The RISC-V Processor

Hakim Weatherspoon  
CS 3410  
Computer Science  
Cornell University  

[Weatherspoon, Bala, Bracy, and Sirer]
Announcements

• Make sure to go to your Lab Section this week
• Completed Proj1 due Friday, Feb 15th
• Note, a Design Document is due when you submit Proj1 final circuit
• Work alone

BUT use your resources
  • Lab Section, Piazza.com, Office Hours
  • Class notes, book, Sections, CSUGLab
Announcements

Check online syllabus/schedule
• http://www.cs.cornell.edu/Courses/CS3410/2019sp/schedule
• Slides and Reading for lectures
• Office Hours
• Pictures of all TAs
• Project and Reading Assignments
• Dates to keep in Mind
  • Prelims: Tue Mar 5th and Thur May 2nd
  • Proj 1: Due next Friday, Feb 15th
  • Proj3: Due before Spring break
  • Final Project: Due when final will be Feb 16th

Schedule is subject to change
Collaboration, Late, Re-grading Policies

“White Board” Collaboration Policy
• Can discuss approach together on a “white board”
• Leave, watch a movie such as Black Lightening, then write up solution independently
• Do not copy solutions

Late Policy
• Each person has a total of five “slip days”
• Max of two slip days for any individual assignment
• Slip days deducted first for any late assignment, cannot selectively apply slip days
• For projects, slip days are deducted from all partners
• 25% deducted per day late after slip days are exhausted

Regrade policy
• Submit written request within a week of receiving score
Announcements

• Level Up (optional enrichment)
  • Teaches CS students tools and skills needed in their coursework as well as their career, such as Git, Bash Programming, study strategies, ethics in CS, and even applying to graduate school.
  • Thursdays at 7-8pm in 310 Gates Hall, starting this week
Big Picture: Building a Processor

A single cycle processor
Goal for the next few lectures

• Understanding the basics of a processor
  • We now have the technology to build a CPU!

• Putting it all together:
  • Arithmetic Logic Unit (ALU)
  • Register File
  • Memory
    - SRAM: cache
    - DRAM: main memory
  • RISC-V Instructions & how they are executed
RISC-V Register File

- RISC-V register file
  - 32 registers, 32-bits each
  - x0 wired to zero
  - Write port indexed via RW
    - on falling edge when WE=1
  - Read ports indexed via RA, RB

Dual-Read-Port
Single-Write-Port
32 x 32
Register File
RISC-V Register File

- **RISC-V register file**
  - 32 registers, 32-bits each
  - x0 wired to zero
  - Write port indexed via RW
    - on falling edge when WE = 1
  - Read ports indexed via RA, RB

- **RISC-V register file**
  - Numbered from 0 to 31
  - Can be referred by number: x0, x1, x2, … x31
  - Convention, each register also has a name:
    - x10 – x17 → a0 – a7, x28 – x31 → t3 – t6
A single cycle processor
RISC-V Memory

- 32-bit address
- 32-bit data (but byte addressed)
- Enable + 2 bit memory control (mc)

00: read word (4 byte aligned)
01: write byte
10: write halfword (2 byte aligned)
11: write word (4 byte aligned)
Putting it all together: Basic Processor

A single cycle processor
To make a computer

Need a program
• Stored program computer

Architectures
• von Neumann architecture
• Harvard (modified) architecture
To make a computer

Need a program
• Stored program computer
• (a Universal Turing Machine)

Architectures
• von Neumann architecture
• Harvard (modified) architecture
Putting it all together: Basic Processor

A RISC-V CPU with a (modified) Harvard architecture

- Modified: instructions & data in common address space, separate instr/data caches can be accessed in parallel
Takeaway

A processor executes instructions
- Processor has some internal state in storage elements (registers)

A memory holds instructions and data
- (modified) Harvard architecture: separate insts and data
- von Neumann architecture: combined inst and data

A bus connects the two

We now have enough building blocks to build machines that can perform non-trivial computational tasks
Next Goal

• How to program and execute instructions on a RISC-V processor?
Instruction Processing

A basic processor
- fetches
- decodes
- executes one instruction at a time

Instructions: stored in memory, encoded in binary
0010000000000100000000000001010
00100000000000010000000000000000
00000000001000100101010

Figures:
- Prog Mem: Program Memory
- Reg. File: Register File
- ALU: Arithmetic Logic Unit
- Data Mem: Data Memory
- PC: Program Counter
- +4: Increment by 4
- control: Control signals
- inst: Instruction
Levels of Interpretation: Instructions

High Level Language
- C, Java, Python, ADA, …
- Loops, control flow, variables

```
for (i = 0; i < 10; i++)
    printf(“go cucs”);
```

Assembly Language
- No symbols (except labels)
- One operation per statement
- “human readable machine language”

```
main: addi x2, x0, 10
    addi x1, x0, 0
loop: slt x3, x1, x2
...```

Machine Language
- Binary-encoded assembly
- Labels become addresses
- The language of the CPU

```
000000001010001000000000100011
001000000000001000000000010000
00000000001000100001100000101010
```

Instruction Set Architecture
- ALU, Control, Register File, …

Machine Implementation (Microarchitecture)
Instruction Set Architecture (ISA)

Different CPU architectures specify different instructions.

Two classes of ISAs

- **Reduced Instruction Set Computers (RISC)**
  - IBM Power PC, Sun Sparc, MIPS, Alpha
- **Complex Instruction Set Computers (CISC)**
  - Intel x86, PDP-11, VAX

Another ISA classification: **Load/Store Architecture**

- Data must be in registers to be operated on.
  - For example: array\[x\] = array\[y\] + array\[z\]
    - 1 add ? OR 2 loads, an add, and a store ?
- Keeps HW simple → many RISC ISAs are load/store
Takeaway

A RISC-V processor and ISA (instruction set architecture) is an example of Reduced Instruction Set Computers (RISC) where simplicity is key, thus enabling us to build it!!
Next Goal

How are instructions executed?
What is the general **datapath** to execute an instruction?
Five Stages of RISC-V Datapath

A single cycle processor – this diagram is not 100% spatial
Five Stages of RISC-V Datapath

Basic CPU execution loop
1. Instruction Fetch
2. Instruction Decode
3. Execution (ALU)
4. Memory Access
5. Register Writeback
Stage 1: Instruction Fetch

Fetch 32-bit instruction from memory
Increment PC = PC + 4
Stage 2: Instruction Decode

Gather data from the instruction
Read opcode; determine instruction type, field lengths
Read in data from register file
(0, 1, or 2 reads for \texttt{jump}, \texttt{addi}, or \texttt{add}, respectively)
Stage 3: Execution (ALU)

Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)
Load/Store? lw x2, x3, 32 → Compute address
Stage 4: Memory Access

- Used by load and store instructions only
- Other instructions will skip this stage

Fetch → Decode → Execute → Memory → WB
Stage 5: Writeback

Write to register file
- For arithmetic ops, logic, shift, etc, load. What about stores?

Update PC
- For branches, jumps
Takeaway

• The datapath for a RISC-V processor has five stages:
  1. Instruction Fetch
  2. Instruction Decode
  3. Execution (ALU)
  4. Memory Access
  5. Register Writeback

• This five stage datapath is used to execute all RISC-V instructions
Next Goal

- Specific datapaths RISC-V Instructions
RISC-V Design Principles

Simplicity favors regularity
  • 32 bit instructions

Smaller is faster
  • Small register file

Make the common case fast
  • Include support for constants

Good design demands good compromises
  • Support for different type of interpretations/classes
Instruction Types

- **Arithmetic**
  - add, subtract, shift left, shift right, multiply, divide

- **Memory**
  - load value from memory to a register
  - store value to memory from a register

- **Control flow**
  - conditional jumps (branches)
  - jump and link (subroutine call)

- **Many other instructions are possible**
  - vector add/sub/mul/div, string operations
  - manipulate coprocessor
  - I/O
RISC-V Instruction Types

**Arithmetic/Logical**
- **R-type**: result and two source registers, shift amount
- **I-type**: result and source register, shift amount in 16-bit immediate with sign/zero extension
- **U-type**: result register, 16-bit immediate with sign/zero extension

**Memory Access**
- **I-type** for loads and **S-type** for stores
- Load/store between registers and memory
- Word, half-word and byte operations

**Control flow**
- **U-type**: jump-and-link
- **I-type**: jump-and-link register
- **S-type**: conditional branches: pc-relative addresses
RISC-V instruction formats

All RISC-V instructions are 32 bits long, have 4 formats

- **R-type**
  \[
  \text{funct7} \quad \text{rs2} \quad \text{rs1} \quad \text{funct3} \quad \text{rd} \quad \text{op}
  \]
  
  7 bits  5 bits  5 bits  3 bits  5 bits  7 bits

- **I-type**
  \[
  \text{imm} \quad \text{rs1} \quad \text{funct3} \quad \text{rd} \quad \text{op}
  \]
  
  12 bits  5 bits  3 bits  5 bits  7 bits

- **S-type**
  \[
  \text{imm} \quad \text{rs2} \quad \text{rs1} \quad \text{funct3} \quad \text{imm} \quad \text{imm} \quad \text{op}
  \]
  
  7 bits  5 bits  5 bits  3 bits  5 bits  7 bits

- **U-type**
  \[
  \text{imm} \quad \text{rd} \quad \text{op}
  \]
  
  20 bits  5 bits  7 bits
### R-Type (1): Arithmetic and Logic

<table>
<thead>
<tr>
<th>op</th>
<th>funct3</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110011</td>
<td>000</td>
<td>ADD rd, rs1, rs2</td>
<td>(R[rd] = R[rs1] + R[rs2])</td>
</tr>
<tr>
<td>0110011</td>
<td>000</td>
<td>SUB rd, rs1, rs2</td>
<td>(R[rd] = R[rs1] - R[rs2])</td>
</tr>
<tr>
<td>0110011</td>
<td>110</td>
<td>OR rd, rs1, rs2</td>
<td>(R[rd] = R[rs1] \lor R[rs2])</td>
</tr>
<tr>
<td>0110011</td>
<td>100</td>
<td>XOR rd, rs1, rs2</td>
<td>(R[rd] = R[rs1] \oplus R[rs2])</td>
</tr>
</tbody>
</table>
Arithmetic and Logic

Diagram:
- Prog. Mem
- Reg. File
- ALU
- Control
- PC
- Fetch
- Decode
- Execute
- Memory
- WB

Flow:
1. Fetch
2. Decode
3. Execute
4. Memory (skip)
5. WB
### R-Type (2): Shift Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>funct3</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110011</td>
<td>001</td>
<td>SLL rd, rs1, rs2</td>
<td>$R[rd] = R[rs1] &lt;&lt; R[rs2]$</td>
</tr>
<tr>
<td>0110011</td>
<td>101</td>
<td>SRL rd, rs1, rs2</td>
<td>$R[rd] = R[rs1] &gt;&gt;&gt; R[rs2]$ (zero ext.)</td>
</tr>
<tr>
<td>0110011</td>
<td>101</td>
<td>SRA rd, rs1, rs2</td>
<td>$R[rd] = R[rt] &gt;&gt;&gt; R[rs2]$ (sign ext.)</td>
</tr>
</tbody>
</table>
Shift

Prog. Mem -> Reg. File -> ALU

PC +4 -> 5 5 5

Fetch Decode Execute Memory skip WB
# I-Type (1): Arithmetic w/ immediates

<table>
<thead>
<tr>
<th>op</th>
<th>funct3</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010011</td>
<td>000</td>
<td>ADDI rd, rs1, imm</td>
<td>(R[rd] = R[rs1] + \text{imm})</td>
</tr>
<tr>
<td>0010011</td>
<td>111</td>
<td>ANDI rd, rs1, imm</td>
<td>(R[rd] = R[rs1] &amp; \text{zero_extend(imm)})</td>
</tr>
<tr>
<td>0010011</td>
<td>110</td>
<td>ORI rd, rs1, imm</td>
<td>(R[rd] = R[rs1] \mid \text{zero_extend(imm)})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>
Arithmetic w/ immediates
U-Type (1): “Load” Upper Immediate

```
000000000000000000001011001010110111
    imm      rd      op
  20 bits  5 bits  7 bits
```

<table>
<thead>
<tr>
<th>op</th>
<th>mnemonic</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110111</td>
<td>LUI rd, imm</td>
<td>R[rd] = imm &lt;&lt; 16</td>
</tr>
</tbody>
</table>
Load Upper Immediate

Diagram showing the pipeline stages of fetching, decoding, executing, and writing back (WB), with connections to the program memory (Prog. Mem), register file (Reg. File), ALU, and control signals. The diagram also highlights the immediate (imm), shamt, extend, and control signals.
RISC-V Instruction Types

• Arithmetic/Logical
  • **R-type**: result and two source registers, shift amount
  • **I-type**: result and source register, shift amount in 16-bit immediate with sign/zero extension
  • **U-type**: result register, 16-bit immediate with sign/zero extension

• Memory Access
  • **I-type** for loads and **S-type** for stores
  • load/store between registers and memory
  • word, half-word and byte operations

• Control flow
  • **U-type**: jump-and-link
  • **I-type**: jump-and-link register
  • **S-type**: conditional branches: pc-relative addresses
RISC-V Instruction Types

• **Arithmetic/Logical**
  - **R-type**: result and two source registers, shift amount
  - **I-type**: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - **U-type**: result register, 16-bit immediate with sign/zero extension

• **Memory Access**
  - **I-type** for loads and **S-type** for stores
  - load/store between registers and memory
  - word, half-word and byte operations

• **Control flow**
  - **U-type**: jump-and-link
  - **I-type**: jump-and-link register
  - **S-type**: conditional branches: pc-relative addresses
Summary

We have all that it takes to build a processor!
• Arithmetic Logic Unit (ALU)
• Register File
• Memory

RISC-V processor and ISA is an example of a Reduced Instruction Set Computers (RISC)
• Simplicity is key, thus enabling us to build it!

We now know the data path for the MIPS ISA:
• register, memory and control instructions