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?

Processing
- control (sequencing)
- datapath

Storage (program and data)

I/O

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?

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

• 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 Memory

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

<table>
<thead>
<tr>
<th>( \text{M}[0] )</th>
<th>( \text{M}[1] )</th>
<th>( \text{M}[2] )</th>
<th>( \text{M}[3] )</th>
<th>( \text{M}[4] )</th>
<th>( \text{M}[N-1] )</th>
</tr>
</thead>
</table>

\[ x0 = 0 \]

**note**

**x1**

**x2**

**general purpose register file**

32x 32-bit words named \( \text{x0} \ldots \text{x31} \)

32-bit memory address:

\[ 2^{32} \text{ by 8-bit locations (4 GBytes)} \]

(there is some magic going on)
Register-Register ALU Instructions

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

- Machine encoding
  
<table>
<thead>
<tr>
<th>0000000</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] ← GPR[rs1] + GPR[rs2]
  - PC ← 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>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
<th>R-type</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>001</td>
<td>rd</td>
<td>0110011</td>
<td>SLL</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td>SLT</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
<td>SLTU</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td>XOR</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRL</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRA</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</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>

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

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

• Assembly (e.g., reg-immediate additions)
  \[ \text{ADDI} \ rd, \ rs1, \ \text{imm}_{12} \]

• Machine encoding

<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
  – \( \text{GPR}[rd] \leftarrow \text{GPR}[rs1] + \text{sign-extend (imm)} \)
  – \( \text{PC} \leftarrow \text{PC} + 4 \)

• Exceptions: none (ignore carry and overflow)

• Variations
  – Arithmetic: \{ADDI, SUBI\}
  – Compare: \{signed, unsigned\} x \{Set if Less Than Imm\}
  – Logical: \{ANDI, ORI, XORI\}
  – **Shifts by unsigned imm[4:0]: \{SLLI, SRLI, SRAI\}

18-447-S21-L02-S17, James C. Hoe, CMU/ECE/CALCM, ©2021
Reg-Immediate ALU Inst. Encodings

<table>
<thead>
<tr>
<th>31</th>
<th>imm[11:0]</th>
<th>20</th>
<th>19</th>
<th>funct3</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></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ADDI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLTI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SLTIU</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>XORI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ORI</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ANDI</td>
</tr>
</tbody>
</table>

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

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

[The RISC-V Instruction Set Manual]

18-447-S21-L02-S18, James C. Hoe, CMU/ECE/CALCM, ©2021
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)

\[
\text{LW } r_d, \text{ offset}_{12}(\text{base})
\]

• 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] ← MEM\_32[byte_address]
– PC ← PC + 4

• Exceptions: none for now

• Variations: LW, LH, LHU, LB, LBU

  e.g., LB ::  GPR[rd] ← sign-extend(MEM\_8[byte_address])
  LBU ::  GPR[rd] ← zero-extend(MEM\_8[byte_address])

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

<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>

- 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
Load/Store Data Alignment

- 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)

\[ SW \ text{rs2}, \ offset_{12}(\text{base}) \]

• Machine encoding

<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

– \( \text{byte\_address}_{32} = \text{sign-extend}(\offset_{12}) + \text{GPR}[\text{base}] \)
– \( \text{MEM}_{32}[\text{byte\_address}] \leftarrow \text{GPR}[\text{rs2}] \)
– \( \text{PC} \leftarrow \text{PC} + 4 \)

• Exceptions: none for now

• Variations: SW, SH, SB

\( \text{e.g.}, \text{SB::} \quad \text{MEM}_8[\text{byte\_address}] \leftarrow (\text{GPR}[\text{rs2}])[7:0] \)
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

\[
\begin{align*}
\text{LW} & \quad r_{\text{temp}} \ 0(r_A) \quad \# r_{\text{temp}} = A[0] \\
\text{add} & \quad r_{\text{temp}} \ r_h \ r_{\text{temp}} \quad \# r_{\text{temp}} = h + A[0] \\
\text{SW} & \quad r_{\text{temp}} \ 32(r_A) \quad \# A[8] = r_{\text{temp}} \\
& \quad \# \text{note } A[8] \text{ is 32 bytes} \\
& \quad \# \text{ from } A[0]
\end{align*}
\]
Load/Store Encodings

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

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   | 31 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | I-type   |
|   |   |   |   |   |   | 31 | 20 | 25 | 24 | 19 | 15 | 14 | 12 | 11 | 7 |   | Opcode   |

[The RISC-V Instruction Set Manual]
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
  - 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

<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>
</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>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<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>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Aimed to simplify decoding and field extraction
Control Flow Instructions

- C-Code

\[
\{ \text{code A} \} \\
\text{if } X==Y \text{ then} \\
\{ \text{code B} \} \\
\text{else} \\
\{ \text{code C} \} \\
\{ \text{code D} \}
\]

Control Flow Graph

- True
- False

Assembly Code (linearized)

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

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

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

• Semantics
  – target = PC + sign-extend(\text{imm}_{13})
  – if GPR[rs1]==GPR[rs2] then \text{PC} \leftarrow \text{target}
    else \text{PC} \leftarrow \text{PC} + 4
    \text{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 \( r_e, r_f, r_g, r_h, r_i, r_j \)

```assembly
bne ri rj L1       # L1 and L2 are addr labels
                   # assembler computes offset
add re rg x0       # e = g
beq x0 x0 L2       # goto L2 unconditionally
L1:  add re rh x0   # e = h
L2:  add rf re x0   # f = e
```
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**
  
  \[
\text{JAL } \text{rd } \text{imm}_{21}\]

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

- **Machine encoding**

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

  \text{UJ-type}

- **Semantics**

  – target = PC + sign-extend(\text{imm}_{21})
  – \text{GPR}[\text{rd}] \leftarrow \text{PC} + 4
  – \text{PC} \leftarrow \text{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
  
  \[
  \begin{array}{cccc|c}
    \text{imm}[11:0] & rs1 & 000 & rd & 1100111 \\
    \text{12-bit} & \text{5-bit} & \text{3-bit} & \text{5-bit} & \text{7-bit}
  \end{array}
  \]

- Semantics
  - target = GPR[\( rs1 \)] + sign-extend(\( \text{imm}_{12} \))
  - target &\( = \) Oxffff_fffe
  - GPR[\( rd \)] \( \leftarrow \) PC + 4
  - PC \( \leftarrow \) target
  
  How far can you jump?

- Exceptions: misaligned target (4-byte)
• ..... A \rightarrow \text{call} B \rightarrow \text{return} C \rightarrow \text{call} B \rightarrow \text{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: grow down
- Free space: grow up
- Dynamic data
- Static data
- Text
- Reserved

High address

Low address

Stack pointer GPR[x2]

Binary executable
Basic Calling Convention

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

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

    ....... **body of callee** (can “nest” additional calls) .......

6. callee loads results to a0, a1 (x10, x11)
7. callee restores saved register values
8. **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