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)
The von Neumann Model

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

```
Main Memory

instruction
instruction
instruction

Central Processing Unit

data
data
data

op rd rs rt

rd <= op(rs,rt)
```
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

![Diagram showing main memory and central processing unit with an instruction at 0xba5eba11 and an operation rd <= op(rs,rt)]
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

<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>

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>rs</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xba5eba11</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rd <= op(rs, rt)

How does CPU distinguish between instructions and data?
Anatomy of a von Neumann Computer

- **Datapath**
  - Internal storage
  - Control
  - Status
  - Address
  - Data

- **Control Unit**
  - Address
  - Instructions

- **Main Memory**
Anatomy of a von Neumann Computer

- **Datapath**
  - Internal storage
  - Address
  - Data

- **Control Unit**
  - Control
  - Status
  - Address
  - Instructions

- **Main Memory**
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

Internal storage

address

data

control

status

address

instructions

dest

asel

fn

bssel

ALU

CCs
Anatomy of a von Neumann Computer

Datapath

Control Unit

Main Memory

Registers

Datapath

Control

Unit

Internal

storage

address

control

status

data

instructions

address

dest

registers

fn

ALU

CCs

asel

b.sel

October 24, 2019
Anatomy of a von Neumann Computer
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
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
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
Approach: Incremental Featurism

We’ll implement datapaths for each instruction class individually, and merge them (using MUXes, etc)
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
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
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
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
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 (3-port)
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
Multi-Ported Register File
Multi-Ported Register File

A diagram illustrating a multi-ported register file with two read ports (RA1, RA2) and one write port. The diagram includes the control signals for enabling (EN) and clock (clk) inputs. The diagram also shows the flow of data from registers to the ports and back.
Multi-Ported Register File

2 combinational READ ports*, 1 clocked WRITE port

*internal logic ensures Reg[0] reads as 0
Multi-Ported Register File

2 combinational READ ports*, 1 clocked WRITE port

*internal logic ensures Reg[0] reads as 0
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

Read address RA
Read data RD
CLK
Write enable WE
Write address WA
Write data WD

$\text{combinational READ ports, 1 clocked WRITE port}$

$\text{Reg}[A]$ $\text{new Reg}[A]$ $\text{new Reg}[A]$ $\text{new Reg}[A]$
Register File Timing

2 combinational READ ports, 1 clocked WRITE port

- Read address (RA)
- Read data (RD)
- Write enable (WE)
- Write address (WA)
- Write data (WD)

Timing diagram showing the relationship between the registers and the control signals (CLK, WE, WA, WD) with delays indicated as $t_{PD}$, $t_s$, and $t_h$. The diagram illustrates the timing sequence for reading and writing to the register file.
Register File Timing

2 combinational READ ports, 1 clocked WRITE port

- Read address (RA)
- Read data (RD)
- Write enable (WE)
- Write address (WA)
- Write data (WD)

Timing intervals:
- \( t_{PD} \)
- \( t_s \) and \( t_h \)
Register File Timing

2 combinational READ ports, 1 clocked WRITE port

<table>
<thead>
<tr>
<th>Read address</th>
<th>RA</th>
<th>Reg[A]</th>
<th>new Reg[A]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read data</td>
<td>RD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write enable</td>
<td>WE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write address</td>
<td>WA</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Write data</td>
<td>WD</td>
<td></td>
<td>new Reg[A]</td>
</tr>
</tbody>
</table>

- $t_{PD}$
- $t_s$
- $t_h$
Register File Timing

2 combinational READ ports, 1 clocked WRITE port

What if WA=RA1?
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.
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.
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
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.
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.
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 it’s 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.
What RISC-V instruction is represented by these 32 bits?
What RISC-V instruction is represented by these 32 bits?
Reference manual specifies the fields as follows:

0000000000001000100000000110110011
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- opcode = 0110011
ALU Instructions

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

Reference manual specifies the fields as follows:

- opcode = 0110011  =>  OpCode Op, R-type encoding
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- opcode = 0110011 => opCode Op, R-type encoding
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- opcode = 0110011 => OpCode Op, R-type encoding
- funct3 = 000
- funct7 = 0000000
- rd = 00011
- rs1 = 00010
- rs2 = 00001
ALU Instructions

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

Reference manual specifies the fields as follows:

- opcode = 0110011  =>  opCode Op, R-type encoding
- funct3 = 000  =>  ADD
- funct7 = 0000000
- rd = 00011
- rs1 = 00010
- rs2 = 00001

October 24, 2019  MIT 6.004 Fall 2019
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- opcode = 0110011  => opCode Op, R-type encoding
- funct3 = 000     => ADD
- funct7 = 0000000
- rd = 00011       => x3
- rs1 = 00010
- rs2 = 00001
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- **opcode** = 0110011 => OpCode Op, R-type encoding
- **funct3** = 000 => ADD
- **funct7** = 0000000
- **rd** = 00011 => x3
- **rs1** = 00010 => x2
- **rs2** = 00001
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- \( \text{opcode} = 0110011 \) => OpCode Op, R-type encoding
- \( \text{funct3} = 000 \) => ADD
- \( \text{funct7} = 0000000 \)
- \( \text{rd} = 00011 \) => x3
- \( \text{rs1} = 00010 \) => x2
- \( \text{rs2} = 00001 \) => x1
What RISC-V instruction is represented by these 32 bits?

Reference manual specifies the fields as follows:

- opcode = 0110011 => OpCode Op, R-type encoding
- funct3 = 000 => ADD
- funct7 = 0000000
- rd = 00011 => x3
- rs1 = 00010 => x2
- rs2 = 00001 => x1

ADD x3, x2, x1
Instruction Fetch/Decode

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

Reset

PC

+4

A Instruction Memory

Inst[31:0]

32


rs1: Inst[19:15]

rs2: Inst[20:24]

INSTRUCTION WORD FIELDS

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

Control Logic

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

- Use PC as memory address
**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

---

Instruction Memory

A Instruction Memory

+4

32

Inst[31:0]

32


rs1: Inst[19:15]

rs2: Inst[20:24]

INSTRUCTION WORD FIELDS

Control Logic

Opcode: Inst[6:0]

Funct3: Inst[14:12]

Funct7: Inst[31:25]

CONTROL SIGNALS

October 24, 2019
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

**CONTROL SIGNALS**

**INSTRUCTION WORD FIELDS**

- Inst[31:0]: rd
- Inst[11:7]: rs1
- Inst[20:24]: rs2
- Inst[6:0]: Opcode
- Inst[14:12]: Func3
- Inst[31:25]: Func7

**CONTROL SIGNALS**

- Reset
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
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><code>ADD rd, rs1, rs2</code></td>
<td>Add</td>
<td><code>reg[rd] &lt;= reg[rs1] + reg[rs2]</code></td>
</tr>
<tr>
<td><code>SUB rd, rs1, rs2</code></td>
<td>Sub</td>
<td><code>reg[rd] &lt;= reg[rs1] - reg[rs2]</code></td>
</tr>
<tr>
<td><code>SLL rd, rs1, rs2</code></td>
<td>Shift Left Logical</td>
<td><code>reg[rd] &lt;= reg[rs1] &lt;&lt; reg[rs2]</code></td>
</tr>
<tr>
<td><code>SLT rd, rs1, rs2</code></td>
<td>Set if &lt; (Signed)</td>
<td><code>reg[rd] &lt;= (reg[rs1] &lt;= _s reg[rs2]) ? 1 : 0</code></td>
</tr>
<tr>
<td><code>SLTU rd, rs1, rs2</code></td>
<td>Set if &lt; (Unsigned)</td>
<td><code>reg[rd] &lt;= (reg[rs1] &lt; _u reg[rs2]) ? 1 : 0</code></td>
</tr>
<tr>
<td><code>XOR rd, rs1, rs2</code></td>
<td>Xor</td>
<td><code>reg[rd] &lt;= reg[rs1] ^ reg[rs2]</code></td>
</tr>
<tr>
<td><code>SRL rd, rs1, rs2</code></td>
<td>Shift Right Logical</td>
<td><code>reg[rd] &lt;= reg[rs1] &gt;&gt; _u reg[rs2]</code></td>
</tr>
<tr>
<td><code>SRA rd, rs1, rs2</code></td>
<td>Shift Right Arithmetic</td>
<td><code>reg[rd] &lt;= reg[rs1] &gt;&gt; _s reg[rs2]</code></td>
</tr>
<tr>
<td><code>OR rd, rs1, rs2</code></td>
<td>Or</td>
<td>`reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td><code>AND rd, rs1, rs2</code></td>
<td>And</td>
<td><code>reg[rd] &lt;= reg[rs1] &amp; reg[rs2]</code></td>
</tr>
</tbody>
</table>
# 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>reg[rd] &lt;= reg[rs1] + reg[rs2]</td>
</tr>
<tr>
<td>SUB rd, rs1, rs2</td>
<td>Sub</td>
<td>reg[rd] &lt;= reg[rs1] - reg[rs2]</td>
</tr>
<tr>
<td>SLL rd, rs1, rs2</td>
<td>Shift Left Logical</td>
<td>reg[rd] &lt;= reg[rs1] &lt;&lt; reg[rs2]</td>
</tr>
<tr>
<td>SLT rd, rs1, rs2</td>
<td>Set if &lt; (Signed)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt;s reg[rs2]) ? 1 : 0</td>
</tr>
<tr>
<td>SLTU rd, rs1, rs2</td>
<td>Set if &lt; (Unsigned)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt;u reg[rs2]) ? 1 : 0</td>
</tr>
<tr>
<td>XOR rd, rs1, rs2</td>
<td>Xor</td>
<td>reg[rd] &lt;= reg[rs1] ^ reg[rs2]</td>
</tr>
<tr>
<td>SRL rd, rs1, rs2</td>
<td>Shift Right Logical</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt;u reg[rs2]</td>
</tr>
<tr>
<td>SRA rd, rs1, rs2</td>
<td>Shift Right Arithmetic</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt;s reg[rs2]</td>
</tr>
<tr>
<td>OR rd, rs1, rs2</td>
<td>Or</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>AND rd, rs1, rs2</td>
<td>And</td>
<td>reg[rd] &lt;= reg[rs1] &amp; reg[rs2]</td>
</tr>
</tbody>
</table>

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

```
+4
PC 00

Instruction Memory

Inst[31:0]


Register File

RA1 WA RD1
RA2 WD RD2

Control Logic

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

funct7  rs2  rs1  f3  rd  0110000
```
Register-Register ALU Datapath

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

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

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

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

Control Logic

AluFunc

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

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

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

funct7 rs2 rs1 f3 rd 0110000
Register-Register ALU Datapath

Opcode: \[\text{Inst}^{[31:0]}\]
- Opcode: \[\text{Inst}^{[6:0]}\]
- Funct7: \[\text{Inst}^{[31:25]}\]
- Funct3: \[\text{Inst}^{[14:12]}\]

Function Values:
- \(\text{Funct3} = 0110000\): Add/Sub
- \(\text{Funct3} = 0110001\): Srl/Sra

Control Logic

Instruction Memory

Opcode: \(\text{Inst}^{[31:0]}\)

Register File

- \(\text{rd} = \text{Inst}^{[11:7]}\)
- \(\text{rs1} = \text{Inst}^{[19:15]}\)
- \(\text{rs2} = \text{Inst}^{[24:20]}\)

ALU

Register ALU Datapath

Function: \(\text{Inst}^{[11:7]}\)

Operand: \(\text{Inst}^{[6:0]}\)

Op type: \(\text{Reg}^{[\text{rd}]} \leftarrow \text{Reg}^{[\text{rs1}]} \text{ op } \text{Reg}^{[\text{rs2}]}\)

Opcode: \(\text{Inst}^{[31:0]}\)

Op type: \(0110011\) => Op type
Reg-Reg ALU

funct3, \(\text{Inst}^{[30]}\) => AluFunc

\(\text{Inst}^{[30]}\): Add/Sub
\(\text{Inst}^{[30]}\): Srl/Sra
Register-Register ALU Datapath

Op type: Reg[rd] ← Reg[rs1] op Reg[rs2]
Register-Register ALU Datapath

Op type: Reg[rd] ← Reg[rs1] op Reg[rs2]
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, immI</td>
<td>Add Immediate</td>
<td>reg[rd] &lt;= reg[rs1] + immI</td>
</tr>
<tr>
<td>SLTI rd, rs1, immI</td>
<td>Set if &lt; Immediate (Signed)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt;_s immI) ? 1 : 0</td>
</tr>
<tr>
<td>SLTIU rd, rs1, immI</td>
<td>Set if &lt; Immediate (Unsigned)</td>
<td>reg[rd] &lt;= (reg[rs1] &lt;_u immI) ? 1 : 0</td>
</tr>
<tr>
<td>XORI rd, rs1, immI</td>
<td>Xor Immediate</td>
<td>reg[rd] &lt;= reg[rs1] ^ immI</td>
</tr>
<tr>
<td>ORI rd, rs1, immI</td>
<td>Or Immediate</td>
<td>reg[rd] &lt;= reg[rs1]</td>
</tr>
<tr>
<td>ANDI rd, rs1, immI</td>
<td>And Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &amp; immI</td>
</tr>
<tr>
<td>SLLI rd, rs1, immI</td>
<td>Shift Left Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &lt;&lt; immI</td>
</tr>
<tr>
<td>SRLI rd, rs1, immI</td>
<td>Shift Right Logical Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt;_u immI</td>
</tr>
<tr>
<td>SRAI rd, rs1, immI</td>
<td>Shift Right Arithmetic Immediate</td>
<td>reg[rd] &lt;= reg[rs1] &gt;&gt;_s immI</td>
</tr>
</tbody>
</table>
## 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>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] + \text{immI}$</td>
</tr>
<tr>
<td>SLTI rd, rs1, immI</td>
<td>Set if $&lt;$ Immediate (Signed)</td>
<td>$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] &lt;_s \text{immI}) \ ? 1 : 0$</td>
</tr>
<tr>
<td>SLTIU rd, rs1, immI</td>
<td>Set if $&lt;$ Immediate (Unsigned)</td>
<td>$\text{reg}[\text{rd}] \leftarrow (\text{reg}[\text{rs1}] &lt;_u \text{immI}) \ ? 1 : 0$</td>
</tr>
<tr>
<td>XORI rd, rs1, immI</td>
<td>Xor Immediate</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \ ^{\text{}} \text{immI}$</td>
</tr>
<tr>
<td>ORI rd, rs1, immI</td>
<td>Or Immediate</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \</td>
</tr>
<tr>
<td>ANDI rd, rs1, immI</td>
<td>And Immediate</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \ &amp; \text{immI}$</td>
</tr>
<tr>
<td>SLLI rd, rs1, immI</td>
<td>Shift Left Logical Immediate</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \ll \text{immI}$</td>
</tr>
<tr>
<td>SRLI rd, rs1, immI</td>
<td>Shift Right Logical Immediate</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{rs1}] \gg_u \text{immI}$</td>
</tr>
<tr>
<td>SRAI rd, rs1, immI</td>
<td>Shift Right Arithmetic Immediate</td>
<td>$\text{reg}[\text{rd}] \leftarrow \text{reg}[\text{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

Instruction Memory

Inst[31:0]

PC

+4

Register File

ra1

ra2

wa

rd1

rd2

Control Logic

AluFunc

WERF

ALU

A

B

AluFunc

WERF

Inst[11:0]

rs1

f3

rd

0010000

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

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

00
Register-Immediate ALU Datapath

**Instruction Memory**

```
+4
```

Inst[31:0]

**Register File**

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

**Control Logic**

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

**ALU**

```
AluFunc
WERF
```

** Opcode => Itype**

```
0010000
```

```
0010011 => OpImm type
```

```
Reg-Imm ALU
```
Register-Immediate ALU Datapath

Op Imm type: \( \text{Reg}[rd] \leftarrow \text{Reg}[rs1] \text{ op SXT}(\text{imm}[11:0]) \)
\( \text{Reg}[rd] \leftarrow \text{Reg}[rs1] \text{ shift}_\text{op} (\text{imm}[4:0]) \)

 Opcode \( \Rightarrow \) Itype
0010011 \( \Rightarrow \) OpImm type
Reg-Imm ALU
Register-Immediate ALU Datapath

Op Imm type: \(\text{Reg}[rd] \leftarrow \text{Reg}[rs1] \text{ op } \text{SXT}([\text{imm}[11:0]])\)
\(\text{Reg}[rd] \leftarrow \text{Reg}[rs1] \text{ shift_op } ([\text{imm}[4:0]])\)

 Opcode \(\Rightarrow\) Itype
\(0010011 \Rightarrow \text{OpImm type Reg-Imm ALU}\)

\(\text{funct3, Inst}[30] \Rightarrow\) AluFunc
\(\text{Inst}[30]: \text{Srli/Srai}\)
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])

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

Instruction Memory

PC 00

Instruction

imm[11:0] rs1 f3 rd 0010000

Control Logic

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

ALU

BSEL
AluFunc
WERF

Register File

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

SXT(Inst[31:20])

BSEL

Shift operation (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])
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>reg[rd] &lt;= mem[reg[rs1] + immI]</td>
</tr>
<tr>
<td>SW rs2, immS(rs1)</td>
<td>Store Word</td>
<td>mem[reg[rs1] + immS] &lt;= reg[rs2]</td>
</tr>
</tbody>
</table>
# 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>reg[rd] &lt;= mem[reg[rs1] + immI]</td>
</tr>
<tr>
<td>SW rs2, immS(rs1)</td>
<td>Store Word</td>
<td>mem[reg[rs1] + immS] &lt;= reg[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

PC 00

Inst[31:0]

imm[11:0] rs1 010 rd 0000011

Control Logic

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

SXT(Inst[31:20])

ALU

BSEL WDSEL AluFunc WERF MWR

Register File


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

Load Instruction Memory

A

Inst[31:0]

Instruction Memory

A

Inst[31:0]

00

+4

RD1 WA

RD2 WE

RA1

RA2

WD

WERF

32 32

1 0

BSEL

ALU

A

B

32

October 24, 2019

MIT 6.004 Fall 2019

L14-84
Load Instruction

![Diagram of instruction flow]

- **PC**: 00
- **Instruction Memory**: Inst[31:0]
- **PC + 4**: Inst[31:0]
- **Instruction Memory Output**:
  - Inst[31:0] to **Register File**
  - rs1: Inst[19:15]
  - rs2: Inst[24:20]
- **Register File**:
  - RA1, RA2, Rd1, Rd2
- **Control Logic**:
  - Opcode: Inst[6:0]
  - Funct3: Inst[14:12]
  - Funct7: Inst[31:25]
- **ALU**:
  - A, B, AluFunc
  - SXT(Inst[31:20])
- **Memory**:
  - WA, WD, WE
  - imm[11:0] => lw

**Example Instruction**:

- **Inst**: 00000111010000011
- **Decoded**: lw
- **Control Signals**:
  - BSEL, WDSEL, AluFunc, WERF, MWR
Load Instruction


0000011 => lw
Load Instruction


0000011 => lw

AluFunc = Addi
Load Instruction

Load: Reg[rd] \leftarrow Mem[Reg[rs1] + SXT(imm[11:0])]

0000011 => lw

AluFunc = Addi
Load Instruction


- Opcode: Inst[6:0]
- Funct3: Inst[14:12]
- Funct7: Inst[31:25]
- imm[11:0] = 010
- rs1 = 010
- rd = 0000011

Diagram:
- Instruction Memory
- Register File
- Control Logic
- ALU
- Data Memory
- PC +4
Load Instruction


Control Logic:
- BSEL
- WDSEL
- AluFunc
- WERF
- MWR

Register File:
- RA1
- RA2
- WA
- RD1
- WD
- RD2
- WE
- WERF

ALU:
- A
- B
- A+B

Data Memory:
- Addr
- RD
- WD
- R/W
- MWR

MWR → Reg[rd]

imm[11:0] rs1 010 rd 0000011
Load Instruction


Opstasy: Inst[6:0]
Funct3: Inst[14:12]
Funct7: Inst[31:25]
Load Instruction


Load Instruction


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

Control Logic

Control Signals:
- BSEL
- WDSEL
- AluFunc
- WERF
- MWR

ALU

Registers:
- RA1, RA2, RD1, RD2
- WA, WD, WE

Data Memory

Instruction Memory

PC: 00

Imm[11:0]
Rs1: 010
Rd: 10000011
Store Instruction

- **PC**: 00
- **Inst[31:0]**
  - **rs1**: Inst[19:15]
  - **rs2**: Inst[24:20]
  - **rd**: Inst[11:7]
  - **imm[11:5]**
  - Opcode: Inst[6:0]
  - **Funct3**: Inst[14:12]
  - **Funct7**: Inst[31:25]
- **Instruction Memory**
- **Register File**
  - RA1
  - RA2
  - WA
  - WD
  - RD1
  - RD2
  - WE
  - WERF
- **Control Logic**
  - BSEL
  - WDSEL
  - AluFunc
  - WERF
  - MWR
- **ALU**
  - A
  - B
  - SXT(Inst[31:20])
- **Data Memory**
  - WD
  - R/W
  - MWR
  - Addr
  - RD

**Example Instruction**: 0100011

- **imm[4:0]**: 010
- **imm[11:5]**: 00
- **rs2**: 010
- **rs1**: 010
- **rd**: 31
- **Inst[31:0]**: 20
- **Inst[25:20]**: 15
- **Inst[19:15]**: 11
- **Inst[14:12]**: 10
- **Inst[6:0]**: 00
- **Inst[11:5]**: 01
- **Inst[4:0]**: 11

**Diagram Components**:
- **Memory Access**: +4
- **Instruction Execution**: AluFunc
- **Register Loading**: WDSEL
- **Data Transfer**: MWR
- **Control Signals**: BSEL, WDSEL, AluFunc, WERF, MWR

**Flow Diagram**
- Instruction Fetch
- Instruction Decode
- Register Read
- ALU Operation
- Memory Access
- Result Write
Store Instruction

0100011 => sw
Store Instruction

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

0100011 => sw
Store Instruction


0100011 => sw

AluFunc = Addi
Store Instruction

Store: Mem[Reg[rs1] + SXT(imm[11:0])] ← Reg[rs2]
immS
0100011 => sw

AluFunc = Addi

October 24, 2019  MIT 6.004 Fall 2019
Store Instruction

Store: \( \text{Mem}[\text{Reg}[rs1] + \text{SXT}(\text{imm}[11:0])] \leftarrow \text{Reg}[rs2] \)

\( \text{immS} \)

\( 0100011 \Rightarrow \text{sw} \)

AluFunc = Addi
Store Instruction

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

immS
Store Instruction

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

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

alu func

A + B

SXT(Inst[31:20])
SXT({Inst[31:25], Inst[11:7]})

register file

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

control logic

BSEL
WDSEL
Alufunc
WERF
MWR

data memory

AD
MWR

PC

+4

imm[11:5]
rs2
rs1
010
imm[4:0]
0100011
Store Instruction

Store: $\text{Mem[Reg[rs1] + SXT(imm[11:0])]} \leftarrow \text{Reg[rs2]}$

$\text{immS}$
Store Instruction

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

immS
Store Instruction

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

PC  00
+4

Control Logic

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

BSEL
WDSEL
AluFunc
WERF
MWR

ALU

A + B

Data Memory

Addr
RD

WD
R/W
MWR

0
32

Register File

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

RA1
WA
> RD1

RA2
WD
> RD2

WDSEL

0    1    2

32

SXT(Inst[31:20])
SXT({Inst[31:25], Inst[11:7]})

32

32

0100011

imm[11:5]  rs2  rs1  010

imm[4:0]
## 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>
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, ...
```

```
ALU Br
a   b
```

branch
ALU for Branch Comparisons

Like ALU but returns a Bool
Branch Instructions

Instruction Memory

PC 00

Inst[31:0]

inst[31:0] rs2 rs1 f3 i[4:1|11] 1100011

Control Logic

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

Register File

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

Control Logic

ALU

BrFunc

Data Memory

Addr RD

32 32 32

0 1 0

Branch Instructions
Branch Instructions

\[
\begin{array}{c|c|c|c|c|c|c}
\text{imm}[12:10:5] & \text{rs2} & \text{rs1} & f3 & \text{i}[4:1:11] & 1100011 \\
\end{array}
\]

1100011 => Branch type

Diagram: Branch Instructions flowchart with register file and control logic.
Branch Instructions

Branch: branch = (Reg[rs1] brFunc Reg[rs2])
immB = SXT({imm[12:1],1'b0})
pc <= branch ? pc + immB : pc + 4

1100011 => Branch type
Branch Instructions

Branching from memory:

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

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

\[
\text{pc} <= \text{branch} ? \text{pc} + \text{immB} : \text{pc} + 4
\]

1100011 => Branch type
Branch Instructions

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

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

\[\text{pc} \leq \text{branch} \land \text{pc} + \text{immB} : \text{pc} + 4\]

\[\text{1100011} \Rightarrow \text{Branch type}\]
# 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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pc &lt;= {(reg[rs1] + immI)[31:1], 1\text{b}0}</td>
</tr>
<tr>
<td>LUI rd, immU</td>
<td>Load Upper Immediate</td>
<td>reg[rd] &lt;= immU</td>
</tr>
</tbody>
</table>
## 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.
## Remaining Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Execution</th>
</tr>
</thead>
</table>
| JAL rd, immJ     | Jump and Link                | \[reg[rd] \leq pc + 4 \]
                    |                              | \[pc \leq pc + immJ\]                                                   |
| JALR rd, immI(rs1) | Jump and Link Register      | \[reg[rd] \leq pc + 4 \]
                    |                              | \[pc \leq \{(reg[rs1] + immI)[31:1], 1’b0\}\]                          |
| LUI rd, immU     | Load Upper Immediate         | \[reg[rd] \leq immU\]                                                   |

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 Instruction

1100111 => Jalr type
Jalr Instruction

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

1100111 => Jalr type
Jalr Instruction

Jalr: immI = SXT(imm[11:0])

Reg[rd] ← pc + 4

pc ← {(Reg[rs1] + immI)[31:1], 1'b0}

1100111 ⇒ Jalr type
Jalr Instruction

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

Reg[rd] ← pc + 4
pc ← \{(Reg[rs1] + immI[31:1], 1'b0\}

October 24, 2019
Jalr Instruction

Jalr: immI = SXT(imm[11:0])

Reg[rd] ← pc + 4
pc ← {(Reg[rs1] + immI)[31:1], 1'b0}
Jalr Instruction

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

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

\[
\text{pc} \leftarrow \{(\text{Reg[rs1]} + \text{immI}[31:1])\cdot\text{1'}\}
\]
Jalr Instruction

Jalr: immI = SXT(imm[11:0])

Reg[rd] ← pc + 4

pc ← {(Reg[rs1] + immI)[31:1], 1'b0}
Single-Cycle RISC-V Processor
Single-Cycle RISC-V Processor

- Register File
- Instruction Memory
- Data Memory
- PC

- 2 read & 1 write ports
- Separate Instruction & Data memories
Single-Cycle RISC-V Processor

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

2 read & 1 write ports

Separate Instruction & Data memories
Single-Cycle RISC-V Processor

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

- 2 read & 1 write ports

Separate Instruction & Data memories
Single-Cycle RISC-V Processor

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

- 2 read & 1 write ports
- separate Instruction & Data memories
Single-Cycle RISC-V Processor

2 read & 1 write ports

Separate Instruction & Data memories
Single-Cycle RISC-V Processor

- Instruction Memory
- Decode
- Register File
- Execute
- Data Memory

- PC
- Inst Memory
- Decode
- Execute
- Data Memory

2 read & 1 write ports

separate Instruction & Data memories
Single-Cycle RISC-V Processor

2 read & 1 write ports

Separate Instruction & Data memories
Single-Cycle RISC-V Processor

- Instruction & Data memories are separate.
- Register File has 2 read & 1 write ports.
- PC, Decode, Execute, Inst Memory, Data Memory are interconnected.
- The diagram shows the flow of data and control signals between the components.
Single-Cycle RISC-V Processor

- **PC**
- **Instruction Memory**
- **Decode**
- **Register File**
- **Execute**
- **Data Memory**

- **2 read & 1 write ports**

Separate Instruction & Data memories
Single-Cycle RISC-V Processor

- Instruction and Data memories are separate.
- The Register File has 2 read ports and 1 write port.
- The diagram shows the flow of data between the PC, Instruction Memory, Decode, Execute, Register File, and Data Memory.
Single-Cycle RISC-V Processor

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

- 2 read & 1 write ports
- separate Instruction & Data memories
Single-Cycle RISC-V Processor

2 read & 1 write ports

Separate Instruction & Data memories
Single-Cycle RISC-V Processor

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

Diagram:
- PC
- Inst Memory
- Decode
- Execute
- Register File
- Data Memory
- Memory[addr]
- Decode_inst
- Reg[rs1], Reg[rs2]
- pc
- instr
- nextPc
- wr_data
Instruction decoder

- We need a function to extract the instruction type and the various fields for each type from a 32-bit instruction.
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
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
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
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
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
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
Execute Function

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

- **Logic:**
  - ALU
  - BrALU
  - NextPC generation
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
Is that all there is to building a processor???

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