Material from chapter 3 of H&P.

Outline: (In this set.)

Unpipelined DLX Implementation. (Diagram only.)

Pipelined Implementation: Hardware, notation, hazards.

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

IF

ID

EX

MEM

WB

PC

IR

NPC

+4

Addr

Mem

Port

Data

Addr

Data

Addr

D In

6..10

11..15

16..20

or

11..15

sign ext.

NPC

=0

A

B

IMM

Data

Addr

Data

Addr

In

Out

ALU

Mem

Addr

MD

IR

ALU

IR

IR

IR
Pipeline Details

Pipeline Segments

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**

Pipeline Registers

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**.
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.
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:

- Magic Cloud
- 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.
Pipeline Hazards

Hazard:
A potential execution problem due overlapping instruction execution.

Interlock:
Hardware that avoids hazards by stalling certain instructions.

Hazard Types:

*Structural Hazard:*  
Needed resource currently busy.

*Data Hazard:*  
Needed value (written by previous instruction) not yet available.

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

Identified by acronym indicating correct operation.

- **RAW**: Read after write.
- **WAR**: Write after read.
- **WAW**: Write after write.

DLX implementation 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.

Covered in more detail with floating-point instructions.
Data Hazards

Chapter-3 DLX Subject to RAW Hazards.

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 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>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>
</tr>
<tr>
<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>
</tr>
</tbody>
</table>

Execution incorrect because ...

... *sub* reads *r1* before *add* computed its value, ...

... *and* reads *r1* before *add* stored its value, and ...

... *xor* reads *r4* before *sub* stored its value.

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:

For code on previous slide, stall until data in register.

<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>
<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>
</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>
</tr>
<tr>
<td>and r6, r1, r8</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>
</tr>
<tr>
<td>xor r9, r4, r11</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>
</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 \textit{hold} signal ...
... which inserts a \texttt{nop} after the stalling instruction and ...
... disables clocking of pipeline latches before the stalling instruction.

\begin{tabular}{lccccc}
! Cycle & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
add r1, r2, r3 & IF & ID & EX & MEM & WB \\
sub r4, r1, r5 & IF & ID & -----& & EX & MEM & WB \\
and r6, r1, r8 & IF & -----& & ID & EX & MEM & WB \\
xor r9, r4, r11 & IF & ID & \rightarrow & EX & MEM & WB \\
\end{tabular}

During cycle 3, a \texttt{nop} is in EX.

During cycle 4, a \texttt{nop} is in EX and MEM.

The two adjacent \texttt{nops} are called a \textit{bubble} ...
... they move through the pipeline with the other instructions.

A third \texttt{nop} is in EX in cycle 7.
Some stalls are avoidable.

Consider again:

<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>
<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>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>r4, r1, 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>
<td></td>
</tr>
<tr>
<td>and</td>
<td>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>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>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>
<td></td>
</tr>
</tbody>
</table>

Note that the new value of $r1$ needed by $\text{sub}$ …

… 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 \textit{bypass} or \textit{forwarding} path?
DLX Implementation With Some Forwarding Paths:

! Cycle! 0! 1! 2! 3! 4! 5! 6! 7! 8! 9! 10
add r1, r2, r3! IF! ID! EX! MEM! WB
sub r4, r1, r5! IF! ID! EX! MEM! WB
and r6, r1, r8! IF! ID! EX! MEM! WB
xor r9, r4, r11! IF! ID! EX! MEM! WB

It works!
DLX Implementation With Some Forwarding Paths:

Not all stalls are avoidable.

<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>
<tbody>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lw</td>
<td></td>
</tr>
<tr>
<td>add</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>sw</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>addi</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>

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 \textit{rd}, show path of Mux logic.
Logic to determine \textit{rd} for register file.
Bypass Control Logic for Lower ALU Mux

IF

ID

EX

MEM

WB

NPC

+4

PC

Addr

Mem Port

Data

IR

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr

Data

Addr
Bypass Control Logic

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 $= \text{Load}$ (some instructions omitted):

- $= \text{Non-link CTI}$: branches and jumps except linking jumps ($\text{jal}$ and $\text{jalr}$).
- $= \text{Store}$: All store instructions.
- $= \text{Type I ALU}$: All Type I ALU instructions.
- $= \text{Load}$: All load instructions.
- $= \text{Type R}$: All Type R instructions.
- $= \text{Link CTI}$: $\text{jal}$ and $\text{jalr}$.
Logic Generating **ID/EX.MUX.**

`=` 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>!I 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</td>
<td>r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sw</td>
<td>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>
</tr>
<tr>
<td>0x108</td>
<td>sub</td>
<td>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>
</tr>
<tr>
<td>0x10c</td>
<td>and</td>
<td>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>
</tr>
<tr>
<td>0x110</td>
<td>or</td>
<td>r12, r13, r14</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...  

**TARGET:** ! TARGET = 0x200  

| 0x200  | xor   | r9, r4, r11 | IF | ID | EX | MEM | WB |

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 1: Don’t fetch them.

Impossible (with pipelining) because ... fetch starts (sw in cycle 1) before branch even decoded.

<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>beqz</td>
<td>r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sw</td>
<td>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>
</tr>
<tr>
<td>0x108</td>
<td>sub</td>
<td>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>
</tr>
<tr>
<td>0x10c</td>
<td>and</td>
<td>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>
</tr>
<tr>
<td>0x110</td>
<td>or</td>
<td>r12, r13, r14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<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>TARGET:</td>
<td>! 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</td>
<td>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>
</tr>
</tbody>
</table>
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.
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 IR.

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.

(Curvy sideways and expensive techniques covered in Chapter 4.)
DLX implementation used so far.
Example of correct execution

<table>
<thead>
<tr>
<th>I 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</td>
<td>r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sw</td>
<td>0(r2), r1</td>
<td>IF</td>
<td>ID</td>
<td>EXx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sub</td>
<td>r4, r2, r5</td>
<td>IF</td>
<td>IDx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>and</td>
<td>r6, r1, r8</td>
<td>IF</td>
<td>FX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>or</td>
<td>r12, r13, r14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... TARGET: ! TARGET = 0x200

<table>
<thead>
<tr>
<th>I 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>0x200</td>
<td>xor</td>
<td>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>
</tr>
</tbody>
</table>

Branch outcome known at end of cycle 2...

... wait for cycle 3 when all doomed instructions (\texttt{sw}, \texttt{sub}, \texttt{and}) in flight...

... and squash them so in cycle 4 they act like \texttt{nops}.

Three cycles (1, 2, and 3), are lost.

Three cycles called a \textit{branch penalty}.

Three cycles is alot of cycles, is there something we can do?
Compute branch target address in ID stage.
One-Cycle Branch Delay Implementation

Compute branch target and condition in ID stage.

Workable because register values not needed to compute branch address and ... 
... branch condition is a simple zero test.

Now how fast will code run?

<table>
<thead>
<tr>
<th>I 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</td>
<td>r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sw</td>
<td>0(r2), r1</td>
<td>IFx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sub</td>
<td>r4, r2, r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>and</td>
<td>r6, r1, r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>or</td>
<td>r12, r13, r14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 
TARGET: ! TARGET = 0x200

<table>
<thead>
<tr>
<th>I 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>0x200</td>
<td>xor</td>
<td>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>
</tr>
</tbody>
</table>

Penalty is only one cycle.

Later in semester penalty reduced to zero and even -1 [sic].