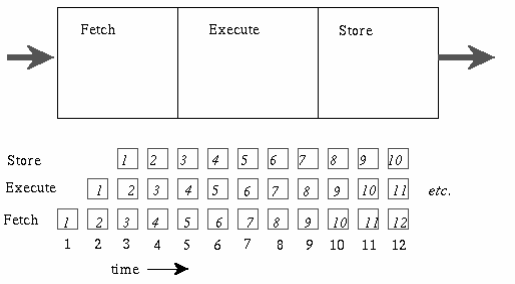
Instruction Pipelining

- Simplest form is a 3-stage linear pipeline
- New instruction fetched each clock cycle
- Instruction finished each clock cycle
- Maximal speedup = 3 achieved if and only if
  - all pipe stages are the same length
  - instructions and operands can be fetched quickly enough
  - results can be stored quickly enough



Copyright 1998-2004 © Leslie S. Smith

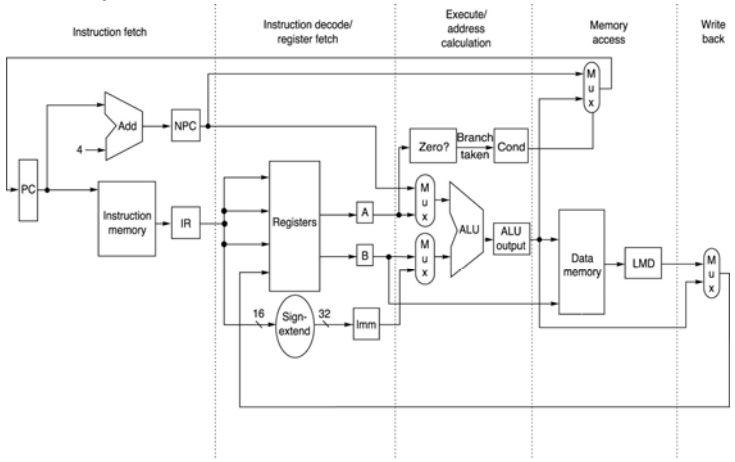
Longer linear pipelines

- Because pipeline speedup is directly proportional to pipe length, longer pipelines are attractive
- e.g.
  - Fetch Instruction
  - Decode Instruction
  - Operand Address Generate or Execute
  - Operand Fetch
  - Store Result
- These stages are unlikely to be all of the same length, so sometimes null stages are added to compensate
  - Fetch Instruction
  - NULL
  - Decode Instruction
  - Operand Address Generate
  - Operand Fetch
  - NULL
  - Store Result
  - NULL

Copyright 1998-2004 © Leslie S. Smith

A pipelining example: DLX

- (Taken from Hennessy & Patterson) DLX has a five - stage architecture. DLX is a RISC processor



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

DLX stages

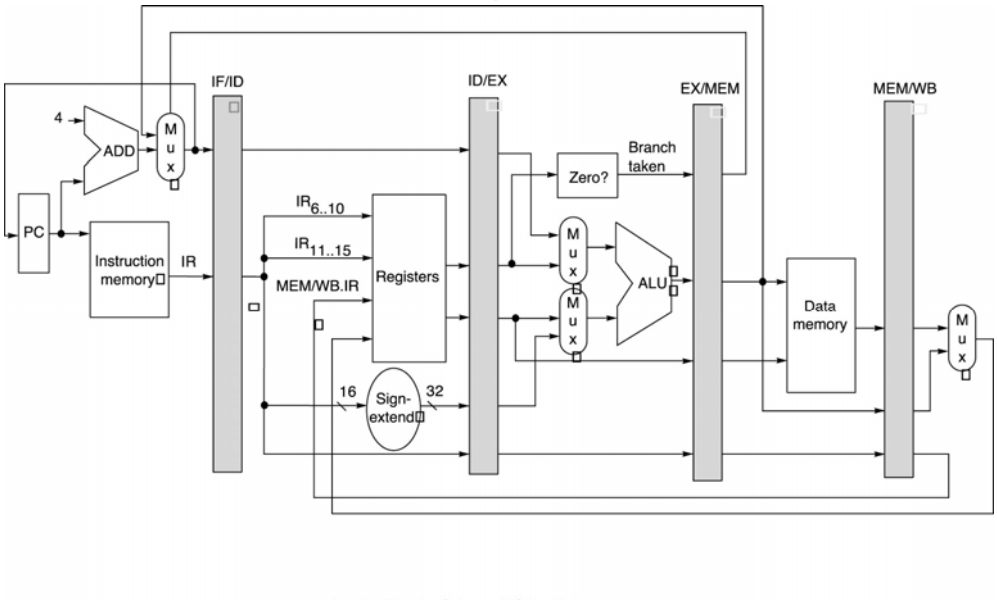
- IF - Instruction Fetch
  - Fetch instruction from memory to IR
  - increment PC (NPC <- PC + 4)
- ID - Instruction fetch/register decode
  - Decode the instruction and access the register file to read the register(s) into temporary registers A and B
  - Also sign-extend lower 16 bits of IR (Imm <- (sign-extended) IR15-0)
- EX- Execution/effective address cycle
  - for a memory reference instruction
    - ALU.Output <- A + Imm
  - for a register-register instruction
    - ALU.Output <- A *function* B
  - for a branch instruction
    - ALU.Output <- NPC + Imm
    - Cond <- A *op* 0

DLX stages (cont)

- MEM - memory access/branch completion cycle
  - for a memory reference instruction
    - LMD <- Mem[Alu.Output] or
    - Mem[ALU.Output] <- B
  - for a branch instruction
    - if (cond) PC <- ALU.Output else PC <- NPC
- WB - Write-back cycle
  - for a register-register ALU instruction
    - Regs[IR16-20] <- ALU.Output]
  - for a Register- Immediate Instruction
    - Regs[IR11-15] <- ALU.Output
  - for a load instruction
    - Regs[11-15] <- LMD
- Notes
  - the memories referred to are all cache memories
    - there are two caches, an instruction cache and a data cache

Copyright 1998-2004 © Leslie S. Smith

DLX Pipelined



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

Copyright 1998-2004 © Leslie S. Smith

Speedup of DLX

- DLX pipeline has a five-stage pipeline, so the speedup is 5 (?)
- No
  - not all instructions use all stages
    - MEM stage is not used at all by register-register ALU instructions
  - the latch itself has an overhead as well
    - even if it is not very large
  - actual speedup is approximately 4 times
    - though this depends on the relative frequency of he different instruction
      - known as the instruction mix.
- More importantly, there are reasons why one cannot expect the pipe to fill up and remain full: there are *hazards*.

Copyright 1998-2004 © Leslie S. Smith

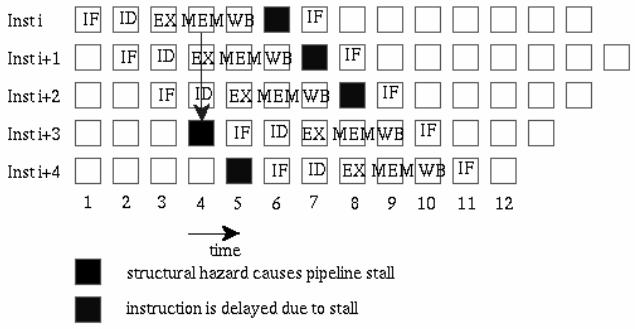
Pipeline Hazards

- There are three different types of hazard which
  - prevent the instruction stream from executing
    - resulting in the pipe not being kept full at all times.
- Structural Hazards
  - these arise form *resource conflicts*
  - the hardware cannot support all the possible combinations of instructions in the pipe
- Data Hazards
  - these arise when an instruction depends on the result of a previous instruction
  - this result may not yet have been stored, or even computed.
- Control Hazards
  - these arise from attempting to pipeline instructions that themselves affect the flow of control
  - that is, they affect the program counter (e.g. jumps, branches, function calls, etc.)

Copyright 1998-2004 © Leslie S. Smith

Structural Hazards

- Can occur (e.g.) if some function requires a unit which is not pipelined
  - e.g. floating-point units are sometimes not pipelined, and the performance of the pipe decreases severely when there are many FP instructions
    - but generally, these are not so very common
- If the DLX did not have a separate instruction and data cache, but had only a single port on to memory
  - the IF and MEM stages could provide a structural hazard



Copyright 1998-2004 © Leslie S. Smith

Avoiding structural hazards

- RISC instructions are much more predictable than CISC ones
  - and this makes structural hazards easier to avoid
- These hazards can be avoided by adding more hardware
  - though not an option on an existing processor, this becomes easier on new versions of a processor because of the improvements in manufacturing technology
- Note that dynamic examination of code can show up the likelihood of particular structural hazards
  - and suggest whether additional hardware will give a reasonable improvement or not.

Copyright 1998-2004 © Leslie S. Smith

Data Hazards

- Can occur because the pipelining reorders execution from the intuitive order.
- Consider

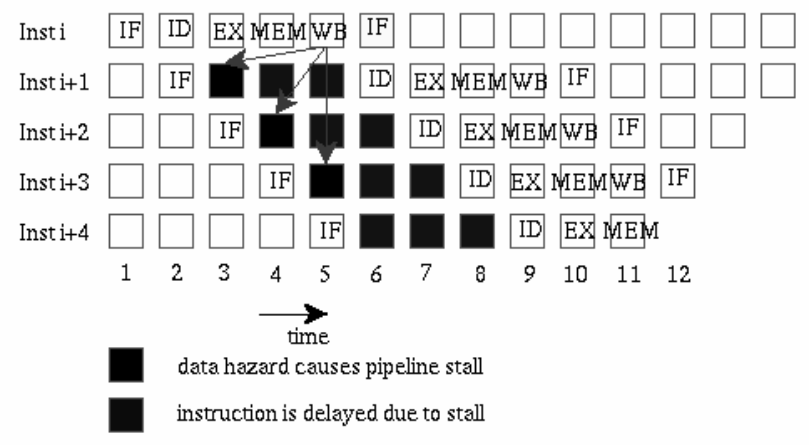
```
ADD   R1, R2, R3   // R1 <- R2 + R3
SUB   R4, R1, R5   // R4 <- R1 - R5
AND   R6, R1, R7   // R6 <- R1 AND R7
OR    R8, R1, R9   // R8 <- R1 OR R9
```

- R1 is computed in the first instruction
  - and written during the last (WB) pipe stage
- and R1 is used in the 3 following instructions
  - and accessed during the ID stage
- but the WB stage of the first instruction will not have run yet
  - so (if nothing was done) the wrong value for R1 would be used

Copyright 1998-2004 © Leslie S. Smith

Solving data hazards I

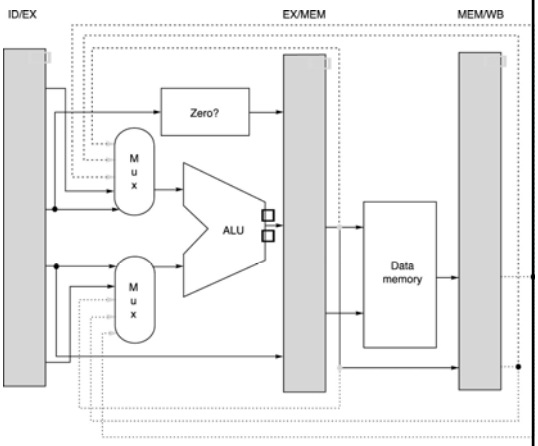
- One can stall the pipeline until the result has been calculated and stored (amended picture)



Copyright 1998-2004 © Leslie S. Smith

Solving data hazards II

- One can used additional hardware to alleviate the problem
  - add registers to make ALU outputs immediately available
  - don't wait until they have been written back to registers (or memory)
- Called *forwarding* (or bypassing, or short-circuiting)
- The ALU.output value
  - from the EX/MEM latch
- Or the loaded data
  - from the MEM/WB latch
- is made available at the ALU input registers
  - and is used instead of the register input if the CPU detects that the register has been updated
- This can completely remove the data hazard described above.



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

Data Hazard Classification

- Data hazards are of one of three different types: RAW (read after write)
  - j tries to read a source operand before i writes to it *(where i<j)*
  - this is the commonest form of data hazard
- WAW (write after write)
  - i tries to write a result after it is written by j
  - thus the wrong result gets left in the memory
- WAR (write after read)
  - j tries to write a value before it is read by i
  - this doesn't happen in the pipeline described here
  - but could occur if results were written early in the pipe
    - as might occur with autoincrement addressing modes.
- One cannot in general use forwarding to solve all of these problems
  - so that some data hazards do require pipeline stalls

Compiler scheduling for data hazards

- Typical straightforward generated code for simple statements like
  A = B+C
- causes stalls.

```
LW   R1, B          IF   ID   EX   MEM   WB
LW   R2, C               IF   ID   EX    MEM   WB
ADD  R3, R2, R1               IF   ID    stall  EX   MEM  WB
SW   A, R3                         IF    ID     stall  EX
```

- But compiler scheduling (code rearranging) can help.

Copyright 1998-2004 © Leslie S. Smith

Rearranging code

- E.g. a = b + c ; d = e + f ; can be rewritten as

```
LW   R1, B
LW   R2, C
LW   R3, E
ADD  R4, R1, R2
LW   R5, F
SW   A, R4
ADD  R6, R3, R5
SW   D, R6
```

- The ADDs and SWs have been rearranged so as to avoid pipeline stalls.

Copyright 1998-2004 © Leslie S. Smith