RISC-V Processor Design
Part 1: The Datapath
Outline:

1. Finish up CMOS circuits (nasty realities)
2. How to build a processor
Nasty Realities: Delays in CMOS circuits

More physically realistic model:

1. Transistors are not perfect switches
   A. They leak when off
   B. They have finite resistance when on

2. All circuit nodes have capacitance
   - To change their voltage level must displace charge

\[
\begin{align*}
\text{nFET} & \quad \text{gate} \\
\text{source} & \quad \text{drain} \\
Y & \quad R_{\text{on}} \quad Z \\
\text{When on, resistance between Y and Z. Likewise for pFET}
\end{align*}
\]

\[
\begin{align*}
V_{\text{dd}} & \quad \text{V}_{\text{in}} \quad \text{V}_{\text{out}} \quad C \\
\text{Represents the sum of all the capacitance at the output of the inverter and everything to which it connects: (drains, wires, transistor-gate capacitance of next gate(s))}
\end{align*}
\]
Transistors as water valves

If electrons are water molecules, transistor resistance like pipe diameters, and capacitors are buckets ...

A “on” p-FET fills up the capacitor with charge.

A “on” n-FET empties the bucket.

\[ \tau \propto R \cdot C \]
Consequences

- For every logic gate, delay from input change to output change
- The exact amount of the delay depends on:
  - type of gate, how many other gates it’s output connects to, IC process details
- For cascaded gates, delay accumulates
- Remember, flip-flops also have details and timing constraints: $\tau_{clk-to-q}$ and $\tau_{setup}$
Therefore, in General ...

\[ T \geq \tau_{\text{clk\rightarrow Q}} + \tau_{\text{CL}} + \tau_{\text{setup}} \]

For correct operation:

The worst case path is called the "critical path"

What can we do to reduce \( T \) (increase frequency)?
More nasty realities: CMOS circuits use electrical energy (consume power)

Energy is the ability to do work (joules).

Power is rate of expending energy (watts).

Energy Efficiency: energy per operation

- **Handheld and portable** (battery operated):
  - Energy Efficiency - limits battery life
  - Power - limited by heat

- **Infrastructure and servers** (connected to power grid):
  - Energy Efficiency - dictates operation cost
  - Power - heat removal contributes to TCO

\[
P = \frac{dE}{dt}
\]
Switching Energy: Fundamental Physics

Every logic transition dissipates energy.

\[ E_{0 \rightarrow 1} = \frac{1}{2} \cdot C \cdot V_{dd}^2 \quad E_{1 \rightarrow 0} = \frac{1}{2} \cdot C \cdot V_{dd}^2 \]
Chip-Level “switching” Power

\[ P = \frac{dE}{dt} \]

\[ P_{sw} = \frac{1}{2} \alpha C V_{dd}^2 F \]

"activity factor", average percentage of capacitance switching per cycle (~ number of nodes to switch)

Total chip capacitance to be switched

Clock Frequency
Reducing power consumption or improving energy efficiency

\[ P_{sw} = \frac{1}{2} \alpha C V_{dd}^2 F \]

• Power proportional to \( F \). Can reduce power by reducing frequency. But that doesn’t improve energy efficiency (just spreads computation over longer time)

• Energy efficiency:
  • \( E_{SW} \propto V_{dd}^2 \) but \( \tau_{logic} \propto V_{dd} \)
  • Therefore can improve energy efficiency by lowering supply voltage and making up for less performance by using parallelism
Great Idea #1: Abstraction

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

```
<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
</tr>
<tr>
<td>Assembly Language Program (e.g., RISC-V)</td>
</tr>
<tr>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program (RISC-V)</td>
</tr>
<tr>
<td>Machine Interpretation</td>
</tr>
<tr>
<td>Hardware Architecture Description (e.g., block diagrams)</td>
</tr>
<tr>
<td>Architecture Implementation</td>
</tr>
<tr>
<td>Logic Circuit Description (Circuit Schematic Diagrams)</td>
</tr>
<tr>
<td>Register File</td>
</tr>
<tr>
<td>ALU</td>
</tr>
</tbody>
</table>
```

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

Anything can be represented as a number, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

We are here!
Welcome To Marionette Time

• The next couple of lectures are going to describe how we build a complete processor
  • Consists of two pieces:
    The **Datapath** that implements the computation
    The **Control Logic** that looks at the instruction and tells the datapath what to do

• Think of it like a Marionette
  • The puppet is the datapath
  • The strings are pulled the control logic

• Today we will mostly focus on the puppet
Recap: Complete RV32I ISA

<table>
<thead>
<tr>
<th>imm[31:12]</th>
<th>rd</th>
<th>0101111</th>
</tr>
</thead>
<tbody>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>0101111</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>001</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs2</td>
<td>rs1</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>110</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>111</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>000</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>010</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>100</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>101</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>110</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>111</td>
</tr>
</tbody>
</table>

Not in CS61C
“State” Required by RV32I ISA

Each instruction reads and updates this state during execution:

- **Registers ($x0..x31$)**
  - Register file (or regfile) $\text{Reg}$ 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 $\text{Reg}[0]$ 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 ($\text{IMEM}$) and data ($\text{DMEM}$)
    - Later we’ll replace these with instruction and data caches
  - Instructions are read (fetched) from instruction memory (assume $\text{IMEM}$ read-only)
  - Load/store instructions access data memory
On every tick of the clock, the processor 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

- 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 \textbf{add}

\begin{itemize}
  \item \textbf{pc} + 4
  \item \textbf{IMEM}
  \item \textbf{alu}
  \item \textbf{RegWEn} (RegWriteEnable)
  \item \textbf{Control Logic}
\end{itemize}
Timing Diagram for add

Clock
- PC
- PC+4
- Inst[31:0]
- Reg[rs1]
- Reg[rs2]
- ALU
- Reg[1]

1000
1004
1008

1004
add x1, x2, x3
add x6, x7, x9

Reg[2]
Reg[3]
Reg[7]
Reg[9]
Implementing the `sub` instruction

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

<table>
<thead>
<tr>
<th></th>
<th>rs2</th>
<th>rs1</th>
<th>000</th>
<th>rd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0110011</td>
</tr>
<tr>
<td>0100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0110011</td>
</tr>
</tbody>
</table>
Datapath for \texttt{add/sub}

\begin{itemize}
  \item \texttt{pc+4}\hspace{1cm}IMEM\hspace{1cm}+4\hspace{1cm}IMEM\hspace{1cm}pc
  \item \texttt{inst[11:7]}\hspace{1cm}\texttt{inst[19:15]}\hspace{1cm}\texttt{inst[24:20]}
  \item \texttt{RegWEn} (1=write, 0=no write)
  \item \texttt{ALUSel} (Add=0/Sub=1)
\end{itemize}
Implementing other R-Format instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mode1</th>
<th>Mode2</th>
<th>Function</th>
<th>Destination</th>
<th>Immediate</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>ADD</td>
</tr>
<tr>
<td>SUB</td>
<td>rs2</td>
<td>rs1</td>
<td>000</td>
<td>rd</td>
<td>0110011</td>
<td>SUB</td>
</tr>
<tr>
<td>SLL</td>
<td>rs2</td>
<td>rs1</td>
<td>001</td>
<td>rd</td>
<td>0110011</td>
<td>SLL</td>
</tr>
<tr>
<td>SLT</td>
<td>rs2</td>
<td>rs1</td>
<td>010</td>
<td>rd</td>
<td>0110011</td>
<td>SLT</td>
</tr>
<tr>
<td>SLTU</td>
<td>rs2</td>
<td>rs1</td>
<td>011</td>
<td>rd</td>
<td>0110011</td>
<td>SLTU</td>
</tr>
<tr>
<td>XOR</td>
<td>rs2</td>
<td>rs1</td>
<td>100</td>
<td>rd</td>
<td>0110011</td>
<td>XOR</td>
</tr>
<tr>
<td>SRL</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRL</td>
</tr>
<tr>
<td>SRA</td>
<td>rs2</td>
<td>rs1</td>
<td>101</td>
<td>rd</td>
<td>0110011</td>
<td>SRA</td>
</tr>
<tr>
<td>OR</td>
<td>rs2</td>
<td>rs1</td>
<td>110</td>
<td>rd</td>
<td>0110011</td>
<td>OR</td>
</tr>
<tr>
<td>AND</td>
<td>rs2</td>
<td>rs1</td>
<td>111</td>
<td>rd</td>
<td>0110011</td>
<td>AND</td>
</tr>
</tbody>
</table>
Implementing the **addi** instruction

- **RISC-V Assembly Instruction:**
  
  \[
  \text{addi} \ x15,x1,-50
  \]

<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>
</tbody>
</table>

  \[
  11111110011110 \ 00001 \ 000 \ 01111 \ 0010011
  \]

  \[
  \text{imm=-50} \quad \text{rs1=1} \quad \text{ADD} \quad \text{rd=15} \quad \text{OP-Imm}
  \]
Datapath for add/sub

pc+4

IMEM

IMEM

inst[11:7]

inst[19:15]

inst[24:20]

Reg[]

DataD

AddrD

ALU

Reg[rs1]

Reg[rs2]

DataD

DataB

DataA

alu

inst[31:0]

RegWEn

(1=write, 0=no write)

Control Logic

ALUSel

(Add=0/Sub=1)
Adding \textbf{addi} to datapath

\begin{itemize}
  \item \textbf{pc} + 4
  \item \textbf{IMEM}
  \item \textbf{Inst[11:7]}
  \item \textbf{Inst[19:15]}
  \item \textbf{Inst[24:20]}
  \item \textbf{IMEM}
  \item \textbf{Inst[31:0]}
  \item \textbf{Reg[rs1]}
  \item \textbf{Reg[rs2]}
  \item \textbf{Imm. Gen}
  \item \textbf{Imm[31:0]}
  \item \textbf{alu}
  \item \textbf{ALUSel=Add}
  \item \textbf{RegWEn=1}
  \item \textbf{BSel=1}
  \item \textbf{Control Logic}
\end{itemize}
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, xor, slli, srli, srai) just by changing ALUSel
Implementing Load Word instruction

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

```

<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>12</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
```

| 0000000001000 | 00010 | 010 | 01110 | 0000011 |

imm=+8  \quad rs1=2  \quad LW  \quad rd=14  \quad LOAD
Adding \texttt{addi} to datapath

\begin{itemize}
\item \texttt{pc+4}
\item \texttt{IMEM}
\item \texttt{IMEM[11:7]}
\item \texttt{IMEM[19:15]}
\item \texttt{IMEM[24:20]}
\item \texttt{IMEM[31:0]}
\item \texttt{Reg[rs1]}
\item \texttt{Reg[rs2]}
\item \texttt{Imm. Gen}
\item \texttt{ImmSel=I}
\item \texttt{RegWEn=1}
\item \texttt{BSel=1}
\item \texttt{ALUSel=Add}
\end{itemize}

\textbf{Control Logic}
Adding \texttt{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:

```
sw x14, 8(x2)
```

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs2</td>
<td>01110</td>
</tr>
<tr>
<td>rs1</td>
<td>00010</td>
</tr>
<tr>
<td>funct3</td>
<td>010</td>
</tr>
<tr>
<td>imm[4:0]</td>
<td>01000</td>
</tr>
<tr>
<td>opcode</td>
<td>0100011</td>
</tr>
</tbody>
</table>

```
offset[11:5] = 0
rs2 = 14
rs1 = 2
SW
offset[4:0] = 8
STORE
```

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

- \texttt{IMEM}
- \texttt{ALU}
- \texttt{Imm. Gen}
- \texttt{Reg[]}
- \texttt{DMEM}

\begin{itemize}
  \item \texttt{inst[11:7]}
  \item \texttt{inst[19:15]}
  \item \texttt{inst[24:20]}
  \item \texttt{inst[31:7]}
  \item \texttt{inst[31:0]}
\end{itemize}

\texttt{ImmSel=S} \quad \texttt{RegWEn=0} \quad \texttt{Bsel=1} \quad \texttt{ALUSel=Add} \quad \texttt{MemRW=Write} \quad \texttt{WBSel=\text{"Don't Care"}
Adding \texttt{sw} to datapath

\begin{itemize}
  \item \texttt{PC} + 4
  \item \texttt{IMEM}
  \item \texttt{Imm. Gen}
  \item \texttt{Reg[\ldots]}
  \item \texttt{DMEM}
  \item \texttt{ALU}
  \item \texttt{wb}
\end{itemize}

\begin{itemize}
  \item \texttt{inst[11:7]}
  \item \texttt{inst[19:15]}
  \item \texttt{inst[24:20]}
  \item \texttt{inst[31:7]}
  \item \texttt{imm[31:0]}
\end{itemize}

\begin{itemize}
  \item \texttt{Reg[rs1]}
  \item \texttt{Reg[rs2]}
  \item \texttt{ALUSel=Add}
  \item \texttt{MemRW=Write}
  \item \texttt{WBSel=\ast}
\end{itemize}

\texttt{* = “Don’t Care”}
### I-Format immediates

<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>
</tbody>
</table>

- **ImmSel=I**
  - 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])
I & S Immediate Generator

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>I-opcode</th>
</tr>
</thead>
</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
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

**Diagram:***

- **PC**: pc+4
- **IMEM**: inst[31:7] to IMM Gen
- **Imm Gen**: imm[31:0]
- **Reg**: DataD
- **ALU**: Reg[rs1], Reg[rs2], imm[31:0], ALUSel, MemRW, MemRW
- **DMEM**: Addr, DataR, DataW, wb
- **WB**: 0, 1

**Signals and Connectors:**
- ImmSel, RegWEn, Bsel, ALUSel, MemRW, WBSel
Adding branches to datapath

Diagram showing the computer architecture with added branch components.
Adding branches to datapath

PCSel=taken/not-taken  inst[31:0]  ImmSel=B  RegWEn=0  BrUn  BrLT  Bsel=1  ASel=1  MemRW=Read  WBSel=*  ALUSel=Add

IMEM

ALU

DMEM

PC+4

1

0

alu

wb

alu

wb

alu

wb
Branch Comparator

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

- BGE branch: $A \geq B$, if $\neg (A < B)$
Implementing JALR Instruction (I-Format)

- 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

<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>
Adding branches to datapath
Adding jalr to datapath
Adding jalr to datapath

- **IMEM**: Instruction memory
- **ALU**: Arithmetic Logic Unit
- **DMEM**: Data memory
- **Branch Comp**: Branch comparison
- **Reg**: Register file
- **Addr**: Address
- **Data**: Data
- **WB**: Write back
- **alu**: ALU output
- **mem**: Memory output
- **wb**: Write back output

- **IMEM** output (inst[24:20]) to **Reg**
- **IMEM** output (inst[11:7]) to **Reg**
- **IMEM** output (inst[19:15]) to **Reg**
- **IMEM** output (inst[31:0]) to **Imm. Gen**
- **Imm. Gen** output (imm[31:0]) to **Reg**
- **IMEM** output (inst[31:7]) to **Imm. Gen**
- **Imm. Gen** output (imm[31:0]) to **Branch Comp**
- **Branch Comp** output (BrUn=*, BrEq=*, BrLT=*) to **Reg**
- **Branch Comp** output (Bsel=1 Asel=0 MemRW=Read ALUSel=Add) to **Reg**
- **Reg** output to **alu**
- **alu** output to **PCSel**
- **IMEM** output (inst[31:0]) to **PCSel**
- **PCSel** output to **alu**
- **alu** output to **DMEM**
- **DMEM** output (DataR) to **alu**
- **alu** output (pc+4) to **IMEM**
- **IMEM** output (inst[31:0]) to **Imm. Gen**
- **Imm. Gen** output (imm[31:0]) to **Branch Comp**
- **Branch Comp** output (Bsel=1 Asel=0 MemRW=Read ALUSel=Add) to **Reg**
- **Reg** output to **alu**
- **alu** output (pc+4) to **IMEM**
Implementing jal Instruction

- 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
Single-Cycle RISC-V RV32I Datapath
And in Conclusion, ...

• Universal datapath
  • Capable of executing all RISC-V instructions in one cycle each
  • datapath is the “union” of all the units used by all the instructions. Muxes provide the options.
  • 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