**DLX Instruction Format**

**I - type instruction**

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

6 bits 5 6 10 11 15 16 31

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

6 bits 5 6 10 11 15 16 20 31

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

6 bits 5 6 31

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):
   - IR ← Mem[PC]
   - NPC ← PC + 4

2. Instruction decode/register fetch cycle (ID):
   - A ← Regs[IR_{6..10}];
   - B ← Regs[IR_{11..15}];
   - Imm ← ((IR_{16})^{16}##IR_{16..31})

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:
  \[ \text{ALUOutput} \ NPC + \text{Imm}; \]
  \[ \text{Cond} \leftarrow (A \text{ op 0}) \]
A Basic Implementation of DLX (continued)

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:

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

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

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 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):

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

• 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>Clock Number</th>
<th>Time in clock cycles →</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Instruction I</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction I+1</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction I+2</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction I+3</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>Instruction I+4</td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

DLX Pipeline Stages:

- **IF** = Instruction Fetch
- **ID** = Instruction Decode
- **EX** = Execution
- **MEM** = Memory Access
- **WB** = Write Back

First instruction, I Completed

Last instruction, I+4 completed

Time to fill the pipeline
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>IF</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>ID</td>
<td>ID/EX.A ← Regs[IF/ID.IR6..10];&lt;br&gt;ID/EX.B ← Regs[IF/ID.IR11..15];&lt;br&gt;ID/EX.NPC ← IF/ID.NPC;&lt;br&gt;ID/EX.IR ← IF/ID.IR;&lt;br&gt;ID/EX.Imm ← (IR16)16##IR16..31;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU instruction</th>
<th>Load or store instruction</th>
<th>Branch instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX</td>
<td>EX/MEM.IR ← ID/EX.IR;</td>
<td>EX/MEM.IR ← ID/EX.IR</td>
</tr>
<tr>
<td></td>
<td>EX/MEM.ALUOutput ←</td>
<td>EX/MEM.ALUOutput ←</td>
</tr>
<tr>
<td></td>
<td>ID/EX.A op ID/EX.B;</td>
<td>ID/EX.A + ID/EX.Imm;</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>EX/MEM.NPC+ID. EX.IIm;</td>
</tr>
<tr>
<td></td>
<td>EX/MEM.ALUOutput ←</td>
<td>EX/MEM.cond ← 0;</td>
</tr>
<tr>
<td></td>
<td>ID/EX.A op ID/EX.Imm;</td>
<td>EX/MEM.B ← ID/EX.B;</td>
</tr>
<tr>
<td>MEM</td>
<td>MEM/WB.IR ← EX/MEM.IR;</td>
<td>MEM/WB.IR ← EX/MEM.IR;</td>
</tr>
<tr>
<td></td>
<td>MEM/WB.ALUOutput ←</td>
<td>MEM/WB.LMD ←</td>
</tr>
<tr>
<td></td>
<td>EX/MEM.ALUOutput;</td>
<td>Mem[EX/MEM.ALUOutput];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mem[EX/MEM.ALUOutput] ←</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EX/MEM.B;</td>
</tr>
<tr>
<td>WB</td>
<td>Regs[MEM/WB.IR16..20] ←</td>
<td>Regs[MEM/WB.IR11..15] ←</td>
</tr>
<tr>
<td></td>
<td>MEM/WB.ALUOutput;</td>
<td>MEM/WB.LMD;</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Regs[MEM/WB.IR11..15] ←</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM/WB.ALUOutput;</td>
<td></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 $\text{CPI} = 1$

• 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.

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}
\]

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

\[
\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}
\]

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

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}
\]
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
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:

\[
\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}
\]

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

\[
= 1.3 \times \text{Clock cycle time}_{\text{ideal}}
\]
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) → Shared Operand → J (Read) → Read after Write (RAW)

I (Write) → Shared Operand → J (Write) → Write after Write (WAW)

I (Read) → Shared Operand → J (Write) → Write after Read (WAR)

I (Read) → Shared Operand → J (Read) → 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.