RISC-V Processor Datapath
Administrivia

• Lecture quizzes will now be due on Saturday each week
• Project 2 Deadlines
  • Part A due this Thursday (2/25)
  • Part B due next Thursday (3/4)
• Project 1 Clobbering
  • Will use the project1-practice autograder
  • Multiply your score by 0.7
  • You must submit test cases for consideration
Great Idea #1: Abstraction
(Lords of Representation/Interpretation)

lw  t0, t2, 0
lw  t1, t2, 4
sw  t1, t2, 0
sw  t0, t2, 4

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

We are here!
Recap: Complete RV32I ISA
“State” Required by RV32I ISA

Each instruction reads and updates this state during execution:

- **Registers ($x0 .. x31$)**
  - Register file (or *regfile*) `Reg` holds 32 registers x 32 bits/register: `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

- **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*)
    - *Later we’ll replace these with instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory (assume *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

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

2. At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

3. Separate instruction/data memory:
   For simplification, memory is asynchronous read (not clocked), but synchronous write (is clocked)
Basic Phases of Instruction Execution

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

| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |

**add rd, rs1, rs2**

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

pc+4

IMEM

inst[11:7]

inst[19:15]

inst[24:20]

Reg[]

DataD

AddrD

AddrA

DataA

AddrB

DataB

Reg[rs1]

Reg[rs2]

alu

inst[31:0]

RegWEn (RegWriteEnable)
(1=write, 0=no write)

Control Logic
Timing Diagram for \textit{add}

- \textbf{PC}: 1000, 1004
- \textbf{PC+4}: 1004, 1008
- \textbf{inst}[31:0]:
  - add $_x1,x2,x3$
  - add $_x6,x7,x9$
- \textbf{Reg[rs1]}: Reg[2], Reg[7]
- \textbf{Reg[rs2]}: Reg[3], Reg[9]
- \textbf{alu}:
Implementing the \texttt{sub} instruction

- Almost the same as \texttt{add}, except now have to subtract operands instead of adding them
- \texttt{inst[30]} selects between \texttt{add} and \texttt{subtract}
Datapath for \textit{add/sub}

\begin{itemize}
  \item \textbf{pc+4} \rightarrow \text{IMEM} \rightarrow \text{Reg[\textbf{rs2}]} \rightarrow \text{alu}
  \item \textbf{pc} \rightarrow \text{IMEM} \rightarrow \text{Reg[\textbf{rs1}]} \rightarrow \text{alu}
  \item \text{inst[31:0]} \rightarrow \text{RegWEn} \rightarrow \text{ALUSel} \rightarrow \text{alu}
\end{itemize}

\begin{itemize}
  \item \text{inst[11:7]} \rightarrow \text{AddrD}
  \item \text{inst[19:15]} \rightarrow \text{AddrA DataA}
  \item \text{inst[24:20]} \rightarrow \text{AddrB DataB}
\end{itemize}

\begin{itemize}
  \item \text{ALUSel} (Add=0/Sub=1)
  \item \text{RegWEn} (1=write, 0=no write)
\end{itemize}

\textbf{Control Logic}
Implementing other R-Format instructions

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

ADD
SUB
SLL
SLT
SLTU
XOR
SRL
SRA
OR
AND
Implementing the \texttt{addi} instruction

- RISC-V Assembly Instruction:
  \texttt{addi x15, x1, -50}

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

\begin{align*}
\text{imm} &= -50 \\
\text{rs1} &= 1 \\
\text{ADD} & \\
\text{rd} &= 15 \\
\text{OP-Imm} & \\
\end{align*}
Datapath for add/sub

Control Logic

IMEM

Reg[]

Reg[rs1]

Reg[rs2]

alu

inst[31:0]

inst[11:7]

inst[19:15]

inst[24:20]

DataD

AddrD

DataA

AddrA

DataB

AddrB

RegWEn

(1=write, 0=no write)

ALUSel

(Add=0/Sub=1)
Adding addi to datapath

Control Logic

pc

+4

pc+4

IMEM

inst[11:7]

inst[19:15]

inst[24:20]

inst[31:0]

Imm. Gen

imm[31:0]

Reg[rs1]

Reg[rs2]

DataD

AddrD

AddrA DataA

AddrB DataB

Reg[rs]

ALU

alu

ALUSel=Add

RegWEn=1

BSel=1

ImmSel=I

inst[31:20]
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**, **slli**, **srli**, **srai**) just by changing ALUSel
Implementing Load Word instruction

- RISC-V Assembly Instruction:
  lw x14, 8(x2)

```
   imm[11:0]  |   rs1    |   funct3  |   rd    |   opcode
  | 12       |   5      |   3      |   5     |   7
  000000001000 | 00010 | 010 | 01110 | 0000011
```

imm=+8  rs1=2  LW  rd=14  LOAD

2/22/18
Adding \texttt{addi} to datapath

\begin{itemize}
\item \textit{IMEM}:
  \begin{itemize}
  \item \texttt{inst[31:7]}
  \item \texttt{inst[19:15]}
  \item \texttt{inst[24:20]}
  \item \texttt{inst[31:0]}
  \end{itemize}
\item \textit{Reg[]}:
  \begin{itemize}
  \item \texttt{AddrD}
  \item \texttt{DataD}
  \item \texttt{Reg[rs1]}
  \item \texttt{Reg[rs2]}
  \end{itemize}
\item \textit{Imm. Gen}:
  \begin{itemize}
  \item \texttt{inst[31:20]}
  \item \texttt{ImmSel=I}
  \item \texttt{RegWEn=1}
  \item \texttt{BSel=1}
  \end{itemize}
\item \textit{ALU}:
  \begin{itemize}
  \item \texttt{ALUSel=Add}
  \item \texttt{alu}
  \end{itemize}
\end{itemize}
Adding \texttt{lw} to datapath
Adding \texttt{lw} to datapath

\begin{itemize}
  \item \texttt{IMEM}
  \item \texttt{ALU}
  \item \texttt{DMEM}
\end{itemize}

\begin{itemize}
  \item \texttt{Imm. Gen}
  \item \texttt{Reg[]}
  \item \texttt{wb}
\end{itemize}

\begin{itemize}
  \item \texttt{wb}
  \item \texttt{alu}
  \item \texttt{mem}
\end{itemize}

\begin{itemize}
  \item \texttt{addr}
  \item \texttt{inst[11:7]}
  \item \texttt{inst[19:15]}
  \item \texttt{inst[24:20]}
  \item \texttt{inst[31:20]}
\end{itemize}

\begin{itemize}
  \item \texttt{imm[31:0]}
\end{itemize}

\begin{itemize}
  \item \texttt{pool}
  \item \texttt{Reg[rs1]}
  \item \texttt{Reg[rs2]}
\end{itemize}

\begin{itemize}
  \item \texttt{Inst[31:0]}
  \item \texttt{ImmSel=I}
  \item \texttt{RegWEn=1}
  \item \texttt{BSel=1}
  \item \texttt{ALUSel=add}
  \item \texttt{MemRW=Read}
  \item \texttt{WBSel=0}
\end{itemize}
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>

writing back to register file.

The funct3 field encodes size and signedness of load data.
Implementing Store Word instruction

- RISC-V Assembly Instruction:

  \[
  \text{sw } x14, \ 8(\times2)
  \]

  
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs2</td>
<td>00010</td>
</tr>
<tr>
<td>rs1</td>
<td>01000</td>
</tr>
<tr>
<td>funct3</td>
<td>010011</td>
</tr>
<tr>
<td>imm[4:0]</td>
<td>00010</td>
</tr>
<tr>
<td>src</td>
<td>010</td>
</tr>
<tr>
<td>base</td>
<td>01000</td>
</tr>
<tr>
<td>width</td>
<td>01001</td>
</tr>
<tr>
<td>offset[4:0]</td>
<td>01000</td>
</tr>
<tr>
<td>opcode</td>
<td>0100011</td>
</tr>
</tbody>
</table>

  
  \[
  \begin{align*}
  \text{offset[11:5]} & = 0 \\
  \text{rs2} & = 14 \\
  \text{rs1} & = 2 \\
  \text{SW} & \\
  \text{offset[4:0]} & = 8
  \end{align*}
  \]

  combined 12-bit offset = 8
Adding `lw` to datapath
Adding \texttt{sw} to datapath

\begin{itemize}
  \item \texttt{pc} + 4
  \item IMEM
  \item \texttt{Inst}[11:7]
  \item \texttt{Inst}[19:15]
  \item \texttt{Inst}[24:20]
  \item \texttt{Inst}[31:7]
  \item \texttt{ImmGen}
  \item \texttt{ImmSel=S}
  \item \texttt{RegWEn=0}
  \item \texttt{Bsel=1}
  \item \texttt{ALUSel=Add}
  \item \texttt{MemRW=Write}
  \item \texttt{WBSel=*
  \item \texttt{DataW}
  \item \texttt{DataR}
\end{itemize}

\texttt{pc+4} \rightarrow \texttt{IMEM} \rightarrow \texttt{Reg[]} \rightarrow \texttt{ALU} \rightarrow \texttt{DMEM} \rightarrow \texttt{wb}

\texttt{Inst}[11:7] \rightarrow \texttt{AddrD} \rightarrow \texttt{DataD}

\texttt{Inst}[19:15] \rightarrow \texttt{AddrA} \rightarrow \texttt{DataA}

\texttt{Inst}[24:20] \rightarrow \texttt{AddrB} \rightarrow \texttt{DataB}

\texttt{Inst}[31:7] \rightarrow \texttt{ImmGen} \rightarrow \texttt{imm}[31:0]

\texttt{ImmSel=S} \rightarrow \texttt{RegWEn=0}

Bsel=1 \rightarrow \texttt{Reg[rs1]} \rightarrow \texttt{ALU} \rightarrow \texttt{DMEM} \rightarrow \texttt{wb}

\texttt{ALUSel=Add} \rightarrow \texttt{MemRW=Write}

\texttt{WBSel=*

*= “Don’t Care”}
Adding \texttt{sw} to datapath

\begin{itemize}
    \item \texttt{pc} + 4
    \item \texttt{IMEM}
    \item \texttt{Reg[]}
    \item \texttt{ALU}
    \item \texttt{DMEM}
    \item \texttt{wb}
\end{itemize}

\begin{tabular}{l}
\texttt{inst[11:7]} \quad \texttt{AddrD} \\
\texttt{inst[19:15]} \quad \texttt{AddrA} \quad \texttt{DataA} \\
\texttt{inst[24:20]} \quad \texttt{AddrB} \quad \texttt{DataB} \\
\texttt{inst[31:7]} \quad \texttt{imm[31:0]} \\
\texttt{BSel=1} \\
\texttt{ALUSel=Add} \\
\texttt{MemRW=Write} \\
\texttt{WBSel=*} 
\end{tabular}

\texttt{ALU} out: 
\begin{itemize}
    \item \texttt{ALU out} \\
    \item \texttt{Reg[rs1]} \\
    \item \texttt{Reg[rs2]} \\
\end{itemize}

\texttt{mem}:
\begin{itemize}
    \item \texttt{Addr} \\
    \item \texttt{DataR} \\
\end{itemize}

\texttt{wb}:
\begin{itemize}
    \item \texttt{0} \\
    \item \texttt{1} \\
\end{itemize}

\texttt{Inst[31:0]}:
\begin{itemize}
    \item \texttt{ImmSel=S RegWEn} \\
    \item \texttt{BSel=1 ALUSel=Add MemRW=Write} \\
\end{itemize}

\texttt{Inst[31:0]}:
\begin{itemize}
    \item \texttt{WBSel=*} \\
    \item \texttt{* = “Don’t Care”} \\
\end{itemize}
I-Format immediates

- 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])
I & S Immediate Generator

- 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

```plaintext
<table>
<thead>
<tr>
<th>IMEM</th>
<th>ALU</th>
<th>IMM. Gen</th>
<th>Reg[]</th>
<th>DMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddrD</td>
<td>DataD</td>
<td>AddrA DataA</td>
<td>AddrB DataB</td>
<td>Addr</td>
</tr>
<tr>
<td>pc</td>
<td>+4</td>
<td>pc+4</td>
<td>wb</td>
<td>pc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr</td>
<td>DataW</td>
<td>AddrR</td>
<td>AddrD</td>
<td>DataD</td>
</tr>
<tr>
<td>pc</td>
<td>alu</td>
<td>mem</td>
<td>wb</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>pc</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pc+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inst[31:7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>imm[31:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ALUSel=Add

PCSel=taken/not-taken inst[31:0] ImmSel=B RegWEn=0 BrUnBrEqBrLT Bsel=1 ASel=1 MemRW=Read WBSel=* |
```

CS 61c
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)
Implementing **JALR** Instruction (I-Format)

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

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

- Sets \text{PC} = \text{Reg[rs1]} + \text{immediate}
- Uses same immediates as arithmetic and loads
  - \text{no} multiplication by 2 bytes
Adding branches to datapath
Adding jalr to datapath
Adding jalr to datapath
Implementing `jal` Instruction

- JAL saves PC+4 in Reg[rd] (the return address)
- Set PC = PC + offset (PC-relative jump)
- Target somewhere within $\pm2^{19}$ locations, 2 bytes apart
  - $\pm2^{18}$ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost
Adding `jal` to datapath

```
IMEM

ALU

Imm.

Gen

+4

DMEM

Branch Comp.

Reg[]

AddrA

DataA

AddrB

DataB

DataD

Addr

DataW

DataR

1

0

pc

inst[11:7]

inst[19:15]

inst[24:20]

inst[31:7]

mm. Gen

imm[31:0]

PCSel

inst[31:0] ImmSel RegWEn BrUnBrEqBrLT BSelASel ALUSel MemRW WBSel

alu

pc+4

mem

alu

pc+4

wb

wb

wb

wb

wb
```

CS 61c
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