DLX Implementation

06-1

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.

Unpipelined Implementation

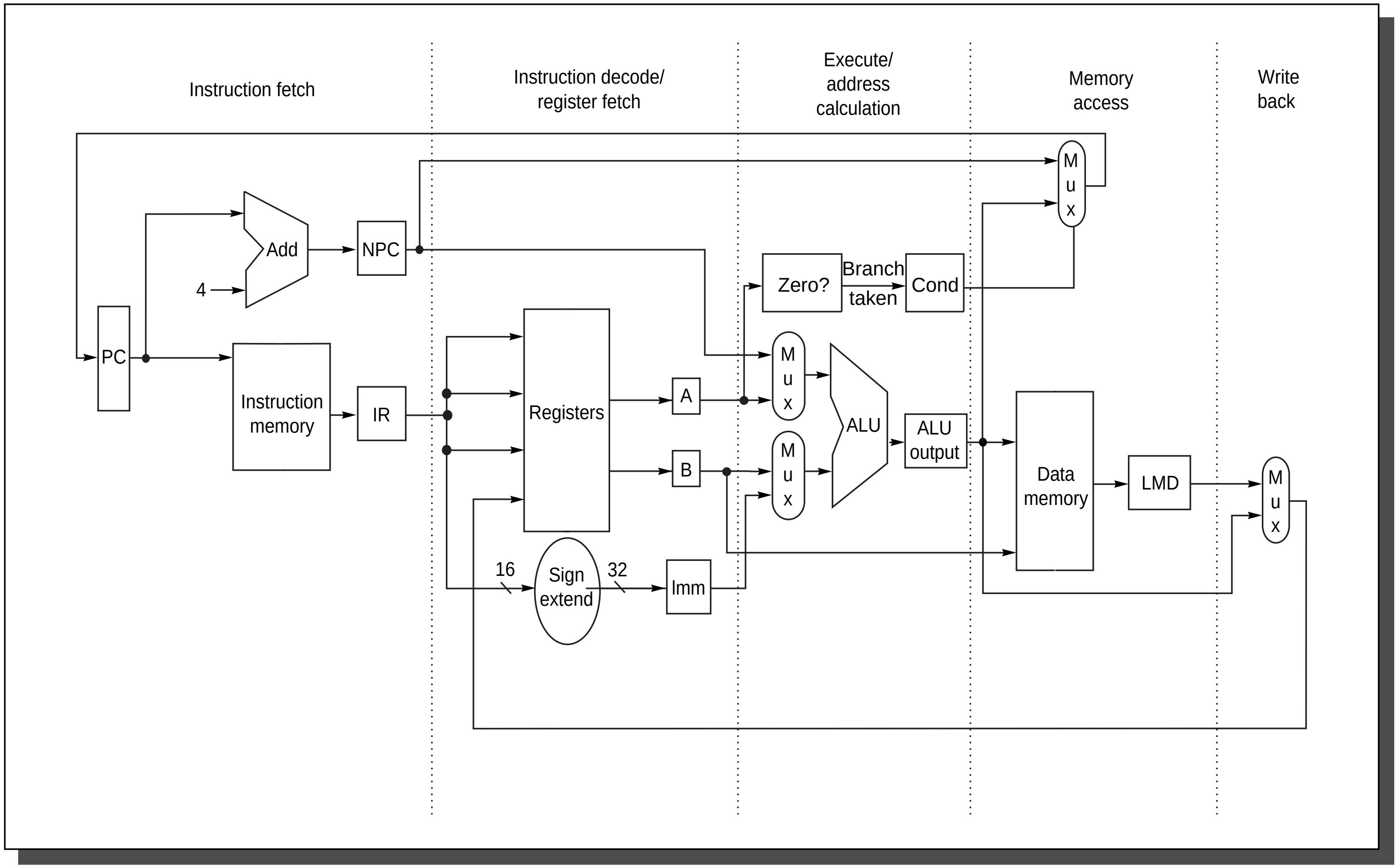
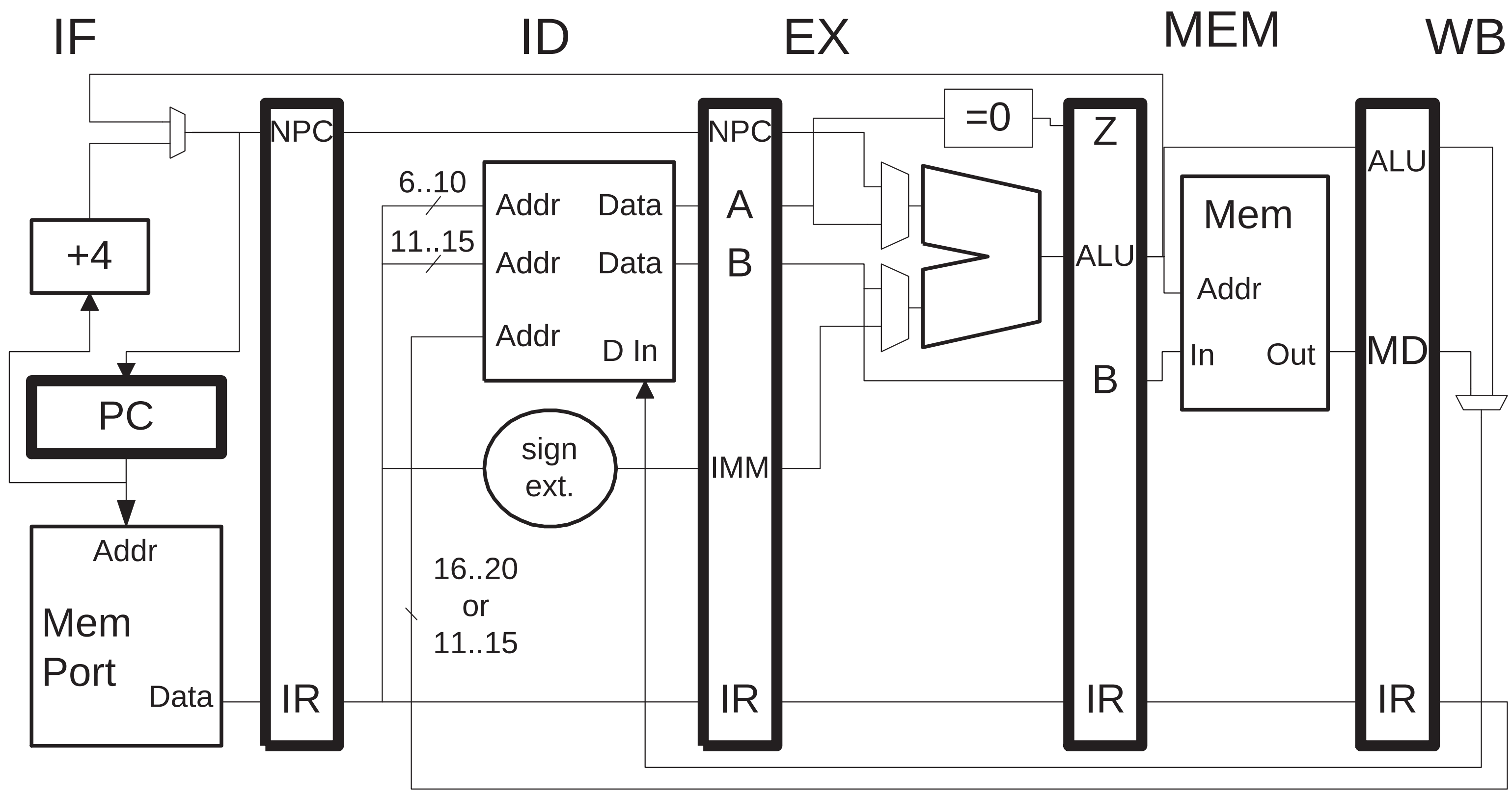


FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.

Pipelined Implementation



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 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`.

06-5

Pipeline Execution Diagram

Pipeline Execution Diagram

Used to show how instructions execute.

Along horizontal axis show time, vertical axis static instructions.

Label points with segment that instruction is in.

```
Cycle             0   1   2   3   4   5   6
add r1, r2, r3    IF  ID  EX  MEM WB
and r4, r5, r6        IF  ID  EX  MEM WB
lw  r7, 8(r9)             IF  ID  EX  MEM WB
```

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.

06-5

Instruction Decoding and Pipeline Control

06-6

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

06-8

A *hazard* is a potential execution problem due to the hardware.

Hazards are caused by overlapping of instructions.

Hazards are avoided using *interlocks* or other means.

Hazard Types:

- *Structural Hazard*
  Needed resource not available.

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

- *Control Hazard*
  Needed instruction not 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

06-10

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

06-12

Chapter-3 DLX Subject to RAW Hazards.

Consider the following **incorrect execution**:

```
! Cycle           0   1   2   3   4   5   6   7
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
```

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.

06-12

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

With stalls, code can execute correctly:

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

```
! 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
```

Arrow shows that instructions stalled.

Stall creates a *bubble* 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` in after the stalling instruction and . . .

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

```
! 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
```

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

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

The two adjacent `nop`s 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:

```
! 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
```

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.

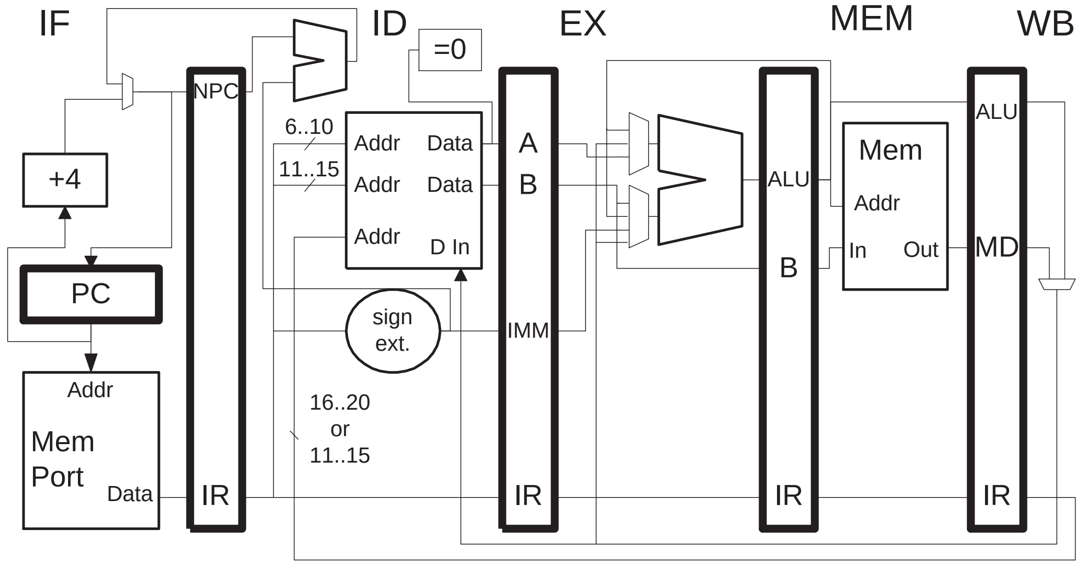
Execution was incorrect because the value had to go around the pipeline to ID.

Why not provide a shortcut?

Why not call a shortcut a *bypass* or *forwarding* path?

DLX Implementation With Some Forwarding Paths:

06-16

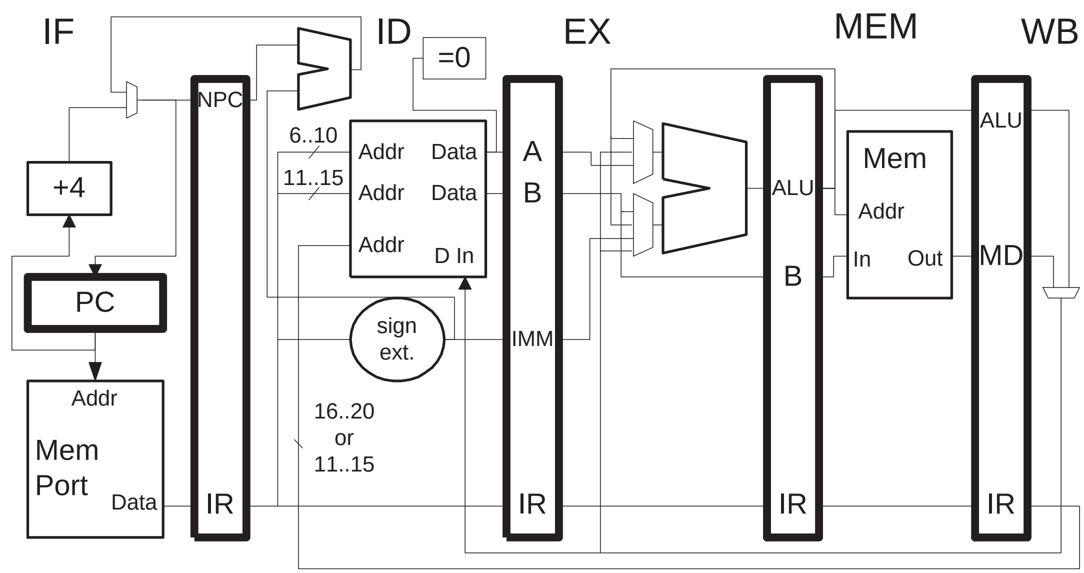



```
! Cycle            0    1    2    3    4    5    6    7    8    9    10
add  r1, r2, r3    IF   ID   EX   MEM  WB
sub  r4, r1, r5         IF   ID   EX   MEM  WB
and  r6, r1, r8              IF   ID   EX   MEM  WB
xor  r9, r4, r11                  IF   ID   EX   MEM  WB
```

It works!

DLX Implementation With Some Forwarding Paths:

Not all stalls are avoidable.

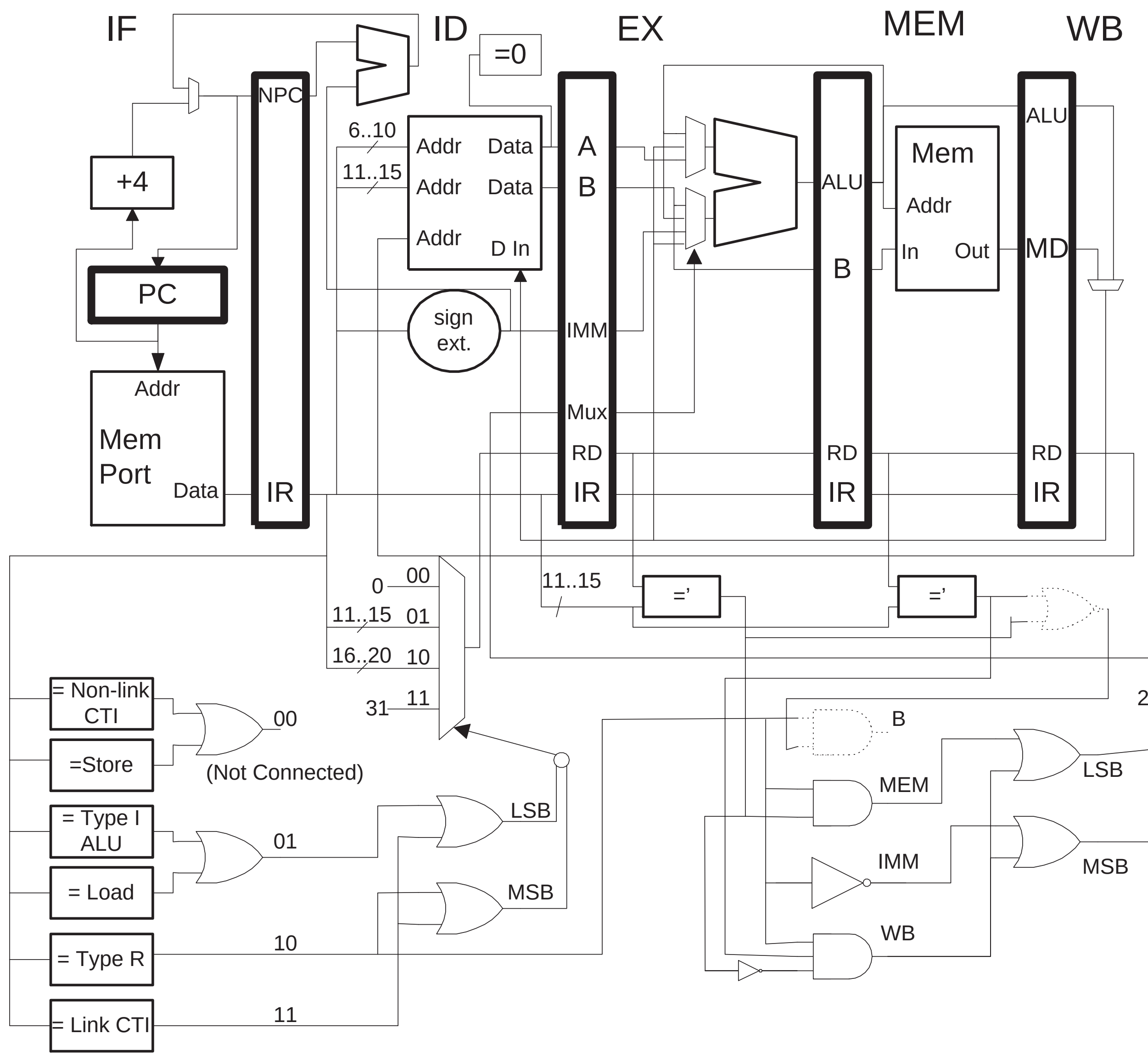



```
! Cycle          0   1   2   3   4   5   6   7   8   9   10
lw   r1, 0(r2)   IF  ID  EX  MEM WB
add  r1, r1, r4      IF  ID  ->  EX  MEM WB
sw   4(r2), r1           IF  ->  ID  ----->  EX  MEM WB
addi r2, r2, #8                  IF  ----->  ID  EX  MEM WB
```

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



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 `= Load` (some instructions omitted):

`= 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`.

$\boxed{=^{\prime}}$ 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.

Example of **incorrect execution**

```
!I Adr          Cycle   0   1   2   3   4   5   6   7   8
0x100  beqz r4, TARGET  IF  ID  EX  MEM WB
0x104  sw   0(r2), r1       IF  ID  EX  MEM WB
0x108  sub  r4, r2, r5          IF  ID  EX  MEM WB
0x10c  and  r6, r1, r8              IF  ID  EX  MEM WB
0x110  or   r12, r13, r14
...
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.

```
!I Adr          Cycle   0   1   2   3   4   5   6   7   8
0x100  beqz r4, TARGET  IF  ID  EX  MEM WB
0x104  sw   0(r2), r1       IF  ID  EX  MEM WB
0x108  sub  r4, r2, r5          IF  ID  EX  MEM WB
0x10c  and  r6, r1, r8              IF  ID  EX  MEM WB
0x110  or   r12, r13, r14
...
TARGET: !  TARGET = 0x200
0x200  xor  r9, r4, r11                 IF  ID  EX  MEM WB
```

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.

Squashing Instructions

Also called: *nulling*, *abandoning*, and *cancelling* instructions.

Squashing: preventing an *in-flight* instruction ...
... from writing registers, memory or any other visible storage.

In-Flight Instruction: An instruction anywhere in the execution pipeline.

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

Squashing Instruction in Example DLX Implementation

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

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

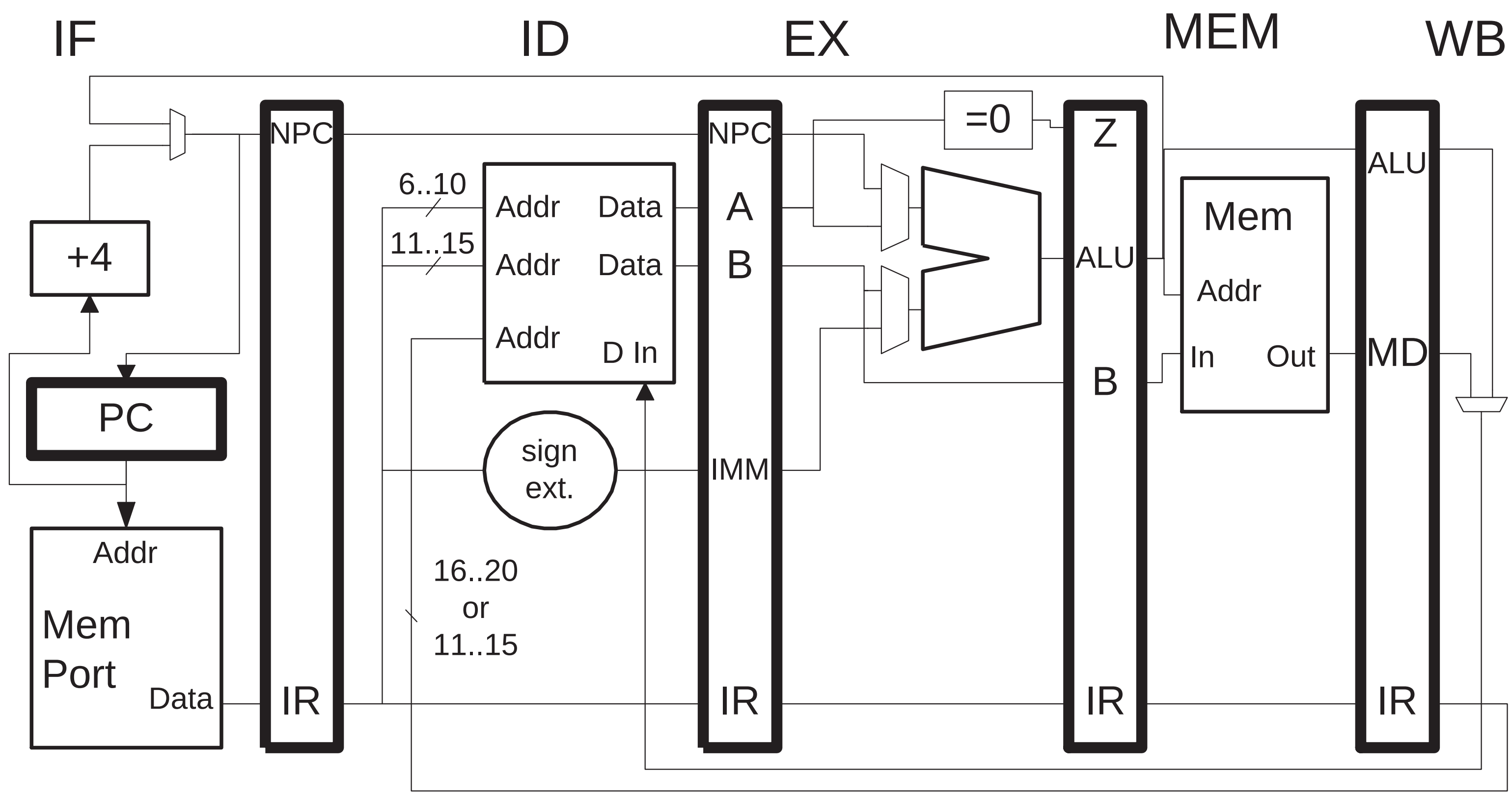
For simplicity, squashing sometimes shown as placing `nop` in `IR`.

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.)

DLX implementation used so far.



Example of correct execution

```
!I Adr           Cycle   0   1   2   3   4   5   6   7   8
0x100  beqz r4, TARGET  IF  ID  EX  MEM WB
0x104  sw   0(r2), r1       IF  ID  EXx
0x108  sub  r4, r2, r5          IF  IDx
0x10c  and  r6, r1, r8              IFX
0x110  or   r12, r13, r14
...
TARGET: !  TARGET = 0x200
0x200  xor  r9, r4, r11                 IF  ID  EX  MEM WB
```

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 `nop`s.

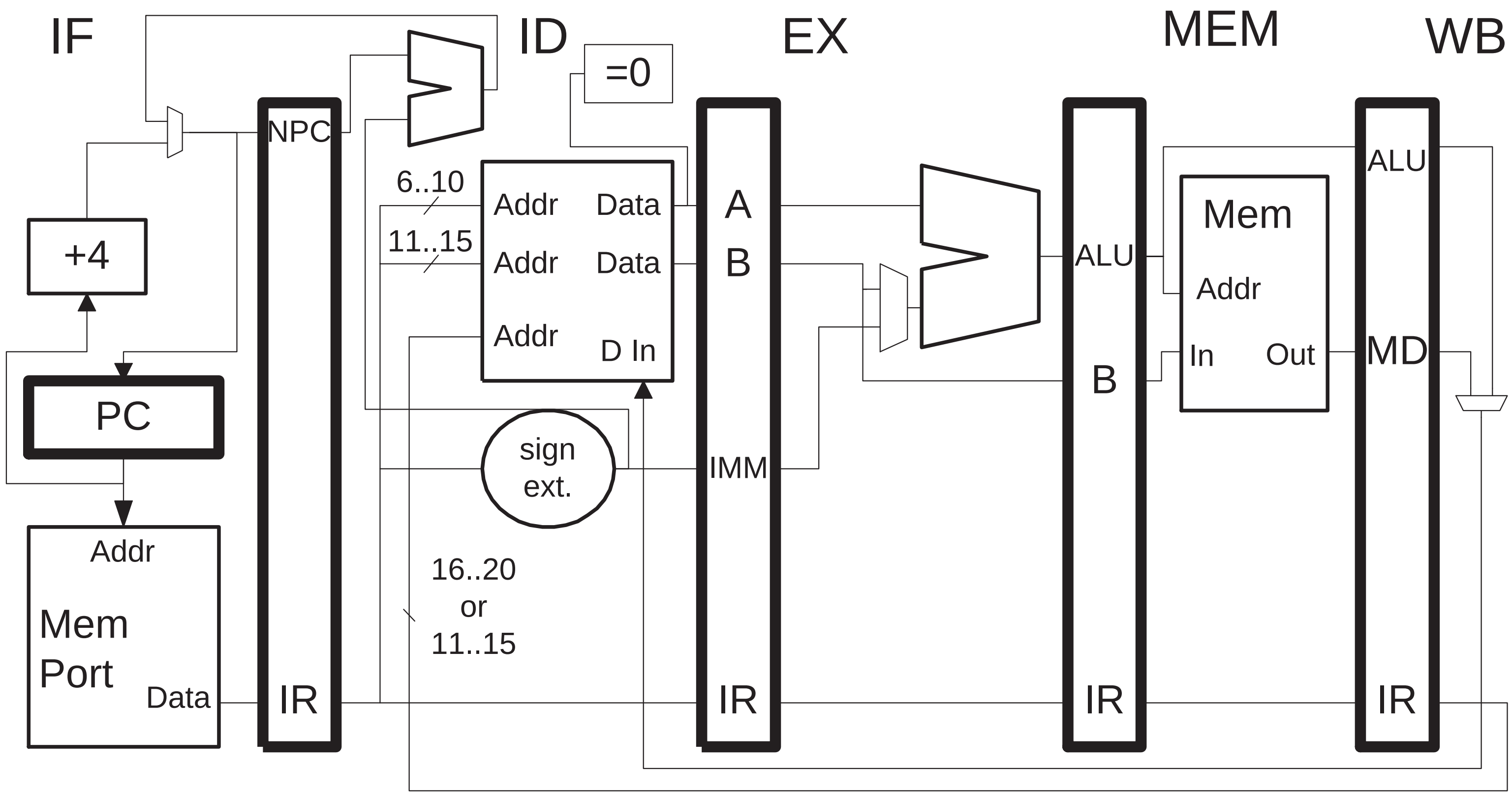
Three cycles (1, 2, and 3), are lost.

Three cycles called a *branch penalty*.

Three cycles is alot of cycles, is there something we can do?

Yes: One-Cycle Branch Delay Implementation

06-28



Compute branch target address in ID stage.

EE 4720 Lecture Transparency. Formatted 13:18, 7 February 2000 from lsli06.

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?

```
!I Adr           Cycle   0   1   2   3   4   5   6   7   8
0x100  beqz r4, TARGET  IF  ID  EX  MEM WB
0x104  sw   0(r2), r1       IFx
0x108  sub  r4, r2, r5
0x10c  and  r6, r1, r8
0x110  or   r12, r13, r14
...
TARGET: !  TARGET = 0x200
0x200  xor  r9, r4, r11         IF  ID  EX  MEM WB
```

Penalty is only one cycle.

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