RISC-V Instruction Formats

Instructor: Steven Ho
Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g. RISCV)

Assembler

Machine Language Program (RISCV)

Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

temp = v[k];

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

v[k+1] = temp;

lw t0, 0(x2)

lw t1, 4(x2)

sw t1, 0(x2)

sw t0, 4(x2)

0000 1001 1100 0110 1010 1111 0101 1000

1010 1111 0101 1000 0000 1001 1100 0110

1100 0110 1010 1111 0101 1000 0000 1001

0101 1000 0000 1001 1100 0110 1010 1111
Agenda

• Stored-Program Concept
• R-Format
• I-Format
• Administrivia
• S-Format
• SB-Format
• U-Format
• UJ-Format
Big Idea: Stored-Program Concept

INSTRUCTIONS ARE DATA

• programs can be stored in memory as numbers
• Before: a number can mean anything
• Now: make convention for interpreting numbers as instructions
Great Ideas in Computer Architecture

Introduction to Machine Language

Instructor: Steven Ho
Instructions as Numbers

• By convention, RISCV instructions are each 1 word = 4 bytes = 32 bits

• Divide the 32 bits of an instruction into “fields”
  – regular field sizes → simpler hardware
  – will need some variation….

• Define 6 types of instruction formats:
  – R-Format    I-Format    S-Format    U-Format
    SB-Format    UJ-Format
The 6 Instruction Formats

- **R-Format:** instructions using 3 register inputs
  - `add`, `xor`, `mul` — arithmetic/logical ops
- **I-Format:** instructions with immediates, loads
  - `addi`, `lw`, `jalr`, `slli`
- **S-Format:** store instructions: `sw`, `sb`
- **SB-Format:** branch instructions: `beq`, `bge`
- **U-Format:** instructions with upper immediates
  - `lui`, `auipc` — upper immediate is 20-bits
- **UJ-Format:** jump instructions: `jal`
The 6 Instruction Formats

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>I</td>
<td>S</td>
<td>SB</td>
<td>U</td>
<td>UJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>27</td>
<td>26</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>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
<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>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>imm[31:12]</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>
Agenda

• Stored-Program Concept
• R-Format
• I-Format
• Administrivia
• S-Format
• SB-Format
• U-Format
• UJ-Format
R-Format Instructions (1/3)

• Define “fields” of the following number of bits each:
  \[7 + 5 + 5 + 3 + 5 + 7 = 32\]

  
  \[
  \begin{array}{cccccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
  \hline
  7 & 5 & 5 & 3 & 5 & 7 & 0 \\
  \end{array}
  \]

• Each field has a name:

  \[
  \begin{array}{cccccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
  \hline
  \text{funct7} & \text{rs2} & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} & 0 \\
  \end{array}
  \]

• Each field is viewed as its own \texttt{unsigned int}
  – 5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-128, etc.
R-Format Instructions (2/3)

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
</table>

- **opcode (7):** partially specifies operation
  - e.g. **R**-types have opcode = 0b0110011,
    **SB** (branch) types have opcode = 0b1100011
- **funct7+funct3 (10):** combined with **opcode**, these two fields describe what operation to perform

- How many R-format instructions can we encode?
  - with opcode fixed at 0b0110011, just funct varies: 
    \[ (2^7) \times (2^3) = (2^{10}) = 1024 \]
R-Format Instructions (3/3)

- **rs1**: \textcolor{red}{(5)}: 1\textsuperscript{st} operand ("source register 1")
- **rs2**: \textcolor{red}{(5)}: 2\textsuperscript{nd} operand (second source register)
- **rd**: \textcolor{red}{(5)}: "destination register" — receives the result of computation

- **Recall**: RISCV has 32 registers
  - A 5 bit field can represent exactly \(2^5 = 32\) things
    (interpret as the register numbers \textcolor{red}{x0-x31})
Reading from the Green Sheet

```
add t0 t1 t2
```

### RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>FMT</th>
<th>NAME</th>
<th>DESCRIPTION (in Verilog)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add,addw</td>
<td>R</td>
<td>ADD (Word)</td>
<td>R[rd] = R[rs1] + R[rs2]</td>
</tr>
</tbody>
</table>

### OPCODES IN NUMERICAL ORDER BY_OPCODE

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>FMT</th>
<th>OPCODE</th>
<th>FUNCT3</th>
<th>FUNCT7 OR IMM</th>
<th>HEXADECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>sd</td>
<td>S</td>
<td>0100011</td>
<td>011</td>
<td>00000000</td>
<td>0x33</td>
</tr>
<tr>
<td>add</td>
<td>R</td>
<td>0110011</td>
<td>000</td>
<td>00000000</td>
<td>0x33</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0110011</td>
<td>000</td>
<td>01000000</td>
<td>0x33</td>
</tr>
</tbody>
</table>

**Example:**

- **opcode (31-27):** 0x33
- **funct7 (26-25):** 0
- **funct3 (24-22):** 7
- **rs1 (21-16):** 6
- **rs2 (15-11):** 0
- **rd (10-5):** 5
R-Format Example

• RISCV Instruction: \texttt{add \ x5, \ x6, \ x7}

Field representation (decimal):

\begin{center}
\begin{tabular}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
0 & 7 & 6 & 0 & 5 & \textbf{0x33} & & \\
\end{tabular}
\end{center}

Field representation (binary):

\begin{center}
\begin{tabular}{cccccccc}
31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
0000000 & 00111 & 00110 & 000 & 00101 & 0110011 & & \\
\end{tabular}
\end{center}

hex representation: \texttt{0x007302B3}
decimal representation: 7,537,331

Called a \textbf{Machine Language Instruction}
## All RV32 R-format instructions

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

Different encoding in funct7 + funct3 selects different operations.
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format
I-Format Instructions (1/4)

• What about instructions with immediates?
  – 5-bit field too small for most immediates

• Ideally, RISCV would have only one instruction format (for simplicity)
  – Unfortunately here we need to compromise

• Define new instruction format that is mostly consistent with R-Format
  – First notice that, if instruction has immediate, then it uses at most 2 registers (1 src, 1 dst)
I-Format Instructions (2/4)

- Define “fields” of the following number of bits each: $12 + 5 + 3 + 5 + 7 = 32$ bits

- Field names:

- **Key Concept:** Only $imm$ field is different from R-format: $rs2$ and $funct7$ replaced by 12-bit signed immediate, $imm[11:0]$
I-Format Instructions (3/4)

- **opcode** (7): uniquely specifies the instruction
- **rs1** (5): specifies a register operand
- **rd** (5): specifies destination register that receives result of computation
I-Format Instructions (4/4)

- **Immediate (12):** 12 bit number
  - All computations done in words, so 12-bit immediate must be *extended* to 32 bits
  - Always *sign-extended* to 32-bits before use in an arithmetic operation

- Can represent $2^{12}$ different immediates
  - imm[11:0] can hold values in range $[-2^{11}, +2^{11}]$
### I-Format Example (1/2)

```
addi x15, x1, -50
```

The instruction `addi x15, x1, -50` performs an addition of the immediate value `-50` to the register `x1` and stores the result in `x15`. The immediate value `-50` is encoded in the instruction as a 12-bit value, where the upper 11 bits (imm[11:0]) are encoded as `000000000000`, and the lower bit (immediate sign) is encoded as `0`, indicating a positive value.

#### Opcodes

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>FMT</th>
<th>Opcode</th>
<th>FUNCT3</th>
<th>FUNCT7 or IMM</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>fence.i</td>
<td>I</td>
<td>0001111</td>
<td>001</td>
<td></td>
<td>0F/1</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>0010011</td>
<td>000</td>
<td></td>
<td>13/0</td>
</tr>
<tr>
<td>slli</td>
<td>I</td>
<td>0010011</td>
<td>001</td>
<td>00000000</td>
<td>13/1700</td>
</tr>
</tbody>
</table>

The instruction is represented in binary as:

```
11111110011110 00001 0 01111 0x13
```

- **31**
  - 31:11 = imm[11:0]
  - 10:5 = rs1
  - 4:2 = func3
  - 1:0 = rd
  - 0 = opcode

#### Register Assignments

- `rd = x15`
- `rs1 = x1`
• RISC-V Instruction: `addi x15, x1, -50`

Field representation (binary):

```
111111001110 00001 000 01111 0010011
```

hex representation: `0xFCE0 8793`
decimal representation: `4,242,573,203`
## All RISCV I-Type 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>01000000</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).
Question: If the number of registers were halved, which statement is true?

(A) There must be more R-type instructions

(B) There must be less I-type instructions

(C) Shift amounts would change to 0-63

(D) I-type instructions could have 2 more immediate bits
Agenda

• Stored-Program Concept
• R-Format
• I-Format
• Administrivia
• S-Format
• SB-Format
• U-Format
• UJ-Format
Administrivia

• Guerrilla Session today! 4-6PM, Cory 540AB
• HW1 Due Thursday (6/28)
• Project Due on Friday (6/29)
  – Submission Policy:
  – Testing framework has been released
  – You can do it!! :D
• HW2 Due next Friday (7/06)
• Review Session this Saturday (6/30), 2-4PM
• Guerilla Session part 2, this Sunday (7/01) 2-4PM, Cory 540AB
  – This will also be helpful for the midterm! (That’s why we moved it up)
Agenda

- Stored-Program Concept
- R-Format
  - I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format
Load Instructions are also I-Type

- 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
- Value loaded from memory is stored in \( rd \)
I-Format Load Example

- `lw x14, 8(x2)`

```
<table>
<thead>
<tr>
<th></th>
<th>imm[11:0]</th>
<th>rs1</th>
<th>func3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LOADC</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>offset[11:0]</th>
<th>base</th>
<th>width</th>
<th>dst</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000000001000</td>
<td>00010</td>
<td>010</td>
<td>01111</td>
<td>0000011</td>
</tr>
</tbody>
</table>
```

- imm=+8
- rs1=2
- LW
- rd=14
- LOAD
### All RV32 Load Instructions

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

• Stored-Program Concept
• R-Format
• I-Format
• Administrivia
  • S-Format
  • SB-Format
  • U-Format
  • UJ-Format
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 as need immediate offset!
• Can’t have both \( rs2 \) and immediate in same place as other instructions!
• Note: 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

```
  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-Format Example

\texttt{sw \ x14, \ 8(x2)}

\begin{verbatim}
\begin{array}{cccccccc}
\text{imm}[11:5] & \text{rs2} & \text{rs1} & \text{func3} & \text{imm}[4:0] & \text{opcode} \\
00000000 & 01110 & 00010 & 010 & 01000 & 0100011 \\
\end{array}
\end{verbatim}

\begin{align*}
\text{off}[11:5] &= 0 \\
\text{rs2} &= 14 \\
\text{rs1} &= 2 \\
\text{SW} &\hspace{1cm} \text{off}[4:0] = 8 \\
\text{STORE} &\hspace{1cm} \text{combined 12-bit offset} = 8
\end{align*}
All RV32 Store Instructions

|-----------|-----|-----|-----|----------|---------|
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- UJ-Format
Branching Instructions

• `beq, bne, bge, blt`
  – Need to specify an **address** to go to
  – Also take **two** registers to compare
  – Doesn’t write into 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)

• 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 as the immediate?
Branching Reach

• **Recall:** RISCV uses 32-bit addresses, and memory is **byte-addressed**

• Instructions are **“word-aligned”**: Address is always a multiple of 4 (in bytes)

• PC ALWAYS points to an instruction
  – PC is *typed* as a pointer to a word
  – can do C-like pointer arithmetic

• Let immediate specify #words instead of #bytes
  – Instead of specifying $\pm 2^{11}$ bytes from the PC, we will now specify $\pm 2^{11}$ words $= \pm 2^{13}$ byte addresses around PC
Branch Calculation

• If we don’t take the branch:

\[ PC = PC + 4 = \text{next instruction} \]

• If we do take the branch:

\[ PC = PC + (\text{immediate} \times 4) \]

• Observations:

– \text{immediate} is number of instructions to move (remember, specifies words) either forward (+) or backwards (–)
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
• 16-bit = half-word
• To enable this, RISC-V scales the branch offset to be half-words 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 $-2^{12}$ to $+2^{12}-2$ 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 (1/2)

• RISCV Code:

```
Loop:    beq   x19,x10,End
        add   x18,x18,x10
        addi  x19,x19,-1
        j     Loop
End:     <target instr>
```

• Branch offset = $4 \times 32$-bit instructions $= 16$ bytes
• (Branch with offset of 0, branches to itself)
Branch Example (1/2)

- RISCV Code:

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

  End: <target instr>

Start counting from instruction AFTER the branch.
Branch Example (1/2)

\( \text{beq x19,x10,offset = 16 bytes} \)

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

\[
\begin{array}{c}
0000000010000000
\end{array}
\]

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

\[
\begin{array}{c}
00000000010000000
\end{array}
\]
RISC-V Immediate Encoding

• Why is it so confusing?!?!

Instruction Encodings, inst[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>21</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>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td></td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit immediates produced, imm[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>20</th>
<th>19</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>5</th>
<th>4</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B
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 (why?)

• What do we do if destination is $> 2^{10}$ instructions away from branch?
  – Other instructions save us:
    
    
    \[
    \begin{align*}
    \text{beq x10, x0, far} & \quad \text{bne x10, x0, next} \\
    \text{# next instr} & \quad \rightarrow \quad \text{j far} \\
    \text{next: # next instr}
    \end{align*}
    \]
# Meet the $taff

<table>
<thead>
<tr>
<th></th>
<th>Sukrit</th>
<th>Suvansh</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Roadside Sign</strong></td>
<td>“Help, can’t find my roadsign”</td>
<td>&quot;Do not sushi enter&quot; (put picture)</td>
</tr>
<tr>
<td><strong>Greatest Weakness</strong></td>
<td>Committing to too many things</td>
<td>Binging TV</td>
</tr>
<tr>
<td><strong>Favorite artist</strong></td>
<td>Kanye West</td>
<td>Joshua Radin</td>
</tr>
<tr>
<td><strong>Favorite meme of all time</strong></td>
<td>All Spongebob memes</td>
<td>Avocado kid</td>
</tr>
</tbody>
</table>
Agenda

• Stored-Program Concept
• R-Format
• I-Format
• Administrivia
• S-Format
• SB-Format
• U-Format
• UJ-Format
Dealing With Large Immediates

• How do we deal with 32-bit immediates?
  – Our I-type instructions only give us 12 bits

• **Solution**: Need a new instruction format for dealing with the rest of the 20 bits.

• This instruction should deal with:
  – a destination register to put the 20 bits into
  – the immediate of 20 bits
  – the instruction opcode
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

- \texttt{lui} writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an \texttt{addi} to set low 12 bits, can create any 32-bit value in a register using two instructions (\texttt{lui}/\texttt{addi}).

\begin{verbatim}
lui x10, 0x87654  # x10 = 0x87654000
addi x10, x10, 0x321  # x10 = 0x87654321
\end{verbatim}
Corner Case

• How to set 0xDEADBEEF?

\[
\text{lui x10, 0xDEADB} \quad \# \quad x10 = 0xDEADB000 \\
\text{addi x10, x10,0xEEF} \quad \# \quad x10 = 0xDEADD\text{AEFF}
\]

\text{addi 12-bit immediate is always sign-extended!}
- if top bit of the 12-bit immediate is a 1, it will subtract -1 from upper 20 bits
Solution

• How to set 0xDEADBEEF?

lui x10, 0xDEADC000  # 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:
l i  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 into `x10`
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- Administrivia
- S-Format
- SB-Format
- U-Format
- **UJ-Format**
UJ-Format Instructions (1/3)

• For branches, we assumed that we won’t want to branch too far, so we can specify a change in the PC

• For general jumps (jal), we may jump to anywhere in code memory
  – Ideally, we would specify a 32-bit memory address to jump to
  – Unfortunately, we can’t fit both a 7-bit opcode and a 32-bit address into a single 32-bit word
  – Also, when linking we must write to an rd register
• `jal` saves PC+4 in register `rd` (the return address)
• Set PC = PC + offset (PC-relative jump)
• Target somewhere within ±2^{19} locations, 2 bytes apart
• ±2^{18} 32-bit instructions
• Reminder: “j” jump is a pseudo-instruction—the assembler will instead use `jal` but sets `rd=x0` to discard return address
• Immediate encoding optimized similarly to branch instruction to reduce hardware cost
UJ-Format Instructions (2/3)

- # j pseudo-instruction
- j Label = jal x0, Label # Discard return address
- # Call function within $2^{18}$ instructions of PC
- jal ra, FuncName

- Why is the immediate so funky?
  - Similar reasoning as for branch immediates
### jalr Instruction (I-Format)

#### Instruction Format

```
jalr rd, rs1, offset
```

- **Writes PC+4 to rd (return address)**
- **Sets PC = rs1 + offset**
- **Uses same immediates as arithmetic & loads**
  - no multiplication by 2 bytes

#### Table

```
<table>
<thead>
<tr>
<th></th>
<th>imm[11:0]</th>
<th>rs1</th>
<th>func3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>base</td>
<td>0</td>
<td>dest</td>
<td>JALR</td>
<td></td>
</tr>
</tbody>
</table>
```

---

61
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, <hi 20 bits>
jalr ra, x1, <lo 12 bits>

# Jump PC-relative with 32-bit offset
auipc x1, <hi 20 bits>
jalr x0, x1, <lo 12 bits>
**Question:** When combining two C files into one executable, we can compile them independently and then merge them together.

When merging two or more binaries:

1) **Jump** instructions don’t require any changes
2) **Branch** instructions don’t require any changes

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(A)</strong></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td><strong>(B)</strong></td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td><strong>(C)</strong></td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td><strong>(D)</strong></td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
# Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>21</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>
<td></td>
</tr>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[31:12]</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>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **R-type**: funct7 field has 7 bits, rs2 and rs1 fields have 5 bits each, funct3 field has 3 bits, rd field has 5 bits, and opcode field has 6 bits.
- **I-type**: imm[11:0] field has 12 bits, rs1 and funct3 fields have 5 bits each, rd field has 5 bits, and opcode field has 6 bits.
- **S-type**: imm[11:5] field has 7 bits, rs2 and rs1 fields have 5 bits each, funct3 field has 3 bits, imm[4:0] field has 5 bits, and opcode field has 6 bits.
- **B-type**: imm[12] and imm[10:5] fields have 7 bits each, rs2 and rs1 fields have 5 bits each, funct3 field has 3 bits, imm[4:1] and imm[11] fields have 5 bits each, and opcode field has 6 bits.
- **U-type**: imm[31:12] field has 20 bits, rd field has 5 bits, and opcode field has 6 bits.
Summary

• The Stored Program concept is very powerful
  – Instructions can be treated and manipulated the same way as data in both hardware and software

• RISCV Machine Language Instructions:

Not in CS61C