Great Ideas in Computer Architecture (a.k.a. Machine Structures)

RISC-V Processor Design
Machine Structures

Software

Hardware

Application (ex: browser)

Operating System (Mac OSX)

Compiler

Assembler

Processor

Memory

I/O system

Datapath & Control

Digital Design

Circuit Design

Transistors

Fabrication

Instruction Set Architecture

CS61C

Garcia, Nikolić

RISC-V (2)

Berkeley
New-School Machine Structures

**Software**

- **Parallel Requests**
  - Assigned to computer
  - e.g., Search “Cats”

- **Parallel Threads**
  - Assigned to core e.g., Lookup, Ads

- **Parallel Instructions**
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions

- **Parallel Data**
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words

**Hardware descriptions**

- All gates work in parallel at same time

**Harness Parallelism & Achieve High Performance**

**Hardware**

- **Smart Phone**
- **Warehouse Scale Computer**

**Computer**

- **Core**
- **Memory** (Cache)
- **Input/Output**

**Logic Gates**

- **Main Memory**
- **Exec. Unit(s)**
- **Functional Block(s)**

```
A0 + B0 = AB
A1 + B1 = CD
```

**Harness Parallelism & Achieve High Performance**
Great Idea #1: Abstraction
(Levels of Representation/Interpretation)

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program (e.g., RISC-V)</td>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program (RISC-V)</td>
<td></td>
</tr>
</tbody>
</table>

- Compiler
  - temp = v[k];
  - v[k] = v[k+1];
  - v[k+1] = temp;

- Assembly
  - lw x3, 0(x10)
  - lw x4, 4(x10)
  - sw x4, 0(x10)
  - sw x3, 4(x10)

- Hardware Architecture Description (e.g., block diagrams)
- Logic Circuit Description (Circuit Schematic Diagrams)
Our Single-Core Processor So Far...

Processor
- Control
- Datapath
  - Program Counter (PC)
  - Registers
  - Arithmetic-Logic Unit (ALU)

Memory
- Program
- Data

Enable?
Read/Write
Address
Write Data
Read Data

Input
Output
The CPU

- **Processor (CPU):** the active part of the computer that does all the work (data manipulation and decision-making)

- **Datapath:** portion of the processor that contains hardware necessary to perform operations required by the processor (the brawn)

- **Control:** portion of the processor (also in hardware) that tells the datapath what needs to be done (the brain)
# Need to Implement All RV32I Instructions

## OpenRISC RISC-V Reference Card

### Base Integer Instructions: RV32I

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shifts</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>Shifts</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Left Logical</td>
<td>R SLL rd,rs1,rs2</td>
<td></td>
<td>Load</td>
<td>Loads</td>
<td>I LB rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Left Log. Imm.</td>
<td>I SLLI rd,rs1,shamt</td>
<td></td>
<td>Load Halfword</td>
<td>I LH rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td>R SRL rd,rs1,rs2</td>
<td></td>
<td>Load Byte Unsigned</td>
<td>I LBU rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right Log. Imm.</td>
<td>I SRLI rd,rs1,shamt</td>
<td></td>
<td>Load Half Unsigned</td>
<td>I LHU rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>R SRA rd,rs1,rs2</td>
<td></td>
<td>Load Word</td>
<td>I LW rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right Arith. Imm.</td>
<td>I SRAI rd,rs1,shamt</td>
<td></td>
<td>Store Byte</td>
<td>S SB rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic</strong></td>
<td>ADD</td>
<td>R</td>
<td>ADD rd,rs1,rs2</td>
<td>Store Halfword</td>
<td>S SH rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD Immediate</td>
<td>I ADDI rd,rs1,imm</td>
<td></td>
<td>Store Word</td>
<td>S SW rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBtract</td>
<td>R SUB rd,rs1,rs2</td>
<td></td>
<td>Branch</td>
<td>B BEQ rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Upper Imm</td>
<td>U LUI rd,imm</td>
<td></td>
<td>Branch =</td>
<td>B BNE rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Upper Imm to PC</td>
<td>U AUIPC rd,imm</td>
<td></td>
<td>Branch &lt;</td>
<td>B BLT rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>XOR</td>
<td>R</td>
<td>OR rd,rs1,rs2</td>
<td>Branch ≥</td>
<td>B BGE rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR Immediate</td>
<td>I XORI rd,rs1,imm</td>
<td></td>
<td>Branch &lt; Unaligned</td>
<td>B BLTU rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>R ORI rd,rs1,imm</td>
<td></td>
<td>Branch ≥ Unaligned</td>
<td>B BGEU rs1,rs2,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR Immediate</td>
<td>I ORI rd,rs1,imm</td>
<td></td>
<td>Jump &amp; Link</td>
<td>J JAL rd,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND Immediate</td>
<td>I ANDI rd,rs1,imm</td>
<td></td>
<td>Jump &amp; Link Register</td>
<td>I JALR rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>Set &lt;</td>
<td>R</td>
<td>SLT rd,rs1,rs2</td>
<td>Synch Synch thread</td>
<td>I FENCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set &lt; Immediate</td>
<td>I SLTI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set &lt; Unsigned</td>
<td>R SLTU rd,rs1,rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set &lt; Imm Unsigned</td>
<td>I SLTIU rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Environment**
- CALL
- ECALL
- BREAK
- EBREAK

*Not in 61C*
Building a RISC-V Processor
On every tick of the clock, the computer executes one instruction.

Current state outputs drive the inputs to the combinational logic, whose outputs settle at the values of the state before the next clock edge.

At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle.
Problem: a single, “monolithic” block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient

Solution: break up the process of “executing an instruction” into stages, and then connect the stages to create the whole datapath
  – smaller stages are easier to design
  – easy to optimize (change) one stage without touching the others (modularity)
Five Stages of the Datapath

- Stage 1: *Instruction Fetch (IF)*
- Stage 2: *Instruction Decode (ID)*
- Stage 3: *Execute (EX) - ALU (Arithmetic-Logic Unit)*
- Stage 4: *Memory Access (MEM)*
- Stage 5: *Write Back to Register (WB)*
Basic Phases of Instruction Execution

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory Access
5. Register Write

PC, IMEM, Reg[1], ALU, DMEM, clock, time
Datapath Components: Combinational

- Combinational elements

- Storage elements + clocking methodology

- Building blocks
- **Register**

- **Write Enable:**
  - Low (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on positive edge of clock
Register file (regfile, RF) consists of 32 registers:
- Two 32-bit output busses: busA and busB
- One 32-bit input bus: busW

Register is selected by:
- RA (number) selects the register to put on busA (data)
- RB (number) selects the register to put on busB (data)
- RW (number) selects the register to be written via busW (data) when Write Enable is 1

Clock input (Clk)
- Clk input is a factor ONLY during write operation
- During read operation, behaves as a combinational logic block:
  - RA or RB valid ⇒ busA or busB valid after “access time.”
- "Magic" Memory
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - For Read: Address selects the word to put on Data Out
  - For Write: Set Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid $\Rightarrow$ Data Out valid after "access time"
State Required by RV32I ISA (1/2)

Each instruction during execution reads and updates the state of: (1) Registers, (2) Program counter, (3) Memory

- **Registers (x0..x31)**
  - Register file (*regfile*) Reg holds 32 registers x 32 bits/registry: Reg[0]..Reg[31]
  - First register read specified by *rs1* field in instruction
  - Second register read specified by *rs2* field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - x0 is always 0 (writes to Reg[0] are ignored)

- **Program Counter (PC)**
  - Holds address of current instruction
State Required by RV32I ISA (2/2)

- Memory (MEM)
  - Holds both instructions & data, in one 32-bit byte-addressed memory space
  - We’ll use separate memories for instructions (IMEM) and data (DMEM)
    - These are placeholders for instruction and data caches
  - Instructions are read (fetched) from instruction memory (assume IMEM read-only)
  - Load/store instructions access data memory
Review: R-Type Instructions

### R-format: ALU

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 7 | 5 | 5 | 3 | 5 | 7 |

<table>
<thead>
<tr>
<th>func7</th>
<th>rs2</th>
<th>rs1</th>
<th>func3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>ADD</td>
<td>rd</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>SUB</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>SLL</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>SLT</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>SLTU</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>XOR</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>SRL</td>
<td>rd</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>SRA</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>OR</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>AND</td>
<td>rd</td>
</tr>
</tbody>
</table>

- E.g. Addition/subtraction
  
  **add** rd, rs1, rs2
  
  \[ R[rd] = R[rs1] + R[rs2] \]

  **sub** rd, rs1, rs2
  
  \[ R[rd] = R[rs1] - R[rs2] \]
Implementing the `add` instruction

### Instructions

- Instruction makes two changes to machine’s state:
  - `Reg[rd] = Reg[rs1] + Reg[rs2]`
  - `PC = PC + 4`

### Table

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

### Binary Codes

- `add` Code: `0000000
- `rs2` Code: `000110011`
Datapath for `add`

Reg[rd] = Reg[rs1] + Reg[rs2]

Control logic:
RegWriteEnable (RegWEn) = 1

```
funct7  rs2  rs1  funct3  rd  opcode
31  25  24  20  19  15  14  12  11  7  6  5  4  3  2  1  0
```
### Implementing the `sub` instruction

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0110011</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sub</td>
</tr>
</tbody>
</table>

#### `sub rd, rs1, rs2`

- Almost the same as `add`, except now have to subtract operands instead of adding them.

---

Garcia, Nikolić
Datapath for add/sub

PC = PC + 4

Reg[rd] = Reg[rs1] +/- Reg[rs2]

Control logic

RegWriteEnable (RegWEn) = 1

alu

Garcia, Nikolić
## Implementing Other R-Format Instructions

<table>
<thead>
<tr>
<th>Code</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
</tr>
</tbody>
</table>

All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function.
Datapath With Immediates
Implementing I-Format - \texttt{addi} instruction

- RISC-V Assembly Instruction:
  \begin{verbatim}
  addi x15, x1, -50
  \end{verbatim}

- Binary Encoding:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{imm[11:0]}</td>
<td>111111001110</td>
</tr>
<tr>
<td>\texttt{rs1}</td>
<td>00001</td>
</tr>
<tr>
<td>\texttt{funct3}</td>
<td>000</td>
</tr>
<tr>
<td>\texttt{rd}</td>
<td>01111</td>
</tr>
<tr>
<td>\texttt{opcode}</td>
<td>0010011</td>
</tr>
</tbody>
</table>

- \texttt{imm} = -50
- \texttt{rs1} = 1
- \texttt{funct3} = \texttt{add}
- \texttt{rd} = 15
- \texttt{opcode} = \texttt{OP-Imm}
Datapath for add/sub

\[ \text{PC} = \text{PC} + 4 \]

\[ \text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{Imm} \]

Control logic

\[ \text{RegWriteEnable (RegWEn)} = 1 \]

\[ \text{ALUSel (add=0/sub=1)} \]

Immediate should be here
Adding addi to Datapath

PC = PC + 4

Reg[rd] = Reg[rs1] + Imm

Control logic

RegWriteEnable (RegWEn) = 1

BSel (rs2=0/Imm=1)

ALUSEl (add=0/sub=1)

imm[11:0]  rs1  000  rd  0010011

RISC-V (31)
Adding `addi` to Datapath

\[
PC = PC + 4
\]

\[
Reg[rd] = Reg[rs1] + \text{Imm}
\]

**Control logic**

- **RegWriteEnable (RegWEn) = 1**
- **BSel (rs2 = 0/Imm = 1)**
- **ALUSel (add = 0/sub = 1)**
Adding \texttt{addi} to Datapath

- \texttt{PC = PC + 4}
- \texttt{Reg[rd] = Reg[rs1] + Imm}

Control logic:
- \texttt{ImmSel = 1}
- \texttt{RegWriteEnable (RegWEn) = 1}
- \texttt{BSel (rs2=0/Imm=1)}
- \texttt{ALUSel (add=0/sub=1)}

IPC architecture:
- \texttt{IMEM}
- \texttt{PC}
- \texttt{Inst}
- \texttt{AddrA, AddrB}
- \texttt{DataA, DataB, DataD}
- \texttt{Add}
- \texttt{alu}
- \texttt{Reg [rs1], Reg [rs2], Reg[rd]}

Logic gates:
- \texttt{RegWriteEnable (RegWEn) = 1}
- \texttt{BSel (rs2=0/Imm=1)}
- \texttt{ALUSel (add=0/sub=1)}
I-Format Immediates

• High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])

• Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])
Adding \texttt{addi} to Datapath

Works for all other I-format arithmetic instructions (\texttt{slti}, \texttt{sltiu}, \texttt{andi}, \texttt{ori}, \texttt{xori}, \texttt{slli}, \texttt{srl}, \texttt{srai}) just by changing ALUSel
Supporting Loads
R+I Arithmetic/Logic Datapath

Control logic

RISC-V (37)
### RISC-V Assembly Instruction (I-type): \texttt{lw x14, 8(x2)}

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001110</td>
<td>00010</td>
<td>010</td>
<td>01110</td>
<td>0000011</td>
</tr>
</tbody>
</table>

- The 12-bit signed immediate is added to the base address in register \texttt{rs1} to form the memory address.
  - This is very similar to the add-immediate operation but used to create address not to create final result.
- The value loaded from memory is stored in register \texttt{rd}.
R+I Arithmetic/Logic Datapath

control logic

Garcia, Nikolić

RISC-V (39)
R+I Arithmetic/Logic Datapath

Control logic:
- ImmSel = I
- RegWEn = 1
- BSel = 1
- ALUSel = Add
- MemRW = Read
- WBSel = 0

Notes:
- +4 Add
- pc+4
- IMEM
- Inst[24:20]
- AddrB DataA
- AddrD DataD
- Inst[11:7]
- Inst[19:15]
- Inst[24:20]
- AddrA
- DataA
- Reg[rs1]
- DataB
- Reg[rs2]
- Imm[31:0]
- Imm. Gen
- clk
- PC
- addr
- inst
- IMEM
- DataR
- DMEM
- alu
- AddrR
- clk
- Reg[ ]
- Inst[31:0]
- Inst[31:20]
- Inst[11:7]
- Inst[19:15]
- Inst[24:20]
Supporting the narrower loads requires additional logic to extract the correct byte/halfword from the value loaded from memory, and sign- or zero-extend the result to 32 bits before writing back to register file.

- It is just a mux + a few gates
Datapath for Stores
Adding **sw** instruction

- **sw**: Reads two registers, `rs1` for base memory address, and `rs2` for data to be stored, as well immediate offset!

**sw x14, 8 (x2)**

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>76</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>imm[4:0]</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**offset[11:5]**  | **src** | **base** | **width** | **offset[4:0]** | **STORE** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>01110</td>
<td>00010</td>
<td>010</td>
<td>01000</td>
<td>0100011</td>
</tr>
</tbody>
</table>

**offset[11:5]** = 0  
rs2 = 14  
rs1 = 2  
**SW** = 8  
STORE

**combined 12-bit offset = 8**
Datapath with lw

---

IMEM

PC

Add

alu

DMEM

Reg[ ]

Inst[24:20]

AddrA

DataA

Inst[19:15]

AddrB

DataB

Inst[11:7]

AddrD

DataD

Inst[31:0]

Control logic

Inst[31:0]

ImmSel

RegWriteEnable

BSel

ALUSel

MemRW

WBSel

---

Inst[31:20]

Imm. Gen

Imm[31:0]

ImmSel

Add

+4

Reg[rs1]

Reg[rs2]

WBSel

MemRW

PC

clk

Inst

addr

IMEM

PC

clk

pc+4

---

Garcia, Nikolić

RISC-V (44)
Adding $sw$ to Datapath

- $Add = Inst[24:20] + 4$
- $AddrB = Inst[19:15]$
- $DataB = AddrDB$
- $DataD = AddrDD$
- $Reg[rs1] = Imm[31:0]$
- $Reg[rs2] = Bsel$
- $DataR = ALUSel = Add$
- $DataW = MemRW = Write$
- $WBSel = *$
- $RegWriteEnable = 0$
- $Bsel = 1$
- $ALUSel = Add$
- $MemRW = Write$
- $WBSel = *$

Control logic:
- $Inst[31:0]$
- $ImmSel = S$
- $RegWriteEnable = 0$
- $Bsel = 1$
- $ALUSel = Add$
- $MemRW = Write$
- $WBSel = *$

(don't care)
Adding sw to Datapath

**Diagram Description:**

- **IMEM:** Instruction Memory
- **PC:** Program Counter
- **clk:** Clock Signal
- **Inst:** Instruction
- **AddrA, AddrB, AddrD:** Address Signals
- **DataA, DataB, DataD:** Data Signals
- **Reg[rs1], Reg[rs2]:** Register Signals
- **alu:** ALU Operation
- **ALU:** Arithmetic Logic Unit
- **DMEM:** Data Memory
- **DataB:** Data from DMEM
- **DataW:** Data write
- **MemRW:** Memory Read/Write
- **WBSel:** Write Back Selection
- **BSel:** Bus Selection
- **ImmSel:** Immediate Selection
- **Imm. Gen:** Immediate Generator
- **Imm[31:0]:** Immediate Value
- **Inst[31:0]:** Instruction Value
- **Inst[11:7], Inst[19:15], Inst[24:20]:** Instruction Fields
- **RegWEn:** Register Write Enable
- **ALUSel:** ALU Select

**Control Logic:**

- **ImmSel = S**
- **RegWEn = 0**
- **BSel = 1**
- **ALUSel = Add**
- **MemRW = Write**
- **WBSel = *"}

**Equations:**

### I+S Immediate Generation

<table>
<thead>
<tr>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
</tr>
<tr>
<td>imm[11:5]</td>
</tr>
<tr>
<td>rs1</td>
</tr>
<tr>
<td>funct3</td>
</tr>
<tr>
<td>rd</td>
</tr>
<tr>
<td>I-opcode</td>
</tr>
<tr>
<td>imm[4:0]</td>
</tr>
<tr>
<td>S-opcode</td>
</tr>
</tbody>
</table>

- Just need a 5-bit mux to select between two positions where low five bits of immediate can reside in instruction
- Other bits in immediate are wired to fixed positions in instruction
All RV32 Store Instructions

- Store byte writes the low byte to memory
- Store halfword writes the lower two bytes to memory

<table>
<thead>
<tr>
<th>Imm[11:5]</th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>imm[4:0]</th>
<th>0100011</th>
</tr>
</thead>
<tbody>
<tr>
<td>sb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

width
### RISC-V B-Format for Branches

- B-format is mostly same as S-Format, with two register sources ($rs1/rb2$) and a 12-bit immediate $imm[12:1]$.
- But now immediate represents values -4096 to +4094 in 2-byte increments.
- The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it).
Datapath So Far

- **IMEM**
  - PC (pc+4)
  - clk
  - addr
  - inst

- **Add**
  - +4

- **DataD**
  - Inst[11:7]
  - AddrD

- **DataA**
  - Inst[19:15]
  - AddrA

- **DataB**
  - Inst[24:20]
  - AddrB

- **Reg[rs1]**
  - DataD

- **Reg[rs2]**
  - DataA

- **ALU**
  - clk

- **DataR**
  - Alu

- **DMEM**
  - addr

- **DataW**
  - DataR

- **Control logic**
  - Inst[31:0]
  - ImmSel
  - RegWriteEnable
  - BSel
  - ALUSel
  - MemRW
  - WB Sel
To Add Branches

- Different change to the state:
  \[
  PC = \begin{cases} 
  PC + 4, & \text{branch not taken} \\
  PC + \text{immediate}, & \text{branch taken}
  \end{cases}
  \]

- Six branch instructions: \texttt{beq, bne, blt, bge, bltu, bgeu}

- Need to compute \texttt{PC + immediate} and to compare values of \texttt{rs1} and \texttt{rs2}
  - But have only one ALU – need more hardware
Adding Branches

Control logic

PCSel = taken/not taken

Inst[31:0] | ImmSel = B | RegWEn = 0 | BrUn | BrLT | Bsel | Asel | ALUSel = add | MemRW = read | WBSel = *
Branch Comparator

BrEq = 1, if A=B

BrLT = 1, if A<B

BrUn = 1 selects unsigned comparison for BrLT, 0=signed

BGE branch: A >= B, if A<B

A<B = !(A<B)
12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes

Standard approach: Treat immediate as in range -2048..+2047, then shift left by 1 bit to multiply by 2 for branches

Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit
Branch Immediates (In Other ISAs)

12-bit immediate encodes PC-relative offset of -4096 to +4094 bytes in multiples of 2 bytes

RISC-V approach: keep 11 immediate bits in fixed position in output value

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

Only one bit changes position between S and B, so only need a single-bit 2-way mux
### RISC-V Immediate Encoding

#### Instruction encodings, inst[31:0]

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 32-bit immediates produced, imm[31:0]

- **I-type**
  - instr[31] -

- **S-type**
  - instr[31] -

- **B-type**
  - instr[31] -

- Upper bits sign-extended from **inst[31]** always
- Only bit 7 of instruction changes role in immediate between S and B

---

Garcia, Nikolić

*RISC-V (57)*
Lighting Up Branch Path

Control logic:
- PCSel = taken/not taken
- Inst[31:0]
- ImmSel = B
- RegWE = 0
- BrUn = 0
- BrEq = 0
- BrLT = 0
- Bsel = 1
- ASEL = add
- MemRW = read
- WBSel = *

RISC-V (58)

Garcia, Nikolić
Adding JALR to Datapath
Let’s Add JALR (I-Format)

- JALR rd, rs, immediate
- Two changes to the state
  - Writes PC+4 to rd (return address)
  - Sets PC = rs1 + immediate
- Uses same immediates as arithmetic and loads
  - no multiplication by 2 bytes
  - LSB is ignored
Datapath So Far, With Branches
Datapath With JALR

RISC-V (62)

Control logic

PCSel = taken

Inst[31:0] ImmSel = l RegWEEn = 1

BrUn = * BrEq = * BrLT = *

Asel = 0 ALUSel = Add MemRW = Read WBSel = 2
Datapath With JALR

- **Add**: Adder for arithmetic operations.
- **IMEM**: Instruction Memory, stores instructions.
- **DMEM**: Data Memory, stores data.
- **PC**: Program Counter, holds the address of the next instruction.
- **Addr**: Address bus for data access.
- **Inst**: Instruction bus for instruction decoding.
- **Reg**: Register files for storing data.
- **Data**: Data bus for data transfer.
- **alu**: ALU for arithmetic and logic operations.
- **WBSel**: Write Back Selection, for write operations.
- **MemRW**: Memory Read/Write, for accessing memory.
- **Branch Comp**: Branch Comparator for conditional branching.
- **InstSel**: Instruction Selection for opcode decoding.
- **ImmSel**: Immediate Selection for immediate values.
- **Asel**: Address Selection for data memory access.
- **ALUSel**: ALU Selection for ALU operations.
- **BrEq**: Branch Equal, for conditional branches.
- **BrLT**: Branch Less Than, for conditional branches.
- **Bsel**: Branch Selection, for condition checking.
- **PCSel**: Program Counter Selection, for control flow.
- **RegWEn**: Register Write Enable, for register write operations.
- **Imm**: Immediate, for arithmetic operations.
- **Gen**: Generator, for generating immediate values.
- **clk**: Clock signal for timing operations.
Adding JAL
Two changes to the state
- \texttt{jal} saves PC+4 in register \texttt{rd} (the return address)
- Set PC = PC + offset (PC-relative jump)

Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- $\pm 2^{18}$ 32-bit instructions

Immediate encoding optimized similarly to branch instruction to reduce hardware cost
Datapath with JAL

García, Nikolić

---

**Control logic**

- **RegWEn** = 1
- **BrUn** = *
- **BrLT** = *
- **Bsel** = 0
- **ALUSel** = Add
- **MemRW** = Read
- **WBSel** = 2

**Addressing**

- **AddrA**
- **AddrB**
- **AddrD**

**Data**

- **DataA**
- **DataB**
- **DataD**

**Branch Condition**

- **Branch Comp**

**Immediate Generation**

- **Imm[31:20]**
- **ImmSel = J**

**Instruction**

- **Inst[31:0]**
- **Inst[19:15]**
- **Inst[11:7]**
- **Inst[24:20]**

**Register**

- **R[rs1]**
- **R[rs2]**

**Memory**

- **IMEM**
- **DMEM**

**Logic Gates**

- **Add**
- **Imm. Gen**
Light Up JAL Path

PCSel = taken
Inst[31:0]
ImmSel = J
Control logic
RegWEn = 1
BrUn = *
BrEq = *
Bsel = 0
Asel = Add
ALUSel = Add
MemRW = Read
WBSel = 2

Garcia, Nikolić

RISC-V (67)
Adding U-Types
U-Format for “Upper Immediate” Instructions

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Has 20-bit immediate in upper 20 bits of 32-bit instruction word
- One destination register, \( \text{rd} \)
- Used for two instructions
  - \text{lui} – Load Upper Immediate
  - \text{auipc} – Add Upper Immediate to PC
Datapath With LUI, AUIPC

Control logic

PCSel = pc+4

Inst[31:0] ImmSel = U

RegWEn = 1

BrUn = * BrEq = * BrLT = *

Bsel = 1 Asel = *

ALUSel = Read MemRW = Read WBSel = 1

Imm. Gen

Inst[31:20]

Inst[19:15]

Inst[11:7]

Inst[24:20]

IMEM

DataD

AddrD

AddrA

DataA

AddrB

MemRW

alu

ALU

DMEM

DataB

AddrB

DataD

AddrD

R[rs1]

R[rs2]

Imm[31:0]

Imm[31:0]

mem

0

1

0

1

0

1

1

0

2

clk

clk

addr inst

+4

Add

0

1

pc+4

1

0

PC

0

1

1

0

Lighting Up LUI

RISC-V (71)

Garcia, Nikolić

Berkeley
UNIVERSITY OF CALIFORNIA
Lighting Up AUIPC

Control logic

RISC-V (72)
“And In Conclusion...”
Complete RV32I Datapath!
### Complete RV32I ISA!

#### OpenRISC Reference Card

**Base Integer Instructions: RV32I**

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
<th>Category</th>
<th>Name</th>
<th>Fmt</th>
<th>RV32I Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shifts</td>
<td>Shift Left Logical</td>
<td>R</td>
<td>SLL rd,rs1,rs2</td>
<td>Loads</td>
<td>Load Byte</td>
<td>I</td>
<td>LB rd,rs1,imm</td>
</tr>
<tr>
<td></td>
<td>Shift Left Log. Imm.</td>
<td>I</td>
<td>SLLI rd,rs1,shamt</td>
<td>Load Halfword</td>
<td>LH rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Logical</td>
<td>R</td>
<td>SRL rd,rs1,rs2</td>
<td>Load Byte Unsigned</td>
<td>LBU rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Log. Imm.</td>
<td>I</td>
<td>SRLI rd,rs1,shamt</td>
<td>Load Half Unsigned</td>
<td>LHU rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arithmetic</td>
<td>R</td>
<td>SRA rd,rs1,rs2</td>
<td>Load Word</td>
<td>LW rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shift Right Arith. Imm.</td>
<td>I</td>
<td>SRAI rd,rs1,shamt</td>
<td>Stores</td>
<td>Store Byte</td>
<td>S</td>
<td>SB rs1,rs2,imm</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADD</td>
<td>R</td>
<td>ADD rd,rs1,rs2</td>
<td>Store Halfword</td>
<td>SH rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD Immediate</td>
<td>I</td>
<td>ADDI rd,rs1,imm</td>
<td>Store Word</td>
<td>SW rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUBtract</td>
<td>R</td>
<td>SUB rd,rs1,rs2</td>
<td>Branches</td>
<td>Branch =</td>
<td>B</td>
<td>BEQ rs1,rs2,imm</td>
</tr>
<tr>
<td></td>
<td>Load Upper Imm</td>
<td>U</td>
<td>LUI rd,imm</td>
<td>Branch =</td>
<td>B</td>
<td>BNE rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add Upper Imm to PC</td>
<td>U</td>
<td>AUIPC rd,imm</td>
<td>Branch &lt;</td>
<td>B</td>
<td>BLT rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>XOR</td>
<td>R</td>
<td>XOR rd,rs1,rs2</td>
<td>Branch ≥</td>
<td>B</td>
<td>BGE rs1,rs2,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XOR Immediate</td>
<td>I</td>
<td>XORI rd,rs1,imm</td>
<td>Branch &lt; Unsigned</td>
<td>BLTU rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>R</td>
<td>OR rd,rs1,rs2</td>
<td>Branch ≥ Unsigned</td>
<td>BGEU rs1,rs2,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR Immediate</td>
<td>I</td>
<td>ORI rd,rs1,imm</td>
<td>Jump &amp; Link</td>
<td>J</td>
<td>JAL rd,imm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>R</td>
<td>AND rd,rs1,rs2</td>
<td>Jump &amp; Link Register</td>
<td>JALR rd,rs1,imm</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AND Immediate</td>
<td>I</td>
<td>ANDI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>Set &lt;</td>
<td>R</td>
<td>SLT rd,rs1,rs2</td>
<td>Synch</td>
<td>Synch thread</td>
<td>I</td>
<td>FENCE</td>
</tr>
<tr>
<td></td>
<td>Set &lt; Immediate</td>
<td>I</td>
<td>SLTI rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set &lt; Unsigned</td>
<td>R</td>
<td>SLTU rd,rs1,rs2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Set &lt; Imm Unsigned</td>
<td>I</td>
<td>SLTIU rd,rs1,imm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Not in 61C

---

Garcia, Nikolić

---

Berkeley

**RISC-V (75)**

---

CC BY-NC-SA
We have designed a complete datapath
- Capable of executing all RISC-V instructions in one cycle each
  - Not all units (hardware) used by all instructions

5 Phases of execution
- IF, ID, EX, MEM, WB
  - Not all instructions are active in all phases

Controller specifies how to execute instructions
- We still need to design it