# ECE 4750 Computer Architecture, Fall 2022

## Topic 2: Fundamental Processor Microarchitecture

School of Electrical and Computer Engineering  
Cornell University

revision: 2022-08-31-08-55

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Processor Microarchitectural Design Patterns</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Transactions and Steps</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Microarchitecture: Control/Datapath Split</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>TinyRV1 Single-Cycle Processor</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>High-Level Idea for Single-Cycle Processors</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Single-Cycle Processor Datapath</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Single-Cycle Processor Control Unit</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Analyzing Performance</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>TinyRV1 FSM Processor</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>High-Level Idea for FSM Processors</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>FSM Processor Datapath</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>FSM Processor Control Unit</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Analyzing Performance</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>TinyRV1 Pipelined Processor</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>High-Level Idea for Pipelined Processors</td>
<td>31</td>
</tr>
</tbody>
</table>
4.2. Pipelined Processor Datapath and Control Unit

5 Pipeline Hazards: RAW Data Hazards
5.1. Expose in Instruction Set Architecture
5.2. Hardware Stalling
5.3. Hardware Bypassing/Forwarding
5.4. RAW Data Hazards Through Memory

6 Pipeline Hazards: Control Hazards
6.1. Expose in Instruction Set Architecture
6.2. Hardware Speculation
6.3. Interrupts and Exceptions

7 Pipeline Hazards: Structural Hazards
7.1. Expose in Instruction Set Architecture
7.2. Hardware Stalling
7.3. Hardware Duplication

8 Pipeline Hazards: WAW and WAR Name Hazards
8.1. Software Renaming
8.2. Hardware Stalling

9 Summary of Processor Performance

10 Case Study: Transition from CISC to RISC
10.1. Example CISC: IBM 360/M30
10.2. Example RISC: MIPS R2K

Topic 2: Fundamental Processor Microarchitecture
1. Processor Microarchitectural Design Patterns

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle Processor</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>FSM Processor</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Pipelined Processor</td>
<td>≈1</td>
<td>short</td>
</tr>
</tbody>
</table>

1.1. Transactions and Steps

- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>addi</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Memory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Write Memory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Write Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
1.2. Microarchitecture: Control/Datapath Split

The diagram illustrates the separation of control signals and status signals from the datapath. The control unit is connected to the datapath through control signals (downward arrows) and status signals (upward arrows). The datapath includes units for accessing memory (imem and dmem), with request (req) and response (resp) signals. The memory units are connected to the datapath with request (req) and response (resp) signals as well.
2. **TinyRV1 Single-Cycle Processor**

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle Processor</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>FSM Processor</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Pipelined Processor</td>
<td>≈1</td>
<td>short</td>
</tr>
</tbody>
</table>

**Technology Constraints**

- Assume technology where logic is not too expensive, so we do not need to overly minimize the number of registers and combinational logic
- Assume multi-ported register file with a reasonable number of ports is feasible
- Assume a dual-ported combinational memory

![Diagram of control unit and datapath](image-url)
## 2.1. High-Level Idea for Single-Cycle Processors

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>addi</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Memory</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Memory</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Diagram

- **Fetch Instruction**
- **Decode Instruction**
- **Read Registers**
- **Register Arithmetic**
- **Read Memory**
- **Write Memory**
- **Write Registers**
- **Update PC**
2.2. Single-Cycle Processor Datapath

**ADD**

\[
\text{add } rd, rs1, rs2 \\
R[rd] \leftarrow R[rs1] + R[rs2] \\
PC \leftarrow PC + 4
\]

**ADDI**

\[
\text{addi } rd, rs1, \text{imm} \\
R[rd] \leftarrow R[rs1] + \text{sext(imm)} \\
PC \leftarrow PC + 4
\]
Implementing ADD and ADDI Instructions

MUL
mul rd, rs1, rs2
R[rd] ← R[rs1] × R[rs2]
PC ← PC + 4
LW
lw rd, imm(rs1)
R[rd] ← M[R[rs1] + sext(imm)]
PC ← PC + 4

SW
sw rs2, imm(rs1)
M[R[rs1] + sext(imm)] ← R[rs2]
PC ← PC + 4
### JAL

**jal rd, imm**

\[ R[rd] \leftarrow PC + 4 \]

\[ PC \leftarrow PC + \text{sext}(imm) \]

**Example:**

- **Imm Type:** imm = \{ inst[31], inst[19:12], inst[20], inst[30:21], 0 \}

### JR

**jr rs1**

\[ PC \leftarrow R[rs1] \]

**Example:**

- **Imm Type:** rs1 = \{ 00000, 00000000000000000000000000000000, 1101111 \}
2. TinyRV1 Single-Cycle Processor

2.2. Single-Cycle Processor Datapath

### BNE

\[ \text{bne } rs1, rs2, \text{ imm} \]

\[
\text{if } (R[rs1] \neq R[rs2]) \quad PC \leftarrow PC + \text{sext}(\text{imm})
\]

\[
\text{else} \quad PC \leftarrow PC + 4
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
\text{imm} & \text{rs2} & \text{rs1} & 001 & \text{imm} & 1100011 \\
\hline
\end{array}
\]

\[
\text{imm} = \{ \text{inst}[31], \text{inst}[7], \\
\text{inst}[30:25], \text{inst}[11:8], 0 \}
\]
Adding a New Auto-Incrementing Load Instruction

Draw on the datapath diagram what paths we need to use as well as any new paths we will need to add in order to implement the following auto-incrementing load instruction.

LW.AI

\[
\text{lw.ai } \text{rd, imm(rsl)} \\
R[\text{rd}] \leftarrow M[ R[rsl] + \text{sext(imm)} ] \\
R[rsl] \leftarrow R[rsl] + 4 \\
\text{PC } \leftarrow \text{PC } + \text{4}
\]

![Datapath Diagram](image-url)
### 2.3. Single-Cycle Processor Control Unit

<table>
<thead>
<tr>
<th>inst</th>
<th>pc sel</th>
<th>imm type</th>
<th>op1 sel</th>
<th>alu func</th>
<th>wb sel</th>
<th>rf</th>
<th>wen</th>
<th>imem req val</th>
<th>dmem req val</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>pc+4</td>
<td>–</td>
<td>rf</td>
<td>+</td>
<td>alu</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td>pc+4</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>mul</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>pc+4</td>
<td>i</td>
<td>imm</td>
<td>+</td>
<td>mem</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td>jr</td>
<td>i</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Need to factor eq status signal into pc_sel signal for BNE!

### 2.4. Analyzing Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}
\]

- Instructions / program depends on source code, compiler, ISA
- Cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation
Estimating cycle time

There are many paths through the design that start at a state element and end at a state element. The “critical path” is the longest path across all of these paths. We can usually use a simple first-order static timing estimate to estimate the cycle time (i.e., the clock period and thus also the clock frequency).

- register read = 1τ
- register write = 1τ
- regfile read = 10τ
- regfile write = 10τ
- memory read = 20τ
- memory write = 20τ
- +4 unit = 4τ
- immgen = 2τ
- mux = 3τ
- multiplier = 20τ
- alu = 10τ
- adder = 8τ
Estimating execution time

Using our first-order equation for processor performance, how long in nanoseconds will it take to execute the vector-vector add example assuming \( n \) is 64?

```
loop:
    lw   x5, 0(x13)
    lw   x6, 0(x14)
    add  x7, x5, x6
    sw   x7, 0(x12)
    addi x13, x12, 4
    addi x14, x14, 4
    addi x12, x12, 4
    addi x15, x15, -1
    bne  x15, x0, loop
jr    x1
```

Using our first-order equation for processor performance, how long in nanoseconds will it take to execute the mystery program assuming \( n \) is 64 and that we find a match on the last element.

```
addi x5, x0, 0
loop:
    lw   x6, 0(x12)
    bne  x6, x14, foo
    addi x10, x5, 0
jr    x1
foo:
    addi x12, x12, 4
    addi x5, x5, 1
    bne  x5, x13, loop
    addi x10, x0, -1
    jr    x1
```
3. **TinyRV1 FSM Processor**

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle Processor</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>FSM Processor</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Pipelined Processor</td>
<td>≈1</td>
<td>short</td>
</tr>
</tbody>
</table>

**Technology Constraints**

- Assume legacy technology where logic is expensive, so we want to minimize the number of registers and combinational logic
- Assume an (unrealistic) combinational memory
- Assume multi-ported register files and memories are too expensive, these structures can only have a single read/write port
3.1. High-Level Idea for FSM Processors

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>addi</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Memory</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Implementing an FSM datapath requires thinking about the required FSM states, but we will defer discussion of how to implement the control logic to the next section.

3.2. FSM Processor Datapath

Implementing an FSM datapath requires thinking about the required FSM states, but we will defer discussion of how to implement the control logic to the next section.
Implementing Fetch Sequence

(pseudo-control-signal syntax)
Implementing ADD Instruction

(pseudo-control-signal syntax)
add rd, rs1, rs2
Full Datapath for TinyRV1 FSM Processor

ADDI Pseudo-Control-Signal Fragment

addi rd, rs1, imm
### MUL Instruction

\[ \text{mul } rd, \text{ rs1, rs2 } \]

- **M0:** \( A \leftarrow RF[x0] \)
- **M1:** \( B \leftarrow RF[\text{rs1}] \)
- **M2:** \( C \leftarrow RF[\text{rs2}] \)
- **M3:** \( A \leftarrow A + ? B; \)  
  \( B \leftarrow B << 1; C \leftarrow C >> 1 \)
- **M4:** \( A \leftarrow A + ? B; \)  
  \( B \leftarrow B << 1; C \leftarrow C >> 1 \)

...  
- **M35:** \( RF[rd] \leftarrow A + ? B; \) goto F0

### LW Instruction

\[ \text{lwr, imm(rs1)} \]

- **L0:** \( A \leftarrow RF[\text{rs1}] \)
- **L1:** \( B \leftarrow \text{sext}(\text{imm}_i) \)
- **L2:** \( \text{memreq.addr} \leftarrow A + B \)
- **L3:** \( RF[rd] \leftarrow \text{RD}; \) goto F0

### SW Instruction

\[ \text{sw rs2, imm(rs1)} \]

- **S0:** \( \text{WD} \leftarrow RF[\text{rs2}] \)
- **S1:** \( A \leftarrow RF[\text{rs1}] \)
- **S2:** \( B \leftarrow \text{sext}(\text{imm}_s) \)
- **S3:** \( \text{memreq.addr} \leftarrow A + B; \) goto F0

### JAL Instruction

\[ \text{jal rd, imm} \]

- **JA0:** \( RF[rd] \leftarrow PC \)
- **JA1:** \( B \leftarrow \text{sext}(\text{imm}_j) \)
- **JA2:** \( PC \leftarrow A + B; \) goto F0

### JR Instruction

\[ \text{jr rs1} \]

- **JR0:** \( PC \leftarrow RF[\text{rs1}]; \) goto F0

### BNE Instruction

\[ \text{bne rs1, rs2, imm} \]

- **B0:** \( A \leftarrow RF[\text{rs1}] \)
- **B1:** \( B \leftarrow RF[\text{rs2}] \)
- **B2:** \( B \leftarrow \text{sext}(\text{imm}_b); \)
  \( \text{if } A == B \) goto F0
- **B3:** \( A \leftarrow PC \)
- **B4:** \( A \leftarrow A - 4 \)
- **B5:** \( PC \leftarrow A + B; \) goto F0
Adding a Complex Instruction

FSM processors simplify adding complex instructions. New instructions usually do not require datapath modifications, only additional states.

\[
\text{add.mm } rd, \ rs1, \ rs2 \\
\]
Adding a New Auto-Incrementing Load Instruction

Implement the following auto-incrementing load instruction using pseudo-control-signal syntax. Modify the datapath if necessary.

```
lw.ai rd, imm(rs1)
```

\[ R[rd] \leftarrow M[ R[rs1] + \text{sext}(\text{imm}_i) ]; R[rs1] \leftarrow R[rs1] + 4 \]
We will study three techniques for implementing FSM control units:

- **Hardwired control units** are high-performance, but inflexible
- **Horizontal µcoding** increases flexibility, requires large control store
- **Vertical µcoding** is an intermediate design point

**Hardwired FSM**

```
Topic 2: Fundamental Processor Microarchitecture
```
3. TinyRV1 FSM Processor

3.3. FSM Processor Control Unit

Control signal output table for hardwired control unit

<table>
<thead>
<tr>
<th>alu func</th>
<th>imm gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4: A + 4</td>
<td>i: sext(IR[31:20])</td>
</tr>
<tr>
<td>+: A + B</td>
<td>s: sext(IR[31:25],IR[11:7])</td>
</tr>
<tr>
<td>+?: A +? B</td>
<td>b: sext(IR[31],IR[7],IR[30:25],IR[11:8],0)</td>
</tr>
<tr>
<td>cmp: A == B</td>
<td>j: sext(IR[31],IR[19:12],IR[20],IR[30:21],0)</td>
</tr>
<tr>
<td>−4: A − 4</td>
<td></td>
</tr>
</tbody>
</table>

F0: memreq.addr ← PC; A ← PC
F1: IR ← RD
F2: PC ← A + 4; goto inst

A0: A ← RF[rs1]
A1: B ← RF[rs2]
A2: RF[rd] ← A + B; goto F0

<table>
<thead>
<tr>
<th>state</th>
<th>Bus Enables</th>
<th>Register Enables</th>
<th>Mux</th>
<th>Func</th>
<th>RF</th>
<th>MReq</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pc</td>
<td>ig</td>
<td>alu</td>
<td>rf</td>
<td>rd</td>
<td>pc</td>
</tr>
<tr>
<td>F0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vertically Microcoded FSM

- Use memory array (called the control store) instead of random logic to encode both the control signal logic and the state transition logic
- Enables a more systematic approach to implementing complex multi-cycle instructions
- Microcoding can produce good performance if accessing the control store is much faster than accessing main memory
- Read-only control stores might be replaceable enabling in-field updates, while read-write control stores can simplify diagnostics and microcode patches
### Control signal store for microcoded control unit

#### Datapath Bus

- **PC**
- **IR**
- **A**
- **B**
- **C**

#### To control unit

- **alu**
- **imm gen**
- **RF**

#### B0: A ← RF[rs1]
B1: B ← RF[rs2]
B2: B ← sext(imm_b); if A == B goto F0
B3: A ← PC
B4: A ← A − 4
B5: PC ← A + B; goto F0

#### Table: Bus Enables

<table>
<thead>
<tr>
<th>state</th>
<th>Bus Enables</th>
<th>Register Enables</th>
<th>Mux</th>
<th>Func</th>
<th>RF</th>
<th>MReq</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pc ig alu rf rd</td>
<td>pc ir a b c</td>
<td>b c</td>
<td>ig alu sel wen val op</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>B0</td>
<td>0 0 0 1 0 0 1 0 0 0</td>
<td>- - - -</td>
<td>rs1</td>
<td>0 0 -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>0 0 0 1 0 0 0 1 0 0</td>
<td>b - - -</td>
<td>rs2</td>
<td>0 0 -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>0 1 0 0 0 0 0 0 1 0 0</td>
<td>b -</td>
<td>cmp -</td>
<td>0 0 -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>1 0 0 0 0 0 0 1 0 0 0</td>
<td>- - - -</td>
<td>0 0 -</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>0 0 1 0 0 0 1 0 0 0 0</td>
<td>- - - -</td>
<td>-4 0 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>0 0 1 0 0 1 0 0 0 0 0</td>
<td>- - - +</td>
<td>0 0 -</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Topic 2: Fundamental Processor Microarchitecture**
3.4. Analyzing Performance

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}
\]

Estimating cycle time

- register read/write $= 1\tau$
- regfile read/write $= 10\tau$
- mem read/write $= 20\tau$
- immgen $= 2\tau$
- mux $= 3\tau$
- alu $= 10\tau$
- 1b shifter $= 1\tau$
- tri-state buf $= 1\tau$
Estimating execution time

Using our first-order equation for processor performance, how long in units of $\tau$ will it take to execute the vector-vector add example assuming $n$ is 64?

```
loop:
    lw x5, 0(x13)
    lw x6, 0(x14)
    add x7, x5, x6
    sw x7, 0(x12)
    addi x13, x12, 4
    addi x14, x14, 4
    addi x12, x12, 4
    addi x15, x15, -1
    bne x15, x0, loop
    jr x1
```

Using our first-order equation for processor performance, how long in units of $\tau$ will it take to execute the mystery program assuming $n$ is 64 and that we find a match on the last element.

```
addi x5, x0, 0
loop:
    lw x6, 0(x12)
    bne x6, x14, foo
    addi x10, x5, 0
    jr x1
foo:
    addi x12, x12, 4
    addi x5, x5, 1
    bne x5, x13, loop
    addi x10, x0, -1
    jr x1
```
4. TinyRV1 Pipelined Processor

\[
\text{Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Avg Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- Instructions / program depends on source code, compiler, ISA
- Avg cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle Processor</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>FSM Processor</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Pipelined Processor</td>
<td>≈1</td>
<td>short</td>
</tr>
</tbody>
</table>

Technology Constraints

- Assume modern technology where logic is cheap and fast (e.g., fast integer ALU)
- Assume multi-ported register files with a reasonable number of ports are feasible
- Assume small amount of very fast memory (caches) backed by large, slower memory
4.1. High-Level Idea for Pipelined Processors

- Anne, Brian, Cathy, and Dave each have one load of clothes
- Washing, drying, folding, and storing each take 30 minutes

### Fixed Time-Slot Laundry

<table>
<thead>
<tr>
<th>Time</th>
<th>Anne's Load</th>
<th>Ben's Load</th>
<th>Cathy's Load</th>
<th>Dave's Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12am</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1am</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2am</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3am</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pipelined Laundry

<table>
<thead>
<tr>
<th>Time</th>
<th>Anne's Load</th>
<th>Ben's Load</th>
<th>Cathy's Load</th>
<th>Dave's Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pipelined Laundry with Slow Dryers

<table>
<thead>
<tr>
<th>Time</th>
<th>Anne's Load</th>
<th>Ben's Load</th>
<th>Cathy's Load</th>
<th>Dave's Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10pm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pipelining lessons

- Multiple transactions operate simultaneously using different resources
- Pipelining does not help the transaction latency
- Pipelining does help the transaction throughput
- Potential speedup is proportional to the number of pipeline stages
- Potential speedup is limited by the slowest pipeline stage
- Potential speedup is reduced by time to fill the pipeline
Applying pipelining to processors

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>addi</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Read Memory</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Memory</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Registers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
4.2. Pipelined Processor Datapath and Control Unit

- Incrementally develop an unpipelined datapath
- Keep data flowing from left to right
- Position control signal table early in the diagram
- Divide datapath/control into stages by inserting pipeline registers
- Keep the pipeline stages roughly balanced
- Forward arrows should avoid “skipping” pipeline registers
- Backward arrows will need careful consideration

\[ \begin{align*}
\text{addi} & \ x_1, \ x_2, \ 1 \\
& \ F \rightarrow D \rightarrow X \rightarrow M \rightarrow W \\
\text{addi} & \ x_3, \ x_4, \ 1 \\
& \ F \rightarrow D \rightarrow X \rightarrow M \rightarrow W \\
\text{addi} & \ x_5, \ x_6, \ 1 \\
& \ F \rightarrow D \rightarrow X \rightarrow M \rightarrow W
\end{align*} \]
Adding a new auto-incrementing load instruction

Draw on the above datapath diagram what paths we need to use as well as any new paths we will need to add in order to implement the following auto-incrementing load instruction.

\[ \text{lw.ai} \ rd, \ \text{imm}(rs1) \]

\[ R[rd] \leftarrow M[R[rs1] + \text{sext}(\text{imm})]; R[rs1] \leftarrow R[rs1] + 4 \]
Pipeline diagrams

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi x1, x2, 1</td>
</tr>
<tr>
<td>addi x3, x4, 1</td>
</tr>
<tr>
<td>addi x5, x6, 1</td>
</tr>
</tbody>
</table>

What would be the total execution time if these three instructions were repeated 10 times?

Hazards occur when instructions interact with each other in pipeline

- **RAW Data Hazards**: An instruction depends on a data value produced by an earlier instruction
- **Control Hazards**: Whether or not an instruction should be executed depends on a control decision made by an earlier instruction
- **Structural Hazards**: An instruction in the pipeline needs a resource being used by another instruction in the pipeline
- **WAW and WAR Name Hazards**: An instruction in the pipeline is writing a register that an earlier instruction in the pipeline is either writing or reading

Stalling and squashing instructions

- **Stalling**: An instruction *originates* a stall due to a hazard, causing all instructions earlier in the pipeline to also stall. When the hazard is resolved, the instruction no longer needs to stall and the pipeline starts flowing again.

- **Squashing**: An instruction *originates* a squash due to a hazard, and squashes all previous instructions in the pipeline (but not itself). We restart the pipeline to begin executing a new instruction sequence.
Control logic with no stalling and no squashing

always_ff @(posedge clk)
if ( reset )
val_B <= 0
else
val_B <= next_val_A
next_val_B = val_B

Control logic with stalling and no squashing

reg_en_B = !stall_B

always_ff @(posedge clk)
if ( reset )
val_B <= 0
else if ( reg_en_B )
val_B <= next_val_A

ostall_B = val_B && (ostall_hazard1_B || ostall_hazard2_B)

stall_B = val_B && (ostall_B || ostall_C || ...)

next_val_B = val_B && !stall_B

<table>
<thead>
<tr>
<th>ostall_B</th>
<th>Originating stall due to hazards detected in B stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>stall_B</td>
<td>Should we actually stall B stage? Factors in ostalls due to hazards and ostalls from later pipeline stages.</td>
</tr>
<tr>
<td>next_val_B</td>
<td>Only send transaction to next stage if transaction in B stage is valid and we are not stalling B stage.</td>
</tr>
</tbody>
</table>

Topic 2: Fundamental Processor Microarchitecture 36
Control logic with stalling and squashing

\[ \text{reg}_{-}\text{en}_B = \neg \text{stall}_B || \text{squash}_B \]

\[
\text{always}\_ff \@ (\text{posedge } \text{clk}) \\
\quad \text{if (reset)} \\
\quad \quad \text{val}_B \leftarrow 0 \\
\quad \text{else if (reg}_{-}\text{en}_B) \\
\quad \quad \text{val}_B \leftarrow \text{next}_{-}\text{val}_A
\]

\[
\text{ostall}_B = \text{val}_B && (\text{ostall}_{-}\text{hazard}_1_B || \text{ostall}_{-}\text{hazard}_2_B)
\]

\[
\text{stall}_B = \text{val}_B && (\text{ostall}_B || \text{ostall}_C || \ldots)
\]

\[
\text{osquash}_B = \text{val}_B && \neg \text{stall}_B && (\text{osquash}_{-}\text{hazard}_1_B || \ldots)
\]

\[
\text{squash}_B = \text{val}_B && (\text{osquash}_C || \ldots)
\]

\[
\text{next}_{-}\text{val}_B = \text{val}_B && \neg \text{stall}_B && \neg \text{squash}_B
\]

<table>
<thead>
<tr>
<th>(\text{ostall}_B)</th>
<th>Originating stall due to hazards detected in B stage.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{stall}_B)</td>
<td>Should we actually stall B stage? Factors in ostalls due to hazards and ostalls from later pipeline stages.</td>
</tr>
<tr>
<td>(\text{osquash}_B)</td>
<td>Originating squash due to hazards detected in B stage. If this stage is stalling, do not originate a squash.</td>
</tr>
<tr>
<td>(\text{squash}_B)</td>
<td>Should we squash B stage? Factors in the originating squashes from later pipeline stages. An originating squash from B stage means to squash all stages earlier than B, so \text{osquash}_B is not factored into \text{squash}_B.</td>
</tr>
<tr>
<td>(\text{next}_{-}\text{val}_B)</td>
<td>Only send transaction to next stage if transaction in B stage is valid and we are not stalling or squashing B stage.</td>
</tr>
</tbody>
</table>
5. Pipeline Hazards: RAW Data Hazards

RAW data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline. We use architectural dependency arrows to illustrate RAW dependencies in assembly code sequences.

\[
\text{addi } x_1, x_2, 1
\]

\[
\text{addi } x_3, x_1, 1
\]

\[
\text{addi } x_4, x_3, 1
\]

Using pipeline diagrams to illustrate RAW hazards

We use microarchitectural dependency arrows to illustrate RAW hazards on pipeline diagrams.

\[
\text{addi } x_1, x_2, 1
\]

\[
\text{addi } x_3, x_1, 1
\]

\[
\text{addi } x_4, x_3, 1
\]
Approaches to resolving data hazards

- **Expose in Instruction Set Architecture:** Expose data hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e., software scheduling for correctness)

- **Hardware Scheduling:** Hardware dynamically schedules instructions to avoid RAW hazards, potentially allowing instructions to execute out of order

- **Hardware Stalling:** Hardware includes control logic that freezes later instructions until earlier instruction has finished producing data value; software scheduling can still be used to avoid stalling (i.e., software scheduling for performance)

- **Hardware Bypassing/Forwarding:** Hardware allows values to be sent from an earlier instruction to a later instruction before the earlier instruction has left the pipeline (sometimes called forwarding)

- **Hardware Speculation:** Hardware guesses that there is no hazard and allows later instructions to potentially read invalid data; detects when there is a problem, squashes and then re-executes instructions that operated on invalid data
5. Pipeline Hazards: RAW Data Hazards

5.1. Expose in Instruction Set Architecture

Insert nops to delay read of earlier write. These nops count as real instructions increasing instructions per program.

```
addi x1, x2, 1
nop
nop
nop
addi x3, x1, 1
nop
nop
nop
addi x4, x3, 1
```

Insert independent instructions to delay read of earlier write, and only use nops if there is not enough useful work.

```
addi x1, x2, 1
addi x6, x7, 1
addi x8, x9, 1
nop
addi x3, x1, 1
nop
nop
nop
addi x4, x3, 1
```

Pipeline diagram showing exposing RAW data hazards in the ISA

<table>
<thead>
<tr>
<th>addi x1, x2, 1</th>
<th>addi x6, x7, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi x8, x9, 1</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>addi x3, x1, 1</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>addi x4, x3, 1</td>
<td></td>
</tr>
</tbody>
</table>

Note: If hazard is exposed in ISA, software scheduling is required for **correctness**! A scheduling mistake can cause undefined behavior.
5.2. Hardware Stalling

Hardware includes control logic that freezes later instructions (in front of pipeline) until earlier instruction (in back of pipeline) has finished producing data value.

Pipeline diagram showing hardware stalling for RAW data hazards

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi x1, x2, 1</td>
<td></td>
</tr>
<tr>
<td>addi x3, x1, 1</td>
<td></td>
</tr>
<tr>
<td>addi x4, x3, 1</td>
<td></td>
</tr>
</tbody>
</table>
```

Note: Software scheduling is not required for correctness, but can improve performance! Programmer or compiler schedules independent instructions to reduce the number of cycles spent stalling.

Modifications to datapath/control to support hardware stalling
5. Pipeline Hazards: RAW Data Hazards

5.3. Hardware Bypassing/Forwarding

Deriving the stall signal

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>addi</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1_en</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs2_en</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rf_wen</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ostall_waddr_X_rs1_D =
val_D && rs1_en_D && val_X && rf_wen_X
&& (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)

ostall_waddr_M_rs1_D =
val_D && rs1_en_D && val_M && rf_wen_M
&& (inst_rs1_D == rf_waddr_M) && (rf_waddr_M != 0)

ostall_waddr_W_rs1_D =
val_D && rs1_en_D && val_W && rf_wen_W
&& (inst_rs1_D == rf_waddr_W) && (rf_waddr_W != 0)

... similar for ostall signals for rs2 source register ...

ostall_D = val_D
&& ( ostall_waddr_X_rs1_D || ostall_waddr_X_rs2_D
|| ostall_waddr_M_rs1_D || ostall_waddr_M_rs2_D
|| ostall_waddr_W_rs1_D || ostall_waddr_W_rs2_D )

5.3. Hardware Bypassing/Forwarding

Hardware allows values to be sent from an earlier instruction (in back of pipeline) to a later instruction (in front of pipeline) before the earlier instruction has left the pipeline. Sometimes called “forwarding”. 

Topic 2: Fundamental Processor Microarchitecture
Pipeline Hazards: RAW Data Hazards

### 5.3. Hardware Bypassing/Forwarding

**Pipeline diagram showing hardware bypassing for RAW data hazards**

\[
\begin{align*}
\text{addi } x1, x2, 1 \\
\text{addi } x3, x1, 1 \\
\text{addi } x4, x3, 1
\end{align*}
\]

**Adding single bypass path to support limited hardware bypassing**

![Control Signal Table](image)

**Deriving the bypass and stall signals**

\[
\begin{align*}
\text{ostall_waddr}_X_{-rs1_D} &= 0 \\
\text{bypass_waddr}_X_{-rs1_D} &= \\
&\quad \text{val}_D \&\& \text{rs1_en}_D \&\& \text{val}_X \&\& \text{rf}_\text{wen}_X \\
&\quad \&\& (\text{inst}_rs1_D == \text{rf}_\text{waddr}_X) \&\& (\text{rf}_\text{waddr}_X != 0)
\end{align*}
\]
5. Pipeline Hazards: RAW Data Hazards

5.3. Hardware Bypassing/Forwarding

Pipeline diagram showing multiple hardware bypass paths

\[
\begin{align*}
\text{add} & \ x2, \ x10, \ 1 \\
\text{add} & \ x2, \ x11, \ 1 \\
\text{add} & \ x1, \ x2, \ 1 \\
\text{add} & \ x3, \ x4, \ 1 \\
\text{add} & \ x5, \ x3, \ 1 \\
\text{add} & \ x6, \ x1, \ x3 \\
\text{sw} & \ x5, \ 0(x1) \\
\text{jr} & \ x6
\end{align*}
\]

Adding all bypass path to support full hardware bypassing

Topic 2: Fundamental Processor Microarchitecture
Handling load-use RAW dependencies

ALU-use latency is only one cycle, but load-use latency is two cycles.

\[
\begin{align*}
lw & \ x1, \ 0(x2) \\
addi & \ x3, \ x1, \ 1 \\
llw & \ x1, \ 0(x2) \\
addi & \ x3, \ x1, \ 1 \\
\end{align*}
\]

\[
\begin{align*}
ostall_{load\_use\_X\_rs1\_D} & = \\
& \text{val}_D \&\& \text{rs1\_en}_D \&\& \text{val}_X \&\& \text{rf\_wen}_X \\
& \&\& (\text{inst\_rs1}_D == \text{rf\_waddr}_X) \&\& (\text{rf\_waddr}_X != 0) \\
& \&\& (\text{op}_X == \text{lw}) \\
\end{align*}
\]

\[
\begin{align*}
ostall_{load\_use\_X\_rs2\_D} & = \\
& \text{val}_D \&\& \text{rs2\_en}_D \&\& \text{val}_X \&\& \text{rf\_wen}_X \\
& \&\& (\text{inst\_rs2}_D == \text{rf\_waddr}_X) \&\& (\text{rf\_waddr}_X != 0) \\
& \&\& (\text{op}_X == \text{lw}) \\
\end{align*}
\]

\[
\begin{align*}
ostall_D &= \\
& \text{val}_D \&\& (\ ostall_{load\_use\_X\_rs1\_D} | | \ ostall_{load\_use\_X\_rs2\_D} ) \\
\end{align*}
\]

\[
\begin{align*}
bypass_{waddr\_X\_rs1\_D} & = \\
& \text{val}_D \&\& \text{rs1\_en}_D \&\& \text{val}_X \&\& \text{rf\_wen}_X \\
& \&\& (\text{inst\_rs1}_D == \text{rf\_waddr}_X) \&\& (\text{rf\_waddr}_X != 0) \\
& \&\& (\text{op}_X != \text{lw}) \\
\end{align*}
\]

\[
\begin{align*}
bypass_{waddr\_X\_rs2\_D} & = \\
& \text{val}_D \&\& \text{rs2\_en}_D \&\& \text{val}_X \&\& \text{rf\_wen}_X \\
& \&\& (\text{inst\_rs2}_D == \text{rf\_waddr}_X) \&\& (\text{rf\_waddr}_X != 0) \\
& \&\& (\text{op}_X != \text{lw}) \\
\end{align*}
\]
Pipeline diagram for simple assembly sequence

Draw a pipeline diagram illustrating how the following assembly sequence would execute on a fully bypassed pipelined TinyRV1 processor. Include microarchitectural dependency arrows to illustrate how data is transferred along various bypass paths.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw x1, 0(x2)</td>
<td></td>
</tr>
<tr>
<td>lw x3, 0(x4)</td>
<td></td>
</tr>
<tr>
<td>add x5, x1, x3</td>
<td></td>
</tr>
<tr>
<td>sw x5, 0(x6)</td>
<td></td>
</tr>
<tr>
<td>addi x2, x2, 4</td>
<td></td>
</tr>
<tr>
<td>addi x4, x4, 4</td>
<td></td>
</tr>
<tr>
<td>addi x6, x6, 4</td>
<td></td>
</tr>
<tr>
<td>addi x7, x7, -1</td>
<td></td>
</tr>
<tr>
<td>bne x7, x0, loop</td>
<td></td>
</tr>
</tbody>
</table>

5.4. RAW Data Hazards Through Memory

So far we have only studied RAW data hazards through registers, but we must also carefully consider RAW data hazards through memory.

sw x1, 0(x2)
lw x3, 0(x4) # RAW dependency occurs if R[x2] == R[x4]