COMPUTER ARCHITECTURE

PIPELINING TECHNIQUES

Dr. Bill Yi
Santa Clara University

(Based on text: David A. Patterson & John L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd Ed., Morgan Kaufmann, 2007)
(Also based on presentation: Dr. Nam Ling, COEN210 Lecture Notes)
COURSE CONTENTS

- Introduction
- Instructions
- Computer Arithmetic
- Performance
- Processor: Datapth
- Processor: Control
- Pipelining Techniques
- Memory
- Input/Output Devices
PIPELINING & ADVANCED TECHNIQUES

- Pipeline Overview & Hazards
- Pipelined Datapath
- Pipelined Control
- Data Hazards, Forwarding, Stalls
- Control Hazards & Exceptions
**Pipelining: Basic Idea**

- **Pipelining:** Multiple instructions **overlapped** in execution
- Improve performance by increasing instruction throughput

*Ideal speedup is number of stages in the pipeline. Do we achieve this?*
Pipelining: Basic Idea

- Improve **instruction throughput** rather than **individual instruction execution time or latency**
- Ideal speedup

\[
\frac{\text{Time between instructions (non-pipelined)}}{\text{Time between instructions (pipelined)}} = \# \text{ of pipe stages}
\]

- Pipeline stage: balancing length of each stage with equal length, limited \# of pipe stages
- All instructions in pipeline take the same number of clock cycles
  - So, there are no operation in some stages sometimes
Pipelining

What makes it easy?
- All instructions are the same length (e.g. MIPS)
- Just a few instruction formats (e.g. MIPS)
- Memory operands appear only in loads and stores (e.g. MIPS)
- Operands aligned in memory (e.g. MIPS)

What makes it hard?
- structural hazards
- control hazards
- data hazards
- exception handling
- trying to improve performance with out-of-order execution, etc.
Three Pipeline Hazards

- **Hazards**: situations in pipelining when next instruction cannot execute in the following clock cycle

- **Structural hazards** -- caused by hardware resource conflicts; hardware cannot support the combination of instructions we want to execute in the same clock cycle.
  - Ex. If we have only one memory for both instruction & data, we have a structural hazard; 
    ----> need two separate memories, one for instruction & one for data

- **Control hazards** -- caused by the need to make a decision based on the results of one instruction while others are executing
  - Ex. branch / jump instructions
    - may cause a *pipeline stall (bubble)*: next instruction is stalled extra clock cycle(s) before starting
    - *branch prediction* (may be complicated): execute the predicted instruction after branch without delay; when guess is wrong, ensure that wrongly guessed branch has no effect and restart the pipeline from proper branch address
    - *delayed decision (branch)*: place an instruction that is not affected by the branch in the delayed branch slot
      \[
      \text{add} \; $4, \; $5, \; $6 \quad \text{beq} \; $1, \; $2, \; 40 \quad \text{add} \; $4, \; $5, \; $6 \quad \text{(in delayed branch slot)}
      \]
Three Pipeline Hazards

- **Data hazards** -- caused by data dependence; an instruction depends on the results of a previous instruction still in the pipeline

- Read after write dependence (RAW)
  I1: \( R2 \leftarrow R1 + R3 \)
  I2: \( R4 \leftarrow R2 + R3 \)

- Write after read dependence (WAR), a problem with concurrent execution
  I1: \( R1 \leftarrow R2 + R3 \)
  I2: \( R3 \leftarrow R4 \times R5 \)

- Write after write dependence (WAW), a problem with concurrent execution
  I1: \( R2 \leftarrow R1 + R3 \)
  I2: \( R2 \leftarrow R4 \times R7 \)

- Data dependence does not cause any problem in non-overlapped sequential execution
- Data dependence may cause bubbles in the pipe, due to waiting for results from earlier instruction
Data Dependence
(Data Hazard)

Problem: starting next instruction before the current is finished

Possible solutions include:

- compiler (software) approach 1: insert “nop” operations, etc. ---> to ensure correct handling of stalls
- compiler (software) approach 2: reordering of code to avoid pipeline stalls
- hardware approach: forwarding, getting missing items early from internal resources
We use MIPS subset: lw, sw, add, sub, and, or, slt, & beq and a simplified version of its pipeline

- A five-stage pipeline:
  - **IF**: instruction fetch
  - **ID**: instruction decode and register file read
  - **EX**: execution or address calculation
  - **MEM**: data memory access
  - **WB**: write back to register

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>
What do we need to add to actually split the datapath into stages?
Pipelined Datapath

- Add buffers (**pipeline registers**) between pipeline stages
- Any information needed in a later stage must be passed to that stage via a pipeline register
- Any problem with register write back?
Pipelined Datapath: Some Considerations

- To pass information from a pipeline stage to a later stage, the information must be placed in a pipeline register; otherwise, the information is lost when the next instruction enters that pipeline stage. Pipeline registers are: IF/ID, ID/EX, EX/MEM, MEM/WB

- Each logical component (instruction memory, register read ports, ALU, data memory, register write port) can only be used within a single pipeline stage (otherwise, structural hazard)

- The ID stage reads register file and the WB stage writes register file: only one register file is needed if both read and write can be performed correctly in the same cycle. We assume that write is in the first half of clock cycle and read in the second half

- We need to preserve the destination write register number for load instruction to be used at the last stage: rt number must be passed to each stage pipeline register

- Similarly, destination rd number must be passed to each pipeline register for R-type instructions
Corrected Pipelined Datapath

- Saving rd/rt register number
Graphically Representing Pipelines

Can help with answering questions like:
- how many cycles does it take to execute this code?
- what is the ALU doing during cycle 4?
- use this representation to help understand datapaths
The Original Control Signals
Assumption:
- PC written every cycle, no separate PC write signal needed
- Pipeline registers written every cycle, no separate pipeline register write signals needed

We have 5 stages. What needs to be controlled in each pipeline stage?
- *Instruction Fetch and PC Increment*: common to all instructions, hence no optional control lines to set
- *Instruction Decode / Register File Read*: common to all instructions, hence no optional control lines to set
- *Execution / address calculation*: instruction dependent; control signals include RegDst, ALUOp, ALUSrc
- *Memory Stage*: instruction dependent; control signals include Branch, MemRead, MemWrite; these are set by beq, lw, and sw
- *Write Back*: instruction dependent; control signals include MemtoReg and RegWrite

Control lines start with the EX stage, we can create the control lines during instruction decode; control signals are then *used and passed down from stage to stage*
Pipeline Control

- Pass control signals along just like the data

### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R-format</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Pipeline Stages

- **IF/ID**: Instruction Fetch/Instruction Decode
- **ID/EX**: Instruction IDentification/Instruction Execute
- **EX/MEM**: Execution/Memory Access
- **MEM/ WB**: Memory Write-Back

### Control Signals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>Write-back stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst ALU Op1 ALU Op0 ALU Src</td>
<td>Branc Mem Read Mem Write</td>
<td>Reg write Mem to Reg</td>
</tr>
<tr>
<td>R-format</td>
<td>1 1 0 0</td>
<td>0 0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>lw</td>
<td>0 0 0 1</td>
<td>0 1 0</td>
<td>1 1</td>
</tr>
<tr>
<td>sw</td>
<td>X 0 0 1</td>
<td>0 0 1</td>
<td>0 X</td>
</tr>
<tr>
<td>beq</td>
<td>X 0 1 0</td>
<td>1 0 0</td>
<td>0 X</td>
</tr>
</tbody>
</table>
Pipelined Datapath with Control
Problem with starting next instruction before first is finished
- dependencies that “go backward in time” are data hazards

<table>
<thead>
<tr>
<th>Value of register $2:</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)
- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
Software (Compiler)
Solution 1

- Have compiler guarantee no hazards
- Method 1: Compiler inserts “nop”

```
sub    $2, $1, $3
nop
nop
nop
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Problem: this really slows us down!
Software (Compiler) 
Solution 2

- Method 2: Move independent instructions around to eliminate data hazards and fill in bubbles (scheduling)

I1    and $18, $9, $10
I2    sub $2, $1, $3
I3    and $12, $2, $5
I4    or $13, $6, $2
I5    add $14, $2, $2
I6    sw $15, 100($2)
I7    sub $16, $7, $8
I8    add $17, $8, $9

Becomes

I1    and $18, $9, $10
I2    sub $2, $1, $3
I7    sub $16, $7, $8
I8    add $17, $8, $9
I3    and $12, $2, $5
I4    or $13, $6, $2
I5    add $14, $2, $2
I6    sw $15, 100($2)
Use temporary results, don’t wait for them to be written
- ALU forwarding: ALU output flows in pipe registers and forwarded to ALU input
- Register file forwarding to handle read/write to same register in same cycle

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
<tr>
<td>Value of EX/MEM</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)
- sub $2$, $1$, $3$
- and $12$, $2$, $5$
- or $13$, $6$, $2$
- add $14$, $2$, $2$
- sw $15$, 100($2$)
Forwarding Hardware

- **Forwarding unit:** compares appropriate register read & register write addresses of two appropriate stages; if equal, directs multiplexors to forward appropriate data to ALU inputs earlier
Load word can still cause a hazard:
- an instruction tries to read a register following a load instruction that writes to the same register
- thus, we need a hazard detection unit to “stall” the instruction

Can't Always Forward

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
</table>

lw $2, 20($1)

and $4, $2, $5

or $8, $2, $6

add $9, $4, $2

slt $1, $6, $7
We can stall the pipeline by keeping an instruction in the **same** stage.

- Same effect as inserting a **nop** instruction.
Hazard Detection Hardware

- Hazard detection unit:
  - If load destination register number (at ID/EX) matches one of next instruction’s source register numbers (at IF/ID): stall pipeline by
    - (1) deasserting (set to 0s) all control signals in EX, MEM, & WB stages (= insert a “nop”);
    - (2) disabling IF/IDWrite (= repeat same ID stage one more time) and PCWrite (= repeat same IF stage one more time)
Hazard Detection Hardware
Stalling until branch complete is too slow, hence execute guess and move on!
- At `beq`, we predict "branch not taken" (branch prediction) and continue the pipe by following instructions.
- When we decide to branch, those instructions are already in the pipe!
- Need to add hardware to flush instructions if we are wrong.
Control (Branch) Hazards

- Possible solutions:
  - Reducing delay of branches:
    - Move branch execution **earlier**, then fewer instructions need to be flushed
    - Move branch decision/preparation to **ID stage**:
      - (1) move *branch adder* from MEM stage to ID stage (since we already have PC value and immediate field in IF/ID pipeline register);
      - (2) instead of using the ALU to subtract, we test equality of register values ("beq") by XOR respective bits and then OR the results (0 = equal). This is much faster than using ALU and can be placed in ID stage
    - Hence only **one** instruction to flush if branch is taken
  - The instruction to be flushed is in IF stage, hence we add a control line **IF.Flush** to zeros the instruction field of IF/ID pipeline register --> transform it into “nop”
Flushing Instructions
Dynamic Branch Prediction

- **Dynamic branch prediction** – prediction of branches at runtime

- Several branch prediction techniques:
  - assume branch not taken (as we have seen so far)
  - check branch history (need branch prediction buffer or branch history table) – 1 or more bits indicating whether the branch was recently taken
  - "branch delay slot": the next instruction after a branch is always executed; rely on compiler to "fill" the slot with something useful
  - other more advanced techniques
Exception Handling

- Exception – another form of control hazard
- Exception handling:
  - flush instructions that followed the interrupted instruction (in addition to IF.Flush, need ID.Flush, EX.Flush, deasserting control lines, preventing writing of inappropriate results)
  - save return address in EPC
  - fetch instruction from service routine
CPI in Pipelined CPU

Example:

<table>
<thead>
<tr>
<th>Instruction Mix</th>
<th>Load</th>
<th>Store</th>
<th>R-type</th>
<th>Branch (cond.)</th>
<th>Jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC instruction mix</td>
<td>22%</td>
<td>11%</td>
<td>49%</td>
<td>16%</td>
<td>2%</td>
</tr>
</tbody>
</table>

Assume:
- Memory access = 200 ps
- ALU operation = 100 ps
- Register file read/write = 50 ps
- Half of load instructions are immediately followed by an instruction that uses the results
- Branch delay on mis-prediction is 1 clock cycle
- 1/4 of branches are mis-predicted
- "Jump" pays 1 full clock cycle of delay (calculate target address in ID stage, then know where to fetch next instruction), so their average time in pipe is 2 clock cycles
- Ignore other hazards

Solution:
- For pipelined, load takes 1 cycle if no load-use dependency and 2 cycles when there is; hence average load = 1.5 clock cycles
- Store = 1 clock cycle
- R-type = 1 clock cycle
- Jump = 2 cycles
- Branch takes 1 cycle when predicted correctly, 2 when not; hence average branch = 1.25 cycles

\[
\text{CPI} = 0.22 \times 1.5 + 0.11 \times 1 + 0.49 \times 1 + 0.16 \times 1.25 + 0.02 \times 2 = 1.17
\]

Cycle time = 200 ps (longest functional unit), average instruction time = \(1.17 \times 200 \text{ ps} = 234 \text{ ps}\)

Comparing to multi-cycle \(4.04 \times 200 \text{ ps} = 808 \text{ ps}\) (pipelined is 3.45 times faster) and single-cycle \(200+50+100+200+50 = 600 \text{ ps}\) (pipelined is 2.56 times faster)
Some Examples

- All modern processors are very complicated
  - The last Intel x86 processor without pipelining was 386 (1985). Later Intel x86 processors (486, Pentium, Pentium Pro, MMX, Pentium II, Pentium III, Pentium IV, ...) employ successively more sophisticated pipelining approaches
  - Each Intel processor since Pentium is capable of executing more than one instruction per clock (all superscalar)
  - Each Intel processor since 486 uses a combination of hardwired control (for simple instructions) and microcode control (for more complex instructions)
  - The first MIPS processor (R2000) was pipelined
Chapter Summary

- Three modes of execution: single-cycle, multi-cycle, pipelined
- Pipelined control strives for 1 clock cycle per instruction, improve instruction throughput, by overlapping of instruction executions
- Hazards: structural, control, data --> limit pipeline benefits
- Pipelined datapath: pipeline registers, use of logical components
- Pipelined control: passing of control signals in pipeline registers
- Data hazards: compiler solutions, forwarding (forwarding unit), stalling (hazard detection unit)
- Control hazards: branch prediction, flushing