Improving Performance: Pipelining

Phases of instruction execution

IF  Instruction Fetch (includes PC increment)
ID  Instruction Decode + fetching values from general purpose registers
EXE Execute arithmetic/logic operations or address computation
MEM Memory access or branch completion
WB  Write Back results to general purpose registers (a.k.a. Commit)
Generalized Phases of Instruction Execution

- Instruction Fetch
  - InstructionRegister = MemRead (INST_MEM, PC)

- Decoding
  - Generate datapath control signals
  - Determine register operands

- Operand Assembly
  - Trivial for some ISAs, not for others
  - E.g. select between literal or register operand; operand pre-scaling
  - Sometimes considered to be part of the Decode phase

- Function Evaluation or Address Calculation (Execution)
  - Add, subtract, shift, logical, etc.
  - Address calculation is simply addition

- Memory Access (if required)
  - Load: ReadData = MemRead (DATA_MEM, MemAddress, Size)
  - Store: MemWrite (DATA_MEM, MemAddress, WriteData, Size)

- Completion
  - Update processor state modified by this instruction
  - Interrupts or exceptions may prevent state update from taking place
Instruction fetch

- Read from memory (typically, Instruction Cache) at address given by PC
- Increment PC, i.e. \( PC = PC + \text{sizeof(instruction)} \)
MIPS R-type instruction format

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>reg rs</td>
<td>reg rt</td>
<td>reg rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

Destination register for R-type format

- **add** $1, $2, $3
- **sll** $4, $5, 16
MIPS I-type instruction format

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>16 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>reg rs</td>
<td>reg rt</td>
<td>immediate value/addr</td>
</tr>
</tbody>
</table>

Destination register for Load

```
lw    $1, offset($2)
beq   $4, $5, .Label1
addi  $1, $2, -10
```

```
lw    $2   $1   address offset
beq   $4   $5   (PC - .Label1) >> 2
addi  $2   $1   0xffff6
```
Reading Registers

- Use source register fields to address the register file and read two registers
- Select the destination register address, according to the format
Extracting the literal operand

- Sign-extend the 16-bit literal field, for those instructions that have a literal

```
Verilog
lit = { 16{inst[15]}, inst[15:0] }
```
Performing the Arithmetic

- Perform arithmetic or logical operation on Read Data 0 and either Read Data 1 or the sign-extended literal
Inside the ALU

- Adder, Logic Unit, and Barrel Shifter are separate combinational logic blocks.
Computing Branch Displacements

- Compute sum of PC and scaled, sign-extended literal displacement
- The main ALU is used for evaluating the branch condition (BEQ, BNE)
Accessing Memory – Loads & Stores

- Load and Store instructions use the ALU result as the effective address
- Store instructions use Read Data 1 as the store data
Controlling Instruction Execution

- Control signals driven by combinational logic, based on instruction opcode
Putting it all together
Motivating Pipelined Instruction Execution

Phases of Instruction Execution

Fetch | Decode | Execute | Memory | Write
Pipelined Instruction Execution

Phases of Instruction Execution

Fetch | Decode | Execute | Memory | Write
Pipelined Instruction Execution

Problem: need a way to separate instruction state between pipeline stages

Solution: clocked pipeline latches (registers)
  - Operands (e.g., register values)
  - Intermediate values (e.g., ld/st address)
  - Control signals
CPU Pipeline Structure

IF

- PC
- Instruction memory
  - Read Address
  - Read Data

DEC

- PC+4
- Decode logic
- Register File
  - Read Addr 0
  - Read Data 0
  - Read Addr 1
  - Read Data 1
  - Write Data
  - Write Addr
- ALU decode
- Add
- ALU
- Sign extend
- mux
- mux

EX

- EX
- MEM
- WB
- PC+4
- Add
- bPC
- branch decision
- zero
- ALU decode
- mux

MEM

- MEM
- WB
- Data Memory
  - Address
  - Read data
  - Write data
- mux

WB

- WB
- mux

4
Representing a sequence of instructions

- Space-time diagram of pipeline
- Think of each instruction as a time-shifted pipeline

Instruction 1

Instruction 2

Instruction 3

Instruction 4

Instruction 5
Information flow constraints

- Information from one instruction to any successor must always move from left to right
Another way to represent pipeline timing

A similar, and slightly simpler, way to represent pipeline timing:

- Clock cycles progress left to right
- Instructions progress top to bottom
- Time at which each instruction is present in each pipeline stage is shown by labelling appropriate cell with pipeline name

<table>
<thead>
<tr>
<th>Instruction \ cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction 1</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 2</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 3</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 4</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 5</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation Issues: Pipeline balance

- Each pipeline stage is a combinational logic network
  - Registered inputs and outputs
  - Longest circuit delay through all stages determines clock period

Ideally, all delays through every pipeline stage are identical

In practice this is hard to achieve
Pipeline Hazards

- Hazards are pipeline events that restrict the pipeline flow
- They occur in circumstances where two or more activities cannot proceed in parallel
- There are three types of hazard:
  - **Structural Hazards**
    - Arise from resource conflicts, when a set of actions have to be performed sequentially because there is not sufficient resource to operate in parallel
  - **Data Hazards**
    - Occur when one instruction depends on the result of a previous instruction, and that result is not yet available. These hazards are exposed by the overlapped execution of instructions in a pipeline
  - **Control Hazards**
    - These arise from the pipelining of branch instructions, and other activities that change the PC.
Structural Hazards

- Multi-cycle operations
- Memory or register file port restrictions

Example structural hazard caused by having only one memory port

<table>
<thead>
<tr>
<th>Instruction \ cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $1,($2)</code></td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 2</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 3</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 4</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 5</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effect is to STALL instruction 4, delaying its entry to IF by one cycle

<table>
<thead>
<tr>
<th>Instruction \ cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $1,($2)</code></td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 2</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 3</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 4</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction 5</td>
<td>IF</td>
<td>DEC</td>
<td>EX</td>
<td>ME M</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Hazards

- Overlapped execution of instructions means information may be required before it is available.

```
ADD R1, R2, R3
SUB R4, R1, R5
AND R6, R1, r7
OR R8, R1, R9
XOR R10, R1, R11
```
Data hazards lead to pipeline stalls

- SUB instruction must wait until R1 has been written to register file
- All subsequent instructions are similarly delayed

```
ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, r7

OR R8, r1, R9

XOR R10, R1, R11
```
Minimising data hazards by data-forwarding

- Key idea is to bypass the register file and forward information, as soon as it becomes available within the pipeline, to the place it is needed.
CPU pipeline showing forwarding paths

Also forwarding through the Reg File (no new datapath needed)
Data hazards requiring a stall

- Hazards involving the use of a Load result usually require a stall, even if forwarding is implemented.

```
LW R1, (R2)
SUB R4, R1, R5
AND R6, R1, r7
OR R8, r1, R9
XOR R10, R1, R11
```
Hazards involving the use of a Load may be avoided by reordering the code.
Code scheduling to avoid stalls (after)

- SUB is entirely independent of other instructions – place after 1st load
- ADD to R1 can be placed after LW to R3 to hide the load delay on R3
General Performance Impact of Hazards

Speedup from pipelining: \[ S = \frac{\text{CPI}_{\text{unpipelined}}}{\text{CPI}_{\text{pipelined}}} \times \frac{\text{clock}_{\text{unpipelined}}}{\text{clock}_{\text{pipelined}}} \]

\[ \text{CPI}_{\text{pipelined}} = \text{ideal CPI} + \text{stall cycles per instruction} = 1 + \text{stall cycles per instruction} \]

\[ \text{CPI}_{\text{unpipelined}} \sim \text{pipeline depth} \]

\[ \frac{\text{clock}_{\text{unpipelined}}}{\text{clock}_{\text{pipelined}}} \sim 1 \]

\[ S = \frac{\text{pipeline depth}}{1 + \text{stall cycles per instruction}} \]
Control Hazards

- When a branch is executed, PC is not affected until the branch instruction reaches the MEM stage.
- By this time 3 instructions have been fetched from the fall-through path.

```
BEQZ R1, label
SUB R4, R2, R5
AND R6, R2, r7
OR R8, r2, R9
: label:
XOR R10, R1, R11
```
Effect of branch penalty on CPI

- In this example pipeline the cost of each branch is:
  - 1 cycle, if the branch is not taken (due to load-delay slot)
  - 4 cycles, if the branch is taken

- If an equal number of branches are taken and not taken, and if 20% of all instructions are branches (a reasonable assumption), then
  - \[ \text{CPI} = 0.8 + 0.2 \times 2.5 = 1.3 \]
  - This is a significant reduction in performance

- If the pipeline was deeper, with 2 stages for ALU and 2 stages for Decode, then:
  - Cost of taken branch would be 6 cycles
  - \[ \text{CPI} = 0.8 + 0.2 \times 3.5 = 1.5 \]

- Deeper pipelines have greater branch penalties, and potentially higher CPI
- Pentium 4 (Prescott) had 31 pipeline stages! (this was too deep)
- Several important techniques have been developed to reduce branch penalties
  - Early branch outcome
  - Delayed branches with branch delay slot(s)
  - Branch prediction (static and dynamic)
Early branch outcome calculation - BEQZ, BNEZ

Instruction memory
Read Address Data

PC

IF
Add
PC+4

[31:26]

[25:21]

[20:16]

[15:0]

[15:11]

Instruction memory
Read Address Data

Register File
Read Addr 0 Read Data 0
Read Addr 1 Read Data 1
Write Data
Write Addr

Sign extend

Decode logic

<< 2

Add

RD0 == 0?

ALU

Data Memory
Read data
Write data

Add

ALU decode

EX

MEM

WB

Data Memory
Address

IF

DEC

EX

MEM

WB

PC+4

m x

m x

m x

m x

m x

m x
Delayed branch execution (branch delay slot)

- Always execute the instruction immediately after the branch, regardless of branch outcome

Before: instruction after the branch gets killed if the branch is taken

After: by moving the SUB instruction into the branch delay slot, and executing it unconditionally, the 1-cycle penalty is eliminated
Impact of Branch Hazards on CPI

Bottom-line: CPI increase of 0.06 – 0.62 cycles