Exceptions in a Pipeline

- Consider arithmetic overflow exception causing immediate exception handling (like Chapter 5)

- Occurs when offending instruction is in EX stage

- set PC to exception handler

- flush all instructions in earlier pipeline stages (IF and ID in this case), i.e., instructions that followed the offending instruction into the pipeline
• flush offending instruction in EX

• record cause as overflow

• save $PC + 4$ in EPC register

• instructions in later stages of pipeline are unaltered
40X  sub  $11,$2,$4
44X  and  $12,$2,$5
48X  or  $13,$2,$5
4CX  add  $1, $2,$1
50X  slt  $15,$6,$7
54X  lw  $16,50($7)

Suppose add causes overflow and first two instructions of exception handler are

40000040X  sw  $25,1000($0)
40000044X  sw  $26,1004($0)
insert figure 6.56
Actions During Cycle In Which add is in EX Stage

- Figure 6.56 shows signals near the end of the cycle

- during the earlier part of the cycle
  - lw is fetched
  - slt is decoded and control unit determines its control settings
  - add in EX combinatorially computes addition
• after overflow is signalled

  – overflow signal is **combinationally fed back** to control unit

  – control unit updates control signals during the same cycle they were computed for slt in its ID stage

  – other lines settle in response to these new control signals into values shown in figure and wait for the clock edge
New control signals are:

- **IF.flush, ID.flush, EX.flush** are asserted that turn lw,slt and add into nops for latching into next stage of the pipeline at the next clock edge

- **EPC write control line** set to latch in address of slt (PC of add +4) on next clock edge

- **inputs to PC set to 40000040X** to latch in on next clock edge
- multiple exceptions can occur

- precise exceptions keep a particular order and associate the exception with the offending instruction

- suppose you have instruction A followed by instruction B

- A causes a data memory fault exception

- B causes an instruction memory fault exception
<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>$ME^{exc}$</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>$IF^{exc}$</td>
<td>ID</td>
<td>EX</td>
<td>$ME^{exc}$</td>
<td>WB</td>
</tr>
</tbody>
</table>

- B causes exception first on cycle 2
- A causes exception on cycle 4
- on cycle 2
  - record $CAUSE = B$ as part of the data to be placed into ID/EX register on next clock edge
  - let control unit flush instruction in IF and set B to a nop that has exception information recorded
  - DO NOT set up branch to exception handler
• on cycle 4

  – A causes exception

  – record $CAUSE = A$ as part of the data to be placed into MEM/WB register on next clock edge

  – let control unit flush instruction in IF, ID, and EX set A to a nop that has exception information recorded

  – DO NOT set up branch to exception handler
• as instructions enter the WB stage they are checked to see if they are nops carrying exception information

• service exceptions in the order they enter the WB stage, i.e., the order the instructions were issued

• it is still feasible to have an overriding class of exceptions that would require service immediately as before

• some times imprecise exceptions are used to simplify design and to mitigate performance loss
insert figure 6.65 showing the complete pipeline
Performance Enhancements

- superpipelining – more stages to pipe (deeper pipeline)

- superscalar – more instructions enter pipeline on each cycle (wider pipeline)

- dynamic pipelining – if a stall is encountered for an instruction consider following instructions and issue those that are able to enter pipeline
longer pipelines:

- more registers
- more potential for hazards
- more HW for stalls/forwarding/exceptions/branch handling
- shorter cycle time
- more instructions per cycle $\rightarrow$ smaller CPI
• superscalar:
  – more complicated logic to issue instructions
  – more HW in pipeline to handle multiple instructions in each stage
  – more simultaneous accesses to data/instruction memory
  – more instructions per cycle → smaller CPI
Superscalar Processing

- Type and number of instructions that can be issued on each cycle are usually restricted
- we assume one ALU and one Memory instruction
- FP ALU, Integer ALU, Memory, and Branch may also be used on some machines
- sometimes more than one of a particular type can be used, e.g., two loads and a store, multiple FP ALU and multiple Integer ALU are often useful for scientific programs
Ideal execution profile for two way superscalar

<table>
<thead>
<tr>
<th>ALU</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>ME</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>ALU</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>MEM</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>ALU</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
<tr>
<td>MEM</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
</tr>
</tbody>
</table>

To get such performance often involves significant coding transformations
loop:    lw  $t0, 0($s1)

addu $t0,$t0,$s2

sw  $t0,0($s1)

addi $s1,$s1,-4

bne  $s1,$0,loop

<table>
<thead>
<tr>
<th>label</th>
<th>ALU/Branch</th>
<th>Memory</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>—</td>
<td>LW</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>2</td>
</tr>
<tr>
<td>addu</td>
<td>—</td>
<td>—</td>
<td>3</td>
</tr>
<tr>
<td>addi</td>
<td>sw</td>
<td>—</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>5</td>
</tr>
<tr>
<td>bne</td>
<td>—</td>
<td>—</td>
<td>6</td>
</tr>
</tbody>
</table>
• addu and lw have dependence and cannot issue together

• addu must stall one extra cycle due to load producing data in MEM and addu needing it in EX.

• addi and sw have an output dependence but both read $1$ in ID stage without problem so they can issue together (although a conservative HW implementation might not allow it)

• bne requires the output of addi in the ID stage (assuming ID branch resolution) so a stall is required

• 6 cycles to do 5 instructions per iteration (assuming perfect BTB prediction)
• We have two superscalar stall slots to fill.

• addi can be moved to fill slot after lw.

• address in sw must be changed since $s1$ when it executes will have been decremented by 4 via addi.

• bne is independent of sw so they can issue together

• stalls removed 4 cycles for 5 instructions — $CPI = 0.8$ not the desired 0.5
loop:   lw  $t0, 0($s1)

        addi  $s1,$s1,-4

        addu  $t0,$t0,$s2

        sw  $t0,4($s1)

        bne  $s1,$0,loop

<table>
<thead>
<tr>
<th>label</th>
<th>ALU/Branch</th>
<th>Memory</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>–</td>
<td>lw</td>
<td>1</td>
</tr>
<tr>
<td>addi</td>
<td>–</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>addu</td>
<td>–</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>bne</td>
<td>sw</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
- the addi could have gone in parallel with lw but stall would still be there

- we still have empty slots in schedule

- need more instructions that are independent

- use a code transformation called unrolling

- reduces loop overhead (good)

- more parallelism potentially (good)

- larger iteration body (maybe bad)

- more registers (bad, but it is required)
• assume unrolling factor is 4

• duplicate code

• remove branches except last
loop: lw $t0, 0($s1)
           addu $t0,$t0,$s2
           sw $t0,0($s1)
           addi $s1,$s1,-4

lw $t0, 0($s1)
           addu $t0,$t0,$s2
           sw $t0,0($s1)
           addi $s1,$s1,-4

lw $t0, 0($s1)
           addu $t0,$t0,$s2
           sw $t0,0($s1)
           addi $s1,$s1,-4

lw $t0, 0($s1)
           addu $t0,$t0,$s2
           sw $t0,0($s1)
           addi $s1,$s1,-4
           bne $s1,$0,loop
• adjust offsets and move all addi to top

• every address (lw and sw) dependent on $s1 must be incremented by 4 when an addi moves above it (below is code after moving 2 addi)

```
loop:
lw $t0, 0($s1)
addi $s1,$s1,-4
addi $s1,$s1,-4
addu $t0,$t0,$s2
sw $t0,8($s1)

lw $t0, 4($s1)
addu $t0,$t0,$s2
sw $t0,4($s1)

lw $t0, 0($s1)
addu $t0,$t0,$s2
sw $t0,0($s1)
addi $s1,$s1,-4
lw $t0, 0($s1)
addu $t0,$t0,$s2
sw $t0,0($s1)
addi $s1,$s1,-4
bne $s1,$0,loop
```
• addi can be combined

• note 4 different independent groups

• they do not all need to use $t0$ as the target

loop:  lw  $t0, 0($s1)
   addi $s1,$s1,-4
   addi $s1,$s1,-4
   addi $s1,$s1,-4
   addi $s1,$s1,-4
   
   addu $t0,$t0,$s2
   sw  $t0,16($s1)

   lw  $t0, 12($s1)
   addu $t0,$t0,$s2
   sw  $t0,12($s1)

   lw  $t0, 8($s1)
   addu $t0,$t0,$s2
   sw  $t0,8($s1)

   lw  $t0, 4($s1)
   addu $t0,$t0,$s2
   sw  $t0,4($s1)
   bne  $s1,$0,loop
- Rename registers to remove false dependences
- load - use stalls still a problem
- determine final code by using a greedy algorithm to schedule slots

```
loop:    lw  $t0, 0($s1)
         addi $s1,$s1,-16
         addu $t0,$t0,$s2
         sw  $t0,16($s1)

         lw  $t1, 12($s1)
         addu $t1,$t1,$s2
         sw  $t1,12($s1)

         lw  $t2, 8($s1)
         addu $t2,$t2,$s2
         sw  $t2,8($s1)

         lw  $t3, 4($s1)
         addu $t3,$t3,$s2
         sw  $t3,4($s1)

         bne  $s1,$0,loop
```
• the load of $t0$ and addi can issue first

• the earliest an ALU can execute is cycle 3 due to load-use stall and no other data available

• load of $t1$ can issue in cycle 2

• the load of $t2$ can issue in cycle 3

<table>
<thead>
<tr>
<th>label</th>
<th>ALU/Branch</th>
<th>Memory</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>addi</td>
<td>lw0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>lw1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu0</td>
<td>lw2</td>
<td>3</td>
</tr>
</tbody>
</table>
• the data from load of $t1$ can be consumed in cycle 4

• the load of $t3$ can issue in cycle 4

• the data from load of $t2$ can be consumed in cycle 5

<table>
<thead>
<tr>
<th>label</th>
<th>ALU/Branch</th>
<th>Memory</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>addi</td>
<td>lw0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw1</td>
<td>2</td>
</tr>
<tr>
<td>addu0</td>
<td></td>
<td>lw2</td>
<td>3</td>
</tr>
<tr>
<td>addu1</td>
<td></td>
<td>lw3</td>
<td>4</td>
</tr>
<tr>
<td>addu2</td>
<td></td>
<td>-</td>
<td>5</td>
</tr>
</tbody>
</table>
- the data from load of $t3$ can be consumed in cycle 6
- no loads left but $t0$ can be stored
- the pattern completes simply

<table>
<thead>
<tr>
<th>label</th>
<th>ALU/Branch</th>
<th>Memory</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>addi</td>
<td>lw0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw1</td>
<td>2</td>
</tr>
<tr>
<td>addu0</td>
<td></td>
<td>lw2</td>
<td>3</td>
</tr>
<tr>
<td>addu1</td>
<td></td>
<td>lw3</td>
<td>4</td>
</tr>
<tr>
<td>addu2</td>
<td></td>
<td>sw0</td>
<td>5</td>
</tr>
<tr>
<td>addu3</td>
<td></td>
<td>sw1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw2</td>
<td>7</td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td>sw3</td>
<td>8</td>
</tr>
</tbody>
</table>
• note that the data memory port is saturated

• computation is memory bound so we cannot do any better

• 8 cycles 14 instructions $CPI = 0.57$

• extra registers needed

• improvement due to reduced loop overhead

• improvement due to superscalar in 6 out of 8 cycles