EE 357 Unit 18

Basic Pipelining Techniques

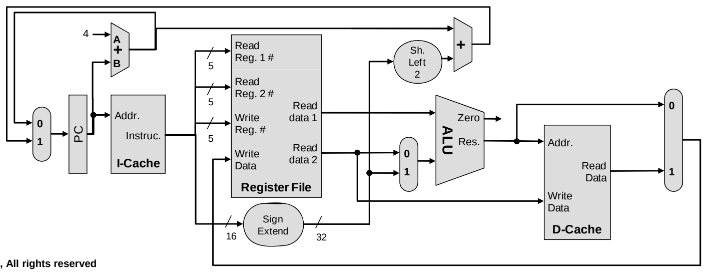
Single & Multi-Cycle Performance

Single-Cycle CPU

- Each piece of the datapath requires only a small period of the overall instruction execution (clock cycle) time yielding low utilization of the HW's actual capabilities

Multi-Cycle CPU

- Sharing resources allows for compact logic design but in modern design we can afford replicated structures if needed
- Each instruction still requires several cycles to complete



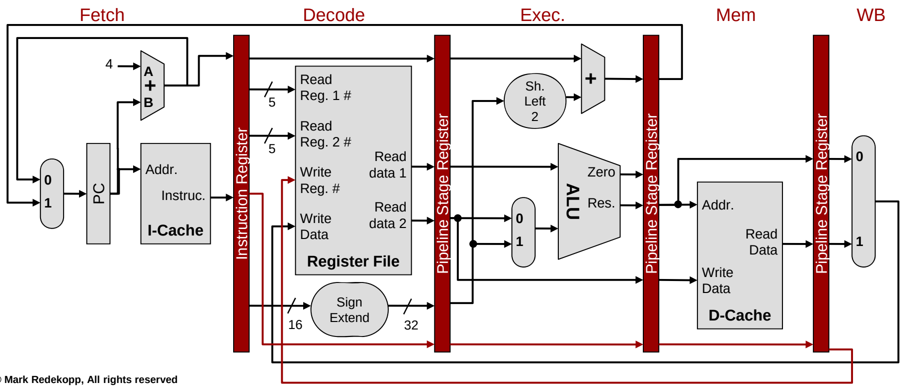
Pipelining

- Combines elements of both designs
  - Datapath of ______________ CPU w/ separate resources
  - Datapath broken into _________ with temporary registers between stages
    - _________ clock cycle
    - A single instruction requires CPI = n
- System can achieve CPI = __________
  - Overlapping Multiple Instructions (separate instruction in each stage at once)

	F	D	Ex	Mem	WB
Clock 1	Inst. 1				
Clock 2	Inst. 2	Inst. 1			
Clock 3	Inst. 3	Inst. 2	Inst. 1		
Clock 4	Inst. 4	Inst. 3	Inst. 2	Inst. 1	
Clock 5	Inst. 5	Inst. 4	Inst. 3	Inst. 2	Inst. 1

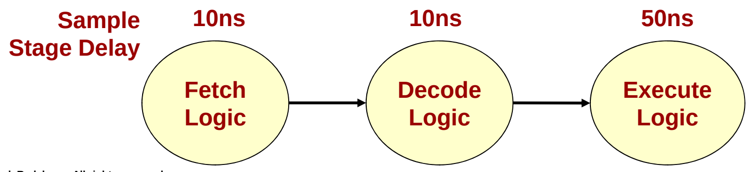
Basic 5 Stage Pipeline

- Same structure as single cycle but now broken into 5 stages
- Pipeline stage registers act as temp. registers storing intermediate results and thus allowing previous stage to be reused for another instruction
  - Also, act as a barrier from signals from different stages intermixing



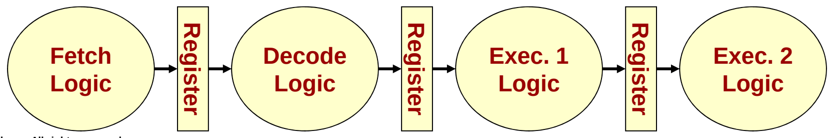
Issues with Pipelining

- ____________ of HW/logic resources between stages because of full utilization
  - Can't have a single cache (both I & D) because each is needed to fetch one instruction while another accesses data]
- Prevent signals in one stage (instruc.) from ___________ another stage (instruc.) and becoming convoluted
- Balancing stage delay
  - Clock period = ____________
  - In example below, clock period = _____ means _____ delay for only ______ of logic delay



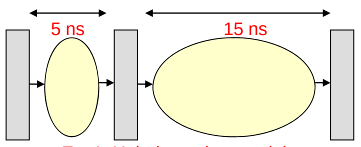
Resolution of Pipelining Issues

- No sharing of HW/logic resources between stages
  - For full performance, no feedback (stage i feeding back to stage i-k)
  - If two stages need a HW resource, _________ the resource in both stages (e.g. an I- AND D-cache)
- Prevent signals from one stage (instruc.) from flowing into another stage (instruc.) and becoming convoluted
  - Stage Registers act as _________ to signals until next edge
- **Balancing stage delay [Important!!!]**
  - Balance or divide long stages (See next slides)

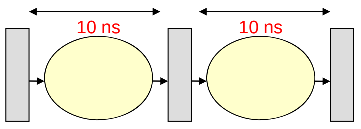


Balancing Pipeline Stages

- Clock period must equal the *LONGEST* delay from register to register
  - In Example 1, clock period would have to be set to ____ [66 MHz], meaning total time through pipeline = 30ns for only ____ ns of logic
- Could try to balance delay in each stage
  - Example 2: Clock period = __ns [100 MHz], while total time through pipeline is still = 20ns



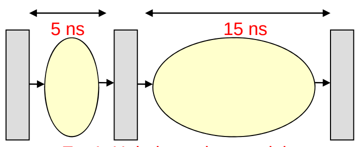
Ex. 1: Unbalanced stage delay
Clock Period = 15ns



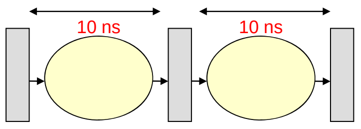
Ex. 2: Balanced stage delay
Clock Period = 10ns (150% speedup)

Pipelining Effects on Clock Period

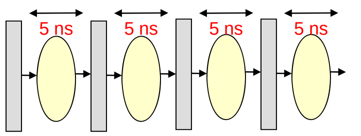
- Rather than just try to balance delay we could consider making more stages
  - Divide long stage into multiple stages
  - In Example 3, clock period could be 5ns [_____ MHz]
  - Time through the pipeline (latency) is still 20 ns, but we've increased our __________ (1 result every 5 ns rather than every 10 or 15 ns)
  - Note: There is a small time overhead to adding a pipeline register/stage (i.e. can't go crazy adding stages)



Ex. 1: Unbalanced stage delay
Clock Period = 15ns



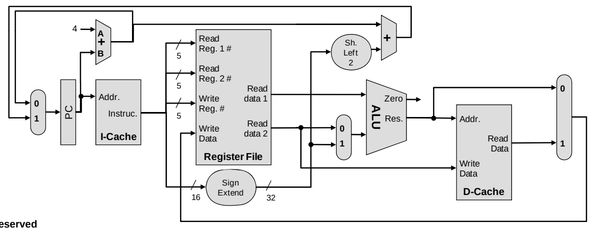
Ex. 2: Balanced stage delay
Clock Period = 10ns (150% speedup)



Ex. 3: Break long stage into multiple stages
Clock period = 5 ns (_____ speedup)

Feed-Forward Issues

- CISC instructions often perform several ALU and memory operations per instructions
  - MOVE.W (A0)+,$8(A0,D1) [M68000/Coldfire ISA]
    - 3 Adds (post-increment, disp., index)
    - 3 Memory operations (I-Fetch + 1 read + 1 write)
  - This makes pipelining hard because of multiple uses of ALU and memory
- Redesign the Instruction Set Architecture to better support pipelining (MIPS was designed with pipelining in mind)

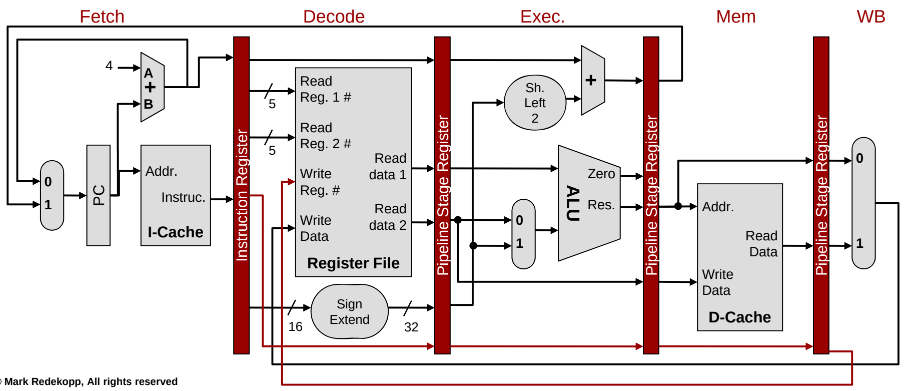


Sample 5-Stage Pipeline

- Examine the basic operations that need to be performed by our instruction classes
  - LW: I-Fetch, Decode/Reg. Fetch, Address Calc., Read Mem., Write to Register
  - SW: I-Fetch, Decode/Reg. Fetch, Address Calc., Write Mem.
  - ALUop: I-Fetch, Decode/Reg. Fetch, ALUop, Write to Reg.
  - Bxx: I-Fetch, Decode/Reg. Fetch, Compare (Subtract), Update PC
- These suggest a 5-stage pipeline:
  - I-Fetch,
  - Decode/Reg. Fetch,
  - ALU (Exec.),
  - Memory,
  - Reg. Writeback

Basic 5 Stage Pipeline

- All control signals needed for an instruction in the following stages are generated in the decode stage and ______________________
  - Since writeback doesn't occur until final stage, write register # is shipped with the instruction through the pipeline and then used at the end
  - Register File can read out the current data being written if read reg # = write reg #

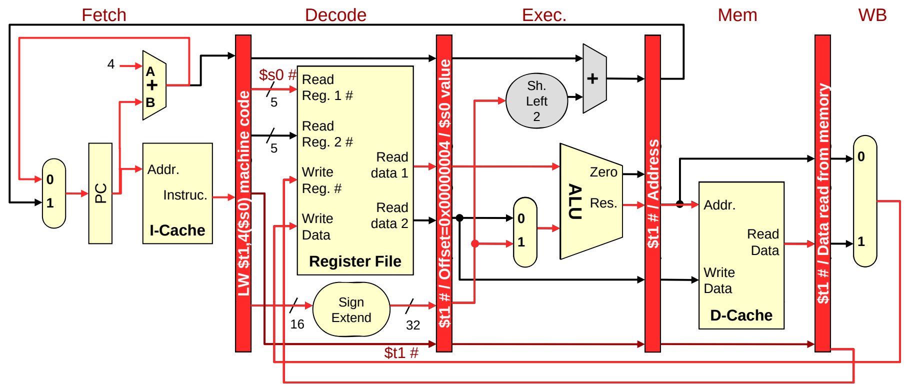


Sample Instructions

Instruction
LW $t1,4($s0)
ADD $t4,$t5,$t6
BEQ $a0,$a1,LOOP

For now let's assume we just execute one at a time though that's not how a pipeline works (multiple instructions are executed at one time).

LW $t1,4($s0)



Fetch LW and increment PC

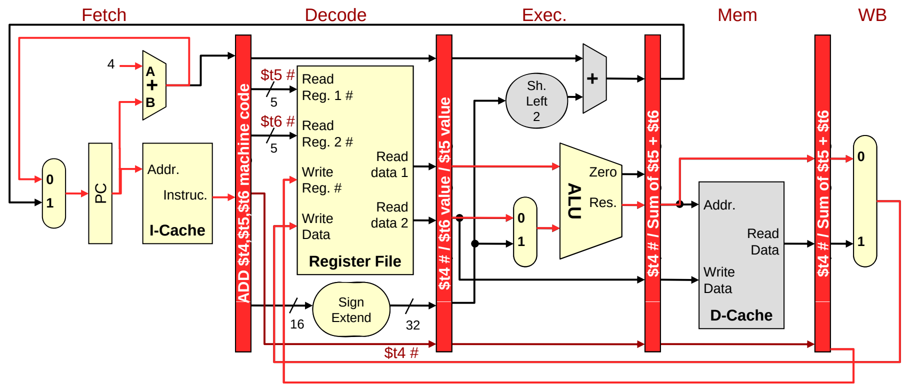
Decode instruction and fetch operands

Add offset 4 to $s0 value

Read word from memory

Write word to $t1

ADD $t4,$t5,$t6



Fetch ADD and increment PC

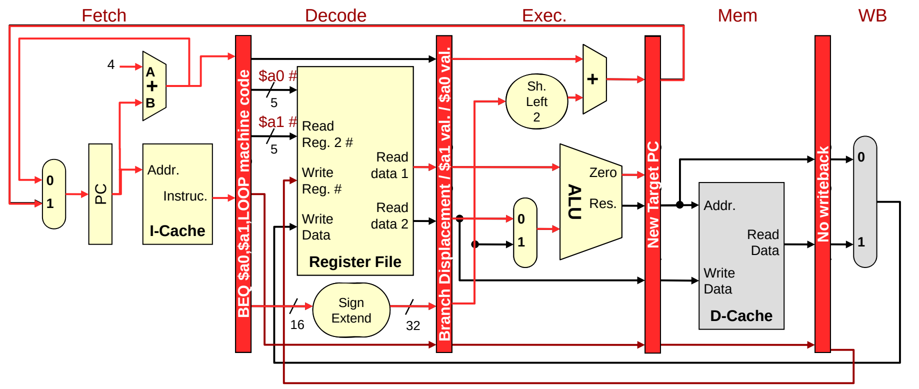
Decode instruction and fetch operands

Add $t5 + $t6

Just pass sum through

Write sum to $t4

BEQ $a0,$a1,LOOP



Fetch BEQ, increment PC, pass on PC+4

Decode instruction and fetch operands, pass on PC+4

Do $a0-$a1 and check if result = 0 Calculate branch target address

Update PC, No Mem. Access

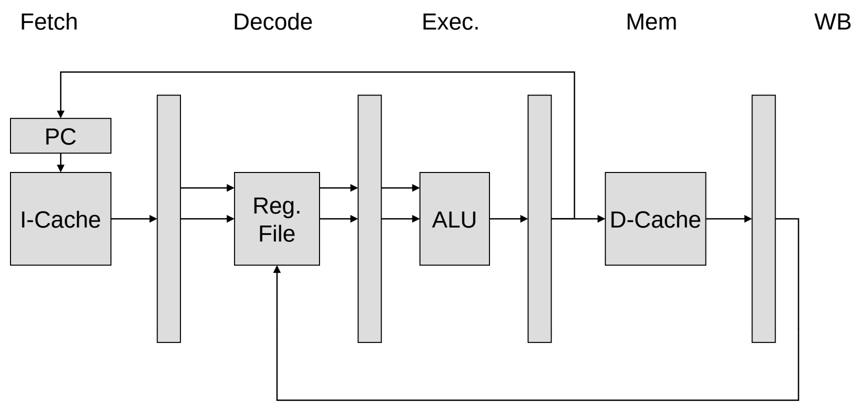
Do Nothing

Pipelining

- Now let's see how all three can be run in the pipeline

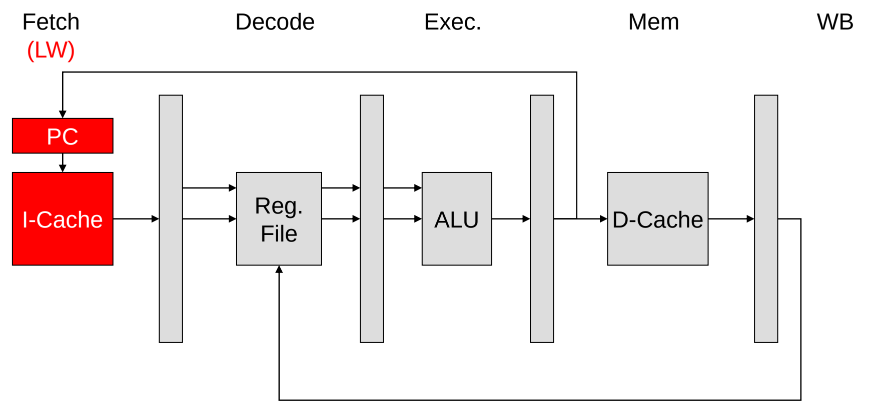
USC Viterbi School of Engineering

5-Stage Pipeline



USC Viterbi School of Engineering

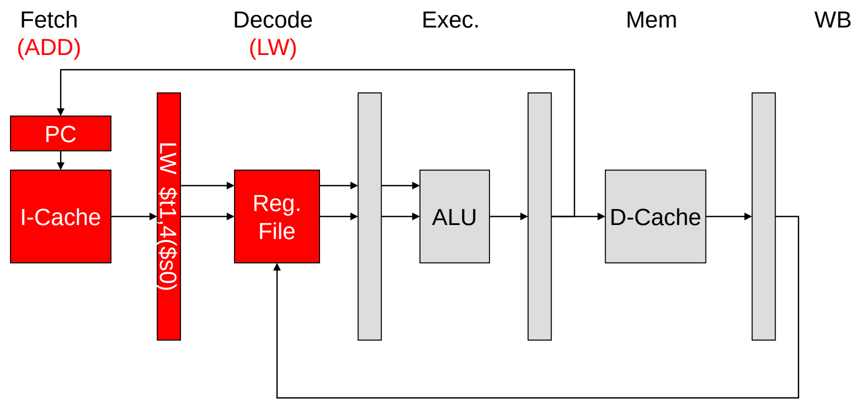
Example



Fetch LW

USC Viterbi School of Engineering

Example

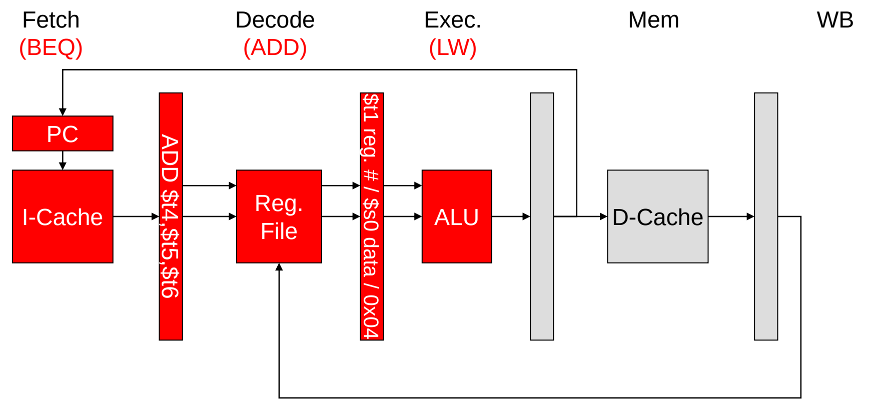


Fetch ADD

Decode instruction and fetch operands

USC Viterbi School of Engineering

Example

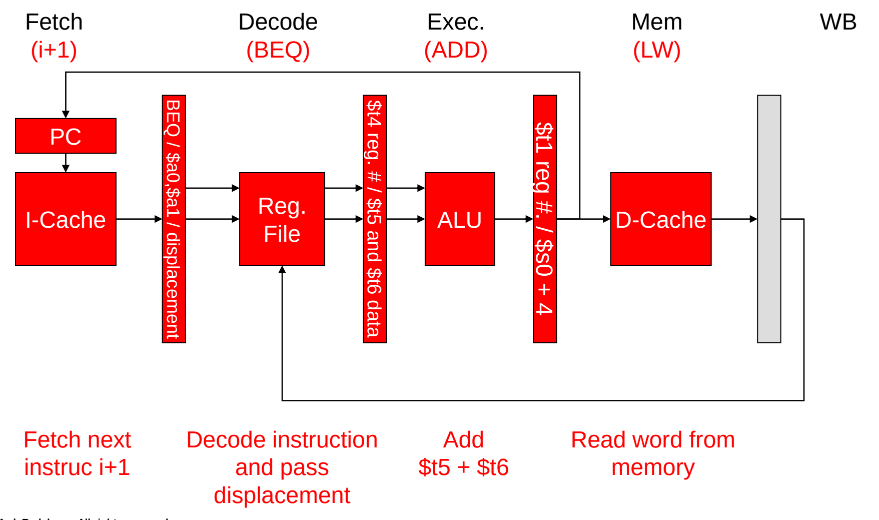


Fetch BEQ

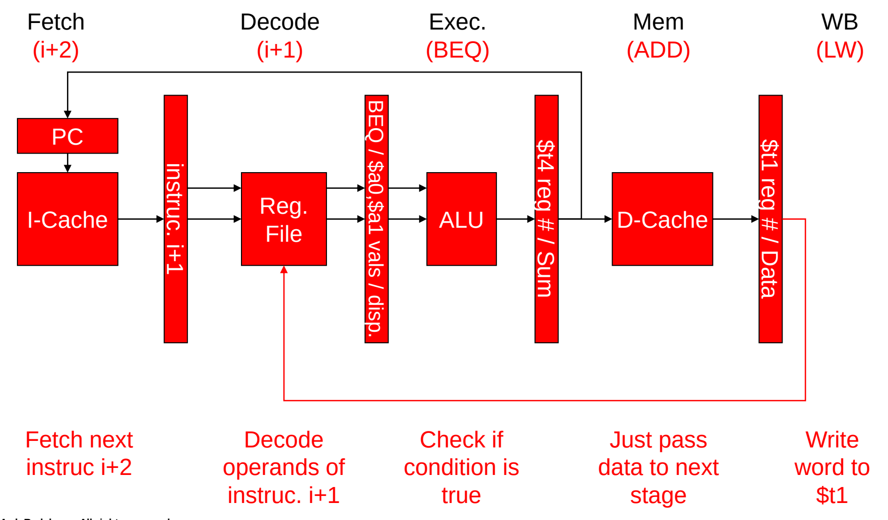
Decode instruction and fetch operands

Add displacement 0x04 to $s0

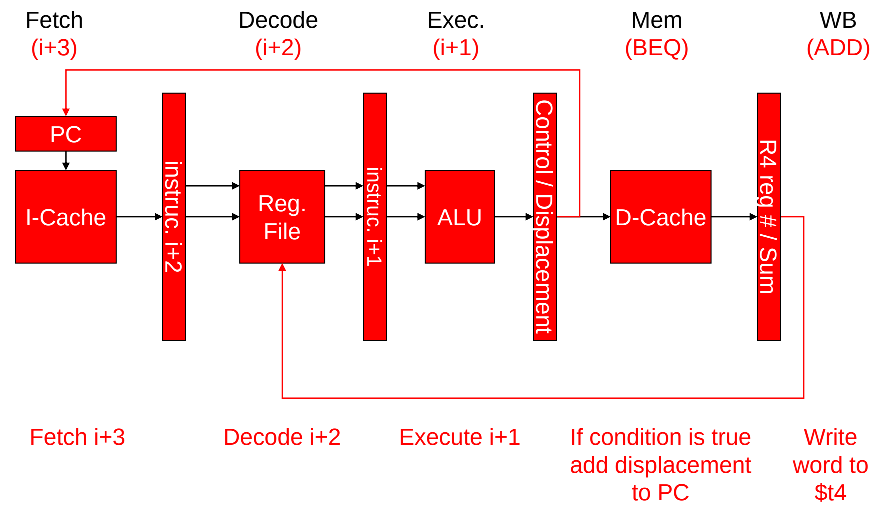
Example



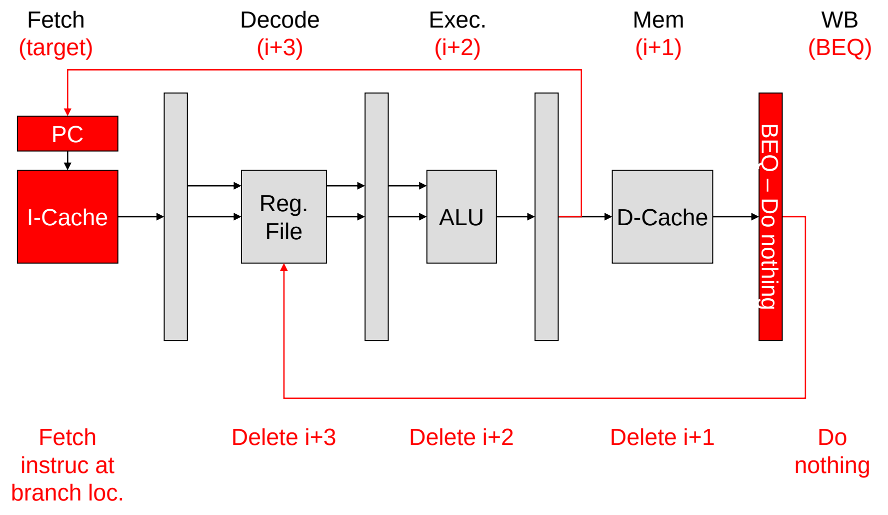
Example



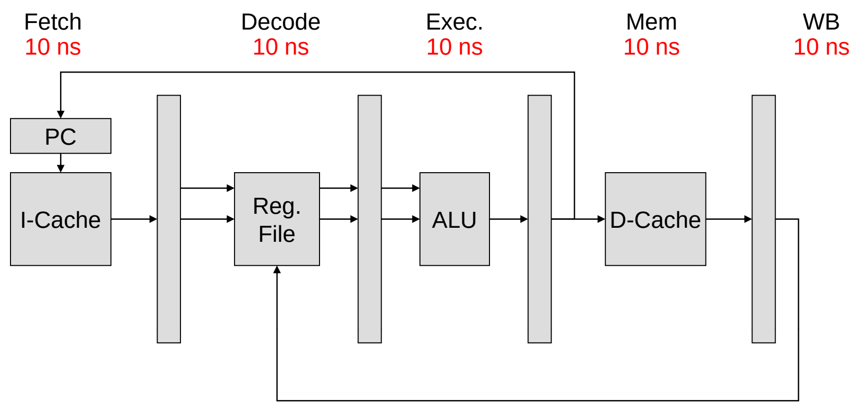
Example



Example



5-Stage Pipeline



Without pipelining (separate execution), each instruction would take _____

With pipelining, each instruction still takes _____ but 1 finishes every _____

Non-Pipelined Timing

- Execute n instructions using a k stage datapath
  - i.e. Multicycle CPU w/ k steps or single cycle CPU w/ clock cycle k times slower
- w/o pipelining: ______ ***cycles***
  - ___________________

	Fetch 10ns	Decode 10ns	Exec. 10ns	Mem. 10ns	WB 10ns
C1	ADD				
C2		ADD			
C3			ADD		
C4				ADD	
C5					ADD
C6	SUB				
C7		SUB			
C8			SUB		
C9				SUB	
C10					SUB
C11	LW				

______ **cycles**

Pipelined Timing

- Execute n instructions using a k stage datapath
  - i.e. Multicycle CPU w/ k steps or single cycle CPU w/ clock cycle k times slower
- w/o pipelining: ***n*k cycles***
  - n instrucs. * k CPI
- w/ pipelining: _________
  - __________ for 1st instruc. + ______ cycles for ______ instrucs.
  - Assumes we keep the pipeline full

	Fetch 10ns	Decode 10ns	Exec. 10ns	Mem. 10ns	WB 10ns
C1	ADD				
C2	SUB	ADD			
C3	LW	SUB	ADD		
C4	SW	LW	SUB	ADD	
C5	AND	SW	LW	SUB	ADD
C6	OR	AND	SW	LW	SUB
C7	XOR	OR	AND	SW	LW
C8		XOR	OR	AND	SW
C9			XOR	OR	AND
C10				XOR	OR
C11					XOR

Pipeline Filling

Pipeline Full

Pipeline Emptying

7 Instrucs. = _______________

Throughput

- Throughput (T) = ____________________________
  - n instructions / clocks to executed n instructions
  - For a large number of instructions, the throughput of a pipelined processor is ___________ **every clock cycle**
  - ***ASSUMES that*** ____________________________

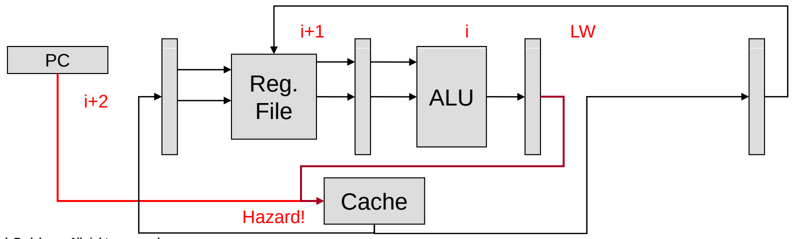
	Non-pipelined	Pipelined
Throughput		

Hazards

- Any sequence of instructions that prevent full pipeline utilization
  - Often causes the pipeline to __________ an instruction
- Structural Hazards = HW organization cannot ______________________ ____
- Data Hazards = Data dependencies
  - Instruction ______ needs result from instruction ___ that is still in pipeline
  - Example:
    - LW $t4, 0x40($s0)
    - ADD $t5,$t4,$t3
  - ADD couldn't decode and get the ______________________________… stalls the pipeline
- Control Hazards = Branches & changes to PC in the pipeline
  - If branch is determined to be taken later in the pipeline, ______________ the instructions in the pipeline that _______________________
- Other causes for stalls: ______________________

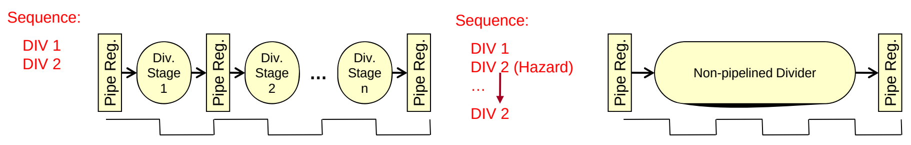
Structural Hazards

- Combinations of instructions that cannot be overlapped in the given order due to HW constraints
  - Often due to lack of HW resources
- Example structural hazard: A single memory rather than separate I & D caches
  - Structural hazard any time an instruction needs to perform a data access (i.e. 'lw' or 'sw')



Structural Hazards Examples

- Another example structural hazard: Fully pipelined vs. non-pipelined functional units with issue latencies
  - Fully pipelined means it may take multiple clocks but a _______ __________________________________
  - Non-fully pipeline means that a new instruction can only be inserted every ______________
  - Example of non-fully pipelined divider
    - Usually issue latencies of 32 to 60 clocks
    - Thus DIV followed by DIV w/in 32 clocks will cause a stall



Data Hazards

- ____________________ Hazard
  - Later instruction reads a result from a previous instruction (data is being communicated between 2 instrucs.)
- Example sequence
  - LW $t1,4($s0)
  - ADD $t5,$t1,$t4

Initial Conditions (assume leading 0's in registers):

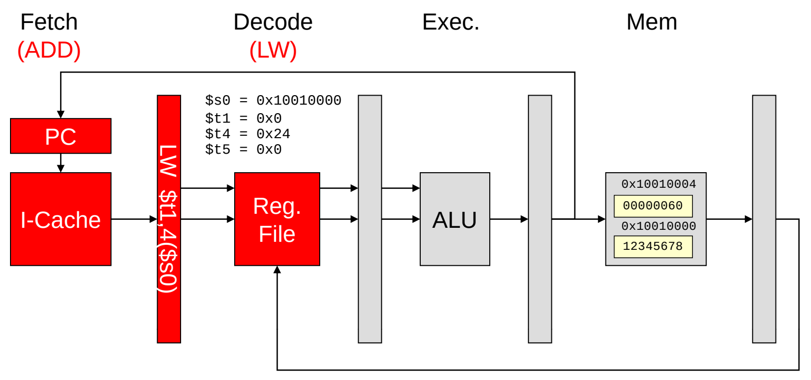
```
$s0 = 0x10010000
$t1 = 0x0
$t4 = 0x24
$t5 = 0x0
```

00000060	0x10010004
12345678	0x10010000

After execution values should be:

```
$s0 = 0x10010000
$t1 = 0x60
$t4 = 0x24
$t5 = 0x84
```

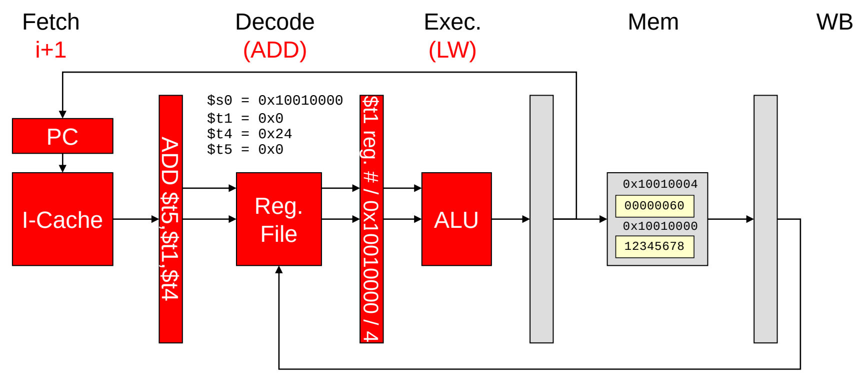
Data Hazards



Fetch ADD

Decode instruction and fetch operands

Data Hazards

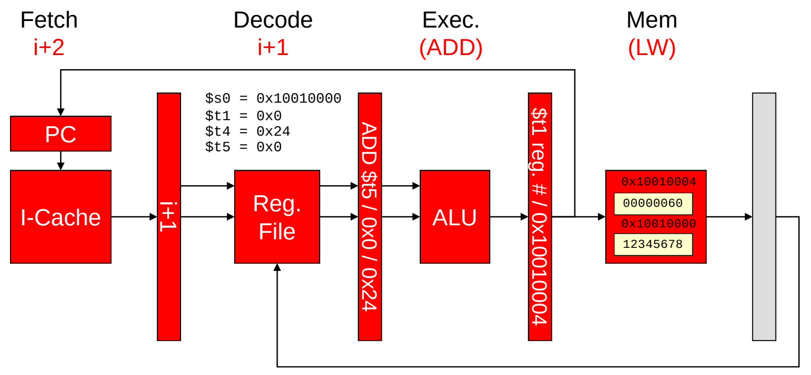


Fetch instruc. i+1

$t1 still = 0x0 rather than the desired 0x60

Add displacement 4 to $t1

Data Hazards



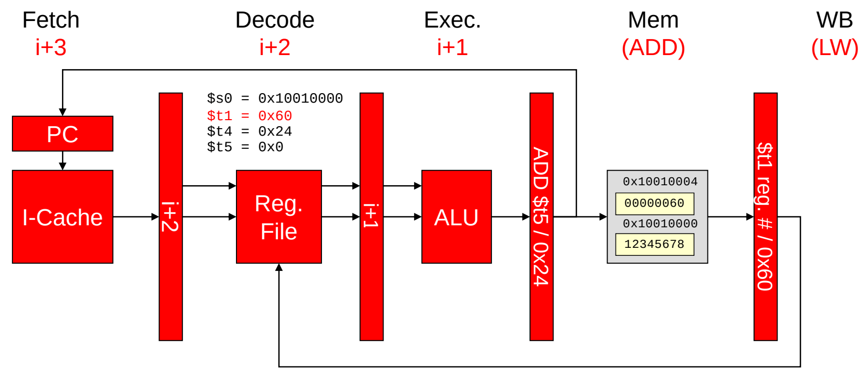
Fetch instruc. i+2

i+1

ADD uses wrong data

Data intended for $t1 is just now read

Data Hazards



Now it's too late the sum of the ADD instruction is wrong!

Data Hazards

Solutions:

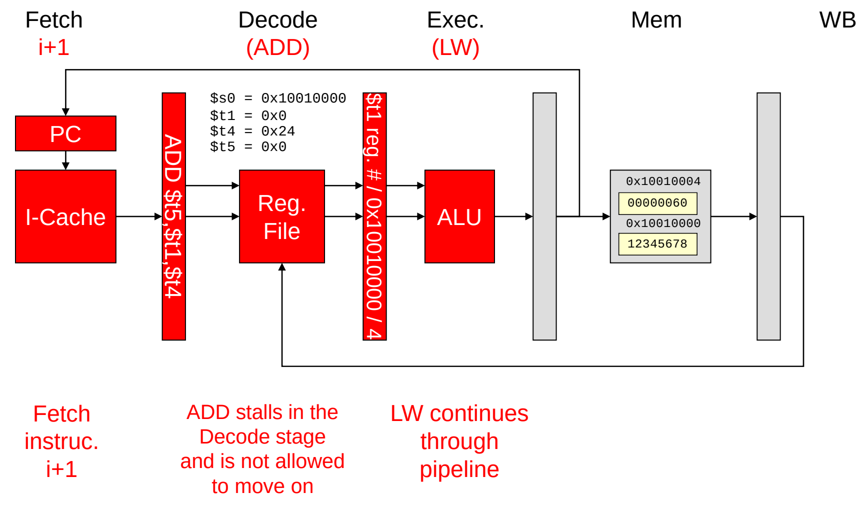
1.

2.

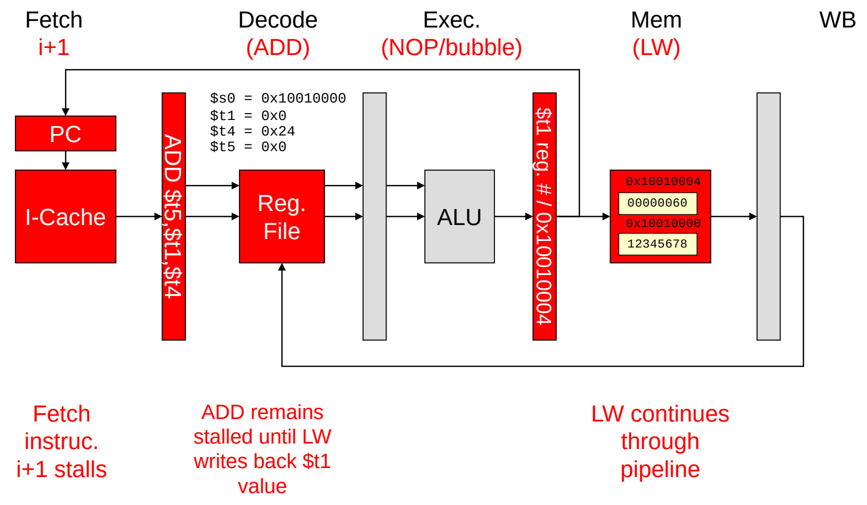
Stalling the Pipeline

- All instructions in front of the stalled instruction can ________________
- All instructions behind the stalled instruction ________________
- Stalling inserts ______________ / nops (no-operations) into the pipeline
  - A "nop" is an actual instruction in the MIPS ISA that does NOTHING

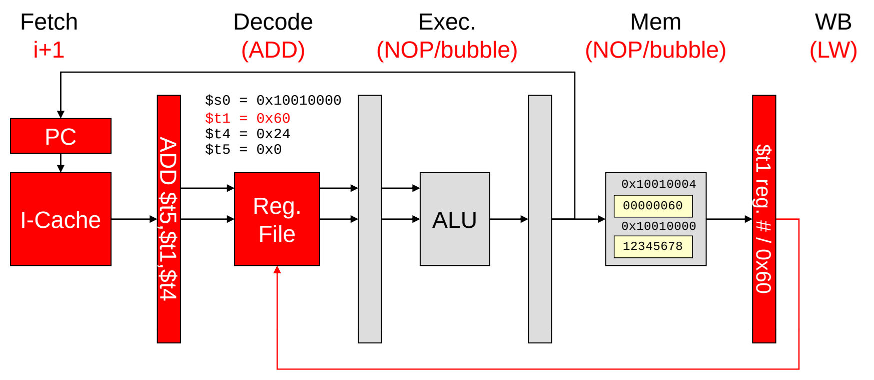
Stalling the Pipeline



Stalling the Pipeline



Stalling the Pipeline

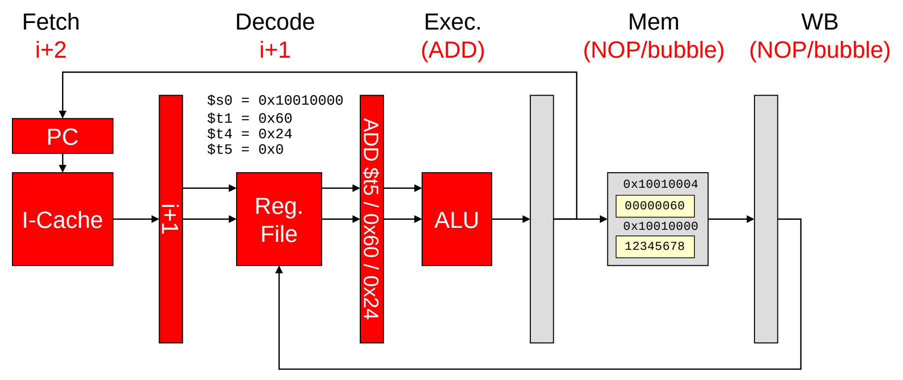


Fetch instruc. i+1 stalls

Reg. file passes new value of $t1 along with $t4 to next stage

LW writes back result to $t1

Stalling the Pipeline



i+2

i+1

Add now has correct value and can proceed

Time Space Diagram

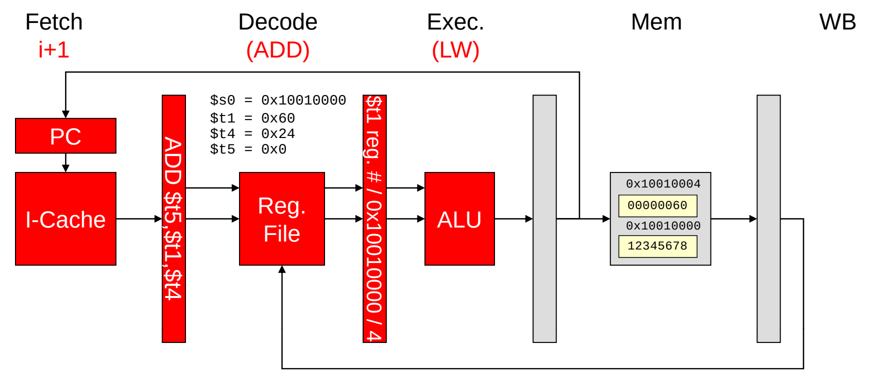
	Fetch 10ns	Decode 10ns	Exec. 10ns	Mem. 10ns	WB 10ns
C1	LW				
C2	ADD	LW			
C3	i	ADD	LW		
C4	i	ADD	nop	LW	
C5	i	ADD	nop	nop	LW
C6	i+1	i	ADD	nop	nop
C7	i+2	i+1	i	ADD	nop
C8	i+3	i+2	i+1	i	ADD

Using Stalls to Handle Dependencies (Data Hazards)

Data Forwarding

- Also known as “bypassing”
- Take results still in the pipeline (but not written back to a GPR) and pass them to dependent instructions
  - To keep the same clock cycle time, results can only be taken from the ___________ of a stage and passed back to the ______________ of a previous stage
  - Cannot take a result produced at the ________ of a stage and pass it to the ____________ of a previous stage because of the stage delays
- Recall that data written to the register file is available for reading in the same clock cycle

Data Forwarding – Example 1

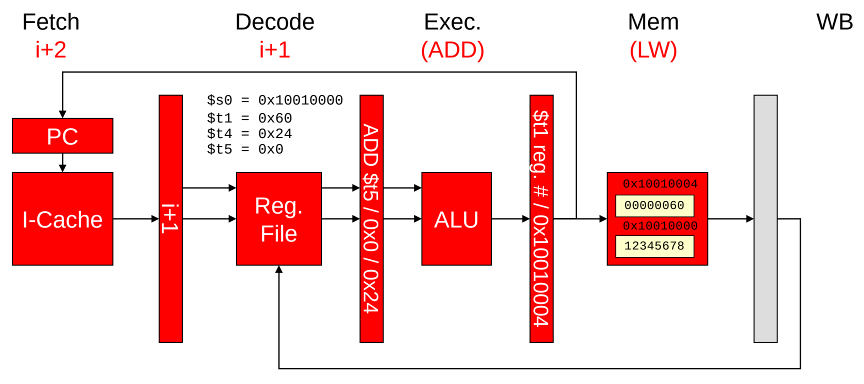


Fetch instruc. i+1

ADD is allowed to fetch the incorrect value of $t1

LW continues through pipeline

Data Forwarding – Example 1



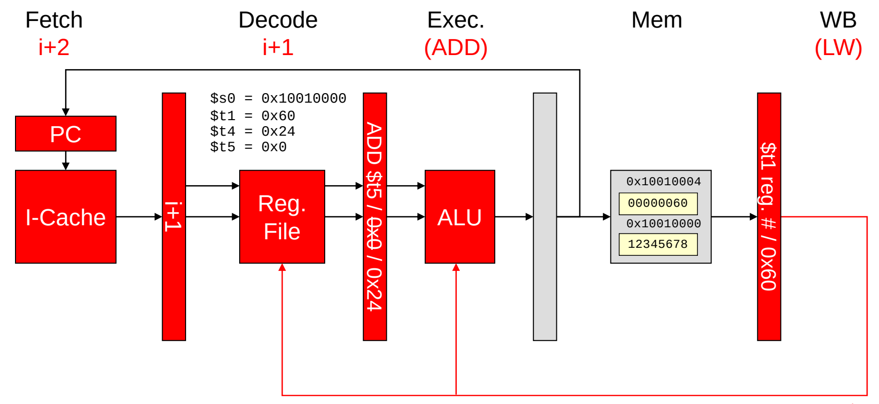
i+2

i+1

ADD cannot get data until after LW does read. So it stalls.

LW continues through pipeline

Data Forwarding – Example 1



i+2

i+1

ADD uses the forwarded data in place of the wrong $t1 value

LW forwards $t1 to EXEC stage and writes back to reg. file

Time Space Diagram

	Fetch 10ns	Decode 10ns	Exec. 10ns	Mem. 10ns	WB 10ns
C1	LW				
C2	ADD	LW			
C3	i	ADD	LW		
C4	i	ADD	nop	LW	
C5	i+1	i	ADD	nop	LW
C6	i+2	i+1	i	ADD	nop
C7	i+3	i+2	i+1	i	ADD

Using Forwarding to Handle Dependencies (Data Hazards)

USC Viterbi School of Engineering

Data Forwarding – Example 2

- ADD $t3,$t1,$t2
- SUB $t5,$t3,$t4
- XOR $t7,$t5,$t3

Initial Conditions (assume leading 0's in registers):

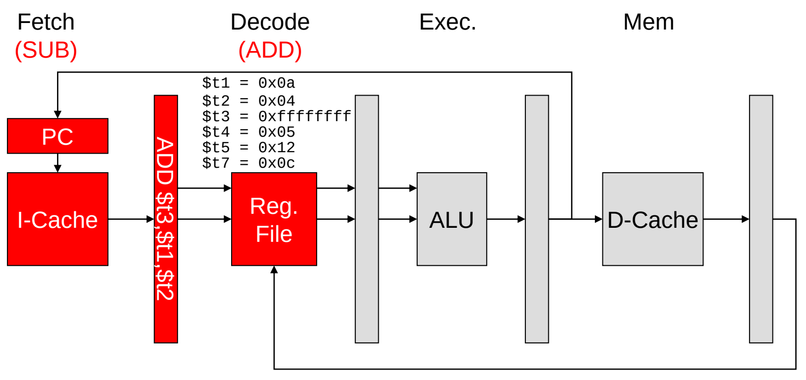
```
$t1 = 0x0a
$t2 = 0x04
$t3 = 0xffffffff
$t4 = 0x05
$t5 = 0x12
```

After execution:

```
$t3 = 0x0e
$t5 = 0x02
$t7 = 0x0c
```

USC Viterbi School of Engineering

Data Forwarding – Example 2

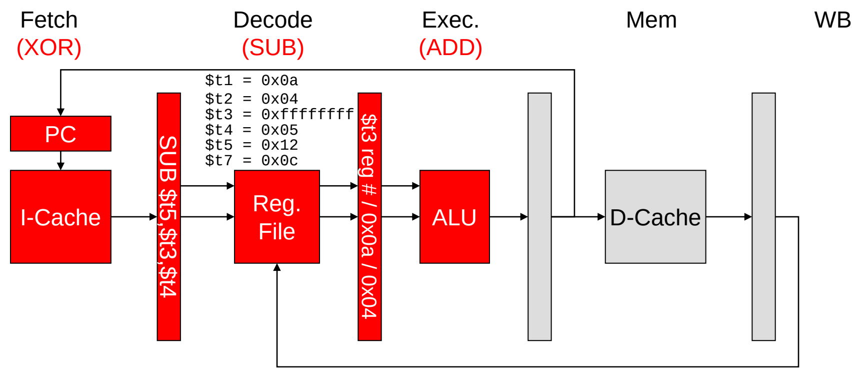


SUB is fetched

ADD decodes and fetches reg. values

USC Viterbi School of Engineering

Data Forwarding – Example 2



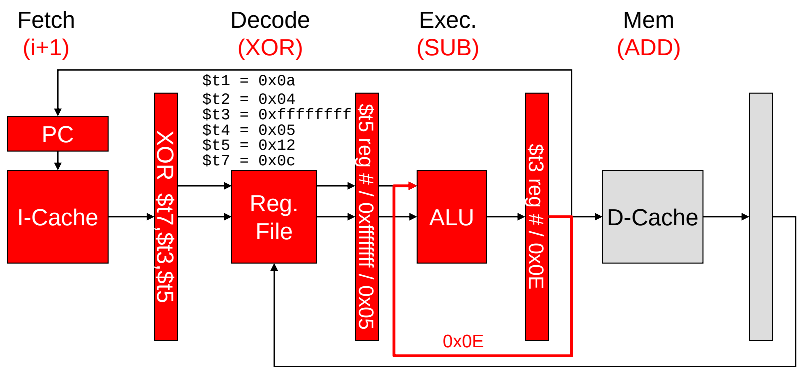
XOR is fetched

SUB decodes and fetches wrong reg. value of $t3

ADD produces the sum

USC Viterbi School of Engineering

Data Forwarding – Example 2



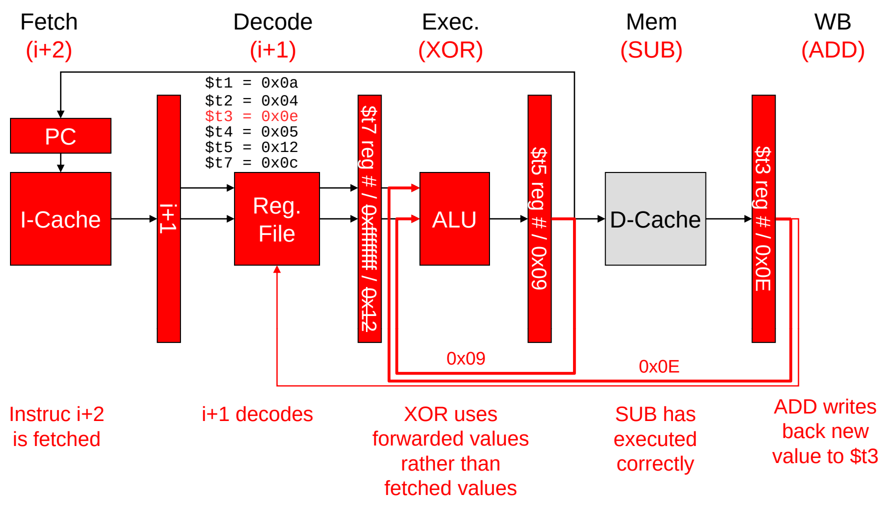
Instruc i+1 is fetched

XOR fetches wrong reg. values for both $t3 and $t5

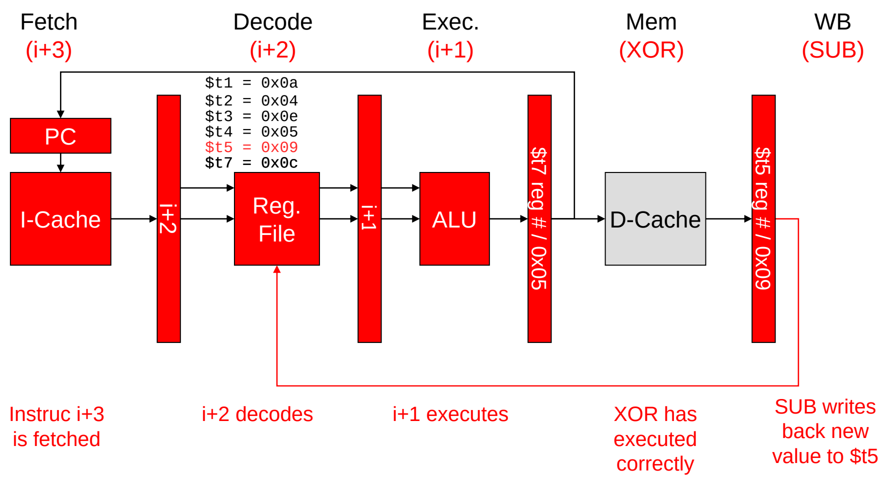
SUB uses forwarded value 0x0e rather than 0xffffffff

ADD forwards the sum to SUB in EXEC stage

Data Forwarding – Example 2



Data Forwarding – Example 2



Time Space Diagram

- ADD $t3,$t1,$t2
- SUB $t5,$t3,$t4
- XOR $t7,$t3,$t5

	Fetch (IF)	Decode (ID)	Exec. (EX)	Mem. (ME)	WB
C1	ADD				
C2	SUB	ADD			
C3	XOR	SUB	ADD		
C4	i	XOR	SUB	ADD	
C5	i+1	i	XOR	SUB	ADD
C6	i+2	i+1	i	XOR	SUB
C7	i+3	i+2	i+1	i	XOR

Using Forwarding to Handle Dependencies

(Requires no stalls/bubbles for dependent instructions)

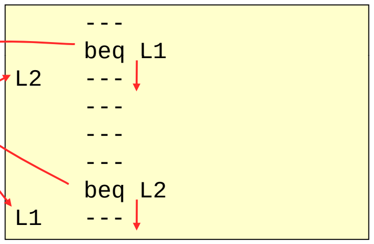
Data Forwarding Summary

- Forwarding paths from…
  - WB to MEM [ADD $t1,$t2,$t3; SW $t1,0($s0)]
  - WB to EX [LW $t1,0($t2); next inst.; SUB $t3,$t1,$t4]
  - MEM to EX [ADD $t1,$t2,$t3; SUB $t3,$t1,$t4]
- Issue Latency = Number of cycles we must **stall** (insert bubbles) before we can issue a dependent instruction

Instruction Type	w/o Forwarding	w/ Full Forwarding
LW	2	___
ALU Instruction	2	___

Control Hazard

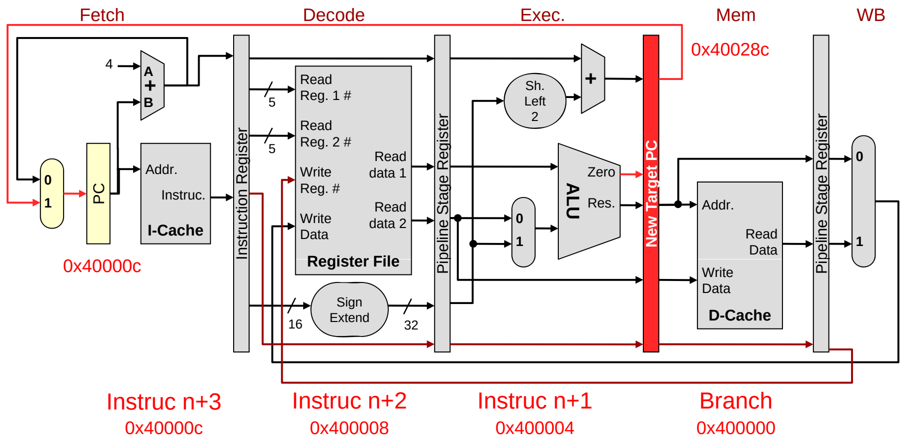
- Branch outcomes: ________ or ___________
- Not known until late in the pipeline
  - Prevents us from fetching instructions that we know will be executed in the interim
  - Rather than stall, predict the outcome and keep fetching appropriately…correcting the pipeline if we guess wrong
- Options
  - Predict ___________
  - Predict ___________



```
     ---
     beq L1
L2   ---
     ---
     ---
     ---
     beq L2
L1   ---
```

Branch Outcome Availability

- Branch outcome only available in MEM stage
  - Incorrect instruction sequence already in pipeline



Branch Penalty

- Penalty = number of instructions that need to be ____________ on misprediction
- Currently our branch outcome and target address is available during the MEM stage, passed back to the Fetch phase and starts fetching correct path (if mispredicted) on the next cycle
- **__cycle_** branch penalty when mispredicted

Predict Not Taken

- Keep fetching instructions from the Not Taken (NT)/sequential stream
- Requires us to “flush”/delete instructions fetched from the NT path if the branch ends up being Taken

Predict Not Taken

```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $s0,$s1,L2  (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

	Fetch (IF)	Decode (ID)	Exec. (EX)	Mem. (ME)	WB
C1	BEQ				
C2	ADD	BEQ			
C3	SUB	ADD	BEQ		
C4	OR	SUB	ADD	BEQ	
C5	BNE	OR	SUB	ADD	BEQ
C6	AND	BNE	OR	SUB	ADD
C7	SW	AND	BNE	OR	SUB
C8	LW	SW	AND	BNE	OR
C9	ADD	nop	nop	nop	BNE
C10	SUB	ADD	nop	nop	nop

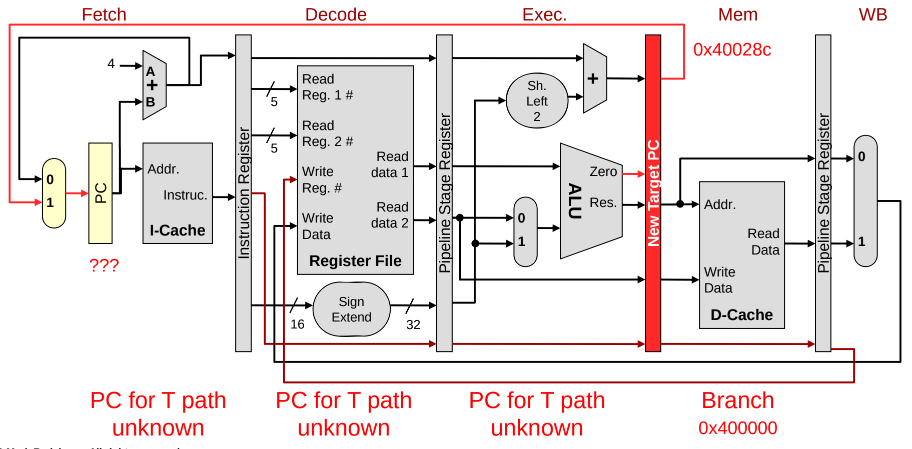
Using Predict NT keeps the pipeline full when we are correct and flushes instructions when wrong (penalty = 3 for our 5-stage pipeline)

Predict Taken

- In our 5-stage pipeline as currently shown, predicting taken is …
- In other architectures we may be able to know the branch target early and thus use this method, however, if we predict incorrectly we still must flush

Predicting Taken

- Branch target address not available until MEM stage



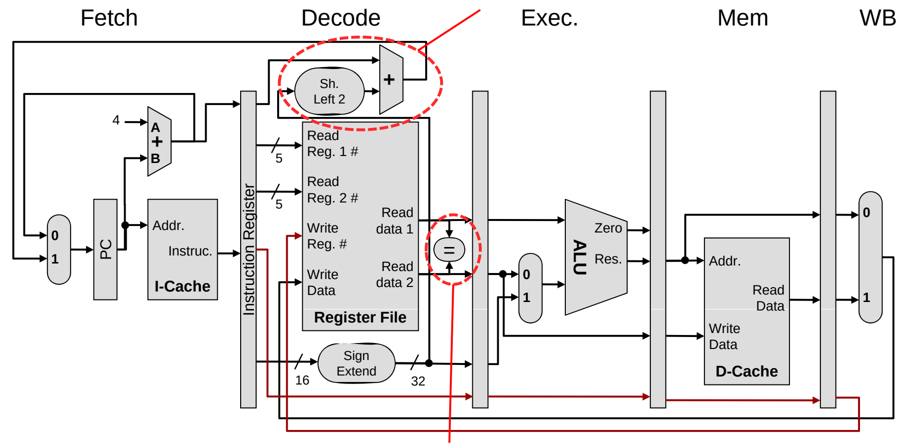
Early Branch Determination

- Goal is to keep the pipeline full and avoid bubbles/stalls
- Number of bubbles/stalls introduced by control hazards (branches) depends on when we determine the outcome and target address of the branch (______________________)
- Currently, these values are available in the MEM stage
- We can try to reorganize the pipeline to make the branch outcome and target address available earlier

Early Branch Determination

- By actually adding a little bit of extra HW we can move the outcome determination and target address calculation to the ____________ stage
  - Again this may cause a small increase in clock period

Reorganized 5-Stage Pipeline



Early Determination w/ Predict NT

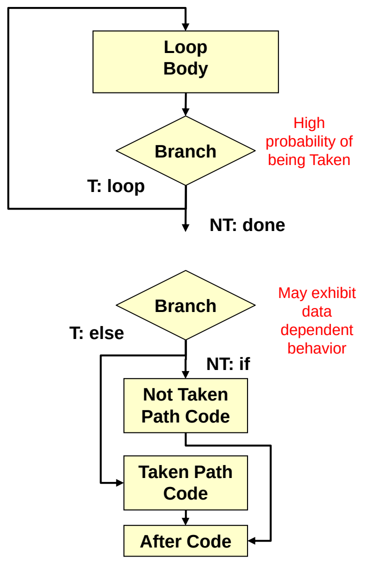
```
    BEQ  $a0,$a1,L1  (NT)
L2: ADD  $s1,$t1,$t2
    SUB  $t3,$t0,$s0
    OR   $s0,$t6,$t7
    BNE  $s0,$s1,L2   (T)
L1: AND  $t3,$t6,$t7
    SW   $t5,0($s1)
    LW   $s2,0($s5)
```

	Fetch (IF)	Decode (ID)	Exec. (EX)	Mem. (ME)	WB
C1	BEQ				
C2	ADD	BEQ			
C3	SUB	ADD	BEQ		
C4	OR	SUB	ADD	BEQ	
C5	BNE	OR	SUB	ADD	BEQ
C6	AND	BNE	OR	SUB	ADD
C7			BNE	OR	SUB
C8				BNE	OR
C9					BNE
C10					

Using early determination & predict NT keeps the pipeline full when we are correct and has a single instruction penalty for our 5-stage pipeline

A Look Ahead: Branch Prediction

- Currently we have a static prediction policy (NT)
- We could allow a ________ prediction *per instruction* (give a _____ with the branch that indicates T or NT)
- We could allow ________ predictions *per instruction* (use its _______________)



Exercise

- Schedule the following code segment on our 5 stage pipeline assuming…
  - Full forwarding paths (even into decode stage for branches)
  - Early branch determination
  - Predict NT (no delay slots)
- Calculate the CPI from time first instruction completes until last BEQ instruction completes
- Show forwarding using arrows in the time-space diagram

```
    ADD  $s0,$t1,$t2
L1: LW   $t3,0($s0)
    SLT  $t1,$t3,$t4
    BEQ  $t1,$zero,L1  (T, NT)
    SUB  $s2,$s3,$s4
    ADD  $s2,$s2,$s5
```

- CPI = ______________________________

	Fetch	Decode	Exec.	Mem.	WB
C1					
C2					
C3					
C4					
C5					
C6					
C7					
C8					
C9					
C10					
C11					
C12					
C13					
C14					
C15					
C16					