5 WAW and WAR Name Hazards 38
  5.1. Software Renaming ............................................. 39
  5.2. Hardware Stalling .............................................. 40

6 Analyzing Processor Performance 41

7 Case Study: MIPS R2K 44
1. Pipelined Processors

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

- 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>Topic 01 Single-Cycle Processor</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Topic 02 FSM Processor</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>this topic → 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
Pipelining laundry analogy

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

Sequential Laundry

<table>
<thead>
<tr>
<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>Ben's Load</td>
<td>Cathy's Load</td>
<td>Dave's Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipelined Laundry

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

Pipelined Laundry with Slow Dryers

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

Visualizing space, time, and transactions with pipeline diagrams
### Comparing single-cycle, FSM, and pipelined 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><strong>Fetch Instruction</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Decode Instruction</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Read Registers</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Register Arithmetic</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
</tr>
<tr>
<td><strong>Read Memory</strong></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Write Memory</strong></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Write Registers</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Update PC</strong></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

![Single-Cycle Processor Diagram]

#### FSM Processor

![ FSM Processor Diagram ]

#### Pipelined Processor

![Pipelined Processor Diagram]
Pipelining a PARCv1 processor

- 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
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.

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

- addiu r1, r2, 1
- addiu r3, r4, 1
- addiu r5, r6, 1

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
2. 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.
Approaches to resolving data hazards

- **Software Scheduling:** Programmer or compiler explicitly avoids scheduling instructions that would create data hazards

- **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

- **Hardware Bypassing:** Hardware allows values to be sent from an earlier instruction to a later instruction before the earlier instruction has left the pipeline

- **Hardware Speculation:** Hardware guesses that there is no hazard and allows later instructions to potentially read invalid data; detects when there is a problem and re-executes instructions that operated on invalid data
2.1. Software Scheduling

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

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

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

\[
\begin{align*}
\text{addiu } r1, r2, 1 \\
\text{addiu } r6, r7, 1 \\
\text{addiu } r8, r9, 1 \\
\text{nop} \\
\text{addiu } r3, r1, 1 \\
\text{nop} \\
\text{nop} \\
\text{nop} \\
\text{addiu } r4, r3, 1
\end{align*}
\]

Pipeline diagram showing software scheduling for RAW data hazards

\[
\begin{align*}
\text{addiu } r1, r2, 1 \\
\text{addiu } r6, r7, 1 \\
\text{addiu } r8, r9, 1 \\
\text{nop} \\
\text{addiu } r3, r1, 1 \\
\text{nop} \\
\text{nop} \\
\text{nop} \\
\text{addiu } r4, r3, 1
\end{align*}
\]
2.2. 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
```

```
addiu r1, r2, 1
addiu r3, r1, 1
```
Modifications to datapath/control to support hardware stalling

Deriving the stall signal

<table>
<thead>
<tr>
<th>Command</th>
<th>(\text{ren0})</th>
<th>(\text{raddr0})</th>
<th>(\text{ren1})</th>
<th>(\text{raddr1})</th>
<th>(\text{wen})</th>
<th>(\text{waddr})</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</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>
</tr>
<tr>
<td>jr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
stall_waddr_X_raddr0_D =
  ren0_D && wen_X && (raddr0_D == waddr_X) && (waddr_X != 0)

stall_waddr_M_raddr0_D =
  ren0_D && wen_M && (raddr0_D == waddr_M) && (waddr_M != 0)

stall_waddr_W_raddr0_D =
  ren0_D && wen_W && (raddr0_D == waddr_W) && (waddr_W != 0)

stall_waddr_X_raddr1_D =
  ren1_D && wen_X && (raddr1_D == waddr_X) && (waddr_X != 0)

stall_waddr_M_raddr1_D =
  ren1_D && wen_M && (raddr1_D == waddr_M) && (waddr_M != 0)

stall_waddr_W_raddr1_D =
  ren1_D && wen_W && (raddr1_D == waddr_W) && (waddr_W != 0)

stall = stall_waddr_X_raddr0_D || stall_waddr_X_raddr1_D
  || stall_waddr_M_raddr0_D || stall_waddr_M_raddr1_D
  || stall_waddr_W_raddr0_D || stall_waddr_W_raddr1_D

reg_en_F = !stall
reg_en_D = !stall
insert_nop_D = stall
2.3. Hardware Bypassing

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”.

Pipeline diagram showing hardware bypassing for RAW data hazards

\[
\begin{align*}
\text{addiu } r1, r2, 1 & \quad \begin{array}{c}
\text{F} \quad \text{D} \quad \text{X} \quad \text{M} \quad \text{W}
\end{array} \\
\text{addiu } r3, r1, 1 & \quad \begin{array}{c}
\text{F} \quad \text{D} \quad \text{X} \quad \text{M} \quad \text{W}
\end{array} \\
\text{addiu } r4, r3, 1 & \quad \begin{array}{c}
\text{F} \quad \text{D} \quad \text{X} \quad \text{M} \quad \text{W}
\end{array}
\end{align*}
\]
Adding single bypass path to support limited hardware bypassing

Deriving the bypass and stall signals

\[
\text{stall} = \text{stall}_w\text{addr}_X,\text{raddr0}_D \mid\mid \text{stall}_w\text{addr}_X,\text{raddr1}_D \\
\mid\mid \text{stall}_w\text{addr}_M,\text{raddr0}_D \mid\mid \text{stall}_w\text{addr}_M,\text{raddr1}_D \\
\mid\mid \text{stall}_w\text{addr}_W,\text{raddr0}_D \mid\mid \text{stall}_w\text{addr}_W,\text{raddr1}_D
\]

\[
\text{reg}_en_F = !\text{stall} \\
\text{reg}_en_D = !\text{stall} \\
\text{insert}_\text{nop}_D = \text{stall}
\]
Handling load-use RAW dependencies

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

```
lw r1, 0(r2)
  F → D → X → M → W
addiu r3, r1, 1
  F → D → X → M → W
```

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

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

```
stall_waddr_X_raddr0_D =
  ren0_D && wen_X && (raddr0_D == waddr_X) && (waddr_X != 0)
  && (op_X == lw)

bypass_waddr_X_raddr0_D =
  ren0_D && wen_X && (raddr0_D == waddr_X) && (waddr_X != 0)
  && (op_X != lw)

stall =           stall_waddr_X_raddr0_D || stall_waddr_X_raddr1_D
              || stall_waddr_M_raddr0_D || stall_waddr_M_raddr1_D
              || stall_waddr_W_raddr0_D || stall_waddr_W_raddr1_D

reg_en_F = !stall
reg_en_D = !stall
insert_nop_D = stall
```
Pipeline diagram showing multiple hardware bypass paths

```assembly
addiu r1, r2, 1
addiu r3, r4, 1
addiu r5, r3, 1
addu r6, r1, r3
sw r5, 0(r7)
jr r6
```

Adding all bypass path to support full hardware bypassing
2.4. 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]
```

Pipeline diagrams showing RAW dependency through memory
**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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
<td>Load r1 from memory at address 0(r2)</td>
</tr>
<tr>
<td>lw r3, 0(r4)</td>
<td>Load r3 from memory at address 0(r4)</td>
</tr>
<tr>
<td>addu r5, r1, r3</td>
<td>Add r1 and r3, store result in r5</td>
</tr>
<tr>
<td>sw r5, 0(r6)</td>
<td>Store r5 at address 0(r6)</td>
</tr>
<tr>
<td>addiu r2, r2, 4</td>
<td>Add immediate 4 to r2, store result in r2</td>
</tr>
<tr>
<td>addiu r4, r4, 4</td>
<td>Add immediate 4 to r4, store result in r4</td>
</tr>
<tr>
<td>addiu r6, r6, 4</td>
<td>Add immediate 4 to r6, store result in r6</td>
</tr>
<tr>
<td>addiu r7, r7, -1</td>
<td>Add immediate -1 to r7, store result in r7</td>
</tr>
<tr>
<td>bne r7, r0, loop</td>
<td>Branch if r7 does not equal r0, jump to 'loop'</td>
</tr>
</tbody>
</table>


3. 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.

```assembly
addiu r1, r2, 1
j foo # Jumps are always taken
addiu r3, r4, 1
foo: addiu r5, r6, 1
    bne r0, r0, bar # This branch is not taken
    addiu r7, r8, 1
bar: addiu r9, r10, 1
```

Using pipeline diagrams to illustrate control hazards

We use microarchitectural dependency arrows to illustrate control hazards on pipeline diagrams. Jump resolution latency is two cycles, and branch resolution latency is three cycles.

```
j foo F D X M W
addiu r5, r6, 1 F D X M W
bne r0, r0, bar F D X M W
addiu r7, r8, 1 F D X M W
```
addiu r1, r2, 1
j foo
addiu r5, r6, 1
bne r0, r0, bar
addiu r7, r8, 1
addiu r9, r10, 1

Approaches to resolving control hazards

• **Software Scheduling**: Programmer or compiler explicitly avoids scheduling instructions that would create control hazards

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

• **Hardware Stalling**: Hardware includes control logic that freezes later instructions until earlier instruction have finished determining the correct control flow

• **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
3.1. Software Scheduling

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.

```
addiu r1, r2, 1
j   foo
nop
addiu r3, r4, 1
foo: addiu r5, r6, 1
     bne r0, r0, bar
     nop
     nop
     addiu r7, r8, 1
bar:  addiu r9, r10, 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 software scheduling for control hazards

```
addiu r1, r2, 1
j   foo
nop
addiu r5, r6, 1
bne r0, r0, bar
nop
nop
addiu r7, r8, 1
addiu r9, r10, 1
```
3.2. Hardware Stalling

Although we could stall to resolve control hazards, the issue is that every instruction essentially creates a control hazard until we know if that instruction is a jump, branch, or other instruction. Stalling will seriously degrade the performance of the common case when we have a sequence of instructions that are not control flow instructions.

\[
\begin{align*}
\text{addiu } r1, r2, 1 \\
\text{j foo} \\
\text{addiu } r5, r6, 1 \\
\text{bne } r0, r0, \text{bar} \\
\text{addiu } r7, r8, 1 \\
\text{addiu } r9, r10, 1
\end{align*}
\]
3.3. 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, r2, 1
j foo
addiu r3, r4, 1
addiu r5, r6, 1
bne r0, r0, bar
addiu r7, r8, 1
addiu r9, r10, 1
```

**Pipeline diagram when branch is taken**

```
addiu r1, r2, 1
j foo
addiu r3, r4, 1
addiu r5, r6, 1
bne r0, r0, bar
addiu r7, r8, 1
addiu r9, r10, 1
addiu r9, r10, 1
```
Modifications to datapath/control to support hardware speculation

Deriving the squash signals

\[
\text{squash}_F = (\text{op}_D == j) \lor (\text{op}_D == \text{jal}) \lor (\text{op}_D == \text{jr}) \\
\lor \text{br\_taken}_X
\]

\[
\text{squash}_D = \text{br\_taken}_X
\]

\[
\text{reg\_en}_F = \neg\text{stall} \lor \text{squash}_F
\]

\[
\text{reg\_en}_D = \neg\text{stall} \lor \text{squash}_D
\]

\[
\text{insert\_nop}_D = \text{stall} \lor \text{squash}_D
\]

\[
\text{insert\_nop}_F = \text{squash}_F
\]
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 leverage both software scheduling and 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 in branch delay slot
addiu r6, r7, 1
...
foo:
    addu r8, r1, r4
    addiu r9, r1, 1
```

---

28
3.4. 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 code 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 code sequence

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

```
addiu EPC, EPC, 4
eret
syscall (excep)
opD
opE
opF
```
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 completed 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

\[
squash_F = (op_D == j) \lor (op_D == jal) \lor (op_D == jr) \\
\quad \lor \text{br_taken}_X \lor \text{exception}_M
\]

\[
squash_D = \text{br_taken}_X \lor \text{exception}_M
\]

\[
squash_X = \text{exception}_M
\]

\[
squash_M = \text{exception}_M
\]
Pipeline diagram of exception handling
4. 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 on pipeline diagrams.

```
addiu r1, r2, 1
addiu r3, r4, 1
lw   r5, 0(r6)
addiu r7, r8, 1
```

Approaches to resolving structural hazards

- **Software Scheduling:** Programmer or compiler explicitly avoids scheduling instructions that would create structural hazards
- **Hardware Stalling:** Hardware includes control logic that freezes later instructions until earlier instruction has finished using the shared resource
- **Hardware Duplication:** Add more hardware so that each instruction can access separate resources at the same time
4.1. Software Scheduling

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

```
addiu r1, r2, 1
addiu r3, r4, 1
lw  r5, 0(r6)
nop
addiu r7, r8, 1
```

Pipeline diagram showing software scheduling for structural hazards

```
<table>
<thead>
<tr>
<th>addiu r1, r2, 1</th>
</tr>
</thead>
<tbody>
<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>
```
4.2. 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

```
addiu r1, r2, 1
addiu r3, r4, 1
lw r5, 0(r6)
addiu r7, r8, 1
```

Deriving the stall signal

```
stall_wport_hazard_D = (op_X == lw)
    && (   (op_D == ADDU) || (op_D == ADDIU)
                || (op_D == MUL) || (op_D == JAL) )

stall =专卖waddr_X_raddr0_D ||专卖waddr_X_raddr1_D
    ||专卖waddr_M_raddr0_D ||专卖waddr_M_raddr1_D
    ||专卖waddr_W_raddr0_D ||专卖waddr_W_raddr1_D
    ||专卖wport_hazard_D
```

Notice that we stall far before the point when the structural hazard actually occurs, because we know exactly how instructions move down the pipeline. Also possible to use dynamic arbitration in the back of the pipeline.
4.3. 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
```
5. **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, r1, 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.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mul</td>
<td>r1, r2, r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>r4, r1, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>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

5.1. **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.

```plaintext
mul r1, r2, r3
addiu r4, r1, 1
addiu r6, r5, 1
```
5.2. 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, r1, 1
addiu r1, r5, 1
```

Deriving the stall signal

\[
\text{stall\_struct\_hazard\_D} = \\
\quad (\text{op\_D} == \text{MUL}) \&\& \neg \text{imul\_rdy\_D}
\]

\[
\text{stall\_waw\_hazard\_D} = \\
\quad \text{wen\_D} \&\& \text{wen\_Z} \&\& (\text{waddr\_D} == \text{waddr\_Z}) \&\& (\text{waddr\_Z} \neq 0)
\]

\[
\text{stall} = \quad \text{stall\_waddr\_X\_raddr0\_D} \mid \mid \text{stall\_waddr\_X\_raddr1\_D} \\
\mid \mid \text{stall\_waddr\_M\_raddr0\_D} \mid \mid \text{stall\_waddr\_M\_raddr1\_D} \\
\mid \mid \text{stall\_waddr\_W\_raddr0\_D} \mid \mid \text{stall\_waddr\_W\_raddr1\_D} \\
\mid \mid \text{stall\_struct\_hazard\_D} \\
\mid \mid \text{stall\_waw\_hazard\_D}
\]
6. Analyzing Processor 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 = 1\(\tau\)
- register write = 1\(\tau\)
- regfile read = 10\(\tau\)
- regfile write = 10\(\tau\)
- memory read = 20\(\tau\)
- memory write = 20\(\tau\)
- +4 unit = 4\(\tau\)
- sext unit = 1\(\tau\)
- br_tgen = 8\(\tau\)
- j_tgen = 1\(\tau\)
- mux = 3\(\tau\)
- multiplier = 20\(\tau\)
- alu = 10\(\tau\)
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?

```
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.

```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
```
7. Case Study: 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

```assembly
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:
```

One-Instr Load-Use Delay Slot

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

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

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 mm$^2$