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 week of 1/28
  – H03: Lab 1, Part B, due week of 2/4

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

• Architectural Level
  - a clock has a 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]
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)
“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

32-bit memory address:
$2^{32}$ by 8-bit locations (4 GBytes)

(there is some magic going on)

**note**  x0=0
x1
x2

general purpose register file
32x 32-bit words named x0...x31
# Register-Register ALU Instructions

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

- **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\}
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-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>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>
</tr>
</tbody>
</table>

| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

32-bit R-type ALU

[from page 54, The RISC-V Instruction Set Manual]
Reg-Immediate ALU Instructions

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

• Machine encoding

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{imm}[11:0]</td>
<td>\text{rs1}</td>
<td>000</td>
<td>\text{rd}</td>
<td>0010011</td>
</tr>
<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\} \times \{Set if Less Than Imm\}
  – Logical: \{ANDI, ORI, XORI\}
  – **Shifts by unsigned \text{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>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.

[from page 54, 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)**
  
  \[
  \text{LW } \text{rd, offset}_{12}(\text{base})
  \]
  
  - **Machine encoding**
    
    + [Offset][11:0] | base | 010 | rd | 0000011
      | 12-bit | 5-bit | 3-bit | 5-bit | 7-bit
    
  - **Semantics**
    
    - byte_address\textsubscript{32} = sign-extend(offset\textsubscript{12}) + GPR[base]
    - GPR[rd] ← MEM\textsubscript{32}[byte_address]
    - PC ← PC + 4
  
  - **Exceptions:** none for now
  
  - **Variations:** LW, LH, LHU, LB, LBU

  - e.g., LB :: GPR[rd] ← sign-extend(MEM\textsubscript{8}[byte_address])
  - LBU :: GPR[rd] ← zero-extend(MEM\textsubscript{8}[byte_address])

  **Note:** RV32I memory is byte-addressable, little-endian
Big Endian vs. Little Endian

(Part I, Chapter 4, Gulliver’s Travels)

• 32-bit signed or unsigned integer word is 4 bytes

• By convention we “write” MSB on left

• 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**

pointer points to the **little end**

• What difference does it make?

  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 allows misaligned loads/stores but warns it could be very slow; if necessary, . . . . . .
Store Instructions

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

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

- 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 \( r_{\text{temp}} \) is a free register

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

\# 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>Field</th>
<th>I-type</th>
<th>S-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>LB</td>
<td>SB</td>
</tr>
<tr>
<td>rsl</td>
<td>LH</td>
<td>SH</td>
</tr>
<tr>
<td>funct3</td>
<td>LW</td>
<td>SW</td>
</tr>
<tr>
<td>rd</td>
<td>LBU</td>
<td></td>
</tr>
<tr>
<td>opcode</td>
<td>LHU</td>
<td></td>
</tr>
</tbody>
</table>

[from page 54, 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

<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>

RV32I Instruction Formats

- All instructions 4-byte long and 4-byte aligned in mem
- R-type: 3 register operands
  
  \[
  \begin{array}{cccccccccc}
  31 & 25 & 24 & 20 & 19 & 15 & 14 & 12 & 11 & 7 & 6 & 0 \\
  \hline
  funct7 & & & & rs2 & & & & rs1 & & & funct3 & rd & & & opcode \\
  \end{array}
  \]

- I-type: 2 register operands (with dest) and 12-bit imm
  
  \[
  \begin{array}{cccccccccc}
  31 & 20 & 19 & 15 & 14 & 12 & 11 & 7 & 6 & 0 & 0 \\
  \hline
  \text{imm[11:0]} & & & & rs1 & & & & funct3 & rd & & & opcode \\
  \end{array}
  \]

- S(B)-type: 2 register operands (no dest) and 12-bit imm
  
  \[
  \begin{array}{cccccccccc}
  31 & 25 & 24 & 20 & 19 & 15 & 14 & 12 & 11 & 7 & 6 & 0 & 0 \\
  \hline
  \end{array}
  \]

- U(J)-type, 1 register operation (dest) and 20-bit imm
  
  \[
  \begin{array}{cccccccccc}
  31 & 12 & 11 & 7 & 6 & 0 & 0 \\
  \hline
  \text{imm[31:12]} & & & & rd & & & & opcode \\
  \end{array}
  \]

Aimed to simplify decoding and field extraction
Control Flow Instructions

- **C-Code**

\[
\begin{align*}
&\{ \text{code A} \} \\
&\text{if } X == Y \text{ then} \\
&\quad \{ \text{code B} \} \\
&\text{else} \\
&\quad \{ \text{code C} \} \\
&\quad \{ \text{code D} \}
\end{align*}
\]

**Control Flow Graph**

- **True**
  - code A
  - if $X == Y$
  - code B
  - code C
  - code D

- **False**
  - code A
  - if $X == Y$
  - goto code C
  - goto code B
  - code D

**Assembly Code** (linearized)

- code A
  - if $X == Y$
  - goto
- code C
  - goto
- 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)
  \[
  \text{BEQ } rs1, rs2, \text{ imm}_{13} \quad \text{Note: implicit } \text{imm}[0]=0
  \]

- 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
  - target = PC + sign-extend(imm<sub>13</sub>)
  - if GPR[rs<sub>1</sub>] == GPR[rs<sub>2</sub>] 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 L2
L1: add r_e r_h x0  # e = h
L2: add r_f r_e x0  # f = e
```

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

• Assembly

\[
\text{JAL } \text{rd } \text{imm}_{21} \quad \text{Note: implicit } \text{imm}[0]=0
\]

• Machine encoding

| imm[20|10:1|11|19:12] | rd | 1101111 | UJ-type |
|---------------------|----|---------|---------|
| 20-bit              | 5-bit | 7-bit   |         |

• Semantics

– target = PC + sign-extend(imm\text{21})
– GPR[rd] ← PC + 4
– PC ← target

How far can you jump?

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

• Assembly
  
  JALR rd, rs1, imm_{12}

• Machine encoding

<table>
<thead>
<tr>
<th></th>
<th>imm[11:0]</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>1100111</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>
<td></td>
</tr>
</tbody>
</table>

• Semantics
  
  – target = GPR[rs1] + sign-extend(imm_{12})
  
  – target &= 0xffff_ffe
  
  – GPR[rd] ← PC + 4
  
  – PC ← target

  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
```

- ..... \(A \rightarrow_{\text{call}} B \rightarrow_{\text{return}} C \rightarrow_{\text{call}} B \rightarrow_{\text{return}} D \) ..... 
- 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>

[from page 100, The RISC-V Instruction Set Manual]
Memory Usage Convention

- **High Address:**
  - Stack space
    - Grow down
  - Free space
    - Grow up
  - Dynamic data
  - Static data
  - Text
- **Low Address:**
  - Reserved

- 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