Recap: Complete RV32I ISA

Not in CS61C
State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- **Registers ($x0 \ldots x31$)**
  - Register file (or *regfile*) \texttt{Reg} holds 32 registers x 32 bits/register: \texttt{Reg[0]} \ldots \texttt{Reg[31]}
  - First register read specified by \texttt{rs1} field in instruction
  - Second register read specified by \texttt{rs2} field in instruction
  - Write register (destination) specified by \texttt{rd} field in instruction
  - \texttt{x0} is always 0 (writes to \texttt{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 (\texttt{IMEM}) and data (\texttt{DMEM})
    - Later we’ll replace these with instruction and data caches
  - Instructions are read (*fetched*) from instruction memory (assume \texttt{IMEM} read-only)
  - Load/store instructions access data memory
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.
Basic Phases of Instruction Execution

1. Instruction Fetch
2. Decode/Register Read
3. Execute
4. Memory
5. Register Write
Implementing the **add** instruction

<table>
<thead>
<tr>
<th>0000000</th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th>0110011</th>
<th>ADD</th>
</tr>
</thead>
</table>

**add rd, rs1, rs2**

- Instruction makes two changes to machine’s state:
  - $\text{Reg}[rd] = \text{Reg}[rs1] + \text{Reg}[rs2]$
  - $PC = PC + 4$
Control Logic

Datapath for **add**
Timing Diagram for add

Clock
PC
PC+4
inst[31:0]
Reg[rs1]
Reg[rs2]
alu
Reg[1]
Reg[2]
Reg[3]
Reg[7]
Reg[9]

1000
1004
1008

add x1, x2, x3
add x6, x7, x9

pc+4
+4
IMEM
inst[31:0]
Reg[]
DataD
AddrD
inst[11:7]
AddrA
DataA
inst[19:15]
AddrB
DataB
inst[24:20]
RegWEn

Clock

PC

PC+4

inst[31:0]

Reg[rs1]

Reg[rs2]

alu

Reg[1]

Reg[2]

Reg[3]

Reg[7]

Reg[9]


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>ADD</td>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>0100000</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**

Control Logic

```
id[31:0]
RegWEen
(1=write, 0=no write)
ALUSel
(Add=0/Sub=1)
```
Implementing other R-Format instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>Result</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
Implementing the **addi** instruction

- RISC-V Assembly Instruction:
  
  ```
  addi x15, x1, -50
  ```

<table>
<thead>
<tr>
<th>31</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>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

| 111111001110 | 00001 | 000 | 01111 | 0010011 |

- `imm=-50`
- `rs1=1`
- `ADD`
- `rd=15`
- `OP-Imm`
Datapath for add/sub

Control Logic

- pc+4
- IMEM
- ALU
- ALUSel (Add=0/Sub=1)
- RegWEn (1=write, 0=no write)
- inst[31:0]
Adding **addi** to datapath

**Control Logic**

- **IMEM**
- **IMEM[11:7]**
- **IMEM[19:5]**
- **IMEM[24:20]**
- **IMEM[31:0]**

**Reg[]**
- **Reg[rs1]**
- **Reg[rs2]**
- **RegWEn=1**
- **BSel=1**
- **ALUSel=Add**

**ALU**
- **alu**
- **PC+4**
- **inst[31:0]**
- **ImmSel=I**
- **imm[31:0]**
- **inst[31:20]**
- **inst[19:5]**
- **inst[11:7]**

**Reg[]**
- **DataD**
- **AddrD**
- **AddrA**
- **DataA**
- **AddrB**
- **DataB**

**Imm. Gen**
- **imm[31:0]**
- **inst[31:20]**

**Control Logic**

- **pc+4**
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 **addi** to datapath

Also works for all other *I*-format arithmetic instruction (slti, sltiu, andi, ori, xori, sll, srl, srai) just by changing ALUSel
TSMC Announces 3nm CMOS Fab

Latest Apple iPhone 8, iPhone X use TSMC’s 10nm process technology.

3nm technology should allow 10x more stuff on the same sized chip (10/3)^2

The new manufacturing plant will occupy nearly 200 acres and cost around $15B, open in around 5 years (~2022).

Currently, fabs use 193nm light to expose masks
For 3nm, some layers will use Extreme Ultra-Violet (13.5nm)
Break!
Implementing Load Word instruction

• RISC-V Assembly Instruction:
  \[ \text{lw} \ x14, \ 8(x2) \]

\[
\begin{array}{cccccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
  \text{imm}[11:0] & \text{rs1} & \text{funct3} & \text{rd} & \text{opcode} \\
  12 & 5 & 3 & 5 & 7 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  31 & 30 & 29 & 28 & 27 & 26 & 25 & 24 \\
  \text{imm=+8} & \text{rs1=2} & \text{LW} & \text{rd=14} & \text{LOAD} \\
\end{array}
\]
Adding **addi** to datapath
Adding **lw** to datapath
Adding **lw** to datapath
### All RV32 Load Instructions

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

- funct3 field encodes size and signedness of load data

- Supporting the narrower loads requires additional circuits 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.
Implementing Store Word instruction

- RISC-V Assembly Instruction:
  \texttt{sw x14, 8(x2)}

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
31 & 25 & 24 & 20 & 19 & 15 & 14 & 12 & 11 & 7 & 6 & 0 \\
\hline
\hline
7 & 5 & 5 & 3 & 5 & 7 \\
\end{tabular}

\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
00000000 & 01110 & 00010 & 010 & 01000 & 0100011 \\
\hline
\hline
=0 & & & & & =8 \\
\hline
\end{tabular}

combined 12-bit offset = 8

10/3/17
Adding $lw$ to datapath
Adding **sw** to datapath
Adding **SW** to datapath

![Diagram showing the datapath with SW added](image-url)

```
Adding sw to datapath

pc+4 -> IMEM
+4 -> IMEM
IMEM -> Reg[]
Reg[] -> ALU
Reg[] -> DMEM
ALU -> DMEM
DMEM -> wb

pc+4 -> pc
inst[19:15] -> AddrA
inst[24:16] -> AddrB
inst[31:20] -> AddrD

Imm[31:0] -> imm[31:0]

Reg[rs1] -> ALU
Reg[rs2] -> ALU

Bsel = 1
ALUSel = Add
MemRW = Write
WBSel = *

* = “Don’t Care”
```
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])
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
Implementing Branches

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- 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)
Adding **sw** to datapath
Adding branches to datapath
Adding branches to datapath
Branch Comparator

- \text{BrEq} = 1, \text{if} \ A=B
- \text{BrLT} = 1, \text{if} \ A < B
- \text{BrUn} = 1 \text{ selects unsigned comparison for BrLT, 0=signed}

- BGE branch: \ A \geq \ B, \text{if} \ !\left(A < B\right)
• Midterm 1 has been graded!

• Regrade Requests will open tonight
  – Due next Tuesday (in one week)
  – Piazza will explain the instructions
Administrivia (2/2)

• Project 1 has been released
  – Part 1 is due next Monday
  – Project Party in Cory 293 on Wednesday 7-9pm (possibly later if needed)

• Homework 2 is due this Friday at 11:59pm
  – Will help to do this before the project!

• No Guerrilla Session this week—will start up again next Tuesday
Break!
Multiply Branch Immediates by Shift?

- 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

```
s imm[10:5]  rs2  rs1  funct3  imm[4:0]  B-opcode
```

- **S-Immediate**
  - Sign-extension
  - \(s\) \(\text{imm}[10:5]\) \(\text{imm}[4:0]\)

- **B-Immediate (shift left by 1)**
  - Sign-extension
  - \(s\) \(\text{imm}[10:5]\) \(\text{imm}[4:0]\) \(0\)

Each instruction immediate bit can appear in one of two places in output immediate value – so need one 2-way mux per bit
RISC-V Branch Immediates

• 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, and rotate LSB of S-format to be bit 12 of B-format

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

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

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]

| 31 | 30 | 25 | 24 | 21 | 20 | 19 | 15 | 14 | 12 | 11 | 8 | 7 | 6 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type |
| imm[11:0] | rs1 | funct3 | rd | opcode | I-type |

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

- I-immediate: 
  - inst[31]
  - inst[30:25]
  - inst[24:21]
  - inst[20]

- S-immediate: 
  - inst[31]
  - inst[30:25]
  - inst[11:8]
  - inst[7]

- B-immediate: 
  - inst[31]
  - inst[7]
  - inst[30:25]
  - inst[11:8]
  - 0

**Upper bits sign-extended from inst[31] always**

**Only bit 7 of instruction changes role in immediate between S and B**
Implementing **JALR** Instruction (I-Format)

<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>opcode</td>
<td></td>
<td></td>
<td></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>
<tr>
<td>offset[11:0]</td>
<td>base</td>
<td>0</td>
<td>dest</td>
<td>JALR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- JALR rd, rs, immediate
  - Writes PC+4 to Reg[rd] (return address)
  - Sets PC = Reg[rs1] + immediate
  - Uses same immediates as arithmetic and loads
    - *no* multiplication by 2 bytes
Adding branches to datapath
Adding jalr to datapath

Adding jalr to the datapath involves modifying the existing pipeline stages to handle the jalr instruction. The diagram illustrates the flow of data and control signals through the different stages of the pipeline:

- **IMEM**: Instruction memory stage
- **alu**: ALU operation stage
- **pc**: Program counter stage
- **Inst[31:0]**: Instruction word
- **Imm[31:0]**: Immediate data
- **RegWEn**: Register write enable
- **BrUn**: Branch on unequal
- **BrEq**: Branch on equal
- **BrLT**: Branch on less than
- **BSel**: Branch select
- **ASel**: ALU select
- **ALUSel**: ALU input select
- **MemRW**: Memory read/write
- **WBSel**: Write back select
- **AddrA**, **AddrB**, **DataA**, **DataB**, **AddrD**, **DataD**: Address and data signals
- **Reg[rs1]**, **Reg[rs2]**: Register operands
- **Branch Comp.**: Branch comparator

The jalr instruction modifies the program counter (PC) by adding a value from a register to the current PC value, effectively changing the program flow. The diagram shows how the PC is updated after the jalr instruction is executed, reflecting the new branch target address.
Adding jalr to datapath
Implementing **jal** Instruction

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>7 JAL</td>
</tr>
</tbody>
</table>

- JAL saves PC+4 in Reg[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
Adding jal to datapath
Adding jal to datapath

Adding jal to datapath
Single-Cycle RISC-V RV32I Datapath
And in Conclusion, ...

• Universal 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
  – what new instructions can be added with just most control?