RISC-V Instruction Formats
Instructor: Stephan Kaminsky

Actual RISC-V code generated for the Tock operating system
Review of Last Lecture

• Pseudo-instructions
• Functions in assembly
  — Six steps to calling a function
• Calling conventions
  — Caller and callee saved registers
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

We were here
We are here
Agenda

- Stored-Program Concept
- R-Format
- I-Format
- S-Format
- SB-Format
- U-Format
- UJ-Format
- Summary
First computers were hard to reprogram

ENIAC (University of Pennsylvania - 1946)

• Blazing fast
  — 10 digit x 10 digit multiply in 2.8 ms

• Programmed with patch cords and switches
  — 2-3 days to set up a new program
Big Idea: Stored-Program Concept

• Instructions can be represented as bit patterns
  — Entire programs stored in memory just like data
  — Reprogramming just takes rewriting memory
    • Rather than rewiring the computer

• Known as “von Neumann” computers after widely distributed tech report on EDVAC project

First Draft of a Report on the EDVAC
by
John von Neumann
Between the
United States Army Ordnance Department
and the
University of Pennsylvania
Moore School of Electrical Engineering
University of Pennsylvania
June 30, 1945
Everything has a memory address

• Since instructions and data are both in memory, addresses can point to either

• C pointers are just memory addresses that point to data

• The Program Counter (PC) just holds a memory address that points to code
How do you distinguish code and data?

Depends on Interpretation

<table>
<thead>
<tr>
<th>Number</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>7,537,331</td>
<td>add x5, x6, x7</td>
</tr>
</tbody>
</table>

- programs can be stored in memory as numbers
- whether a number is code or a value is all in how you interpret it

- Need a method for interpreting numbers as instructions
Binary compatibility

Programs are distributed in binary form
• Bound to a specific instruction set

New machines in the same family want to run old programs ("binaries") as well as programs using new instructions
• Leads to "backwards-compatible" instruction set growing over time
Usually, the assembler does this translation.
Instructions as Numbers

• By convention, RISC-V 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:** the jump instruction: jal
The 6 Instruction Formats

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>27</th>
<th>26</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></td>
<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>R</td>
<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>B</td>
<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>
</tr>
</tbody>
</table>
Agenda

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

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

\[
\begin{array}{ccccccc}
& & & 7 & 5 & 5 & 3 & 5 & 7 \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
& & & & & & & & \\
\end{array}
\]

• Each field has a name:

\[
\begin{array}{ccccccc}
\text{funct7} & \text{rs2} & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
31 & 31 & 31 & 31 & 31 & 31 & 0
\end{array}
\]

• Each field is viewed as its own unsigned int

—5-bit fields can represent any number 0-31, while 7-bit fields can represent any number 0-127, etc.
• **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 (5):** 1st operand ("source register 1")
- **rs2 (5):** 2nd operand (second source register)
- **rd (5):** "destination register" — receives the result of computation

**Recall:** RISC-V has 32 registers

— A 5 bit field can represent exactly $2^5 = 32$ things
  (interpret as the register numbers 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>23/3</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>R</td>
<td>0110011</td>
<td>000</td>
<td>00000000</td>
<td>33/0/00</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0110011</td>
<td>000</td>
<td>01000000</td>
<td>33/0/20</td>
</tr>
</tbody>
</table>

```
<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>0x0</td>
<td>0x7</td>
<td>0x6</td>
<td>0x0</td>
<td>0x5</td>
<td>0x33</td>
</tr>
</tbody>
</table>
```
R-Format Example

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

Field representation (hexadecimal):

\[
\begin{array}{cccccccc}
\text{31} & \text{30} & \text{29} & \text{28} & \text{27} & \text{26} & \text{25} & \text{24} \\
0x0 & 0x7 & 0x6 & 0x0 & 0x5 & 0x33 & & \\
\end{array}
\]

Field representation (binary):

\[
\begin{array}{cccccccc}
\text{31} & \text{30} & \text{29} & \text{28} & \text{27} & \text{26} & \text{25} & \text{24} \\
0000000 & 00111 & 00110 & 000 & 00101 & 0110011 & & \\
\end{array}
\]

hex representation: 0x007302B3

decimal representation: 7,537,331

Called a \textbf{Machine Language Instruction}
# 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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>ADD</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>SUB</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td>SLL</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
<td>SLT</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td>SLTU</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>XOR</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td>SRL</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>SRA</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>OR</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>AND</td>
</tr>
</tbody>
</table>

Different encoding in funct7 + funct3 selects different operations
Agenda

- Stored-Program Concept
- R-Format
  - I-Format
  - S-Format
  - SB-Format
  - U-Format
  - UJ-Format
- Summary
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

|   | 31 | 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 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 12 | 5  | 3  | 5  | 7  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

• Field names:

|   | 31 | 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 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | imm[11:0] | rs1 | func3 | rd | opcode |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

• **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
• **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 $[-2^{11}, +2^{11}-1)$
I-Format Example (1/2)

```
addi x15, x1, -40
```

- \( rd = x15 \)
- \( rs1 = x1 \)

<table>
<thead>
<tr>
<th>OPCODES IN NUMERICAL ORDER BY OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNEMONIC</td>
</tr>
<tr>
<td>fence.i</td>
</tr>
<tr>
<td>addi</td>
</tr>
<tr>
<td>slli</td>
</tr>
</tbody>
</table>

```

31 1111110110000 00001 0 01111 0x13
    imm[11:0]  rs1  func3  rd  opcode
```
I-Format Example (2/2)

• RISCV Instruction: `addi x15, x1, -40`

Field representation (binary):

```
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
1111 1101 1000 0000 1 000 0111 1 001 0011
```

hex representation: 0xFD80 8793

decimal representation: 4,253,058,963
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>
<th>ADDI</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0010011</td>
<td>SLTI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0010011</td>
<td>SLTIU</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0010011</td>
<td>XORI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0010011</td>
<td>ORI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0010011</td>
<td>ANDI</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</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)

This is part of the funct7 field
Load Instructions are also I-Type

\[
\begin{array}{cccccc}
31 & \text{imm}[11:0] & rs1 & \text{func3} & rd & \text{opcode} \\
\hline
0 & \text{offset}[11:0] & base & \text{width} & dst & \text{LOAD} \\
\end{array}
\]

- 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 addresses, not to create final result
- Value loaded from memory is stored in \( rd \)
I-Format Load Example

- \texttt{lw \ x14, \ 8(x2)}

```
31
imm[11:0] rs1 func3 rd opcode
0
offset[11:0] base width dst LOAD
0000000001000 00010 010 01110 0000011
imm=+8 rs1=2 LW rd=14 LOAD
```
**All RV32 Load Instructions**

- **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

<table>
<thead>
<tr>
<th>funct3</th>
<th>imm[11:0]</th>
<th>rs1</th>
<th>rd</th>
<th>0000011</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
<td>0000011</td>
</tr>
<tr>
<td>LH</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>001</td>
<td>0000011</td>
</tr>
<tr>
<td>LW</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>0000011</td>
</tr>
<tr>
<td>LBU</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>0000011</td>
</tr>
<tr>
<td>LHU</td>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
<td>0000011</td>
</tr>
</tbody>
</table>

The funct3 field encodes size and signedness of load data.
Agenda

• Stored-Program Concept
• R-Format
• I-Format
• S-Format
• SB-Format
• U-Format
• UJ-Format
• Summary
S-Format Used for Stores

- Store needs to read two registers, \( r_{s1} \) for base memory address, and \( r_{s2} \) for data to be stored, as well as need immediate offset!
- Can’t have both \( r_{s2} \) and immediate in same place as other instructions!
- Note: stores don’t write a value to the register file, no \( r_{d} \)!
- RISC-V design decision is move low 5 bits of immediate to where \( r_{d} \) field was in other instructions – keep \( r_{s1}/r_{s2} \) fields in same place
- *register names more critical than immediate bits in hardware design*

```
|--------|-----------|-----------|-----------|-----------|---------|-------|
```

7/2/20
CS61C Su20 - Lecture 8
**S-Format Example**

\[ \text{sw } x14, \ 8(x2) \]

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

\[
\text{off}[11:5] = 0 \\
\text{rs2}=14 \quad \text{rs1}=2 \quad \text{SW} \quad \text{off}[4:0] = 8 \quad \text{STORE}
\]

combined 12-bit offset = 8
All RV32 Store Instructions

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

• Stored-Program Concept
• R-Format
• I-Format
• S-Format
• **SB-Format**
• U-Format
• UJ-Format
• Summary
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 word offset to PC
  — Branches generally change the PC by a small amount
  — Can specify $\pm 2^{11}$ aligned word 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**

• 32-bit Instructions are “**word-aligned**”: Address is always a multiple of 4 (in bytes)
  — PC ALWAYS points to an instruction
  — Note: if your system supports RISC-V compressed instructions (16 bit instructions), all instructions are **half word-aligned**.

• 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
Example Instruction Compression

Code taken from Tock embedded operating system
RISC-V SB-Format for Branches

- SB-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 =** 4×32-bit instructions = 16 bytes
• **(Branch with offset of 0, branches to itself)**
• 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.
beq x19, x10, offset = 16 bytes

13-bit immediate, imm[12:0], with value 16

imm[0] discarded, always zero
RISC-V Immediate Encoding

Why is it so confusing?!?!

<table>
<thead>
<tr>
<th>Instruction Encodings, inst[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bit immediates produced, imm[31:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</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

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
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:

\[
\text{beq } x10,x0,\text{far} \quad \text{bne } x10,x0,\text{next} \\
\# \text{ next instr} \quad \rightarrow \quad \text{j far} \\
\text{next: } \# \text{ next instr}
\]
Agenda

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

- `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
```
Corner Case

• How to set 0xDEADBEEF?

```assembly
lui x10, 0xDEADB  # x10 = 0xDEADB000
addi x10, x10,0xEEF   # x10 = 0xDEADAE0F
```

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
- We have: 0xDEADB000 + 0xFFFFFEFF = 0xDEADAE0F
How to set 0xDEADBEEF?

\[
\text{lui } x10, \ 0\text{xDEAD}\text{C} \quad \# x10 = 0\text{xDEADC000}
\]

\[
\text{addi } x10, \ x10, 0\text{xEEF} \quad \# x10 = 0\text{DEADBEEF}
\]

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:
\[
\text{li } x10, \ 0\text{xDEADBEEF} \quad \# \text{Creates two instructions}
\]
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
• S-Format
• SB-Format
• U-Format
• **UJ-Format**
• Summary
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
UJ-Format Instructions (2/3)

- `jal` saves PC+4 in register rd (the 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
- 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
### jalr Instruction (I-Format)

**opcode**

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| imm[11:0] | rs1 | func3 | rd | opcode |
| offset | base | 0 | dest | JALR |

- `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
# Uses of jalr

The jalr instruction is used for function calls. It can be used in the following ways:

1. To call a function at any 32-bit absolute address:
   ```
   lui x1, <hi 20 bits>
   jalr ra, x1, <lo 12 bits>
   ``

2. To jump PC-relative with a 32-bit offset:
   ```
   auipc x1, <hi 20 bits>
   jalr x0, x1, <lo 12 bits>
   ```

3. As a pseudo-instruction for ret and jr:
   ```
   ret = jr ra = jalr x0, ra, 0
   jalr x1 = jalr ra, x1, 0
   ```

Note: jalr instruction format:

```
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
imm[11:0] rs1 func3 rd opcode
offset  base  0  dest JALR
```
Agenda

• Stored-Program Concept
• R-Format
• I-Format
• S-Format
• SB-Format
• U-Format
• UJ-Format
• Summary
### Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th></th>
<th>R-type</th>
<th>I-type</th>
<th>S-type</th>
<th>B-type</th>
<th>U-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>funct7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>rs1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>funct3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **R-type**
- **I-type**
- **S-type**
- **B-type**
- **U-type**
- **J-type**
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:

```
00000000  shamt  rsl  001  rd  0010011
00000000  shamt  rsl  101  rd  0010011
01000000  shamt  rsl  101  rd  0010011
00000000  rs2   rsl  000  rd  0110011
00000000  rs2   rsl  000  rd  0110011
00000000  rs2   rsl  001  rd  0110011
00000000  rs2   rsl  000  rd  0110011
00000000  rs2   rsl  010  rd  0110011
00000000  rs2   rsl  010  rd  0110011
00000000  rs2   rsl  011  rd  0110011
00000000  rs2   rsl  110  rd  0110011
00000000  rs2   rsl  111  rd  0110011
```

Not in CS61C

- SLLI
- SRLI
- SRAI
- ADD
- SUB
- SLT
- SLTU
- XOR
- SRL
- SRA
- AND
- FENCE
- FENCE.I
- ECALL
- EBREAK
- CSRWR
- CSRRS
- CSRRC
- CSRWR1
- CSRRS1
- CSRRC1

7/2/20 CS61C Su20 - Lecture 8 65