18-447 Lecture 3:
RISC-V Instruction Set Architecture

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  - get bootstrapped on RISC-V RV32I to start Lab 1
    (will return to visit general ISA issues on 4th meeting)

• Notices
  - Check Canvas and Piazza regularly
  - Student survey (on Canvas), due next Wed
  - H02: Lab 1, Part A, due Week 3
  - H03: Lab 1, Part B, due Week 4

• Readings
  - P&H Ch2 (for today)
  - P&H Ch4.1~4.4 (for next time)
What we mean by "architecture"?
(with quotes)
How to specify a clock design?

function/performance/implementation

• “Architecture”
  – a clock has an hour hand and a minute hand, ….

  Can read a clock w. o. knowing how it keeps time
  Can make a clock w. o. knowing how time is used

• Microarchitecture (think blueprint)
  – a particular clockwork has a certain set of gears
    arranged in a certain configuration

• Realization
  – machined alloy gears vs stamped sheet metal

[Computer Architecture, Blaauw and Brooks, 1997]
How to specify a computer design?

• “Architecture”
  – a computer does ....?????....
    Can read a clock w.o. knowing how it keeps time
    Can make a clock w.o. knowing how time is used

• Microarchitecture (think blueprint)
  – a particular computer design has a certain datapath and a certain control logic

• Realization
  – CMOS vs ECL vs vacuum tubes

[Computer Architecture, Blaauw and Brooks, 1997]
Stored Program Architecture
a.k.a. von Neumann

- Memory holds both program and data
  - instructions and data in a linear memory array
  - instructions can be modified as data
- Sequential instruction processing
  1. program counter (PC) identifies current instruction
  2. fetch instruction from memory
  3. update state (e.g. PC and memory) as a function of current state according to instruction
  4. repeat

Dominant paradigm since its conception
Instruction Set Architecture (ISA):
A Concrete Specification

[images from Wikipedia]
“ISA” in a Nut Shell

- A stable programming target (to last for decades)
  - binary compatibility for SW investments
  - permits adoption of foreseeable technology

Better to compromise immediate optimality for future scalability and compatibility

- Dominant paradigm has been “von Neumann”
  - programmer-visible state: mem, registers, PC, etc.
  - instructions to modified state; each prescribes
    - which state elements are read
    - which state elements—including PC—updated
    - how to compute new values of update state

Atomic, sequential, in-order
3 Instruction Classes (as convention)

- Arithmetic and logical operations
  - fetch operands from specified locations
  - compute a result as a function of the operands
  - store result to a specified location
  - update PC to next sequential instruction address

- Data “movement” operations (no compute)
  - fetch operands from specified locations
  - store operand values to specified locations
  - update PC to next sequential instruction address

- Control flow operations (affects only PC)
  - fetch operands from specified locations
  - compute a branch condition and a target address
  - if “branch condition is true” then PC ← target address
    else PC ← next seq. inst addr
Complete “ISA” Picture

• User-level ISA
  – state and instructions available to user programs
  – single-user abstraction on top a “virtualization”

  For this course and for now, RV32I of RISC-V

• “Virtual Environment” Architecture
  – state and instructions to control virtualization
    (e.g., caches, sharing)
  – user-level, but for need-to-know uses

• “Operating Environment” Architecture
  – state and instructions to implement virtualization
  – privileged/protected access reserved for OS
RV32I Programmer-Visible State

- **Program Counter**: 32-bit “byte” address of current instruction
- **Register File**:
  - **General Purpose Register File**: 32x 32-bit words named x0...x31
  - **Note**: x0=0

- **Memory**: $2^{32}$ by 8-bit locations (4 GBytes) indexed using 32-bit “byte” addresses

*(take this literally for now; magic to come)*
Register-Register ALU Instructions

- Assembly (e.g., register-register addition)
  \[ \text{ADD } rd, \text{ rs1, rs2} \]

- Machine encoding: R-type

<table>
<thead>
<tr>
<th>00000000</th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>3-bit</td>
<td>5-bit</td>
<td>7-bit</td>
</tr>
</tbody>
</table>

- Semantics
  - \( GPR[rd] \leftarrow GPR[rs1] + GPR[rs2] \)
  - \( PC \leftarrow PC + 4 \)

- Exceptions: none (ignore carry and overflow)

- Variations
  - Arithmetic: \{ADD, SUB\}
  - Compare: \{signed, unsigned\} set if less than
  - Logical: \{AND, OR, XOR\}
  - Shift: \{Left, Right-Logical, Right-Arithmetic\}
## R-Type Reg-Reg Instruction Encodings

<table>
<thead>
<tr>
<th></th>
<th>funct7</th>
<th>rs2</th>
<th>rsl</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
<th>R-type</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>
</tr>
<tr>
<td></td>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>rd</td>
<td>0110011</td>
<td>ADD</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>SUB</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
<td>SLL</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td>SLT</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
<td>SLTU</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td>XOR</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRL</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRA</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td>OR</td>
</tr>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>AND</td>
</tr>
</tbody>
</table>

32-bit R-type ALU
Assembly Programming 101

• Break down high-level program expressions into a sequence of elemental operations

• E.g. High-level Code

\[ f = ( g + h ) - ( i + j ) \]

• Assembly Code
  – suppose \( f, g, h, i, j \) are in \( r_f, r_g, r_h, r_i, r_j \)
  – suppose \( r_{\text{temp}} \) is a free register

\[
\begin{align*}
\text{add} & \quad r_{\text{temp}} & r_g & r_h & \quad \# r_{\text{temp}} = g+h \\
\text{add} & \quad r_f & r_i & r_j & \quad \# r_f = i+j \\
\text{sub} & \quad r_f & r_{\text{temp}} & r_f & \quad \# f = r_{\text{temp}} - r_f
\end{align*}
\]
Reg-Immediate ALU Instructions

- Assembly (e.g., reg-immediate additions)
  \[ \text{ADDI } rd, rs1, \text{ imm}_{12} \]
- Machine encoding: I-type
  
<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>12-bit</td>
<td>5-bit</td>
<td>3-bit</td>
<td>5-bit</td>
<td>7-bit</td>
</tr>
</tbody>
</table>

- Semantics
  - GPR[rd] ← GPR[rs1] + sign-extend (imm)
  - PC ← PC + 4
- Exceptions: none (ignore carry and overflow)
- Variations
  - Arithmetic: \{ADDI, SUBI\}
  - Compare: \{signed, unsigned\} set if less than
  - Logical: \{ANDI, ORI, XORI\}
  - **Shifts by unsigned imm[4:0]: \{SLLI, SRLI, SRAI\}**
# I-Type Reg-Immediate ALU Inst. Encodings

<table>
<thead>
<tr>
<th>31</th>
<th>imm[11:0]</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>
<th>I-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADDI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLTI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLTIU</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XORI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ORI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ANDI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLLI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SRLI</td>
</tr>
<tr>
<td></td>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SRAI</td>
</tr>
</tbody>
</table>

- **sign-extended immediate**
- **unsigned**
- **matches**
- **R-type encoding**
- **32-bit I-type ALU**

Note: SLTIU does **unsigned** compare with **sign-extended immediate**

[The RISC-V Instruction Set Manual]
Load-Store Architecture

• RV32I ALU instructions
  – operates only on register operands
  – next PC always PC+4

• A distinct set of load and store instructions
  – dedicated to copying data between register and memory
  – next PC always PC+4

• Another set of “control flow” instructions
  – dedicated to manipulating PC (branch, jump, etc.)
  – does not affect memory or other registers
Load Instructions

• Assembly (e.g., load 4-byte word)
  \[ \text{LW } \text{rd, offset}_{12}(\text{base}) \]

• Machine encoding: I-type

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{offset}[11:0] & \text{base} & \text{010} & \text{rd} & \text{0000011} \\
\hline
12-bit & 5-bit & 3-bit & 5-bit & 7-bit \\
\hline
\end{array}
\]

• Semantics
  – \( \text{byte_address}_{32} = \text{sign-extend}(\text{offset}_{12}) + \text{GPR}[\text{base}] \)
  – \( \text{GPR}[\text{rd}] \leftarrow \text{MEM}_{32}[\text{byte_address}] \)
  – \( \text{PC} \leftarrow \text{PC} + 4 \)

• Exceptions: none for now

• Variations: LW, LH, LHU, LB, LBU
  e.g., LB :: \( \text{GPR}[\text{rd}] \leftarrow \text{sign-extend}(\text{MEM}_8[\text{byte_address}]) \)
  \( \text{LBU} :: \text{GPR}[\text{rd}] \leftarrow \text{zero-extend}(\text{MEM}_8[\text{byte_address}]) \)

\text{RV32I is byte-addressable, little-endian (until v20191213)}
When data size > address granularity

- 32-bit signed or unsigned integer word is 4 bytes
- By convention we “write” MSB on left: 0x40:49:0f:db

- On a byte-addressable machine . . . . . . .

<table>
<thead>
<tr>
<th>MSB</th>
<th>Big Endian</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 0</td>
<td>byte 1</td>
<td>byte 2</td>
</tr>
<tr>
<td>byte 4</td>
<td>byte 5</td>
<td>byte 6</td>
</tr>
<tr>
<td>byte 8</td>
<td>byte 9</td>
<td>byte 10</td>
</tr>
<tr>
<td>byte 12</td>
<td>byte 13</td>
<td>byte 14</td>
</tr>
<tr>
<td>byte 16</td>
<td>byte 17</td>
<td>byte 18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSB</th>
<th>Little Endian</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 3</td>
<td>byte 2</td>
<td>byte 1</td>
</tr>
<tr>
<td>byte 7</td>
<td>byte 6</td>
<td>byte 5</td>
</tr>
<tr>
<td>byte 11</td>
<td>byte 10</td>
<td>byte 9</td>
</tr>
<tr>
<td>byte 15</td>
<td>byte 14</td>
<td>byte 13</td>
</tr>
<tr>
<td>byte 19</td>
<td>byte 18</td>
<td>byte 17</td>
</tr>
</tbody>
</table>

Pointer points to the **big end**

- What difference does it make?
Load/Store Data Alignment

- Access granularity not same as addressing granularity
  - physical implementations of memory and memory interface optimize for natural alignment boundaries (i.e., return an aligned 4-byte word per access)
  - unaligned loads or stores would require 2 separate accesses to memory
- Common for RISC ISAs to disallow misaligned loads/stores; if necessary, use a code sequence of aligned loads/stores and shifts
- RV32I (until v20191213) allowed misaligned loads/stores but warns it could be very slow; if necessary, . . .

<table>
<thead>
<tr>
<th>MSB</th>
<th>byte-3</th>
<th>byte-2</th>
<th>byte-1</th>
<th>byte-0</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte-7</td>
<td>byte-6</td>
<td>byte-5</td>
<td>byte-4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Store Instructions

- Assembly (e.g., store 4-byte word)
  \[ \text{SW } rs2, \text{ offset}_{12}(\text{base}) \]
- Machine encoding: S-type

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>3-bit</td>
<td>5-bit</td>
<td>7-bit</td>
</tr>
</tbody>
</table>

- Semantics
  - \[ \text{byte\_address}_{32} = \text{sign-extend}(\text{offset}_{12}) + \text{GPR}[\text{base}] \]
  - \[ \text{MEM}_{32}[\text{byte\_address}] \leftarrow \text{GPR}[rs2] \]
  - \[ \text{PC} \leftarrow \text{PC} + 4 \]
- Exceptions: none for now
- Variations: SW, SH, SB
  e.g., SB:: \[ \text{MEM}_{8}[\text{byte\_address}] \leftarrow (\text{GPR}[rs2])[7:0] \]
Assembly Programming 201

• E.g. High-level Code

\[ A[8] = h + A[0] \]

where \( A \) is an array of integers (4 bytes each)

• Assembly Code

  – suppose \&A, h are in \( r_A, r_h \)
  – suppose \( r_{temp} \) is a free register

\[
\begin{align*}
\text{LW } & r_{temp} \ 0( r_A ) \quad \# \ r_{temp} = A[0] \\
\text{add } & r_{temp} \  r_h \ r_{temp} \quad \# \ r_{temp} = h + A[0] \\
\text{SW } & r_{temp} \ 32( r_A ) \quad \# \ A[8] = r_{temp} \\
\end{align*}
\]

# note A[8] is 32 bytes from A[0]
Load/Store Encodings

- Both needs 2 register operands and 1 12-bit immediate

<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>
<th>I-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LB</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LH</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LW</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LBU</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LHU</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rsl</td>
<td>100</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>rsl</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>
<th>S-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>SB</td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>SH</td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>SW</td>
</tr>
<tr>
<td>imm[11:5]</td>
<td>rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
RV32I Immediate Encoding

• Most RISC ISAs use 1 register-immediate format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bit</td>
<td>5-bit</td>
<td>5-bit</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

– rt field used as a source (e.g., store) or dest (e.g., load)
– also common to opt for bigger 16-bit immediate

• RV32I adopts 2 different register-immediate formats (I vs S) to keep rs2 operand at inst[24:20] always

• RV32I encodes immediate in non-consecutive bits
### RV32I Instruction Formats

- All instructions 4-byte long and 4-byte aligned in mem
- **R-type**: 3 register operands
  - ![R-type instruction format](image)
- **I-type**: 2 register operands (with dest) and 12-bit imm
  - ![I-type instruction format](image)
- **S/B-type**: 2 register operands (no dest) and 12-bit imm
  - ![S/B-type instruction format](image)
- **U/J-type**: 1 register operand (dest) and 20-bit imm
  - ![U/J-type instruction format](image)

Aimed to simplify decoding and field extraction.
Control Flow Instructions

• C-Code

```c
{ code A }
if X==Y then
    { code B }
else
    { code C }
{ code D }
```

Control Flow Graph

- True
  - code A
    - if X==Y
  - code B
  - code C
- False
  - code D

Assembly Code (linearized)

```
code A
    if X==Y
goto code C
    goto code B
code D
```

Basic blocks (1-way in, 1-way out, all or nothing)
(Conditional) Branch Instructions

- Assembly (e.g., branch if equal)
  
  \[ \text{BEQ } rs1, rs2, \text{imm}_{13} \]
  
  Note: implicit \text{imm}[0]=0
  
  Note: real assembler expects a target label or address

- Machine encoding: B-type

  \[
  \begin{array}{cccccc}
  \text{imm}[12|10:5] & \text{rs2} & \text{rs1} & 000 & \text{imm}[4:1|11] & 1100011 \\
  7-bit & 5-bit & 5-bit & 3-bit & 5-bit & 7-bit
  \end{array}
  \]

- Semantics
  - target = PC + sign-extend(\text{imm}_{13})
  - if GPR[rs1]==GPR[rs2] then PC ← target
  - else PC ← PC + 4

  How far can you jump?

- Exceptions: misaligned target (4-byte) if taken

- Variations
  - BEQ, BNE, BLT, BGE, BLTU, BGEU
Assembly Programming 301

• E.g. High-level Code

```plaintext
if (i == j) then
  e = g
else
  e = h
f = e
```

• Assembly Code

- suppose $e, f, g, h, i, j$ are in $r_e, r_f, r_g, r_h, r_i, r_j$

```plaintext
bne $r_i$ $r_j$ L1  # L1 and L2 are addr labels
             # assembler computes offset
add $r_e$ $r_g$ x0  # e = g
beq x0 x0 L2       # goto L2 unconditionally

L1: add $r_e$ $r_h$ x0  # e = h
L2: add $r_f$ $r_e$ x0  # f = e
```
Assembly Programming 302

- If you write C code:
  ```c
  for (int i=0; i<16; i++) {
    sum+=A[i];
  }
  ```

- GCC –O generates code for:
  ```c
  for (int* a=&A[0]; a<&A[16]; a++) {
    sum+=*a;
  }
  ```

- Assembly Code (suppose sum, A, a are in r\text{sum}, r\text{A}, r\text{a})

```assembly
addi r\text{a} r\text{A} 0       # a=\&A[0]
L1:  lw r\text{tmp} 0(r\text{a})  # sum+==*a
     add r\text{sum} r\text{sum} r\text{tmp}
addi r\text{a} r\text{a} 4       # a++
     addi r\text{tmp} r\text{A} 64  # tmp=\&A[16]
     bltu r\text{a} r\text{tmp} L1
```
Function Call and Return

A function return need to:
1. jump back to different callers
2. know where to jump back to
Jump and Link Instruction

- **Assembly**

  \[ \text{JAL } rd \text{ imm}_{21} \]

  \text{Note: implicit imm[0]=0}

  \text{Note: real assembler expects a target label or address}

- **Machine encoding: J-type**

  \[
  \begin{array}{c|c|c}
  \text{imm[20|10:1|11|19:12]} & \text{rd} & 1101111 \\
  \text{20-bit} & \text{5-bit} & \text{7-bit}
  \end{array}
  \]

- **Semantics**

  - target = PC + sign-extend(imm_{21})
  - GPR[rd] $\leftarrow$ PC + 4
  - PC $\leftarrow$ target

  \text{How far can you jump?}

- **Exceptions: misaligned target (4-byte)**
Jump Indirect Instruction

• Assembly
  \[ \text{JALR } rd, \text{ rs1, imm}_{12} \]

• Machine encoding: I-type
  \[
  \begin{array}{cccccc}
  \text{imm}[11:0] & \text{rs1} & \text{000} & \text{rd} & 1100111 \\
  \text{12-bit} & \text{5-bit} & \text{3-bit} & \text{5-bit} & \text{7-bit}
  \end{array}
  \]

• Semantics
  – target = GPR[rs1] + sign-extend(imm\(_{12}\))
  – target &= 0xffff\_ffe
  – GPR[rd] \leftarrow \text{PC} + 4
  – PC \leftarrow \text{target}

  \text{How far can you jump?}

• Exceptions: misaligned target (4-byte)
Assembly Programming 401

- ..... A → call B → return C → call B → return D ..... 
- How do you pass argument between caller and callee?
- If A set x10 to 1, what is the value of x10 when B returns to C?
- What registers can B use?
- What happens to x1 if B calls another function
Caller and Callee Saved Registers

• Callee-Saved Registers
  – caller says to callee, “The values of these registers should not change when you return to me.”
  – callee says, “If I need to use these registers, I promise to save the old values to memory first and restore them before I return to you.”

• Caller-Saved Registers
  – caller says to callee, “If there is anything I care about in these registers, I already saved it myself.”
  – callee says to caller, “Don’t count on them staying the same values after I am done.

• Unlike endianness, this is not arbitrary

When to use which?
## RISC-V Register Usage Convention

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5–7</td>
<td>t0–2</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10–11</td>
<td>a0–1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12–17</td>
<td>a2–7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18–27</td>
<td>s2–11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28–31</td>
<td>t3–6</td>
<td>Temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>
Memory Usage Convention

- stack space
- free space
- dynamic data
- static data
- text
- reserved

Stack pointer GPR[\(x2\)]

Grow down
Grow up

Binary executable

High address
Low address
Basic Calling Convention

1. caller saves caller-saved registers
2. caller loads arguments into a0~a7 (x10~x17)
3. caller jumps to callee using \textbf{JAL} x1

4. callee allocates space on the stack (dec. stack pointer)
5. callee saves callee-saved registers to stack

\textbf{function}\{ 
\quad \textbf{...... body of callee (can “nest” additional calls) .......} 
\}

6. callee loads results to a0, a1 (x10, x11)
7. callee restores saved register values
8. \textbf{JALR} x0, x1

9. caller continues with return values in a0, a1
Terminologies

- **Instruction Set Architecture**
  - machine state and functionality as observable and controllable by the programmer

- **Instruction Set**
  - set of commands supported

- **Machine Code**
  - instructions encoded in binary format
  - directly consumable by the hardware

- **Assembly Code**
  - instructions in “textual” form, e.g. `add r1, r2, r3`
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code
    (mostly true: compound instructions, labels ....)
We didn’t talk about

- Privileged Modes
  - user vs. supervisor
- Exception Handling
  - trap to supervisor handling routine and back
- Virtual Memory
  - each process has 4-GBytes of private, large, linear and fast memory?
- Floating-Point Instructions