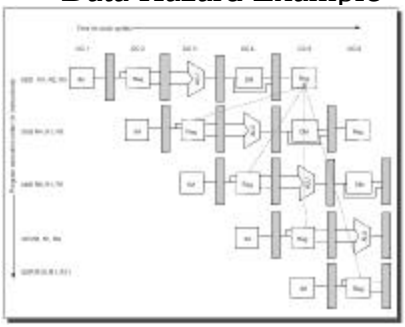
Pipelining Part 2

CS448

Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to operands that differs from the normal sequential order
- Example:
  - ADD **R1**, R2, R3
  - SUB R4, **R1**, R5
  - AND R6, **R1**, R7
  - OR R8, **R1**, R9
  - XOR R10, **R1**, R11
- Looks pretty innocent, what is the problem?

Data Hazard Example



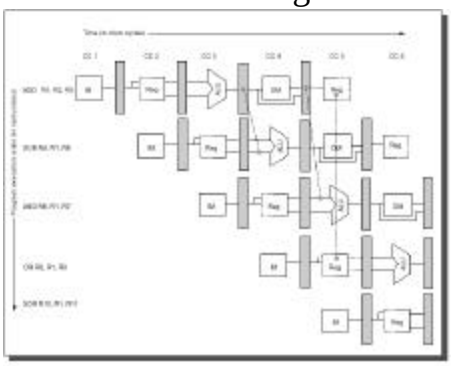
Split Cycle

Results of first ADD not available when the SUB needs it!
Any instructions correct?
Could be even worse with memory-based operands

Forwarding

- Technique to minimize data stalls as in the previous example
- Note we've actually computed the correct result needed by the other instructions, but it's in an earlier stage
  - ADD R1, R2, R3 R1 at ALUOutput
  - SUB R4, R1, R4 Need R1 at ALUInput
- Forward this data to subsequent stages where it may be needed
  - ALU result automatically fed back to input latch for next stage
  - Need control logic to detect if the feedback should be selected, or the normal input operands

Forwarding



5

Scoreboarding

- One way to implement the control needed for forwarding
- Scoreboard stores the state of the pipeline
  - What stage each instruction is in
  - Status of each destination register, source register
  - Can determine if there is a hazard and know which stage needs to be forwarded to what other stage
    - Controls via multiplexer selection
- If state of the pipeline is incomplete
  - Stalls and get pipeline bubbles

6

Another Data Hazard Example

- What are the hazards here?
  - ADD R1, R2, R3
  - LW R4, 0(R1)
  - SW 12(R1), R4
- Need forwarding to other stages than the same one

7

Data Hazard Classification

- Three types of data hazards
- Instruction i comes before instruction j
  - RAW : Read After Write
    - j tries to read a source before i writes it, so j incorrectly gets the old value. Solve via forwarding.
  - WAW : Write After Write
    - j tries to write an operand before it is written by i, so we end up writing values in the wrong order
    - Only occurs if we have writes in multiple stages
      - Not a problem with DLX integer instructions
      - We'll see this when we do floating point

8

Data Hazard Classification

- WAR : Write After Read
  - j tries to write a destination before it is read by i, so i incorrectly gets the new value
  - For this to happen we need a pipeline that writes results early in the pipeline, and then other instruction read a source later in the pipeline
  - Can this happen in DLX?
  - This problem led to a flaw in the VAX
- RAR : Read After Read
  - Is this a hazard?

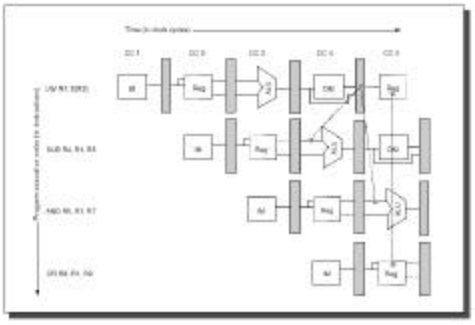
9

Forwarding is not Infallible

- Unfortunately, forwarding does not handle all cases, e.g.:
  - LW R1, 0(R2)
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
- Load of R1 not available until MEM, but we need it for the second instruction in ALU

10

Data Hazard Requiring Stall



Result needed before it is even computed!

11

Data Hazard Stall

- Need hardware (pipeline interlock) to detect the data hazard and introduce a vertical pipeline bubble
- Other stalls possible too
  - Cache miss, stall until data available

LW R1,(4R1)	IF	ID	EX	MEM	WB			
SUB R4,R1,R5		IF	ID	EX	MEM	WB		
AND R6,R1,R7			IF	ID	EX	MEM	WB	
OR R8,R1,R9				IF	ID	EX	MEM	WB

LW R1,(4R1)	IF	ID	EX	MEM	WB			
SUB R4,R1,R5		IF	ID	stall	EX	MEM	WB	
AND R6,R1,R7			IF	stall	ID	EX	MEM	WB
OR R8,R1,R9				stall	IF	ID	EX	MEM	WB

12

Compilers to the Rescue

- Compilers can help arrange instructions to avoid pipeline stalls, called Instruction Scheduling
- Compiler knows delay slots (the next instruction that may conflict with a load) for typical instruction types
  - Try to move other instructions into this slot that don't conflict
  - If one can't be found, insert a NOP
  - More formal methods to do this using dataflow graphs

13

Compiler Scheduling Example

- A=B+C; D =E+F
  - LW R1, B
  - LW R2, C
  - ADD R3, R1, R2 ← Need to stall for R2
  - SW A, R3
  - LW R4, E
  - LW R5, F
  - ADD R6, R4, R5 ← Need to stall for R5
  - SW D, R6

14

Compiler Scheduling Example

- A=B+C; D =E+F
  - LW R1, B
  - LW R2, C
  - LW R4, E ← Swap instr, no stall
  - ADD R3, R1, R2
  - LW R5, F
  - SW A, R3
  - ADD R6, R4, R5
  - SW D, R6

15

Compiler Scheduling a Big Help

- Percentage of loads causing stalls with DLX
  - TeX
    - Unscheduled 65%
    - Scheduled 25%
  - SPICE
    - Unscheduled 42%
    - Scheduled 14%
  - GCC
    - Unscheduled 54%
    - Scheduled 31%

16

Control Hazards

- Control hazards result when we branch to a new location in the program, invalidated everything we have loaded in our pipeline
  - Potentially a greater performance loss than data hazards
  - Simplest solution: Stall until we know the branch
    - Actually a three cycle stall, since we may need a new IF

Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		IF	stall	stall	IF	ID	EX	MEM	WB	
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

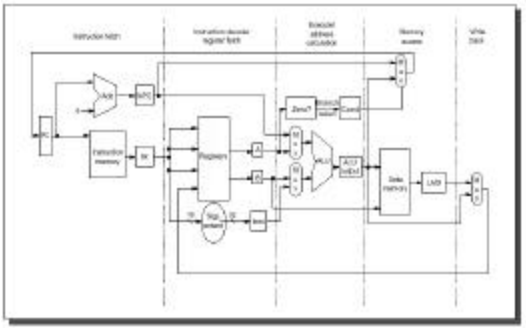
17

Control Hazards

- Big hit in performance – can reduce pipeline efficiency by over 1/2
- To reduce the clock cycles in a branch stall:
  - Find out whether the branch is taken or not taken earlier in the pipeline
    - Avoids longer stalls of everything else in the pipeline
  - Compute the taken PC earlier
    - Lets us fetch the next instruction with fewer stalls

18

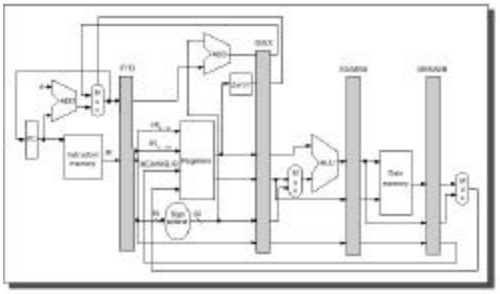
Original DLX Datapath



Branch not computed until EX stage

19

Revised DLX Datapath



Move branch logic to ID stage to reduce branch penalty
Downside – may make ID stage longer

20

Software-Based Branch Reduction Penalty

- Design ISA to reduce branch penalty
  - DLX's BNEZ, BEQZ, allows condition code to be known during the ID stage
- Branch Prediction
  - Compute likelihood of branching vs. not branching, automatically fetch the most likely target
  - Can be difficult; we need to know branch target in advance

21

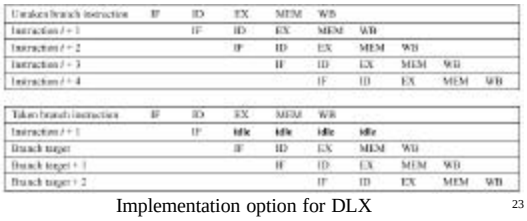
Branch Behavior

- How often are branches taken?
- For DLX from chapter 2:
  - 17% branches
  - 3% jumps or calls
- Taken vs. Not varies with instruction use
  - If-then statement taken about 50% of the time
  - Branches in loops taken 90% of the time
  - Flag test branches taken very rarely
- Overall, 67% of conditional branches taken on average
  - This is bad, because taking the branch results in the pipeline stall for our typical case where we are fetching subsequent instructions in the pipeline

22

Dealing with Branches

- Several options for dealing with branches
  1. Pipeline stall until branch target known (previous case we examined)
  2. Continue fetching as if we won't take the branch, but then invalidate the instructions if we do take the branch



Implementation option for DLX

23

Dealing with Branches

3. Always fetch the branch target
   - After all, most branches are taken
   - Can't do in DLX because we don't know the target in advance of the branch outcome
   - Other architectures could precompute the target before the outcome

24

Delayed Branch Option

4. Delayed Branch - Perform instruction scheduling into branch delay slots (instructions after a branch)
   - Always execute instructions following a branch regardless of whether or not we take it
   - Compilter will find some instructions we'll always execute, regardless of whether or not we take the branch, and put in there
   - Put a NOP if we can't find anything

25

Delayed Branch with One Delay Slot

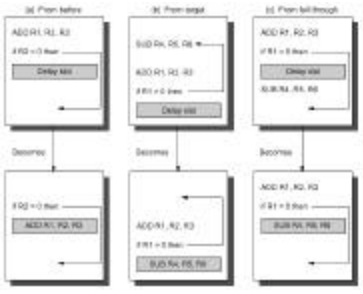
Untaken branch instruction	IF	ID	EX	MEM	WB			
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB		
Instruction i + 2			IF	ID	EX	MEM	WB	
Instruction i + 3				IF	ID	EX	MEM	WB
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB			
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB		
Branch target			IF	ID	EX	MEM	WB	
Branch target + 1				IF	ID	EX	MEM	WB
Branch target + 2					IF	ID	EX	MEM	WB

Instruction in delay slot always executed
Another branch instruction not allowed to be in the delay slot

26

Example: Delay Slot Scheduling



B) and C) execute code that may or may not be used, but better than a NOP

Form of branch prediction – compiler predicts based on context

27

Delay Slot Effectiveness

- Book – variations on scheme described here, branch nullifying if branch not taken
- On benchmarks
  - Delay slot allowed branch hazards to be hidden 70% of the time
  - About 20% of delay slots filled with NOPs
  - Delay slots we can't easily fill: when target is another branch
- Philosophically, delay slots good?
  - No longer hides the pipeline implementation from the programmers (although it will if through a compiler)
  - Does allow for compiler optimizations, other schemes don't
  - Not very effective with modern machines that have deep pipelines, too difficult to fill multiple delay slots

28

Performance of Branch Schemes

- We can simulate the four schemes on DLX
- Given CPI=1 as the ideal:
  - Pipeline Speedup = $\frac{Pipeline\_Depth}{1 + Branch\_Frequency \times Branch\_Penalty}$
  - Results: Delayed branch slightly better

Scheduling scheme	Branch penalty per conditional branch		Penalty per unconditional branch	Average branch penalty per branch		Effective CPI with branch stalls	
	Integer	FP		Integer	FP	Integer	FP
Stall pipeline	1.00	1.00	1.00	1.00	1.00	1.17	1.15
Predict taken	1.00	1.00	1.00	1.00	1.00	1.17	1.15
Predict not taken	0.62	0.70	1.0	0.69	0.74	1.12	1.11
Delayed branch	0.25	0.35	0.0	0.21	0.30	1.04	1.04

29