Data Hazards

Compiler Scheduling

Pipeline scheduling or instruction scheduling: Compiler generates code to eliminate hazard

Consider:

```
a = b + c;
d = e - f;
```

Assume loads have a latency of one clock cycle:

```
LW Rb, b
LW Rc, c
LW Re, e;         Swapped with next to avoid stall.
ADD Ra, Rb, Rc
LW Rf, f
SW a, Ra;         Store/Load exchanged to avoid stall.
SUB Rd, Re, Rf; Forwarded (Rd)
SW d, Rd
```

Both load interlocks eliminated

Data Hazards

Compiler Scheduling: Observations

Note that pipeline scheduling *increases* the number of registers used

Compiler algorithms that perform this optimization can do so easily for code in *basic blocks*

Basic blocks are code sequences with no branches

If one instruction executes, they all do

Method is simple and effective for DLX with a latency for loads of 1 cycle

As latencies become longer, more aggressive strategies are needed

Data Hazards

Instruction Issue:

The process of letting an instruction move from ID to EX

For DLX integer pipe:

All data hazards can be checked during ID phase of pipe and instruction stalled if necessary (i.e. load interlock)

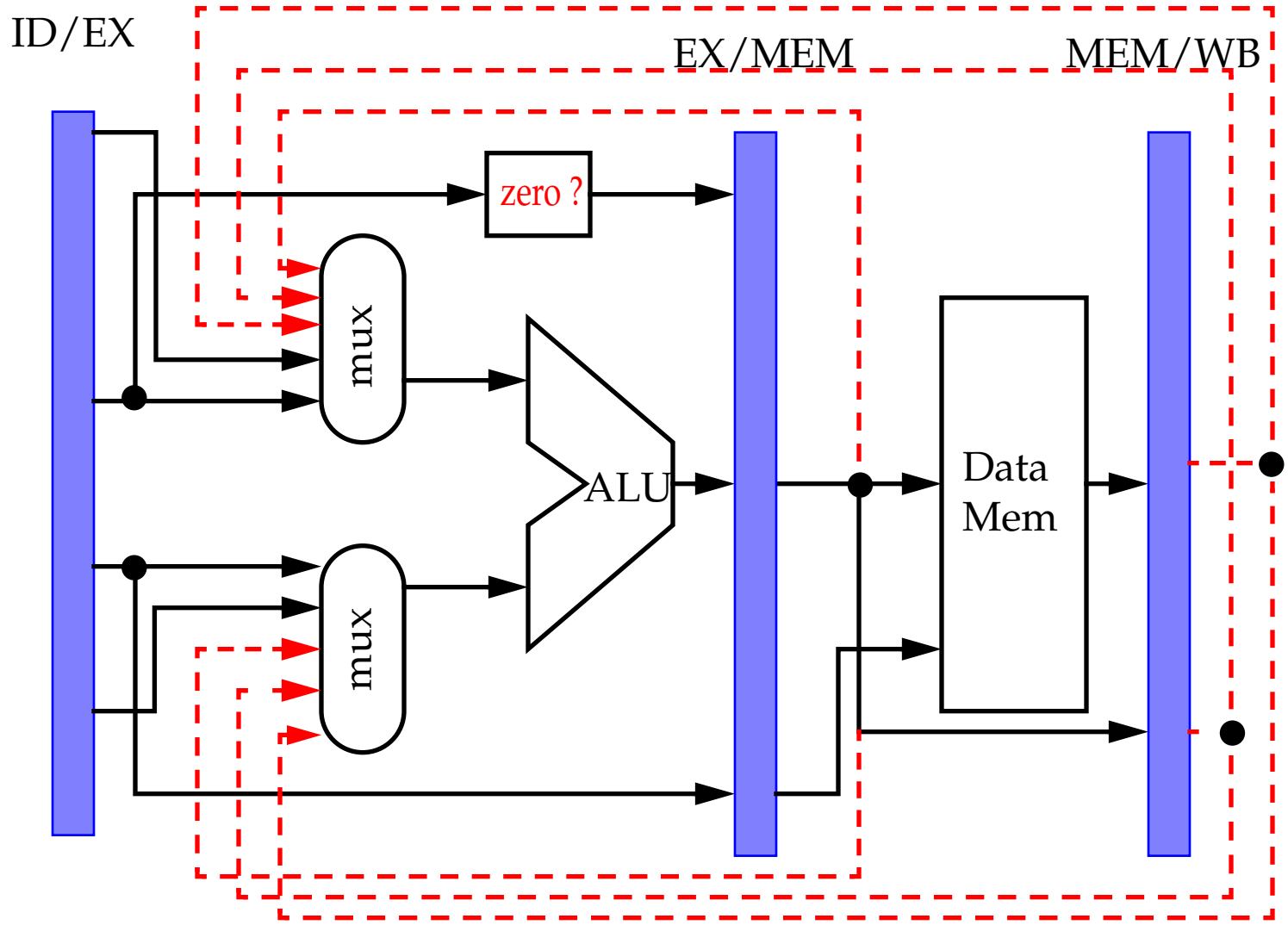
Forwarding always works for R-R instructions, but only sometimes for loads

Situation	Example code	Action
Dependence requiring stall	LW **R1**, 45(R2) ADD R5, **R1**, R7	Comparators detect the use of R1 and stall the ADD before EX

Dependence overcome by forwarding	LW **R1**, 45(R2)
	ADD R5, R6, R7
	SUB R8, **R1**, R7
		Comparators detect use of R1 in SUB and forward result of load to ALU

Data Hazards

New DLX Datapath:



Control Hazards

These have a greater performance impact on DLX than data hazards

However, they are similar in that:

Data	Control
Occur when 2 instructions overlap such that their accesses to a particular register are *reordered*	Occur when 2 instructions accesses to the **PC** are *reordered* and the first instruction modifies the PC (RAW)

Since every instruction uses the PC in its first cycle (IF), this hazard always occurs when an instruction writes the PC, (i.e. after a branch)

For DLX, the PC is not normally updated until the first half of WB (after address calculation and comparison)

Control Hazards

Solutions:

- *Simple*: Stall the pipeline

  This results in a 3 cycle stall for every branch

Clock Number

	1	2	3	4	5	6	7	8	9
Branch instr.	IF	ID	EX	MEM	WB				
Branch successor		IF	stall	stall	IF	ID	EX	MEM	WB
Branch successor + 1						IF	ID	EX	MEM
Branch successor + 2							IF	ID	EX

Why does this happen ?

Note that if the branch is NOT taken, then the instruction in IF is the correct one

This can be taken advantage of

Analysis: With a 30% branch frequency and an ideal CPI of 1, this simple solution achieves only about **half** of the ideal (1.9 CPI)!

Control Hazards

Solutions:

Reducing the branch penalty can be achieved by:

- Deciding whether or not the branch is taken earlier in the pipeline
- Computing the target address earlier in the pipeline

Note that these two methods have *limited* usefulness individually

i.e. It doesn't help to know the target without knowing the outcome of the branch

For the DLX:

BEQZ and BNEZ require testing a register

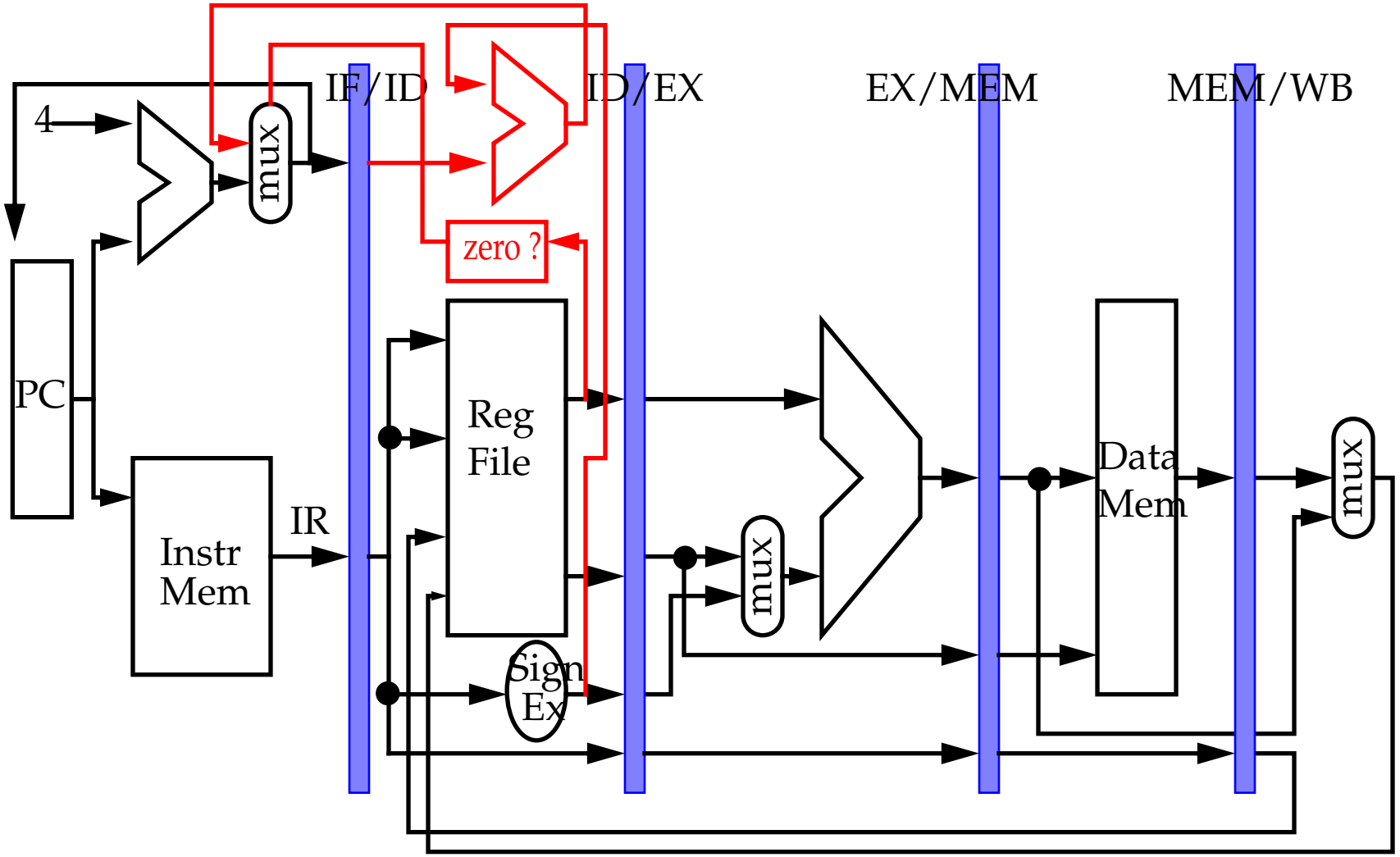
We can do this by moving the *zero test* unit into ID

Both PCs must be computed in order to take advantage of this (need an extra adder)

Branch delay can be reduced to 1 cycle

Control Hazards

DLX Revised datapath:



What about data hazards? (ALU instruction followed by a branch?)

Control Hazards

Branch Behavior: Observations using SPEC subset on DLX:

These are *dynamic* frequencies - not *static* frequencies:

What matters is executions, not the number of times the instructions occur in the programs

- Conditional branches are much more common:

  Conditional outnumber unconditional about 3-4 to 1

  14% to 16% is normal for integer benchmarks (FP benchmarks are much more varied at 3%-12%)

- Forward branches more common:

  Forward branches outnumber backwards 3 to 1

Frequency of *taken* branches:

67% of conditional branches are taken on average

60% of the forward and 85% of the backward branches

Control Hazards

Reducing branch penalties (**static prediction** schemes):

Assume we moved the address calculation and decision of whether to take the branch back into ID

Therefore, if the comparison is to *zero*, we know the address and the decision at the end of ID

If comparing one register to another, we wait until after EX to decide if the branch is taken

Control Hazards

Reducing branch penalties (**static** prediction schemes):

- *Simple*: Freeze/flush the pipeline:

  This method always flushes the pipeline of instructions up until the branch destination and condition are known.

  Branch penalty is fixed and cannot be reduced by software (the compiler.)

- Treat every branch as *not taken* (**predict-not-taken**):

  Continue to fetch instructions.

  Flush the pipeline if the branch is taken.

  Note that successor instructions can NOT change the state of the machine (or, if they do, we must be able to restore the state if branch is taken).

  This results in a 1 cycle stall for DLX since the decision (for zero compares) is known after ID.

Control Hazards

Reducing branch penalties (**static** prediction schemes):

- Treat every branch as *taken* (**predict-taken**)

  Wait until the target address is computed and then fetch instructions using the new PC value.

  Flush the pipeline if the branch is NOT taken.

  For DLX, this doesn't do much good since BOTH the branch target address and the decision (for zero compares) is known after ID.

  If the decision is delayed until EX (i.e. Reg-reg compares for DLX), this method may help.

Control Hazards

Reducing branch penalties (**static** prediction schemes):

- **Delayed branch**

  An execution cycle with a branch delay of length n is:

```
branch instruction
sequential successor_1
sequential successor_2
...
sequential successor_n
branch target if taken
```

Branch delay slots

Note that the instruction(s) in the branch delay slot(s) after the branch are always executed.

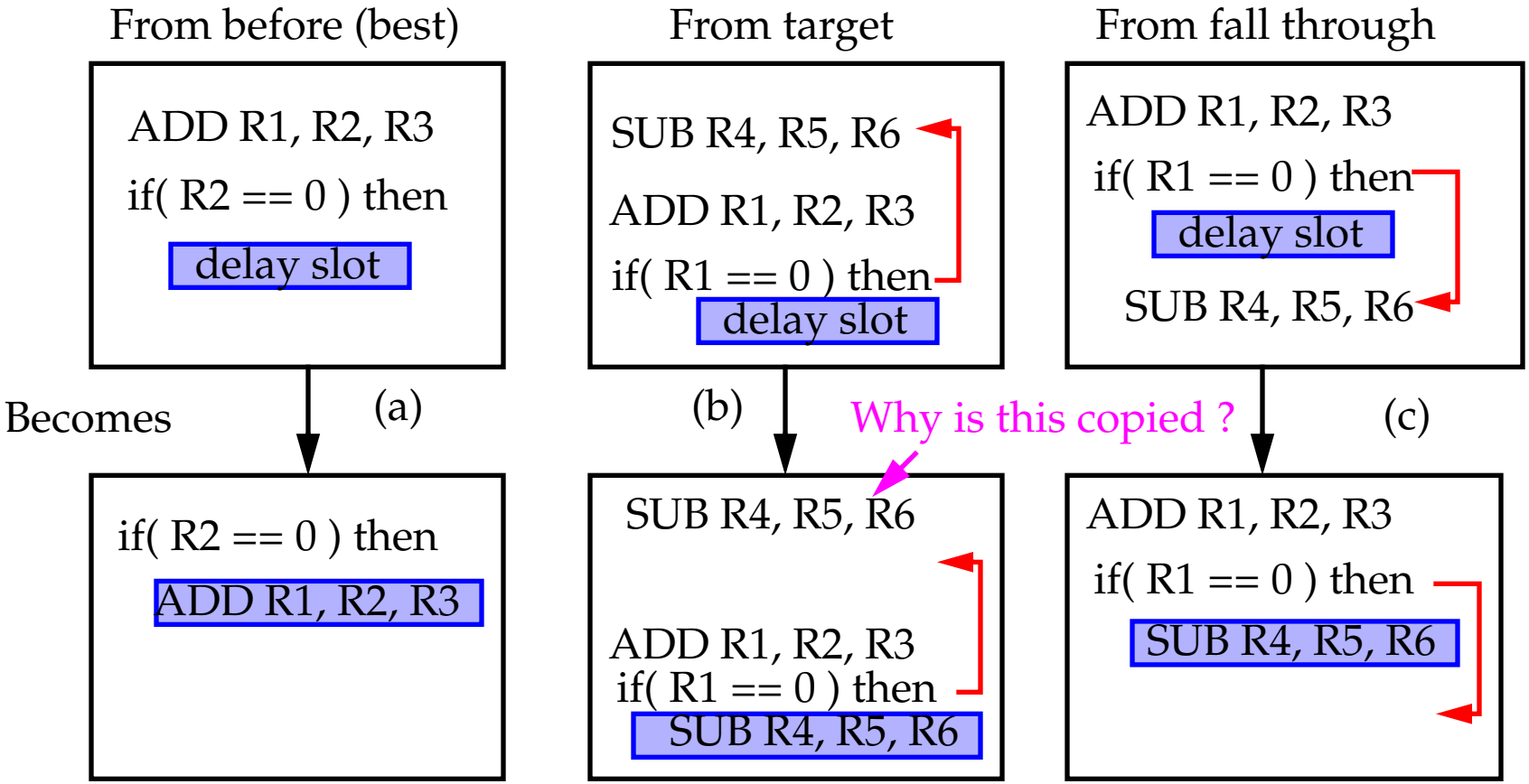
The compiler should try to put a "useful" instruction here.
If none are available, then a "no-op" is inserted in the delay slot.

This improves performance by letting the CPU do useful work while waiting for the branch target and condition to be determined.

Control Hazards

Delayed branch: DLX has a one branch-delay slot.

Possible compiler scheduling optimizations:



For schemes b and c,

It must be O.K. to execute the SUB instruction if the prediction is wrong.

Or the hardware must provide a way of cancelling the instruction.

Control Hazards

Delayed branch:

Limitations to the usefulness of this approach:

- There are restrictions on the instructions that are scheduled into the delay slots (i.e. data dependencies.)
- The compiler's ability to predict accurately whether or not a branch is taken determines how much useful work is actually done.

Many machines have introduced a *cancelling* or *nullifying* branch instruction.

Includes the direction that the branch is predicted to go.

If branch is predicted *incorrectly*, CPU turns the instruction in the branch delay slot into a no-op.

For machines with a branch delay of 1 (i.e. DLX), the compiler can go along way towards improving performance.

However, for longer branch delays, it gets more difficult to fill the delay slots with useful work.