MIPS Pipelined Implementation

Outline: (In this set.)

Unpipelined Implementation. (Diagram only.)

Pipelined 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.
Very Simple MIPS Implementation

From EE 3755.

Features

Avoid duplication of hardware: One Memory Port, One Adder (ALU).

Relatively complex control logic needed to re-use ALU, etc.
In this implementation hardware is duplicated.

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

Note: diagram omits connections for some instructions.
Pipeline Details

Pipelining Idea

Split hardware into \( n \) equally sized (in time) stages ...
... separate the stages using special registers called pipeline latches ...
... increase the clock frequency by \( \leq n \times \) ...
... avoid problems due to overlapping of execution.

Pipeline Stages

Pipeline divided into stages.

Each stage occupied by at most one instruction.

At any time, each stage can be occupied by its own instruction.

Stages given names: IF, ID, EX, ME, WB

Sometimes ME written as MEM.
**Pipeline Latches:**
Registers separating pipeline stages.

Written at end of each cycle.

To emphasize role shown in diagram as bar separating stages.

Registers named using pair of stage 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 stages 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>ME</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>ME</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>ME</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 stage should appear at most once . . .

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

. . . since a stage 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:

- **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.

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></td>
<td>r1, r2, r3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td></td>
<td>r4, r1, r5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td></td>
<td>r6, r1, r8</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td>r9, r4, r11</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</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.
Dependencies

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

If \( B \) is dependent on \( A \) then \( B \) should appear to execute after \( A \).

Dependency Types:

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

\[
\begin{align*}
\text{add} & \quad r1, r2, r3 \\
\text{sub} & \quad r2, r4, r5
\end{align*}
\]

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.

```assembly
beq $1, $0 SKIP  # Delayed branch
nop
add $2, $3, $4
SKIP:
sub $5, $6, $7
```

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

The **sub** is not control dependent on the **beq**.
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.

MIPS implementations above only subject to RAW hazards.

RAR not a hazard since read order irrelevant (without an intervening write).
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.
Pipelined MIPS Subject to RAW Hazards.

Consider the following incorrect execution of code containing data dependencies.

<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>ME</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>ME</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>ME</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>ME</td>
<td>WB</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.

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>ME</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>ME</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>ME</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>ME</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*, stages 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 ME WB</td>
</tr>
<tr>
<td>sub r4, r1, r5</td>
<td>IF ID -----&gt; EX ME WB</td>
</tr>
<tr>
<td>and r6, r1, r8</td>
<td>IF -----&gt; ID EX ME WB</td>
</tr>
<tr>
<td>xor r9, r4, r11</td>
<td>IF ID -&gt; EX ME WB</td>
</tr>
</tbody>
</table>

During cycle 3, a *nop* is in EX.

During cycle 4, a *nop* is in EX and ME .

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</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>ME</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>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></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>ME</td>
<td>WB</td>
<td></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>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

IF

ID

NPC

Addr

Data

25:21

20:16

Addr

Data

Addr

D In

rsv

rtv

IMM

NPC

MD

dst

dst

dst

Mem Port

Addr

D In

D Out

ALU

rtv

E

dst

dest. reg

format immed

Decode
Bypassed MIPS
MIPS Implementation With Some Forwarding Paths

It works!
Some stalls unavoidable.

Some stalls unavoidable.

<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 r1, 0(r2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1, r1, r4</td>
<td>IF</td>
<td>ID -&gt;</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw 4(r2), r1</td>
<td>IF -&gt;</td>
<td>ID</td>
<td>-------&gt;</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi r2, r2, 8</td>
<td>IF</td>
<td>-------&gt;</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</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 dst, show path of Mux logic.
Logic to determine \texttt{dst} for register file.

Design me next!
Bypass Control Logic for Lower ALU Mux

ID IR signals in purple.
Bypass Control Logic

Control logic not minimized (for clarity).

Control Logic Generating $\text{dst}$.

Present in previous implementations, just not shown.

Determines which register gets written based on instruction.

Instruction categories used in boxes such as $\text{= is Store}$ (some instructions omitted):

- $\text{= is Type R}$: All Type R instructions.
- $\text{= is Store}$: All store instructions.
- $\text{= is Branch}$: branches such as $\text{beq}$ and $\text{bltz}$.
- $\text{= is JR}$, $\text{= is J}$, etc.: Matches the exact instruction.
Bypass Control Logic, Continued

Logic Generating ID/EX_MUX.

The 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</th>
<th>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>I</td>
<td>Adr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>bgtz r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sub r4, r2, r5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sw 0(r2), r1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>and r6, r1, r8</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x110</td>
<td>or r12, r13, r14</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...  
TARGET: ! TARGET = 0x200  
0x200 xor r9, r4, r11 | IF | ID | EX | ME | WB |

Branch is taken yet two instructions past delay slot (**sub**) complete execution.

Branch target finally fetched in cycle 4.

Problem: Two instructions following delay slot.
Handling Instructions Following a Taken Branch Delay Slot

Option 1: Don’t fetch them.

Possible (with pipelining) because ...
... fetch starts (*sw* in cycle 2) ...
... after branch decoded.

(Would be impossible ...
... for non-delayed branch.)

<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>bgtz</td>
<td>r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sub</td>
<td>r4, r2, r5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sw</td>
<td>0(r2), r1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</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>ME</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>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TARGET:</td>
<td>! TARGET = 0x200</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>ME</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 ME stage and registers modified in WB stage . . .
 . . . so instructions must be stopped before beginning of ME stage.

Can we do it? Depends where branch instruction is.

In example, need to squash sw before cycle 5.

During cycle 3 bgtz in ME . . .
 . . . it has been decoded and the branch condition is available . . .
 . . . so we know whether the branch is taken . . .
 . . . so sw can easily be squashed before cycle 5.

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.

(Curvysideways and expensive techniques covered in Chapter 4.)
MIPS 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>bgtz r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sub r4, r2, r5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sw 0(r2), r1</td>
<td>IF</td>
<td>IDx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>and r6, r1, r8</td>
<td>IFx</td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 
TARGET: ! TARGET = 0x200

0x200 xor r9, r4, r11 IF ID EX ME WB

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

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

Two cycles called a branch penalty.

Two cycles is alot of cycles, is there something we can do?
Compute branch target address in ID stage.
Zero-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 can be computed quickly.

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>bg tz r4, TARGET</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td>sub r4, r2, r5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>ME</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x108</td>
<td>sw 0(r2), r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10c</td>
<td>and r6, r1, r8</td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... TARGET: ! TARGET = 0x200

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

No penalty, not a cycle wasted!!
Non-Bypassed MIPS
Bypassed MIPS