Pipelining

CS 365 Lecture 12
Prof. Yih Huang

Traditional Execution

1 2 3 4 1 2 3 4 5 1 2 3

add    ld    beq
Pipelined Execution

Basic Ideas

- Do not wait for an instruction to complete to start the next.
  - Start the Cycle 0 of the next instruction when the previous one enters Cycle 1.
- Instruction executions are overlapped.
- Pipelining increases *instruction throughput*, as opposed to decreasing the execution time of individual instructions.
Easier Said Than Done?

- In every cycle, activities of all five stages take place.
- Many problems arises with overlapped executions.
  - Structural hazards
  - Control hazards
  - Data hazards

Pipeline Hazards I

- **Structural Hazards**
  - The data path cannot support the combination of instructions that we want to execute in the same cycle
- Consider what happens when an R-type instruction is followed by a BEQ?

```
add if rr + rw
beq if rr -
```
Summary of MIPS Lite Instruction Executions

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td>A = Reg[IR[25-21]]</td>
<td>B = Reg[IR[20-16]]</td>
<td>ALUOut = PC + (sign-extend [IR[15-0]] &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend [IR[15-0]]</td>
<td>if (A = B) then PC = ALUOut</td>
<td>PC = PC[31-28] II [IR[25-0] &lt;&lt; 2]</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg[IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory[ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resource Conflicts on the Multicycle Datapath
Pipeline Hazards II

Control Hazards: When we decide to branch, other instructions are in the pipeline!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq</td>
<td>IF RR -</td>
</tr>
<tr>
<td>add</td>
<td>IF RR + RW</td>
</tr>
<tr>
<td>sub</td>
<td>IF RR + RW</td>
</tr>
</tbody>
</table>

Target: ld

Which one is the next?

Pipeline Hazards III

Data Hazards: (data dependencies)
an instruction depends on the result of a previous instruction still in the pipeline.

Add r1, r2, r3

Sub r4, r1, r10

Writing new value of r1

Reading new value of r1, not available yet
Lessons

- To achieve pipelining and avoid hazards, we need to redesign
  - the datapath and
  - instruction execution steps,

Pipeline Stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>R-Type</th>
<th>LD</th>
<th>ST</th>
<th>BEQ</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1:</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IR ← Mem[PC]</td>
<td>PC ← PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 2:</td>
<td>RR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read Reg[rs] and Reg[rt]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 3:</td>
<td>EX</td>
<td>RS op RT</td>
<td>Calculate RS+Immd</td>
<td>Calculate RS+Immd</td>
<td>Calculate PC + Immd Compare RS and RT</td>
</tr>
<tr>
<td>(use ALU)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 4:</td>
<td>DM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read memory</td>
<td>Write memory</td>
<td>Set PC accordingly</td>
<td></td>
</tr>
<tr>
<td>Stage 5:</td>
<td>RW/WB</td>
<td>Write to Rd</td>
<td>Write to Rt</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graphically Representing Pipelines

Solving Structural Hazards: Pipelined Datapath
Discussions

- Make sure you understand why two extra adders are added.

- Is the assumption of using separate instruction and data memory reasonable?

Consider Control Hazard

When is the decision made?
Solution 1: Stalling The Pipeline

IF  RR  EX  DM  RW
OR
IF  RR  EX  DM  RW
IF  RR  EX  DM  RW
IF  RR  EX  DM  RW
Decision is in

Solution 2: Branch Prediction

- Make a guess about the branch decision and start execute the guessed path before the decision is in (aka speculative execution).
- If guess was wrong, abandon those in the pipeline and jump to the right target.

- Branch Prediction Strategies
  - Predict branch fails
  - Predict branch succeeds
  - Look into history
Predict Branch Fails

- We guess the branch will fail. That is, the next IF will fetch the next sequential execution.
- If guess is right, just proceed.
- If guess is wrong, abandon sequential instructions and fetch the instruction from the target address.

Predict branch fails: Guess Is Right

beq IF RR EX DM RW
IF RR EX DM RW
IF RR EX DM RW
IF RR EX DM RW
IF RR EX DM RW
IF RR EX DM RW

Decision is in; Do not branch
Discussion

- Notice that programmers are not aware of branch predictions; right or wrong guesses affect only performance.

- **Delayed Branches**: Make it official that CPU always executes the instr following a branch. The branch determines the next next instruction.
  - Notice the programmer awareness
Example

```
BEQ r1, r2, target
    add r10, r11, r12
    add r20, r21, r22
    sub r30, r10, r20
    
    Delay slot, executed regardless of the branch decision

Target: sub r30, r10, r20
```

Delayed Branch with Wrong Guess

```
beq
    IF RR EX DM RW
    IF RR EX DM RW
    IF RR EX DM RW
    IF RR EX DM RW
    IF RR EX DM RW

Decision is in: Do branch
```

Delay slot is always finished
Discussions

- Compilers/programmers must be smart enough to make good use of the delay slots.
- The problem is not entirely solved: We still need to stall before the decision comes in.
- The situations are exacerbated by deeper pipelining.
- Branch predictions are still important.

Smart Branch Predictions

- Observations
  - Branches of if-else statements are hard to predict.
  - Branches of loops typically repeat previous decisions.
  - For performance, loops are more important than other control structures.
- Many modern processors use special-purpose hardware to remember the targets of recent branch instructions.
Recall Data Hazards

Sub $2, $1, $3
And $12, $2, $5
Or $13, $6, $2
Add $14, $2, $2
Sw $15, 100($2)

$2 available

Solution 1: Stalling

- Postpone subsequent instructions until data is available
- Simple but inefficient
**Solution 2: Internal Forwarding**

- New value of $2$ is available after the EX of Sub, but not in $2$ yet
- Use special circuits to forward the new value to subsequent instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source Registers</th>
<th>Destination Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub</td>
<td>r2, r1, r3</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>r12, r2, r5</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>r13, r6, r2</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>r14, r2, r2</td>
<td></td>
</tr>
<tr>
<td>Sw</td>
<td>r15, 100(r2)</td>
<td></td>
</tr>
</tbody>
</table>

**Types of Forwarding**

- **ALU forwarding**: forward an ALU output to subsequent instructions
  - This is the case we have seen

- **Memory Forwarding**: forward a memory output (ld result) to subsequent instructions.
Memory Forwarding

Ld  $1, 4($2)  \[ld \text{ RR EX DM RW}\]
Or  $10, $1, $3  \[Or \text{ RR EX RW}\]
Sub $20, $1, $10  \[Sub \text{ RR EX RW}\]

Notice that we still have to stall for one cycle

Delayed Loads

- Make it official in the ISA that the result of a load will not be available to the next instruction.
- Have the compiler find something useful to do in load slots. (by reordering)
MIPS Solution

- Delayed result with internal forwarding.

```
Ld  RR  EX  DM  RW

Or  RR  EX  RW

Sub RR  EX  RW
```

Nop or something useful

Summary

- Three types of pipeline hazards
  - Structural hazards
  - Control hazards
  - Data hazards
- Compilers/programmers could reorder the code to avoid hazards and eliminate bubbles.
- Ideal cases: no bubbles; one instr per cycle.
- Stalling (bubbles) is the last resort but must be supported by hardware.
Exercise: Code Reordering

for (i=0; i<N; i++)
    z[i] = x[i] + y[i];

Registers
- r1 points to x[i]  - r4 holds x[i]
- r2 points to y[i]  - r5 holds y[i]
- r3 points to z[i]  - r6 holds z[i]
- r7 holds i
- r8 holds N

loop: lw r4, 0(r1)

    lw r5, 0(r2)

    add r6, r4, r5
    sw r6, 0(r3)
    addi r1,r1,4
    addi r2,r2,4
    addi r3,r3,4
    addi r7,r7,1
    bne r7,r8, loop
Loop Unrolling

```
loop:  lw    r4, 0(r1)
        lw    r5, 0(r2)
        add   r6, r4, r5
        sw    r6, 0(r3)
        addi  r7, r7, 1
        beq   r7, r8, exit
        lw    r4, 4(r1)
        lw    r5, 4(r2)
        add   r6, r4, r5
        sw    r6, 4(r3)
        addi  r1, r1, 8
        addi  r2, r2, 8
        addi  r3, r3, 8
        addi  r7, r7, 1
        bne   r7, r8, loop

exit:
```