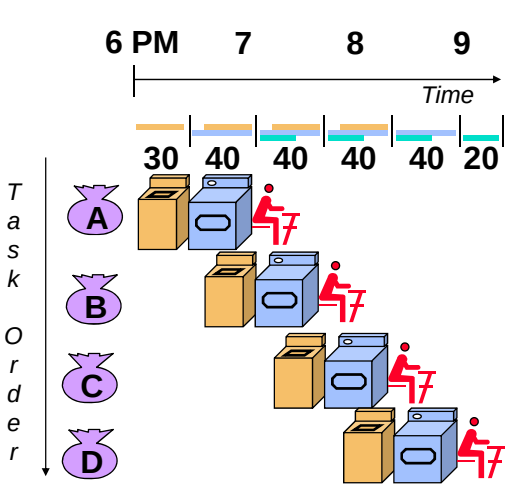
CS420/520
Computer Architecture I

Hazards in a Pipeline Processor
(CA4: Appendix A)

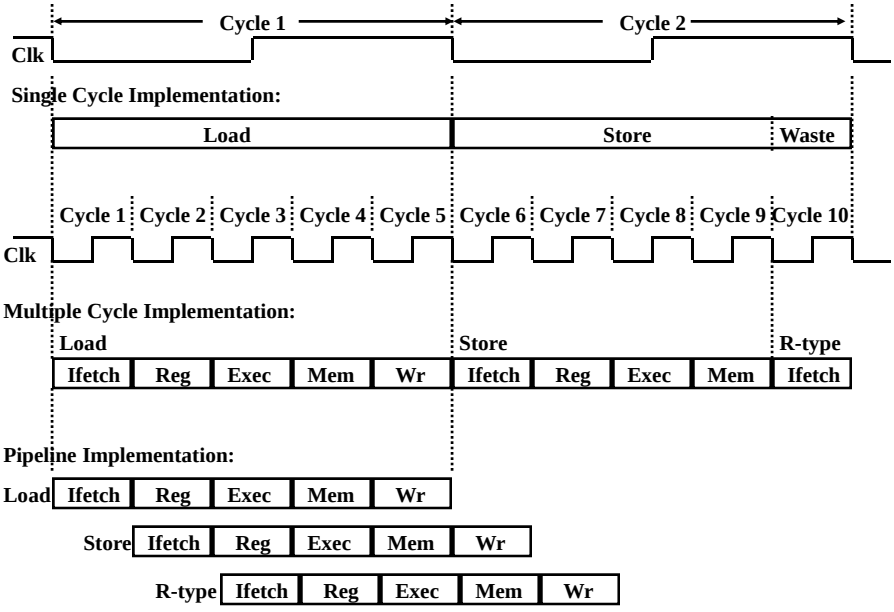
Dr. Xiaobo Zhou
Department of Computer Science

Review: Pipelining Lessons

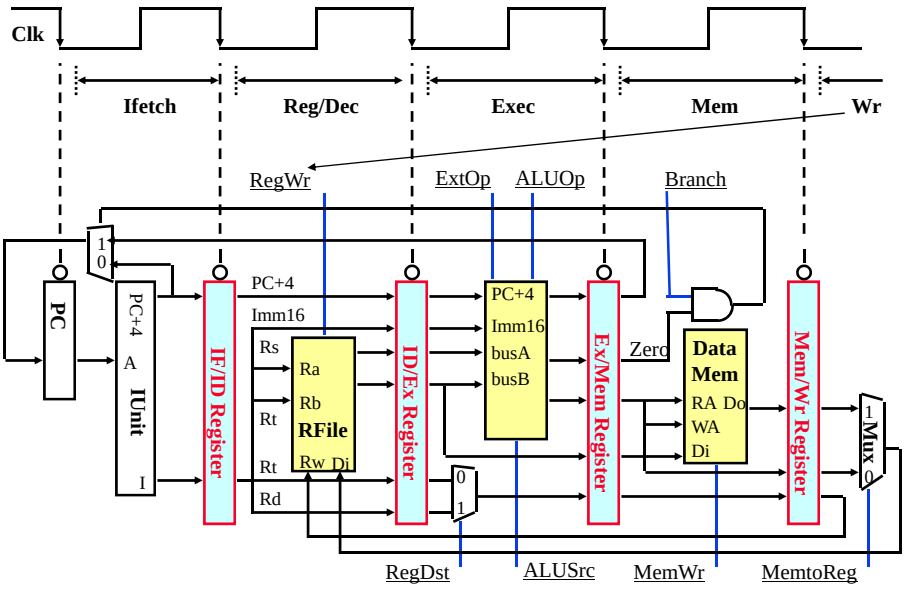


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

Review: Single Cycle, Multiple Cycle, vs. Pipeline

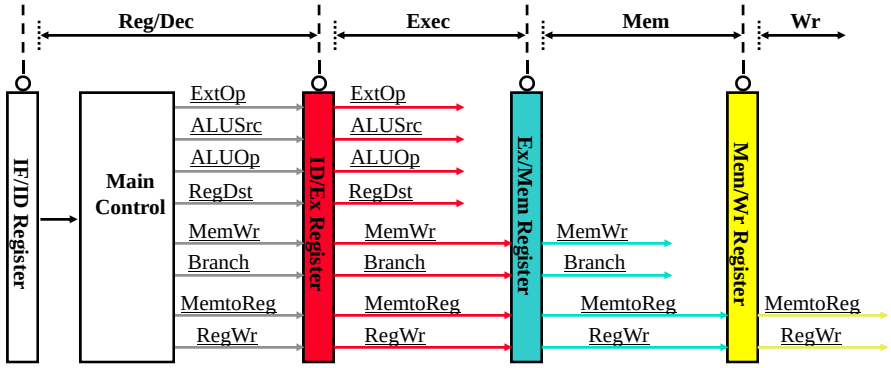


Review: A Pipelined Datapath



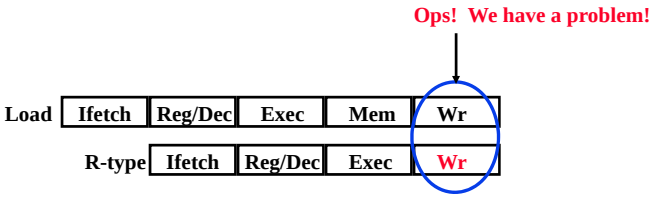
Review: Pipeline Control "Data Stationary Control"

° The Main Control generates the control signals during Reg/Dec
  • Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  • Control signals for Mem (MemWr Branch) are used 2 cycles later
  • Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Pipeline Summary

° Pipeline Processor:
  • Natural enhancement of the multiple clock cycle processor
  • Each functional unit can only be used once per instruction
  • If a instruction is going to use a functional unit:
    - it must use it at the same stage as all other instructions



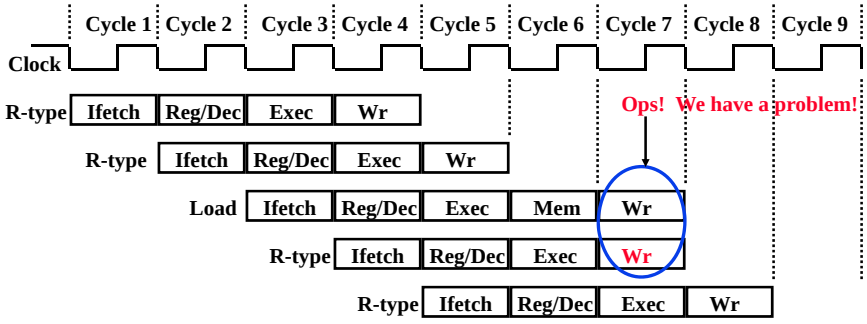
  • Pipeline Control:
    - Each stage's control signal depends ONLY on the instruction that is currently in that stage

Can Pipelining Get Us into Trouble?

° Yes: Pipeline Hazards
- structural hazards: attempt to use the same resource two different ways at the same time
  - E.g., combined washer/dryer would be a structural hazard
- control hazards: attempt to make a decision before condition is evaluated
  - branch instructions
- data hazards: attempt to use item before it is ready
  - E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
  - instruction depends on result of prior instruction still in the pipeline

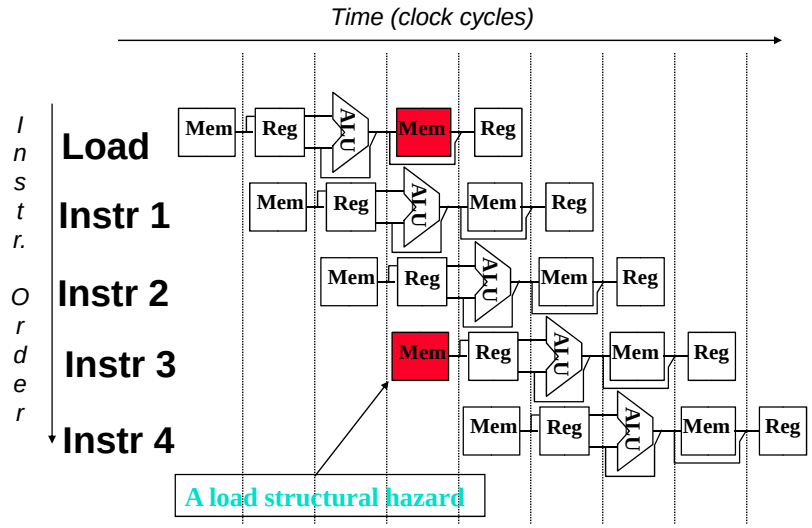
° Can always resolve hazards by waiting (stall)
- pipeline control must detect the hazard
- take action (or delay action) to resolve hazards

Pipelining the R-type and Load Instruction

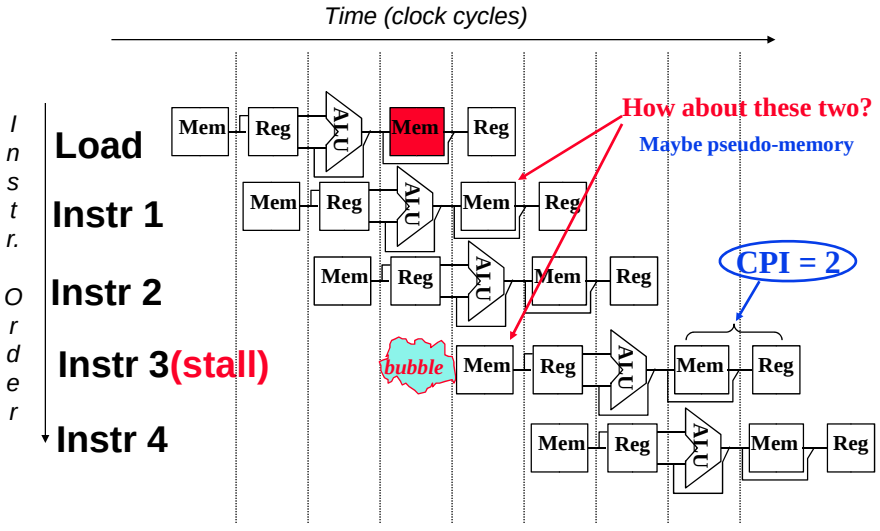


° We have a problem:
- Two instructions try to write to the register file at the same time!
- Only one write port -> a structural hazard
  - This one can be solved to have all instructions to have 5 stages

Single Memory is a Structural Hazard



Option 1: Stall to resolve Memory Structural Hazard



Load Structural Hazard Performance Impact

° Suppose: 1) memory data reference instructions constitute 40% of the instruction mix of a program.

2) ideal CPI (no hazards) is 1.

3) the processor with the structural hazard has a clock rate that is 1.05 times higher than the clock rate of the processor with out the structural hazard.

° Question: pipeline w/ or w/o the structural hazard, which faster? By how much?

Answer:

For pipeline w/ the structural hazard:

Ave. instruction exec. Time = average CPI * clock cycle time

(AIET) = (60% * 1 + 40% * 2) * CCT_ideal / 1.05

= 1.333 * CCT_ideal

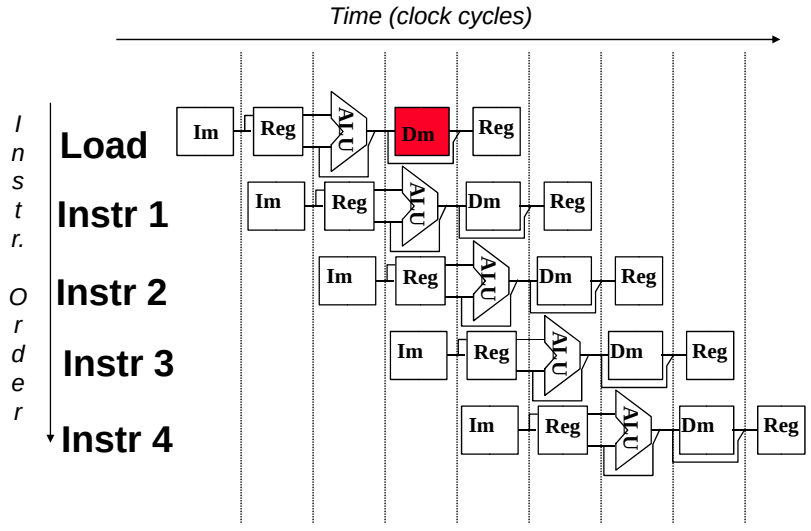
For pipeline w/o the structural hazard :

Ave. instruction exec. Time = 1 * CCT_ideal

Speedup = AIET_w/ hazard / AIET_w/o hazard = 1.333

Option 2: Duplicate to Resolve Structural Hazard

• Separate Instruction Cache (Im) & Data Cache (Dm)



Why Allowing Structural Hazards?

° A processor w/o structural hazards will always have a lower CPI, if other factors are equal, then why a designer allows structural hazards?

Answer:

Cost!

Duplication/separation of IC and DC:

a) costly itself

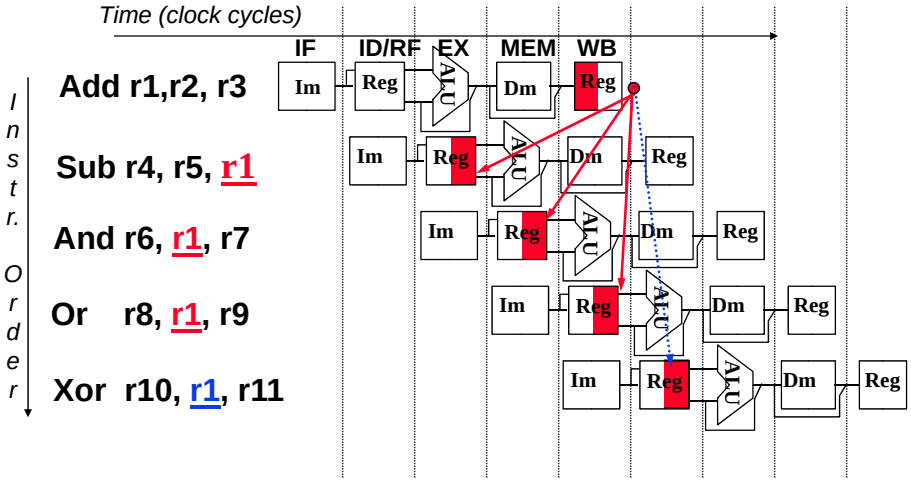
b) processor requires twice as much total memory bandwidth, if it needs to support IC and DC accesses in the same cycle.

Data Hazard on r1

```
add r1 ,r2, r3
sub r4, r1, r5
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1, r11
```

Data Hazard on r1:

- **Dependencies backwards in time are hazards**



Even Worse: Unpredictable Behavior!

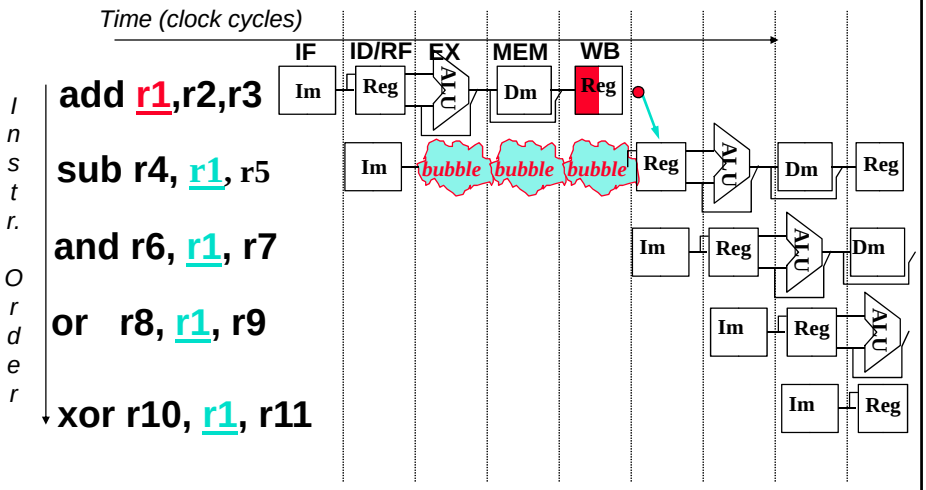
- **Interrupts – events other than branches and jumps that change the normal flow of instruction execution.**
  - **E.g., asynchronous I/O interrupts**
    - **I/O interrupt is not associated with any instruction**
    - **I/O interrupt does not prevent any instruction from completion**
      - pick your own convenient point to take an interrupt
- **If an interrupt occurs between Add and Sub instructions**
  - **The WB stage of the Add will complete**
  - **The value of R1 at that point for the subsequent Sub instruction will be the Right result of the Add.**
    - **Execution behavior is unpredictable!**

add r1,r2,r3

sub r4, r1, r5

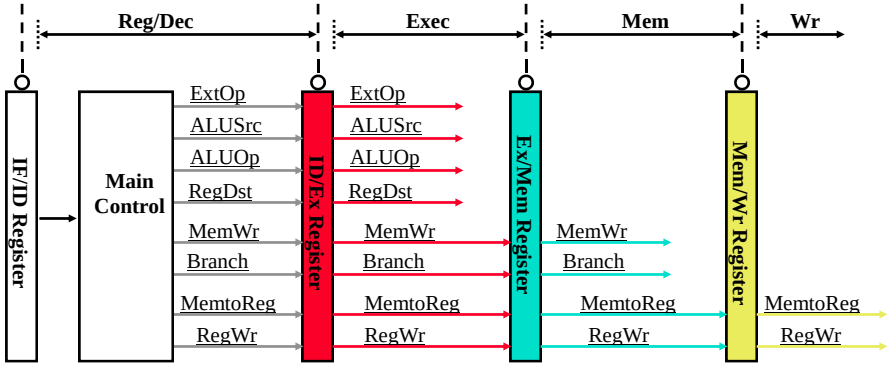
Option1: HW Stalls to Resolve Data Hazard

- **Dependencies backwards in time are hazards**



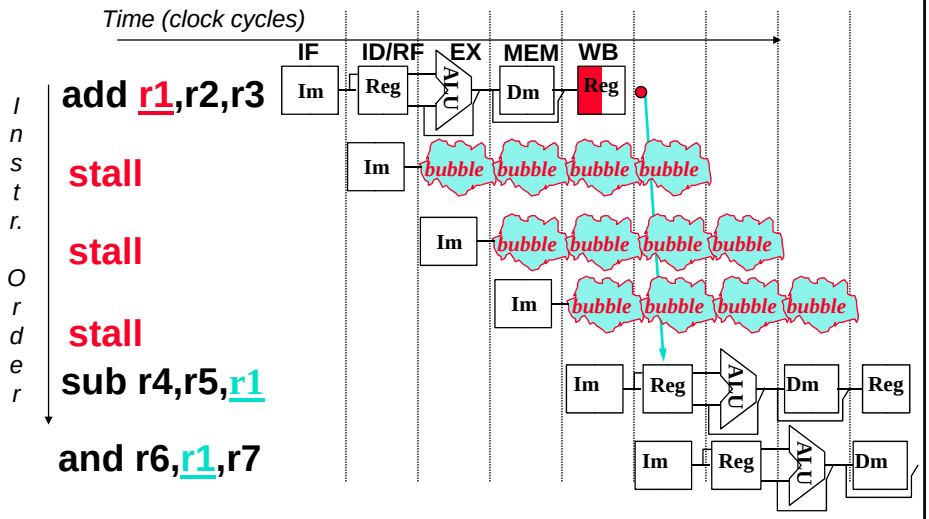
But recall use of "Data Stationary Control"

- **The Main Control generates the control signals during Reg/Dec**
  - **Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later**
  - **Control signals for Mem (MemWr Branch) are used 2 cycles later**
  - **Control signals for Wr (MemtoReg MemWr) are used 3 cycles later**



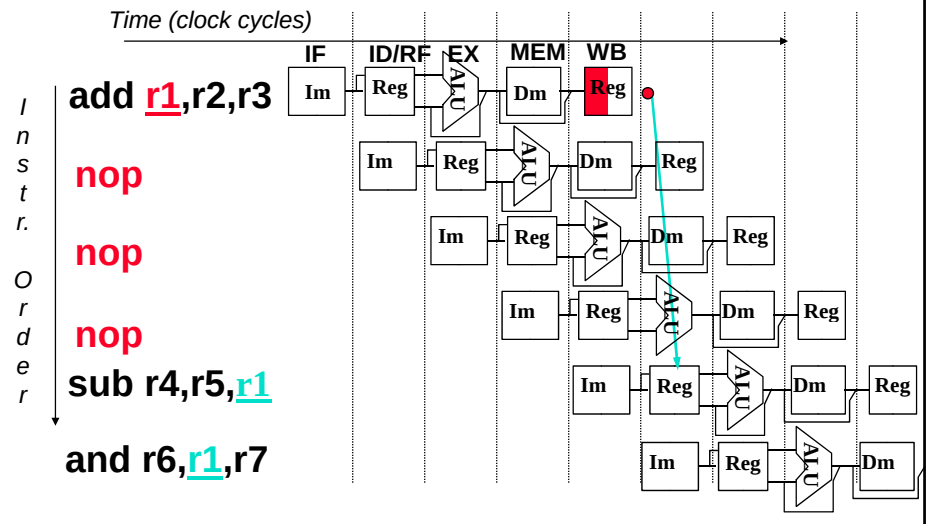
Option 1: How HW really stalls pipeline

- HW doesn't change PC => keeps fetching same instruction & sets control signals to to benign values (0)



Option 2: SW inserts independent instructions

- Worst case inserts NOP instructions



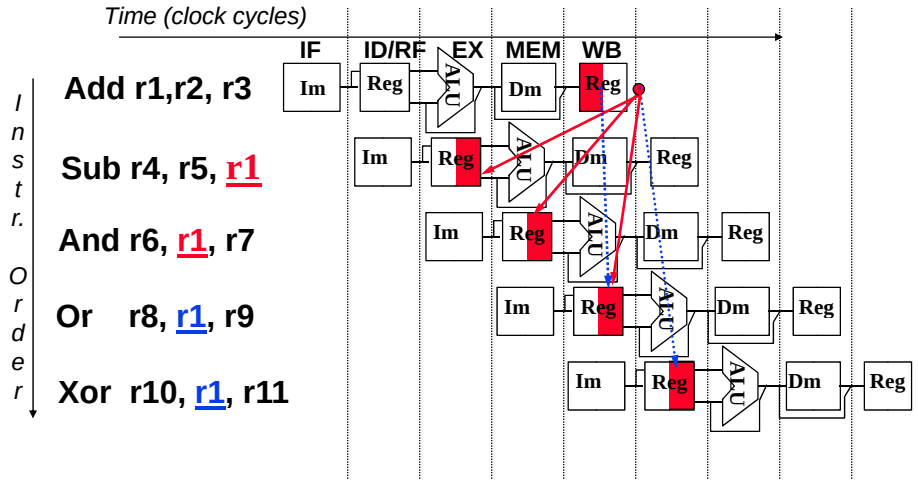
Simultaneous Readings and Writing

- The register file is used in two stages
  - One for reading in ID/RF
  - One for writing in WB
- We need to perform two reads and one write every clock cycle
- To handle reads and a write to the same register
  - performing register write in the first half of the clock cycle and the read in the second half (hardware implementation)

Advantage?

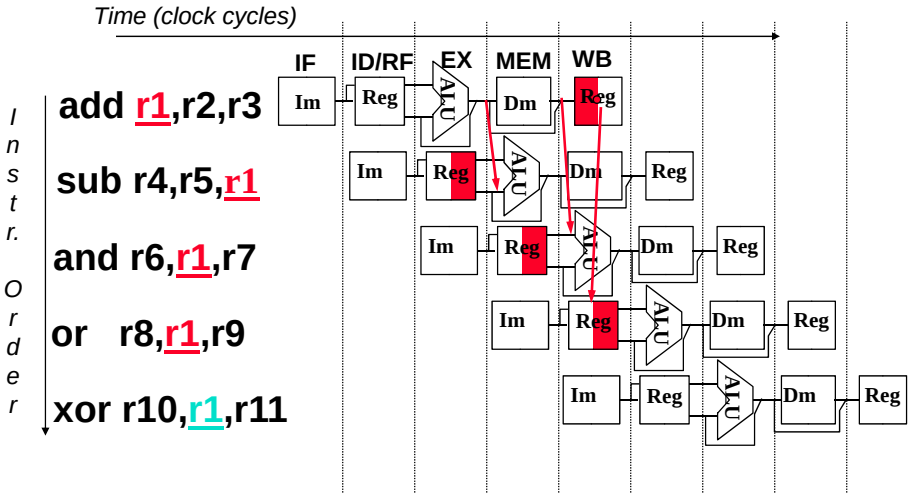
Advantage of Half-stage (Simultaneous) Writing

- Dependencies backwards in time are hazards



Option 3 Insight: Data is available!

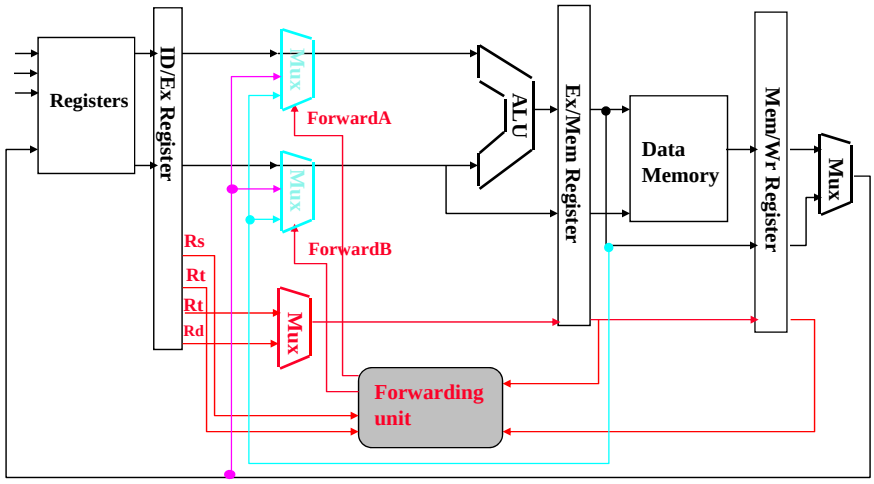
- **Pipeline registers already contain needed data**



HW Change I for "Forwarding" (Bypassing):

- **Increase multiplexors to add 2 paths from pipeline registers**
- **Assumes register read during write gets new value (write then read)**

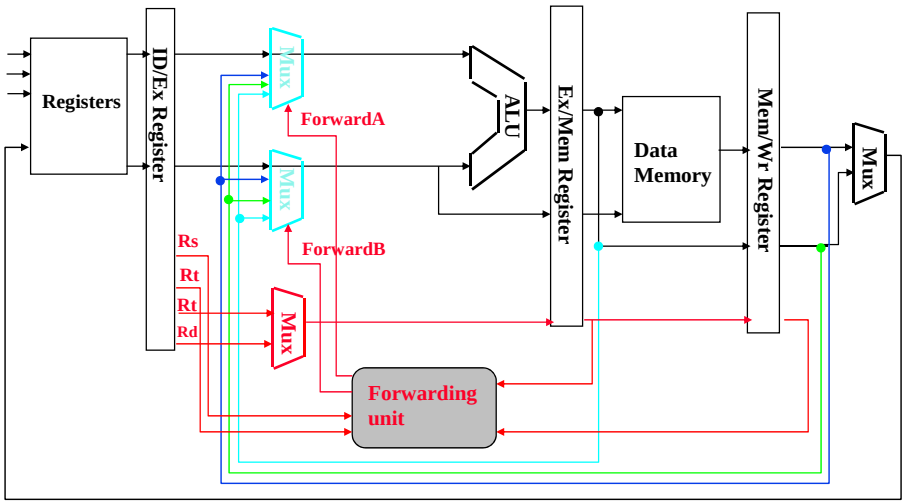
(otherwise more results to be forwarded)



HW Change II for "Forwarding" (Bypassing):

- Increase multiplexors to add 3 paths from pipeline registers
- Assumes register read during write gets new value (write then read)

(otherwise more results to be forwarded)



Bypassing (cont.)

- Two (or three) extra inputs on each ALU multiplexer and the addition of paths to the new inputs
  - the ALU output at the end of the EX (EXEC/MEM → EX)

```
add $1, $2, $3
add $4, $1, $1
```

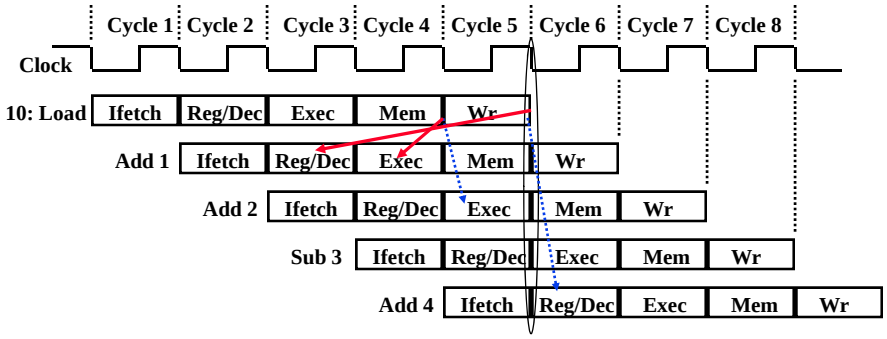
  - the ALU output at the end of the MEM stage (MEM/WB → EX)

```
add $1, $2, $3
add $5, $6, $7
lw  $4, $1 (100)
```

  - the memory output at the end of the MEM stage (MEM/WB → EX)

```
lw $1, $2 (100)
add $5, $6, $7
add $2, $1, $1
```

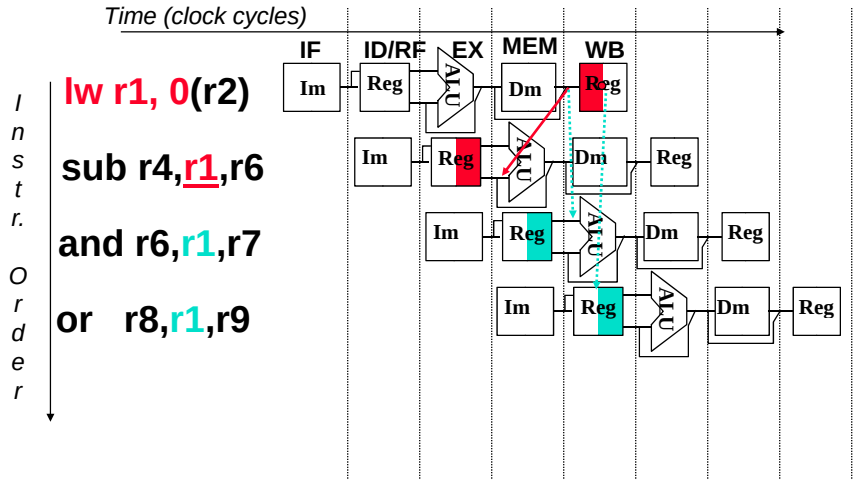
The Delay Load Phenomenon



° Although Load is fetched during Cycle 1:
- The data is NOT written into the Reg File until the end of Cycle 5
- We cannot read this value from the Reg File until Cycle 6
- 3-instruction delay before the load take effect if no bypassing

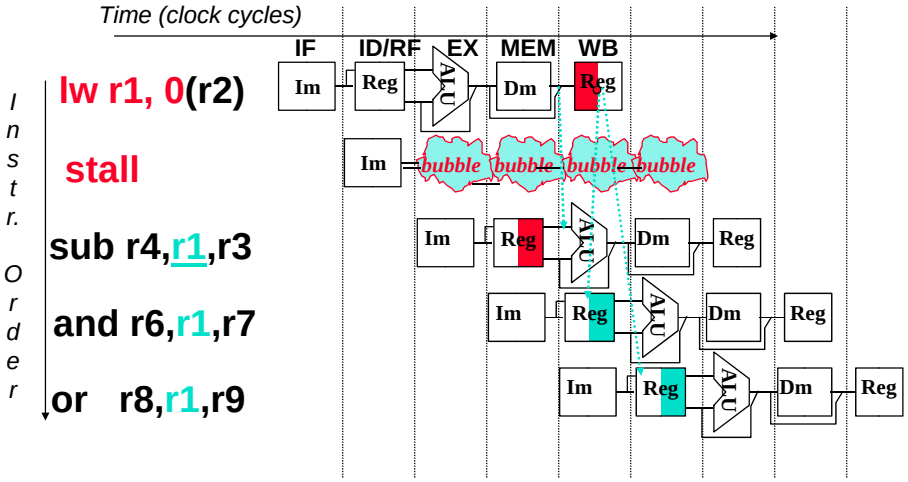
° This is referred to as Delay Load:
- Clever design techniques can reduce the delay to ONE instruction

Forwarding reduces Data Hazard to 1 cycle:



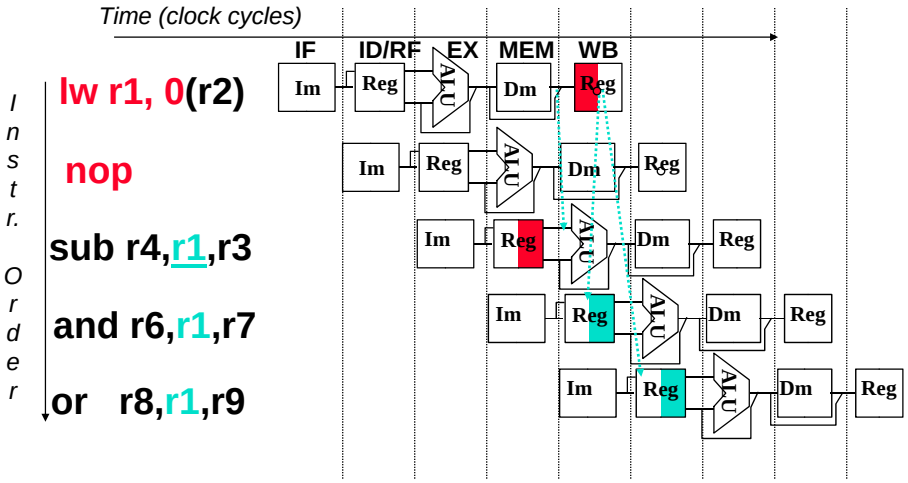
Option2: HW Stalls to Resolve Data Hazard

- "pipeline interlock": checks for hazard & stalls



Option 3: SW Inserts Independent Instructions

- Worst case inserts NOP instructions
- MIPS I solution: No HW checking



Option 4: Software/Compiler Scheduling / ILP

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d, e, and f in memory.

Slow code:

```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```


Option 5: Hardware/Dynamic Scheduling / ILP

- Static/compiler pipeline scheduling by the compiler tries to minimize stalls by separating dependent instructions so that they will not lead to hazards
- Dynamic hardware scheduling tries to avoid stalls when dependences, which could generate hazards, are present.

Pipeline Data Hazard Detection (Delay Load)

Situation	Example code sequence	Action
No dependence	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD **R1**,45(R2) DADD R5,**R1**,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,**R1**,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,**R1**,R7	No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

° **Comparing the destination and sources of adjacent instructions**

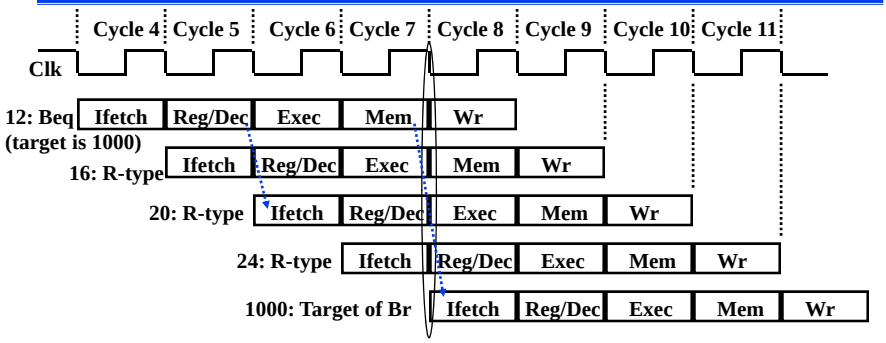
Pipeline Interlock Control (for Delay Load)

Opcode of ID/EX (ID/EX.IR 0…5)	Opcode field of IF/ID (IF/ID.IR 0…5)	Matching operand fields
Load	**Reg-Reg ALU**	**ID/EX.IR[rt] == IF/ID.IR[rs]**
Load	**Reg-Reg ALU**	**ID/EX.IR[rt] == IF/ID.IR[rt]**
Load	**Load, Store, ALU imme, branch**	**ID/EX.IR[rt] == IF/ID.IR[rs]**

° **The logic to detect the need for load interlock during the ID stage of an instruction requires three/two comparisons**

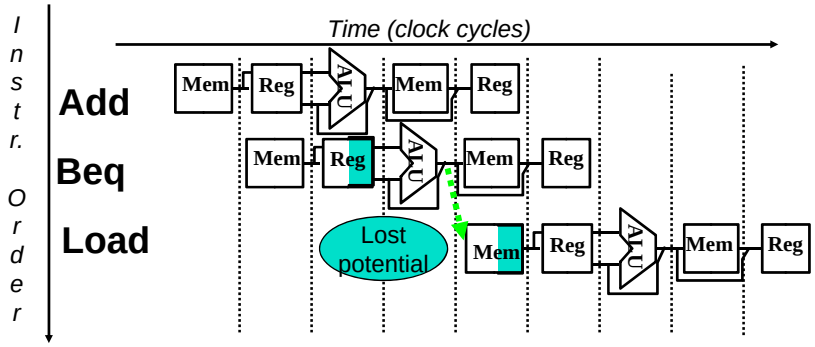
- **Why three: is R-type 'rs' in the same bits position of the instruction as that of I-type 'rs'? Though in MIPS, they are!**

From Last Lecture: The Delay Branch Phenomenon



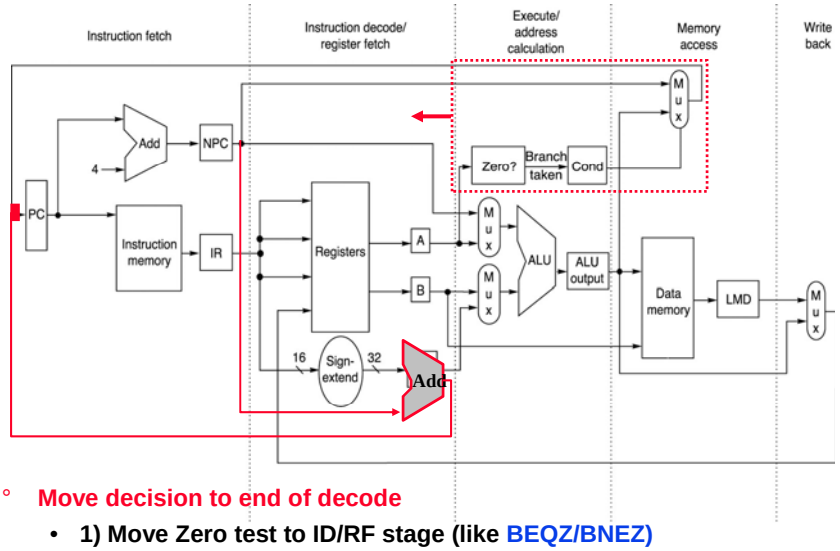
- Although Beq (4 cycle vs. 3 cycle BEQZ/BNEZ) is fetched in Cycle 4:
  - Target address is NOT written into the PC until the end of Cycle 7
  - Branch's target is NOT fetched until Cycle 8
  - 3-instruction delay before the branch take effect
- This is referred to as Control Hazard (greater loss than data hazards):
  - make a deci. based on result of an instr. whi. others are executing
  - Clever design techniques can reduce the delay to ONE instruction

General Control Hazard Solution: Stall



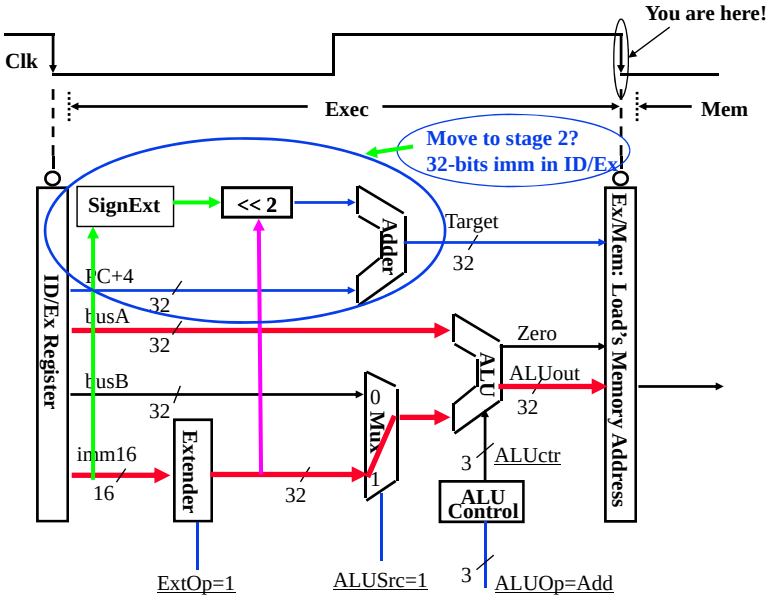
- Stall: wait until decision is clear
- Impact: 3 or 2 lost cycles (i.e., if ZERO detection happens and PC updates in the end of stage 3 in branch instruction) => slow
- Move decision to end of decode
  - Move Zero test to ID/RF stage (like BEQZ/BNEZ)
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch vs. 3

Recall: An Abstracted Multiple Cycle Datapath

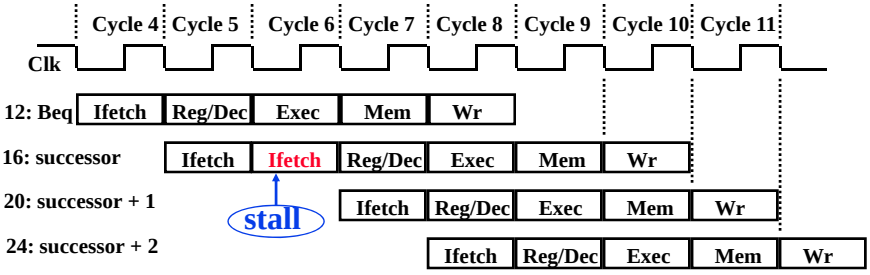


- Move decision to end of decode
  - 1) Move Zero test to ID/RF stage (like BEQZ/BNEZ)
  - 2) Adder to calculate new PC in ID/RF stage; 1 cycle penalty vs. 3

Recap: A View of the Pipeline Execution Unit



Taken Branch vs. Not-Taken Branch



How this *stall* can be implemented by "control"?

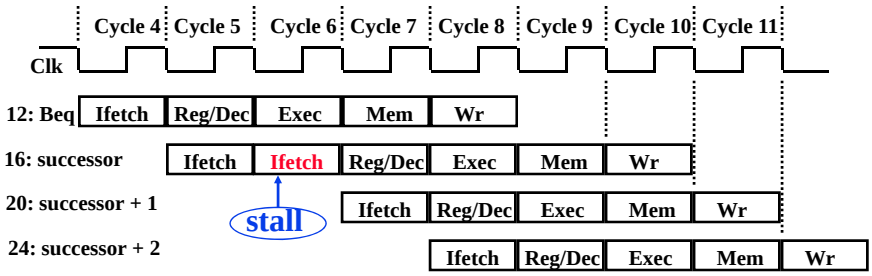
- ***Taken*** **branch: If a branch changes the PC to its target address**
- ***Not-Taken (untaken)*** **branch: If a branch sequentially falls through**
- **If the branch above is not taken, the second IF for branch successor is redundant**
  - **How to take the advantage since the right instruction was indeed fetched?**

Reducing Pipeline Branch Penalties:

- **Four simple compile-time schemes**
  - **STATIC: fixed for each branch during the entire execution; software try to minimize the branch penalty by using knowledge of the hardware and of branch behavior**
- **More powerful HW and SW techniques for both static and dynamic branch prediction**
  - **Instruction Level Parallelism (ILP)**

Technique 1: freezing/flushing

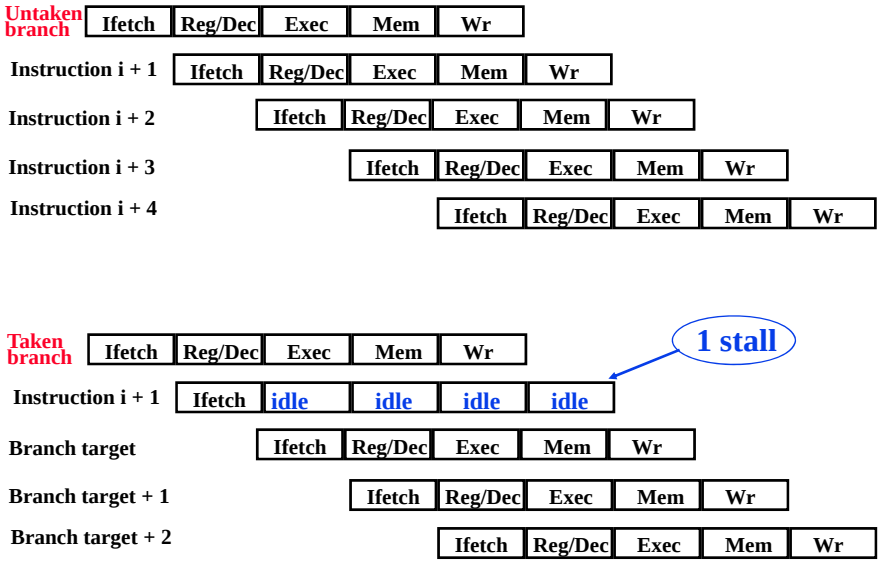
° **Freezing or flushing**
- **Holding or deleting any instructions after the branch until the branch decision & destination is known**
  - **Simplicity for both HW and SW**
    – HW doesn't change PC => keeps fetching same instruction & sets control signals to benign values (0)
  - **Early solutions (fixed penalty, CPI 2 for branch instructions)**



Technique 2: not-taken branch

° **Single Direction Prediction; not-taken branch**
- **Treat every branch as not taken**
- **Allowing HW to continue as if the branch were not executed**
  - **simplifies the instruction fetch**
  - **Older pipelined processors, e.g., Intel i486**
- **What if branch is taken?**
  - **For integer benchmarks, branches are taken about 60%**
  - **Turn the fetched instruction into a *noop*, and restart IF at the target address**
    - **no "damage" has been done yet to Registers & Memory !**
  - **For integer benchmarks, branches are taken about 60%**

Techniques 2 (Predicted-not-taken) Diagram



Technique 3: taken branch

- Single Direction Prediction; taken branch
  - Treat every branch as taken -- a higher prediction accuracy (60%)
  - Question 1: does it makes sense in our five-stage pipelining?
    - Does not know the target address any earlier than we know branch outcome
  - Question 2: when does it makes sense?
    - In some processors, branch target address is available before the branch outcome
    - But often more HW to calculate the branch address earlier
- Better prediction accuracy: Backwards Taken/Forwards Not Taken
  - Majority of backwards branches are loop branches, which usually iterate many times before exiting; branches are likely taken
  - Pros: no ISA modification since the sign of target displacement is encoded in the branch instruction.
  - Example: HP PA-RISC2.0 ISA

Technique 4: delayed branch

° Execution cycle

branch instruction

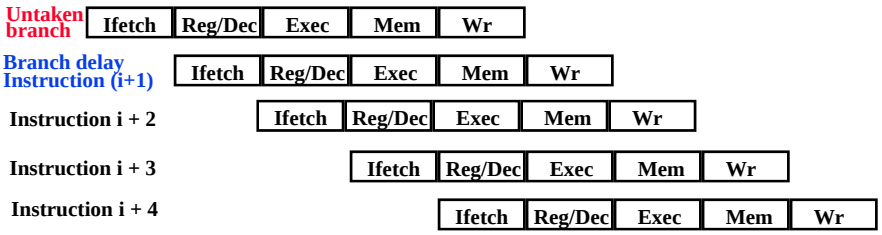
sequential successor 1

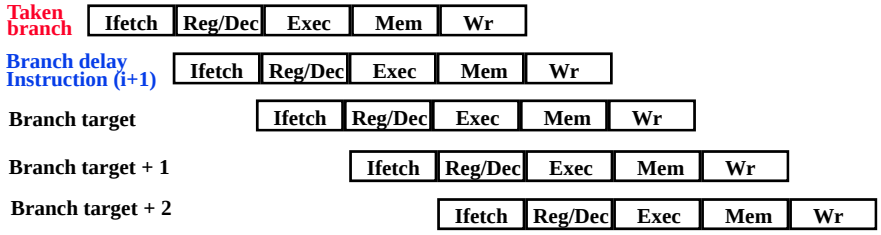
branch target if taken or sequential successor 2 if not taken

- A HW component: branch delay slot (1 for MIPS)
  - Instruction inside is executed whether branch is taken or not
  - Thus, what is the job for the compiler?

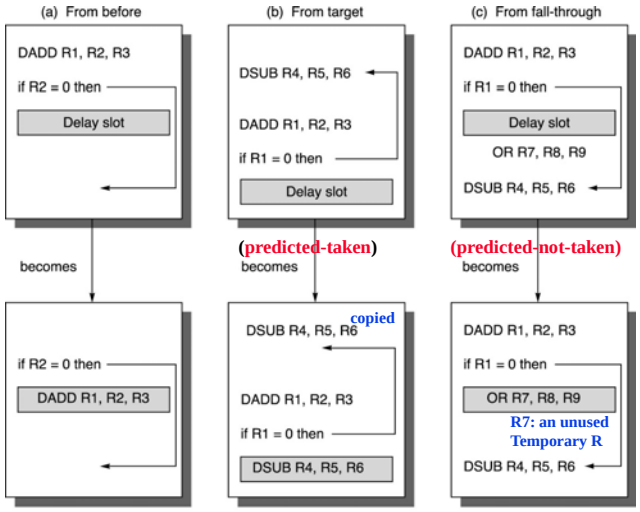
–Make the successor instruction valid and useful!

Technique 4 Example





Scheduling the Branch Delay Slot (BDS)



Make the successor instruction valid and useful! Worst case:

It must be OK to execute BDS when the branch goes in the unexpected direction.

Summary of Pipelining Hazards

° **Speed Up and Pipeline Depth; if ideal CPI is 1, then:**

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

° **What makes it easy**
- **all instructions are the same length**
- **just a few instruction formats**
- **memory operands appear only in loads and stores**

° **What makes it hard: hazards limit performance on computers:**
- **structural: need more HW resources**
- **data: need forwarding (& simultaneous write), compiler scheduling**
- **control: early evaluation & PC, delayed branch, prediction**

° **Compilers key to reducing cost of data and control hazards**

° **More reading:**
- **CA 5: Appendix C.2-C.4**
- **CO 4: Chapter 4.5-4.8**