MIPS Implementation

Material from Chapter 3 of H&P (for DLX).
Material from Chapter 6 of P&H (for MIPS).

Outline: (In this set.)
Unpipelined DLX Implementation. (Diagram only.)
Pipelined DLX and MIPS Implementations: Hardware, notation, hazards.
Dependency Definitions.
Data Hazards: Definitions, stalling, bypassing.
Control Hazards: Squashing, one-cycle implementation.

Outline: (Covered in class but not yet in set.)
Operation of nonpipelined implementation, elegance and power of pipelined implementation. (See text.)
Computation of CPI for program executing a loop.

Unpipelined Implementation

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

Note: diagram omits connections for some instructions.

Pipelined DLX Implementation

Pipelined MIPS Implementation

Note: diagram omits connections for some instructions.
**Pipeline Details**

**Pipeline Segments** a.k.a. **Pipeline Stages**

Divide pipeline into *segments*.

Each segment occupied by at most one instruction.

At any time, different segments can be occupied by different instructions.

Segments given names: IF, ID, EX, MEM, WB

Sometimes MEM shortened to ME.

---

**Pipeline Registers** a.k.a. **Pipeline Latches**

Registers separating pipeline segments.

Written at end of each cycle.

To emphasize role, drawn as part of dividing bars.

Registers named using pair of segment names and register name.

For example, IF/ID.IR, ID/EX.IR, ID/EX.A (used in text, notes).

if_id_ir, id_ex_ir, id_ex_rs_val (used in Verilog code).

---

**Pipeline Execution Diagram**

Diagram showing the pipeline segments that instructions occupy as they execute.

Time on horizontal axis, instructions on vertical axis.

Diagram shows where instruction is at a particular time.

<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>
</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>
</tr>
<tr>
<td>and r4, r5, r6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r7, 8(r9)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A vertical slice (*e.g.*, at cycle 3) shows processor activity at that time.

In such a slice a segment should appear at most once . . .

. . . if it appears more than once execution not correct . . .

. . . since a segment can only execute one instruction at a time.

---

**Instruction Decoding and Pipeline Control**

**Pipeline Control**

Setting control inputs to devices including . . .

. . . multiplexor inputs . . .

. . . function for ALU . . .

. . . operation for memory . . .

. . . whether to clock each register . . .

. . . et cetera.
Options for controlling pipeline:

- Decode in ID
  Determine settings in ID, pass settings along in pipeline latches.

- Decode in Each Stage
  Pass opcode portions of instruction along.
  Decoding performed as needed.

Real systems decode in ID.

For clarity, diagrams misleadingly imply decoding in stage needed ... ... by passing entire instruction along.
Example given later in this set.

Remember
Operands read from registers in ID ...
... and results written to registers in WB.

Consider the following incorrect execution:

<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>
<tbody>
<tr>
<td>add</td>
<td>r1, r2, r3 IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>r4, r1, r5 IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>r6, r1, r8 IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>r9, r4, r11 IF ID EX MEM WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution incorrect because ...
... sub reads r1 before add writes (or even finishes computing) r1, ...
... and reads r1 before add writes r1, and ...
... xor reads r4 before sub writes r4.

Incorrect execution due to ...
... dependencies in program ...
... and hazards in hardware (pipeline).
Incorrect execution above is the “fault” of the hardware ...
... because the ISA does not forbid dependencies.

Dependency:
A relationship between two instructions ...
... indicating that their execution should be (or appear to be) in program order.

Hazard:
A potential execution problem in an implementation due to overlapping instruction execution.

There are several kinds of dependencies and hazards.
For each kind of dependence there is a corresponding kind of hazard.
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.

Control Dependence Example
```
beqz r1, SKIP  # Not a delayed branch
add r2, r3, r4
SKIP:
sub r5, r6, r7
```

The `add` is control dependent on the `beqz`.
The `sub` is not control dependent on the `beqz`.
Pipeline Hazards

Hazard:
A potential execution problem in an implementation due to overlapping instruction execution.

Interlock:
Hardware that avoids hazards by stalling certain instructions when necessary.

Hazard Types:

Structural Hazard:
Needed resource currently busy.

Data Hazard:
Needed value not yet available or overwritten.

Control Hazard:
Needed instruction not yet available or wrong instruction executing.

Data Hazards

Identified by acronym indicating correct operation.

- RAW: Read after write, akin to data dependency.
- WAR: Write after read, akin to anti dependency.
- WAW: Write after write, akin to output dependency.

DLX and MIPS implementations above only subject to RAW hazards.
RAR not a hazard since read order irrelevant (without an intervening write).

Interlocks

When threatened by a hazard:

- **Stall** (Pause a part of the pipeline.)
  Stalling avoids overlap that would cause error.
  This does slow things down.
- Add hardware to avoid the hazards.
  Details of hardware depend on hazard and pipeline.
  Several will be covered.

Structural Hazards

Cause: two instructions simultaneously need one resource.

Solutions:

- Stall.
- Duplicate resource.

Pipelines in this section do not have structural hazards.
Covered in more detail with floating-point instructions.
Data Hazards

HP Chapter-3 DLX and MIPS Subject to RAW Hazards.

Consider the following incorrect execution of code containing data dependencies.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0 1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>and r6, r1, r8</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>xor r9, r4, r11</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

Execution incorrect because ...

... sub reads r1 before add writes (or even finishes computing) r1, ...

... and reads r1 before add writes r1, and ...

... xor reads r4 before sub writes r4.

Problem fixed by stalling the pipeline.

Stall: To pause execution in a pipeline from IF up to a certain stage.

With stalls, code can execute correctly:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>and r6, r1, r8</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>xor r9, r4, r11</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

Arrow shows that instructions stalled.

Stall creates a bubble, segments without valid instructions, in the pipeline.

With bubbles present, CPI is greater than its ideal value of 1.

Stall Implementation

Stall implemented by asserting a hold signal ...

... which inserts a nop (or equivalent) after the stalling instruction and ...

... disables clocking of pipeline latches before the stalling instruction.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>and r6, r1, r8</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>xor r9, r4, r11</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

During cycle 3, a nop is in EX.

During cycle 4, a nop is in EX and MEM.

The two adjacent nops are called a bubble ...

... they move through the pipeline with the other instructions.

A third nop is in EX in cycle 7.

Bypassing

Some stalls are avoidable.

Consider again:

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0 1 2 3 4 5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r1, r2, r3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>and r6, r1, r8</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>xor r9, r4, r11</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

Note that the new value of r1 needed by sub ...

... has been computed at the end of cycle 2 ...

... and isn’t really needed until the beginning of the next cycle, 3.

Execution was incorrect because the value had to go around the pipeline to ID.

Why not provide a shortcut?

Why not call a shortcut a bypass or forwarding path?
**Non-Bypassed MIPS**

![Non-Bypassed MIPS Diagram]

**Bypassed MIPS**

![Bypassed MIPS Diagram]

**MIPS Implementation With Some Forwarding Paths:**

![MIPS Implementation Diagram]

Not all stalls are avoidable.

Stall due to `lw` could not be avoided (data not available in cycle 3).

Stall in cycles 5 and 6 could be avoided with a new forwarding path.
Bypass Control Logic for Lower ALU Mux

Start with logic for \( rd \), show path of Mux logic.

Logic to determine \( rd \) for register file.

Control logic not minimized (for clarity).

Control Logic Generating ID/EX.RD.

Present in previous implementations, just not shown.

Determines which register gets written based on instruction.

Instruction categories used in boxes such as

- **Non-link CTI**: branches and jumps except linking jumps (jal and jalr).
- **Store**: All store instructions.
- **Type I ALU**: All Type I ALU instructions.
- **Load**: All load instructions.
- **Type R**: All Type R instructions.
- **Link CTI**: jal and jalr.
Bypass Control Logic, Continued

Logic Generating **ID/EX.MUX.**

- A box determines if two register numbers are equal.
- Register number zero is not equal register zero, nor any other register.
  (The bypassed zero value might not be zero.)

Control Hazards

Cause: on **taken** CTI several wrong instructions fetched.

Consider:

Example of *incorrect execution*

<table>
<thead>
<tr>
<th>1! ADR</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>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>beqz r4, TARGET</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>0x104</td>
<td>sw 0(r2), 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>0x108</td>
<td>sub r4, r2, r5</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>0x100c</td>
<td>and r6, r1, r8</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>0x110</td>
<td>or r12, r13, r14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TARGET: ! TARGET = 0x200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>xor r9, r4, 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>
</tbody>
</table>

Branch is taken yet following three instructions complete execution.

Branch target finally fetched in cycle 4.

Problem: What do we do about three instructions following branch?
### Handling Instructions Following a Taken Branch

**Option 2:** Fetch them, but squash (stop) them in a later stage.

This will work if instructions squashed ... before modifying architecturally visible storage (registers and memory).

Memory modified in MEM stage and registers modified in WB stage ... so instructions must be stopped before beginning of MEM stage.

Can we do it? That depends where the branch instruction is.

In example, we need to squash `sw` by end of cycle 2.

During cycle 2 `beqz` in EX ...

... it has been decoded and the branch condition is available ...

... so we know whether the branch is taken.

If the branch is taken, the `sw` and following instructions can be squashed in time.

Option 2 will be used.

### Instruction Squashing

**In-Flight Instruction:**

An instruction in the execution pipeline.

Later in the semester a more specific definition will be used.

**Squashing:** [an instruction] preventing an in-flight instruction ...

... from writing registers, memory or any other visible storage.

Squashing also called: *nulling, abandoning*, and *cancelling*.

Like an insect, a squashed instruction is still there (in most cases) but can do no harm.

### Squashing Instruction in Example DLX Implementation

Two ways to squash.

- Prevent it from writing architecturally visible storage.

  Replace destination register control bits with zero. (Writing zero doesn’t change anything.)

  Set memory control bits (not shown so far) for no operation.

- Change Operation to `nop`.

  Would require changing many control bits.

  Squashing shown that way here for brevity.

  Illustrated by placing a `nop` in ID.

Why not replace squashed instructions with target instructions?

Because there is no straightforward and inexpensive way ... to get the instructions where and when they are needed.

(Curiously, and expensive techniques covered in Chapter 4.)

### DLX implementation used so far.
Example of correct execution

<table>
<thead>
<tr>
<th>I Addr</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>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>beq  r4, TARGET</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>0x104</td>
<td>sw  0(r2), r1</td>
<td>IF</td>
<td>ID</td>
<td>EXx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sub r4, r2, r5</td>
<td>IF</td>
<td>IDx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>or  r12, r13, r14</td>
<td>IFX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>TARGET: ! TARGET = 0x200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>xor r9, r4, 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>
</tbody>
</table>

Branch outcome known at end of cycle 2 ...
... wait for cycle 3 when all doomed instructions (sw, sub, and) in flight ...
... and squash them so in cycle 4 they act like nops.

Three cycles (1, 2, and 3), are lost.
Three cycles called a branch penalty.
Three cycles is alot of cycles, is there something we can do?

One-Cycle Branch Delay Implementation

Compute branch target and condition in ID stage.

Now how fast will code run?

<table>
<thead>
<tr>
<th>I Addr</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>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>beq  r4, TARGET</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>0x104</td>
<td>sw  0(r2), r1</td>
<td>IF</td>
<td>ID</td>
<td>EXx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sub r4, r2, r5</td>
<td>IF</td>
<td>IDx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>or  r12, r13, r14</td>
<td>IFX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>TARGET: ! TARGET = 0x200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x200</td>
<td>xor r9, r4, 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>
</tbody>
</table>

Penalty is only one cycle.
Later in semester penalty reduced to zero and even -1 [sic].

Non-Bypassed MIPS