Building a Single-Cycle RISC-V Processor

Reminders:
Lab 3 due today
Lab 4 out today
Compilers Wrap-Up
Code Generation

- Translate generated IR to assembly

- Register allocation: Map variables to registers
  - If variables > registers, map some to memory, and load/store them when needed

- Translate each assignment to instructions
  - Some assignments may require > 1 instr if our ISA doesn’t have op

- Emit each basic block: label, assignments, and branch or jump

- Lay out basic blocks, removing superfluous jumps

- ISA and CPU-specific optimizations
  - e.g., if possible, reorder instructions to improve performance
Putting It All Together: GCD

Source code

```c
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

IR

```c
start
x = 3
y = x + 7
if (x != y)
    if (x > y)
        x = x - y;
    else {
        y = y - x;
    }
if (x != y)
end
```

Optimized IR

```c
start
x = 3
y = 10
if (x > y)
    x = x - y;
else {
    y = y - x;
}
if (x != y)
end
```
Putting It All Together: GCD

1. Allocate registers:
   x: t0, y: t1

   BBL0:
   li t0, 3
   li t1, 10
   j BBL1

   BBL1:
   slt t2, t1, t0
   bnez t2, BBL2
   j BBL3

   BBL2:
   sub t0, t0, t1
   j BBL4

   BBL3:
   sub t1, t1, t0
   j BBL4

   BBL4:
   beq t0, t1, end
   j BBL1

   j BBL1

2. Produce each basic block for target processor:

   BBL0:  li t0, 3
          li t1, 10
          j BBL1

   BBL1:  slt t2, t1, t0
          bnez t2, BBL2
          j BBL3

   BBL2:  sub t0, t0, t1
          j BBL4

   BBL3:  sub t1, t1, t0
          j BBL4

   BBL4:  beq t0, t1, end
          j BBL1

   j BBL1

3. Lay out BBLs, removing superfluous branches:

   BBL0:  li t0, 3
          li t1, 10
          j BBL1

   BBL1:  slt t2, t1, t0
          bnez t2, BBL2
          j BBL4

   BBL2:  sub t0, t0, t1
          j BBL4

   BBL3:  sub t1, t1, t0
          j BBL4

   BBL4:  beq t0, t1, end
          j BBL1

   end:
Summary: Modern Compilers

**Frontend (analysis)**
- Produces IR if correct program
- Produces meaningful errors

**Backend (synthesis)**
- Produces optimized program

**Source code**
- Tokens
- Syntax tree
- Type-checked syntax tree

**High-quality assembly (often > hand-coded!)**

**Steps**
1. Lexical analysis
2. Syntactic analysis
3. Semantic analysis
4. Generate IR
5. Optimize IR
6. Generate ASM

**Notes**
- Produced IR if correct program
- Produces meaningful errors
- IR produces optimized program
- High-quality assembly (often > hand-coded!)

March 16, 2023
Building a Single-Cycle RISC-V Processor
The von Neumann Model

- Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:

  - **Main memory** holds programs and their data
  - **Central processing unit** accesses and processes memory values
  - **Input/output devices** to communicate with the outside world
Key Idea: Stored-Program Computer

- Express program as a sequence of **coded instructions**
- Memory holds both data and instructions
- CPU fetches, interprets, and executes successive instructions of the program

**Central Processing Unit**

```plaintext
<table>
<thead>
<tr>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>instruction</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>data</td>
</tr>
</tbody>
</table>
```

**How does CPU distinguish between instructions and data?**

```
0xba5eba11
```

**rd <= op(rs1, rs2)**
Anatomy of a von Neumann Computer

- **Instructions** coded as binary data
- **Program Counter** or PC: Address of the instruction to be executed
- Logic to translate instructions into control signals for datapath
Instructions

- Instructions are the fundamental unit of work

- Each instruction specifies:
  - An operation or opcode to be performed
  - Source operands and destination for the result

- In a von Neumann machine, instructions are executed sequentially
  - CPU logically implements this loop:
  - By default, the next PC is current PC + size of current instruction unless the instruction says otherwise
Processor Performance

- “Iron Law” of performance:

\[
\text{Perf} = \frac{1}{\text{Time}} = \frac{\text{Time}}{\text{Program}} \cdot \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}
\]

- Options to reduce execution time:
  - Executed instructions $\downarrow$ (work/instruction $\uparrow$)
  - Cycles per instruction (CPI) $\downarrow$
  - Cycle time $\downarrow$ (frequency $\uparrow$)

- Today: Simple single-cycle processor, executes one instruction from start to end each clock cycle
  - CPI = 1, but low frequency
  - Later: Pipelining to increase frequency
Approach: Incremental Featurism

We’ll implement datapaths for each instruction class individually, and merge them (using MUXes)

Steps:
1. ALU instructions
2. Load & store instructions
3. Branch & jump instructions

Component Repertoire:
- Registers
- Muxes
- "Black box" ALU
- Register File
- Memories

March 16, 2023
MIT 6.191 Spring 2023
Multi-Ported Register File

2 combinational READ ports*, 1 clocked WRITE port

*internal logic ensures Reg[0] reads as 0
Register File Timing

2 combinational READ ports, 1 clocked WRITE port

What if WA=RA1?
RD1 reads “old” value of Reg[RA1] until next clock edge!

March 16, 2023
MIT 6.191 Spring 2023
Memory Timing

- For now (lab 4), we will assume that our memories behave just like our register file in terms of timing
  - Loads are combinational: Data is returned in the same clock cycle as load request
  - Stores are clocked
  - In lab 4, you will see the memory module referred to as a magic memory, since it’s not realistic

- Next week we will learn about the implementation of these memories
- In following labs and design project, we will use realistic memories for our processor
Instruction Fetch/Decode

Use Program Counter (PC) to fetch the next instruction:

- Use PC as memory address
- Add 4 to PC, load new value at end of cycle
- Fetch instruction from memory
- Decode instruction:
  - Use some instruction fields directly (register indexes, immediate values)
  - Use opcode, funct3, and funct7 bits to generate control signals
# ALU Instructions

Differ only in the ALU op to be performed

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD rd, rs1, rs2</td>
<td>Add</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] + \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>SUB rd, rs1, rs2</td>
<td>Sub</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] - \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>SLL rd, rs1, rs2</td>
<td>Shift Left Logical</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \ll \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>SLT rd, rs1, rs2</td>
<td>Set if &lt; (Signed)</td>
<td>( \text{reg}[\text{rd}] \leq (\text{reg}[\text{rs1}] &lt;_s \text{reg}[\text{rs2}]) ? 1 : 0 )</td>
</tr>
<tr>
<td>SLTU rd, rs1, rs2</td>
<td>Set if &lt; (Unsigned)</td>
<td>( \text{reg}[\text{rd}] \leq (\text{reg}[\text{rs1}] &lt;_u \text{reg}[\text{rs2}]) ? 1 : 0 )</td>
</tr>
<tr>
<td>XOR rd, rs1, rs2</td>
<td>Xor</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \oplus \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>SRL rd, rs1, rs2</td>
<td>Shift Right Logical</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \ll_u \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>SRA rd, rs1, rs2</td>
<td>Shift Right Arithmetic</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \ll_s \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>OR rd, rs1, rs2</td>
<td>Or</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \lor \text{reg}[\text{rs2}] )</td>
</tr>
<tr>
<td>AND rd, rs1, rs2</td>
<td>And</td>
<td>( \text{reg}[\text{rd}] \leq \text{reg}[\text{rs1}] \land \text{reg}[\text{rs2}] )</td>
</tr>
</tbody>
</table>

These instructions are grouped in a category called OP with fields \((\text{AluFunc}, \text{rd}, \text{rs1}, \text{rs2})\)
Register-Register ALU Datapath

Function codes:

- funct7
- rs2
- rs1
- f3
- rd

Binary: 0110011

Op type: Reg[rd] ← Reg[rs1] op Reg[rs2]

0110011 => OP type
Reg-Reg ALU

AluFunc

Inst[30]: Add/Sub
Inst[30]: Srl/Sra

March 16, 2023

MIT 6.191 Spring 2023
Register-Register ALU Datapath

OP type: Reg[rd] ← Reg[rs1] op Reg[rs2]
## ALU Instructions
with one Immediate operand

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI rd, rs1, const</td>
<td>Add Immediate</td>
<td>reg[rd] &lt;= reg[rs1] + const</td>
</tr>
<tr>
<td>SLTI rd, rs1, const</td>
<td>Set if &lt; Immediate (Signed)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt;_s const) ? 1 : 0</td>
</tr>
<tr>
<td>SLTIU rd, rs1, const</td>
<td>Set if &lt; Immediate (Unsigned)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt;_u const) ? 1 : 0</td>
</tr>
<tr>
<td>XORI rd, rs1, const</td>
<td>Xor Immediate</td>
<td>reg[rd] &lt;= reg[rs1] ^ const</td>
</tr>
<tr>
<td>ORI rd, rs1, const</td>
<td>Or Immediate</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>ANDI rd, rs1, const</td>
<td>And Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &amp; const</td>
</tr>
<tr>
<td>SLLI rd, rs1, shamdt</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &lt;&lt; shamdt</td>
</tr>
<tr>
<td>SRLI rd, rs1, shamdt</td>
<td>Shift Right Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt;_u shamdt</td>
</tr>
<tr>
<td>SRAI rd, rs1, shamdt</td>
<td>Shift Right Arithmetic Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt;_s shamdt</td>
</tr>
</tbody>
</table>

These instructions are grouped in a category called OPIMM with fields (AluFunc, rd, rs1, immI32)
Register-Immediate ALU Datapath

Op Imm type: Reg[rd] ← Reg[rs1] op SXT(imm[11:0])
Reg[rd] ← Reg[rs1] shift_op (imm[4:0])

0010011 => OpImm type
Reg-Imm ALU
Op Imm type: Reg[rd] ← Reg[rs1] op SXT(imm[11:0])
Reg[rd] ← Reg[rs1] shift_op (imm[4:0])

0010011 => OpImm type
Reg-Imm ALU

funct3, Inst[31:25] => AluFunc
Inst[30]: Srli/Srai
Register-Immediate ALU Datapath

Op Imm type: Reg[rd] ← Reg[rs1] op SXT(imm[11:0])
Reg[rd] ← Reg[rs1] shift_op (imm[4:0])
# Load and Store Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW rd, offset(rs1)</td>
<td>Load Word</td>
<td>$\text{reg[rd]} \leftarrow \text{mem[reg[rs1] + offset]}$</td>
</tr>
<tr>
<td>SW rs2, offset(rs1)</td>
<td>Store Word</td>
<td>$\text{mem[reg[rs1] + offset]} \leftarrow \text{reg[rs2]}$</td>
</tr>
</tbody>
</table>

LW and SW need to access memory for execution, so they need to compute an effective memory address.
# Load Instruction

The load instruction is represented as:

```
```

The diagram shows the flow of data and control signals in the load operation. The instruction is fetched from memory and decoded by the decoder. The decoded instruction is then processed by the register file, ALU, and data memory. The load function is executed to load the memory data into the register.

- **Opcode:** lw
- **BSEL:** 1
- **AluFunc:** Add

The diagram includes the following details:

- **PC:** 00
- **Inst[31:0]:**imm[11:0] rs1 010 rd 0000011
- **Load instruction:** 0000011 => lw
- **AluFunc = Add**
- **BSEL = 1**

The diagram also shows the flow of data through the instruction memory, decoder, register file, ALU, and data memory.
Load Instruction


March 16, 2023
MIT 6.191 Spring 2023
Store Instruction


0100011 => sw

AluFunc = Add
BSEL = 1
Store Instruction

Store: Mem[Reg[rs1] + SXT(imm[11:0])] ← Reg[rs2]
immS32
Branch Instructions
differ only in the aluBr operation they perform

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ rs1, rs2, label</td>
<td>Branch =</td>
<td>pc &lt;= (reg[rs1] == reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BNE rs1, rs2, label</td>
<td>Branch !=</td>
<td>pc &lt;= (reg[rs1] != reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td>BLT rs1, rs2, label</td>
<td>Branch &lt;</td>
<td>pc &lt;= (reg[rs1] &lt; s reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td></td>
<td>(Signed)</td>
<td></td>
</tr>
<tr>
<td>BGE rs1, rs2, label</td>
<td>Branch ≥</td>
<td>pc &lt;= (reg[rs1] ≥ s reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td></td>
<td>(Signed)</td>
<td></td>
</tr>
<tr>
<td>BLTU rs1, rs2, label</td>
<td>Branch &lt;</td>
<td>pc &lt;= (reg[rs1] &lt; u reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td></td>
<td>(Unsigned)</td>
<td></td>
</tr>
<tr>
<td>BGEU rs1, rs2, label</td>
<td>Branch ≥</td>
<td>pc &lt;= (reg[rs1] ≥ u reg[rs2]) ? label : pc + 4</td>
</tr>
<tr>
<td></td>
<td>(Unsigned)</td>
<td></td>
</tr>
</tbody>
</table>

These instructions are grouped in a category called BRANCH with fields (brFunc, rs1, rs2, immB32)
ALU for Branch Comparisons

Like ALU, but returns a Bool

\[
\text{func} - \text{GT, LT, EQ, ...}
\]

\[
\begin{align*}
\text{a} & \quad \text{b} \\
\text{ALU} & \quad \text{Br} \\
\text{branch} &
\end{align*}
\]
Branch: branch = (Reg[rs1] brFunc Reg[rs2])

immB32 = SXT({imm[12:1],1'b0})

pc <= branch ? pc + immB32 : pc + 4

1100011 => Branch type

funct3 => BrFunc
## Remaining Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL rd, label</td>
<td>Jump and Link</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc &lt;= label</td>
</tr>
<tr>
<td>JALR rd, offset(rs1)</td>
<td>Jump and Link Register</td>
<td>reg[rd] &lt;= pc + 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc &lt;= {(reg[rs1] + offset)[31:1], 1'b0}</td>
</tr>
<tr>
<td>LUI rd, luiConstant</td>
<td>Load Upper Immediate</td>
<td>reg[rd] &lt;= luiConstant &lt;&lt; 12</td>
</tr>
</tbody>
</table>

Each of these instructions is in a category by itself and needs to extract different fields from the instruction. jal and jalr update both pc and reg[rd].
### Jalr Instruction

#### Jalr: immI32 = SXT(imm[11:0])

\[
\text{Reg}[\text{rd}] \leftarrow \text{pc} + 4
\]

\[
\text{pc} \leftarrow \{(\text{Reg}[\text{rs1}] + \text{immI32})[31:1], 1'b0\}
\]

1100111 =⇒ Jalr type
Jalr Instruction

Jalr: immI32 = SXT(imm[11:0])
Reg[rd] ← pc + 4
pc ← {(Reg[rs1] + immI32)[31:1], 1'b0}
Single-Cycle RISC-V Processor

- PC
- Decode
- Execute
- Register File
- Data Memory
- Inst Memory

- 2 read & 1 write ports
- separate Instruction & Data memories

March 16, 2023
MIT 6.191 Spring 2023
Is that all there is to building a processor???

No. You’ve gotta print up all those little “RISC-V Inside” stickers.