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 L04)

- **Notices**
  - Student survey on Blackboard, **due 1/25**
  - Lab 1, Part A, **due week of 1/30**
  - Lab 1, Part B, **due week of 2/6**

- **Readings**
  - RISC-V Instruction Set Manual, Ch 1, 2, 20
  - P&H Ch 2
  - P&H 4.1~4.4 (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 one works

- **Implementation Level**
  - A particular clock design 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 versus stamped sheet metal gears  
  - CMOS versus ECL versus vacuum tubes

[Computer Architecture, Blaauw and Brooks, 1997]

---

Stored Program Architecture

a.k.a. von Neumann

- By far the most common architectural paradigm
- Memory holds both program and data
  - instructions and data in a linear memory array  
  - instructions can be modified just like data
- Sequential instruction processing
  1. **program counter (PC)** identifies the current instruction
  2. instruction is fetched from memory
  3. instruction execution causes some state (e.g. memory) to be updated as a specific function of current state
  4. program counter is advanced (according to instruction)
  5. repeat

* atomic  
* sequential  
* inorder
von Neumann vs Dataflow

- Consider a von Neumann program
  - What is the significance of the program 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
  - an instruction executes when all operands received
  - no program counter, no intermediate state

Which one feels more natural to you?

Parallel Random Access Memory

Do you naturally think parallel or sequential?
Instruction Set Architecture (ISA)

“ISA” in a nut shell

- A stable programming target (typically 15~20 years)
  - guarantees binary compatibility for SW investments
  - permits adoption of foreseeable technology advances
    better to compromise immediate optimality for
    future scalability and compatibility

- Dominant paradigm has been “von Neumann”
  - program visible state
    • memory, registers, program counters, etc.
  - instructions to modified state; each prescribes
    • which state elements are read as operands
    • which state elements are updated; and how to
      compute new values from current state
    • where is the next instruction
3 Instruction Classes (as convention)

- Arithmetic and logical operations
  - fetch operands from specified locations \( \text{(reg and/or mem)} \)
  - 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 \( \text{(no compute)} \)
  - fetch operands from specified locations
  - store operand values to specified locations
  - update PC to the next sequential instruction

- Control flow operations \( \text{(affects only PC)} \)
  - fetch operands from specified locations
  - compute a branch condition and a target address
  - if “branch condition is true” then PC \( \leftarrow \) target address
    else PC \( \leftarrow \) next seq. instruction

Complete ISA Picture

- User-level ISA
  - program visible state and instructions available to user programs
  - single-user abstraction on top of HW/SW 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 not used by your average user programs

- “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 the current instruction

|------|------|------|------|------|--------|

**Note:** x0=0

x1
x2

General Purpose Register File
32 32-bit words
named x0...x31

Memory
$2^{32}$ by 8-bit locations (4 Giga Bytes)
32-bit address
(there is some magic going on)

Register-Register ALU Instructions

◆ Assembly (e.g., register-register addition)
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}
Reg-Reg Instruction Encodings

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

32-bit R-type ALU

[from page 54, *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
    \[
    \begin{align*}
    \text{add } & r_{\text{temp}} \ r_g \ r_h \ # \ r_{\text{temp}} = g+h \\
    \text{add } & r_{\text{f}} \ r_i \ r_j \ # \ r_{\text{f}} = i+j \\
    \text{sub } & r_{\text{f}} \ r_{\text{temp}} \ r_{\text{f}} \ # \ f = r_{\text{temp}} - r_{\text{f}}
    \end{align*}
    \]
Reg-Immediate ALU Instructions

- Assembly (e.g., reg-immediate additions)
  ADDI rd, rs1, 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
  - GPR[rd] ← GPR[rs1] + sign-extend (imm)
  - PC ← PC + 4
- Exceptions: none (ignore carry and overflow)
- Variations
  - Arithmetic: {ADDI, SUBI}
  - Compare: {signed, unsigned} x {Set if Less Than Immediate}
  - Logical: {ANDI, ORI, XORI}
  - **Shifts by unsigned imm[4:0]: {SLLI, SRLI, SRAI}

Reg-Immediate ALU Inst. Encodings

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)
  LW rd, offset_{12}(base)

- Machine encoding

<table>
<thead>
<tr>
<th>offset[11:0]</th>
<th>base</th>
<th>010</th>
<th>rd</th>
<th>0000011</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])

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>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>

pointer points to the big end

<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>

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
- Was 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)
  
  \( SW \) \( rs2, \text{offset}_{12}(\text{base}) \)

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

- 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

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

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

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

RV32I Instruction Formats

- All instructions 4-byte long and 4-byte aligned in mem
- R-type: 3 register operands
- I-type: 2 register operands (with dest) and 12-bit imm
- S(B)-type: 2 register operands (no dest) and 12-bit imm
- U(J)-type, 1 register operation (dest) and 20-bit imm

Aimed to simplify decoding and field extraction
RV32I Immediate Encoding

- RV32I adopts 2 different register-immediate formats (I vs S) to keep dest operand at inst[11:7] always
- Most RISCs had 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 longer 16-bit immediate

- RV32I encodes immediate in non-consecutive bits

Control Flow Instructions

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

  Control Flow Graph

  ![Control Flow Graph](image)

  Assembly Code (linearized)

  ```assembly
  code A
  if X==Y
    goto code C
  else
    goto code B
  ```

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

- Assembly (e.g., branch if equal)
  
  \[
  \text{BEQ } rs1, \text{ rs2, imm}_{13}
  \]

  Note: implicit imm[0] (always 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_{13})
  - if GPR[rs1]==GPR[rs2] then \( PC \leftarrow \text{target} \)
  - else \( PC \leftarrow PC + 4 \)

  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

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

- Assembly Code

  ```
  bne r_i, r_j, L1  # L1 and L2 are addr labels
  add r_e, r_g, x0  # assembler computes offset
  beq x0, x0, L2
  L1: add r_e, r_h, x0  # e = h
  L2: add r_f, r_e, 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 Instruction

- **Assembly**
  
  JAL rd imm_{21}

  Note: implicit imm[0] (always 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>

- **Semantics**

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

  How far can you jump?

- **Exceptions:** misaligned target (4-byte)

  *Note*: use “JAL x0 label” instead of “BEQ x0 x0 label”
Jump Indirect Instruction

- Assembly
  
  JALR rd, rs1, imm\(_{12}\)

- Machine encoding

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

- ..... 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>

[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
  - Binary executable

- **Low address**
  - Reserved

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
   - Body
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 expressed in “textual” format
    - e.g. `add r1, r2, r3`
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code
    (mostly true: compound instructions, address 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**