Improving Performance: Pipelining

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)
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**
  - Add, subtract, shift, logical, etc.
  - Address calculation is simply unsigned 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

Note: INST_MEM and DATA_MEM may be same or separate physical memories
Instruction fetch

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

The R-type instruction format is composed of the following fields:

- **6 bits** for the opcode
- **5 bits** for `reg rs`
- **5 bits** for `reg rt`
- **5 bits** for `reg rd`
- **5 bits** for `shamt`
- **6 bits** for `funct`

**Destination register for R-type format**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Args</th>
<th>Special</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$1, $2, $3</td>
<td>special</td>
<td>$2, $3, $1, add</td>
</tr>
<tr>
<td>sll</td>
<td>$4, $5, 16</td>
<td>special</td>
<td>$5, $4, 16, sll</td>
</tr>
</tbody>
</table>
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>rs</td>
<td>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
- Can’t share ALU, it might be needed for comparisons during branch operations
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
Decoding Instructions

- Control signals driven by combinational logic, based on instruction opcode
Pipelined Instruction Execution

Phases of Instruction Execution:
- Fetch
- Decode
- Execute
- Memory
- Write

Time:
1. Fetch
2. Decode
3. Execute
4. Memory
5. Write

Actions:
- Clock
- Time
- Action (Fetch, Decode, Execute, Memory, Write)

Diagram shows the pipelined execution of instructions across different phases and time units.
CPU Pipeline Structure

- **PC**: Program Counter
- **Instruction memory**: Reads instructions
- **Register File**: Reads data and addresses
- **ALU**: Performs arithmetic and logical operations
- **Data Memory**: Reads and writes data
- **Branch decision**: Determines if a branch should be taken
- **IF**: Instruction Fetch
- **DEC**: Decode
- **EX**: Execute
- **MEM**: Memory Access
- **WB**: Write Back

Key steps include:
- **PC**: Incremented for next instruction
- **Instruction memory**: Read instruction
- **Register File**: Read addresses and data
- **ALU**: Performs operations (e.g., add, subtract)
- **Data Memory**: Reads or writes data
- **Branch decision**: Determines if branch is taken
- **Instruction execution**: Pipeline stages (IF, DEC, EX, MEM, WB)

Flow of control:
- **PC**: Incremented to fetch next instruction
- **Instruction memory**: Reads instruction
- **Register File**: Reads data and addresses
- **ALU**: Performs operations
- **Data Memory**: Reads or writes data
- **Branch decision**: Determines if branch is taken
- **Instruction execution**: Pipeline stages (IF, DEC, EX, MEM, WB)

**Notes**:
- **alu**: ALU operations (e.g., add, subtract)
- **sign extend**: Sign extension for 32-bit operations
- **branch**: Branch instruction handling
- **mux**: Multiplexors for data routing

**Pipeline Stages**:
- **IF**: Instruction Fetch
- **DEC**: Decode
- **EX**: Execute
- **MEM**: Memory Access
- **WB**: Write Back

**Branch Handling**:
- Branch instructions (e.g., jump, call) affect program flow
- Branch decision based on zero or non-zero conditions

**Data Flow**:
- Data movement between registers, memory, and ALU
- Instruction flow through the pipeline

**Pipeline Interaction**:
- Data and control signals flow through the pipeline stages
- Multiplexors (mux) control data routing

**Reference**:
- Inf3 Computer Architecture - 2014-2015
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
Representing a sequence of instructions

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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
<tr>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
<td>IF</td>
<td>Reg</td>
<td>ALU</td>
<td>Mem</td>
<td>Reg</td>
</tr>
</tbody>
</table>
Information flow constraints

- Information from one instruction to any successor must always move from left to right

![Diagram showing information flow constraints]
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

- This form is used in H&P, and throughout the remainder of these notes.

<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>
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>lw $1,($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>
<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>
<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>
<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>
<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>
<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>lw $1,($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>
<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>
<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>
<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>
<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>
<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
Data hazards requiring a stall

- Hazards involving the use of a Load result usually require a stall, even if forwarding is implemented.
Code scheduling to avoid stalls (before)

- Hazards involving the use of a Load may be avoided by reordering the code

```
LW R1, 2(R2)
LW R3, 4(R1)
ADD R4, R4, R3
ADD R1, R1, 4
SUB R9, R9, 1
```
Code scheduling to avoid stalls (after)

- SUB is entirely independent of other instructions – place after 1\textsuperscript{st} 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: \[ S = \frac{CPI\text{\textsubscript{unpipelined}}}{CPI\text{\textsubscript{pipelined}}} \times \frac{\text{clock}\text{\textsubscript{unpipelined}}}{\text{clock}\text{\textsubscript{pipelined}}} \]

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

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

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

\[ S = \frac{\text{pipeline depth}}{1 + \text{stall cycles per instruction}} \]
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
  - CPI = 0.8 + 0.2*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
  - CPI = 0.8 + 0.2*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
  - Branch prediction (static and dynamic)
Early branch outcome calculation - BEQZ, BNEZ
Delayed branch execution

- 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
Case Study: Pipelining in MIPS R4000

- Introduced in early 90s
  - 1.2 million transistors, 250 Mhz peak frequency
  - 64-bit CPU – one of the first!
- Notable feature: pipelined memory accesses
Load-to-use latency in the MIPS R4000

2-cycle load delay slot
Impact of Empty Load-delay Slots on CPI

Bottom-line: CPI increase of 0.01 – 0.27 cycles
Branch delay in MIPS R4000

3-cycle branch taken delay
Impact of Branch Hazards on CPI

Bottom-line: CPI increase of 0.06 – 0.62 cycles