>
</tr>
<tr>
<td></td>
<td>Branch ≠ B</td>
<td>BNE</td>
<td>rs1,rs2,imm</td>
</tr>
<tr>
<td></td>
<td>Branch &lt; B</td>
<td>BLT</td>
<td>rs1,rs2,imm</td>
</tr>
<tr>
<td></td>
<td>Branch ≥ B</td>
<td>BGE</td>
<td>rs1,rs2,imm</td>
</tr>
<tr>
<td></td>
<td>Branch 2 B</td>
<td>BGEU</td>
<td>rs1,rs2,imm</td>
</tr>
<tr>
<td>Jump &amp; Link</td>
<td>JAL J</td>
<td>JAL</td>
<td>rd,imm</td>
</tr>
<tr>
<td></td>
<td>Jump &amp; Link Register I</td>
<td>JALR</td>
<td>rd,r1,imm</td>
</tr>
</tbody>
</table>

#### Not in 61C

- Branches: BGEU
- Jump & Link: JALR