Predict Not Taken

Branch	IF	ID	EX	MEM	WB			
I+1		IF	ID	EX	MEM	WB		
I+2			IF	ID	EX	MEM	WB	
I+3				IF	ID	EX	MEM	WB

branch not taken

Branch	IF	ID	EX	MEM	WB			
I+1		IF	(bubble)	(bubble)	(bubble)	(bubble)		
Branch Target			IF	ID	EX	MEM	WB	
T+1				IF	ID	EX	MEM	WB

branch taken

Delayed Branch

Branch	IF	ID	EX	MEM	WB			
I+1 (delay slot)		IF	ID	EX	MEM	WB		
I+2			IF	ID	EX	MEM	WB	
I+3				IF	ID	EX	MEM	WB

branch not taken

Branch	IF	ID	EX	MEM	WB			
I+1 (delay slot)		IF	ID	EX	MEM	WB		
Branch Target			IF	ID	EX	MEM	WB	
T+1				IF	ID	EX	MEM	WB

branch taken

Filling the delay slot (e.g., in the compiler)

Can be done when?
Improves performance when?

```
lw R1, 10000(R7)
add R5, R6, R7
beqz R5, label:
[          ]
sub R8, R1, R3
add R4, R8, R9
and R2, R4, R8



label: lw  R2, 1024(R8)
```


Problems filling delay slot

- 1. need to predict direction of branch to be most effective
- 2. limited by correctness restrictions
- correctness restriction can be removed by a canceling branch

  branch likely or branch not likely

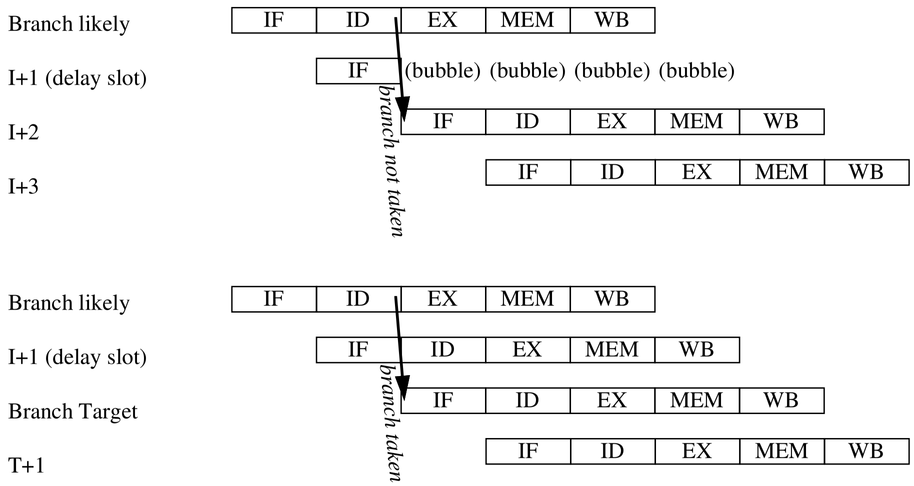
  e.g.,

  beqz likely

  delay slot instruction ← squashed/nullified/canceled if branch not taken

  fall-through instruction

Branch Likely



Delay Slot Utilization

- 18% of delay slots left empty
- 11% of delay slots (1) use canceling branches and (2) end up getting canceled

Branch Performance

CPI = BCPI + pipeline stalls from branches per instruction
= 1.0 + branch frequency * branch penalty
assume 20% branches, 67% taken:

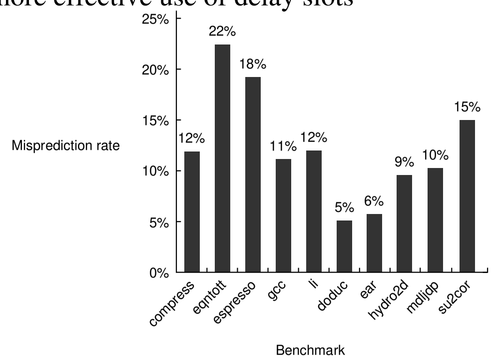
branch scheme	taken penalty	not taken penalty	CPI
stall			
predict taken			
predict not taken			
delayed branch			

Delay Slots, the scorecard

- Pros
- Cons

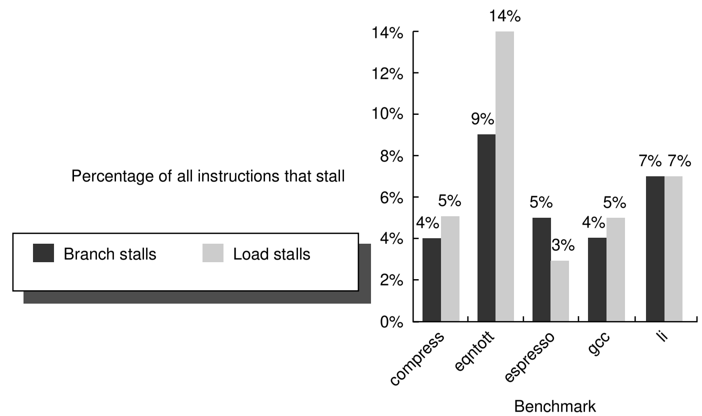
Static Branch Prediction

- Static branch prediction takes place at compile time, dynamic branch prediction during program execution
- static bp done by software, dynamic bp done in hardware
- Static branch prediction enables
  - more effective code scheduling around hazards (how?)
  - more effective use of delay slots



DLX Integer Pipeline Performance

- Only stalls for load hazards and branch hazards, both of which can be reduced (but not eliminated) by software



But now, the real world interrupts...

- Pipelining is not as easy as we have made it seem so far...
  - interrupts and exceptions
  - long-latency instructions

Exceptions and Interrupts

- Transfer of control flow (to an exception handler) without an explicit branch or jump
- are often unpredictable
- examples
  - I/O device request
  - OS system call
  - arithmetic overflow/underflow
  - FP error
  - page fault
  - memory-protection violation
  - hardware error
  - undefined instruction

Classes of Exceptions

- synchronous vs. asynchronous
- user-initiated vs. coerced
- user maskable vs. nonmaskable
- within instruction vs. between instructions
- resume vs. terminate

when the pipeline can be stopped just before the faulting instruction, and can be restarted from there (if necessary), the pipeline supports *precise exceptions*

Basic Exception Methodology

- turn off writes for faulting instruction and following
- force a trap into the pipeline at the next IF
- save the PC of the faulting instruction (not quite enough for delayed branches)

Exceptions Can Occur In Several Places in the pipeline

- IF -- page fault on memory access, misaligned memory access, memory-protection violation
- ID -- illegal opcode
- EX -- arithmetic exception
- MEM -- page fault, misaligned access, memory-protection
- WB -- none

(and, of course, asynchronous can happen anytime)

LW	IF	ID	EX	MEM	WB		
ADD		IF	ID	EX	MEM	WB	
SUB			IF	ID	EX	MEM	WB

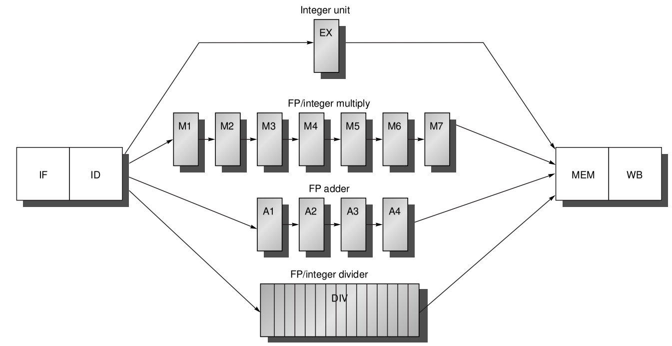
Simplifying Exceptions in the ISA

- Each instruction changes machine state only once
  - autoincrement
  - string operations
  - condition codes
- Each instruction changes machine state at the end of the pipeline (when you know it will not cause an exception)

Handling Multicycle Operations

- Unrealistic to expect that all operations take the same amount of time to execute
- FP, some memory operations will take longer
- This violates some of the assumptions of our simple pipeline

Multiple Execution Pipelines



FU	Latency	Initiation interval
Integer	0	1
Memory	1	1
FP add	3	1
FP multiply	6	1
FP divide	24	24

New problems

- structural hazards
  - divide unit
  - WB stage
- WAW hazards are possible (but probably rare)
- out-of-order completion
- RAW hazards much more frequent

structural hazards and WAW hazards

- structural hazards
  - divide unit
  - WB stage

ADDD	IF	ID	A1	A2	A3	A4	MEM	WB
...		IF	ID	EX	MEM	WB		
...			IF	ID	EX	MEM	WB	
LD				IF	ID	EX	MEM	WB

- WAW hazards

ADDD F8, ...	IF	ID	A1	A2	A3	A4	MEM	WB
LD F8, ...		IF	ID	EX	MEM	WB		

Hazard Detection in the ID stage

- An instruction can only *issue* (proceed past the ID stage) when:
  - there are no structural hazards (divide unit is free, WB port will be free when needed)
  - no RAW data hazards
  - no WAW hazards with instructions in long pipes

o-o-o completion and precise exceptions

MULTD	IF	ID	M1	M2	M3	M4	M5	M6	M7 ...
ADDD		IF	ID	A1	A2	A3	A4	MEM	WB
LD			IF	ID	EX	MEM	WB		

- Two solutions (with slight variations)
  - imprecise interrupts (possibly with software fixes to make it look precise)
  - buffer results, forcing in-order WB's

MIPS R4000 Pipeline

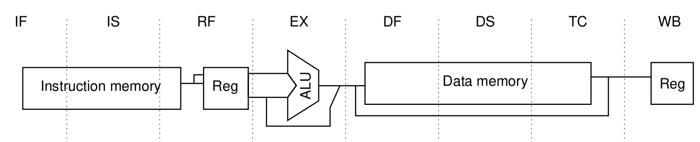
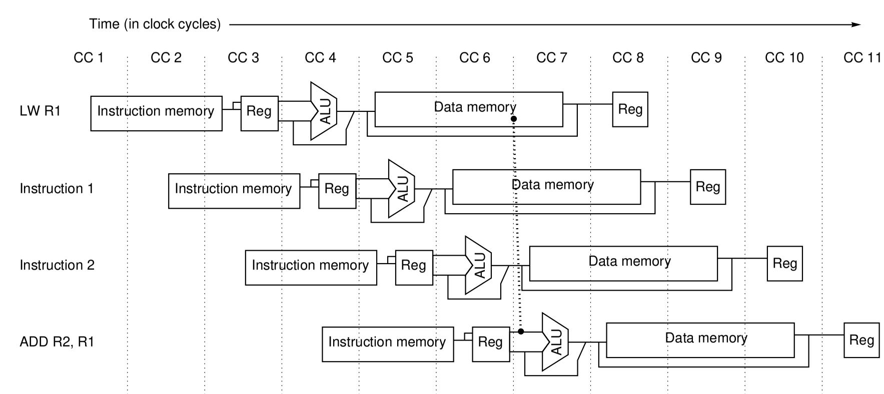


FIGURE 3.50 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches.

- scalar, superpipelined
  - **IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.**
  - ***IS–second half of access to instruction cache.***
  - **RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.**
  - **EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.**
  - **DF–data fetch, first half of access to data cache.**
  - ***DS–second half of access to data cache.***
  - ***TC–tag check, determine whether the data cache access hit.***
  - **WB–write back for loads and register-register operations.**

R4000 Data Load Hazard



- Is there an integer arithmetic data hazard?

R4000 Data Load Hazard

lw R1,...	IF	IS	RF	EX	DF	DS	TC	WB
add R3, R1, R2		IF	IS	RF	(stall)	(stall)	EX	DF

- 2-cycle load delay

R4000 Branch Hazard

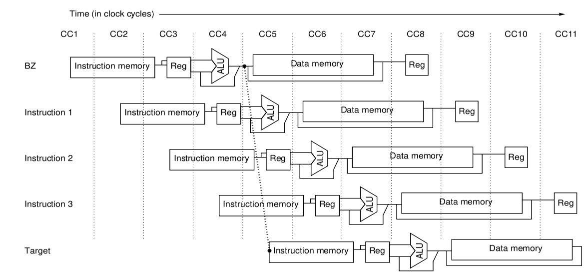


FIGURE 3.53 The basic branch delay is three cycles, since the condition evaluation is performed during EX.

- predict not taken, branch delay slot
- not taken -> no penalty (unless branch likely or no delay slot instruction)
- taken -> 2 stall cycles if delay slot instruction used

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

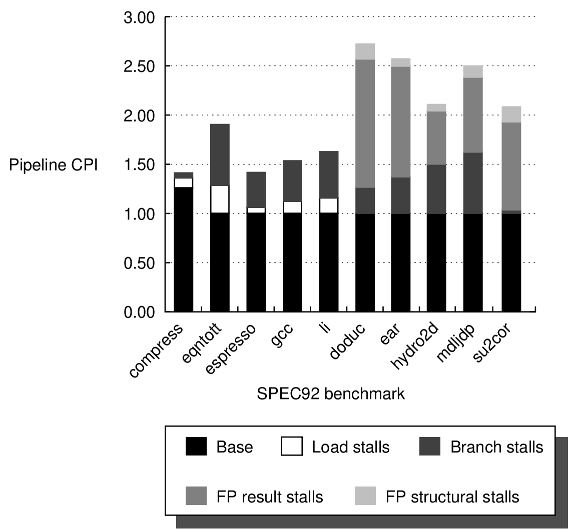
MIPS FP Pipe Stages

FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	…
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R, D+R, D+A, D+R, A, R		
Square root	U	E	$(A+R)^{108}$		…	A	R		
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M	*First stage of multiplier*	*A*	*Mantissa ADD stage*
N	*Second stage of multiplier*	*D*	*Divide pipeline stage*
R	*Rounding stage*	*E*	*Exception test stage*
S	*Operand shift stage*		
U	*Unpack FP numbers*		

R4000 Performance



Key Points

- Data Hazards can be significantly reduced by forwarding
- Branch hazards can be reduced by early computation of condition and target, branch delay slots, branch prediction
- Data hazard and branch hazard reduction require complex compiler support
- Exceptions are hard, precise exceptions are really hard
- variable-length instructions introduce structural hazards, WAW hazards, more RAW hazards