Pipelining
Overlap execution of multiple instructions. Implementation method similar to assembly line concept.

Definitions:
Pipeline stage: smallest step of task
Gain from pipelining: Number of stages
Pipelining improves Throughput (number of instructions executed per unit time) not the time to execute instruction
Machine cycle time: time of slowest stage "to move instruction one step".
Pipelining improves performance by N fold where N is number of stages.
Pipelining is invisible to programmer "no software change"
Limitation of pipelining: Partioning of instruction execution, slowest stage, overhead between stages (latch), hazards.

N. Mekhiel
Instruction Execution

Instruction Format:–

1–R Type

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs1</th>
<th>rt</th>
<th>rd</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..</td>
<td>5</td>
<td>6..</td>
<td>10..</td>
<td>11..</td>
</tr>
</tbody>
</table>

Example:

```plaintext
add $17, $18, $19  ;  $17      \[\$18\] + \[\$19\]
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rs1</th>
<th>rt</th>
<th>rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>18</td>
<td>19</td>
<td>17</td>
<td>arith</td>
</tr>
</tbody>
</table>

2– I Type

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs1</th>
<th>rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..</td>
<td>5</td>
<td>6..</td>
<td>10..</td>
</tr>
</tbody>
</table>

Example:

```plaintext
lw $17, 100($18) ;  $17 \Leftarrow \text{MeM}[100+ (\$18)]
```

<table>
<thead>
<tr>
<th>Instruction</th>
<th>rs1</th>
<th>rt</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>18</td>
<td>17</td>
<td>100</td>
</tr>
</tbody>
</table>

3– J Type

<table>
<thead>
<tr>
<th>opcode</th>
<th>PC Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..</td>
<td>5 6</td>
</tr>
</tbody>
</table>

Example: J L1 ; PC=L1+PC

N. Mekhiel
5 Steps to Execute Instructions:

- **IF:**
  Instruction Fetch: Fetch Instruction from Memory to IR

- **ID:**
  Instruction Decode: Decode Instruction, Get operands

- **EX:**
  Execute Instruction

- **Mem:**
  Read /Write Memory for load/store instructions

- **WB:**
  Write back results to Register File

N. Mekhiel
Executing Instructions (Fetch and Decode):

IF: \( IR = M[PC] \), \( PC = PC + 4 \)

ID:
- \( A = R[IR\ 6..10] \); rs1
- \( B = R[IR11..15] \); rs2
- \( Imm = IR(16)^{16} \## IR\ 16..31 \)

N. Mekhiel
Executing Instructions (Execute and Memory):

Execution:
− R type: add R1, R2, R3
  \( \text{ALUout} = A \text{ OP } B \); 
  \( \text{ALUout} = (R2 + R3) \)
− I− Type: addi R1, #500
  \( \text{ALUout} = A \text{ OP } \text{Imm.} \)

Load/store : LW R1, 100(R2)
\( \text{ALUout} = A + \text{Imm} = R2 + 100 \)

Branch : Jeqz R1, 100
\( \text{ALUout} = (\text{PC}+4) + \text{Imm} = \text{PC}+4 + 100 \) (If \( R1 =0 \))
\( \text{OR PC = PC} + 4 \) (If \( R1 = !0 \))

Memory: Load/store and branch completion

\( \text{LMD} = M[\text{ALUout}] \) ; LW
\( M[\text{ALUout}] = B \) ; SW

\( \text{PC=} \text{IF (cond) ALU out} \)
\( \text{else PC}+4 \)

N. Mekhiel
Executing Instructions (Write Back):

Write Back:

Register – Register (ALU operation)
ADD R1, R2, R3 ;
Reg [IR(16..20)] = ALUout ; R1= R2+R3

Reg–Imm
ADD R1, R2, 100
Reg[IR(16..20)] = ALUout
R1= R2 + 100

Load Instruction
LW R1, 100(R2)
Reg[IR(16..20)] = LMD ; R1 = M[100+R2]
Executing Instructions (ALU Operations):

Example: add R1, R2, R3 If PC =1000

DLX Pipeline Datapath

N. Mekhiel
Executing Instructions (Load):

Example: LW R1, 100(R2) If PC =1000

DLX Pipeline Datapath

N. Mekhiel
Example: `beqz R1, 200` If PC = 1000

**DLX Pipeline Datapath**

N. Mekhiel
Example: Find the performance of unpipelined DLX if each instruction takes 4 cycles and the load takes 5 cycles. The frequency of use for load is 26 %.

\[ T = 4 \times (1 - .26) + .26 \times 5 = 4.26 \text{cycles} \]

**Pipelined DLX**

Using the same data path with 5 stages:-

- IF, ID, EX, Mem, WB

Each stage is responsible for completing one task each clock cycle.

Executing one Instruction each cycle.

5 Instructions are being overlapped in pipeline (executing 5 instructions in parallel).

N. Mekhiel
### Executing Instructions in Ideal Pipeline

Example (Ideal Pipeline)

- Each instruction takes 5 cycles to complete.
- Pipeline improves throughput by overlapping instruction executions.
- Ideal pipeline executes 1 Instruction per cycle.

<table>
<thead>
<tr>
<th>Instruction #</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>F1</td>
</tr>
<tr>
<td>2</td>
<td>F2</td>
</tr>
<tr>
<td>3</td>
<td>F3</td>
</tr>
<tr>
<td>4</td>
<td>F4</td>
</tr>
<tr>
<td>5</td>
<td>F5</td>
</tr>
</tbody>
</table>

N. Mekhiel
Requirements to Support Pipelining:

- Need separate Instruction and Data Memory.
- Memory BandWidth is 5 Times (1 per clock cycle compared to 1 every 5 cycles)
- Need Latches between pipeline stages to allow new instruction to change the result of stage. This could limit speed of clock due to latch delay (Max number of stages limited by latch delay).
- Control must forward Data and control signals from stage to stage by copying it to next stage latch
  Example: ADD R1, R2, R3
  In WB stage, the results of ALU must be written to R1, so (IR 16..20) must be in WB latch.

N. Mekhiel
Performance Issues in Pipelining

Ideal Throughput = 1 Instruction per cycle

Limitations due to: Overhead of latches between pipeline stages, imbalance between stages, and Hazards.

Example: In unpipelined processor that has a clock cycle= 1 ns, and all ALU operations and branches uses 4 cycles and 5 cycles fro memory. Frequencies are: 40% for ALU, 20% branches and 40% memory. If clock cycle time increases by .2 ns, find speedup of pipelined processor.

\[
T_{for\text{unpipelined}} = .4 \times 4 + .2 \times 4 + .4 \times 4 = 4.4\text{ns}
\]

\[
T_{of\text{pipelined\text{instruction}}} = 1 \times 1.2
\]

\[
speedup = 4.4 \div 1.2 = 3.7
\]

Pipeline Hazards

Hazard: Situation that prevent next instruction from executing

N. Mekhiel
Types of Hazards:
1- Structural Hazards: Hardware cannot support all possible combinations due to resource conflicts. (Example Inst. Mem, Data Mem)
Data Hazards: An instruction depends on the results of previous instruction that has not yet completed.
Control Hazards: In branches, the fetching of next instruction is not known in time.

Dealing with Hazards
Simplest solution: stall pipeline, all instructions after stalled instruction must stall too.

Performance of Pipeline with Stall:
\[ speedup = \frac{\text{average-instruction-time-unpipelined}}{\text{average-instruction-time-pipelined}} \]
\[ speedup = \frac{((\text{pipeline-depth}) \times \text{clock-unpipelined})}{((1 + \text{stall-cycles-per-inst}) \times \text{clock-pipelined})} \]
Example: If FP multiply stall pipeline by 5 cycles, and has frequency of 14%. Find performance of pipeline.
\[ T = (1 + 0.14 \times 5) = 1.7 \text{Cycles} \]
N. Mekhiel
Executing Instructions in Pipeline

DLX Pipeline Datapath

N. Mekhiel
Instruction Execution

Execution Steps in Pipeline:

1– IF: Instruction Fetch
   IR ← Mem[PC]
   NPC ← PC+4

2– ID: Instruction Decode
   A ← IR(6..10)
   B ← IR(11..16)
   Decode Instruction

3– EX: Execution
   a– memory references
      ALUout ← A + [(IR16) ## IR(16..31)]
      if store SMD ← B
   b– ALU operations
      ALUout ← A op B
      if immediate ALUout ← A + [(IR16) ## IR(16..31)]
   c– Branch or Jump
      ALUout ← NPC + [(IR16) ## IR(16..31)]
      Cond ← A op 0

N. Mekhieil
Execution Steps in Pipeline

4– Mem: Memory
   a– memory references
      LMD ← Mem[ALUout] if load
      Mem[ALUout] ← SMD if store
   b– Branch
      if (Cond) then PC ← ALUout
      else PC ← PC+4

5–WB: Write Back
   a– ALU operations
      Reg[IR(16..20)] ← ALUout
      if Immediate: Reg[IR(11..15)] ← ALUout
   b– load
      Reg[IR(11..15)] ← LMD

N. Mekhiel
Pipeline Hazards

Example (Pipeline Hazards)

1– Structural Hazards:
   Not enough resources or unit is not pipelined

Examples:–
–Cycle 4, both instruction1 and instruction4 use the memory.
  solution: use separate Instruction and Data Memory

–Cycle 5, Register file is used for WB and also read operand in Decode.
  solution: use different read and write ports.

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1  2  3  4  5  6  7  8  9</td>
</tr>
<tr>
<td>1</td>
<td>lw$2, 20($16)</td>
<td>F1</td>
</tr>
<tr>
<td>2</td>
<td>add$4, $2, $6</td>
<td>F2</td>
</tr>
<tr>
<td>3</td>
<td>bnz $7, Loop</td>
<td>F3</td>
</tr>
<tr>
<td>4</td>
<td>sub$1, $3, $5</td>
<td>F4</td>
</tr>
</tbody>
</table>

N. Mekhieil
Data Hazards

Pipeline changes the order of read/write accesses to operands causing Data hazards. This causes it to not be the same as seen by sequential order. Types:-

- **RAW:**
  - Write to X
  - Read X
  - If read X before write to X

- **WAW:**
  - Write 5 to X
  - Write 7 to X
  - If write 7 is before write 5

- **WAR:**
  - Read X
  - Write to X
  - If write to X before Read X

- **RAR:**
  - Not a Hazard

N. Mekhiel
Data Hazards

Example (with data hazards)

− Last four Instructions depend on results in $2
− Result in $2 available at end of WB (Clock#5)
− Source operands for four instructions need $2 at ID (2nd cycle)
− Can split Write, read in register file to remove data hazard at cycle #5

<table>
<thead>
<tr>
<th>Instruction</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>sub$2,$1,$#3</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>$2</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and$12,$2,$5</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or$13,$6,$2</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add$14,$2,$2</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw$15,100($2)</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

hazard hazard okay by splitting wr,rd no hazard

N. Mekhiel
Example (with data hazards)

Dealing with Data Hazards

Example with data hazards

Solution: Stall Pipeline (use nop)

<table>
<thead>
<tr>
<th>Instruction</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>sub$2,$1,$#3</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and$12,$2,$5</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or$13,$6,$2</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add$14,$2,$2</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw$15,100($2)</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Performance= 11 cycles compared to 9 cycles for ideal pipeline

22% slower (too much)

N. Mekhiel
Data Hazards

Example (with data hazards)
Solution with forwarding (zero stalls)

1–Results are valid and ready at the end of EX stage (early than WB).
2–Source operands are needed in EX stage (later than ID).
3–Keep ALUout (results) around and forward it to ALUin (source).
4–Need forwarding paths, multiplexors and detection of dependency.

<table>
<thead>
<tr>
<th>Instruction</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>sub$2,$1,$#3</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td>$2$</td>
<td>Reg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and$12,$2,$5</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or$13,$6,$2</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add$14,$2,$2</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw$15,100($2)</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N. Mekhiel
Data Hazards

Solution: Data Forwarding

Dependency Detection

hazards conditions:–

1–EX/MEM.Reg.Rd = ID/EX.Reg.(Rs/Rt)

2–MEM/WB.Reg.Rd = ID/EX.Reg.(Rs/Rt)

Dependency Detection in the given Example:–

1– sub$2,$1,$2 (EX/MEM.Reg.$2) = and$12,$2,$5(ID/EX.Reg.$2)

2– sub$2,$1,$2 (MEM/WB.Reg.$2) = or$13,$6,$2(ID/EX.Reg.$2)
## Data Hazards

**When Forwarding Fails**

Cannot forward in negative time

must stall the pipeline

<table>
<thead>
<tr>
<th>Instruction</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>lw$2, 20($8)</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and$4,$2,$5</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or$6,$1,$2</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add$9,$3,$7</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N. Mekhiel
Data Hazards

When Forwarding Fails

Solution: Stall pipeline (insert bubble)

<table>
<thead>
<tr>
<th>Instruction</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>lw$2, 20($8)</td>
<td>IM</td>
<td>Reg</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and$4,$2,$5</td>
<td>IM</td>
<td>Reg</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or$6,$1,$2</td>
<td>IM</td>
<td>Reg</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add$9,$3,$7</td>
<td>IM</td>
<td>Reg</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N. Mekhiel
Data Hazards

Performance of Pipeline with stalls

SPEC Benchmarks show % of loads causing stalls as:

- compress 24%
- gcc 23%
- espresso 12%
- eqntott 41%

average = 34%

assume that 50% of instructions are loads

CPI = 1 + .5*.34*1 = 1.17 cycle per instruction

Solution: Pipeline Scheduling

1– Static scheduling using Compiler
   Compiler can schedule instructions between loads and the following instructions to avoid stalls.

2– Dynamic scheduling using Hardware
   Could use scoreboard to detect dependency and allow out of order instruction execution to eliminate the stalls.

N. Mekhiel
Forwarding to DM for Store Instruction

Example:

ADD R1, R2, R3
LW R4, 0(R1) ; RAW
SW 12 (R1), R4; RAW

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add R1, R2, R3</td>
<td>F1</td>
<td>D1</td>
<td>EX1</td>
<td>M1</td>
<td>WB1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW R4, 0(R1)</td>
<td>F2</td>
<td>D2</td>
<td>EX2</td>
<td>M2</td>
<td>WB2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW 12(R1), R4</td>
<td>F3</td>
<td>D3</td>
<td>EX3</td>
<td>M3</td>
<td>WB3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1−ALUout is valid at end of cycle 3=R1,
   LW Instruction needs R1 at start of cycle 4 (forward from ALUout)

2−Data from memory for LW is valid at end of cycle 5, SW needs it on cycle 6 (forward from DM)

N. Mekhiel
Software Scheduling to Avoid Load Data Hazard

Compiler can schedule and rearrange code to avoid load hazards.
Compiler uses "pipeline scheduling" to avoid having the following instruction using dest of load.

Example: \( C = A + B; \ D = E - F; \)

- LW R1, B ; R1=B
- LW R2, A ; R2=A
- ADD R3, R1, R2 ; STALL FOR ONE CYCLE due to load R2
- SW C, R3; C= A+B
- LW R4, E ;
- LW R5, F
- SUB R6, R4, R5 ; STALL ONE CYCLE due to load R5
- SW D, R6

N. Mekhial
Scheduling the Code
LW R1, B ; R1=B
LW R2, A ; R2=A
LW R4, E
ADD R3, R1, R2 ;
LW R5, F
SW C, R3
SUB R6, R4, R5 ; STALL ONE CYCLE due to load R5
SW D, R6

Tradeoffs: Need to use different register for E (More registers)

Control for Pipeline
Must be able to forward data to different units
Must be able to stall if forwarding cannot be used
Details depend on Implementation.

N. Mekhiel
Control Hazards

Control Hazards cost more performance loss than data hazards.

The problem is more complex because:-

- branch target address unknown until Execution stage

- CC result of zero detect unit to decide if branch is taken/not taken at Memory stage

- Don’t know that instruction is a branch until ID Stage

N. Mekhilef
Dealing with Control Hazards:
Simple solution: Stall pipeline for 3 cycles
After stall pipeline, must repeat IF stage
**Performance cost due to control stalls**

<table>
<thead>
<tr>
<th></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>40 beq R1, R3, 36</td>
<td>F1</td>
<td>D1</td>
<td>EX1</td>
<td>M1</td>
<td>WB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 AND R12, R2, R5</td>
<td>F2</td>
<td>stall</td>
<td>stall</td>
<td>stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48 OR R13, R6, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52 ADD R14, R2, R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80 LW R4,100(R7)</td>
<td></td>
<td></td>
<td></td>
<td>CC</td>
<td>F5</td>
<td>D5</td>
<td>EX5</td>
<td>M5</td>
<td>WB5</td>
</tr>
</tbody>
</table>

Need to stall 3 cycles, CC results are detected in Mem stage

Do not know that instruction is a branch until ID stage

Simple solution: stall pipeline

N. Mekhiel
Assume 30% branch frequency, control stalls 3 cycles.

\[ \text{Performance} = 1 + 0.3 \times 3 = 1.9 CPI \]

This is about 50% reduction in performance (Too much)

**Methods For Reducing branch Penalty**

- Find result of condition code earlier
- compute the branch target earlier
- can perform both in ID stage by adding extra circuits (adder, comparator)

N. Mekhiel
Methods For Reducing branch Penalty

Branch Hazard reduced by moving Zero detect and branch target calculation to ID

N. Mekhie
Cost of reduced branch penalty
Cost of branch penalty is still 1 cycle.
Always stall until branch direction is known in ID stage, must fetch instruction again if it is a branch.
Performance with 1 stall and 30% frequency of branch \( \text{Performance} = 1 + 0.3 \times 1 = 1.3 \), 30% loss in performance.
Improving cost of control hazard by prediction

- Predict branch is not taken
  Execute successor Instructions in sequence
  If branch is taken, need to turn fetch instruction to no op.
  If 47% of branches are not taken, then it will save 47% of branch cost.
  \( \text{performancecost} = 1 + 0.3 \times (1 - 0.47) \times 1 = 1.15 \) cost only 15% loss in performance.

N. Mekhiel
Improving cost of control hazard by prediction

- **Predict branch is taken**
  53% of branches are taken
  But BTA is not known until ID stage
  Always need 1 cycle stall
  \[\text{Performance} = 1 + .3 \times 1 = 1.3\] ; No advantage in DLX

- **Delayed branch**
  Define branch to take place after a following instruction. This instruction will be executed whether the branch is taken or not.
  Example: 3 cycles delayed branch
  branch instruction
  sequential instruction1
  sequential instruction2
  sequential instruction3
  branch target if taken
  1 delay slot for DLX allows proper decision to hide branch latency
  The important task is where to get instructions to fill the delay slot:
<table>
<thead>
<tr>
<th>Instruction</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>i–untaken branch</td>
<td>Fi</td>
<td>Di</td>
<td>EXi</td>
<td>Mi</td>
<td>WBi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**i+1 Delay Slot**

| i+1 Delay Slot | Fi+1 | Di+1 | Ei+1 | Mi+1 | Wi+1 |   |   |   |   |

| i+2 Instruction |   |   |   |   |   |   |   |   |   |

| i+3 Instruction |   |   |   |   |   |   |   |   |   |

| Branch Target Inst. |   |   |   |   |   |   |   |   |   |

| Branch Target Inst+1 |   |   |   |   |   |   |   |   |   |

---

N. Mekhiel
Where to get Instructions to fill Branch Delay Slot

- From before branch. It is always okay

- From Target: Valuable only when branch is taken, and must be okay to execute if branch not taken

- From Fall Through: Valuable only when branch is not taken. Must be okay to execute if branch is taken

N. Mekhiel
Improving cost of control hazard by prediction

- **Cancelling Branches:**
  Need prediction for direction of branch
  If branches behaves as predicted, then branch delay slot instruction is executed as above
  If prediction is wrong, the instruction in delay slot is turned to NOOP.
  Advantages: More freedom for compiler to find instructions to fill slot.

**Compiler Effectivness in Filling Delay Slot**
Fills about 60% of branch delay slots
80% of instructions on delay slot do useful work
\[ \text{Performance improvements} = 0.8 \times 0.6 = 0.48 \text{ or 48\% of slots are eliminated} \]

**Methods of Static Predictions:**
1-From behavior of programs of most applications (backward branches are most likely taken)
2-Use profile information collected from earlier runs.

N. Mekhiel
Evaluating the Performance of the 4 Schemes of branch handling

\[ Pipelinespeedup = \frac{\text{pipelinedepth}(\text{Ideal})}{(1 + \text{branch-frequency} \times \text{branch - penalty})} \]

Example: Assume branch frequency = 16%, taken branch = 67%, delay slot could fill only 70% and only 65% are useful instructions.
1-Always stall \( performance = 1 + 0.16 \times 1 = 1.16 \text{ CPI} \)
2-Predict branch not taken \( performance = 1 + 0.16 \times (0.67) = 1.11 \text{ CPI} \)
3-Predict branch taken in DLX \( performance = 1 + 0.16 \times 1 = 1.16 \text{ CPI} \)
4-Delayed branch \( performance = 1 + 0.16 \times (0.3 + 0.7 \times 0.35) = 1.08 \text{ CPI} \)

N. Mekhiel
Dealing with Exceptions

Harder to handle in a pipeline architecture
Instructions are executed in steps, and we do not know if the instruction should change the state of the CPU safely so it can restart from same condition if an exception occurs.

Types of Exceptions:-

- 1-I/O device request
- 2-OS service call
- 3-Arithmetic errors
- 4-Page Fault
- 5-Memory protection violation
- 7-Hardware malfunction
Types of Exception based on their behavior

• 1-Synchronous versus Asynchronous: Synchronous occurs at the same place in program execution. Asynchronous occurs at any place and most likely from external events.

• 2-User requested versus coerced (coerced caused by hardware that is not under user control).

• 3-User maskable versus nonmaskable (disabled or not)

• 4- Within versus between instructions: in the middle of instruction execution (harder), VS between instructions

• 5-Resume versus terminate: If program stops after the interrupt, it terminate. If program continues execution it resumes. Terminate is easier to implement.

It is harder to implement interrupts that require to resume if they are within instruction.

N. Mekhiel
Handling Interrupts

- Collect State
- Correct cause of interrupt
- Restore state
- Try instruction in progress

Handling Interrupts in Pipeline It is difficult to handle interrupts in pipeline architecture because:
1- They might occur within instruction (EX or MEM stages)
2- They must be restartable. For example if virtual memory page fault result from data fetch (MEM stage), By the time it is seen, several other instructions will be executed. The page fault must be restartable.

N. Mekhiel
Handling interrupts in pipeline requires the following:-

- Force a trap into pipeline on next IF
- Until trap is taken, turn off all writes for faulting instruction and all that follow
- After OS receives control, it saves PC of faulting instruction (return from exception)

When using delayed branch, then we need to save more than one PC value (taken and not taken).

**Precise Interrupts:**
If we can stop pipeline so that the instruction just before interrupt causing instruction completes and those after it can restart from scratch.
Some machines does not support precise interrupts.

N. Mekhiel
**Types of Exceptions on MIPS:**

<table>
<thead>
<tr>
<th>pipeline stage</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on fetch; memory protection</td>
</tr>
<tr>
<td>ID</td>
<td>Illegal OP Code</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic fault</td>
</tr>
<tr>
<td>MEM</td>
<td>Data fetch page fault; memory protection</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

**Handling Exceptions in MIPS:**

Multiple exceptions might occur in same clock cycle from pipeline stages (MEM and EX) or they might occur out of order. Exceptions must be handled as in nonpipelined datapath (in order). Hardware posts each interrupt in a status vector associated with each instruction as it goes down the pipeline. If the status vector indicates an exception, any control signal that may cause data to be written must be turned off (REG, MEM). When instruction enters WB, status vector is checked and any exception if set are handled as if they would occur in a nonpipelined machine.

N. Mekhiel
Extending MIPS Pipeline to Handle Multi-cycle Operations

FP operations take longer to perform, need to allow pipeline to handle longer operations by repeating EX cycle. Can use multiple floating point function units to allow overlapping of operations.

Modified MIPS Pipeline

Function units allow pipeline operations and multiple operations

Each function unit has different latency and throughput (pipelined/non pipelined) as:-

- ALU INT unit has 1 cycle EX, latency=0
- FP Multiply has 7 cycles EX, latency=6
- FP Add has 4 cycles EX, latency=3
- FP Divide has 25 cycles EX, latency=24 (non-pipelined)

N. Mekhial
MultiCycle Pipeline

N. Mekhiel
Pipeline with Multiple Cycle Function Units

N. Mekhied
The Problems with variable length Pipeline to support Multicycle:

- Structural hazards for using nonpipelined Divide function unit. Any other Divide instruction must wait until the first Divide instruction completes.

- Increase the frequency of RAW hazards because the WB is delayed.

- Structural hazard due to having two writes at the same cycle. FP Register file has one write port.

- WAW hazards are possible

- Instructions can complete out of order affecting exceptions

N. Mekhiel
### Example of Multi-Cycle Pipeline

<table>
<thead>
<tr>
<th>Instruction</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F4, 0(R2)</td>
<td>F1</td>
<td>D1</td>
<td>EX1</td>
<td>ME1</td>
<td>WB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multd F0, F4, F6</td>
<td>F2</td>
<td>D2</td>
<td>STL</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>ME2</td>
<td>WB2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F2, F0, F8</td>
<td>F3</td>
<td>STL</td>
<td>D3</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>ME3</td>
<td>WB3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD F2, 0(R2)</td>
<td></td>
<td>STL</td>
<td>F4</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>STL</td>
<td>D4</td>
<td>EX4</td>
<td>STL</td>
<td>STL</td>
<td>ME4</td>
<td>WB4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N. Mekhiel
**Problems with Multicycle Pipeline:**

**Structural Hazard (one write port in FP RF)**

<table>
<thead>
<tr>
<th>Instruction</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multd F0, F4, F6</td>
<td></td>
<td>F1</td>
<td>D1</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>ME1</td>
<td>WB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction2</td>
<td></td>
<td>F2</td>
<td>D2</td>
<td>EX2</td>
<td>ME2</td>
<td>WB2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction3</td>
<td></td>
<td>F3</td>
<td>D3</td>
<td>EX3</td>
<td>ME3</td>
<td>WB3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F2, F0, F8</td>
<td></td>
<td>F4</td>
<td>D4</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>ME4</td>
<td>WB4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STL</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>ME4</td>
<td>WB4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

 Structural Hazard (one write port)

N. Mekhiel
Dealing with structural hazard: two writes at same time

• 1-Use shift register that indicates when an instruction that has been issued can use the RF. If an instruction is in ID needs to use RF the same time as the issued instruction, the instruction inID is stalled for one cycle. Shift one bit the register and repeat the above.

• 2- Second scheme is to Stall a conflicting instruction at MEM stage. Simpler but not optimal as it might allow stalls to occur from more than one place complicating control.

N. Mekhiel
WAW Hazard in Multicycle Pipeline

Example of WAW Hazard

<table>
<thead>
<tr>
<th>Instruction</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>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD F2, F0, F8</td>
<td>F1</td>
<td>D1</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>ME1</td>
<td>WB1</td>
</tr>
<tr>
<td>Instruction2</td>
<td>F2</td>
<td>D2</td>
<td>EX2</td>
<td>ME2</td>
<td>WB2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F2, 0(R2)</td>
<td>F3</td>
<td>D3</td>
<td>EX3</td>
<td>ME3</td>
<td>WB3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dealing With WAW Hazard

- Delay Issuing LD instruction until ADD is in MEM stage
- STAMP out ADD results, so it does not write and let LD issue right away

Detection of WAW

If an instruction in ID wants to write to same register as an instruction already issued.

N. Mekhiel
Before Issuing FP Instruction, Control Check the following:

- Check for structural hazards. Wait until required function unit is not busy (Divide, one write port).

- Check for RAW hazards. Wait until source registers are not listed as destination by any Exec unit.

- Check WAW hazards. If any instruction in pipeline has same destination as this instruction, stall issue in ID.

- Check forwarding. If destination register in MEM/WB or EX/MEM or A4/MEM, M7/MEM D/MEM is source register of FP instruction, activate MUX to forward it.

N. Mekhiel
Maintaining Precise Exceptions with MIPS Multiple Cycle Execution Units

Example:
DIVF F0, F2, F4 ; finishes at C28
ADDF F10, F10, F8 ; finishes at C9
SUBF F12, F12, F14 ; finishes at C14

Example:
DIVF F0, F2, F4 ___________________________ Cycle28
ADDF F10, F10, F8 ——— Cycle9
SUBF F12, F12, F14 ___________________________ Cycle14

Complication for precise exception due to out of order completion

Instructions ADDF, SUBF complete before DIVF OUT OF ORDER.
No problem because there is no data dependency, but will result in imprecise exception.

N. Mekhiel
Example
If SUBF causes FP Exception, when ADDF completed and has overwritten its source F10 before DIVF complete, it is very difficult to restore the state to what it was before.

Dealing with Exception
1-Ignore the problem and settle for imprecise exception
2-Buffer results until all operations that were issued earlier are complete.
3-Allow exceptions to become somewhat imprecise, but keep enough information for trap handler to create precise exception (software)
4-Hybrid scheme that allows instructions to issue if all previous instructions are guaranteed to complete without causing exception

Performance of FP Pipeline: FP SPEC benchmarks, total number of stalls/ FP instruction ranges from .65 to 1.21 cycles.

N. Mekhiel
A.1.a

Assume R3=R2+396

Find timing without forwarding and flush branch.

L: LW R1, 0(R2) F1D1E1M1W1
ADDI R1, R1, #1 F2D2S-D2E2M2W2
SW R1, 0(R2) F3D3S-D3E3M3W3
ADDI R2, R2, #4 F4D4E4M4W4
SUB R4, R3, R2 F5D5S-D5E5M5W5
BNZ R4, Loop F6D6S-D6E6M6W6

1 2 3 4 5 6 7 8 9 1011121314151617

Time=18+ 17x98
A.1.b

with forwarding

Loop: LW R1, 0(R2) F1D1E1M1W1
    ADDI R1, R1, #1 F2D2S–E2M2W2
    SW R1, 0(R2) F3S–D3E3M3W3
    ADDI R2, R2, #4 F4D4E4M4W4
    SUB R4, R3, R2 F5D5E5M5W5
    BNZ R4, Loop F6D6E6M6W6
            FN
    1 2 3 4 5 6 7 8 9 10

Time= 11 + 10x98

{\bf A.1.c}
Use forwarding + Scheduling
Loop: LW R1, 0(R2)
    ADDI R2, R2, #4
    ADDI R1, R1, #1
    SUB R4, R3, R2
    BNZ R4, Loop
    SW R1, –4(R2)

    time=10 + 6x98

N. Mekhilef
A.2

Show the timing of FP Pipeline with no forwarding Branch after D
LD F0, 0(R2) FDEMW
LD F4, 0(R3) FDEMW
MUL F0, F0, F4 FDSDEEEEEEMW
ADD F2, F0, F2 FSSSSSSSSDEEEEEMW
ADD R2, R2, #8 FDEMW
ADD R3, R3, #8 FDEMW
SUB R5, R4, R2 FSDEMW
BNZ R5, Loop FSSDEMW

1234567891234567891234

Time=9+9+5=22
A.2

Show the timing of FP Pipeline with forwarding.

Branch after D

LD F0, 0(R2) FDEMW
LD F4, 0(R3) FDEMW
MUL F0,F0,F4 FDSEEEEEEMW
ADD F2,F0,F2 FSDSSSSSEEEEEEMW
ADD R2,R2,#8 FSSSSSSDEMW
ADD R3,R3,#8 FDEMW
SUB R5,R4,R2 FDSEMW
BNZ R5, Loop FSDEMW

F

123456789123456789

Time=9+9=18
A.3 Forwarding and scheduling and delayed branch

LD F0, 0(R2) FDEMW
LD F4, 0(R3) FDEMW
ADD R2,R2,#8 FDEMW
MUL F0,F0,F4 FDEEEEEEEEMW
SUB R5,R4,R2 FDEMW
ADD F2,F0,F2 FDSSSSSEEEEMW
BNZ R5, Loop FSSSSSDEMW
ADD R3,R3,#8 FDEMW
FSDEMW

12345678912345

Time=9+5=14