1 Processor Microarchitectural Design Patterns 3
  1.1. Transactions and Steps ........................................... 3
  1.2. Microarchitecture: Control/Datapath Split ................... 4

2 PARCv1 Single-Cycle Processor 5
  2.2. Single-Cycle Processor Datapath ............................... 7
  2.3. Single-Cycle Processor Control Unit ......................... 12
  2.4. Analyzing Performance ........................................... 12

3 PARCv1 FSM Processor 15
  3.1. High-Level Idea for FSM Processors ......................... 16
  3.2. FSM Processor Datapath .......................................... 16
  3.3. FSM Processor Control Unit .................................... 23
  3.4. Analyzing Performance ........................................... 27

4 PARCv1 Pipelined Processor 29
  4.1. High-Level Idea for Pipelined Processors ................ 30
  4.2. Pipelined Processor Datapath and Control Unit ............ 32
5 **Pipeline Hazards: RAW Data Hazards**

5.1. Expose in Instruction Set Architecture ........................................ 39
5.2. Hardware Stalling ................................................................. 40
5.3. Hardware Bypassing/Forwarding .............................................. 41
5.4. RAW Data Hazards Through Memory ...................................... 45

6 **Pipeline Hazards: Control Hazards**

6.1. Expose in Instruction Set Architecture ........................................ 48
6.2. Hardware Speculation ............................................................. 49
6.3. Interrupts and Exceptions ...................................................... 52

7 **Pipeline Hazards: Structural Hazards**

7.1. Expose in Instruction Set Architecture ........................................ 58
7.2. Hardware Stalling ................................................................. 58
7.3. Hardware Duplication ............................................................ 59

8 **Pipeline Hazards: WAW and WAR Name Hazards**

8.1. Software Renaming ............................................................... 61
8.2. Hardware Stalling ................................................................. 62

9 **Summary of Processor Performance**

10 **Case Study: Transition from CISC to RISC**

10.1. Example CISC: IBM 360/M30 ............................................... 67
10.2. Example RISC: MIPS R2K ..................................................... 70
1. **Processor Microarchitectural Design Patterns**

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

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

<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>Instruction Type</th>
<th>addu</th>
<th>addiu</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>j</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>
<td>✔</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✔</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>
<td>✔</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✔</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>
<td>✔</td>
</tr>
<tr>
<td>Write Memory</td>
<td>✔</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>
<td>✔</td>
</tr>
<tr>
<td>Update PC</td>
<td>✔</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

![Diagram illustrating control unit, datapath, and memory with control signals and status signals.]

- Control Unit
- Datapath
- Memory

Control Signals: imem req_val, imem req, imem resp, dmem req_val, dmem req, dmem resp

Status Signals: Status Signals flowing from the Control Unit to the Datapath and Memory.
2. PARCv1 Single-Cycle Processor

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

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

<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
### 2.1. High-Level Idea for Single-Cycle Processors

<table>
<thead>
<tr>
<th>Operation</th>
<th>addu</th>
<th>addiu</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>j</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>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</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>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</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>
<td></td>
</tr>
<tr>
<td>Write Memory</td>
<td></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>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

![Single-Cycle Processor Diagram](diagram.png)
2.2. Single-Cycle Processor Datapath

Implementing ADDU Instruction

Implementing ADDIU Instruction
2. PARCv1 Single-Cycle Processor  2.2. Single-Cycle Processor Datapath

Implementing ADDU and ADDIU Instructions

Adding the MUL Instruction
Adding the LW and SW Instructions

Adding the J Instruction
Adding the JAL and JR Instructions

Adding the BNE Instruction
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.

\[
\text{lw.ai } rt, \text{ offset}(rs) \\
R[rt] \leftarrow M[R[rs] + \text{sext}(\text{offset}) ]; R[rs] \leftarrow R[rs] + 4
\]
## 2.3. Single-Cycle Processor Control Unit

<table>
<thead>
<tr>
<th>inst</th>
<th>pc sel</th>
<th>op1 sel</th>
<th>alu sel</th>
<th>wb sel</th>
<th>rf waddr</th>
<th>rf wren</th>
<th>imem req</th>
<th>dmem req</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>pc+4</td>
<td>rf</td>
<td>+</td>
<td>alu</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>addiu</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>rf</td>
<td>×</td>
<td>mul</td>
<td>rd</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>pc+4</td>
<td>sext</td>
<td>+</td>
<td>mem</td>
<td>rt</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>j_targ</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>jal</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>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw.ai</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 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
- sext unit    = 1τ
- br_tgen      = 8τ
- j_tgen       = 1τ
- mux          = 3τ
- multiplier   = 20τ
- alu          = 10τ
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?

```assembly
loop:
    lw    r12, 0(r4)
    lw    r13, 0(r5)
    addu  r14, r12, r13
    sw    r14, 0(r6)
    addiu r4, r4, 4
    addiu r5, r5, 4
    addiu r6, r6, 4
    addiu r7, r7, -1
    bne   r7, r0, loop
    jr     r31
```

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.

```assembly
addiu r12, r0, 0
loop:
    lw    r13, 0(r4)
    bne   r13, r6, foo
    addiu r2, r12, 0
    jr     r31
foo:
    addiu r4, r4, 4
    addiu r12, r12, 1
    bne   r12, r5, loop
    addiu r2, r0, -1
    jr     r31
```
3. PARCv1 FSM Processor

### High-Level Idea for FSM Processors

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

- Instructions / program depends on source code, compiler, ISA
- 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>$\approx 1$</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>addu</th>
<th>addiu</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>j</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>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</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>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</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>
<td></td>
</tr>
<tr>
<td>Write Memory</td>
<td></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>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 ADDU Instruction

(pseudo-control-signal syntax)

```
addu rd, rs, rt
```
3. PARCv1 FSM Processor

3.2. FSM Processor Datapath

Full Datapath for PARCv1 FSM Processor

ADDIU Pseudo-Control-Signal Fragment

addiu rd, rs, imm
MUL Instruction
mul rd, rs, rt

M0: A ← RF[r0]
M1: B ← RF[rs]
M2: C ← RF[rt]
M3: A ← A +? B;
   B ← B << 1; C ← C >> 1
M4: A ← A +? B;
   B ← B << 1; C ← C >> 1
...
M35: RF[rd] ← A +? B; goto F0

J Instruction
j  targ

J0: B ← targ << 2
J1: PC ← A jt B; goto F0

JAL Instruction
jal  targ

JA0: RF[31] ← PC
JA1: B ← targ << 2
JA2: PC ← A jt B; goto F0

JR Instruction
jr  rs

JR0: PC ← RF[rs]; goto F0

BNE Instruction
bne rs, rt, offset

B0: A ← RF[rs]
B1: B ← RF[rt]
B2: A ← sext(offset) << 2;
   if A == B goto F0
B3: B ← PC
B4: PC ← 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.

\[
\text{addu.mm rd, rs, rt} \\
M[R[rd]] \leftarrow M[R[rs]] + M[R[rt]]
\]
Adding a New Auto-Incrementing Load Instruction

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

\[ \text{lw.ai } rt, \text{ offset(rs)} \]

\[ R[rt] \leftarrow M[R[rs] + \text{sext(offset)}]; R[rs] \leftarrow R[rs] + 4 \]
3.3. FSM Processor Control Unit

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**

![Hardwired FSM Diagram](image)
Control signal output table for hardwired control unit

F0: memreq.addr ← PC; A ← PC
F1: IR ← RD
F2: PC ← A + 4; A ← A + 4; goto inst
A0: A ← RF[rs]
A1: B ← RF[rt]
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 iau alu rf rd</td>
<td>pc ir a b c wd b c iau alu sel wen val op</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>0 0 0 0 1 0 1 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>0 0 1 0 0 1 0 1 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>A0</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>A2</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</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

- **Datapath Bus**
  - **PC**: Control signal store for microcoded control unit
  - **IR**: Instruction Register
  - **A**: ALU Input A
  - **B**: ALU Input B
  - **C**: ALU Output
  - **RF**: Register File
  - **RD**: Data Bus
  - **WD**: Control Word Bus

### Datapath Control Signals

- **pc_en**: PC Enable
- **ir_en**: IR Enable
- **a_en**: A Enable
- **b_en**: B Enable
- **c_en**: C Enable
- **a_sel**: A Select
- **b_sel**: B Select
- **c_sel**: C Select
- **ir_addr**: IR Address
- **alu_addr**: ALU Address
- **rd_bus_en**: RD Bus Enable
- **al_mul**: ALU Multiply
- **al_iau**: ALU IAU
- **alu_bus_en**: ALU Bus Enable
- **iau**: IAU
- **rd_en**: RD Enable
- **memreq_addr**: Memory Request Address
- **memreq_data**: Memory Request Data
- **memresp_addr**: Memory Response Address
- **memresp_data**: Memory Response Data
- **b_en**: B Enable
- **rd_bus_en**: RD Bus Enable
- **rd_en**: RD Enable
- **memreq_addr**: Memory Request Address
- **memreq_data**: Memory Request Data
- **memresp_addr**: Memory Response Address
- **memresp_data**: Memory Response Data

### ALU Functions

- **alu func**: ALU Function
  - +4: A + 4
  - +: A + B
  - +?: A +? B
  - cmp: A == B
  - jt: { A[31:28], B[27:0] }

### IAU Functions

- **iau func**: IAU Function
  - si: sext(IR[15:0])
  - ts: IR[25:0] << 2
  - sis: sext(IR[15:0]) << 2

### Bus Enables

- **Bus Enables**
  - **state**: Current State
  - **pc**: PC Enable
  - **iau**: IAU Enable
  - **alu**: ALU Enable
  - **rf**: RF Enable
  - **rd**: RD Enable
  - **bus_en**: Bus Enable
  - **memreq**: Memory Request
  - **memresp**: Memory Response

### Register Enables

- **Register Enables**
  - **state**: Current State
  - **pc**: PC Enable
  - **ir**: IR Enable
  - **a**: A Enable
  - **b**: B Enable
  - **c**: C Enable
  - **wd**: WD Enable

### Mux

- **Mux**
  - **state**: Current State
  - **b**: B Enable
  - **c**: C Enable
  - **func**: Function
  - **rf**: RF Enable
  - **memreq**: Memory Request
  - **memresp**: Memory Response

### Next State

- **Next State**
  - **state**: Current State
  - **op**: Operation
  - **next**: Next State

---

- **B0**: A ← RF[rs]
- **B1**: B ← RF[rt]
- **B2**: A ← sext(offset) << 2; if A == B goto F0
- **B3**: B ← PC
- **B4**: PC ← A + B; goto F0

---

### Table

<table>
<thead>
<tr>
<th>state</th>
<th>pc</th>
<th>iau</th>
<th>alu</th>
<th>rf</th>
<th>rd</th>
<th>pc</th>
<th>ir</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>wd</th>
<th>b</th>
<th>c</th>
<th>iau</th>
<th>alu</th>
<th>sel</th>
<th>wen</th>
<th>val</th>
<th>op</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>rs</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>rt</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>sis cmp</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>b</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

26
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 = $20\tau$
- iau unit = $1\tau$
- 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 nanoseconds will it take to execute the vector-vector add example assuming n is 64?

```
loop:
  lw    r12, 0(r4)
  lw    r13, 0(r5)
  addu  r14, r12, r13
  sw    r14, 0(r6)
  addiu r4, r4, 4
  addiu r5, r5, 4
  addiu r6, r6, 4
  addiu r7, r7, -1
  bne   r7, r0, loop
  jr    r31
```

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.

```
addiu r12, r0, 0
loop:
  lw    r13, 0(r4)
  bne   r13, r6, foo
  addiu r2, r12, 0
  jr    r31
foo:
  addiu r4, r4, 4
  addiu r12, r12, 1
  bne   r12, r5, loop
  addiu r2, r0, -1
  jr    r31
```
4. PARCv1 Pipelined Processor

Time

Program = Instructions

Program × Cycles × Time


• Instructions / program depends on source code, compiler, ISA
• 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 Slot</th>
<th>7pm</th>
<th>8pm</th>
<th>9pm</th>
<th>10pm</th>
<th>11pm</th>
<th>12am</th>
<th>1am</th>
<th>2am</th>
<th>3am</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anne's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ben's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cathy's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dave's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pipelined Laundry**

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

**Pipelined Laundry with Slow Dryers**

<table>
<thead>
<tr>
<th>Time Slot</th>
<th>7pm</th>
<th>8pm</th>
<th>9pm</th>
<th>10pm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anne's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
</tr>
<tr>
<td>Ben's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
</tr>
<tr>
<td>Cathy's Load</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
<td>🥰</td>
</tr>
<tr>
<td>Dave's Load</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>addu</th>
<th>addiu</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>j</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>
<td>✓</td>
</tr>
<tr>
<td>Decode Instruction</td>
<td>✓</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>
<td>✓</td>
</tr>
<tr>
<td>Register Arithmetic</td>
<td>✓</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>
<td>✓</td>
</tr>
<tr>
<td>Write Memory</td>
<td></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>
<td>✓</td>
</tr>
<tr>
<td>Update PC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Single-Cycle

FSM

Pipelined
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
- Divided 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

```
addiu r1, r2, 1
addiu r3, r4, 1
addiu r5, r6, 1
```
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.

\[
\text{lw.ai } rt, \text{imm}(rs) \\
R[rt] \leftarrow M[R[rs] + \text{sext(imm)}]; R[rs] \leftarrow R[rs] + 4
\]
Pipeline diagrams

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu r1, r2, 1</td>
<td></td>
</tr>
<tr>
<td>addiu r3, r4, 1</td>
<td></td>
</tr>
<tr>
<td>addiu r5, r6, 1</td>
<td></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>
Control logic with stalling and squashing

\[ \text{reg}_\text{en}_B = !\text{stall}_B \]

\[
\begin{align*}
\text{always}_\text{ff} @(&\text{posedge clk}) \\
\quad &\text{if ( reset )} \\
\quad &\quad \text{val}_B <= 0 \\
\quad &\text{else if ( reg}_\text{en}_B \text{ )} \\
\quad &\quad \text{val}_B <= \text{next}_\text{val}_A
\end{align*}
\]

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

\[ \text{ostall}_B = \text{val}_B \&\& !\text{squash}_B \&\& ( \text{ostall}_\text{hazard1}_B \; || \; \text{ostall}_\text{hazard2}_B ) \]

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

\[ \text{osquash}_B = \text{val}_B \&\& !\text{squash}_B \&\& \text{stall}_B \&\& ( \text{osquash}_\text{hazard1}_B \; || \; \ldots ) \]

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

<table>
<thead>
<tr>
<th>\text{squash}_B</th>
<th>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 osquash_B is not factored into squash_B.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ostall}_B</td>
<td>A squash takes priority, since a squashed transaction is invalid and thus it should not originate a stall.</td>
</tr>
<tr>
<td>\text{stall}_B</td>
<td>A squash takes priority, since a squashed transaction is invalid and thus it should not actually stall.</td>
</tr>
<tr>
<td>\text{osquash}_B</td>
<td>Originating squash due to hazards detected in B stage. A squash takes priority, since a squashed transaction is invalid and thus it should not originate a squash. A stall also takes priority, since a stalling transactions should not originate a squash.</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.

\[
\begin{align*}
\text{addiu } & r1, r2, 1 \\
\text{addiu } & r3, r1, 1 \\
\text{addiu } & r4, r3, 1 \\
\end{align*}
\]

Using pipeline diagrams to illustrate RAW hazards

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

```
addiu r1, r2, 1
addiu r3, r1, 1
addiu r4, r3, 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.

```
addiu r1, r2, 1
nop
nop
nop
addiu r3, r1, 1
nop
nop
nop
addiu r4, r3, 1
```

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

```
addiu r1, r2, 1
addiu r6, r7, 1
addiu r8, r9, 1
nop
addiu r3, r1, 1
addiu r3, r1, 1
nop
addiu r4, r3, 1
addiu r4, r3, 1
```

**Pipeline diagram showing software scheduling for RAW data hazards**

```
addiu r1, r2, 1
addiu r6, r7, 1
addiu r8, r9, 1
nop
addiu r3, r1, 1
addiu r3, r1, 1
nop
addiu r4, r3, 1
addiu r4, r3, 1
```

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

```
addiu r1, r2, 1
addiu r3, r1, 1
addiu r4, r3, 1
```

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>addu</th>
<th>addiu</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>j</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs_en</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rt_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>
<tr>
<td>rf_waddr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ostall_waddr_X_rs_D =
val_D && rs_en_D && val_X && rf_wen_X
&& (inst_rs_D == rf_waddr_X) && (rf_waddr_X != 0)

ostall_waddr_M_rs_D =
val_D && rs_en_D && val_M && rf_wen_M
&& (inst_rs_D == rf_waddr_M) && (rf_waddr_M != 0)

ostall_waddr_W_rs_D =
val_D && rs_en_D && val_W && rf_wen_W
&& (inst_rs_D == rf_waddr_W) && (rf_waddr_W != 0)

... similar for ostall signals for rt source register ...

ostall_D = val_D
&& ( ostall_waddr_X_rs_D || ostall_waddr_X_rt_D
|| ostall_waddr_M_rs_D || ostall_waddr_M_rt_D
|| ostall_waddr_W_rs_D || ostall_waddr_W_rt_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”.

41
Pipeline diagram showing hardware bypassing for RAW data hazards

```
addiu r1, r2, 1
addiu r3, r1, 1
addiu r4, r3, 1
```

Adding single bypass path to support limited hardware bypassing

\[
\text{Deriving the bypass and stall signals}
\]

\[
\text{ostall_waddr}_X_{-}\text{rs}_D = 0
\]

\[
\text{bypass_waddr}_X_{-}\text{rs}_D = \text{val}_D \&\& \text{rs}_\text{en}_D \&\& \text{val}_X \&\& \text{rf}_\text{wen}_X
\]

\[
\&\& (\text{inst}_\text{rs}_D == \text{rf}_\text{waddr}_X) \&\& (\text{rf}_\text{waddr}_X != 0)
\]
Pipeline diagram showing multiple hardware bypass paths

addiu r2, r10, 1
addiu r2, r11, 1
addiu r1, r2, 1
addiu r3, r4, 1
addiu r5, r3, 1
addu r6, r1, r3
sw r5, 0(r1)
jr r6

Adding all bypass path to support full hardware bypassing

---

5. Pipeline Hazards: RAW Data Hazards

5.3. Hardware Bypassing/Forwarding

---
Handling load-use RAW dependencies

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

```
lw r1, 0(r2)
addiu r3, r1, 1

lw r1, 0(r2)
addiu r3, r1, 1
```

```
ostall_load_use_X_rs_D =
  val_D && rs_en_D && val_X && rf_wen_X
  && (inst_rs_D == rf_waddr_X) && (rf_waddr_X != 0)
  && (op_X == lw)

ostall_load_use_X_rt_D =
  val_D && rt_en_D && val_X && rf_wen_X
  && (inst_rt_D == rf_waddr_X) && (rf_waddr_X != 0)
  && (op_X == lw)

ostall_D =
  val_D && (ostall_load_use_X_rs_D || ostall_load_use_X_rt_D )

bypass_waddr_X_rs_D =
  val_D && rs_en_D && val_X && rf_wen_X
  && (inst_rs_D == rf_waddr_X) && (rf_waddr_X != 0)
  && (op_X != lw)

bypass_waddr_X_rt_D =
  val_D && rt_en_D && val_X && rf_wen_X
  && (inst_rt_D == rf_waddr_X) && (rf_waddr_X != 0)
  && (op_X != lw)
```
Pipeline diagram for simple assembly sequence

Draw a pipeline diagram illustrating how the following assembly sequence would execute on a fully bypassed pipelined PARCv1 processor. Include microarchitectural dependency arrows to illustrate how data is transferred along various bypass paths.

```
lw r1, 0(r2)
lw r3, 0(r4)
addu r5, r1, r3
sw r5, 0(r6)
addiu r2, r2, 4
addiu r4, r4, 4
addiu r6, r6, 4
addiu r7, r7, -1
bne r7, r0, loop
```

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 r1, 0(r2)
lw r3, 0(r4) # RAW dependency occurs if R[r2] == R[r4]
```
6. Pipeline Hazards: Control Hazards

Control hazards occur when whether or not an instruction should be executed depends on a control decision made by an earlier instruction. We use architectural dependency arrows to illustrate control dependencies in assembly code sequences.

**Static Instr Sequence**

```assembly
addiu r1, r0, 1
j    foo
opA
opB
foo: addiu r2, r3, 1
     bne r0, r1, bar
     opC
     opD
     opE
bar: addiu r4, r5, 1
```

**Dynamic Instr Sequence**

```assembly
addiu r1, r0, 1
j    foo
opA
opB
addiu r2, r3, 1
bne r0, r1, bar
addiu r4, r5, 1
```

Using pipeline diagrams to illustrate control hazards

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

```
addiu r1, r0, 1
j    foo
addiu r2, r3, 1
bne r0, r1, bar
addiu r4, r5, 1
```
The jump resolution latency and branch resolution latency are the number of cycles we need to delay the fetch of the next instruction in order to avoid any kind of control hazard. Jump resolution latency is two cycles, and branch resolution latency is three cycles.

```
addiu r1, r0, 1
j foo
addiu r2, r3, 1
bne r0, r1, bar
addiu r4, r5, 1
```

**Approaches to resolving control hazards**

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

- **Software Predication:** Programmer or compiler converts control flow into data flow by using instructions that conditionally execute based on a data value

- **Hardware Speculation:** Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects when there is a problem and re-executes instructions the instructions that are along the correct control flow

- **Software Hints:** Programmer or compiler provides hints about whether a conditional branch will be taken or not taken, and hardware can use these hints for more efficient hardware speculation
6. Pipeline Hazards: Control Hazards

6.1. Expose in Instruction Set Architecture

Expose branch delay slots as part of the instruction set. Branch delay slots are instructions that follow a jump or branch and are always executed regardless of whether a jump or branch is taken or not taken. Compiler tries to insert useful instructions, otherwise inserts nops.

```assembly
addiu r1, r0, 1  
j foo
nop
opA
opB

foo: addiu r2, r3, 1
bne r0, r1, bar
nop
nop
opC
opD
opE

bar: addiu r4, r5, 1
```

Assume we modify the PARCv1 instruction set to specify that J, JAL, and JR instructions have a single-instruction branch delay slot (i.e., one instruction after a J, JAL, and JR is always executed) and the BNE instruction has a two-instruction branch delay slot (i.e., two instructions after a BNE are always executed).

Pipeline diagram showing using branch delay slots for control hazards

```
| addiu r1, r0, 1 |
|-----------------
| j foo           |
| nop             |
| addiu r2, r3, 1 |
| bne r0, r1, bar |
| nop             |
| nop             |
| addiu r4, r5, 1 |
```
6.2. Hardware Speculation

Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects when there is a problem and re-executes instructions the instructions that are along the correct control flow. For now, we will only consider a simple branch prediction scheme where the hardware always predicts not taken.

**Pipeline diagram when branch is not taken**

```
addiu r1, r0, 1
j foo
opA
addiu r2, r3, 1
bne r0, r1, bar
opC
opD
```

**Pipeline diagram when branch is taken**

```
addiu r1, r0, 1
j foo
opA
addiu r2, r3, 1
bne r0, r1, bar
opC
opD
addiu r4, r5, 1
```
6.2. Hardware Speculation

Modifications to datapath/control to support hardware speculation

Deriving the squash signals

\[
\text{osquash}_j_D = (\text{op}_D == j) \lor (\text{op}_D == \text{jal}) \lor (\text{op}_D == \text{jr})
\]

\[
\text{osquash}_\text{br}_X = \text{br}_\text{taken}_X
\]

Our generic stall/squash scheme gives priority to squashes over stalls. A squashed instruction is invalid, so it should not stall the pipeline.

\[
\text{squash}_D = \text{val}_D \land \text{osquash}_X
\]

\[
\text{ostall}_D = \text{val}_D \land \neg \text{squash}_D \land (\text{ostall}_\text{hazard}_1_D \lor \ldots)
\]

\[
\text{stall}_D = \text{val}_D \land \neg \text{squash}_D \land \text{ostall}_D
\]

\[
\text{osquash}_D = \text{val}_D \land \neg \text{squash}_D \land \neg \text{stall}_D \land \text{osquash}_j_D
\]

Important: PC select logic must give priority to older instructions (i.e., prioritize branches over jumps)!
Pipeline diagram for simple assembly sequence

Draw a pipeline diagram illustrating how the following assembly sequence would execute on a fully bypassed pipelined PARCv1 processor that uses hardware speculation which always predicts not-taken. Unlike the “standard” PARCv1 processor, you should also assume that we add a single-instruction branch delay slot to the instruction set. So this processor will partially expose the control hazard in the instruction, but also use hardware speculation. Include microarchitectural dependency arrows to illustrate both data and control flow.

```
addiu r1, r2, 1
bne   r0, r3, foo  # assume R[rs] != 0
addiu r4, r5, 1   # instruction is in branch delay slot
addiu r6, r7, 1
...
foo:
  addu   r8, r1, r4
addiu r9, r1, 1
```
6.3. Interrupts and Exceptions

Interrupts and exceptions alter the normal control flow of the program. They are caused by an external or internal event that needs to be processed by the system, and these events are usually unexpected or rare from the program’s point of view.

• Asynchronous Interrupts
  – Input/output device needs to be serviced
  – Timer has expired
  – Power disruption or hardware failure

• Synchronous Exceptions
  – Undefined opcode, privileged instruction
  – Arithmetic overflow, floating-point exception
  – Misaligned memory access for instruction fetch or data access
  – Memory protection violation
  – Virtual memory page faults
  – System calls (traps) to jump into the operating system kernel

Interrupts and Exception Semantics

• Interrupts are asynchronous with respect to the program, so the microarchitecture can decide when to service the interrupt

• Exceptions are synchronous with respect to the program, so they must be handled immediately
To handle an interrupt or exception the hardware/software must:

- Stop program at current instruction \( I \), ensure previous insts finished
- Save cause of interrupt or exception in privileged arch state
- Save the PC of the instruction \( I \) in a special register (EPC)
- Switch to privileged mode
- Set the PC to the address of either the interrupt or the exception handler
- Disable interrupts
- Save the user architectural state
- Check the type of interrupt or exception
- **Handle the interrupt or exception**
- Enable interrupts
- Switch to user mode
- Set the PC to EPC if \( I \) should be restarted
- Potentially set PC to EPC+4 if we should skip \( I \)

**Handling a misaligned data address and syscall exceptions**

**Static Instr Sequence**

```
addiu r1, r0, 0x2001
lw    r2, 0(r1)
syscall
opB
opC...
```

```
exception_handler:
  opD     # disable interrupts
  opE     # save user registers
  opF     # check exception type
  opG     # handle exception
  opH     # enable interrupts
  addiu EPC, EPC, 4
  eret
```

**Dynamic Instr Sequence**

```
addiu r1, r0, 0x2001
lw    r2, 0(r1) (excep)
opD
opE
opF
opG
opH
addiu EPC, EPC, 4
eret
```

```
syscall (excep)
```

```
addiu EPC, EPC, 4
```
Interrupts and Exceptions in a PARCv4? Pipelined Processor

- How should we handle a single instruction which generates multiple exceptions in different stages as it goes down the pipeline?
  - Exceptions in earlier pipeline stages override later exceptions for a given instruction

- How should we handle multiple instructions generating exceptions in different stages at the same or different times?
  - We always want the execution to appear as if we have completely executed one instruction before going onto the next instruction
  - So we want to process the exception corresponding to the earliest instruction in program order first
  - Hold exception flags in pipeline until commit point
  - Commit point is after all exceptions could be generated but before any architectural state has been updated
  - To handle an exception at the commit point: update cause and EPC, squash all stages before the commit point, and set PC to exception handler

- How and where to handle external asynchronous interrupts?
  - Inject asynchronous interrupts at the commit point
  - Asynchronous interrupts will then naturally override exceptions caused by instructions earlier in the pipeline
Modifications to datapath/control to support exceptions

Deriving the squash signals

\[
\begin{align*}
osquash_j_D &= (\text{op}_D == \text{j}) || (\text{op}_D == \text{jal}) || (\text{op}_D == \text{jr}) \\
osquash_{br} X &= \text{br\_taken}\_X \\
osquash_{xcept} M &= \text{exception}\_M
\end{align*}
\]

Control logic needs to redirect the front end of the pipeline just like for a jump or branch. Again, squashes take priority over stalls, and PC select logic must give priority to older instructions (i.e., prioritize exceptions, over branches, over jumps)!
Pipeline diagram of exception handling

```
addiu r1, r0, 0x2001
lw r2, 0(r1)   # assume causes misaligned address exception
syscall        # causes a syscall exception

exception_hander:
opD   # disable interrupts
opE   # save user registers
opF   # check exception type
opG   # handle exception
opH   # enable interrupts
addiu EPC, EPC, 4
eret
```
7. Pipeline Hazards: Structural Hazards

Structural hazards occur when an instruction in the pipeline needs a resource being used by another instruction in the pipeline. The PARCv1 processor pipeline is specifically designed to avoid any structural hazards.

Let’s introduce a structural hazard by allowing ADDU, ADDIU, MUL, and JAL instructions to write to the register file in the M stage instead of waiting until the W stage. We would need to add another writeback mux in the W stage and carefully handle bypassing.

Using pipeline diagrams to illustrate structural hazards

We use structural dependency arrows to illustrate structural hazards.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu r1, r2, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu r3, r4, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r5, 0(r6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu r7, r8, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Approaches to resolving structural hazards

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

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

- **Hardware Duplication:** Add more hardware so that each instruction can access separate resources at the same time
7. Pipeline Hazards: Structural Hazards

7.1. Expose in Instruction Set Architecture

Insert independent instructions or nop's to delay an ADDU, ADDIU, MUL, or JAL instructions if they follow a LW instruction.

Pipeline diagram showing software scheduling for structural hazards

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu r1, r2, 1</td>
</tr>
<tr>
<td>addiu r3, r4, 1</td>
</tr>
<tr>
<td>lw r5, 0(r6)</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>addiu r7, r8, 1</td>
</tr>
</tbody>
</table>

7.2. Hardware Stalling

Hardware includes control logic that freezes an ADDU, ADDIU, MUL, or JAL instruction if a LW instruction is ahead in the pipeline.

Pipeline diagram showing hardware stalling for structural hazards

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu r1, r2, 1</td>
</tr>
<tr>
<td>addiu r3, r4, 1</td>
</tr>
<tr>
<td>lw r5, 0(r6)</td>
</tr>
<tr>
<td>addiu r7, r8, 1</td>
</tr>
</tbody>
</table>

Deriving the stall signal

\[
\text{ostall_wport_hazard}_D = \text{val}_D \land \text{rf_wen}_D \land \text{val}_X \land (\text{op}_X = \text{lw})
\]

Stall far before the structural hazard actually occurs, because we know exactly how instructions move down the pipeline. Also possible to use dynamic arbitration in the back-end of the pipeline.
7.3. Hardware Duplication

Add a second write port so that an ADDU, ADDIU, MUL, or JAL instruction can writeback to the register file at the same time as a LW.

Does allowing early writeback help performance in the first place?

```
addiu r1, r2, 1
addiu r3, r1, 1
addiu r4, r3, 1
addiu r5, r4, 1
addiu r6, r5, 1
addiu r7, r6, 1
```
8. Pipeline Hazards: WAW and WAR Name Hazards

WAW dependencies occur when an instruction overwrites a register than an earlier instruction has already written. WAR dependencies occur when an instruction writes a register than an earlier instruction needs to read. We use architectural dependency arrows to illustrate WAW and WAR dependencies in assembly code sequences.

```
mul r1, r2, r3
addiu r4, r6, 1
addiu r1, r5, 1
```

WAW name hazards occur when an instruction in the pipeline writes a register before an earlier instruction (in back of the pipeline) has had a chance to write that same register.

WAR name hazards occur when an instruction in the pipeline writes a register before an earlier instruction (in back of pipeline) has had a chance to read that same register.

The PARCv1 processor pipeline is specifically designed to avoid any WAW or WAR name hazards. Instructions always write the registerfile in-order in the same stage, and instructions always read registers in the front of the pipeline and write registers in the back of the pipeline.

Let’s introduce a WAW name hazard by using an iterative variable latency multiplier, and allowing other instructions to continue executing while the multiplier is working.
Using pipeline diagrams to illustrate WAW name hazards

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul r1, r2, r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu r4, r6, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu r1, r5, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Approaches to resolving structural hazards

- **Software Renaming**: Programmer or compiler changes the register names to avoid creating name hazards
- **Hardware Renaming**: Hardware dynamically changes the register names to avoid creating name hazards
- **Hardware Stalling**: Hardware includes control logic that freezes later instructions until earlier instruction has finished either writing or reading the problematic register name

8.1. Software Renaming

As long as we have enough architectural registers, renaming registers in software is easy. WAW and WAR dependencies occur because we have a finite number of architectural registers.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register 1</th>
<th>Register 2</th>
<th>Register 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul r1, r2, r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu r4, r6, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu r7, r5, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.2. Hardware Stalling

Simplest approach is to add stall logic in the decode stage similar to what the approach used to resolve other hazards.

```
mul r1, r2, r3
addiu r4, r6, 1
addiu r1, r5, 1
```

Deriving the stall signal

\[
ostall\text{-}struct\_hazard\_D = val\_D \&\& (op\_D == MUL) \&\& \neg imul\_rdy\_D
\]

\[
ostall\text{-}waw\_hazard\_D =
val\_D \&\& rf\_wen\_D \&\& val\_Z \&\& rf\_wen\_Z
\&\& (rf\_waddr\_D == rf\_waddr\_Z) \&\& (rf\_waddr\_Z != 0)
\]

9. Summary of Processor Performance

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

Results for vector-vector add example

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>Inst</th>
<th>CPI</th>
<th>Cycle Time</th>
<th>Exec Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Cycle Processor</td>
<td>576</td>
<td>1.0</td>
<td>74 (\tau)</td>
<td>43 k(\tau)</td>
</tr>
<tr>
<td>FSM Processor</td>
<td>576</td>
<td>6.5</td>
<td>40 (\tau)</td>
<td>150 k(\tau)</td>
</tr>
<tr>
<td>Pipelined Processor</td>
<td>576</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9. Summary of Processor Performance

Estimating cycle time for pipelined processor

- register read = 1τ
- register write = 1τ
- regfile read = 10τ
- regfile write = 10τ
- memory read = 20τ
- memory write = 20τ
- +4 unit = 4τ
- sext unit = 1τ
- br_tgen = 8τ
- j_tgen = 1τ
- mux = 3τ
- multiplier = 20τ
- alu = 10τ
9. Summary of Processor Performance

Estimating execution time

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

```assembly
loop:
    lw    r12, 0(r4)
    lw    r13, 0(r5)
    addu  r14, r12, r13
    sw    r14, 0(r6)
    addiu r4, r4, 4
    addiu r5, r5, 4
    addiu r6, r6, 4
    addiu r7, r7, -1
    bne   r7, r0, loop
    jr     r31
```
Using our first-order equation for processor performance, how long in $\tau$ will it take to execute the mystery program assuming $n$ is 64 and that we find a match on the last element.

```
addiu r12, r0, 0
loop:
    lw    r13, 0(r4)
    bne   r13, r6, foo
    addiu r2, r12, 0
    jr    r31
foo:
    addiu r4, r4, 4
    addiu r12, r12, 1
    bne   r12, r5, loop
    addiu r2, r0, -1
    jr    r31
```
10. Case Study: Transition from CISC to RISC

- Microcoding thrived in the 1970’s
  - ROMs significantly faster than DRAMs
  - For complex instruction sets, microcode was cheaper and simpler
  - New instructions supported without modifying datapath
  - Fixing bugs in controller is easier
  - ISA compatibility across models relatively straight-forward

— Maurice Wilkes, 1954
## 10.1. Example CISC: IBM 360/M30

<table>
<thead>
<tr>
<th></th>
<th>M30</th>
<th>M40</th>
<th>M50</th>
<th>M65</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath width (bits)</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>µinst width (bits)</td>
<td>50</td>
<td>52</td>
<td>85</td>
<td>87</td>
</tr>
<tr>
<td>µcode size (1K µinsts)</td>
<td>4</td>
<td>4</td>
<td>2.75</td>
<td>2.75</td>
</tr>
<tr>
<td>µstore technology</td>
<td>CCROS</td>
<td>TCROS</td>
<td>BCROS</td>
<td>BCROS</td>
</tr>
<tr>
<td>µstore cycle (ns)</td>
<td>750</td>
<td>625</td>
<td>500</td>
<td>200</td>
</tr>
<tr>
<td>Memory cycle (ns)</td>
<td>1500</td>
<td>2500</td>
<td>2000</td>
<td>750</td>
</tr>
<tr>
<td>Rental fee ($K/month)</td>
<td>4</td>
<td>7</td>
<td>15</td>
<td>35</td>
</tr>
</tbody>
</table>

TROS = transformer read-only storage (magnetic storage)  
BCROS = balanced capacitor read-only storage (capacitive storage)  
CCROS = card capacitor read-only storage (metal punch cards, replace in field)

Only the fastest models (75,95) were hardwired

**IBM 360/M30 microprogram for register-register logical OR**
IBM 360/M30 microprogram for register-register binary ADD

Analyzing Microcoded Machines

• John Cocke and group at IBM
  - Working on a simple pipelined processor, 801, and advanced compilers
  - Ported experimental PL8 compiler to IBM 370, and only used simple register-register and load/store instructions similar to 801
  - Code ran faster than other existing compilers that used all 370 instructions! (up to 6 MIPS, whereas 2 MIPS considered good before)

• Joel Emer and Douglas Clark at DEC
  - Measured VAX-11/780 using external hardware
  - Found it was actually a 0.5 MIPS machine, not a 1 MIPS machine
  - 20% of VAX instrs = 60% of µcode, but only 0.2% of the dynamic execution

• VAX 8800, high-end VAX in 1984
  - Control store: 16K×147b RAM, Unified Cache: 64K×8b RAM
  - 4.5× more microstore RAM than cache RAM!
From CISC to RISC

- Key changes in technology constraints
  - Logic, RAM, ROM all implemented with MOS transistors
  - RAM ≈ same speed as ROM

- Use fast RAM to build fast instruction cache of user-visible instructions, not fixed hardware microfragments
  - Change contents of fast instruction memory to fit what app needs

- Use simple ISA to enable hardwired pipelined implementation
  - Most compiled code only used a few of CISC instructions
  - Simpler encoding allowed pipelined implementations
  - Load/Store Reg-Reg ISA as opposed to Mem-Mem ISA

- Further benefit with integration
  - Early 1980’s → fit 32-bit datapath, small caches on single chip
  - No chip crossing in common case allows faster operation
10.2. Example RISC: MIPS R2K

- MIPS R2K is one of the first popular pipelined RISC processors
- MIPS R2K implements the MIPS I instruction set
- MIPS = Microprocessor without Interlocked Pipeline Stages

- MIPS I used software scheduling to avoid some RAW hazards by including a single-instruction load-use delay slot
- MIPS I used software scheduling to avoid some control hazards by including a single-instruction branch delay slot

### One-Instr Branch Delay Slot

```
addiu r1, r2, 1
j    foo
addiu r3, r4, 1  # BDS
...
foo:
  addiu r5, r6, 1
  bne  r7, r8, bar
  addiu r9, r10, 1  # BDS
  ...
bar:
```

Present in all MIPS instruction sets; not possible to deprecate and still enable legacy code to execute on new microarchitectures

### One-Instr Load-Use Delay Slot

```
lw  r1, 0(r2)
lw  r3, 0(r4)
addiu r2, r2, 4  # LDS
addu  r5, r1, r3
```

Deprecated in MIPS II instruction set; legacy code can still execute on new microarchitectures, but code using the MIPS II instruction set can rely in hardware stalling
MIPS R2K Microarchitecture

The pipelined datapath and control were located on a single die. Cache control and memory management unit were also integrated on-die, but the actual tag and data storage for the cache was located off-chip.

Used two-phase clocking to enable five pipeline stages to fit into four clock cycles. This avoided the need for explicit bypassing from the W stage to the end of the D stage.

Two-phase clocking enabled a single-cycle branch resolution latency since register read, branch address generation, and branch comparison can fit in a single cycle.
MIPS R2K VLSI Design

Process: 2 µm, two metal layers
Clock Frequency: 8–15 MHz
Size: 110K transistors, 80 mm²
CISC/RISC Convergence

**MIPS R10K** uses sophisticated out-of-order engine; branch delay slot not useful

– Gwennap, MPR, 1994

**Intel Nehalem** frontend breaks x86 CISC into smaller RISC-like µops; µcode engine handles rarely used complex instr

– Kanter, Real World Technologies, 2009
Microprogramming Today

- Microprogramming is far from extinct

- Played a crucial role in microprocessors of the 1980s (DEC VAX, Motorola 68K series, Intel 386/486)

- Microprogramming plays assisting role in many modern processors (AMD Phenom, Intel Nehalem, Intel Atom, IBM Z196)
  - 761 Z196 instructions executed with hardwired control
  - 219 Z196 “complex” instructions always executed with microcode
  - 24 Z196 instructions conditionally executed with microcode

- Patchable microcode common for post-fabrication bug fixes (Intel processors load µcode patches at bootup)