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

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

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

- lw x3, 0(x10)
- lw x4, 4(x10)
- sw x4, 0(x10)
- sw x3, 4(x10)

Machine Language Program (RISC-V)

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

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 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
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)
  - U-format for 20-bit upper immediate instructions
  - J-format for jumps (minor variant of U-format)
R-Format Instruction Layout

- 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>Field</th>
<th>Bit Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>31-26</td>
<td>Combined with funct3 to describe the operation</td>
</tr>
<tr>
<td>rs2</td>
<td>25-22</td>
<td>Source register 2 for the operation</td>
</tr>
<tr>
<td>rs1</td>
<td>21-18</td>
<td>Source register 1 for the operation</td>
</tr>
<tr>
<td>funct3</td>
<td>17-14</td>
<td>Combined with funct7 to describe the operation</td>
</tr>
<tr>
<td>rd</td>
<td>13-10</td>
<td>Destination register for the operation</td>
</tr>
<tr>
<td>opcode</td>
<td>9-0</td>
<td>Specifies the instruction</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**: 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

<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>

**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
  ```

- R-Format Example

<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>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

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

- What is correct encoding of \texttt{add x4, x3, x2}?
  1) \(4021 \text{ 8233}_{\text{hex}}\)
  2) \(0021 \text{ 82b3}_{\text{hex}}\)
  3) \(4021 \text{ 82b3}_{\text{hex}}\)
  4) \(0021 \text{ 8233}_{\text{hex}}\)
  5) \(0021 \text{ 8234}_{\text{hex}}\)

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>15</td>
<td>14</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\text{add}

\text{sub}

\text{xor}

\text{or}

\text{and}
All RV32 R-format Instructions

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>funct7</th>
<th>rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>000</td>
<td></td>
<td>0110011</td>
<td>add</td>
</tr>
<tr>
<td>010000</td>
<td></td>
<td></td>
<td>000</td>
<td></td>
<td>0110011</td>
<td>sub</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>001</td>
<td></td>
<td>0110011</td>
<td>sll</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>010</td>
<td></td>
<td>0110011</td>
<td>slt</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>011</td>
<td></td>
<td>0110011</td>
<td>sltu</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>100</td>
<td></td>
<td>0110011</td>
<td>xor</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td>0110011</td>
<td>srl</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>110</td>
<td></td>
<td>0110011</td>
<td>sra</td>
</tr>
<tr>
<td>010000</td>
<td></td>
<td></td>
<td>101</td>
<td></td>
<td>0110011</td>
<td>or</td>
</tr>
<tr>
<td>000000</td>
<td></td>
<td></td>
<td>111</td>
<td></td>
<td>0110011</td>
<td>and</td>
</tr>
</tbody>
</table>

Different encoding in funct7 + funct3 selects different operations
Can you spot two new instructions?
I-Format Layout
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, \texttt{rs2} and \texttt{funct7} replaced by 12-bit signed immediate, \texttt{imm[11:0]}

Remaining fields (\texttt{rs1, funct3, rd, opcode}) same as before

\texttt{imm[11:0]} can hold values in range \([-2048_{10}, +2047_{10}]\)

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
  ```

<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>

  | 1111110011110 | 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>sham</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>sham</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
</tr>
<tr>
<td>0100000</td>
<td>sham</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).
RISC-V Loads
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>

offset[11:0] base width dest LOAD

<table>
<thead>
<tr>
<th>offset[11:0]</th>
<th>base</th>
<th>width</th>
<th>dest</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>12219141112101000</td>
<td>0100</td>
<td>01110</td>
<td>000011</td>
<td>LOAD</td>
</tr>
</tbody>
</table>

imm=+8 rs1=2 lw rd=14 LOAD

(load word)
### All RV32 Load Instructions

<table>
<thead>
<tr>
<th>funct3</th>
<th>rd</th>
<th>rs1</th>
<th>imm[11:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>000</td>
<td>000</td>
<td>0x0</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>000</td>
<td>0x0</td>
</tr>
<tr>
<td>010</td>
<td>000</td>
<td>000</td>
<td>0x0</td>
</tr>
<tr>
<td>100</td>
<td>000</td>
<td>000</td>
<td>0x0</td>
</tr>
<tr>
<td>101</td>
<td>000</td>
<td>000</td>
<td>0x0</td>
</tr>
</tbody>
</table>

- **lb** 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

The funct3 field encodes size and ‘signedness’ of load data.
S-Format Layout
S-Format Used for Stores

- Store needs to read two registers, \textit{rs1} for base memory address, and \textit{rs2} for data to be stored, as well immediate offset!
- Can’t have both \textit{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 to move low 5 bits of immediate to where \textit{rd} field was in other instructions – keep \textit{rs1}/\textit{rs2} fields in same place
  - Register names more critical than immediate bits in hardware design
S-Format Example

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

\begin{verbatim}
  0000000  01110  00010  010  01000  0100011
\end{verbatim}

\begin{verbatim}
  0000000  01110  00010  010  01000  0100011
\end{verbatim}

\begin{verbatim}
  7  5  5  3  5  7


offset[11:5]  =0  rs2=14  rs1=2  SW  =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>

width