Control Hazards

- Instructions that alter the PC result in
  - discarding of started partial computations
  - refilling of the pipe.
- Worse...(i.e. more difficult to ensure correctness)
  - partial computations started which are after the branch must not result in permanent data alteration
  - this is not a serious problem in DLX
    - because memory alterations occur at the last (WB) stage
  - but IS a serious problem in efficiently pipelining CISC architectures
    - for example, due to predecrement or postincrement addressing modes
    - MOVE.L (A0++), (A1)
      - move the longword pointed to by A0 to where A1 points, incrementing A0 afterwards
    - MOVE.L (--A0), (A1)
      - decrement A0, then move the longword pointed to by A0 to where A1 points
    - if these instructions occur after a branch, A0 must not be (permanently) altered

Copyright 1998-2004 © L.S. Smith

What happens when a Branch Occurs

- What is the result of a branch instruction?

Branch		IF	ID	EX	MEM	WB		
Branch+1	(| WB)	IF	stall	stall	IF'	ID	EX	MEM
Branch+2			(WB)	XXX	XXX	XXX	IF	ID	EX
Branch+3				(WB)	XXX	XXX	XXX	IF	ID

- There is a 3-cycle stall
  - The first IF in instruction Branch+1 is wasted, and does no useful work
- This is the worst-case penalty
  - but it does occur every time a branch is *taken*
    - (for conditional branches, a branch is said to be *taken* if the condition is TRUE, and *not taken* if the condition is false)
  - branches are frequently occurring events
    - so it's important to reduce their overhead.
  - Note that longer pipelines suffer even more when the PC is altered

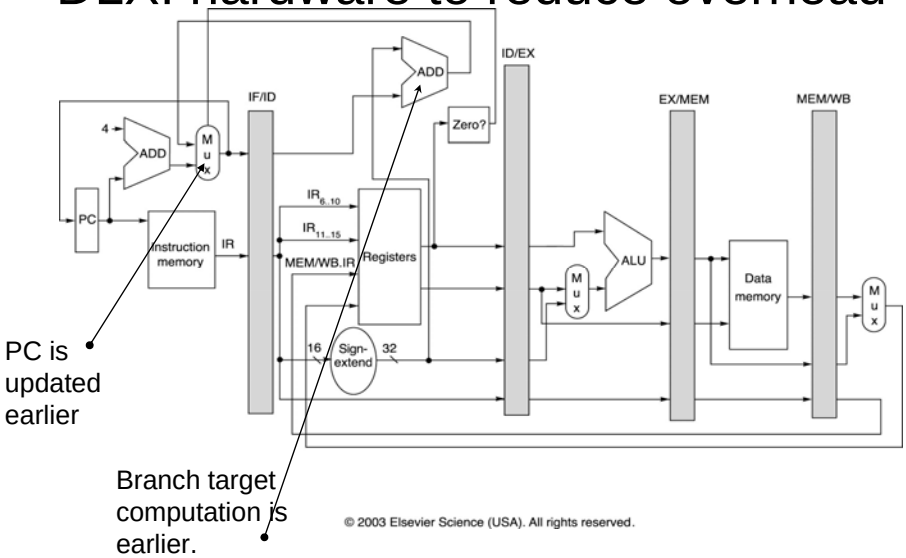
Copyright 1998-2004 © L.S. Smith

Reducing the overhead of control hazards (1)

- How to reduce the control hazard overhead:
  - Find out whether the branch is *taken* as early as possible
    - (if it's a conditional branch)
  - Compute the taken PC earlier
- Both of these steps need to be taken
- In DLX
  - we can move the test and branch target calculation back to the ID stage
  - this has a cost: an additional ALU purely for branch address computation

Copyright 1998-2004 © L.S. Smith

DLX: hardware to reduce overhead



PC is updated earlier

Branch target computation is earlier.

© 2003 Elsevier Science (USA). All rights reserved.

Copyright 1998-2004 © L.S. Smith

Effect of change in hardware

- Note that we now have available the target of a branch instruction immediately after the ID pipeline stage
- Taken branch instruction

Branch		IF	ID	EX	MEM	WB
Branch+1	(WB)	IF	idle	idle	idle	
Branch target		(WB)	IF	ID	EX	
Branch target+1			(WB)	IF	ID	EX ...

Copyright 1998-2004 © L.S. Smith

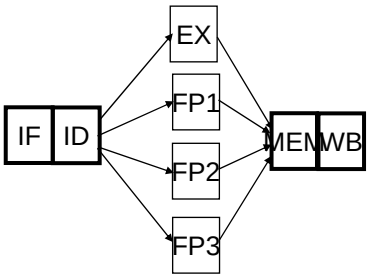
Reducing the overhead of control hazards (2)

- Generally, most conditional branches are not taken
  - (results from dynamic analysis of code performance)
- so we should optimise performance under this condition
- and take a predict-not-taken approach
  - i.e. optimise performance assuming that branches will not be taken
  - while at the same time ensuring that we do not write back any results until we are sure about the effect of the branch.
- Another alternative is to start executing both the instruction after the branch instruction and the instruction at the target address
  - 'greedy' execution
  - in a linearly pipelined processor, this is only useful if the branch condition cannot be tested early in the pipe

Copyright 1998-2004 © L.S. Smith

Multi-cycle operations and pipelining

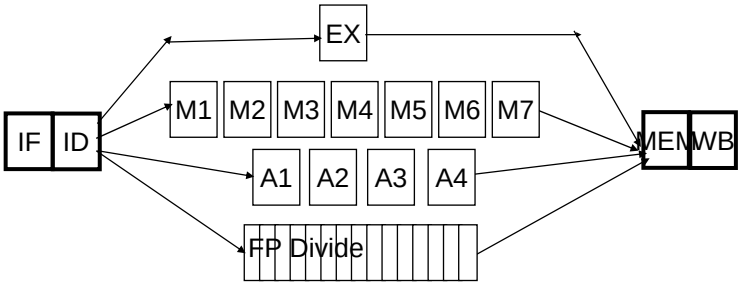
- The DLX processor integer execution unit completes operations in 1 cycle
- This is unlikely to be possible for floating-point (FP) operations
  - either we would require a slow clock, or an extraordinarily complex FP ALU
- Let's assume 3 FP execution units:
- FP1: FP/integer multiply
- FP2: FP adder
- FP3: FP/integer divider
- Each FP stage has a delay
- larger than 1
- realistically:
  - FP multiply: 6
  - FP add: 3
  - FP divide 14



Copyright 1998-2004 © L.S. Smith

Realistic FP pipelining

- We can expect to pipeline the FP units - but pipelining the divide unit is difficult and unlikely to be sensible usage of silicon real estate
- A pipeline that supports multiple outstanding FP operations



Copyright 1998-2004 © L.S. Smith

Effects of FP pipeline on execution

- Consider the sequence of instructions:

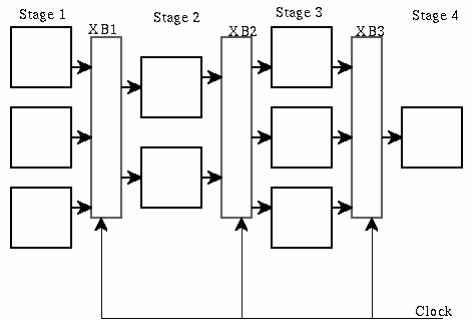
FPMUL	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
FPADD		IF	ID	A1	A2	A3	A4	MEM	WB		
LOAD			IF	ID	EX	MEM	WB				
STORE				IF	ID	EX	MEM	WB			

- Instructions can complete is a different order from that in which they were dispatched
- Because the divide unit is not pipelined, structural hazards can occur
  - these need to be detected , and the pipe stalled
  - but because FP divides are rare, these should not cause a large performance penalty
- Since instructions have different running times, more than 1 register write may be required in a single cycle
  - may need multi-ported register files
- WAW and RAW hazards become more common.

Copyright 1998-2004 © L.S. Smith

Multiple functional units and delay equalisation

- Another way of improving pipelining performance is to replicate functional units, in order to equalise the delays in each pipeline stage
- A *Crossbar switch* can be used to select which functional unit will be connected to which other functional unit
- By using more than one (identical) unit , and switching between them, pipe stage delays can be equalised, making the pipeline efficient.
- Pipe hazards remain a problem.



Copyright 1998-2004 © L.S. Smith

Exception handling in a pipelined processor

- Exceptions (=interrupts, faults, etc.) can occur in a pipelined processor
  - These need to be dealt with correctly
  - and the pipe needs to be restartable for some of these
    - most notably page faults
- Synchronous and asynchronous events
  - synchronous events are events that always occur in the same place inside a program
  - asynchronous events are caused by devices external to the CPU/memory, and are normally dealt with after completion of the current instruction.
- Synchronous events can occur inside the execution of an instruction, and stop that instruction from completing
  - e.g. accessing an operand which needs fetched from a non-resident page
  - or attempting an invalid instruction
- Some events are *resume* events, and some are *terminating* events.

Copyright 1998-2004 © L.S. Smith

Precise and imprecise exceptions

- If the pipe can be stopped in such a way that
  - all instructions before the faulting instruction are completed
  - all instructions after the faulting instruction can be restarted from scratch
- it is said to have *precise exceptions*.
- Precise exceptions are relatively easy to provide for integer operations but more difficult to provide in FP operations

Copyright 1998-2004 © L.S. Smith

Possible solutions to imprecise exceptions

- ignore the problem, and have imprecise exceptions
  - not compatible with virtual memory page fault exceptions
- buffer all the results until all instructions dispatched earlier are complete
  - expensive, since number of results can be large
- allow some imprecision, but keep enough information so that the trap routines can finish any instructions already started *after* the exception
  - this may entail limiting the number of overlapping FP instructions
  - used in SPARC architecture
- allow instruction issue only if earlier instructions will not cause an exception
  - this can mean stalling the pipeline sometimes
  - it also means detecting possible exceptions early in the pipeline

Copyright 1998-2004 © L.S. Smith