Pipelining

- RISC/MIPS64 five stage pipeline
- Basic pipeline performance
- Pipeline hazards
- Branch hazards
- More pipeline performance
- Challenges of pipelining

RISC Datapath

1. Instruction Fetch Cycle (IF)
2. Instruction decode/register fetch cycle (ID)
3. Execution/effective address cyclic (EX)
4. Memory Access (MEM)
5. Write-back (WB)
Classic RISC Five Stage Pipeline (Fig. A.1)

<table>
<thead>
<tr>
<th>Instruction number</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 /</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction / + 1</td>
<td>IF</td>
<td>ID</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>ID</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>ID</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>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.
Pipelining yields performance gains

- Pipelining increases instruction throughput, but does not reduce the execution time of any single instruction.
- Idealized maximum speedup = number of pipeline stages
- Idealized maximum speedup achieved when one instruction completes each clock cycle.
  \[ CPI(\text{unpipelined}) = P \]
  \[ CPI(\text{pipelined}) = 1 \]

  \[ \text{Speedup} = \frac{IC \times P \times \text{clock}}{IC \times 1 \times \text{clock}} = P \]

Limits on pipeline performance

- Imbalance among pipeline stages (limited by the slowest stage)
- Overhead due to pipeline register delays
- Overhead due to clock skew (max delay between when clock signal arrives at any two registers)
- Overhead to fill/drain the pipeline
  - **PIPELINE HAZARDS**
Pipeline Hazards

- Structural hazards - conflicts for HW resources
- Data hazards - conflicts for access to data
- Control hazards - uncertainty about the next instruction to enter the pipeline

Pipelining Performance w/ hazards

- Hazards are dealt with by stalling the pipeline. Stalls delay one or more stages for the instruction in the pipeline, causing the CPI to increase.

\[
\text{CPI}(\text{unpipelined}) = P \\
\text{CPI}(\text{pipelined}) = 1 + \text{stall cycles per instruct.}
\]

\[
\text{Speedup} = P / (1 + \text{stall cycles per instruct.})
\]
Structural Hazards: insufficient HW resources

Dealing with structural hazards

- Add more hardware
- Stall the pipeline

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Load instruction</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 1</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 2</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 3</td>
<td>stall</td>
</tr>
<tr>
<td>Instruction i + 4</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 5</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction i + 6</td>
<td>IF</td>
</tr>
</tbody>
</table>

Figure A.5 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction i + 3). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 5. Sometimes timing diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction i + 3 does not begin execution until cycle 5. We use the form above, since it takes less space in the figure.
Data Hazards:

Incorrect access to data because pipelining changes the order of read/write accesses to data

- DADD R1,R2,R3
- DSUB R4,R1,R5
- AND R6,R1,R7
- OR R8,R1,R9
- XOR R10,R1,R11

Dealing with data hazards

- Forwarding - HW technique that gets needed data from the pipeline registers (rather than waiting for it to be WB in the regular registers) and moving it to where that data is needed in a timely manner.

- Stalling (with pipeline interlock)
Data hazard example #1 (Fig. A.6)

Overcoming data hazard example #1 with forwarding (Fig. A.7)
Data hazard example 2: forwarding doesn’t work (Fig. A.9)

Overcoming data hazard example #2 by stalling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
<th>Stage 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, 0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
<th>Stage 4</th>
<th>Stage 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, 0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

Figure A.10 In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.
Control Hazards

- Created by branch statements

BEQZ LOC
ADD R1, R2, R3
.
.
LOC SUB R1, R2, R3

- PC needs to be computed but it happens too late (at the end of ID, the 2nd stage)

Four Branch Hazard Alternatives

#1 STALL: until branch direction is known

#2 Predict Branch Not Taken: guess that the branch will not be taken and execute successor instruction. Cancel out instructions in the pipeline if wrong guess.

#3 Predict Branch Taken: opposite of above.

#4 Delayed Branch: insert useful instructions into the pipeline until the branch direction is known.
Solution #1: Stall

- PC is computed in ID stage
- Thus only a 1 cycle stall is incurred for branch hazards

Solution #1 (STALL) incurs a penalty of 1 cycle.

#2 Predict Not Taken (Fig. A.12)

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction / + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Instruction / + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction / + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction / + 4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction / + 1</td>
<td>IF</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Figure A.12 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.
#2 Predict Not Taken

- Penalty: 0 if not taken, 1 if taken

Assume 30% conditional branches of which 60% are not taken:
\[ \text{CPI} = (0.70)(1) + (0.30)((0.60)(1) + (0.40)(1+1)) \]

#3 Predict Taken

- Similar situation and analysis as Predict Not Taken
#4 Delayed Branch

Delay Slot = the slots in the pipeline that would be stalls (we are going to fill them with instructions and try to avoid stalls)

(a) Non-cancelling Delayed Branch
   Useful instructions are inserted into the delay slots

(b) Cancelling Delayed Branch
   Some instructions are inserted into the delay slots and cancelled if wrong guess (as in Predict Not Taken and Predict Taken)

#4a: Delayed Branch - non-cancelling

- Fill the slot with a useful instruction
  penalty = 0
- Sometimes no useful instruction can be found by the compiler
  penalty = 1
#4b: Delayed Branch - cancelling with predict taken

- Try to fill with a useful instruction: since we predict taken, it can be chosen from instructions at the taken location
- If the prediction was right
  DLX penalty = 0
- If the prediction was wrong, the instruction must be cancelled
  DLX penalty = 1

![Delayed Branch - cancelling with predict taken](image-url)
Where to get instructions to fill the branch delay slot(s)?

- From before the branch instruction
- From the target address (only OK if branch taken)
- From the fall through (only OK if branch not taken)

- Compiler effectiveness for single branch delay slot:
  - about 60% of slots are filled
  - about 80% of instructions in delay slots are useful (not cancelled)
Details of the MIPS datapath
(Fig. A.17 - does not include pipeline HW)

MIPS datapath details (p. A27-A28)

- Instruction Fetch (IF)
  IR <-- Mem[PC]
  NPC <-- PC + 4

- Instruction Decode/Register Fetch (ID)
  A <-- Regs[rs]
  B <-- Regs[rt]
  Imm <-- sign extended immediate field of IR

- Execution/Effective address (EX)

- Memory access/Branch completion (MEM)

- Write-back (WB)
MIPS pipeline details (p. A27-A28)

- Execution/Effective address (EX)
  - If memory reference: ALUOutput <-- A + Imm
  - If Reg-Reg ALU: ALUOutput <-- A func B
  - If Reg-Imm ALU: ALUOutput <-- A op Imm
  - If Branch: ALUOutput <-- NPC + (Imm << 2)

- Memory access/Branch completion (MEM)
  - If memory reference: LMD <-- Mem[ALUOutput]
  - If Branch: if (cond) PC <-- ALUOutput

- Write-back (WB)
  - If Reg-Reg ALU: Regs[rd] <-- ALUOutput
  - If Reg-Imm ALU: Regs[rt] <-- ALUOutput
  - If Load: Regs[rt] <-- LMD
Details of the MIPS pipeline HW

MIPS pipeline events compared to MIPS (non-pipelined) datapath (Fig. A.18)

Pipelined

- Instruction Fetch (IF)
  IF/EX.IR ← Mem[PC]
- Non-pipelined
  IR ← Mem[PC]
  NPC ← PC + 4

IF/ID.NPC,PC ← (if (EX/MEM.opcode == branch) &
  IF/ID.NPC,PC ← (if (EX/MEM.opcode == branch) &
    EX/MEM ALUOutput
  else {PC + 4})


**How HW detects data hazards**

(See Fig. A.21)

LW  R1, 0(R2) in EX stage
ADD  R3, R1, R4 in ID stage

Check opcode field of ID/EX (bits 0-5): LOAD
Check opcode field of IF/ID (bits 0-5): ALU

Check operands for RAW:
ID/EX.IR[rt] == IF/ID.IR[rs]

**How HW accomplishes forwarding**

(See Fig. A.22 which show all 10 needed comparisons)

Example of first comparison:

PipelineReg. source: EX/MEM
Opcode of source:   Reg-Reg ALU
PipelineReg. destination: ID/EX
Opcode of destination: Reg-Reg ALU, ALU imm, load, store, branch
Destination of forward: Top ALU input
Comparison test for forward: EX/MEM.IR[rd] == ID/EX.IR[rs]

etc. etc. etc.
Pipeline HW has to deal with exceptions/interrupts: (Fig. A.27)

Resume? Terminate?

- I/O interrupt
- OS kernel call
- Trace instruction execution
- Breakpoint
- Integer overflow
- FP overflow or underflow
- Page fault
- Misaligned memory access
- Memory protection violation
- Undefined instruction
- Hardware malfunction
- Power failure
Precise exceptions:

Are supported by a pipelined system that stops the pipeline just before the faulting instruction completes and after the fault is handled, restarts the following instructions from scratch.