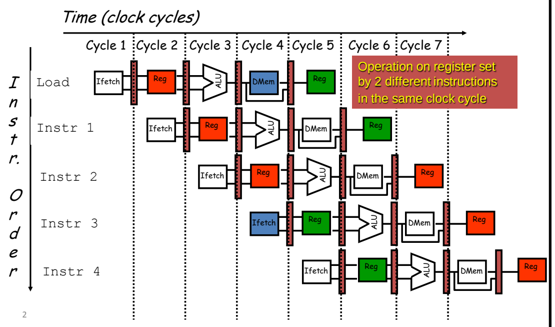
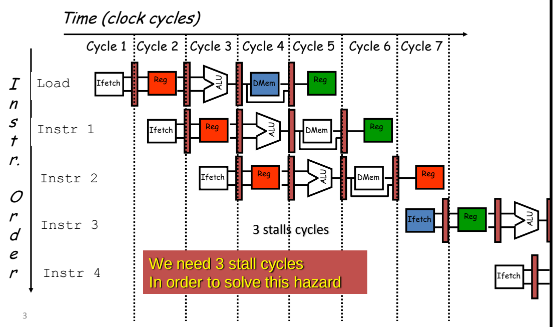
Pipeline Hazards

Register File/Structural Hazards



Register File/Structural Hazards

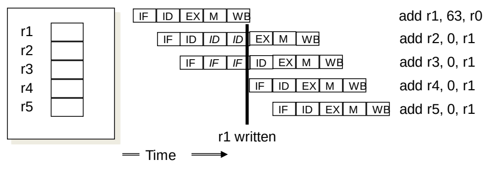


Pipelining is Not That Easy for Computers

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - **Structural hazards**: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - **Data hazards**: Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - **Control hazards**: Arise from the pipelining of conditional branches and other instructions that change the PC
- A possible solution is to "stall" the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline
  - You can always resolve pipeline hazards by waiting

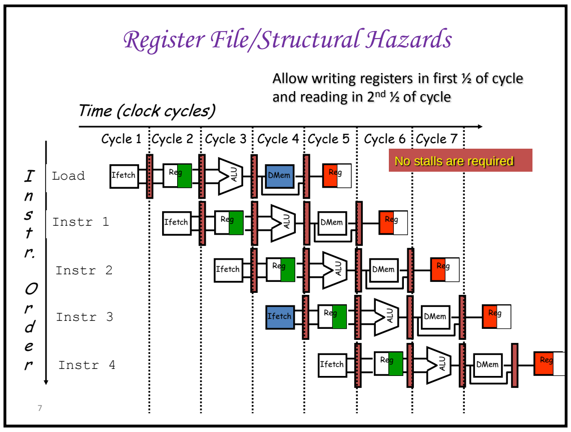
Stalling for Data Hazards

- Operation
  - First instruction progresses unimpeded
  - Second waits in ID until first hits WB (***2 stall cycles***)
  - Third waits in IF until second allowed to progress



Structural Hazards

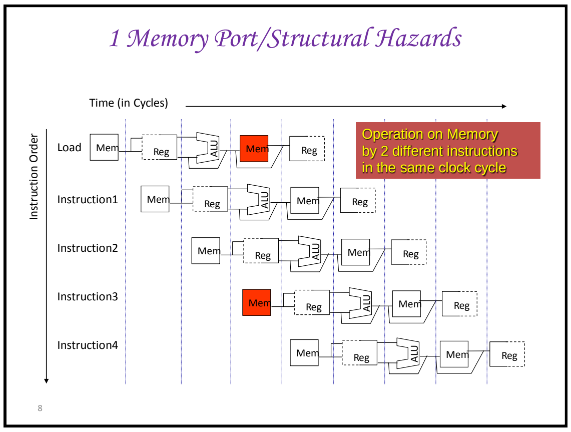
- In pipelined processors, overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, a structural hazard has occurred, for example:
  - when a machine has only one register file write port
  - or when a pipelined machine has a shared single-memory pipeline for data and instructions.



Register File/Structural Hazards

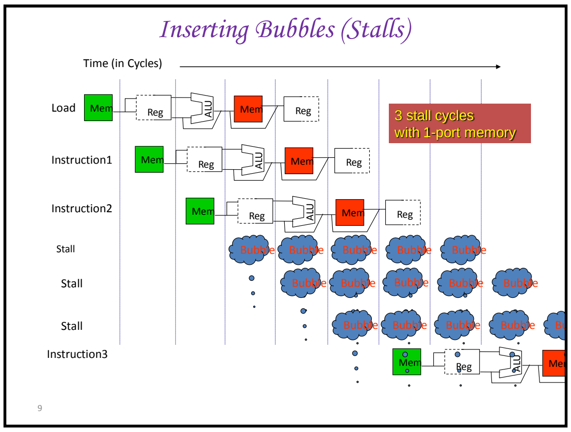
Allow writing registers in first ½ of cycle and reading in 2nd ½ of cycle

7



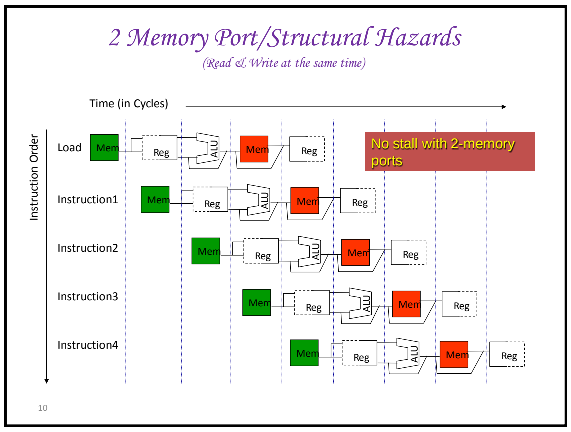
1 Memory Port/Structural Hazards

8



Inserting Bubbles (Stalls)

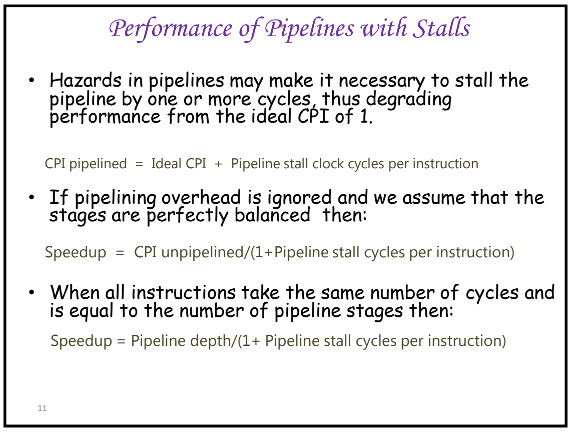
9



2 Memory Port/Structural Hazards

(Read & Write at the same time)

10



Performance of Pipelines with Stalls

- Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles, thus degrading performance from the ideal CPI of 1.

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction

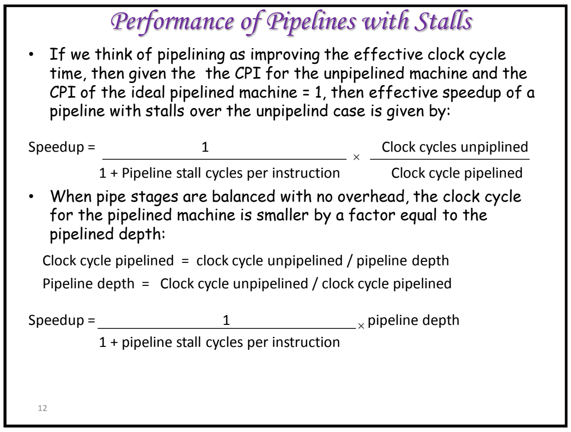
- If pipelining overhead is ignored and we assume that the stages are perfectly balanced then:

Speedup = CPI unpipelined/(1+Pipeline stall cycles per instruction)

- When all instructions take the same number of cycles and is equal to the number of pipeline stages then:

Speedup = Pipeline depth/(1+ Pipeline stall cycles per instruction)

11



Performance of Pipelines with Stalls

- If we think of pipelining as improving the effective clock cycle time, then given the the CPI for the unpipelined machine and the CPI of the ideal pipelined machine = 1, then effective speedup of a pipeline with stalls over the unpipelind case is given by:

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

- When pipe stages are balanced with no overhead, the clock cycle for the pipelined machine is smaller by a factor equal to the pipelined depth:

Clock cycle pipelined = clock cycle unpipelined / pipeline depth

Pipeline depth = Clock cycle unpipelined / clock cycle pipelined

$$\text{Speedup} = \frac{1}{1 + \text{pipeline stall cycles per instruction}} \times \text{pipeline depth}$$

12

Speed Up Equation for Pipelining

Viewpoint: Improving clock cycle time ($CPI_{unpipelined}$ =1).

$$Speedup = \frac{CPI_{unpipelined}}{CPI_{pipelined}} \times \frac{CycleTime_{unpipelined}}{CycleTime_{pipelined}}$$

$$= \frac{1}{1+CPI_{stall}} \times \frac{CycleTime_{unpipelined}}{CycleTime_{pipelined}}$$

$$Depth_{pipelined} = \frac{CycleTime_{unpipelined}}{CycleTime_{pipelined}}$$

Pipeline stall cycles per instruction

$$Speedup = \frac{Depth_{pipelined}}{1+CPI_{stall}}$$

13

Speed Up Equation for Pipelining

Viewpoint: Decreasing CPI (ignoring the cycle time overhead of pipelining).

$$CPI_{pipelined} = CPI_{ideal} + CPI_{stall}$$

$$Speedup = \frac{CPI_{unpipelined}}{CPI_{ideal}+CPI_{stall}}$$

$$= \frac{Depth_{pipelined}}{1+CPI_{stall}}$$

14

Example: Dual-port vs. Single-port Memory

- Machine A: Dual ported memory (0 stalls)
- Machine B: Single ported memory (1 stall), but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads/stores are 40% of instructions executed

```
SpeedUp_A = (Pipeline Depth/(1 + 0)) x (CycleTime_unpipe/CycleTime_pipe)
          = Pipeline Depth
SpeedUp_B = Pipeline Depth/(1 + 0.4 x 1)
                 x (CycleTime_unpipe/(CycleTime_unpipe / 1.05)
             = (Pipeline Depth/1.4) x  1.05
             = 0.75 x Pipeline Depth
```

$SpeedUp_A$ / $SpeedUp_B$ = Pipeline Depth/(0.48 x Pipeline Depth) = 1.33

- Machine A is 1.33 times faster

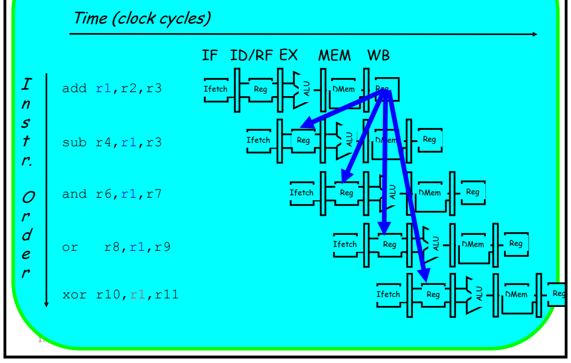
15

Pipeline Hazards

- Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  - ***Structural hazards*:** Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - ***Data hazards*:** Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - ***Control hazards*:** Arise from the pipelining of conditional branches and other instructions that change the PC
- **We can always resolve hazards by waiting**

16

Data Hazard on R1



17

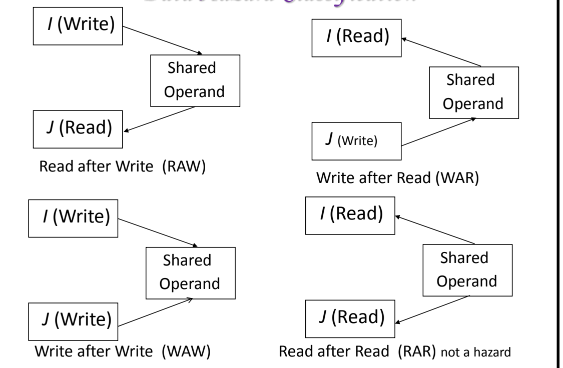
Data Hazard Classification

Given two instructions *I*, *J*, with *I* occurring before *J* in an instruction stream:

- **RAW (read after write):** *A true data dependence*
  J tried to read a source before *I* writes to it, so *J* incorrectly gets the old value.
- **WAW (write after write):** *A name dependence*
  J tries to write an operand before it is written by *I*
  The writes end up being performed in the wrong order.
- **WAR (write after read):** *A name dependence*
  J tries to write to a destination before it is read by *I*, so *I* incorrectly gets the new value.
- **RAR (read after read):** (Usually) not a hazard

18

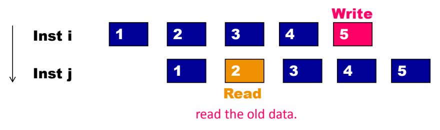
Data Hazard Classification



Data Hazards Present in Current MIPS Pipeline

- **Read after Write (RAW) Hazards:** Possible?
  - Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.
  - Yes possible, when an instruction requires an operand generated by a preceding instruction with distance less than four.

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

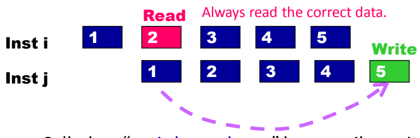


  - **Resolved by:**
    - Forwarding or Stalling.

Data Hazards Present in Current MIPS Pipeline

- **Write After Read (WAR)** – not possible
  Error if Instr_J tries to write operand *before* Instr_I reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

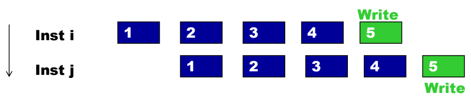


- Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1".

Data Hazards Present in Current MIPS Pipeline

- **Write After Write (WAW) - not possible**
- Error if Instr_J tries to write operand *before* Instr_I writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```



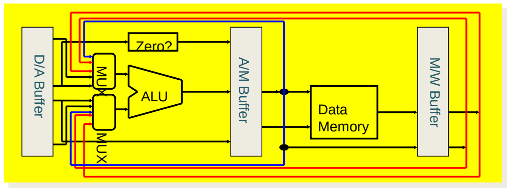
- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".

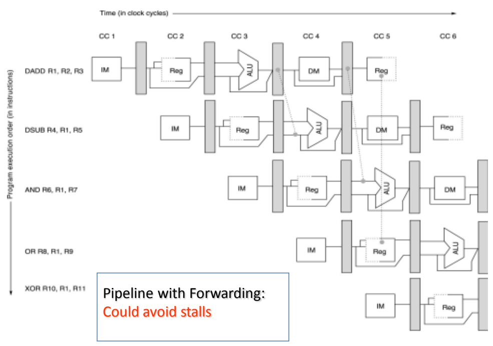
Data Hazards

- Solutions for Data Hazards
  - Stalling
  - Forwarding:
    - connect new value directly to next stage
  - Reordering, renaming, …

Minimizing Data Hazard Stalls by Forwarding

- Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.
- Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU, input register, memory write port etc.)

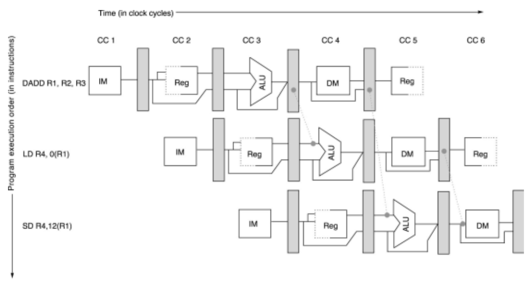




Pipeline with Forwarding:
Could avoid stalls

A set of instructions that depend on the DADD result uses forwarding paths to avoid the data hazard

Load/Store Forwarding Example



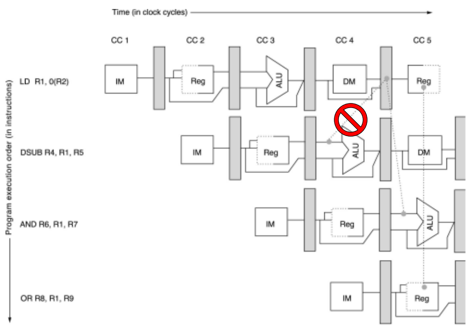
Forwarding of operand required by stores during MEM

Data Hazards Requiring Stall Cycles

- In some code sequence cases, potential data hazards cannot be handled by bypassing. For example:

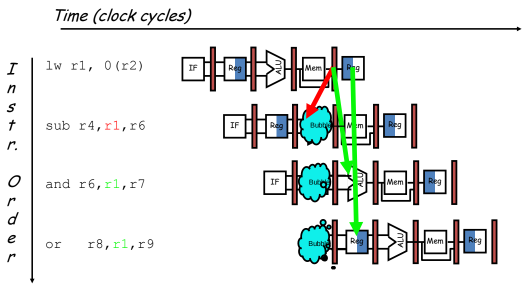
```
L.D     R1, 0 (R2)
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

- The L.D (load double word) instruction has the data in clock cycle 4 (MEM cycle).
- The DSUB instruction needs the data of R1 in the beginning of that cycle.
- Hazard prevented by hardware pipeline interlock causing a stall cycle.



The load instruction can bypass its result to the AND and OR instructions without stalls. But it cannot bypass the result to DSUB without a stall. We cannot forward to "negetive time".

Data Hazard Even with Forwarding



```
lw r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or   r8,r1,r9
```

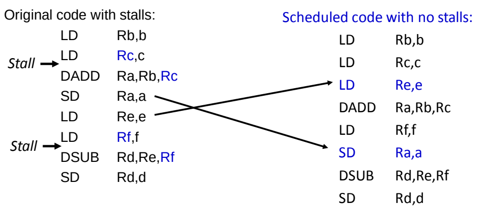
Compiler Instruction Scheduling Example

- For the code sequence:

  a = b + c
  d = e - f

  a, b, c, d ,e, and f are in memory

- Assuming a load require a delay of one extra clock cycle before its result is available for the next instruction's ALU input, the following code or pipeline compiler schedule eliminates stalls:



Original code with stalls:			Scheduled code with no stalls:	
	LD	Rb,b	LD	Rb,b
Stall →	LD	Rc,c	LD	Rc,c
	DADD	Ra,Rb,Rc	LD	Re,e
	SD	Ra,a	DADD	Ra,Rb,Rc
	LD	Re,e	LD	Rf,f
Stall →	LD	Rf,f	SD	Ra,a
	DSUB	Rd,Re,Rf	DSUB	Rd,Re,Rf
	SD	Rd,d	SD	Rd,d

Pipeline Hazards

- Hazards reduce the ideal speedup gained from pipelining and are classified into three classes:
  - ***Structural hazards*:** Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - ***Data hazards*:** Arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - ***Control hazards*:** Arise from the pipelining of conditional branches and other instructions that change the PC
- **Can always resolve hazards by waiting**

31

Control Hazards

A *control hazard* is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

A branch is either

- Taken: PC <= PC + 4 + Immediate
- Not Taken: PC <= PC + 4

if (cond) PC ← ALUOutput else PC ← NPC

32

Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known.
- In current MIPS pipeline, the conditional branch is resolved in the MEM stage resulting in three stall cycles as shown below:

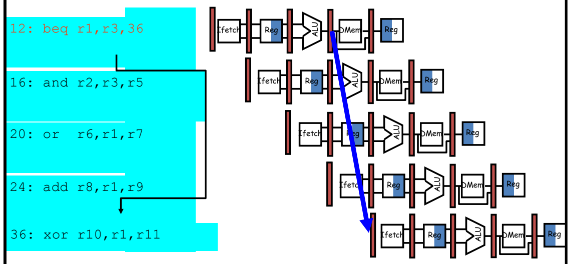
Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		IF	stall	stall	IF	ID	EX	MEM	WB	
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

Three clock cycles are wasted for every branch for current MIPS pipeline

33

Control Hazard on Branches: Three-Cycle Stall

If CPI = 1, 30% branch, Stall 3 cycles ==> new CPI = 1.9!



```
12: beq r1,r3,36
16: and r2,r3,r5
20: or  r6,r1,r7
24: add r8,r1,r9
36: xor r10,r1,r11
```

34

Reducing Branch Stall Cycles

- Pipeline hardware measures to reduce branch stall cycles:

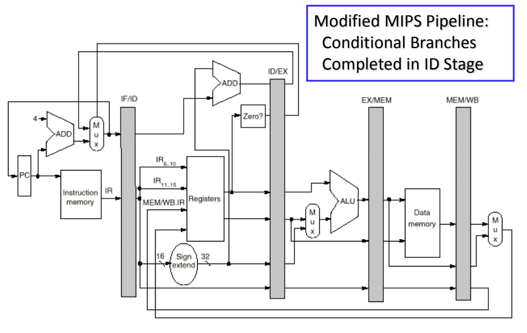
  1- Find out whether a branch is taken earlier in the pipeline.

  2- Compute the taken PC earlier in the pipeline.

  In MIPS:
  - **In MIPS branch instructions BEQZ, BNZ, test a register for equality to zero.**
  - **This can be completed in the ID cycle by moving the zero test into that cycle.**
  - **Both PCs (taken and not taken) must be computed early.**
  - **Requires an additional adder because the current ALU is not useable until EX cycle.**
  - **This results in just a single cycle stall on branches.**

35

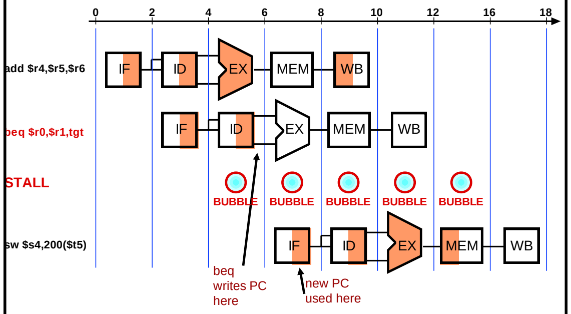
Modified MIPS Pipeline: Conditional Branches Completed in ID Stage



The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline.

36

Control Hazard - Stall



37

Reducing Branch Penalties

- One scheme is to *flush* or *freeze* the pipeline whenever a conditional branch is decoded by deleting or holding any instructions in the pipeline until the branch destination is known (zero pipeline registers, control lines).
- Another method is to *predict that the branch is not taken* where the state of the machine is not changed until the branch outcome is definitely known. Execution here continues with the next instruction; *stall occurs here when the branch is taken.*
- Another method is to *predict that the branch is taken* and begin fetching and executing at the target; *stalls occur here if the branch is not taken.*

38

Predict Not-Taken Scheme

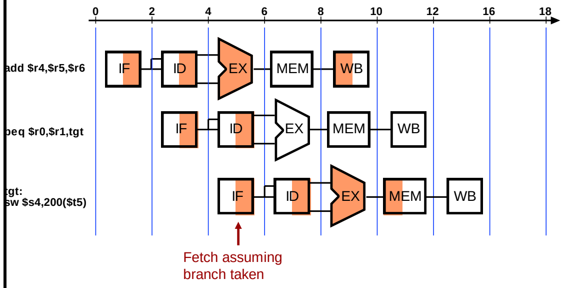
Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

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

The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

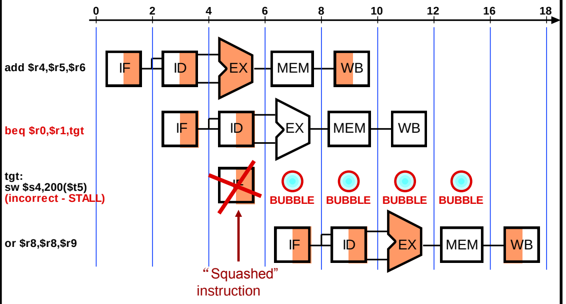
39

Control Hazard - Correct Predict-Taken



40

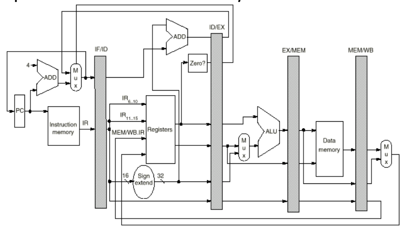
Control Hazard - Incorrect Predict-Taken



41

Canceling Branches

- When the branch goes as predicted, the instruction in the branch delay slot is executed normally.
- When the branch does not go as predicted the instruction is turned into a no-op.
- The effectiveness of this method depends on whether we predict the branch correctly.



42

Static Compiler Branch Prediction

- Two basic methods exist to statically predict branches at compile time:

1 By examination of program behavior and the use of information collected from earlier runs of the program.
   - **For example, a program profile may show that most branches are taken. The simplest scheme in this case is to just predict the branch as taken**
   - **Different branch instructions may have different behavior**

2 To predict branches on the basis of branch direction, choosing backward branches **(loop)** as taken and forward branches **(if)** as not taken.

Profile-Based Compiler Branch Misprediction Rates for SPEC92

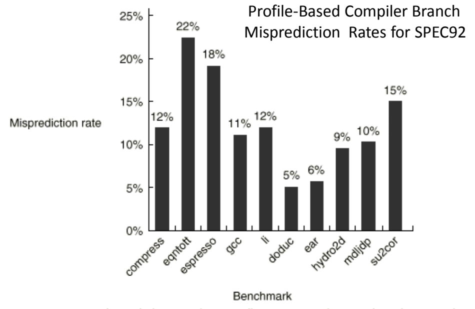
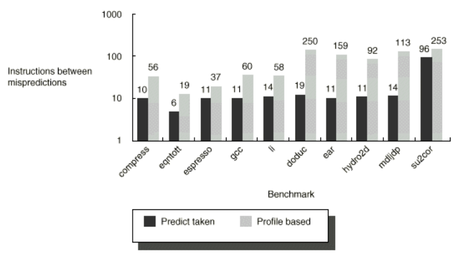


FIGURE 2.3 **Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%.**



Accuracy of a predict-taken strategy and a profile-based predictor as measured by the number of instructions executed between mispredicted branches and shown on log scale

Pipeline Performance Example

- Assume the following MIPS instruction mix:

Type	Frequency	
Arith/Logic	40%	
Load	30%	of which 25% are followed immediately by an ALU instruction using the loaded value
Store	10%	
branch	20%	of which 45% are taken

- What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using a predict branch not-taken scheme?
- CPI = Ideal CPI + Pipeline stall clock cycles per instruction

```
     =    1  +        stalls by loads  +   stalls by branches
     =    1  +         .3 x .25 x 1    +        .2 x .45 x 1
     =    1  +         .075            +        .09
     =  1.165
```

Dynamic Branch Prediction

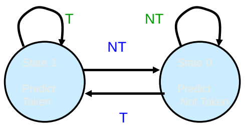
Dynamic Branch Prediction

- Builds on the premise that history matters
  - Observe the behavior of branches in previous instances and try to predict future branch behavior
  - Try to predict the outcome of a branch early on in order to avoid stalls
  - Branch prediction is critical for multiple issue processors
    - In an n-issue processor, branches will come n times faster than a single issue processor

Basic Branch Predictor

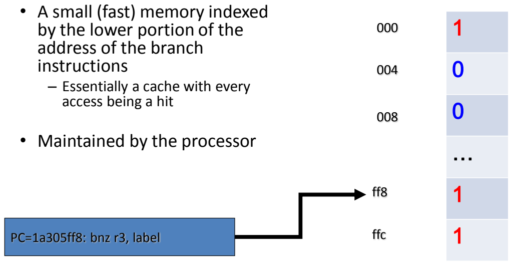
- 1-bit predictor: use a *1-bit branch predictor buffer* or *branch history table*
- 1 bit of memory stating whether the branch was recently taken or not
- Bit entry updated each time the branch instruction is executed

		Branch outcome	
Prediction	**State**	**Taken**	**Not Taken**
Taken	1	1	0
Not Taken	0	1	0



Branch-Prediction Buffer

- A small (fast) memory indexed by the lower portion of the address of the branch instructions
  - Essentially a cache with every access being a hit
- Maintained by the processor



1024-entry branch-prediction buffer indexed by 10 bits bits (12th — 3rd least significant bits)

1-bit Branch Prediction Buffer

- Problem – even simplest branches are mispredicted twice

```
        LD R3, #10
Outer_loop:
        LD R1, #5
Loop:   LD R2, 0(R5)
        ADD R2, R2, R4
        STORE R2, 0(R5)
        ADD R5, R5, #4
        SUB R1, R1, #1
        BNEZ R1, Loop
        SUB R3, R3, #1
        BNEZ R3, Outer_loop
```

First time: prediction = 0 but the branch is taken ⇒ change prediction to 1 miss

Time 2, 3, 4: prediction = 1 and the branch is taken

Time 5: prediction = 1 but the branch is not taken ⇒ change prediction to 0 miss

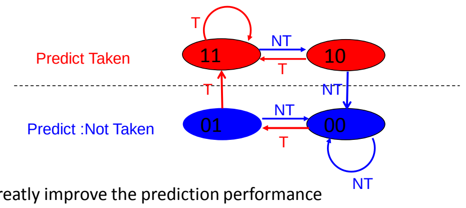
Limitations of 1-bit Predictors

- 1 mis-prediction changes the prediction
  - Only considers the taken/not-taken (T/NT) of the last time
- No consideration of the biased distribution of T/NT for branches
  - Braches are highly biased on T/NT, e.g., one braches prefer T and another prefer NT
  - Every change of branch outcome is likely to generate two mis-predictions

High mis-prediction rate!

2-bit Predictors

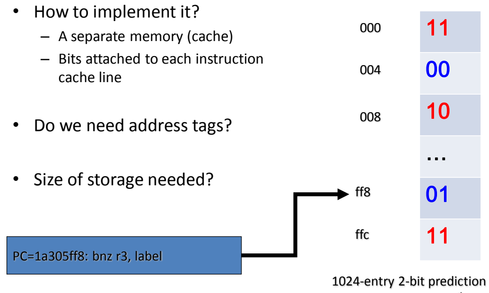
- 2-bit scheme ➔ 2 mis-predictions change the prediction



- Greatly improve the prediction performance

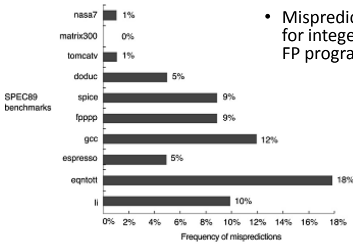
2-bit Prediction Buffer

- How to implement it?
  - A separate memory (cache)
  - Bits attached to each instruction cache line
- Do we need address tags?
- Size of storage needed?



1024-entry 2-bit prediction buffer indexed by 10 bits (12th — 3rd least significant bits)

Performance of 2-bit Predictor



- 4096-entry 2-bit prediction buffer on SPEC89
- Misprediction rates average 11% for integer programs and 4% for FP programs.

Further improvements?

- 2-bit to more-bits?
  - No much help because 2-bit predictor already captures the biased preferences of branches
- More entries (unlimited?)
  - No much help because in run-time there are not so many concurrent branches

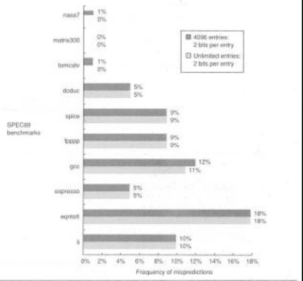
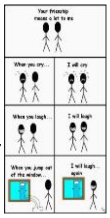


Figure 2.6 Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although this data is for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with perhaps as many as 8K entries needed to match an infinite 2-bit predictor.

Correlating Branch Predictors

You jump, I jump !
--- Jack in Titanic

You jump, I laugh
-- Cartoons



Friends mean a lot.

U cry, I cry.

U laugh, I laugh.

U jump, I laugh, again! Haha!

Correlating Branch Predictors

- Correlated branches

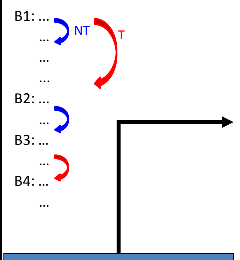
```
if (a==8) b = 5;
if (a==9) b = 22;
…

if (a==3) a = 0;
if (b==9) b = 0;
if (a!=b) c = 0;
…
```

- We may predict the branch directions based on the outcome of the last few branches

Correlating Branch Predictors

- Consider last *m* branches' decisions (T or NT)
  - E.g., m=3
- For each pattern of these *m* prior branches, construct an n-bit predictor
  - n bits (e.g., 2 bits) for each predictor
  - Based on the state of the last m branches, and the state of the predictor, make the prediction for the current branch
  - Current branch's direction (taken or not taken) will affect the prediction of the next branch
- (m,n) pridictor: "To see the last m branches, each predictor has n-bit" (e.g.,m=3,n=2)

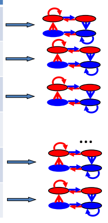
Example: m=3, n=2



PC=1a305ff8: bnz r3, label

B1 (0=NT	B2 1=T)	B3
0	0	0
0	0	1
0	1	0
…	…	…
1	1	0
1	1	1

2-bit predictor



Comparison

- (m,n) predictor
  - It is called *global* predictor
- Original 2-bit predictor is actually (0,2) predictor
  - It is called local predictor

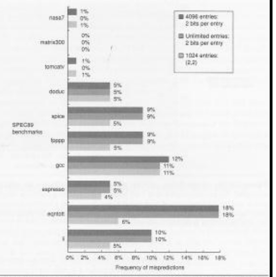


Figure 2.7 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although this data is for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

61

Storage Size

- Given a (m,n) predictor
  - Each entry has 2^m predictors
  - Each predictor has n-bit
- To carry E entries we need, $2^m \cdot n \cdot E$ bits

62

Tournament Predictor

- A combination of the local and global predictor
- Select the predictor with the best prediction rate
- Slightly better

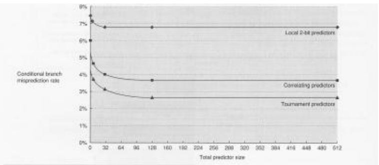


Figure 2.8 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor, which is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although this data is for an older version of SPEC, data for more recent SPEC benchmarks would show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

How to implement a tournament predictor?

63

Tackle Branch Hazards

#1: Stall until branch direction is clear
#2: Determine branch outcome and target earlier
#3: Predict the branch outcome and target
Static branch prediction
Predict Branch Not Taken
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

Predict Branch Taken
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome

Dynamic branch prediction
1-bit predictor, 2-bit predictor
Correlating branch predictor
tournament predictor
#4: Delayed Branch

64

Reduction of Branch Penalties: Delayed Branch

Define branch to take place AFTER one or more following instruction(s)

```
branch instruction
  sequential successor_1
  sequential successor_2
  ........
  sequential successor_n
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in the 5-stage MIPS pipeline

Requires compiler help

65

Reduction of Branch Penalties: Delayed Branch

- When delayed branch is used, the branch is delayed by *n* cycles, following this execution pattern:

  conditional branch instruction
  sequential successor$_1$
  sequential successor$_2$
  ....
  sequential successor$_n$
  branch target if taken

- The sequential successor instruction are said to be in the branch delay slots. These instructions are executed whether or not the branch is taken.

66

Delayed Branch Example (1-slot)

Untaken branch instruction	IF	ID	EX	MEM	WB			
Branch delay 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			
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB		
Branch target			IF	ID	EX	MEM	WB	
Branch target + 1				IF	ID	EX	MEM	WB
Branch target + 2					IF	ID	EX	MEM	WB

The behavior of a delayed branch is the same whether or not the branch is taken.

67

Reduction of Branch Penalties:

Delayed Branch

- In practice, almost all machines that utilize delayed branches have a single instruction delay slot.
- The job of the compiler is to make the successor instructions valid and useful instructions.
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

68

Delayed Branch-delay Slot Scheduling Strategies

The branch-delay slot instruction can be chosen from three cases:

A **An independent instruction from before the branch:**
Always improves performance when used. The branch must not depend on the rescheduled instruction.

B **An instruction from the target of the branch:**
Improves performance if the branch is taken and may require instruction duplication. This instruction must be safe to execute if the branch is not taken.

C **An instruction from the fall through instruction stream:**
Improves performance when the branch is not taken. The instruction must be safe to execute when the branch is taken.

69

Delayed Branch

- Instruction in branch delay slot is always executed
- Compiler (tries to) move a useful instruction into delay slot.

(a) From before the Branch: Always helpful when possible

```
ADD R1, R2, R3
BEQZ R2, L1
DELAY SLOT
 -
L1:
```

```
BEQZ    R2, L1
ADD R1, R2, R3
 -
L1:
```

- If the ADD instruction were: ADD R2, R1, R3 the move would not be possible

70

Delayed Branch

(b) From the Target: Helps when branch is taken. May duplicate instructions

```
ADD R2, R1, R3
BEQZ R2, L1
DELAY SLOT
-
L1: SUB R4, R5, R6
L2:
```

```
ADD     R2,  R1, R3
BEQZ    R2, L2
SUB R4, R5, R6
-
L1:     SUB R4, R5, R6
L2:
```

Instructions between BEQZ and SUB (in fall through) must not use R4.

71

Delayed Branch

(c) From Fall Through: Helps when branch is not taken.

```
ADD R2, R1, R3
BEQZ R2, L1
DELAY SLOT
 SUB R4, R5, R6
   -
L1:
```

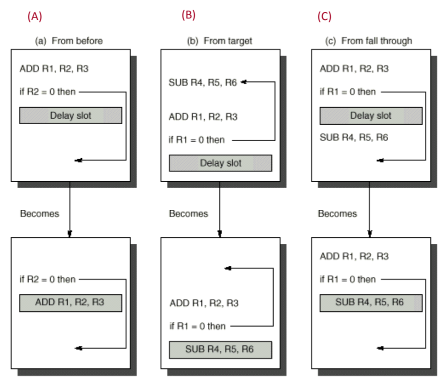
```
ADD     R2,  R1, R3
        BEQZ   R2, L1
SUB R4, R5, R6
        -

L1:
```

Instructions at target (L1 and after) <u>**must not**</u> use R4 till set again.

- **Cancelling (Nullifying) Branch:**
  Branch instruction indicates direction of prediction.
  If **mispredicted** the instruction in the delay slot is cancelled.
  Greater flexibility for compiler to schedule instructions.

Branch-delay Slot: Canceling Branches

- In a canceling branch, a static compiler branch direction prediction is included with the branch-delay slot instruction.
- When the branch goes as the compiler expects, the instruction in the branch delay slot is executed normally.
- When the branch does not go as expected the instruction is turned into a no-op.
- Canceling branches eliminate the conditions on instruction selection in delay instruction strategies B, C
- The effectiveness of this method depends on whether we predict the branch correctly.
- **In practice 50% of time, we have no stalls (nop).**



Performance of Branch Schemes

- The effective pipeline speedup with branch penalties: (assuming an ideal pipeline CPI of 1)

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{branch penalty}$$

$$\text{Pipeline speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Evaluating Branch Alternatives (MIPS)

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Scheduling	*Branch scheme penalty*	*CPI*	*speedup v. unpipelined*
Stall pipeline	1	1.14	4.4
Predict taken	1	1.14	4.4
Predict not taken	1	1.09	4.5
Delayed branch	0.5	1.07	4.6

Conditional & Unconditional = 14%, 65% change PC (taken)

Delayed Branch

- Limitations of delayed branch
  - Compiler may not find appropriate instructions to fill delay slots. Then it fills delay slots with no-ops.
  - Visible architectural feature – likely to change with new implementations
    - Pipeline structure is **exposed** to compiler. Need to know how many delay slots.

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper