## DLX Instruction Format

### I-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs1</th>
<th>rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

Encodes: Loads and stores of bytes, words, half words. All immediates (rd ← rs1 op immediate)

Conditional branch instructions (rs1 is register, rd unused)

Jump register, jump and link register (rd = 0, rs = destination, immediate = 0)

### R-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs1</th>
<th>rs2</th>
<th>rd</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>11 bits</td>
</tr>
</tbody>
</table>

Register-register ALU operations: rd ← rs1 func rs2
Function encodes the data path operation:
Add, Sub .. Read/write special registers and moves.

### J-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

Jump and jump and link. Trap and return from exception
A Basic Multi-Cycle Implementation of DLX

- Every integer DLX instruction can be implemented in at most five clock cycles:

1. Instruction fetch cycle (IF):

   \[
   \begin{align*}
   IR & \leftarrow \text{Mem}[PC] \\
   \text{NPC} & \leftarrow PC + 4
   \end{align*}
   \]

2. Instruction decode/register fetch cycle (ID):

   \[
   \begin{align*}
   A & \leftarrow \text{Regs}[IR_{6..10}]; \\
   B & \leftarrow \text{Regs}[IR_{11..15}]; \\
   \text{Imm} & \leftarrow ((IR_{16})^{16}\#\text{IR}_{16..31})
   \end{align*}
   \]

Note: IR (instruction register), NPC (next sequential program counter register)
A, B, Imm are temporary registers
A Basic Implementation of DLX (continued)

3 Execution/Effective address cycle (EX):

- Memory reference:
  \[ \text{ALUOutput} \leftarrow A + \text{Imm}; \]

- Register-Register ALU instruction:
  \[ \text{ALUOutput} \leftarrow A \text{ func } B; \]

- Register-Immediate ALU instruction:
  \[ \text{ALUOutput} \leftarrow A \text{ op Imm}; \]

- Branch:
  \[ \text{ALUOutput} \leftarrow \text{NPC + Imm}; \]
  \[ \text{Cond} \leftarrow (A \text{ op } 0) \]
4 Memory access/branch completion cycle (MEM):

- Memory reference:
  
  \[
  \text{LMD} \leftarrow \text{Mem[ALUOutput]} \quad \text{or} \quad \text{Mem[ALUOutput]} \leftarrow B; \]

- Branch:
  
  \[
  \text{if (cond) } \text{PC} \leftarrow \text{ALUOutput} \quad \text{else} \quad \text{PC} \leftarrow \text{NPC} \]

Note: LMD (load memory data) register
A Basic Implementation of DLX (continued)

5 Write-back cycle (WB):

- Register-Register ALU instruction:

  \[ \text{Regs[IR}_{16..20}] \leftarrow \text{ALUOutput}; \]

- Register-Immediate ALU instruction:

  \[ \text{Regs[IR}_{11..15}] \leftarrow \text{ALUOutput}; \]

- Load instruction:

  \[ \text{Regs[IR}_{11..15}] \leftarrow \text{LMD}; \]

Note: LMD (load memory data) register
A Multi-Cycle DLX Datapath Implementation

FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.
Pipelining: Definitions

• Pipelining is an implementation technique where multiple operations on a number of instructions are overlapped in execution.
• An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction.
• Each step is called a *pipe stage* or a *pipe segment*.
• The stages or steps are connected one to the next to form a pipe -- instructions enter at one end and progress through the stage and exit at the other end.
• Throughput of an instruction pipeline is determined by how often an instruction exists the pipeline.
• The time to move an instruction one step down the line is equal to the *machine cycle* and is determined by the stage with the longest processing delay.
Pipelining: Design Goals

• The length of a machine clock cycle is determined by the time required for the slowest pipe stage.

• An important pipeline design consideration is to balance the length of each pipeline stage.

• If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

\[
\text{Time per instruction on unpipelined machine} \quad \frac{\text{Number of pipe stages}}{\text{Number of pipe stages}}
\]

• Under these ideal conditions:
  – Speedup from pipelining equals the number of pipeline stages: \( n \),
  – One instruction is completed every cycle, \( \text{CPI} = 1 \).
Simple DLX Pipelined Instruction Processing

<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>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
<td>EX</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
</tbody>
</table>

DLX Pipeline Stages:
- IF = Instruction Fetch
- ID = Instruction Decode
- EX = Execution
- MEM = Memory Access
- WB = Write Back

Time to fill the pipeline

First instruction, I completed

Last instruction, I+4 completed
FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.
A Pipelined DLX Datapath

- Obtained from multi-cycle DLX datapath by adding buffer registers between pipeline stages
- Assume register writes occur in first half of cycle and register reads occur in second half.

FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.
<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IF</strong></td>
<td>IF/ID.IR ← Mem[PC];&lt;br&gt;IF/ID.NPC,PC ← (if EX/MEM.cond (EX/MEM.NPC) else (PC+4));</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>ID/EX.A ← Regs[IF/ID.IR6..10]; ID/EX.B ← Regs[IF/ID.IR11..15];&lt;br&gt;ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;&lt;br&gt;ID/EX.Imm ← (IR16)^16##IR16..31;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ALU instruction</strong></th>
<th><strong>Load or store instruction</strong></th>
<th><strong>Branch instruction</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EX</strong></td>
<td>EX/MEM.IR ← ID/EX.IR;&lt;br&gt;EX/MEM.ALUOutput←&lt;br&gt;ID/EX.A op ID/EX.B;&lt;br&gt;EX/MEM.ALUOutput ←&lt;br&gt;ID/EX.A op ID/EX.Imm;&lt;br&gt;EX/MEM.cond ← 0;</td>
<td>EX/MEM.IR ← ID/EX.IR&lt;br&gt;EX/MEM.ALUOutput ←&lt;br&gt;ID/EX.A + ID/EX.Imm&lt;br&gt;EX/MEM.ALUOutput ←&lt;br&gt;ID/EX.NPC+ID/EX.Imm&lt;br&gt;EX/MEM.cond ← 0&lt;br&gt;EX/MEM.B ← ID/EX.B&lt;br&gt;(ID/EX.A op 0);&lt;br&gt;</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>MEM/WB.IR ← EX/MEM.IR;&lt;br&gt;MEM/WB.ALUOutput ←&lt;br&gt;EX/MEM.ALUOutput;</td>
<td>MEM/WB.IR ← EX/MEM.IR&lt;br&gt;MEM/WB.LMD ←&lt;br&gt;Mem[EX/MEM.ALUOutput];&lt;br&gt;or&lt;br&gt;Mem[EX/MEM.ALUOutput] ←&lt;br&gt;EX/MEM.B;&lt;br&gt;</td>
</tr>
<tr>
<td><strong>WB</strong></td>
<td>Regs[MEM/WB.IR16..20] ←&lt;br&gt;MEM/WB.ALUOutput;&lt;br&gt;or&lt;br&gt;Regs[MEM/WB.IR11..15] ←&lt;br&gt;MEM/WB.ALUOutput;</td>
<td>Regs[MEM/WB.IR11..15] ←&lt;br&gt;MEM/WB.LMD;</td>
</tr>
</tbody>
</table>

**FIGURE 3.5** Events on every pipe stage of the DLX pipeline.
Basic Performance Issues In Pipelining

• Pipelining increases the CPU instruction throughput: The number of instructions completed per unit time. Under ideal condition instruction throughput is one instruction per machine cycle, or CPI = 1

• Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).

• It usually slightly increases the execution time of each instruction over unpipelined implementations due to the increased control overhead of the pipeline and pipeline stage registers delays.
Pipelining Performance Example

- Example: For an unpipelined machine:
  - Clock cycle = 10ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.
  - If pipelining adds 1ns to the machine clock cycle then the speedup in instruction execution from pipelining is:

Non-pipelined Average instruction execution time = Clock cycle x Average CPI

= 10 ns x ((40% + 20%) x 4 + 40%x 5) = 10 ns x 4.4 = 44 ns

In the pipelined five implementation five stages are used with an average instruction execution time of: 10 ns + 1 ns = 11 ns

Speedup from pipelining = Instruction time unpipelined / Instruction time pipelined

= 44 ns / 11 ns = 4 times
Pipeline Hazards

- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle.

- Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  - *Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - *Data hazards*: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - *Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC.
Performance of Pipelines with Stalls

• Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction

• If pipelining overhead is ignored and we assume that the stages are perfectly balanced then:

Speedup = CPI unpipelined / (1 + Pipeline stall cycles per instruction)

• When all instructions take the same number of cycles and is equal to the number of pipeline stages then:

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)
Performance of Pipelines with Stalls

• If we think of pipelining as improving the effective clock cycle time, then given the CPI for the unpipelined machine and the CPI of the ideal pipelined machine = 1, then effective speedup of a pipeline with stalls over the unpipelined case is given by:

\[
\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles}} \times \frac{\text{Clock cycles unpipelined}}{\text{Clock cycle pipelined}}
\]

• When pipe stages are balanced with no overhead, the clock cycle for the pipelined machine is smaller by a factor equal to the pipelined depth:

\[
\text{Clock cycle pipelined} = \frac{\text{clock cycle unpipelined}}{\text{pipeline depth}}
\]

\[
\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{clock cycle pipelined}}
\]

\[
\text{Speedup} = \frac{1}{1 + \text{pipeline stall cycles per instruction}} \times \text{pipeline depth}
\]
Structural Hazards

• In pipelined machines overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

• If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:
  – when a machine has only one register file write port
  – or when a pipelined machine has a shared single-memory pipeline for data and instructions.
→ stall the pipeline for one cycle for register writes or memory data access
DLX with Memory Unit Structural Hazards

3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.
Resolving A Structural Hazard with Stalling

FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.
A Structural Hazard Example

• Given that data references are 40% for a specific instruction mix or program, and that the ideal pipelined CPI ignoring hazards is equal to 1.

• A machine with a data memory access structural hazards requires a single stall cycle for data references and has a clock rate 1.05 times higher than the ideal machine. Ignoring other performance losses for this machine:

Average instruction time = CPI X Clock cycle time

Average instruction time = \((1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05}\)

= 1.3 \times \text{Clock cycle time}_{\text{ideal}}
Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined machine resulting in incorrect execution.

- Data hazards usually require one or more instructions to be stalled to ensure correct execution.

- Example:

```
  ADD   R1, R2, R3
  SUB   R4, R1, R5
  AND   R6, R1, R7
  OR    R8, R1, R9
  XOR   R10, R1, R11
```

  - All the instructions after ADD use the result of the ADD instruction
  - SUB, AND instructions need to be stalled for correct execution.
Figure 3.9 The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.
Minimizing Data hazard Stalls by Forwarding

• Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.

• Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)

• For example, in the DLX pipeline with forwarding:
  – The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
  – Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed.
  – If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.
Pipelined DLX with Forwarding

3.10 A set of instructions that depend on the ADD result use forwarding paths to avoid the data hazard.
Load/Store Forwarding Example

FIGURE 3.11 Stores require an operand during MEM, and forwarding of that operand is shown here.
Data Hazard Classification

Given two instructions $I$, $J$, with $I$ occurring before $J$ in an instruction stream:

- **RAW** (read after write): *A true data dependence*
  $J$ tried to read a source before $I$ writes to it, so $J$ incorrectly gets the old value.

- **WAW** (write after write): *A name dependence*
  $J$ tries to write an operand before it is written by $I$.
  The writes end up being performed in the wrong order.

- **WAR** (write after read): *A name dependence*
  $J$ tries to write to a destination before it is read by $I$,
  so $I$ incorrectly gets the new value.

- **RAR** (read after read): Not a hazard.
Data Hazard Classification

- **I (Write)**
  - **J (Read)**
    - Shared Operand
    - Read after Write (RAW)

- **I (Write)**
  - **J (Write)**
    - Shared Operand
    - Write after Write (WAW)

- **I (Read)**
  - **J (Write)**
    - Shared Operand
    - Write after Read (WAR)

- **I (Read)**
  - **J (Read)**
    - Shared Operand
    - Read after Read (RAR)

Not a hazard
Data Hazards Present in Current DLX Pipeline

- **Read after Write (RAW) Hazards:** Possible?
  - Results from true data dependencies between instructions.
  - Yes possible, when an instruction requires an operand generated by a preceding instruction with distance less than four.
  - Resolved by:
    - Forwarding or Stalling.

- **Write after Read (WAR):**
  - Results when an instruction overwrites the result of an instruction before all preceding instructions have read it.

- **Write after Write (WAW):**
  - Results when an instruction writes into a register or memory location before a preceding instruction have written its result.

- **Possible? Both WAR and WAW are impossible in the current pipeline. Why?**
  - Pipeline processes instructions in the same sequential order as in the program.
  - All instruction operand reads are completed before a following instruction overwrites the operand.
    - Thus WAR is impossible in current DLX pipeline.
  - All instruction result writes are done in the same program order.
    - Thus WAW is impossible in current DLX pipeline.
Data Hazards Requiring Stall Cycles

• In some code sequence cases, potential data hazards cannot be handled by bypassing. For example:

  LW    R1, 0 (R2)
  SUB   R4, R1, R5
  AND   R6, R1, R7
  OR    R8, R1, R9

• The LW (load word) instruction has the data in clock cycle 4 (MEM cycle).

• The SUB instruction needs the data of R1 in the beginning of that cycle.

• Hazard prevented by hardware pipeline interlock causing a stall cycle.
FIGURE 3.12 The load instruction can bypass its results to the AND and OR instructions, but not to the SUB, since that would mean forwarding the result in "negative time."
Hardware Pipeline Interlocks

• A hardware pipeline interlock detects a data hazard and stalls the pipeline until the hazard is cleared.

• The CPI for the stalled instruction increases by the length of the stall.

• For the Previous example, (no stall cycle):

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, 0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R4,R1,R5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND R6,R1,R7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Stall Cycle:

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, 0(R1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R4,R1,R5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND R6,R1,R7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIGURE 3.13 The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle.
<table>
<thead>
<tr>
<th>Situation</th>
<th>Example code sequence</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>LW R1, 45 (R2)</td>
<td>No hazard possible because no dependence exists on R1 in the immediately following three instructions.</td>
</tr>
<tr>
<td></td>
<td>ADD R5, R6, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB R8, R6, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
<td></td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>LW R1, 45 (R2)</td>
<td>Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.</td>
</tr>
<tr>
<td></td>
<td>ADD R5, R1, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB R8, R6, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
<td></td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LW R1, 45 (R2)</td>
<td>Comparators detect use of R1 in SUB and forward result of load to ALU in time for SUB to begin EX.</td>
</tr>
<tr>
<td></td>
<td>ADD R5, R6, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB R8, R1, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR R9, R6, R7</td>
<td></td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>LW R1, 45 (R2)</td>
<td>No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.</td>
</tr>
<tr>
<td></td>
<td>ADD R5, R6, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUB R8, R6, R7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OR R9, R1, R7</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 3.17** Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions.
<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison (if equal then forward)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR₁₆..₂₀ = ID/EX.IR₆..₁₀</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR₁₆..₂₀ = ID/EX.IR₁₁..₁₅</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR₁₆..₂₀ = ID/EX.IR₆..₁₀</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR₁₆..₂₀ = ID/EX.IR₁₁..₁₅</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR₁₁..₁₅ = ID/EX.IR₆..₁₀</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR₁₁..₁₅ = ID/EX.IR₁₁..₁₅</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR₁₁..₁₅ = ID/EX.IR₆..₁₀</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR₁₁..₁₅ = ID/EX.IR₁₁..₁₅</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR₁₁..₁₅ = ID/EX.IR₆..₁₀</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR₁₁..₁₅ = ID/EX.IR₁₁..₁₅</td>
</tr>
</tbody>
</table>

**FIGURE 3.19** Forwarding of data to the two ALU inputs (for the instruction in EX) can occur from the ALU result (in EX/MEM or in MEM/WB) or from the load result in MEM/WB.
FIGURE 3.20 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.
Compiler Instruction Scheduling for Data Hazard Stall Reduction

- Many types of stalls resulting from data hazards are very frequent. For example:

  \[
  A = B + C
  \]

produces a stall when loading the second data value (B).

- Rather than allow the pipeline to stall, the compiler could sometimes schedule the pipeline to avoid stalls.

- Compiler pipeline or instruction scheduling involves rearranging the code sequence (instruction reordering) to eliminate the hazard.
Compiler Instruction Scheduling Example

- For the code sequence:
  \[ a = b + c \]
  \[ d = e - f \]

- Assuming loads have a latency of one clock cycle, the following code or pipeline compiler schedule eliminates stalls:

Original code with stalls:

- LW Rb,b
- LW Rc,c
- ADD Ra,Rb,Rc
- SW a,Ra
- LW Re,e
- LW Rf,f
- SUB Rd,Re,Rf
- SW d,Rd

Scheduled code with no stalls:

- LW Rb,b
- LW Rc,c
- ADD Re,e
- LW Re,e
- ADD Ra,Rb,Rc
- LW Rf,f
- SW a,Ra
- SUB Rd,Re,Rf
- SW d,Rd
Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known.

- In current DLX pipeline, the conditional branch is resolved in the MEM stage resulting in three stall cycles as shown below:

```
<table>
<thead>
<tr>
<th>Branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch successor</td>
<td>IF</td>
<td>stall</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Branch successor + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch successor + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>Branch successor + 3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch successor + 4</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch successor + 5</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three clock cycles are wasted for every branch for current DLX pipeline
Reducing Branch Stall Cycles

Pipeline hardware measures to reduce branch stall cycles:

1- Find out whether a branch is taken earlier in the pipeline.
2- Compute the taken PC earlier in the pipeline.

In DLX:

– In DLX branch instructions BEQZ, BNEZ, test a register for equality to zero.
– This can be completed in the ID cycle by moving the zero test into that cycle.
– Both PCs (taken and not taken) must be computed early.
– Requires an additional adder because the current ALU is not useable until EX cycle.
– This results in just a single cycle stall on branches.
Modified DLX Pipeline: Conditional Branches Completed in ID Stage

FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.
Compile-Time Reduction of Branch Penalties

• One scheme discussed earlier is to flush or freeze the pipeline by whenever a conditional branch is decoded by holding or deleting any instructions in the pipeline until the branch destination is known (zero pipeline registers, control lines)).

• Another method is to predict that the branch is not taken where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next instruction; stall occurs here when the branch is taken.

• Another method is to predict that the branch is taken and begin fetching and executing at the target; stall occurs here if the branch is not taken.
# Predict Branch Not-Taken Scheme

<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 $i + 1$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 2$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 3$</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction $i + 4$</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 $i + 1$</td>
<td>IF</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
<td>idle</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>WB</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 3.26** The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).
Static Compiler Branch Prediction

Two basic methods exist to statically predict branches at compile time:

1. By examination of program behavior and the use of information collected from earlier runs of the program.
   - For example, a program profile may show that most forward branches and backward branches (often forming loops) are taken. The simplest scheme in this case is to just predict the branch as taken.

2. To predict branches on the basis of branch direction, choosing backward branches as taken and forward branches as not taken.
FIGURE 3.25 Together the forward and backward taken branches account for an average of 67% of all conditional branches.
Profile-Based Compiler Branch Misprediction Rates

FIGURE 3.36 Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.
FIGURE 3.37  Accuracy of a predict-taken strategy and a profile-based predictor as measured by the number of instructions executed between mispredicted branches and shown on a log scale.
Reduction of Branch Penalties: Delayed Branch

• When delayed branch is used, the branch is delayed by $n$ cycles, following this execution pattern:
  conditional branch instruction
  sequential successor$_1$
  sequential successor$_2$
  ........
  sequential successor$_n$
  branch target if taken

• The sequential successor instruction are said to be in the branch delay slots. These instructions are executed whether or not the branch is taken.

• In Practice, all machines that utilize delayed branching have a single instruction delay slot.

• The job of the compiler is to make the successor instructions valid and useful instructions.
Delayed Branch Example

<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>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 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>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</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>WB</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 3.27** The behavior of a delayed branch is the same whether or not the branch is taken.
Delayed Branch-delay Slot Scheduling Strategies

The branch-delay slot instruction can be chosen from three cases:

A An independent instruction from before the branch:
   Always improves performance when used. The branch must not depend on the rescheduled instruction.

B An instruction from the target of the branch:
   Improves performance if the branch is taken and may require instruction duplication. This instruction must be safe to execute if the branch is not taken.

C An instruction from the fall through instruction stream:
   Improves performance when the branch is not taken. The instruction must be safe to execute when the branch is taken.

The performance and usability of cases B, C is improved by using a canceling or nullifying branch.
FIGURE 3.28 Scheduling the branch-delay slot.
Branch-delay Slot: Canceling Branches

• In a canceling branch, a static compiler branch direction prediction is included with the branch-delay slot instruction.

• When the branch goes as predicted, the instruction in the branch delay slot is executed normally.

• When the branch does not go as predicted the instruction is turned into a no-op.

• Canceling branches eliminate the conditions on instruction selection in delay instruction strategies B, C

• The effectiveness of this method depends on whether we predict the branch correctly.
<table>
<thead>
<tr>
<th>Scheduling strategy</th>
<th>Requirements</th>
<th>Improves performance when?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) From before branch</td>
<td>Branch must not depend on the rescheduled instructions.</td>
<td>Always.</td>
</tr>
<tr>
<td>(b) From target</td>
<td>Must be OK to execute rescheduled instructions if branch is not taken. May</td>
<td>When branch is taken. May enlarge program if instructions are duplicated.</td>
</tr>
<tr>
<td></td>
<td>need to duplicate instructions.</td>
<td></td>
</tr>
<tr>
<td>(c) From fall through</td>
<td>Must be OK to execute instructions if branch is taken.</td>
<td>When branch is not taken.</td>
</tr>
</tbody>
</table>

**FIGURE 3.29** Delayed-branch scheduling schemes and their requirements.

<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>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 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>Branch delay instruction ((i + 1))</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</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>WB</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 3.30** The behavior of a predicted-taken cancelling branch depends on whether the branch is taken or not.
### DLX Performance Using Canceling Delay Branches

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% conditional branches</th>
<th>% conditional branches with empty slots</th>
<th>% conditional branches that are cancelling</th>
<th>% cancelling branches that are cancelled</th>
<th>% branches with cancelled delay slots</th>
<th>Total % branches with empty or cancelled delay slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>14%</td>
<td>18%</td>
<td>31%</td>
<td>43%</td>
<td>13%</td>
<td>31%</td>
</tr>
<tr>
<td>eqntott</td>
<td>24%</td>
<td>24%</td>
<td>50%</td>
<td>24%</td>
<td>12%</td>
<td>36%</td>
</tr>
<tr>
<td>espresso</td>
<td>15%</td>
<td>29%</td>
<td>19%</td>
<td>21%</td>
<td>4%</td>
<td>33%</td>
</tr>
<tr>
<td>gcc</td>
<td>15%</td>
<td>16%</td>
<td>33%</td>
<td>34%</td>
<td>11%</td>
<td>27%</td>
</tr>
<tr>
<td>li</td>
<td>15%</td>
<td>20%</td>
<td>55%</td>
<td>48%</td>
<td>26%</td>
<td>46%</td>
</tr>
<tr>
<td><strong>Integer average</strong></td>
<td><strong>17%</strong></td>
<td><strong>21%</strong></td>
<td><strong>38%</strong></td>
<td><strong>34%</strong></td>
<td><strong>13%</strong></td>
<td><strong>35%</strong></td>
</tr>
<tr>
<td>doudc</td>
<td>8%</td>
<td>33%</td>
<td>12%</td>
<td>62%</td>
<td>8%</td>
<td>41%</td>
</tr>
<tr>
<td>ear</td>
<td>10%</td>
<td>37%</td>
<td>36%</td>
<td>14%</td>
<td>5%</td>
<td>42%</td>
</tr>
<tr>
<td>hydro2d</td>
<td>12%</td>
<td>0%</td>
<td>69%</td>
<td>24%</td>
<td>16%</td>
<td>17%</td>
</tr>
<tr>
<td>mdljdp2</td>
<td>9%</td>
<td>0%</td>
<td>86%</td>
<td>10%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>su2cor</td>
<td>3%</td>
<td>7%</td>
<td>17%</td>
<td>57%</td>
<td>10%</td>
<td>17%</td>
</tr>
<tr>
<td>FP average</td>
<td>8%</td>
<td>16%</td>
<td>44%</td>
<td>34%</td>
<td>9%</td>
<td>25%</td>
</tr>
<tr>
<td><strong>Overall average</strong></td>
<td><strong>12%</strong></td>
<td><strong>18%</strong></td>
<td><strong>41%</strong></td>
<td><strong>34%</strong></td>
<td><strong>11%</strong></td>
<td><strong>30%</strong></td>
</tr>
</tbody>
</table>

FIGURE 3.31 Delayed and cancelling delay branches for DLX allow branch hazards to be hidden 70% of the time on average for these 10 SPEC benchmarks.
Performance of Branch Schemes

- The effective pipeline speedup with branch penalties:
  (assuming an ideal pipeline CPI of 1)

\[
\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}
\]

Pipeline stall cycles from branches = Branch frequency \times branch penalty

\[
\text{Pipeline speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}
\]
Pipeline Performance Example

• Assume the following DLX instruction mix:

<table>
<thead>
<tr>
<th>Type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>40%</td>
</tr>
<tr>
<td>Load</td>
<td>30% of which 25% are followed immediately by an instruction using the loaded value</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
</tr>
<tr>
<td>branch</td>
<td>20% of which 45% are taken</td>
</tr>
</tbody>
</table>

• What is the resulting CPI for the pipelined DLX with forwarding and branch address calculation in ID stage when using a branch not-taken scheme?

• CPI = Ideal CPI + Pipeline stall clock cycles per instruction

\[
\text{CPI} = 1 + \text{stalls by loads} + \text{stalls by branches}
\]

\[
= 1 + .3 \times .25 \times 1 + .2 \times .45 \times 1
\]

\[
= 1 + .075 + .09
\]

\[
= 1.165
\]
Branch Penalty Example

- For a pipeline similar to the MIPS R4000, it takes three pipeline stages before the branch target address is known and an additional cycle before the branch condition is evaluated.

- Assuming no stalls on the registers in the conditional comparison. The branch penalty for the three simplest branch prediction schemes:

<table>
<thead>
<tr>
<th>Branch Scheme</th>
<th>Penalty unconditional</th>
<th>Penalty untaken</th>
<th>Penalty taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flush pipeline</td>
<td>2.0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Predict taken</td>
<td>2.0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Predict untaken</td>
<td>2.0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Pipelining and Handling of Exceptions

• Exceptions are events that usually occur in normal program execution where the normal execution order of the instructions is changed (often called: interrupts, faults).

• Types of exceptions include:
  • I/O device request
  • Invoking an operating system service
  • Tracing instruction execution
  • Breakpoint (programmer-requested interrupt).
  • Integer overflow or underflow
  • FP anomaly
  • Page fault (not in main memory)
  • Misaligned memory access
  • Memory protection violation
  • Undefined instruction
  • Hardware malfunctions
<table>
<thead>
<tr>
<th>Exception event</th>
<th>IBM 360</th>
<th>VAX</th>
<th>Motorola 680x0</th>
<th>Intel 80x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>Input/output interruption</td>
<td>Device interrupt</td>
<td>Exception (Level 0...7 autovector)</td>
<td>Vectored interrupt</td>
</tr>
<tr>
<td>Invoking the operating system service from a user program</td>
<td>Supervisor call interruption</td>
<td>Exception (change mode supervisor trap)</td>
<td>Exception (unimplemented instruction)--- on Macintosh</td>
<td>Interrupt (INT instruction)</td>
</tr>
<tr>
<td>Tracing instruction execution</td>
<td>Not applicable</td>
<td>Exception (trace fault)</td>
<td>Exception (trace)</td>
<td>Interrupt (single-step trap)</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Not applicable</td>
<td>Exception (breakpoint fault)</td>
<td>Exception (illegal instruction or breakpoint)</td>
<td>Interrupt (breakpoint trap)</td>
</tr>
<tr>
<td>Integer arithmetic overflow or underflow; FP trap</td>
<td>Program interruption (overflow or underflow exception)</td>
<td>Exception (integer overflow trap or floating underflow fault)</td>
<td>Exception (floating-point coprocessor errors)</td>
<td>Interrupt (overflow trap or math unit exception)</td>
</tr>
<tr>
<td>Page fault (not in main memory)</td>
<td>Not applicable (only in 370)</td>
<td>Exception (translation not valid fault)</td>
<td>Exception (memory-management unit errors)</td>
<td>Interrupt (page fault)</td>
</tr>
<tr>
<td>Misaligned memory accesses</td>
<td>Program interruption (specification exception)</td>
<td>Not applicable</td>
<td>Exception (address error)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Memory protection violations</td>
<td>Program interruption (protection exception)</td>
<td>Exception (access control violation fault)</td>
<td>Exception (bus error)</td>
<td>Interrupt (protection exception)</td>
</tr>
<tr>
<td>Using undefined instructions</td>
<td>Program interruption (operation exception)</td>
<td>Exception (opcode privileged/reserved fault)</td>
<td>Exception (illegal instruction or breakpoint/unimplemented instruction)</td>
<td>Interrupt (invalid opcode)</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Machine-check interruption</td>
<td>Exception (machine-check abort)</td>
<td>Exception (bus error)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Power failure</td>
<td>Machine-check interruption</td>
<td>Urgent interrupt</td>
<td>Not applicable</td>
<td>Nonmaskable interrupt</td>
</tr>
</tbody>
</table>

The names of common exceptions vary across four different architectures.
Characteristics of Exceptions

- **Synchronous vs. asynchronous:**
  - Synchronous: occurs at the same place with the same data and memory allocation
  - Asynchronous: Caused by devices external to the processor and memory.

- **User requested vs. coerced:**
  - User requested: The user task requests the event.
  - Coerced: Caused by some hardware event.

- **User maskable vs. user nonmaskable:**
  - User maskable: Can be disabled by the user task using a mask.

- **Within vs. between instructions:**
  - Whether it prevents instruction completion by happening in the middle of execution.

- **Resuming vs. terminating:**
  - Terminating: The program execution always stops after the event.
  - Resuming: the program continues after the event. The state of the pipeline must be saved to handle this type of exception. The pipeline is restartable in this case.
Handling of Resuming Exceptions

• A resuming exception (e.g. a virtual memory page fault) usually requires the intervention of the operating system.

• The pipeline must be safely shut down and its state saved for the execution to resume after the exception is handled as follows:

1. Force a trap instruction into the pipeline on the next IF.

2. Turn of all writes for the faulting instruction and all instructions in the pipeline. Place zeroes into pipeline latches starting with the instruction that caused the fault to prevent state changes.

3. The execution handling routine of the operating system saves the PC of the faulting instruction and other state data to be used to return from the exception.
Exception Handling Issues

- When using delayed branches, as many PCs as the length of the branch delay plus one need to be saved and restored to restore the state of the machine.

- After the exception has been handled special instructions are needed to return the machine to the state before the exception occurred (RFE, Return to User code in DLX).

- **Precise exceptions** imply that a pipeline is stopped so the instructions just before the faulting instruction are completed and those after it can be restarted from scratch.

- Machines with arithmetic trap handlers and demand paging must support precise exceptions.
Exceptions in DLX

The following represent problem exceptions for the DLX pipeline stages:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Exception Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch; misaligned memory access; memory-protection violation.</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data fetch; misaligned memory access; memory-protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

Example: LW IF ID EX MEM WB
ADD IF ID EX MEM WB
can cause a data page fault and an arithmetic exception at the same time (LW in MEM and ADD in EX)
Handled by dealing with data page fault and then restarting execution, then the second exception will occur but not the first.
Precise Exception Handling in DLX

• The instruction pipeline is required to handle exceptions of instruction $i$ before those of instruction $i+1$

• The hardware posts all exceptions caused by an instruction in a status vector associated with the instruction which is carried along with the instruction as it goes through the pipeline.

• Once an exception indication is set in the vector, any control signals that cause a data value write is turned off.

• When an instruction enters WB the vector is checked, if any exceptions are posted, they are handled in the order they would be handled in an unpipelined machine.

• Any action taken in earlier pipeline stages is invalid but cannot change the state of the machine since writes where disabled.