Revisiting Branch Hazard Solutions
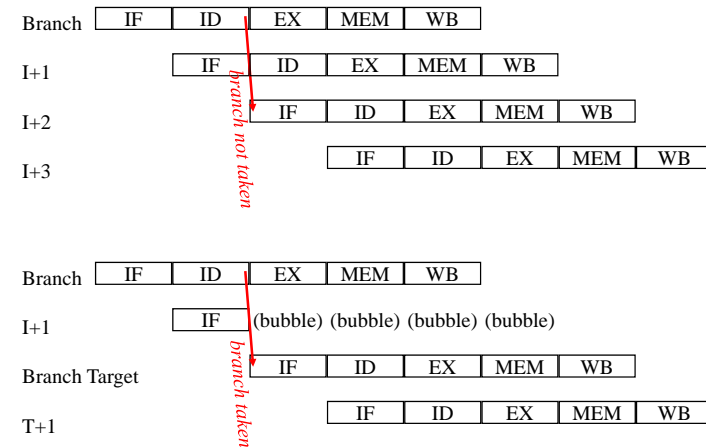
- Stall
- Predict Not Taken
- Predict Taken
- Branch Delay Slot

Predict Not Taken
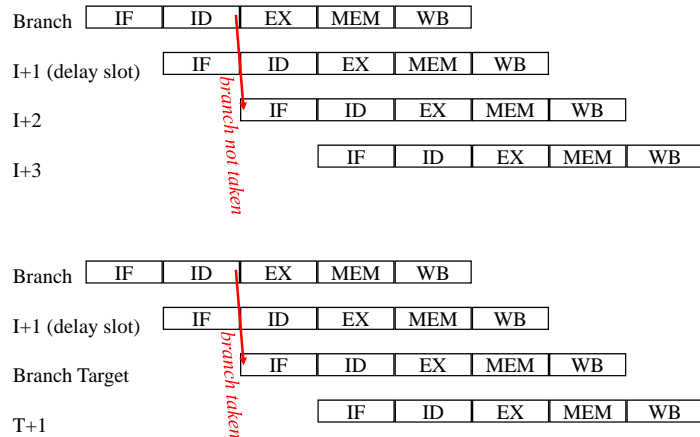
Branch	IF	ID	EX	MEM	WB
I+1		IF	ID	EX	MEM	WB
I+2			IF	ID	EX	MEM	WB
I+3				IF	ID	EX	MEM	WB

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, R1
beqz R5, label:

sub R8, R1, R3
add R4, R8, R9
and R2, R4, R8



label: add  R2, R5, R8
```

Problems filling delay slot

1. need to predict _____ of branch to be most effective
2. limited by _____ restriction

- _____ restriction can be removed by a canceling branch

  branch likely (or branch not likely???)
  e.g.,
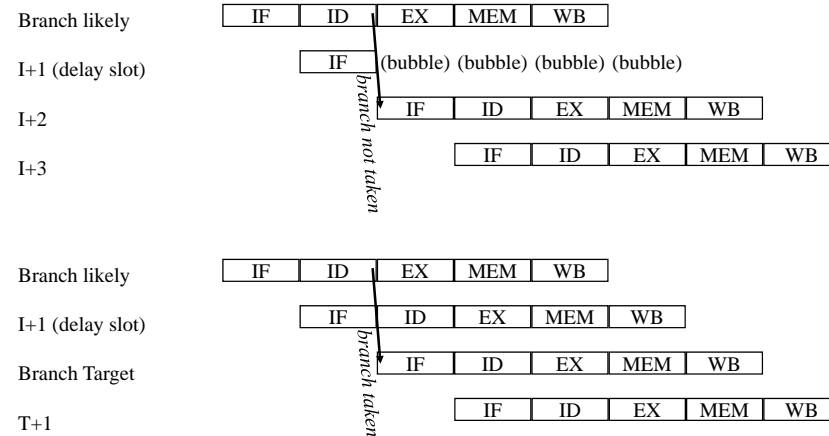 *beqz likely*
  delay slot instruction ← squashed/nullified/canceled if branch not taken
  fall-through instruction

Branch Likely

| Branch likely | IF | ID | EX | MEM | WB |
| I+1 (delay slot) | | IF | (bubble) (bubble) (bubble) (bubble) |

branch not taken

I+2 — IF | ID | EX | MEM | WB
I+3 — IF | ID | EX | MEM | WB

Branch likely — IF | ID | EX | MEM | WB
I+1 (delay slot) — IF | ID | EX | MEM | WB

branch taken

Branch Target — IF | ID | EX | MEM | WB
T+1 — IF | ID | EX | MEM | WB

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

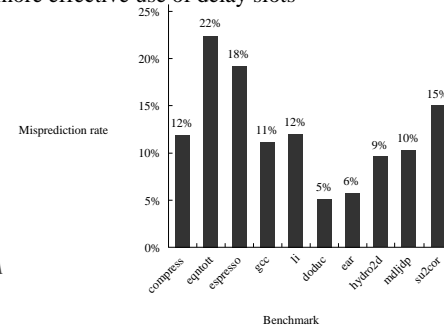- Pros

- 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
- How to make static branch predictions?
- Static branch prediction enables
  - more effective code scheduling around hazards (how?)
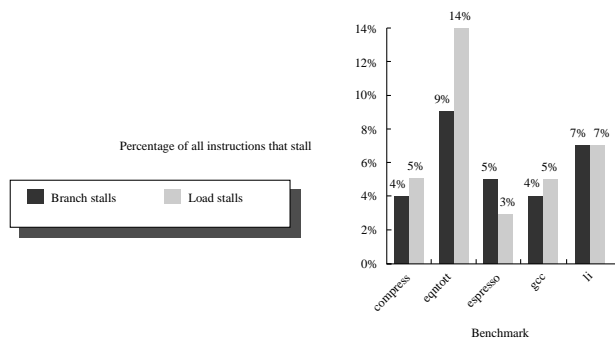 - more effective use of delay slots

MIPS 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
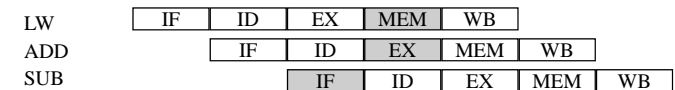
Simplifying Exceptions in the ISA

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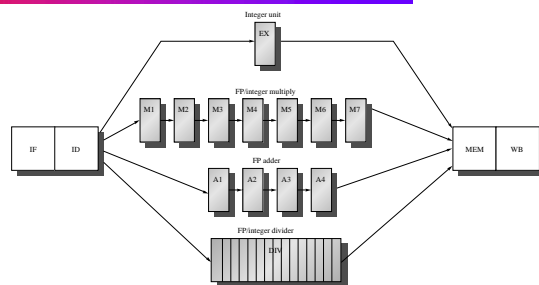
Dean Tullsen

Handling Multicycle Operations

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

Dean Tullsen

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

Dean Tullsen

New problems

- structural hazards
  – divide unit
  – WB stage
- WAW hazards are possible
- out-of-order completion
- WAR hazards still not possible

Dean Tullsen

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 (that forwarding can't handle)
  - no WAW hazards with instructions in long pipes

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