In-Order Dual-Issue Superscalar TinyRV1 Processor

Superscalar Pipeline Hazards
   2.1. RAW Hazards
   2.2. Control Hazards
   2.3. Structural Hazards
   2.4. WAW and WAR Name Hazards

Analyzing Performance of Superscalar Processors
1. In-Order Dual-Issue Superscalar TinyRV1 Processor

- Processors studied so far are fundamentally limited to CPI $\geq 1$
- Superscalar processors enable CPI $< 1$ (i.e., IPC $> 1$) by executing multiple instructions in parallel
- Can have both in-order and out-of-order superscalar processors, but we will start by exploring in-order

- Continue to assume combinational memories
- **F Stage**: fetch two instructions at once
- **D Stage**: 4 read ports, decode 2 inst, “issue” inst to correct pipe
- **X/M Stage**: separate into A and B pipes (see next page)
- **W Stage**: 2 write ports
More abstract way to illustrate same dual-issue superscalar pipeline

Different instructions use the A-pipe and/or the B-pipe

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>addi</th>
<th>mul</th>
<th>lw</th>
<th>sw</th>
<th>jal</th>
<th>jr</th>
<th>bne</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-Pipe</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>B-Pipe</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Example pipeline diagram for dual-issue superscalar processor

- Multiple instructions in stages F, D, W allowed because superscalar processor has duplicated hardware to avoid structural hazards
- **Fetch Block** – group of instructions fetched as unit
- **Swizzle** – instructions “swapped” from natural fetch position to appropriate execution pipe
2. Superscalar Pipeline Hazards

Seems so easy, but why is pipelining hard?

- RAW Hazards
- Control Hazards
- Structural Hazards
- WAR/WAR Name Hazards

2.1. RAW Hazards

Let’s first assume we only use stalling to resolve RAW hazards

```
addi x1, x2, 1
addi x3, x4, 1
add  x5, x1, x3
addi x6, x5, 1
addi x7, x8, 1
addi x9, x8, 1
```

A fully-bypassed superscalar processor is possible, but expensive
Revisit previous assembly sequence with full bypassing

<table>
<thead>
<tr>
<th>addi  x1, x2, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi  x3, x4, 1</td>
</tr>
<tr>
<td>add   x5, x1, x3</td>
</tr>
<tr>
<td>addi  x6, x5, 1</td>
</tr>
<tr>
<td>addi  x7, x8, 1</td>
</tr>
<tr>
<td>addi  x9, x8, 1</td>
</tr>
</tbody>
</table>

Activity: Draw a pipeline diagram for following instruction sequence. Include all microarchitectural dependency arrows.

<table>
<thead>
<tr>
<th>addi  x1, x2, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw   x3, 0(x4)</td>
</tr>
<tr>
<td>lw   x5, 0(x3)</td>
</tr>
<tr>
<td>addi  x6, x7, 1</td>
</tr>
<tr>
<td>addi  x8, x5, 1</td>
</tr>
<tr>
<td>addi  x9, x8, 1</td>
</tr>
</tbody>
</table>
2.2. Control Hazards

Consider following two static instruction sequences.

1. 0x1000 addi x1, x2, 1
2. 0x1004 jal x0, foo
3. ...
4. foo:
5. 0x2000 addi x3, x4, 1
6. 0x2004 addi x5, x6, 1

1. # assume R[x1] != R[x2]
2. 0x1000 bne x1, x2, foo
3. ...
4. foo:
5. 0x2000 addi x3, x4, 1
6. 0x2004 addi x5, x6, 1

Pipeline diagram for left sequence. Jumps are resolved in D stage.

Pipeline diagram for right sequence. Branches are resolved in A0 stage.
Unaligned fetch blocks

Consider the following static instruction sequence

1. 0x000 opA
2. 0x004 opB
3. 0x008 opC
4. 0x00c jal x0, 0x100
5. ...
6. 0x100 opD
7. 0x104 jal x0, 0x204
8. ...
9. 0x204 opE
10. 0x208 jal x0, 0x30c
11. ...
12. 0x30c opF
13. 0x310 opG
14. 0x314 opH

Layout of fetch blocks in instruction cache. Numbers indicate which instructions belong to which fetch block.

- Unaligned fetch blocks within a cache line are challenging
- Unaligned fetch blocks across cache lines are very challenging
2. Superscalar Pipeline Hazards

2.2. Control Hazards

**Aligned fetch blocks**

Only fetch aligned fetch blocks, possibly discarding first instruction. Reconsider the same static instruction sequence

1. 0x000 opA
2. 0x004 opB
3. 0x008 opC
4. 0x00c jal x0, 0x100
5. ...
6. 0x100 opD
7. 0x104 jal x0, 0x204
8. ...
9. 0x204 opE
10. 0x208 jal x0, 0x30c
11. ...
12. 0x30c opF
13. 0x310 opG
14. 0x314 opH

Layout of fetch blocks in instruction cache. Numbers indicate which instructions belong to which fetch block.

![Diagram of instruction cache layout with fetch blocks labeled with hexadecimal addresses and corresponding operations.](image-url)
Supporting precise exceptions

Consider following instruction sequence. Assume commit point is in the A1/B1 stage and the xxx instruction causes an illegal instruction exception originating in the D stage.

1. add  x1, x2, x3
2. xxx  # causes illegal instruction exception
3. addi x4, x5, 1
4. addi x6, x7, 1
5. ...
6. exception_handler:
7. opX
8. opY
9. opZ

What if add caused an arithmetic overflow exception?
2.3. Structural Hazards

Structural hazards *are not* possible in the canonical single-issue TinyRV1 pipeline, but structural hazards *are* possible in the canonical dual-issue TinyRV1 pipeline if two instructions in the same fetch block want to use the same pipe.

```
mul x1, x2, x3
mul x4, x5, x6
lw  x7, 0(x8)
sw  x9, 0(x10)
```

2.4. WAW and WAR Name Hazards

WAW name hazards *are not* possible in the canonical single-issue TinyRV1 pipeline, but WAW name hazards *are* possible in the canonical dual-issue TinyRV1 pipeline if two instructions in the same fetch block write the same register.

```
addi x1, x2, 1
addi x1, x3, 1
```

WAR name hazards *are not* possible in the canonical single-issue TinyRV1 pipeline. Are WAR name hazards possible in the canonical dual-issue TinyRV1 pipeline?

```
addi x1, x2, 1
addi x2, x3, 1
```
3. Analyzing Performance of Superscalar Processors

Consider the classic vector-vector add loop over arrays with 64 elements. This loop has a CPI of 1.33 on the canonical single-issue TinyRV1 processor. What is the CPI on the canonical dual-issue TinyRV1 processor?

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