Implementing RISC-V Processor in Hardware
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

Main Memory

- instruction
- instruction
- instruction
- data
- data
- data

op rd rs rt

rd <= op(rs, rt)

0xba5eba11

How does CPU distinguish between instructions and data?
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

```
Fetch instruction
Decode instruction
Read src operands
Execute
Write dst operand
Compute next PC
```
Approach: Incremental Featurism

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

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

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

October 24, 2019
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!
Memory Timing

- For now (lab 6), 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 6, you will see the memory module referred to as a magic memory since its not really realistic.

- Next week we will learn about various different memory models and their tradeoffs.
- In the design project, we will use realistic memories for our processor.
ALU Instructions

What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- \( \text{opcode} = 0110011 \) \( \Rightarrow \) opCode Op, R-type encoding
- \( \text{funct3} = 000 \) \( \Rightarrow \) ADD
- \( \text{funct7} = 0000000 \)
- \( \text{rd} = 00011 \) \( \Rightarrow \) x3
- \( \text{rs1} = 00010 \) \( \Rightarrow \) x2
- \( \text{rs2} = 00001 \) \( \Rightarrow \) x1

ADD x3, x2, x1
Instruction Fetch/Decode

Use a counter to FETCH the next instruction: PROGRAM COUNTER (PC)

- 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 numbers, 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>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] + \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>SUB rd, rs1, rs2</td>
<td>Sub</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] - \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>SLL rd, rs1, rs2</td>
<td>Shift Left Logical</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] &lt;&lt; \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>SLT rd, rs1, rs2</td>
<td>Set if &lt; (Signed)</td>
<td>$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] &lt;_s \text{reg}[\text{rs2}]) ? 1 : 0$</td>
</tr>
<tr>
<td>SLTU rd, rs1, rs2</td>
<td>Set if &lt; (Unsigned)</td>
<td>$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] &lt;_u \text{reg}[\text{rs2}]) ? 1 : 0$</td>
</tr>
<tr>
<td>XOR rd, rs1, rs2</td>
<td>Xor</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] ^ \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>SRL rd, rs1, rs2</td>
<td>Shift Right Logical</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] &gt;&gt;_u \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>SRA rd, rs1, rs2</td>
<td>Shift Right Arithmetic</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] &gt;&gt;_s \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>OR rd, rs1, rs2</td>
<td>Or</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \mid \text{reg}[\text{rs2}]$</td>
</tr>
<tr>
<td>AND rd, rs1, rs2</td>
<td>And</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] &amp; \text{reg}[\text{rs2}]$</td>
</tr>
</tbody>
</table>

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

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

Opcode => Itype
0110011 => Op type
Reg-Reg ALU

Inst[30]: Add/Sub
Inst[30]: Srl/Sra
Register-Register ALU Datapath

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

Control Logic

Instruction Memory

ALU

Register File

PC

Opcode: Inst[6:0]
Funct3: Inst[14:12]
Funct7: Inst[31:25]
# 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, immI</td>
<td>Add Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] + \text{immI}$</td>
</tr>
<tr>
<td>SLTI rd, rs1, immI</td>
<td>Set if $&lt;$ Immediate (Signed)</td>
<td>$\text{reg}[rd] \leftarrow (\text{reg}[rs1] &lt;_s \text{immI}) \ ? \ 1 : 0$</td>
</tr>
<tr>
<td>SLTIU rd, rs1, immI</td>
<td>Set if $&lt;$ Immediate (Unsigned)</td>
<td>$\text{reg}[rd] \leftarrow (\text{reg}[rs1] &lt;_u \text{immI}) \ ? \ 1 : 0$</td>
</tr>
<tr>
<td>XORI rd, rs1, immI</td>
<td>Xor Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] \ ^\ immI$</td>
</tr>
<tr>
<td>ORI rd, rs1, immI</td>
<td>Or Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] \</td>
</tr>
<tr>
<td>ANDI rd, rs1, immI</td>
<td>And Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] \ &amp; \ immI$</td>
</tr>
<tr>
<td>SLLI rd, rs1, immI</td>
<td>Shift Left Logical Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] \ll \text{immI}$</td>
</tr>
<tr>
<td>SRLI rd, rs1, immI</td>
<td>Shift Right Logical Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] \gg_u \text{immI}$</td>
</tr>
<tr>
<td>SRAI rd, rs1, immI</td>
<td>Shift Right Arithmetic Immediate</td>
<td>$\text{reg}[rd] \leftarrow \text{reg}[rs1] \gg_s \text{immI}$</td>
</tr>
</tbody>
</table>

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

Op Imm type: $\text{Reg}[rd] \leftarrow \text{Reg}[rs1] \text{ op } SXT(\text{imm}[11:0])$

Reg[rd] $\leftarrow \text{Reg}[rs1] \text{ shift}_\text{op} (\text{imm}[4:0])$

Op code $\Rightarrow$ Itype

$0010011 \Rightarrow \text{OpImm type Reg-Imm ALU}$

Opcode: $\text{Inst}[6:0]$

Funct3: $\text{Inst}[14:12]$

Funct7: $\text{Inst}[31:25]$

funct3, $\text{Inst}[30] \Rightarrow \text{AluFunc}$

Inst[30]: Srli/Srai
Register-Immediate ALU Datapath

Op Imm type: Reg[rd] \leftarrow Reg[rs1] \text{ op } \text{SXT}(\text{imm}[11:0])
Reg[rd] \leftarrow Reg[rs1] \text{ shift\_op } (\text{imm}[4:0])
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, immI(rs1)</td>
<td>Load Word</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{mem}[\text{reg}[\text{rs1}] + \text{immI}]$</td>
</tr>
<tr>
<td>SW rs2, immS(rs1)</td>
<td>Store Word</td>
<td>$\text{mem}[\text{reg}[\text{rs1}] + \text{immS}] \leftarrow \text{reg}[\text{rs2}]$</td>
</tr>
</tbody>
</table>

LW and SW need to access memory for execution and thus, are required to compute an effective memory address.
Load Instruction


0000011 => lw

AluFunc = Addi

October 24, 2019
Load Instruction


Instruction Memory

PC

+4

Inst[31:0]

Control Logic

Opcode: Inst[6:0]
Funct3: Inst[14:12]
Funct7: Inst[31:25]

Register File

rs1: Inst[19:15]
rs2: Inst[24:20]

SXT(Inst[31:20])

ALU

A+B

Data Memory

Sr

imm[11:0] rs1 010 rd 0000011

October 24, 2019

MIT 6.004 Fall 2019
Store Instruction

Store: Mem[Reg[rs1] + SXT(imm[11:0])] ← Reg[rs2]
immS

0100011 => sw

AluFunc = Addi
Store Instruction

Store: Mem[Reg[rs1] + SXT(imm[11:0])] ← Reg[rs2]
immS
# 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, immB</td>
<td>Branch =</td>
<td>pc &lt;= (reg[rs1] == reg[rs2]) ? pc + immB : pc + 4</td>
</tr>
<tr>
<td>BNE rs1, rs2, immB</td>
<td>Branch !=</td>
<td>pc &lt;= (reg[rs1] != reg[rs2]) ? pc + immB : pc + 4</td>
</tr>
<tr>
<td>BLT rs1, rs2, immB</td>
<td>Branch &lt; (Signed)</td>
<td>pc &lt;= (reg[rs1] &lt;s reg[rs2]) ? pc + immB : pc + 4</td>
</tr>
<tr>
<td>BGE rs1, rs2, immB</td>
<td>Branch ≥ (Signed)</td>
<td>pc &lt;= (reg[rs1] ≥s reg[rs2]) ? pc + immB : pc + 4</td>
</tr>
<tr>
<td>BLTU rs1, rs2, immB</td>
<td>Branch &lt; (Unsigned)</td>
<td>pc &lt;= (reg[rs1] &lt;u reg[rs2]) ? pc + immB : pc + 4</td>
</tr>
<tr>
<td>BGEU rs1, rs2, immB</td>
<td>Branch ≥ (Unsigned)</td>
<td>pc &lt;= (reg[rs1] ≥u reg[rs2]) ? pc + immB : pc + 4</td>
</tr>
</tbody>
</table>

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

Like ALU but returns a Bool

func
- GT, LT, EQ, ...

branch

ALU Br

a
b

gu

AluFunc
BrFunc

32

branch
Branch Instructions

**Instruction Memory**

**ALU Br**

**Control Logic**

**Register File**

**Data Memory**

Branch:  \( \text{branch} = (\text{Reg}[\text{rs1}] \text{ brFunc} \text{ Reg}[\text{rs2}]) \)

\( \text{immB} = \text{SXT}([\text{imm}[12:1],1'b0]) \)

\( \text{pc} \leq \text{branch} ? \text{pc} + \text{immB} : \text{pc} + 4 \)

1100011 => Branch type
## Remaining Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL rd, immJ</td>
<td>Jump and Link</td>
<td>reg[rd] &lt;= pc + 4&lt;br&gt;pc &lt;= pc + immJ</td>
</tr>
<tr>
<td>JALR rd, immI(rs1)</td>
<td>Jump and Link Register</td>
<td>reg[rd] &lt;= pc + 4&lt;br&gt;pc &lt;= (reg[rs1] + immI)[31:1], 1'b0</td>
</tr>
<tr>
<td>LUI rd, immU</td>
<td>Load Upper Immediate</td>
<td>reg[rd] &lt;= immU</td>
</tr>
</tbody>
</table>

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

```
Jalr: immI = SXT(imm[11:0])
Reg[rd] ← pc + 4
pc ← {(Reg[rs1] + immI)[31:1], 1'b0}
```

1100111 => Jalr type

```
imm[11:0]  rs1  000  rd  1100111
```

October 24, 2019
Jalr: \( \text{immI} = \text{SXT}(\text{imm}[11:0]) \)

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

\[ \text{pc} \leftarrow \{(\text{Reg}[\text{rs1}] + \text{immI}[31:1])\cdot1'b0\} \]
Single-Cycle RISC-V Processor

- **Register File**
- **Instruction Memory**
- **Data Memory**
- **PC**
- **Decode**
- **Execute**
- **Data**
- **Memory**
- **instr**
- **pc**
- **Decoding**
- **Execution**
- **Reg[rs1]**
- **Reg[rs2]**
- **nextPc**
- **2 read & 1 write ports**
- **separate Instruction & Data memories**
Instruction decoder

- We need a function to extract the instruction type and the various fields for each type from a 32-bit instruction
- Fields we have identified so far are:
  - Instruction type: OP, OPIMM, BRANCH, JAL, JALR, LUI, LOAD, STORE, Unsupported
  - Function for alu: aluFunc
  - Function for brAlu: brFunc
  - Register fields: rd, rs1, rs2
  - Immediate constants: immI(12), immB(12), immJ(20), immU(20), immS(12) but each is used as a 32-bit value with proper sign extension

Notice that no instruction has all the fields
Execute Function

- **Inputs:**
  - Values read from register file
  - Decoded instruction fields
  - PC

- **Logic:**
  - ALU
  - BrALU
  - NextPC generation

- **Outputs:**
  - Destination register
  - Data to write to register file or memory
  - Address for load and store operations
  - NextPC
RISC-V Inside

Is that all there is to building a processor???

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