18-447 Lecture 2: 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 revisit general ISA issues in L4)

• Notices
  – Student survey on Canvas, due next Wed
  – H02: Lab 1, Part A, due noon, Friday 2/19
  – H03: Lab 1, Part B, due noon, Friday 2/26

• Readings
  – P&H Ch2
  – P&H Ch4.1~4.4 (next time)
How to specify what a computer does?

- Architectural Level
  - a clock has an hour hand and a minute hand, .....
  - a computer does ....?????....

  You can read a clock without knowing how it works

- Microarchitecture Level
  - a particular clockwork has a certain set of gears arranged in a certain configuration
  - a particular computer design has a certain datapath and a certain control logic

- Realization Level
  - machined alloy gears vs stamped sheet metal
  - CMOS vs ECL vs vacuum tubes

[Computer Architecture, Blaauw and Brooks, 1997]
So what makes a computer a computer?

Having program stored as data is an extremely important step in the evolution of computer architectures.
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 some state (e.g. PC and memory) as a function of current state according to instruction
  4. repeat

Dominant paradigm since its invention
Very Different Architectures Exist

• Consider a von Neumann program
  – what is the significance of the instruction order?
  – what is the significance of the storage locations?

\[
\begin{align*}
v & := a + b; \\
w & := b \times 2; \\
x & := v - w; \\
y & := v + w; \\
z & := x \times y;
\end{align*}
\]

• Dataflow program instruction ordering implied by data dependence
  – instruction specifies who receives the result
  – instruction executes when operands received
  – no program counter, no intermediate state

[Dataflow figure and example from Arvind]
Parallel Random-Access Machine

Do you naturally think parallel or sequential?
Instruction Set Architecture (ISA)
Commercialization in the 50s

- UNIVAC (1951) the first commercial computer contract price $400K, actual cost ~$1M, sold 48 copies
- IBM 701 (1952) “leased” 19 units, $12K per month (www-1.ibm.com/ibm/history/exhibits/701/701_customers.html)
- IBM 650 (1953) sold ~2000 units at $200K ~ 400K
- IBM System/360, 1964 **Redefined Industry!!**
  - a family of *binary compatible* computers (previously, IBM had 4 incompatible lines)
  - 19 combinations of varying speed and memory capacity from $200K ~ $2M
  - ISA still alive today in z/Architecture mainframes
“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”
  – program visible state: memory, 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 the next sequential instruction

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

- 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. instruction
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 Program Visible State

- **Program Counter**: 32-bit “byte” address of current instruction

- **Memory**: 32-bit memory address: $2^{32}$ by 8-bit locations (4 GBytes) (there is some magic going on)

- **Register File**: 32x 32-bit words named x0...x31

  **Note**: x0=0, x1, x2

18-447-S21-L02-S13, James C. Hoe, CMU/ECE/CALCM, ©2021
Register-Register ALU Instructions

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

• Machine encoding

<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
  – \( \text{GPR[rd]} \leftarrow \text{GPR[rs1]} + \text{GPR[rs2]} \)
  – \( \text{PC} \leftarrow \text{PC} + 4 \)

• Exceptions: none (ignore carry and overflow)

• Variations
  – Arithmetic: \{ADD, SUB\}
  – Compare: \{signed, unsigned\} x \{Set if Less Than\}
  – Logical: \{AND, OR, XOR\}
  – Shift: \{Left, Right-Logical, Right-Arithmetic\}
### Reg-Reg Instruction Encodings

<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>R-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></td>
<td></td>
</tr>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rsl</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>
</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>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rsl</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
<td>SLL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td>SLT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
<td>SLTU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td>XOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rsl</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td>OR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rsl</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>AND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[The RISC-V Instruction Set Manual]
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

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

• Assembly (e.g., reg-immediate additions)
  \[ \text{ADDI } rd, rs1, \text{ imm}_{12} \]
• Machine encoding
  \[
  \begin{array}{cccccc}
  \text{imm[11:0]} & \text{rs1} & 000 & \text{rd} & 0010011 \\
  \hline
  \text{12-bit} & \text{5-bit} & \text{3-bit} & \text{5-bit} & \text{7-bit}
  \end{array}
  \]
• Semantics
  – \( GPR[rd] \leftarrow GPR[rs1] + \text{sign-extend (imm)} \)
  – \( PC \leftarrow PC + 4 \)
• Exceptions: none (ignore carry and overflow)
• Variations
  – Arithmetic: \{ADDI, SUBI\}
  – Compare: \{signed, unsigned\} \times \{\text{Set if Less Than Imm}\}
  – Logical: \{ANDI, ORI, XORI\}
  – **Shifts by unsigned imm[4:0]: \{SLLI, SRLI, SRAI\}
### Reg-Immediate ALU Inst. Encodings

<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></td>
<td></td>
<td></td>
<td></td>
<td>ADDI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0010011</td>
</tr>
<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>
</tbody>
</table>

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

---

**32-bit I-type ALU**

<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>00000000</td>
<td>shamt</td>
<td>001</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>101</td>
<td>rd</td>
<td>0010011</td>
</tr>
<tr>
<td>01000000</td>
<td>shamt</td>
<td>101</td>
<td>rd</td>
<td>0010011</td>
</tr>
</tbody>
</table>

**R-type encoding**

[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 effect memory or other registers
Load Instructions

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

- Machine encoding

<table>
<thead>
<tr>
<th>offset[11:0]</th>
<th>base</th>
<th>010</th>
<th>rd</th>
<th>00000011</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
  - byte_address\_32 = sign-extend(offset\_12) + GPR[base]
  - GPR[rd] \leftarrow MEM\_32[byte_address]
  - PC \leftarrow PC + 4

- Exceptions: none for now

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

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

- 32-bit signed or unsigned integer word is 4 bytes
- By convention we “write” MSB on left

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 0</td>
<td>byte 1</td>
</tr>
<tr>
<td>byte 4</td>
<td>byte 5</td>
</tr>
<tr>
<td>byte 8</td>
<td>byte 9</td>
</tr>
<tr>
<td>byte 12</td>
<td>byte 13</td>
</tr>
<tr>
<td>byte 16</td>
<td>byte 17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte 3</td>
<td>byte 2</td>
</tr>
<tr>
<td>byte 7</td>
<td>byte 6</td>
</tr>
<tr>
<td>byte 11</td>
<td>byte 10</td>
</tr>
<tr>
<td>byte 15</td>
<td>byte 14</td>
</tr>
<tr>
<td>byte 19</td>
<td>byte 18</td>
</tr>
</tbody>
</table>

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

- What difference does it make?
  - pointer points to the **big end**
  - pointer points to the **little end**

check out htonl(), ntohl() in in.h

18-447-S21-L02-S21, James C. Hoe, CMU/ECE/CALCM, ©2021
Load/Store Data Alignment

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

- Common case is aligned loads and stores
  - 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, . . .
Store Instructions

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

• Machine encoding

\[
\begin{array}{cccccc}
\text{offset}[11:5] & rs2 & \text{base} & 010 & \text{ofst}[4:0] & 0100011 \\
\text{7-bit} & \text{5-bit} & \text{5-bit} & \text{3-bit} & \text{5-bit} & \text{7-bit}
\end{array}
\]

• Semantics
  – \(\text{byte\_address}_{32} = \text{sign\_extend}(\text{offset}_{12}) + \text{GPR[base]}\)
  – \(\text{MEM}_{32}[\text{byte\_address}] \leftarrow \text{GPR[rs2]}\)
  – \(\text{PC} \leftarrow \text{PC} + 4\)

• 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 $r_{\text{temp}}$ is a free register

```
LW  r_{\text{temp}}  0(r_A)  # r_{\text{temp}} = A[0]
add r_{\text{temp}}  r_h  r_{\text{temp}}  # r_{\text{temp}} = h + A[0]
SW  r_{\text{temp}}  32(r_A)  # A[8] = r_{\text{temp}}
```

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

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

---

**I-type**

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

**S-type**

|----|-----------|-----|-----|--------|----------|--------|
RV32I Immediate Encoding

- RV32I adopts 2 different register-immediate formats (I vs S) to keep rs2 operand at inst[24:20] always
- Most RISCs had 1 register-immediate format
  - \( \text{rt} \) field used as a source (e.g., store) or dest (e.g., load)
  - also common to opt for longer 16-bit immediate
- 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

```
  31 25 24 20 19 15 14 12 11 7 6 0
  | funct7 | rs2 | rs1 | funct3 | rd | opcode |
```

- I-type: 2 register operands (with dest) and 12-bit imm

```
  31 25 24 20 19 15 14 12 11 7 6 0
  | imm[11:0] | rs1 | funct3 | rd | opcode |
```

- S(B)-type: 2 register operands (no dest) and 12-bit imm

```
  31 25 24 20 19 15 14 12 11 7 6 0
```

- U(J)-type, 1 register operation (dest) and 20-bit imm

```
  31 12 11 7 6 0
  | imm[31:12] | rd | opcode |
```

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
  - `if X==Y`
  - `code B`
- False
  - `code C`

### Assembly Code (linearized)

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

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

• Assembly (e.g., branch if equal)
  \[ \text{BEQ } rs1, rs2, \text{imm}_{13} \]  
  \text{Note: implicit \text{imm}[0]=0} 

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

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

```java
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

```assembly
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
```

18-447-S21-L02-S30, James C. Hoe, CMU/ECE/CALCM, ©2021
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
  \[ JAL \text{ rd imm}_{21} \]

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

• Machine encoding

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20-bit</td>
<td>5-bit</td>
<td>7-bit</td>
</tr>
</tbody>
</table>

UJ-type

• Semantics
  – target = PC + sign-extend(imm\textsubscript{21})
  – GPR[rd] \leftarrow PC + 4
  – PC \leftarrow target

  \[ \text{How far can you jump?} \]

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

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

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

• Semantics
  – target = GPR[rs1] + sign-extend(imm_{12})
  – target &\text{=} 0xffff_ffe
  – GPR[rd] \leftarrow \text{PC + 4}
  – \text{PC} \leftarrow \text{target} \quad \text{How far can you jump?}

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

**Caller**

```assembly
... code A ...
JAL x1, _myfxn
... code C ...
JAL x1, _myfxn
... code D ...
```

**Callee**

```assembly
_myfxn: ...
    ... code B ...
    JALR x0, x1, 0
```

- ..... A → \textit{call} B → \textit{return} C → \textit{call} B → \textit{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

- **high address**
  - stack space
    - grow down
    - free space
      - grow up
      - dynamic data
      - static data
      - text
    - reserved
  - binary executable
  - stack pointer GPR[x2]

- **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

\begin{quote}
\ldots\ldots \text{body of callee (can "nest" additional calls)} \ldots\ldots
\end{quote}

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