RISC-V is now a rising star in the industry, largely due to its open-source advantage, better power consumption performance promise, reliable security functions and lower political risk impact yet.

https://www.eetasia.com/risc-v-to-shake-up-8-6b-semiconductor-ip-market/

Review

• Finite state machines: Common example of sequential logic
  • Moore’s machine: Output depends only on the current state
  • Mealy’s machine: Output depends on the current state and the input
• Large state machines can be factored
• Common Verilog patterns for FSMs
• Common job interview questions 😊
Building a RISC-V Processor

Berkeley RISC-V ISA

www.riscv.org

• An open, license-free ISA
  • Runs GCC, LLVM, Linux distributions, ...
  • RV32, RV64, and RV128 variants for 32b, 64b, and 128b address spaces
• Originally developed for teaching classes at Berkeley, now widely adopted
• Base ISA only ~40 integer instructions
• Extensions provide full general-purpose ISA, including IEEE-754/2008 floating-point
• Designed for extension, customization
• Developed at UC Berkeley, now maintained by RISC-V Foundation
• Open and commercial implementations
• RISC-V ISA, datapath, and control covered in CS61C; summarized here
RISC-V Processor Design

- Spec: Unprivileged ISA, RV32I (and a look at RV64I)

  - Specification (e.g. in plain text)
  - Model (e.g. in C/C++)
  - Tests and test vectors
  - Architecture (e.g. in-order, out-of-order)
  - Validation
  - Verification
  - RTL Logic Design (e.g. in Verilog)

- Tests provided as a part of the project
- Architecture: Single-cycle and pipelined in-order processor
  - Expanded from CS61C

One-Instruction-Per-Cycle RISC-V Machine

- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settles 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
State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- **Registers (x0..x31)**
  - Register file (regfile) Reg holds 32 registers x 32 bits/ register: \( \text{Reg[0]} .. \text{Reg[31]} \)
  - 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

- **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
    - Load/store instructions access data memory

Stages of the Datapath: Overview

- **Problem**: A single, “monolithic” CL block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction and completing with the register access) is 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
5. Register Write

Clock

Time
Datapath Components: Combinational

- Combinational Elements

- Storage Elements + Clocking Methodology

- Building Blocks

Datapath Elements: State and Sequencing (1/4)

- Register

- Write Enable:
  - Negated (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on positive edge of clock

```
always @(posedge clk)
  if (wen) dataout <= datain;
endmodule
```
Datapath Elements: State and Sequencing (2/4)

- Register file (regfile, RF) consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
  - x0 is wired to 0

- 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 \( \Rightarrow \) busA or busB valid after "access time."

Datapath Elements: State and Sequencing (3/4)

- Reg file in Verilog

```verilog
module rv32i_regs (  
  input clk, wen,  
  input [4:0] rw,  
  input [4:0] ra,  
  input [4:0] rb,  
  input [31:0] busw,  
  output [31:0] busa,  
  output [31:0] busb  
);  
  reg [31:0] regs [0:30];  
  always @ (posedge clk)  
    if (wen) regs[rw] <= busw;  
    assign busa = (ra == 5'd0) ? 32'd0: regs[ra];  
    assign busb = (rb == 5'd0) ? 32'd0: regs[rb];  
endmodule
```

- How does RV64I register file look like?
Datapath Elements: State and Sequencing (4/4)

- “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 → Data Out valid after “access time”
- Real memory later in the class

---

Review: Complete RV32I ISA

**Open Reference Card**

<table>
<thead>
<tr>
<th>Base Integer Instructions RV32I</th>
<th>Category</th>
<th>Name</th>
<th>Format</th>
<th>RV32I Base</th>
<th>Category</th>
<th>Name</th>
<th>Format</th>
<th>RV32I Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shifts: Shift Left Logical</td>
<td>R</td>
<td>SLL</td>
<td>rd,rs1,rs2</td>
<td>Loads</td>
<td>I</td>
<td>BL</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Shift Left Logical</td>
<td>I</td>
<td>SLW</td>
<td>rd,rs1,shamt</td>
<td>Load Halfword</td>
<td>I</td>
<td>SH</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td>R</td>
<td>SRL</td>
<td>rd,rs1,rs2</td>
<td>Load Byte</td>
<td>I</td>
<td>LUI</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td>I</td>
<td>SRLI</td>
<td>rd,rs1,shamt</td>
<td>Load Half</td>
<td>I</td>
<td>ADDI</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>R</td>
<td>SRA</td>
<td>rd,rs1,rs2</td>
<td>Load Word</td>
<td>I</td>
<td>ALOI</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>I</td>
<td>SRAI</td>
<td>rd,rs1,shamt</td>
<td>Load</td>
<td>I</td>
<td>ANDI</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>ADD</td>
<td>ADD</td>
<td>rd,rs1,rs2</td>
<td>Store Word</td>
<td>S</td>
<td>SW</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Add Immediate</td>
<td>ADDI</td>
<td>ADDI</td>
<td>rd,imm</td>
<td>Store Word</td>
<td>S</td>
<td>SW</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>SUB</td>
<td>rd,rs1,rs2</td>
<td>Branches</td>
<td>Branch =</td>
<td>B</td>
<td>BEQ</td>
<td>rd,rs2,imm</td>
</tr>
<tr>
<td>Add Upper Imm</td>
<td>U</td>
<td>ADDI</td>
<td>rd,imm</td>
<td>Branches</td>
<td>Branch #</td>
<td>B</td>
<td>BNE</td>
<td>rd,rs2,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>XOR</td>
<td>XOR</td>
<td>rd,rs1,rs2</td>
<td>Branch</td>
<td>Branch &gt;</td>
<td>B</td>
<td>BEQ</td>
<td>rd,rs2,imm</td>
</tr>
<tr>
<td>Logical</td>
<td>OR</td>
<td>ORI</td>
<td>rd,imm</td>
<td>Branches</td>
<td>Branch &gt;</td>
<td>B</td>
<td>BGE</td>
<td>rd,rs2,imm</td>
</tr>
<tr>
<td>Load Upper Imm</td>
<td>U</td>
<td>ORI</td>
<td>rd,imm</td>
<td>Branches</td>
<td>Branch &gt;</td>
<td>B</td>
<td>BGEU</td>
<td>rd,rs2,imm</td>
</tr>
<tr>
<td>Add Immediate</td>
<td>AND</td>
<td>ANDI</td>
<td>rd,rs1,rs2</td>
<td>Jump &amp;</td>
<td>Jump &amp; Register</td>
<td>JAL</td>
<td>rd,imm</td>
<td></td>
</tr>
<tr>
<td>COMPARE</td>
<td>SLT</td>
<td>SLT</td>
<td>rd,rs1,rs2</td>
<td>Synchron</td>
<td>Synchron</td>
<td>I</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>COMPARE</td>
<td>SLT</td>
<td>SLT</td>
<td>rd,rs1,imm</td>
<td>Synchron</td>
<td>Synchron</td>
<td>I</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>COMPARE</td>
<td>Set &lt;</td>
<td>Set &lt;</td>
<td>rd,rs1,imm</td>
<td>Environment</td>
<td>CALL</td>
<td>I</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>COMPARE</td>
<td>Set &lt;</td>
<td>Set &lt;</td>
<td>rd,rs1,imm</td>
<td>Environment</td>
<td>CALL</td>
<td>I</td>
<td></td>
<td>R</td>
</tr>
</tbody>
</table>

- Need datapath and control to implement these instructions
Quiz

1) We should use the main ALU to compute PC=PC+4 in order to save some gates.
2) The ALU is a sequential element.
3) Program counter is a register.

www.yellkey.com/picture

R-Format Instructions: Datapath
Summary of RISC-V Instruction Formats

<table>
<thead>
<tr>
<th>Field's bit positions</th>
<th>Name of field</th>
<th>Number of bits in field</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 30 25 24 21 20 19</td>
<td>funct7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>rs2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>rs1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>funct3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>rd</td>
<td>5</td>
</tr>
<tr>
<td>15 14 12 11 8 7 6</td>
<td>opcode</td>
<td>7</td>
</tr>
</tbody>
</table>

- **R-type**
- **I-type**
- **S-type**
- **B-type**
- **U-type**
- **J-type**

• 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32

• Examples
  - *opcode* is a 7-bit field that lives in bits 6-0 of the instruction
  - *rs2* is a 5-bit field that lives in bits 24-20 of the instruction
### R-Format Instructions opcode/ funct fields

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to \texttt{0110011}_2 for all R-Format register-register arithmetic instructions
- **funct7+funct3**: combined with opcode, these two fields describe what operation to perform

**Question:** You have been professing simplicity, so why aren't opcode and funct 7 and funct 3 a single 17-bit field?  
- Simpler implementation is more important than simpler spec

### R-Format Instructions register specifiers

<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>20 19</th>
<th>15 14</th>
<th>12 11</th>
<th>7 6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

- **rs1** (Source Register #1): specifies register containing first operand
- **rs2** : specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)
R-Format Example

- RISC-V Assembly Instruction:
  \[ \text{add} \ x18, x19, x10 \]

```
+---+---+--+-+--++---+---+---+
<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>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

0000000 01010 10011 000 10010 0110011

```
add \text{rs2}=10 \text{rs1}=19 \text{add} \text{rd}=18 \text{Reg-Reg OP}
```

Implementing the \textit{add} instruction

```
+---+---+--+-+--++---+---+---+
<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>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

0000000 \text{rs2} \text{rs1} 000 \text{rd} 0110011

```
add \text{rs2} \text{rs1} \text{add} \text{rd} \text{Reg-Reg OP}
```

```
add \text{rd}, \text{rs1}, \text{rs2}
```

- Instruction makes two changes to machine's state:
  - \( \text{Reg}[ \text{rd} ] = \text{Reg}[ \text{rs1} ] + \text{Reg}[ \text{rs2} ] \)
  - \( \text{PC} = \text{PC} + 4 \)
Datapath for **add**

\[ PC = PC + 4 \quad \text{Reg}[rd] = \text{Reg}[rs1] + \text{Reg}[rs2] \]

Timing Diagram for **add**

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

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
rs2 & rs1 & 0 & 0 & 0 & rd & opcode & \\
5 & 5 & add & 5 & add & Reg-Reg OP & \\
\end{array} \]
Implementing the *sub* instruction

<table>
<thead>
<tr>
<th></th>
<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>add</td>
<td>00000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>011011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>01000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>011011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*sub* rd, rs1, rs2

- Almost the same as *add*, except now have to subtract operands instead of adding them
- *inst[30]* selects between *add* and *subtract*

---

# Datapath for *add/sub*

- IMEM
- ALU
- Add
- ALUSel
  - (add=0, sub=1)
- RegWEn
  - (1=Write, 0=NoWrite)
- Control logic
- DataD
- AddrA
- AddrB
- Inst[11:7]
- Inst[19:15]
- Inst[24:20]
- Reg[ ]
- pc+4
- AddrD
- DataB
- Inst[31:0]
## Implementing other R-Format instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs2</th>
<th>rs1</th>
<th>funct7</th>
<th>rd</th>
<th>funct3</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>add</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>sub</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
<td>sll</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td>slt</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
<td>sltu</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td>xor</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>srl</td>
</tr>
<tr>
<td>0100000</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>sra</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td>or</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>and</td>
</tr>
</tbody>
</table>

- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

---

### Administrivia

- Homework 3 is due next Monday
  - Homework 4 will be posted this week, due before midterm1
- Lab 4 this week
- Lab 5 next week
- Midterm 1 on October 7, 7-8:30pm
Instruction Encoding

• Instructions are encoded to simplify logic
  • sub and sra differ in Inst[30] from add and srl
  • RV64I widens registers (XLEN=64)
  • Additional instructions manipulate 32-bit values, identified by a suffix W
  • ADDW, SUBW
  • RV64I opcode field for ‘W’ instructions is 011011 (0110011 for RV32I)

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs2</th>
<th>rs1</th>
<th>rd</th>
<th>011011</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
</tr>
<tr>
<td>0000000</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
</tr>
</tbody>
</table>
I-Format Instruction Layout

• Only one field is different from R-format, rs2 and funct7 replaced by 12-bit signed immediate, \( \text{imm}[11:0] \)
• Remaining fields (rs1, funct3, rd, opcode) same as before
• \( \text{imm}[11:0] \) can hold values in range \([-2048_{\text{ten}}, +2047_{\text{ten}}]\)
• Immediate is always sign-extended to 32-bits before use in an arithmetic operation
• Other instructions handle immediates > 12 bits

All RV32 I-format Arithmetic Instructions

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>rd</th>
<th>opcode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>000</td>
<td></td>
<td>0010011</td>
<td>addi</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td></td>
<td>0010011</td>
<td>slti</td>
</tr>
<tr>
<td></td>
<td>011</td>
<td></td>
<td>0010011</td>
<td>sltiu</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
<td>0010011</td>
<td>xori</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
<td>0010011</td>
<td>ori</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td></td>
<td>0010011</td>
<td>andi</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>001</td>
<td>slli</td>
</tr>
<tr>
<td>00000000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
<td>srai</td>
</tr>
<tr>
<td>01000000</td>
<td>shamt</td>
<td>rs1</td>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

The same Inst[30] immediate bit is used to distinguish “shift right logical” (SRLI) from “shift right arithmetic” (SRAI)

“Shift-by-immediate” instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)
Implementing I-Format - \texttt{addi} instruction

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

\begin{verbatim}
<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</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></td>
<td></td>
<td></td>
<td>opcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
111111001110 00001 000 01111 0010011
\end{verbatim}

\begin{verbatim}
imm=-50  rs1=1  add  rd=15  OP-Imm
\end{verbatim}

\textbf{Datapath for add/sub}

Immediate should be here

\begin{verbatim}
ALUSel (add=0/sub=1)
\end{verbatim}
Adding `addi` to Datapath

```
+4
Add

pc+4
PC

addr
inst
IMEM

Inst[11:7]
Inst[19:15]
Inst[24:20]

AddD

AddD
AddA
AddB

DataD

Reg[]

ALU

Reg[rs1]
Reg[rs2]

alu

Imm[31:0]

Inst[31:0]

Control logic

RegWEn
(1=Write, 0=NoWrite)

ALUSel
(add=0/sub=1)

BSEL
(rs2=0/Imm=1)

ImmSel
(add=0/imm=1)

Imm[31:0]

Imm[31:0]

ImmGen

Inst[31:20]

Inst[24:20]

Inst[11:7]

Inst[19:15]

Inst[]

BSEL
ImmSel

Reg[1]

RegWEn

0
1

```

Nikolić, Fall 2021

EECS151 L08 RISC-V
I-Format immediates

- \text{inst}[31]-

\begin{array}{cccccc}
31 & 30 & 29 & 28 & 27 & 26 \\
\text{imm}[11:0] & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} & \text{inst}[31:0]\end{array}

\text{---inst}[31]-\text{(sign-extension)}-\text{inst}[30:20]\\

- \text{Imm}[31:0]

- High 12 bits of instruction (\text{inst}[31:20]) copied to low 12 bits of immediate (\text{imm}[11:0])
- Immediate is sign extended by copying value of \text{inst}[31] to fill the upper 20 bits of the immediate value (\text{imm}[31:12])
- Sign extension often in critical path

R+I Datapath

Works for all other I-format arithmetic instructions (\text{slti, sltiu, andi, ori, xori, slli, srai}) just by changing ALUSel
Add lw to Datapath

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

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>5</th>
<th>3</th>
<th>5</th>
<th>7</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>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>width</td>
<td>dest</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000001000</td>
<td>00010</td>
<td>010</td>
<td>01110</td>
<td>0000011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The 12-bit signed immediate is added to the base address in register 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 rd.
All RV32 Load Instructions

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0000011</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0000011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0000011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0000011</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0000011</td>
</tr>
</tbody>
</table>

• 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 for load extend, similar to sign extension for immediates.

Nikolić, Fall 2021
EECS151 L08 RISC-V

S-Format Instructions: Datapath

Nikolić, Fall 2021
EECS151 L08 RISC-V
S-Format Used for Stores

- Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!
- Can’t have both rs2 and immediate in same place as other instructions!
- Note that stores don’t write a value to the register file, rd!
- RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place
  - register names more critical than immediate bits in hardware design

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)
```

```
00000000 01110 00010 010 01000 0100011
offset[11:5] rs2=14 rs1=2 SW offset[4:0] STORE
```

Combined 12-bit offset = 8
Datapath with $1w$

Adding $sw$ to Datapath
All RV32 Store Instructions

<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>Imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>imm[4:0]</td>
<td>0100011</td>
</tr>
<tr>
<td>Imm[11:5]</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>imm[4:0]</td>
<td>0100011</td>
</tr>
</tbody>
</table>

- Store byte, halfword, word

I+S Immediate Generation

• 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
Summary

• RISC-V ISA
  • Open, with increasing adoption

• RISC-V processor
  • A large state machine
  • Datapath + control
  • Reviewed R-, I-, S-format instructions and corresponding datapath elements