Basic/Intermediate Pipelining

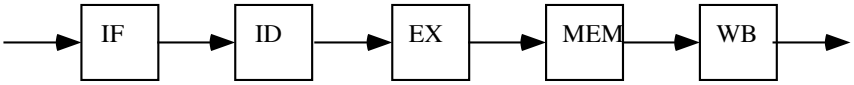
(See class textbook –5th Edition - Appendix C;
Older editions of the class textbook carry the same material;
Introductory version of textbook covers this material 'lightly')

- "**pipelining**" = ability to execute multiple instructions at the same time
- pipelines are organized into "stages", each stage is one hardware module
- machine clock cycle time (clock period) = time required for slowest stage to complete its work
- to maximize performance, work should be evenly distributed over pipeline stages, so all stages have nearly the same completion times

Pipelined RISC machine

- older CISC machines cannot be easily pipelined, since instructions vary immensely in complexity, from 1 c.c. to 100 c.c. per instruction
- original RISC machines supported pipelineable instructions, which execute in typically 5-10 clock cycles (depending upon pipeline depth)
- newer machines are more complex, using '**dynamic scheduling**' (to be studied)

Typical 5-Stage Pipelined CPU



- All stages perform separate tasks
- **Instruction-Fetch stage** : IR <- Mem[PC], % fetch instruction & inc PC
  NPC <- PC+4
- **Instruction-Decode stage**: A <- Reg[A field of instruction], % fetch A operand
  B <- Reg[B field of instruction], % fetch B operand
  Imm <- Immediate field of IR % fetch immed operand
- **Execute stage**: for ALU instruction, compute result:
  ALU output = A op B (from OP field of ALU instructions)
- for **MEM** instructions (Load/Store), compute effective address:
  ALU output <- A + Imm % assume displacement mode
- for **BRANCH** instructions, compute branch target and branch outcome:
  ALU output <- NPC + Imm % compute branch target
  Cond <- (A op 0) % compare with 0

• **MEM stage** : for **Mem** instructions, perform Memory operation if necessary:

LMD <- Mem[ALU output latch] % perform LOAD

Mem[ALU output] <- B % perform STORE

• **MEM stage** : for **Branch** Instruction, perform PC update:

if (cond) PC <- ALU output % update PC if necessary

else PC <- NPC

• **Write Back stage**: (write back results to register file if necessary):

- for Reg-Reg ALU instructions:

Reg[destination field of instr] <- ALU output

- for Reg-Imm ALU instructions:

Reg[destination field of instr] <- ALU output

- for Load instructions:

Reg[destination field of instr] <- LMD

• consider the above CPU ***without pipelining*** first; each instruction must pass through all 5 stages, requiring 5 clock cycles per instruction (= a multi-cycle unpipelined CPU)

• (LMD = 'load memory data' register)

Time per CPU Stage

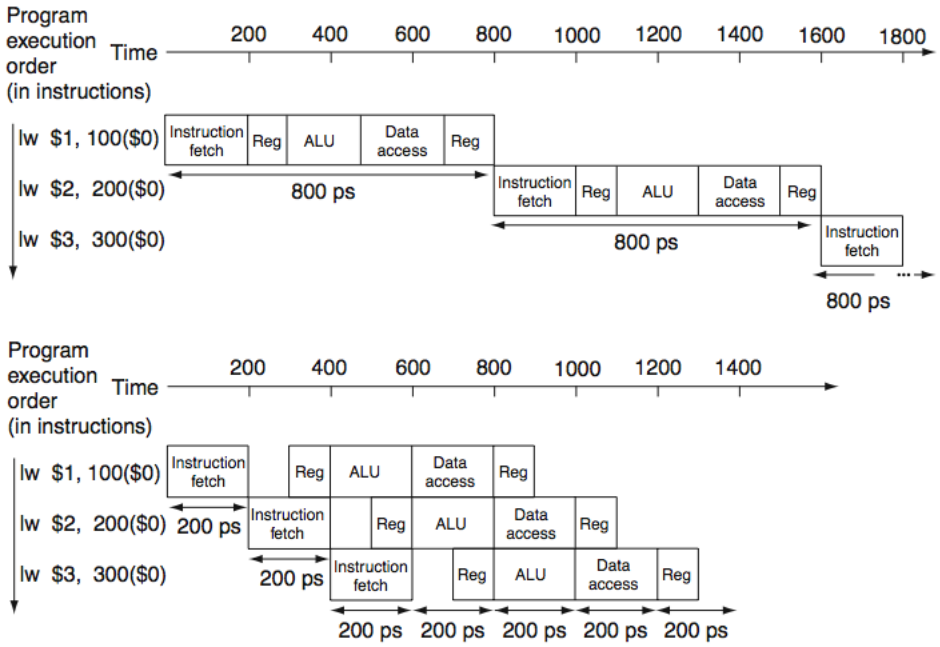
(introductory text - Fig. 62, pg, 373)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

FIGURE 6.2 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Nonpipelined vs. pipelined CPU

(introductory text - fig. 6.3, pg, 373)

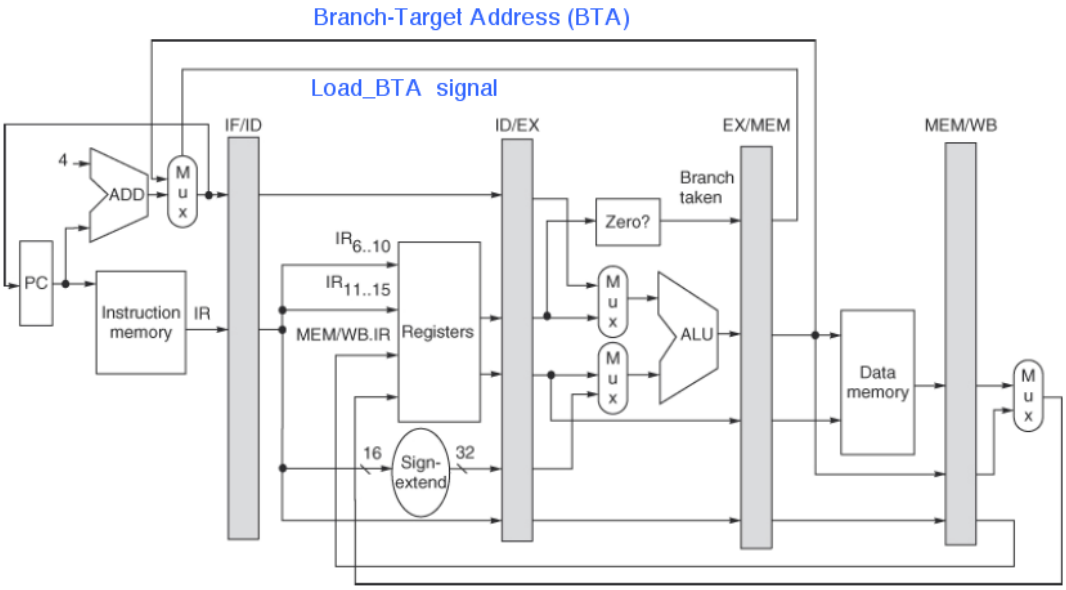


Now Add Pipelining-> the RISC machine

- previous hardware is suitable for pipelining : all instructions require 5 c.c. to complete, and all instructions pass through the same 5 stages, in the same order
- without pipelining, all instructions take 5 c.c., so average CPI = 5
- without pipelining, CPU time for 1,000 instructions = 5,000 c.c.
- to pipeline, add "***pipeline latches***" between all stages
- in every clock cycle, a new instruction enters the pipeline in stage # 1, and the contents of all stages move right one stage
- instructions enter and exit the pipeline one instruction per clock cycle
- every instruction stays within the pipeline for 5 clock cycles -> "latency = 5 cc."
- CPU time to execute 1,000 instructions = 1000 cc. plus 4 cc to ***flush*** the pipeline
- the effective CPI =1004 cc/1000 inst = 1.004 cc per instruction, about 5 times faster than the same hardware circuit without the pipeline latches

- => Pipelining a 5 stage CPU results in about 5x performance improvement over unpipelined CPU

Pipelined CPU with Pipeline Registers (class text - Fig. C.22, pg. C-35)



• Add pipeline registers between stages. The clock rate can be increased significantly due to pipelining, since the combinational logic delays are significantly lowered .

Basic RISC Pipeline - "Space-Time" Diagrams

(class text – Fig C.1, pg C-7)

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
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

FIGURE 3.2 Simple DLX pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a machine that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the implementation on pages 127–129: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

• with pipelining, CPU works on 5 instructions at a time; each instruction is in a different pipeline stage, so they don't collide.

• look at clock cycle 5, where 5 different instructions are in the CPU at the same time

• but, pipelining has some stringent requirements;

• with pipelining, **IF** unit must fetch & deliver instructions 5x as fast

• with pipelining, **MEM** unit must fetch & deliver data 5x as fast

• register file used twice in every clock cycle; used once in stage 2 (**ID** stage) when A,B operands being read; used one in stage 5 (**WB** stage), when C operand is written back to register file -> pipelining requires "***dual ported***" register file

Pipeline Hazards (class text)

• "***hazards***" prevent an instruction from executing in its designated clock cycle & reduce performance, when compared to an "ideal" pipeline without hazards

• 3 types of hazards:

- ***structural*** hazards - conflict for scarce hardware resources

- ***data*** hazards - results of new instruction depend upon results of previous instructions, which haven't been computed yet

- ***control*** hazards - occur from pipelining branches ; by the time we figure out whether to branch, we have already started executing down the wrong path of the branch

Datapaths shifted in time (class text - Fig. C.2, pg. C-8)

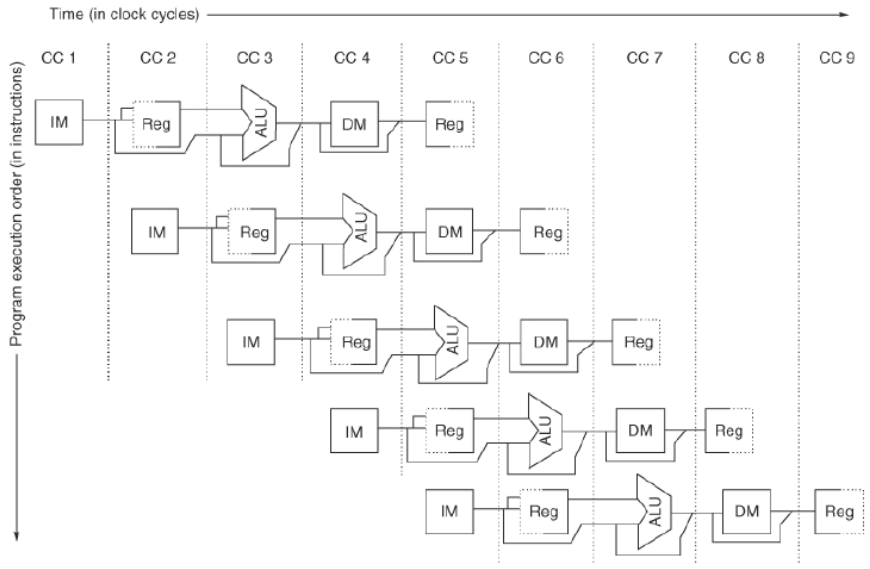


Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of

• pipelining can be viewed as using all 5 of the hardware units, in each clock cycle

• the datapath (with 5 pipeline stages) can be viewed as shifted in time, where each stage is re-used in each time-slot

Structural Hazard due to 1 Port Memory (Fig. C.4, pg C-14)

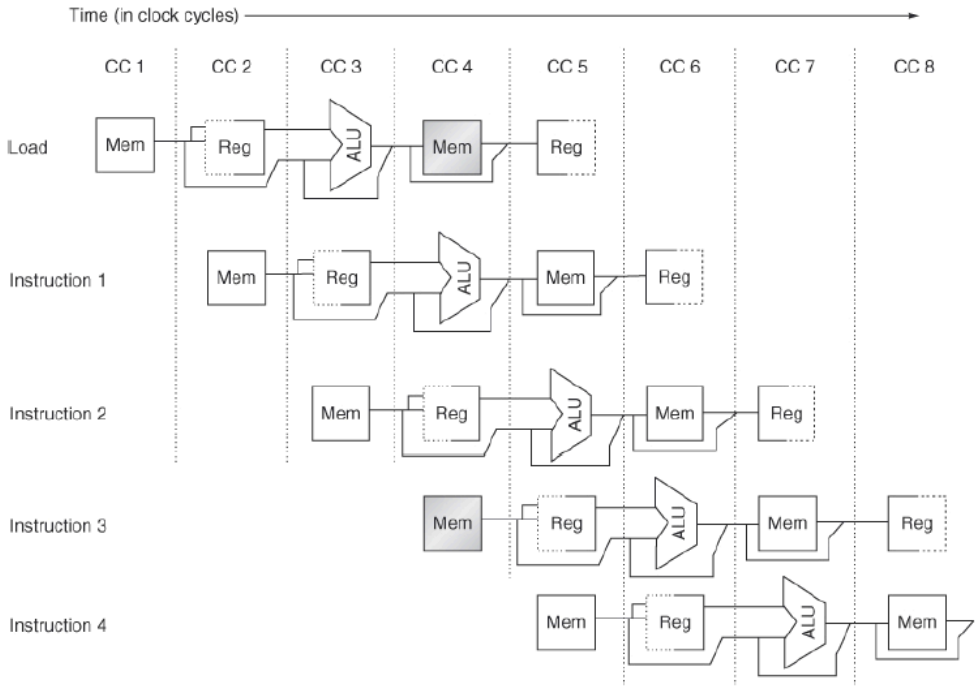


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

2013,

Structural Hazard (1 Port Memory) causes Stall

(class text - Fig. C.5, pg C-15)

	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i+1$		IF	ID	EX	MEM	WB				
Instruction $i+2$			IF	ID	EX	MEM	WB			
Instruction $i+3$				stall	IF	ID	EX	MEM	WB	
Instruction $i+4$						IF	ID	EX	MEM	WB
Instruction $i+5$							IF	ID	EX	MEM
Instruction $i+6$								IF	ID	EX

FIGURE 3.8 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction $i+3$). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction 3 does not begin execution until cycle 5. We use the form above, since it takes less space.

- a single memory port handles both instructions and data, creating a ***hazard*** when data is fetched for a previous Load instruction (in **MEM** stage), and a new instruction is fetched in **IF** stage

Pipeline Performance with Stalls (class text – pg C-12)

$$speedup_{pipelining} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

- CPI unpipelined = 5 cc
- CPI pipelined = (1 + pipeline stalls per inst)

$$speedup_{pipelining} = \frac{5}{(1 + \text{pipeline stall cycles per instruction})}$$

- what separates an "ideal" pipeline from a real pipeline are **the stall cycles per instruction**, that the real machine encounters due to hazards. More generally;

$$speedup_{pipelining} = \frac{\text{pipeline depth}}{(1 + \text{pipeline stall cycles per instruction})} * \frac{\text{clock cycle}_{\text{unpipelined}}}{\text{clock cycle}_{\text{pipelined}}}$$

Pipeline stalls per inst. = sum(% instr. type j)* (stalls instr. type j)

Speedup, Dual port vs Single Port (class text – pg. C-14)

- machine A: dual-ported memory (separate memory ports for instructions & data)
- machine B: single-ported (shared) memory, with slightly faster clock (1.05x as fast)
- ideal CPI (without any hazards) = 1 cc/inst, for both machines
- data references form 40 % of instruction mix
- **Solution using Average CPIs**:

$$\text{Average Instruction Time} = \text{CPI} * \text{Clock Cycle time}$$

$$\text{Average Instruction Time}_{A} = 1.0 * \text{Clock Cycle time}$$

$$\text{Average Instruction Time}_{B} = \text{CPI} * \text{Clock Cycle time}$$

$$= (1 + 0.4 * 1) * \frac{\text{Clock Cycle time}_{\text{ideal}}}{1.05}$$

$$= 1.3 * \text{Clock cycle time}_{\text{ideal}}$$

- machine without memory hazard is 1.3 times faster. So, by adding another memory port, RISC machines gain about 30 % advantage over single -ported machines

Structural Hazards

• structural hazards exist because the cost of avoiding them is high (duplication of hardware)

• Example: many new machines do not fully pipeline the Floating-Point-Unit (FPU) -> this is a ***structural hazard***

• suppose RISC machine has a 5 cc latency for FP MULT with no pipelining (ie it takes 5 cc to compute a FP MULT result)

• if calls to FP MULT are separated by > 5 cc on average, then there is no degradation in performance at all due to the potential structural hazard of an unpipelined FP MULT

• most programs usually don't generate enough FP MULTs to fully exploit a pipeline, so inexpensive CPUs may use an unpipelined or partially pipelined MULT unit

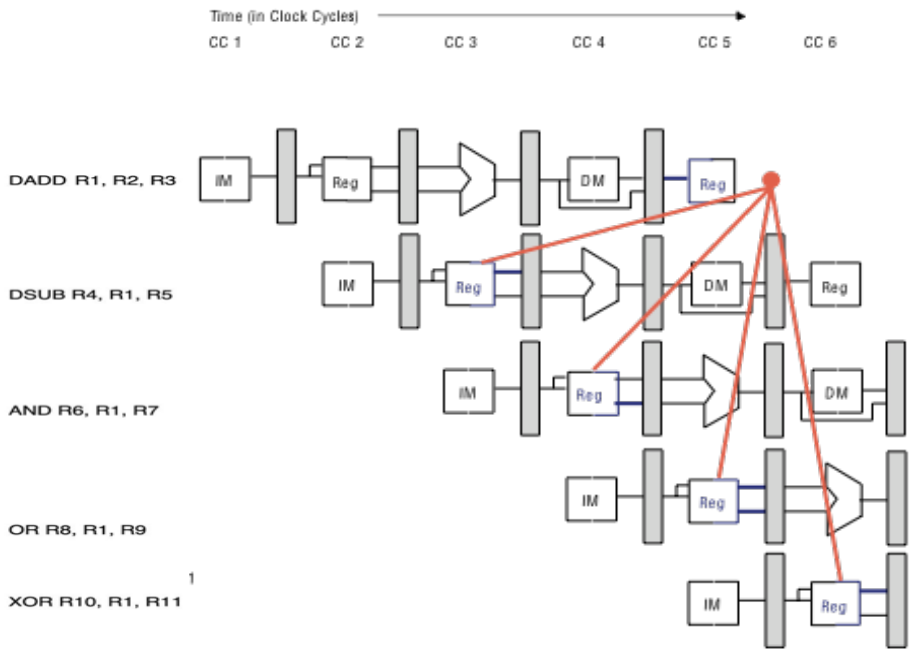
Data Hazards (class text – pg. C-16)

• ***data hazards*** occur when results of new instruction depend upon results of a previous instruction which hasn't finished executing yet

• Consider this instruction sequence:

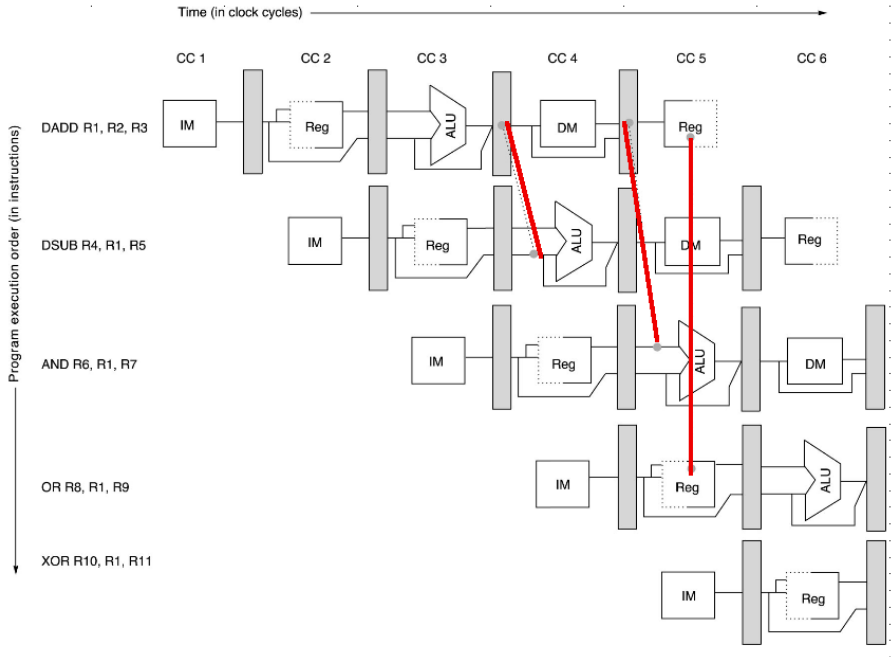
```
ADD     R1, R2, R3      -- destination R1 not written back until WB stage
SUB     R4, R5, R1      -- incorrect operand R1 fetched in ID stage
AND     R6, R1, R7      --    "
OR      R8, R1, R9      --    "
XOR     R10, R1, R11    -- destination R1 written back now !!!
```

Data Hazards in ALU stage (class text - Fig. C.6, pg C-17)



• result of ADD not written back until WB stage in cc 5 -> data hazard for next 3 instructions

Data Forwarding (class text, Fig. C.7, pg C-18)



problem overcome by "data forwarding" - add datapaths and multiplexers to hardware, to allow the correct data (R1) to follow the arrows in the above figure

Data Forwarding Circuitry

		Clock number					
		1	2	3	4	5	6
ADD	**R1**, R2, R3	IF	ID	EX	MEM	WB	
SUB	R4, R5, **R1**		IF	ID	EX	MEM	WB
AND	R6, **R1**, R7			IF	ID	EX	MEM
OR	R8, **R1**, R9				IF	ID	EX
XOR	R10, **R1**, R11					IF	ID

- all these arithmetic instructions use **R1**
- ADD doesn't write back R1 until cc 5
- without data forwarding, SUB, AND, OR read incorrect values of R1 in their ID stages (in cc. 2,3,4)
- arrows indicate forwarded data on the above timing diagram, to correct the problem

Another ALU Forwarding Example (class text, Fig. C.8, pg. C-19)

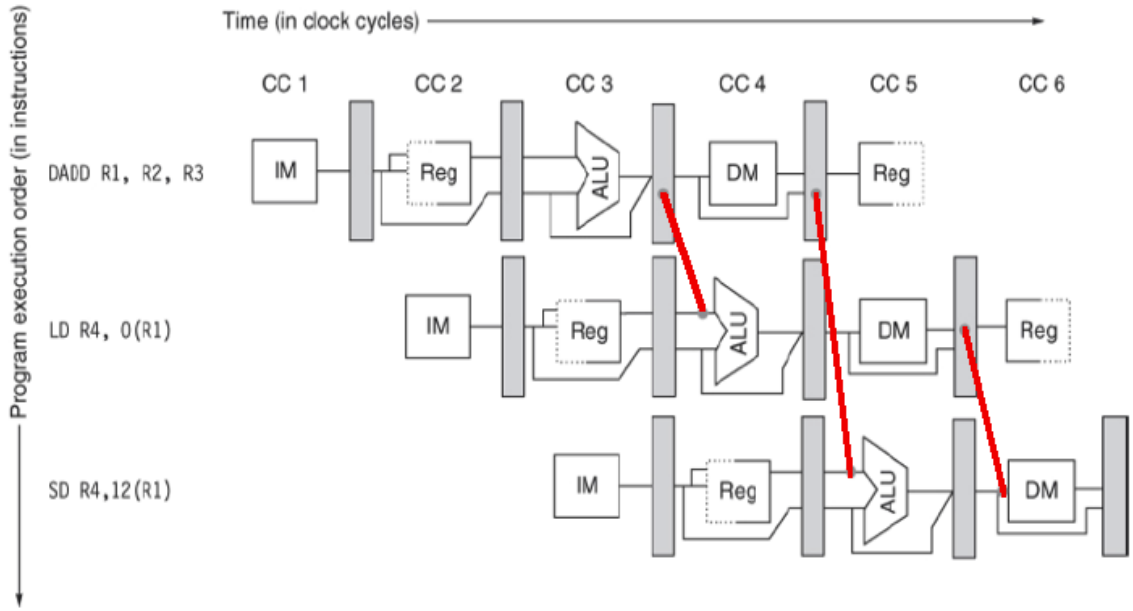
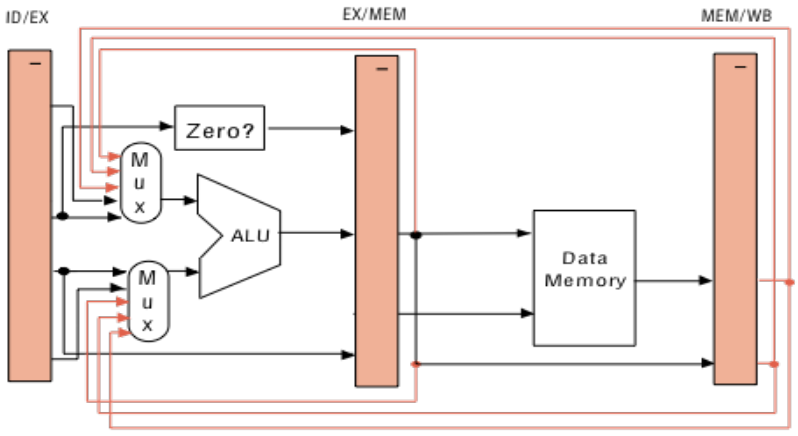


Figure C.8 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

Data Forwarding Hardware (class text – Fig. C-27, pg. C-41)

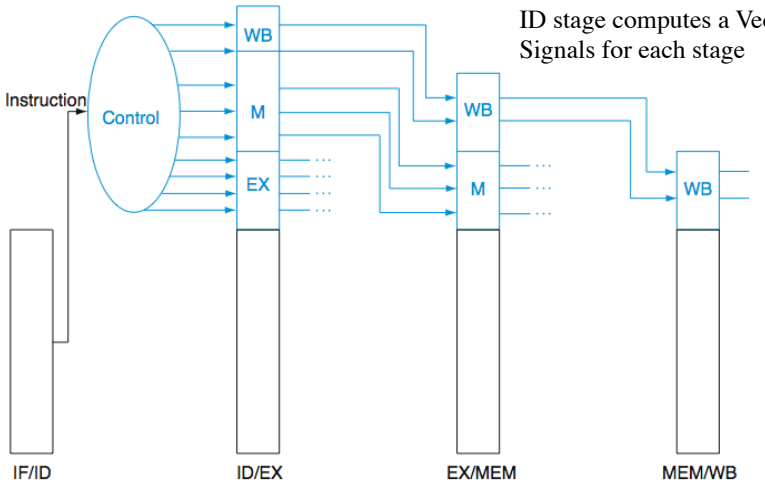


• data forwarding to ALU requires multiple 4-to-1 muxes, 32 bits wide, on ALU inputs, to select forwarded data, along with controller to enable the forwarding

Data Forwarding Controller

(introductory text – fig. 6.16, pg 403)

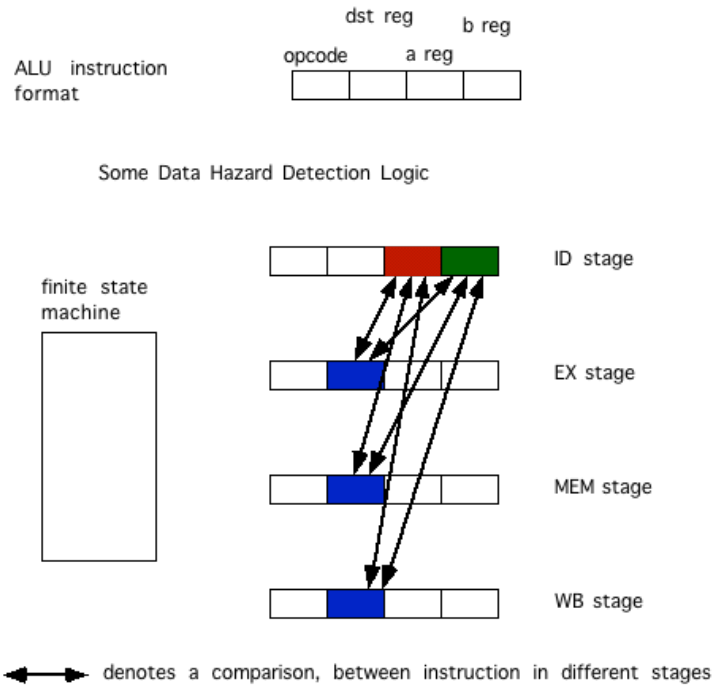
ID stage computes a Vector of control Signals for each stage



• Data hazards can be detected in the ID stage, which can generate the appropriate control signals in the next few clock cycles, to resolve the data hazards

• the ID unit needs to examine the **new instruction** it receives, and the **last 2 instructions** (which are entering the EX and MEM stages) ; The ID unit can keep local copies of the last 2 instructions, in two 32-bit-wide shift registers in the ID stage

Comparison Hardware to Detect Data hazards in ID Stage



Data Forwarding, for Load & Stores

• with forwarding circuitry, current results of EX or MEM stages can be used by a subsequent instruction entering EX stage, by enabling the multiplexers from EX/MEM stages back into the EX stage

• the EX stage always checks to see if either of the 2 previous instructions have just computed an operand that it needs; if so, it fetches the operand using the forwarding circuitry

• requires **Combinational Logic** to examine current instr. + last 2 instructions

• Here is an example

ADD **R1**, R2, R3
LW R4, 8(**R1**)
SW 12(**R1**), R4

	Clock number					
	1	2	3	4	5	6
ADD R1, R2, R3	IF	ID	EX	MEM	WB	
LW R4, 8(R1)		IF	ID	EX	MEM	WB
SW 12(R1), R4			IF	ID	EX	MEM
				IF	ID	EX
					IF	ID

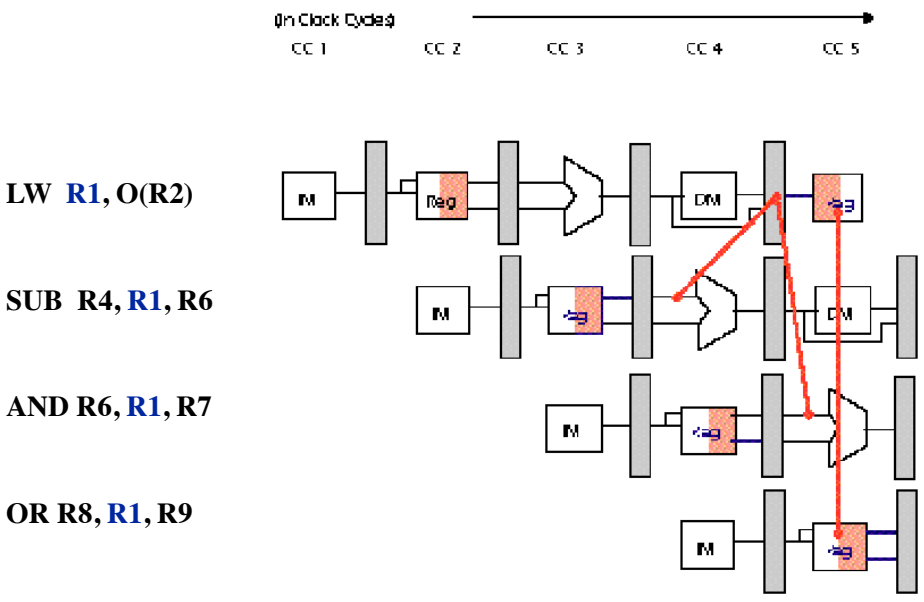
No Stalls here

Load Data Hazards Forcing Extra Stalls (class text – Fig. C.10, pg. C-21)

		1	2	3	4	5	6	7
LW	**R1**, 0(R2)	IF	ID	EX	MEM	WB		
SUB	R4, **R1**, R6		IF	ID	**stall**	EX	MEM	WB
AND	R6, **R1**, R7			IF	**stall**	ID	EX	MEM
OR	R8, **R1**, R9				**stall**	IF	ID	EX

- a timing diagram for this situation is shown on following slide
- Assuming a cache hit, the Load instruction receives **R1** from memory at end of the MEM stage in cc 4 (only if we have a cache hit; otherwise many more stalls will be inserted)
- the SUB instruction cannot enter the EX stage 3 until after the MEM stage forwards **R1, ie after cc 4**, so a stall is inserted, delaying the SUB instruction entry into the EX stage by 1 cc
- due to the pipelined structure, all subsequent instructions are stalled at same cc 4
- ID unit detects hazard when SUB instruction is in the ID unit (it retains copies of 2 previous instructions, for hazard checking); it asserts a global 'Stall' signal, which stops the IF unit from fetching and incrementing the PC, but which allows other stages to continue

Load Data Hazards Forcing Extra Stalls (class text - Fig. C.9, pg. C-20)



- **figure C.9:** LW can forward data to AND and OR instructions but not to SUB, which would forward the result in "negative time" on the timing diagram -> this necessitates the 1 cc "stall"

Load Data Hazards Forcing Extra Stalls (class text – Fig. C.10, pg. C-21)

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	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	

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	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

Figure A.10 In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

This slide illustrates the same information as the previous slides
It shows both cases (untaken branch - top, and the taken branch-bottom)

CPI, Load Data Hazards Stalls (class text 2nd edition, - pg 155)

- Suppose 30 % of instructions are LOADS, and 50 % of the time the instruction immediately after the LOAD depends upon the result of the LOAD, creating a hazard & stall.
- How much faster is ideal pipeline (CPI = 1) versus real pipeline with load stalls ?
- Answer: Average CPI for instruction immediately after LOAD is 1.5, since CPU needs 1 clock cycle just to fetch the instruction, and half the time it encounters a hazard in its execution (it needs the data from the LOAD) & stalls
- effective CPI of all instructions = (0.7 * 1 cc + 0.3 * 1.5 cc) = 1.15
- effective CPI of ideal machine = 1.0
- so, ideal machine is about 15 % faster.
- Compilers can re-arrange instruction execution sequence to minimize stalls
- Here are 2 formulas for CPI, which effectively compute a weighted average:

$$CPI = \sum_i (1 + (\text{fraction inst. type i}) * (\text{stalls per inst. type i}))$$

$$CPI = \sum_i (\text{fraction inst. type i}) * (\text{CPI inst. type i})$$

Compiler Scheduling to Reduce Load Stalls

(class text - pg. 155, 2nd edition)

• consider 2 lines of C code:

a = b + c; /* a, b, c, d are variables in memory */

d = e - f;

• Assume Loads have a latency of 1 cc (that means they cause 1 stall if the immediately following instruction relies upon the loaded value)

• Unscheduled code:

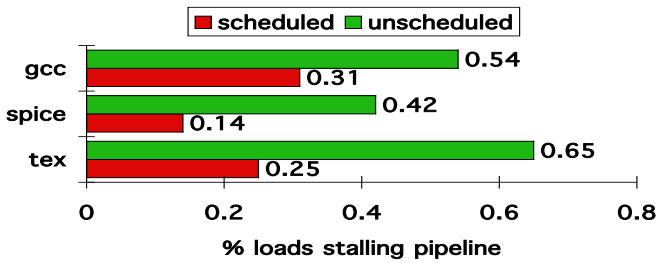
```
LW      Rb, b
LW      Rc, c
stall 1 cc
ADD     Ra, Rb, Rc
SW      a, Ra
LW      Re, e
LW      Rf, f
stall 1 cc
SUB     Rd, Re, Rf
SW      d, Rd
```

Compiler scheduled 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
```


Compiler Scheduling to Reduce Load Stalls

(class text - Fig 3.16, pg. 157, 2nd edition)



• without compiler scheduling, up to 65 % of all Loads can cause stalls

• with compiler scheduling, Load stalls reduced by at least 50 %

• scheduling will reduce average CPI got load instructions moderately

Branch Hazards (class text, pg. C-21)

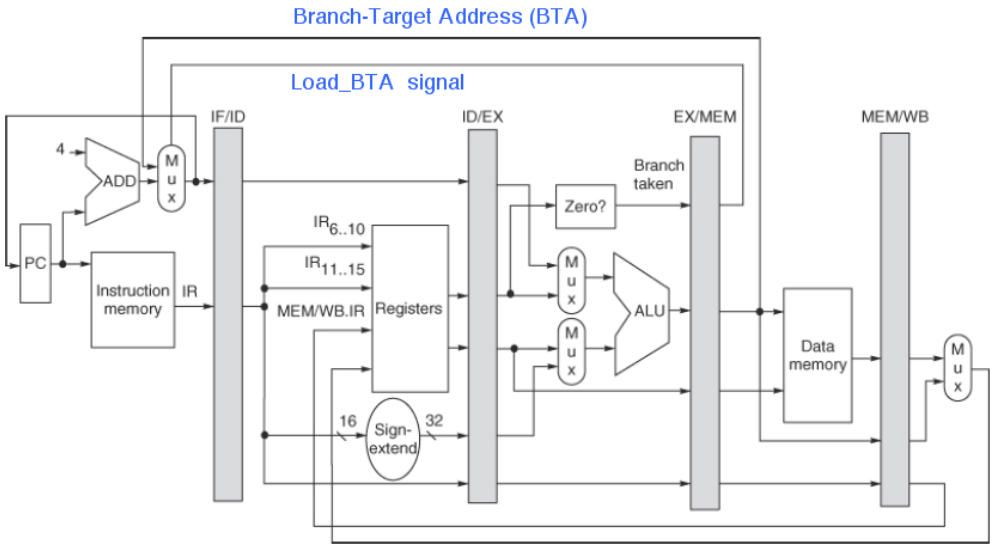
- Basic problem: we don't know whether to take the branch for several cc. By this time, CPU has fetched several incorrect instructions and started execution.
- Consider: BRANCH if A<0 to branch-target-address
- **EX stage computes A<0, MEM stage forwards branch outcome to IF stage, and the 'Load_BTA' signal arrives at the IF unit during clock cycle 4** (see below) -> the 3 instructions following the branch will be fetched by the IF_Unit !
- Leads to a 3 cc "**branch delay**" for the basic pipeline

	1	2	3	4	5	6	7	8
Branch	IF	ID	EX	MEM*	WB			
Branch+1		IF	ID	EX	MEM	WB		
Branch+2			IF	ID	EX	MEM	WB	
Branch+3				IF	ID	EX	MEM	WB
Branch+4					IF	ID	EX	MEM

(In this example, the 'LOAD_BTA' signal asserted when Branch instruction reaches the MEM stage in cc 4, and changes the PC before start of cc 5)

Original MIPS Datapath, with 3 cc Branch Delay

(class text – Fig C.22, pg. C-35)



- Recall ; the main problem is that we don't forward the 'LOAD_BTA' signal to the IF-unit until the MEM stage; this creates the 3 clock cycle delay

Performance Impact of 3 cc Branch Delay

- Branch delays incur big "slow down" affects on real machines.
- Consider an ideal machine with no branch delays; CPI = 1.0
- Consider a typical machine like the Pentium (x86 family) or PowerPC; up to 30 % of all instructions can be branches
- if each branch instruction causes 3 stall cycles, the new overall CPI is 1.9 !

  CPI = (0.7 * 1cc + 0.3 * 4cc) = 1.9 cc per instruction

- Ideal machine is about 90 % faster than real machine with branch stalls
- A good solution to branches can yield a speedup of up to 90 %, nearly a factor of 2.
- Solution: **add hardware to resolve branches earlier in pipeline**; as soon as possible, which is in the ID stage. See next slides.
- Even with this solution, we are still left with a 1 cc branch delay. For a typical machine with 30% branches, the new CPI is now:

  CPI = (0.7 * 1cc + 0.3 * 2cc) = 1.3 cc per instruction

- a real machine is about 30 % slower than the ideal machine, due to 1 cc branch penalty. 30 % is a lot in the highly competitive CPU market ! We must try to solve.

New CPU Datapath, with 1 cc Branch Delay (Fig. C.28, pg. C-42)

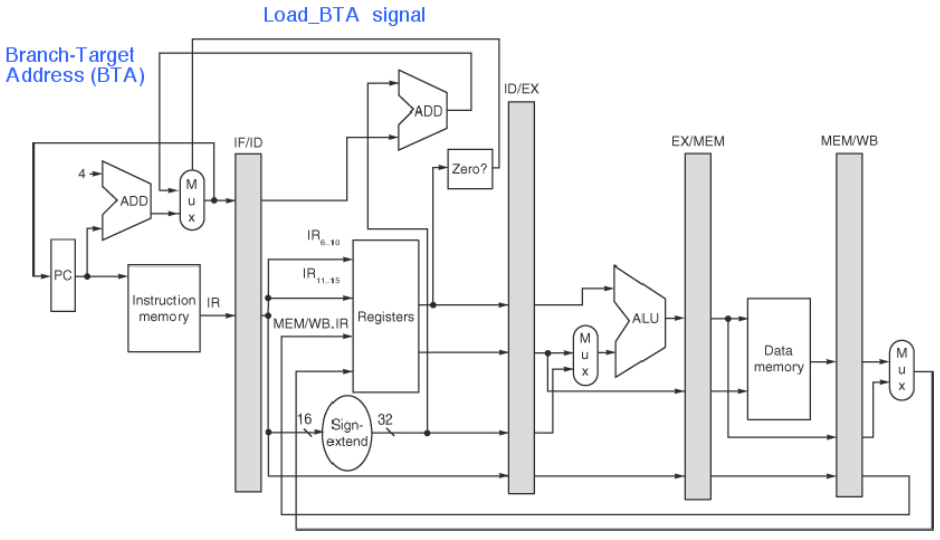


Figure C.28 The stall from branch hazards can be reduced by moving the zero test and branch-target calculation into the ID phase of the pipeline. Notice that we have made two important changes, each of which removes 1 cycle from the 3-cycle stall for

- Branch test now moved up into the ID stage; We can determine branch and update the PC before the ID stage finishes. Net result -> 1 cc branch delay penalty.

Reducing the 1 cc Branch Delay

- There are 4 basic ways to reduce the 1 cc branch penalty; done at compile time:

  - **Accept the penalty** - put 1 **No-Op** in the branch delay slot & accept 1cc penalty

  - **Delayed Branch** - Define instruction following the branch to be **always executed**; let compiler try to fill the " 1cc branch delay slot" with a useful instruction, otherwise fill it with a **No-Op**; approach used in MIPS machine

  - **Branch Prediction with Instruction Cancelling** - (also called **Cancelling Branches** in text) Every Branch Instruction includes **static** prediction in the Op-Code (taken / not taken). Hardware always executes instruction in branch delay slot; hardware "kills" instruction if branch is predicted incorrectly. For incorrectly predicted branches, 1 cc. is wasted - the instruction in the branch delay slot. Net affect: CPI for branch = 1 cc if predicted correctly, 2 cc if predicted incorrectly. If we can predict correctly often enough, we can improve performance here.

  - this technique essentially puts a conditionally executed instruction into the branch delay slot, so its easier for the compiler to find an instruction to put in the slot; it doesn't have to be guaranteed to 'safe', since it will be cancelled if branch prediction is wrong (Cancelling an instruction is relatively easy - simply disable its write-back)

Predict Not-Taken (class text, Fig. C.12, pg C-22)

Untaken branch instruction	IF	ID	EX	MEM	WB				
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				
Instruction i + 1		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure A.12 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

Untaken branch : CPI = 1 cc per branch instruction
Taken branch: CPI = 2cc per branch instruction
Average CPI: about 1.5 CPI, depending upon frequency of taken/untaken branches

Delayed Branch: Filling the 1 cc Branch Delay

• 3 ways a compiler can try to fill the 1 cc branch delay in **Delayed Branch** scheme:

- take an instruction from **before the branch** and move into branch delay slot
- take instruction **from branch target address** and move in branch delay slot
- take instruction from **"fall-through"** path and insert into slot

• in all 3 cases, the instruction must be "safe" to move into the slot, i.e., the instruction moved in the branch delay slot cannot change any final results even if the branch is predicted incorrectly (since we are not cancelling instructions on incorrect predictions)

• lets suppose we can fill the 1 cc branch delay slot 60 % of the time. For the typical machine considered earlier with 30% branches, the new CPI is now:

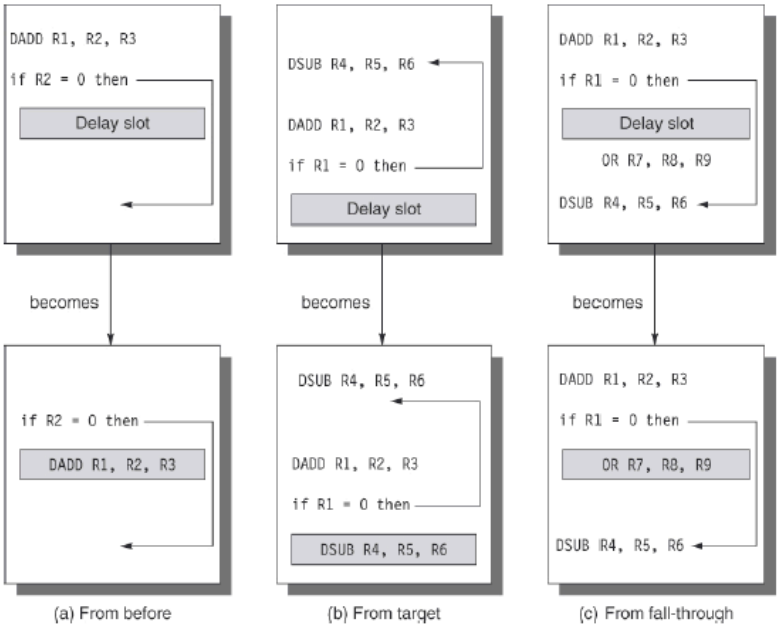
CPI = (0.7 * 1cc + 0.3 *(1cc*0.6 + 2cc*0.4)) = 1.12 cc per instruction

• The effective CPI of the branch is 1cc 60% of the time, and 2cc 40% of the time

• a real machine is about 12 % slower than the ideal machine with CPI = 1. This is better than the 30 % slowdown when we did not fill the branch delay.

• (Using Branch Prediction with Instruction Cancelling scheme, it is easier to put an instruction into the delay slot, since it doesn't have to be 'safe', since it will be cancelled in the branch is predicted incorrectly. For delayed branch, instr. must be safe)

Delayed Branch: Filling the 1 cc Branch Delay (Fig. C.14, pg. C-24)



• (a) is safe and is preferred. In (b), assume R4 is un-used before the branch. We save 1cc if we take the branch. In (c) assume R7 is unused before the branch. (Note: DADD R1… cannot be moved after branch due to control dependency on R1.)

Delayed Branch: Filling the 1 cc Branch Delay

(class text - Fig. 3.29, pp. 179, 2nd ed)

Scheduling strategy	Requirements	Improves performance when?
(a) From before branch	Branch must not depend on the rescheduled instructions.	Always.
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge program if instructions are duplicated.
(c) From fall through	Must be OK to execute instructions if branch is taken.	When branch is not taken.

FIGURE 3.29 Delayed-branch scheduling schemes and their requirements. The origin of the instruction being scheduled into the delay slot determines the scheduling strategy. The compiler must enforce the requirements when looking for instructions to schedule the delay slot. When the slots cannot be scheduled, they are filled with no-op instructions. In strategy (b), if the branch target is also accessible from another point in the program—as it would be if it were the head of a loop—the target instructions must be copied and not just moved.

Branch Prediction Rates

(class text – Fig. C.17, pg C-27)

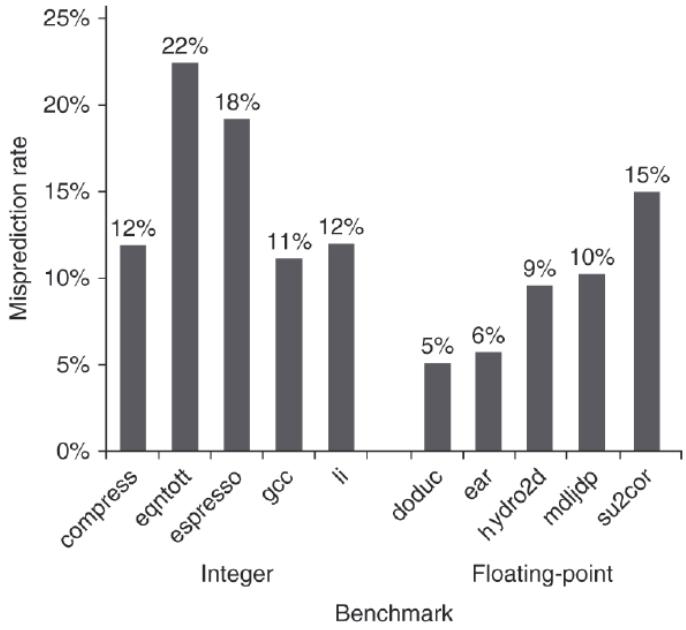


Figure C.17 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

Performance of Branch Schemes

(class text - pp. A24, 3rd edition)

$$speedup_{pipelining} = \frac{\text{pipeline depth}}{(1 + \text{pipeline stall cycles per instruction})} * \frac{\text{clock cycle}_{\text{unpipelined}}}{\text{clock cycle}_{\text{pipelined}}}$$

• assume clocks are constant. Pipeline stall cycles per instruction, from branches = Branch frequency * branch penalty

$$speedup_{pipelining} = \frac{\text{pipeline depth}}{(1 + \text{Branch frequency} * \text{Branch penalty})}$$

• for more detailed model, one could consider both conditional and unconditional branches, since the frequency and penalties are slightly different. For even more detail, one could consider prediction;

$$\text{pipeline stall cycles from branches} = \sum_{\substack{\text{conditional \&} \\ \text{unconditional branches}}} \text{Branch frequency}_{\text{type}} * \text{Branch penalty}_{\text{type}}$$

$$\text{pipeline stall cycles from branches} = \sum_{\substack{\text{conditional \&} \\ \text{unconditional branches\&} \\ \text{predicted branches}}} \text{Branch frequency}_{\text{type}} * \text{Branch penalty}_{\text{type}}$$

Options for Compiler Prediction

(class text - pp. 176, 2nd ed)

• compiler always predicts “taken” - right about 60 % of time, from SPEC benchmark

• predict backward branches as taken, forward branches as not taken - might be better or worse than first way

• predict on basis of execution profile gathered from earlier runs of same program; right about 85 % of the time (based on benchmarks)

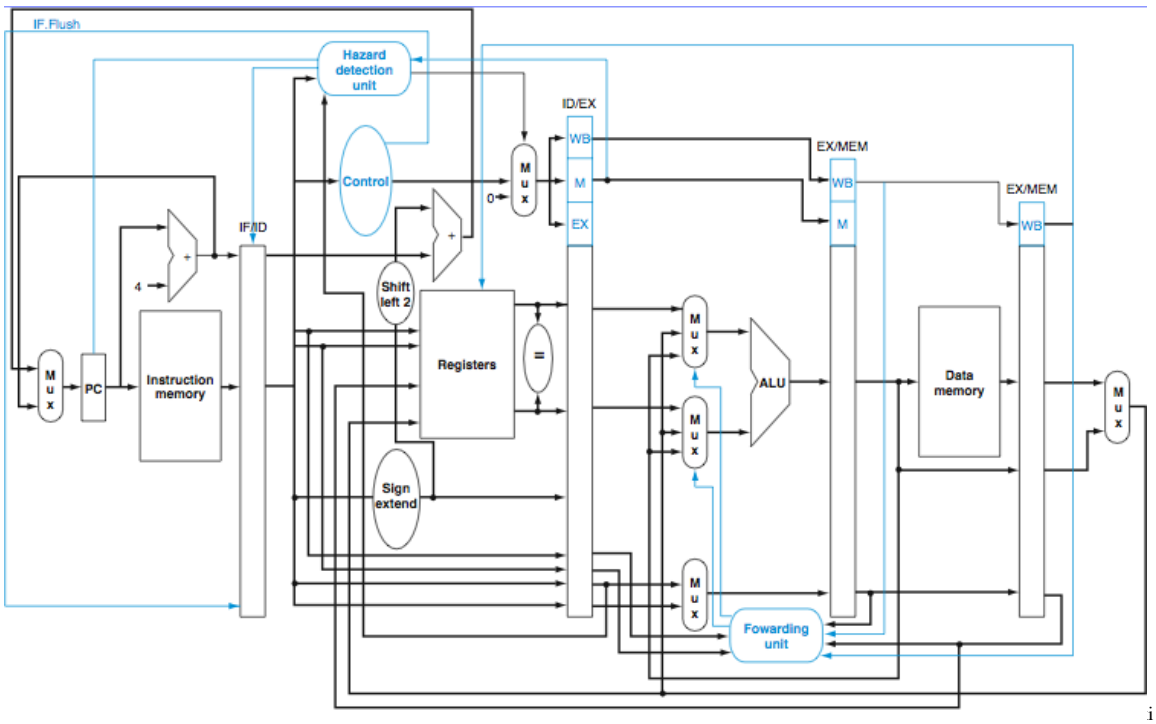
• Overall for MIPS, branches stall about 5 % of the time, Loads stall about 8 % of the time, after all compiler optimizations

Final CPU Datapath & Control Figure

- The introductory textbook introduces a "Flush" signal which goes to several pipeline stages, to discard the instruction in those stages
- This Flush signal creates the stalls in our space-time diagrams
- An asserted "Flush" signal causes the CPU to discard instructions in the affected pipeline stages, due to an unexpected event, such as an incorrectly predicted branch (if we are using branch prediction), or a data hazard which cannot be resolved through forwarding and which needs more time to clear

Final CPU Datapath & Control Figure

(introductory text - Fig 6.41, pg 427)



Exceptions

(class text – Appendix C, pg. C-43)

• "Exceptions" = "Interrupts", caused by several events:

- I/O device request
- invoking operating system from user program
- breakpoint (programmer requested interrupt)
- integer arithmetic under / overflow
- FP arithmetic anomaly (and there are many of these)
- Page Fault (read to Virtual Memory page, which is not in main memory)
- memory-protection violation
- execution of undefined instruction (CPU generates interrupt)

• exceptions are usually handled in a way that is completely "transparent" to the program being interrupted.

• If a CPU can handle an exception transparently, ie, save the state of the interrupted process, perform the exception routine, and then restore the state of the process and restart the process, we say the CPU is "restartable"

Exceptions (class text – pg. C-43)

• an "EXCEPTION" instruction causes the CPU to enter the Exception handling routine. On an Exception, the CPU could do this:

- save PC of faulting instruction at the end of the clock cycle
- force an EXCCEPTION inst. into the pipeline at next cc
- Until EXCEPTION inst. is taken, disable all writes for the faulting inst. and the insts.
- enter exception handling routine, which resolves exception and restarts execution at faulting instruction

• "precise exceptions" - when a pipeline can be stopped such that instructions before the faulting inst. are completed, and those after the faulting inst. did not change state and can be re-started from scratch

• Problem: consider FP-DIV inst. with a latency of 20 cc, which raises an exception at cc. 19; at this point, 19 other inst. have already been fetched into pipeline, and up to 14 of these have completed execution and written results, perhaps even overwriting the operands of the faulting FP instruction; can be very difficult to recover state before the faulting instruction

• many recent machines have 2 modes of operation, "precise" & "imprecise". In Alpha, PowerPC chip, & MIPS chips, precise mode can be 10x slower than imprecise modes

Exceptions in MIPS (class text, pg. C-43)

• to be precise, exceptions must be handled in order of occurence, assuming all instructions were executed one at a time (as in an unpipelined machine)

• consider:

	1	2	3	4	5	6
LD	IF	ID	EX	MEM**	WB	
ADD		IF *	ID	EC	MEM	WB

• * denotes page-fault at cc 2 which occurs first; ** denotes page-fault at cc 4;

• exceptions occur **out-of-order**: to be precise, pipeline must process LD exception 1st -> pipeline cannot simply handle exceptions as they occur in time

• **Solution:** every instruction in the pipeline has a status vector, which moves down the pipeline with the instruction. A set bit in the vector denotes an exception. Once an exception is indicated, all writes for that instruction are disabled.

• When an instruction enters the WB stage, its status vector is checked, and any set bits cause the exception to be handled; net result: exceptions are handled in order.

• in above example, LD exception will be handled first, in cc 5; the ADD exception will be handled later, after the 1st exception clears

MIPS Exceptions

(class text - Fig C.32, pg. C-48)

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure A.28 Exceptions that may occur in the MIPS pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

• up to 4 simultaneous exceptions can occur in MIPS, for 4 separate instructions in the pipeline

Multi-Cycle FP Operations

(class text – pg. C-51)

- The MIPS integer ALU is also called the "integer" unit
- a real MIPS machine has 4 separate EX units
  - Integer (the regular integer ALU that we have already been considering)
  - FP & Integer MULT
  - FP Add/substract
  - FP & Integer DIV
- typically the lower 3 units are not fully pipelined due to excessive hardware cost -> potential for structural hazards & data hazards
- results of computation must be recirculated within these units, reusing critical hardware, until they are finished -> "**multi-cycle**" operation
- net result is that we cannot issue a new instruction every clock cycle into these units; we must wait for earlier instructions to finish execution within these units
- time an instruction spends in a unit (after its issue clock cycle) = "**latency**"; time between successive instruction issues to the same unit = "**initiation rate**"
- the multi-cycle operations introduce many potential data hazards, since instructions finish "out-of-order"

Multi-Cycle Floating-Point Operations (Fig. C.33, pg. C-52)

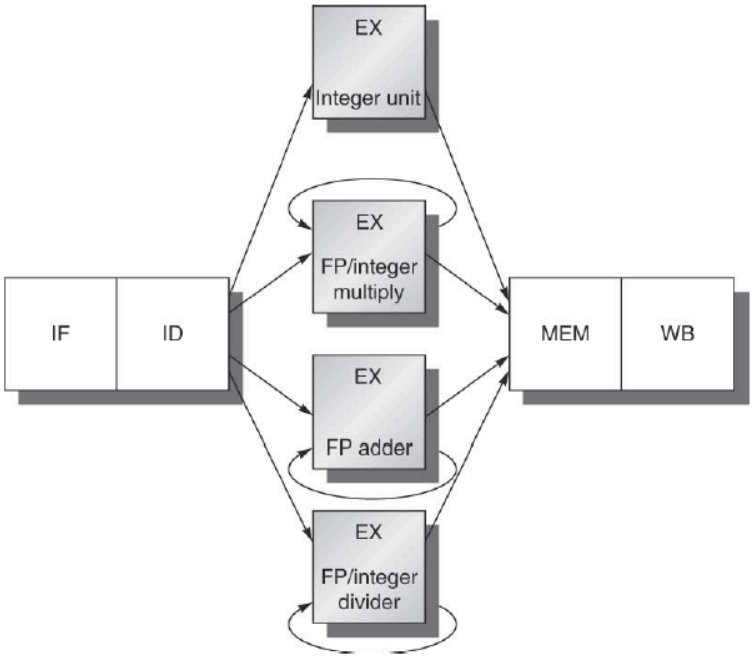


Figure C.33 The MIPS pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Option (1) for Adding FP Pipelining (Fig C.35, pg. C-54)

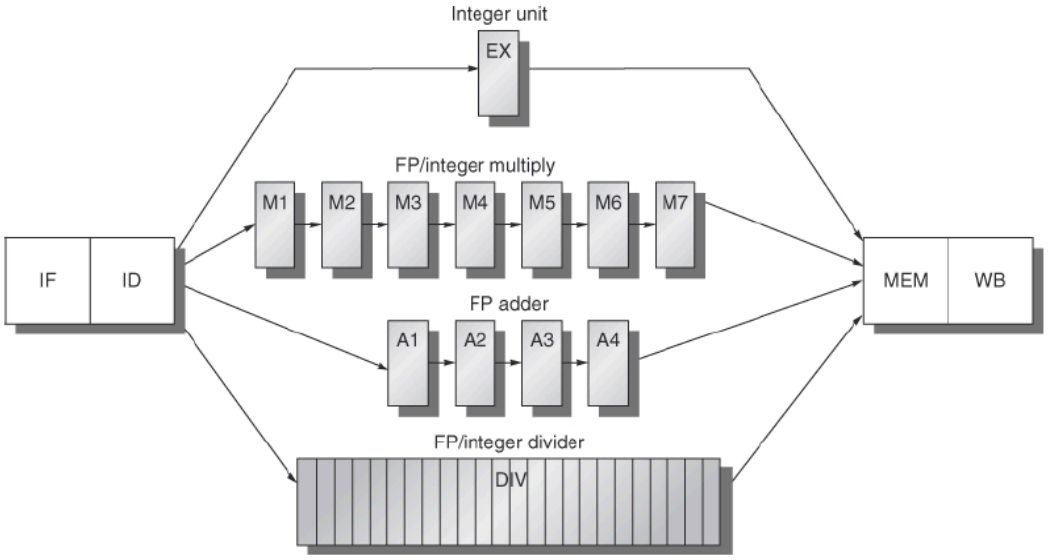
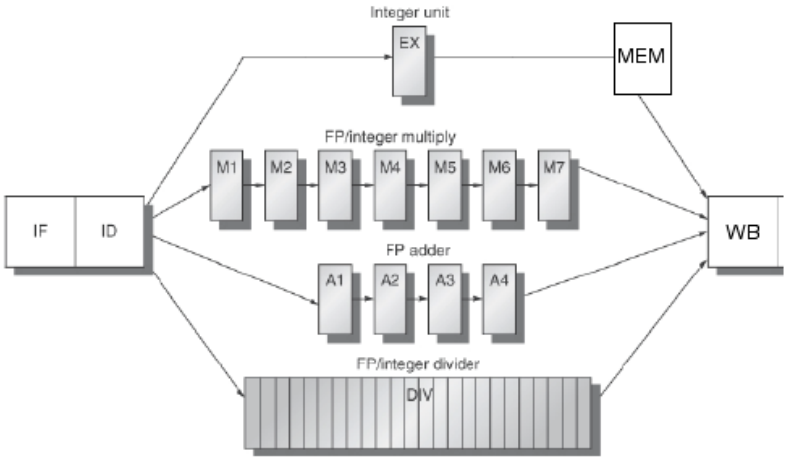


Figure C.35 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

Option (2) for Adding FP Pipelining (Fig C.35, pg. C-54)



- In this pipeline, only the integer instructions and LOAD/STORE instructions pass through the MEM stage
- The floating point instructions bypass the MEM stage, and go directly into the WRITE-BACK stage.
- In options (1) and (2), the computer designer must decide on how many instructions can enter the MEM stage or the WRITE-BACK stage, per clock cycle

FP Latency & Initiation Rate (class text – Fig. C.34, pg. C-53)

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure A.30 Latencies and initiation intervals for functional units.

- latency == ***additional*** # clock cycles (after given inst. enters unit) before result can be used; latency = the number of extra pipeline stages (above 1) needed for a unit
- results of integer ALU can be used in next cc, so its latency = 0
- in above table, Integer ALU, FP ADD, FP MULT are fully pipelined, so initiation rate = 1 inst. per clock cycle; FP DIV is not pipelined, since initiation rate = 1 inst. every 25 clock cycles
- above table allows 4 outstanding FP Adds, seven outstanding FP/INT Mults, one FP divide, for MIPS system on last slide
- number of FP register writes per clock cycle may be > 1 now

FP Pipeline Timing (class text – Fig. C-37, pg. C-55)

	Clock cycle number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Figure A.33 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The SD must be stalled an extra cycle so that its MEM does not conflict with the ADD.D. Extra hardware could easily handle this case.

- FP MULT unit has a 7 stage internal pipeline, with stages M1, ..., M7
- FP ADD unit has a 4 stage internal pipeline, with stages A1, ..., A4
- MUL.D stalls in cc 3 while operand F4 fetched in MEM stage
- ADD.D stalls in cc 6-11 while operand F0 computed in MULT unit
- SD.D stalls in cc 14,15 while operand F2 computed; stalls in cc 16 due to MEM conflict
- This slide illustrates that only 1 instruction can access cache memory in the MEM stage per clock cycle (memory access is expensive and takes time, and is a common structural hazard)

Handling the MEM and WB Stages

- slides 51 and 52 illustrate 2 options for the pipelines, when we add floating-point instructions
- for each option, there are several other design decisions that a computer-designer must make

- How many instructions can enter the MEM stage, per clock cycle ?
- How many instructions can access the cache memory, per clock cycle ?
- How many instructions can enter the WRITE-BACK stage, per clock cycle ?
- How many write-backs to the registers are allowed, per clock cycle ?

- Allowing more functionally typically requires more hardware design time, more test time, and more transistors, which all increase the chip costs
- Different chip manufacturers typically make different design decisions, so there is no single answer
- These design decisions result in different machine performances, and often differentiate the machines produced by companies like INTEL, AMD, MIPS and ARM

- We we allow multiple-issue (or super-scalar) designs, then on average multiple instructions must enter the MEM and WB per clock cycle, and the same design decisions exist

FP Hazards and Forwarding (class text – pg. C-55)

- complications:
  - structural hazards exist since DIV unit is not pipelined
  - since instructions have varying run times, there can be multiple writes to registers per clock cycle (if two instructions finish at same time & must write)
  - '**WW hazards'** are now possible, since inst. no longer reach WB in order.
  - instructions complete in different order than issued, causing problems with exceptions
  - because of longer latencies, stalls for **'WR hazards'** will be more frequent and will last longer

Note on Data-Hazards due to out-of-order execution:
- a **WW hazard** ("**Write-Write**") happens when 2 writes finish in wrong order
- a **WR hazard** ("**Write-Read**") happens when the write should be followed by the Read, but the 2 instructions finish in wrong order
- these hazards can be avoided by adding stalls, to force in-order execution

FP - Multiple Register Writes May Occur Often

(class text – Fig. C.28, pg. C-56)

	Clock cycle number										
Instruction	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

Figure A.34 Three instructions want to perform a write back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, since an earlier divide in the FP unit could also finish on the same clock. Note that although the MUL.D, ADD.D, and L.D all are in the MEM stage in clock cycle 10, only the L.D actually uses the memory, so no structural hazard exists for MEM.

- In a single-issue machine, afor maxim,um-performancemultiple instructions may enter the
- it may not be worthwhile, dding hardware to allow for multiple writes, since this event may be infrequent

FP Hazard Detection Logic (class text – App. C)

- assume INT & FP instructions have their own registers; so they operate relatively independently (except for occassional data moves between them)
- to avoid hazards, the following 3 checks can be performed in the ID stage
- (1) check for structural hazard; make sure the FP unit is ready to accept a new operation
- (2) check for '**WR hazard**': (the write of the operand should happen first, followed by the read of the operand); make sure the operand register is not listed as a pending destination register for any instruction in the FP pipeline which hasn't finished writing yet; otherwise, stall until the hazard clears; many comparisons must be made in parallel here
- (3) Check for '**WW hazard**'; determine if any instruction in the FP pipeline has the same destination register as the instruction in the ID stage; if so, stall the instruction in the ID stage until the hazard clears

- when an instruction moves from the ID stage to some unit, we say that the instruction "**issues**", otherwise it stalls

Notes

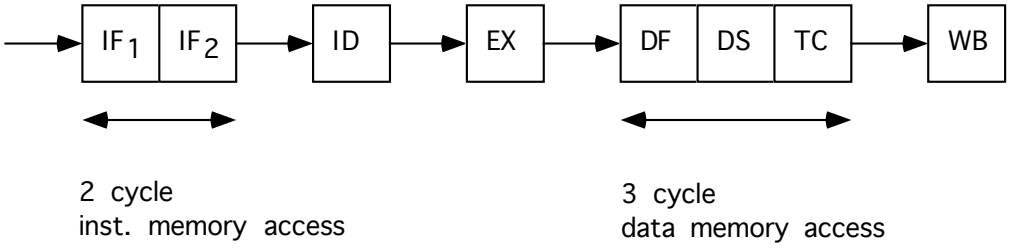
WW hazard example:

DIVD F0,F2,F4
ADDD F0,F2,F4

The ADDD will attempt to write F0 before DIVD and be forced to stall on WB until The hazard clears

-a read/write hazard is never a problem in a single-issue static scheduled pipeline, because given 2 instructions j and j+1, instruction j must have both its operands available when it starts execution (otherwise it stalls), and once it starts execution, it not longer needs its operands and the next instruction can overwrite those operands

8 Stage MIPS R4000 Pipeline (class text – Pg. C-61)



- real caches tend to be slower than CPUs, so MIPS R4000 spreads cache accesses over multiple clock cycles, leading to a deeper (8 stage) pipeline and higher clock rates
- instruction fetch uses 2 clock cycles IF1, IF2 - allows 2 cc for cache to respond; ID stage now also checks for cache hit or miss; a miss will stall the pipeline
- data memory access uses 3 clock cycles DF, DS, TC - allows 2 cc for data cache to respond (Data Access -First, Data Access - Second), along with 1 cc to check if returned result is valid or not (Tag-Check)
- instruction and data caches must also be pipelined too; the instruction cache must accept a new operation in the first stage IF1, while still working on the read in IF2
- the same pipelining applies for the data cache

Basic/Intermediate Pipeline Summary

- the classic 5 stage pipeline is still used in low to mid-range embedded processors
- the MIPS R4000 8-stage pipeline is a 64 bit embedded processor, used for example in the Nintendo-64 game systems and in laser printers
- there is an NEC version of this chip without the Floating Point hardware; FP is done through software