Multicycle Pipeline Operations

Material may be added to this set.

Material Covered

Section 3.7.

Long-Latency Operations (Topics)

Typical long-latency instructions: floating point

Pipelined v. non-pipelined execution units

Initiation interval and latency

Placement in Chapter-3 DLX pipeline

Timing diagrams
Long-Latency Instructions (Operations)

Common Long-Latency Instructions

Fastest (shortest—but still long—latency): Floating-Point Add, Subtract, Conversions

DLX: addf, addd, cvti2f (convert integer to float), ltd (compare less-than of doubles), etc.

Intermediate Speed: Multiply

DLX: multd, multf.

Slowest Speed: Divide, Modulo, Square Root

DLX: divd, divf.
Implementation of Long-Latency Instructions

Implementation balances cost and performance.

Low Cost: Unpipelined, Single Functional Unit, Data Recirculates

Whole functional unit occupied by instruction during computation . . .
. . . so it can execute only one instruction at a time.

Intermediate Cost: Multiple Unpipelined Functional Units

Functional units occupied by instruction during computation . . .
. . . each can execute a different instruction.

Cost a multiple of single-unit cost.

Highest Cost: Pipelined Functional Unit

Functional unit pipelined, at best each stage can hold a different instruction.

Cost disadvantage depends on how unpipelined units implemented.
Floating Point in Chapter-3 DLX Implementation

Floating Point Functional Units

- **FP Add**
  
  Four stages, fully pipelined: Latency 3, Initiation Interval 1.

  Used for FP Add, FP Subtract, FP Comparisons, etc.

- **FP Multiply**
  
  Seven stages, fully pipelined: Latency 6, Initiation Interval 1.

  Used for FP Multiply and Integer Multiply.

- **FP Divide**
  
  Twenty five stages, unpipelined: Latency 24, Initiation Interval 24.
Hazards With Long-Latency Instructions in Chapter-3 Pipeline

Structural Hazards

Functional Unit Structural Hazards

Because an instruction can occupy a functional unit (*e.g.*, DIV) more than one cycle . . .

. . . a following instruction needing that unit may be stalled.

(Occurs when initiation interval greater than one.)

Register Write (MEM Stage) Structural Hazards

Because different units have different latencies . . .

. . . instructions that started at different times can finish at the same time . . .

. . . only one can write results (unless extra register file ports added).

Data Hazards

RAW Hazards

As with integer operations, result not ready in time.

With long-latency operations instructions may wait longer.
WAW Hazards

Occurs when two nearby instructions write same register ... ... and second instruction finishes first.

WAR Hazards

Cannot occur in Chapter-3 pipeline because instructions start in order.

Precise Exceptions

A headache because an instruction can be ready to write ... ... long before a preceding instruction raises an exception.
Handling Functional Unit Structural Hazards

Example, 4-cycle latency unpipelined divide.

Unless FU changed, instructions must be stalled to avoid hazard.

```
divid f0, f2, f4    IF  ID  DIV  DIV  DIV  DIV  DIV  WB
divid f6, f8, f10   IF  ID  ------------------>  DIV  DIV  DIV  DIV  WB
```

Hazard easily handled:

- Units provide a *ready-next-cycle* signal to ID stage.

- Instruction stalled if ready-next-cycle for needed unit is 0.
Eliminating Hazards

Provide more than one functional unit.

Example, provide two 4-cycle latency divide units, $DV_a$ and $DV_b$.

\[
\begin{array}{cccccccc}
\text{divd f0, f2, f4} & \text{IF} & \text{ID} & DV_a & DV_a & DV_a & DV_a & WB \\
\text{divd f6, f8, f10} & \text{IF} & \text{ID} & DV_b & DV_b & DV_b & DV_b & WB \\
\end{array}
\]

Pipeline functional unit.

Example, use 5-cycle latency, initiation interval 2, pipelined divide ... 
... and live with single stall cycle.

\[
\begin{array}{cccccccc}
\text{divd f0, f2, f4} & \text{IF} & \text{ID} & DVO & DVO & DV1 & DV1 & DV2 & WB \\
\text{divd f6, f8, f10} & \text{IF} & \text{ID} & --> & DVO & DVO & DV1 & DV1 & DV2 & WB \\
\end{array}
\]
Handling Register Write Structural Hazards

Example (stall to avoid hazard in cycle 8)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</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>multd</td>
<td>f0, f2, f4</td>
<td>IF</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>WB</td>
</tr>
<tr>
<td>addi</td>
<td>r1, r1, #1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addd</td>
<td>f6, f8, f10</td>
<td>IF</td>
<td>ID</td>
<td>--</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>
Handling Register Write Structural Hazards

Method 1: Delay instruction in ID. (Used above.)

Include a shift register called a reservation register.

Each cycle the reservation register is shifted.

A 1 indicates a “reservation” to enter WB.

Bit position indicates time . . .

. . . with the LSB indicating two cycles later . . .

. . . the next bit indicating three cycles later . . .

. . . and so on.

The ID stage controller, based on the opcode of the instruction . . .

. . . knows the number of cycles before WB will be entered.

It checks the corresponding reservation register bit . . .

. . . if it’s 1 then IF and ID are stalled . . .

. . . if it’s 0 then the bit is set to 1 and the instruction proceeds.

If such a stall occurs the reservation register is still shifted . . .

. . . and so a 0 will eventually move into the bit position.
Method 2: Delay instructions ready to enter WB.

Each functional unit provides a signal …
… indicating when it has an instruction ready to enter WB.

One of those signals is chosen (using some method) …
… the corresponding instruction moves to WB …
… while the others are stalled.
Comparison of Method 1 and 2

Method 1 is easier to implement ... 
... since logic remains in one stage.

In contrast, logic for method 2 would span several stages ... 
... since stages back to IF might need to be stalled ... 
... and so critical paths would be long.

Method 2 is more flexible ... 
... since priority could be given to longer-latency instructions.
Handling RAW Hazards

The interlock mechanism for RAW hazards . . .
. . . must keep track of registers with pending writes . . .
. . . and use this information to stall instructions.

Consider, `add f1, f2, f3`.

Check if any uncompleted preceding instructions write `f2` or `f3`.

If so, stall until register(s) written or can be bypassed to adder.
Possible RAW Interlock Implementations.

Brute Force: Check all following stages

As done for integer operations, check following stages . . .
. . . for pending write to register.

Each stage of every pipelined unit must be checked.

Too expensive.

Register file includes *ready bit* for each register.

Ready bit normally 1, indicating no pending writes (so value valid).

When instruction issued, bit set to 0 . . .
. . . when instruction completes and result written, set back to 1.

Instruction stalls if either operand’s ready bit is 0 . . .
. . . *and* cannot be bypassed.
WAW Hazards

Example with 3-stage pipelined multiply and one-stage add, no MEM.

\[
\begin{align*}
\text{mulf } & f0, f1, f2 & \text{IF } & \text{ID } & \text{M0 } & \text{M1 } & \text{M2 } & \text{WB} \\
\text{addf } & f0, f3, f4 & \text{IF } & \text{ID } & \text{AO } & \text{WB} & \text{! Incorrect execution!!} \\
\end{align*}
\]

Handling WAW Hazards

The interlock mechanism for RAW hazards handles WAW hazards in which there is an intervening read.

Example with 3-stage pipelined multiply and one-stage add, no MEM.

\[
\begin{align*}
\text{mulf } & f0, f1, f2 & \text{IF } & \text{ID } & \text{M0 } & \text{M1 } & \text{M2 } & \text{WB} \\
\text{subf } & f5, f0, f6 & \text{IF } & \text{ID } & \text{--------} & \text{AO } & \text{WB} \\
\text{addf } & f0, f3, f4 & \text{IF } & \text{--------} & \text{ID } & \text{AO } & \text{WB } & \text{! No problem.} \\
\end{align*}
\]

If there is no intervening write the earlier instruction is squashed.

\[
\begin{align*}
\text{mulf } & f0, f1, f2 & \text{IF } & \text{ID } & \text{M0x} \\
\text{addf } & f0, f3, f4 & \text{IF } & \text{ID } & \text{AO } & \text{WB} \\
\end{align*}
\]
WAR Hazards

Possible when register read delayed.

Can’t happen in Chapter-3 DLX because instructions

(1) read registers in ID

(2) pass through ID in program order

(3) and produce results only after leaving ID.

Consider:

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>multf</td>
<td>f0, f1, f2</td>
<td>IF</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>WB</td>
</tr>
<tr>
<td>addf</td>
<td>f1, f3, f4</td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There would be a WAR hazard if addf wrote \textit{f1} before multf read it.

That can’t happen since multf would leave ID (with \textit{f1}) as addf just enters ID.
CPI and Multicycle Operations

CPI more sensitive to dependencies between instructions.

CPI Loop Example

Consider:

```
LOOP:
  ld  f0, 0(r1)
  addi r1, r1, #8
  gtd  f0, f2
  bftp LOOP
  addi r2, r2, #1
  j LOOP
  xor  r3, r4, r5
```

Note dependency between `gtd` and `bftp`.

What is the CPI during the execution of this loop?
When branch not taken:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld f0, 0(r1)</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>addi r1, r1, #8</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>gtd f0, f2</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td>bfpt LOOP</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>----------&gt;</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>addi r2, r2, #1</td>
<td></td>
<td>IF</td>
<td>----------&gt;</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>j LOOP</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>xor r3, r4, r5</td>
<td></td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Second iteration will execute exactly as first.

Therefore, can base iteration time on corresponding points in consecutive iterations.


For a large number of iterations. CPI: \( \frac{11}{6} = 1.8333 \).
When branch taken.

LOOP:
Cycle: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
ld  f0, 0(r1)   IF ID EX MEM WB
addi r1, r1, #8  IF ID EX MEM WB
gtd  f0, f2     IF ID A0 A1 A2 A3 WB
bfpt LOOP       IF ID -------------> EX MEM WB IF ID
addi r2, r2, #1  IF -------------> x IF
j LOOP          IF
xor r3, r4, r5

Note: Second iteration will execute exactly as first.

Iteration time: 9 cycles. Instructions: 4.

For a large number of iterations: CPI is \( \frac{9}{4} = 2.25 \).
Precise Exceptions

Problem is registers written out of order ...
... so some registers must be *unwritten* ...
... so that when handler starts ...
... it must seem as though ...
... all instructions before faulting instructions executed ...
... while no instructions after faulting instruction execute.

\[
\begin{align*}
multf & f0, f1, f2 & IF & ID & M0 & M1 & M2 & M3 & M4 & M5 & *M6* & WB \\
addf & f1, f3, f4 & IF & ID & A0 & A1 & A2 & A3 & WB
\end{align*}
\]

To do this either ...  
... add lots of stalls so instructions do finish in order ...  
... limit those instructions that can raise precise exceptions ...  
... or need to *unexecute* instructions.

The first option is fine for debugging, too slow otherwise.

The second option requires lots of hardware.
Method 1: Stall so that instructions complete in order.

```
multf f0, f1, f2 IF ID M0 M1 M2 M3 M4 M5 M6 WB
addf f1, f3, f4 IF ID ----------> A0 A1 A2 A3 WB
```

This works, (WB in program order) but reduces performance.
Method 2: Early Detection of Exceptions

FP unit raises exceptions early in computation . . .
. . . if computation passes that point, it will finish without exceptions.

For example, 26-cycle DIV unit may check operands by cycle 3 . . .
. . . if computation reaches cycle 4 there is no possibility of an exception.

Instructions only stall until preceding instruction checked for exceptions.

For example, suppose the FP multiply unit finds exceptions by end of M5.

Then at cycle 8 (below) addf can write (no chance of an exception in M6).

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>multf f0,f1,f2</td>
<td>IF</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>WB</td>
</tr>
<tr>
<td>addf f1,f3,f4</td>
<td>IF</td>
<td>ID</td>
<td>-&gt;</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Method 3: Have precise and non-precise FP operations.

Let the names of imprecise instructions end in \textit{ip}.

Second \textit{addf} doesn’t stall since an exception in \textit{multfip} need not be precise.

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{multf}</td>
<td>f0,f1,f2</td>
<td>IF</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{addf}</td>
<td>f1,f3,f4</td>
<td>IF</td>
<td>ID</td>
<td>--------&gt;</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{multfip}</td>
<td>f5,f6,f7</td>
<td>IF</td>
<td>--------&gt;</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\textit{addf}</td>
<td>f6,f8,f9</td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Method 4: FP instructions precise when followed by special test instruction.

Call the special instruction `testexc`.

No stalls (and imprecise exceptions) where `testexc` not used.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</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>multif IF ID M0 M1 M2 M3 M4 M5 M6 WB</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>testexc IF ID -----------------&gt; EX MEM WB</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addf IF -----------------&gt; ID A0 A1 A2 A3 WB</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multif IF ID M0 M1 M2 M3 M4 M5 M6 WB</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addf IF ID A0 A1 A2 A3 WB</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unexecuting Instructions

An instruction is unexecuted . . .

. . . by restoring the previous contents of any register it wrote.

Method 1: *History File*

History file holds replaced values.

These are used to undo writes.
Method 2: Writes to register file are buffered.

Register writes (register number and new value) …
… are first placed in a buffer …
… possibly out of program order.

Writes from buffer to register file performed in order …
… waiting for long-latency operations to complete.

Register reads check the buffer first, then the register file.

When an exception occurs …
… only writes preceding the faulting instruction …
… are made from the buffer to the register file.

Disadvantage: Checking both buffer and register file is time-consuming.
Method 3: *Future File*

Two register files maintained, *main* and *future*.

Future file written as instruction complete . . .

. . . main file written in program order.

Future file is used for reading registers.

At an exception, . . .

. . . main file updated up to faulting instruction . . .

. . . future file is effectively erased . . .

. . . its contents replaced by main register file before handler starts.
Stalls per FP operation on SPEC 92 FP benchmarks.

Running SPEC 92 benchmarks on DLX compiled using old version of gcc.

Uses **perfect** cache.

Value indicates stall cycles per instruction type.

*E.g.*, running doduc, there are an average of 1.7 stall cycles due to each compare.

Stall cycles are due to RAW hazards except for *divide structural* bars.

**FIGURE 3.48** Stalls per FP operation for each major type of FP operation.
Number of stalls determined by:

- latency of functional unit,
- characteristics of program, and
- quality of compiler.

Example:

```
Cycle: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
multf f0,f1,f2 IF ID M0 M1 M2 M3 M4 M5 M6 WB
addf f3,f0,f4 IF ID -------------------------> A0 A1 A2 A3 WB
```

Here, six stall cycles “charged” to `multf`.

Lower latency (better functional unit) would mean fewer stall cycles.
Example, better scheduling:

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>multf</td>
<td>f0,f1,f2</td>
<td>IF</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gtf</td>
<td>f5,f6,f7</td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subd</td>
<td>f8,f10,f12</td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addf</td>
<td>f3,f0,f4</td>
<td>IF</td>
<td>ID</td>
<td>---------------&gt;</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here \texttt{multf} charged with only four cycles because of \texttt{gtf} and \texttt{subd}.

The existence of such instructions depends on program characteristics.

Discovery and scheduling (arrangement) of such instructions depends on compiler.
Running SPEC 92 benchmarks on DLX compiled using old version of gcc.

Uses **perfect** cache.

Value indicates stalls per instruction by cause.

Stalls caused primarily by RAW hazards.

**FIGURE 3.49** The stalls occurring for the DLX FP pipeline for the five FP SPEC benchmarks.
Running SPEC 92 benchmarks on R4000.

In R4000:

- Load latency is two cycles.
- Uses **perfect** cache.
- Branch delay two cycles.
- FP functional units partially pipelined.

---

**Performance of MIPS R4000.**

**Pipeline CPI**

- **Base**
- **Load stalls**
- **Branch stalls**
- **FP result stalls**
- **FP structural stalls**

**SPEC92 benchmark**
Dependencies

Dependency:
A relationship between two instructions ... 
... indicating that their execution should be in program order.

If there is a dependency between instruction $A$ and instruction $B$ ... 
... and $B$ follows $A$ in program order ... 
... then $B$ is said to be dependent on $A$.

If $B$ is dependent on $A$ then $A$ should normally execute before $B$.

Dependency Types:

- *True, Data, or Flow Dependence* (Three different terms used.)
- *Name Dependence*
- *Control Dependence*
Data Dependence

Data Dependence: (a.k.a., True and Flow Dependence)
A dependence between two instructions ... 
... indicating data needed by the second is produced by the first.

Example:

```
add  r1, r2, r3
sub  r4, r1, r5
and  r6, r4, r7
```

The `sub` is dependent on `add` (via `r1`).

The `and` is dependent on `sub` (via `r4`).

The `and` is dependent `add` (via `sub`).

Execution may be incorrect if ... 
... a program having a data dependence ... 
... is run on a processor having an uncorrected RAW hazard.
Name Dependencies

There are two kinds: antidependence and output dependence.

**Antidependence:**
A dependence between two instructions ...
... indicating a value written by the second ...
... that the first instruction reads.

Antidependence Example

```
add  r1, r2, r3
sub  r2, r4, r5
```

*sub* is antidependent on the *add*.

Execution may be incorrect if ...
... a program having an antidependence ...
... is run on a processor having an uncorrected WAR hazard.
Output Dependence:
A dependence between two instructions …
 … indicating that both instructions write the same location …
 … (register or memory address).

Output Dependence Example

```
add  r1, r2, r3
sub  r1, r4, r5
```

The sub is output dependent on add.

Execution may be incorrect if …
 … a program having an output dependence …
 … is run on a processor having an uncorrected WAW hazard.
**Control Dependence:**
A dependence between a branch instruction and a second instruction ... 
... indicating that whether the second instruction executes ... 
... depends on the outcome of the branch.

```
beqz r1, SKIP
add  r2, r3, r4
sub  r5, r6, r7
```

The **add** is control dependent on the **beqz**.

The **sub** is not control dependent on the **beqz**.
Scheduling

Motivation

Stalls are bad.

Some stalls can be avoided by rearranging instructions.

Others can be avoided by restructuring code.

Avoiding stalls this way is free if done by compiler or programmer.

*Schedule*: Organizing instructions to improve execution efficiency.

*Static Scheduling*: Organizing of instructions by compiler or programmer to improve execution efficiency.
Scheduling Examples

Unscheduled Code

```
addf f0, f1, f2
subf f3, f0, f4
multf f5, f6, f7
ld f8, 0(r1)
addi r1, r1, #8
subi r2, r2, #1
```
Unscheduled Code on Chapter-3 DLX

Note: Shared integer and FP WB.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>addf</td>
<td>f0, f1, f2</td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subf</td>
<td>f3, f0, f4</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
</tr>
<tr>
<td>multf</td>
<td>f5, f6, f7</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
</tr>
<tr>
<td>ld</td>
<td>f8, 0(r1)</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>r1, r1, #8</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subi</td>
<td>r2, r2, #1</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution has four stall cycles.

Schedule code by moving integer instructions between addf and subf.
Scheduled Code on Chapter-3 DLX

Instructions reordered by compiler or programmer to remove stalls.

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>addf f0, f1, f2</td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f8, 0(r1)</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulf f5, f6, f4</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>M0</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>addi r1, r1, #8</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subf f3, f0, f4</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>A0</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subi r2, r2, #1</td>
<td></td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution has zero stall cycles.
Loop Unrolling:
A code restructuring technique for loops in which . . .
. . . the computations performed by several iterations of the original loop . . .
. . . are performed by one iteration of the unrolled loop.

The unrolled loop performs the same amount of work . . .
. . . but uses fewer instructions and induces fewer stalls.

Loop is said to be unrolled twice . . .
. . . if two iterations of original loop performed by one of unrolled loop.

Loop is said to be unrolled n times . . .
. . . if n iterations of original loop performed by one of unrolled loop.

A loop unrolled once is the same as the original loop.
Loop Unrolling Example

Suppose loop below runs for 24 iterations.

Execution on DLX:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</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>lw</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bneq</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution on DLX. \( \frac{7}{5} = 1.5 \text{ CPI} \ldots 

\ldots \text{ execution time } 24 \times 7 = 168 \text{ cycles.}
Unrolled twice:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>

**LOOP:**

- `lw r1, 0(r2)` IF ID EX MEM WB IF ID EX
- `lw r10, 4(r2)` IF ID EX MEM WB IF ID
- `addi r2, r2, #8` IF ID EX MEM WB IF
- `add r3, r3, r1` IF ID EX MEM WB
- `add r3, r3, r10` IF ID EX MEM WB
- `sub r5, r4, r2` IF ID EX MEM WB
- `bneq r5, LOOP` IF ID EX MEM WB
- `and` IFx

Instruction execution time: $\frac{8}{7} = 1.14$ CPI...

... execution time $12 \times 8 = 96$ cycles.

Double benefit: ...

... faster execution per instruction and ...

... fewer instructions.
Scheduled:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>

**LOOP:**

- `lw r1, 0(r2)` IF ID EX MEM WB IF
- `addi r2, r2, #4` IF ID EX MEM WB
- `add r3, r3, r1` IF ID EX MEM WB
- `sub r5, r4, r2` IF ID EX MEM WB
- `bneq r5, LOOP` IF ID EX MEM WB
- `and r5, LOOP` IFx

Scheduled: \( \frac{6}{5} = 1.2 \text{ CPI} \ldots \)

\( \ldots \) execution time \( 24 \times 6 = 144 \text{ cycles} \).

Not as good as unrolled loop, 96 cycles.
How Unrolling Improves Performance

Suppose original loop had 24 iterations and unrolled twice.

Unrolled loop runs for 12 iterations.

Twelve, instead of 24 end-of-loop branches . . .
. . . eliminates 12 branch condition test instructions, 12 branch instructions, . . .
. . . and 12 bubbles inserted after branch.

If indexed addressing allowed (*e.g.*, \texttt{lw \ r10, 4(r2)}) . . .
. . . eliminates 12 address increment instructions.

With more instructions per iterations, its easier to eliminate RAW hazard stalls by scheduling.
Instruction-Level Parallelism:
The average number of instructions in a machine-language program . . .
. . . that can be simultaneously started [per cycle] . . .
. . . when execution is only limited by true dependencies.

Note: Text definition is less specific.

Number of instructions started per cycle, IPC, is $\frac{1}{CPI}$.

Provides a bound on performance of an implementation of an ISA.
ILP for SPEC92 programs in MIPS:

Based on graph it’s possible to attain a CPI of \( \frac{1}{54.8} \) for gcc ... ... which is much better than 1 for Chapter-3 DLX.

These IPC’s are much higher than believed attainable.
Example 1: No control-transfer instructions, no name dependencies.

```
lw   r1, 0(r2)
sub  r4, r1, r5
and  r6, r1, r7
xor  r8, r4, r6
slt  r9, r8, r10
or   r11, r8, r12
addi r13, r8, #1
```
Execution on DLX:

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>
</tr>
</thead>
<tbody>
<tr>
<td>lw r1, 0(r2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF</td>
<td>ID</td>
<td>--&gt;</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and r6, r1, r7</td>
<td>IF</td>
<td>--&gt;</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor r8, r4, r6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>slt r9, r8, r10</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or r11, r8, r12</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi r13, r8, #1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On DLX, execution speed $\frac{7}{8} = 0.875$ inst/cycle.
Execution on ideal machine used for determining ILP:

To find ILP use 0-cycle latencies and true dependencies:

```
Cycle: 0 1 2 3 4 5 6 7 8 9 10 11
lw r1, 0(r2) St
sub r4, r1, r5 St
and r6, r1, r7 St
xor r8, r4, r6 St
slt r9, r8, r10 St
or r11, r8, r12 St
addi r13, r8, #1 St
```

ILP is $\frac{7}{4} = 2.75$ inst/cycle, much better.

Note: No stall after load.

Simultaneous execution of instructions at cycle 1 and 3.
Example 1a: No control-transfer instructions, name dependencies.

```
lw  r1, 0(r2)
add  r2, r1, r3  ! Data dependence between lw and add
xor  r1, r4, r5  ! Anti dependence between add and xor.
add  r6, r1, r6  ! Data dependence between xor and add.
```

Execution on the Chapter-3 DLX implementation.

```
!Cycle:  0 1 2 3 4 5 6 7 8 9 10 11
lw  r1, 0(r2)  IF  ID  EX  MEM  WB
add  r2, r1, r3  IF  ID  -->  EX  MEM  WB
xor  r1, r4, r5  IF  -->  ID  EX  MEM  WB
add  r6, r1, r6  IF  ID  EX  MEM  WB
```

ILP Analysis.

```
!Cycle:    0  1
lw  r1, 0(r2)  St
add  r2, r1, r3  St  ! Wait due to data dependency.
xor  r1, r4, r5  St  ! No need to wait for name dependency.
add  r6, r1, r6  St
```
Example 2: No control-transfer instructions, load/stores.

On Chapter-3 DLX:

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</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>add</td>
<td>r1, r2, r3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>0(r10), r1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>r4, 0(r11) ! r10 = r11</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>r5, 0(r12) ! r10 != r12</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>r6, r4, r7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r8, r5, r9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ILP Analysis:

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>r1, r2, r3</td>
<td>St</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>0(r10), r1</td>
<td>St</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>r4, 0(r11) ! r10 = r11</td>
<td>St</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td>r5, 0(r12) ! r10 != r12</td>
<td>St</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>r6, r4, r7</td>
<td>St</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r8, r5, r9</td>
<td>St</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ILP is \(\frac{6}{4} = 1.5\).

To achieve this hardware must determine effective-address relationships.
ILP and Basic Blocks

**Basic Block:**
Consecutive instructions that are always executed consecutively.

Equivalently: consecutive instructions in which ...
... only the first may be a branch target ...
... and only the last be a control transfer.

All members of a basic block get executed the same number of times.

L1:
\[\text{add } r1, r2, r3 \quad \text{! Basic block 1.}\]

L0:
\[\text{sub } r2, r3, r4 \quad \text{! Basic block 2.} \]
\[\text{and } r5, r6, r7 \quad \text{! Basic block 2.} \]
\[\text{bneq } r5, \text{TARGET} \quad \text{! Basic block 2.} \]
\[\text{xor } r6, r7, r8 \quad \text{! Basic block 3.} \]

TARGET:
\[\text{or } r9, r10, r11 \quad \text{! Basic block 4.} \]

L2:

Code contains four basic blocks.
ILP and Basic Blocks

Much ILP comes from *ignoring* control dependencies . . .
. . . that is, simultaneously executing instructions in different basic blocks.

Example 3: Control Transfers

On Chapter-3 DLX:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bneq r4, SKIP</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r5, r6, r7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKIP:</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>or r8, r9, r10</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bneq r8 SKIP2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi r1, r1, #5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SKIP2:</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>xor r11, r11, r12</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ILP Analysis:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>add r1, r2, r3</td>
<td>St</td>
</tr>
<tr>
<td>1</td>
<td>bneq r4, SKIP</td>
<td>St</td>
</tr>
<tr>
<td></td>
<td>add r5, r6, r7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SKIP:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or r8, r9, r10</td>
<td>St</td>
</tr>
<tr>
<td></td>
<td>bneq r8 SKIP2</td>
<td>St</td>
</tr>
<tr>
<td></td>
<td>addi r1, r1, #5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SKIP2:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xor r11, r11, r12</td>
<td>St</td>
</tr>
</tbody>
</table>

ILP: $\frac{5}{2}$.

Instruction overlap determined by operands only, not branches.