## Midterm Exam Information

- Exam Date/Time: Thurs. March 10, in class
- Exam format:
  - Closed Book/ Closed notes
  - One 8.5' x 11'' (on side) sheet of notes permitted
  - Approx. 5 problems
  - Questions will be problem-solving in nature
- Exam will be based upon material covered in lecture
  - Lecture Notes should be your primary study guide
  - Relevant portions of text book:
    - Appendices A & B
    - Chapters 1 and 2
      - Can ignore sections 1.5, 1.6, 1.7 and subsection on Value Prediction in Chapter 2 (page 130)

## Simple Performance Comparison

- Machine A is n times faster than machine B iff
  perf(A)/perf(B) = time(B)/time(A) = n
- Machine A is x% faster than machine B iff
  perf(A)/perf(B) = time(B)/time(A) = 1 + x/100
- E.g. time(A) = 10s, time(B) = 15s
  - 15/10 = 1.5 => A is 1.5 times faster than B
  - 15/10 = 1.5 => A is 50% faster than B

## Processor Performance Equation

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

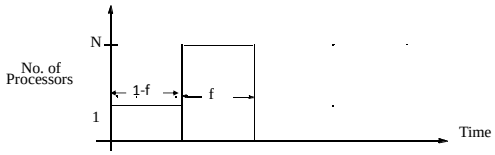
(code size) (CPI) (cycle time)

## PPE Example

- Machine A: clock 1ns, CPI 2.0, for program P
- Machine B: clock 2ns, CPI 1.2, for program P
- Which is faster and how much?
  Time/Program = instr/program x cycles/instr x sec/cycle
  Time(A) = N x 2.0 x 1 = 2N
  Time(B) = N x 1.2 x 2 = 2.4N
  Compare: Time(B)/Time(A) = 2.4N/2N = 1.2
- So, Machine A is 20% faster than Machine B for this program

## Amdahl's Law

(Originally formulated for *vector processing)*

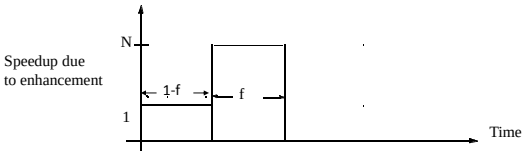



- f = fraction of program that is vectorizable
- (1-f) = fraction that is *serial*
- N = speedup for vectorizable portion
- Overall speedup:

$$Speedup = \frac{1}{(1-f) + \frac{f}{N}}$$

## Generalization of Amdahl's Law

(To apply to any processor performance enhancement*)*




- f = fraction of program that can take advantage of the enhancement
- (1-f) = fraction that cannot take advantage
- N = speedup for enhanced portion
- Overall speedup:

$$Speedup = \frac{1}{(1-f) + \frac{f}{N}}$$

## Amdahl's Law Example

- An enhancement to a processor architecture is proposed that would decrease the CPI for floating point multiply instructions from 20 cycles to 1 cycle (a speedup of 20). The CPI of all other instructions will be unchanged. What will be the overall processor speedup resulting from this modification?

## Amdahl's Law Example (continued)

Suppose that, in the original design, floating point multiplies accounted for 6% of the total execution time of a "typical program"

Then by Amdahl's law the speedup due to the enhanced floating point multiply will be

$$S = \frac{1}{(1 - 0.06) + 0.06/20} = 1.06$$

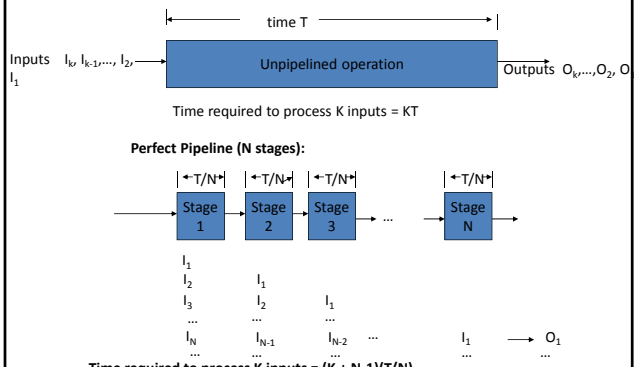
## Amdahl's Law Example (continued)

Now suppose that, for a different program, floating point multiplies account for 60% of the total execution time in the original design

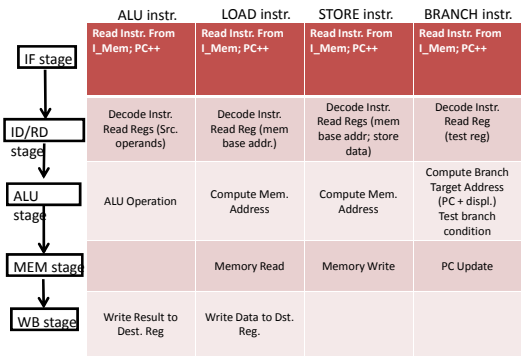
Then by Amdahl's law the speedup due to the enhanced floating point multiply (for this particular program) will be

$$S = \frac{1}{(1 - 0.6) + 0.6/20} = 2.33$$

## Ideal Pipeline Performance

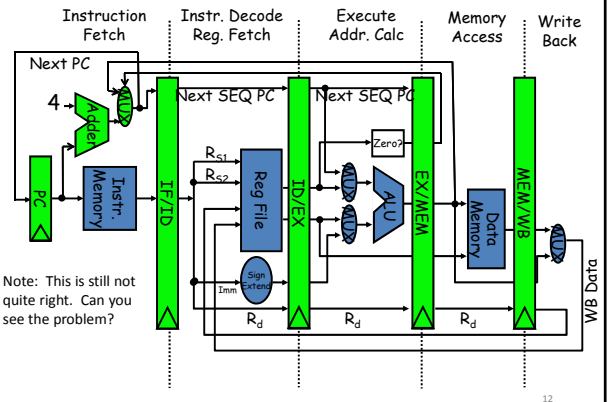



## The Unified Pipeline

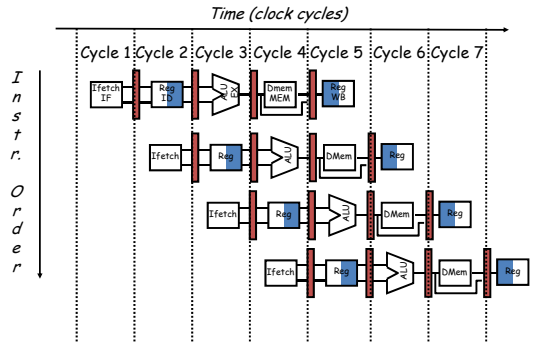



| | ALU instr. | LOAD instr. | STORE instr. | BRANCH instr. |
|---|---|---|---|---|
| IF stage | Read Instr. From I_Mem; PC++ | Read Instr. From I_Mem; PC++ | Read Instr. From I_Mem; PC++ | Read Instr. From I_Mem; PC++ |
| ID/RD stage | Decode Instr. Read Regs (Src. operands) | Decode Instr. Read Reg (mem base addr.) | Decode Instr. Read Regs (mem base addr; store data) | Decode Instr. Read Reg (test reg) |
| ALU stage | ALU Operation | Compute Mem. Address | Compute Mem. Address | Compute Branch Target Address (PC + displ.) Test branch condition |
| MEM stage | | Memory Read | Memory Write | PC Update |
| WB stage | Write Result to Dest. Reg | Write Data to Dst. Reg. | | |

## 5 Steps of MIPS-like Datapath (corrected)

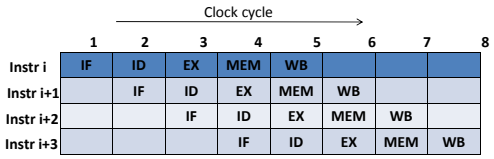


## Visualizing Pipelining




13

## An Even Simpler View

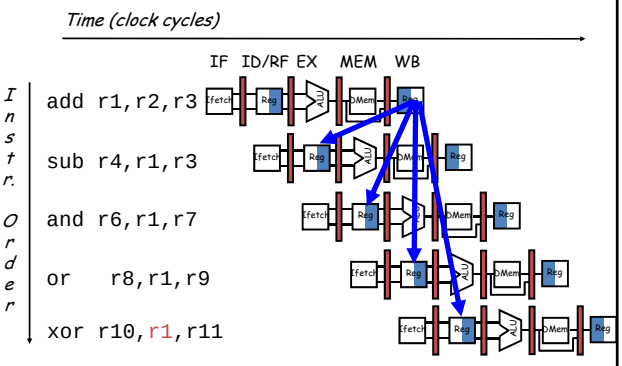



## But, Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

15

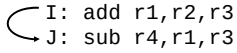
## Data Hazard on R1




16

## Three Generic Data Hazards

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

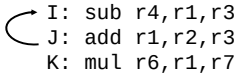



- Caused by a "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

17

## Three Generic Data Hazards

- Write After Read (WAR)
  Instr$_J$ writes operand *before* Instr$_I$ reads it

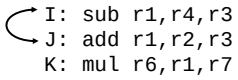



- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Register Reads are always in stage 2, and
  - Register Writes are always in stage 5

18

## Three Generic Data Hazards

- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.
- 




- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Register Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

19

## Resolution of Pipeline Hazards

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

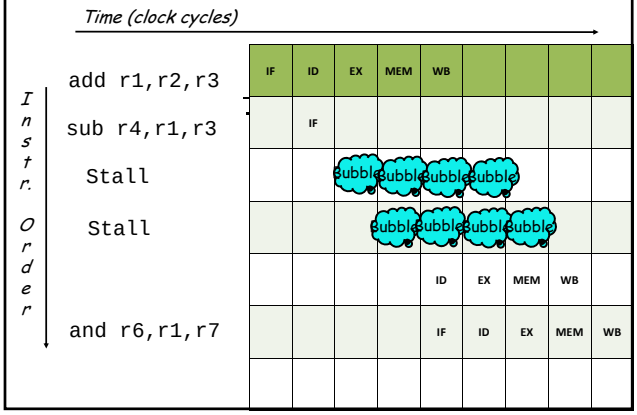
20

## Data Hazard Mitigation

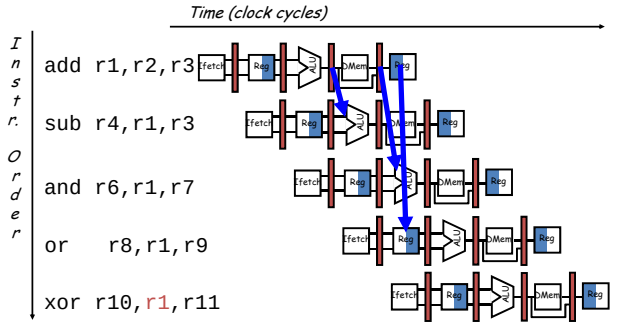
- A better response – forwarding
  - Also called bypassing
- Comparators ensure register is read after it is written
- Instead of stalling until write occurs
  - Use mux to select forwarded value rather than register value
  - Control mux with hazard detection logic

21

## Data Hazard on R1

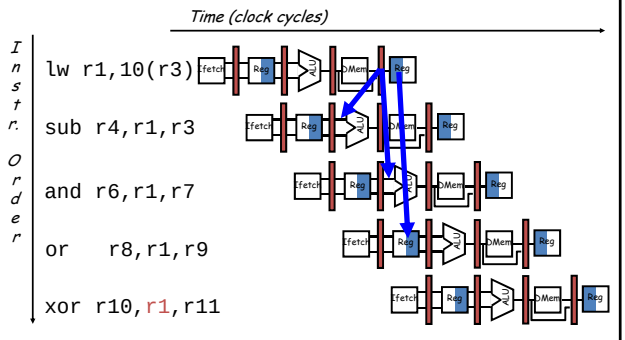



## Forwarding to Avoid Data Hazards



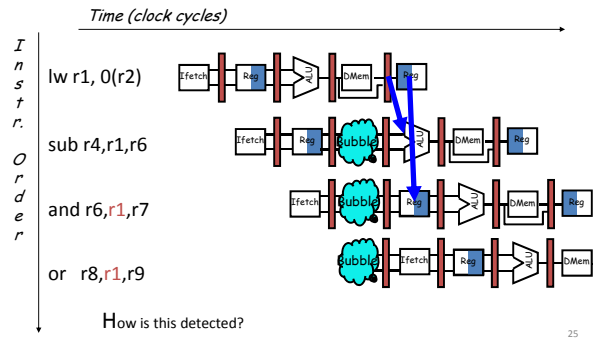

23

## RAW Data Hazards Involving Loads




24

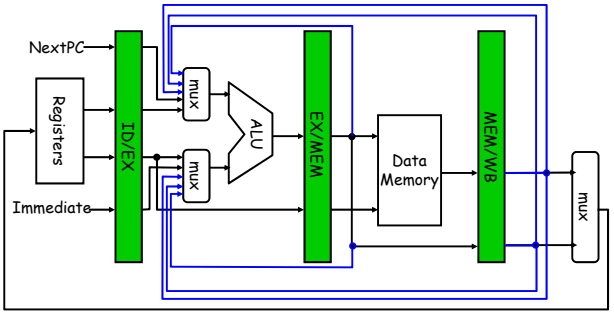
## Data Hazard Even with Forwarding




How is this detected?

25

## HW Change for Forwarding




What circuit detects and resolves this hazard?

26

## Control Dependences

- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction
- Unconditional Branches (including subroutine calls
  - Branch can't take place until branch target address is calculated
- Exceptions
  - Interrupts
  - Hardware Exceptions
  - Trap Instructions

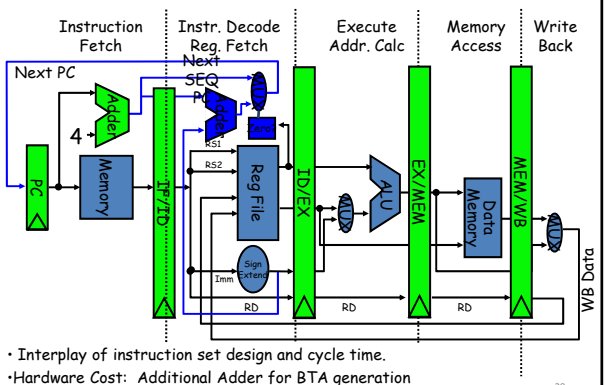
27

## Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch outcome(taken/not-taken) sooner, AND
  - Compute branch target address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

28

## Pipelined MIPS Datapath




- Interplay of instruction set design and cycle time.
- Hardware Cost: Additional Adder for BTA generation



## Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
- Execute successor instructions in sequence
- "Cancel" 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

#3: 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



## Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER following instruction(s)

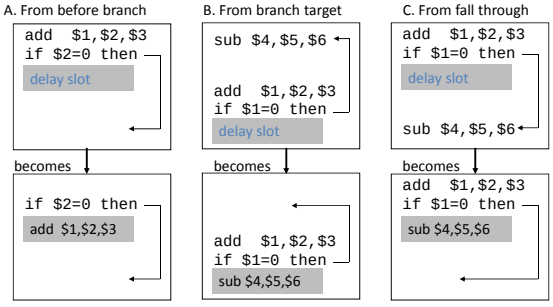
```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n* *(branch shadow)*

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



## Scheduling Branch Delay Slots




- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

## Evaluating Branch Alternatives

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

Assume 4% unconditional branch, 6% conditional branch- untaken, 10% conditional branch-taken

| *Scheduling scheme* | *Branch penalty* | *CPI* | *speedup v. unpipelined* | *speedup v. stall* |
|---|---|---|---|---|
| Stall pipeline | 1 | 1.2 | 4.17 | 1.0 |
| Predict not taken | 1* | 1.14 | 4.39 | 1.05 |
| Delayed branch | 0.5 | 1.10 | 4.55 | 1.09 |

* Only for wrong prediction

Assumes Branch Outcome determination and BTA generation in decode stage, 50% of delay slots filled with useful instructions for delayed branching

33

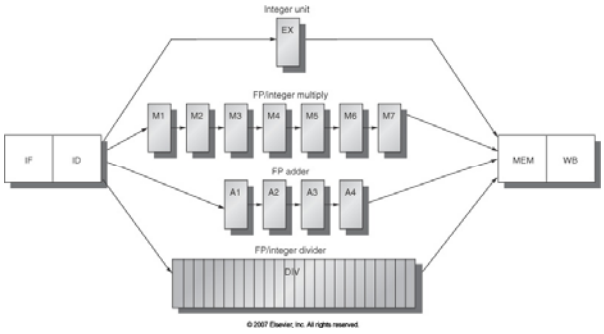
## Limitations of Our Simple 5-stage Pipeline

- Assumes single cycle EX stage for all instructions
- This is not feasible for
  - Complex integer operations
    - Multiply
    - Divide
    - Shift (possibly)
  - Floating Point Operations



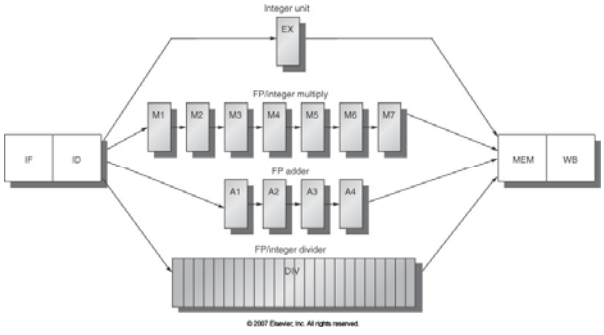
34

## Diversified Pipeline




35

## Diversified Pipeline




36

## Problems with Diversified Pipeline

- Many more RAW hazard opportunities due to longer fp instruction execution times
- New Structural Hazards:
  - Divide instructions at distance < 25 (Due to non-pipelined Divide Unit.
  - Multiple Register Writes/Cycle due to variable instruction execution times
- Out-of-order instruction completion—Why is this a problem?
- WAW Hazards are possible (WAR not possible. Why?)

37

## Structural Hazard--FP Register Write Port

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL.D | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| I+1 | | IF | ID | ... | ... | ... | ... | ... | ... | ... | ... |
| I+2 | | | IF | ... | ... | ... | ... | .... | .... | ... | ... |
| ADD.D | | | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB |
| I+4 | | | | | IF | ... | ... | ... | ... | ... | ... |
| I+5 | | | | | | IF | ... | ... | ... | ... | ... |
| LOAD.D | | | | | | | IF | ID | EX | MEM | WB |

**Three FP Register Writes in Same Cycle**



38 38

## Diversified Pipeline--WAW Hazard

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL.D F0,F2,F4 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| I+1 | | IF | ID | ... | ... | ... | ... | ... | ... | ... | ... |
| I+2 | | | IF | ... | ... | ... | ... | ... | ... | ... | ... |
| I+3 | | | | IF | ... | ... | ... | ... | ... | ... | ... |
| I+4 | | | | | IF | ... | ... | ... | ... | ... | ... |
| LOAD.D F0,10(R3) | | | | | | IF | ID | EX | MEM | WB | |

39

## Diversified pipeline—Out of Order Completion

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| DIV.D F0,F2,F4 | IF | ID | D1 | D2 | D3 | D4 | D5 | D6 | D7 | D8 | D9 |
| ADD.D F2,F10,F8 | | IF | ID | EX | A1 | A2 | A3 | A4 | MEM | WB | |
| LOAD.D F4,10(R3) | | | IF | ID | EX | MEM | WB | | | | |

Note that both the ADD and LOAD complete before the DIV

Suppose a hardware exception occurs during the DIV, after stage 8. What is the PC address of the exception?

Also note that the ADD and LOAD have overwritten the source operands for the DIV so there is no way to restore the state before the DIV

40

## Can the Compiler Help?

```
Loop: L.D    F0,0(R1);F0=vector element
      ADD.D  F4,F0,F2;add scalar from F2
      S.D    0(R1),F4;store result
      DADDUI R1,R1,-8;decrement pointer 8B (DW)
      BNEZ   R1,Loop ;branch R1!=zero
```

- Assume the following pipeline latencies:
  - Ignore delayed branch in these examples

| *Instruction producing result* | *Instruction using result* | *stalls between in cycles* |
|---|---|---|
| FP ADD | Another FP ALU op | 3 |
| FP ADD | Store double | 2 |
| Load double | FP ALU op | 1 |
| Load double | Store double | 0 |
| Integer op | Integer op | 0 |

41

## Reorganized Code to Reduce Stalls

Swap DADDUI and S.D by changing address of S.D:

```
1 Loop: L.D    F0,0(R1)
2       DADDUI R1,R1,-8
3       ADD.D  F4,F0,F2
4       stall
5       stall
6       S.D    8(R1),F4 ;altered offset when move DADUI
7       BNEZ   R1,Loop
```

| *Instruction producing result* | *Instruction using result* | *Latency in clock cycles* |
|---|---|---|
| FP ALU op | Another FP ALU op | 3 |
| FP ALU op | Store double | 2 |
| Load double | FP ALU op | 1 |

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead;
Can we(the compiler) do better?

42

## Loop Unrolling to Improve Performance

```
1 Loop:L.D    F0,0(R1)
3      ADD.D  F4,F0,F2
6      S.D    0(R1),F4    ;drop DADDUI & BNEZ
7      L.D    F6,-8(R1)
9      ADD.D  F8,F6,F2
12     S.D    -8(R1),F8   ;drop DADDUI & BNEZ
13     L.D    F10,-16(R1)
15     ADD.D  F12,F10,F2
18     S.D    -16(R1),F12 ;drop ADDUI & BNEZ
19     L.D    F14,-24(R1)
21     ADD.D  F16,F14,F2
24     S.D    -24(R1),F16
25     DADDUI R1,R1,#-32  ;alter to 4*8
26     BNEZ   R1,LOOP
```

1 cycle stall

2 cycles stall

*27 clock cycles, or 6.75 per iteration*
(Assumes R1 is multiple of 4)

43

## Loop Unrolling with Code Rearrangement

```
1 Loop:L.D    F0,0(R1)
2      L.D    F6,-8(R1)
3      L.D    F10,-16(R1)
4      L.D    F14,-24(R1)
5      ADD.D  F4,F0,F2
6      ADD.D  F8,F6,F2
7      ADD.D  F12,F10,F2
8      ADD.D  F16,F14,F2
9      S.D    0(R1),F4
10     S.D    -8(R1),F8
11     S.D    -16(R1),F12
12     DSUBUI R1,R1,#32
13     S.D    8(R1),F16 ; 8-32 = -24
14     BNEZ   R1,LOOP
```

*14 clock cycles, or 3.5 per iteration*

44

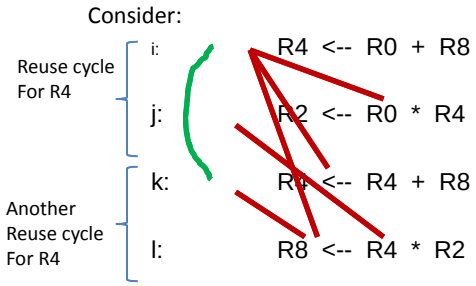
## Hardware-based Performance Optimization-- Dynamic Scheduling

- Dynamic scheduling - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
  - Handles cases when dependences unknown at compile time
  - Allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
  - Allows code to be compiled independently of details of a particular pipeline
  - Simplifies the compiler
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling (more about this later)

45

## Dynamic Scheduling Example

Consider:

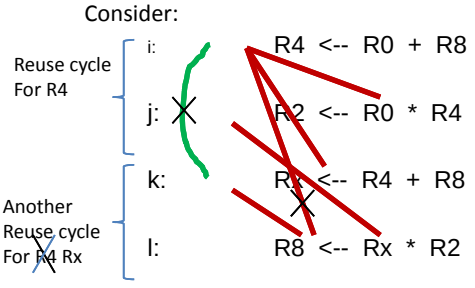



RAW Hazards WAW Hazards

46

## Dynamic Scheduling Example

Consider:

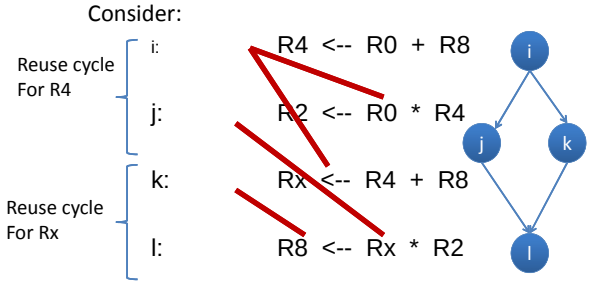



RAW Hazards WAW Hazards

47

## Dynamic Scheduling Example

Consider:




RAW Hazards

48

## Dynamic Scheduling

- Key idea: Allow instruction(s) following a stall to proceed

```
DIVD   F0,F2,F4
ADDD   F10,F0,F8
SUBD   F12,F8,F14
```

- Enables out-of-order execution and allows out-of-order completion (e.g., SUBD)
- Will distinguish when an instruction *begins execution* and when it *completes execution*; between these times, the instruction is *in execution*
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder



## Dynamic Scheduling—Starting Point

- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- *Issue*—Decode instructions, check for structural hazards
- *Read operands*—Wait until no data hazards, then read operands

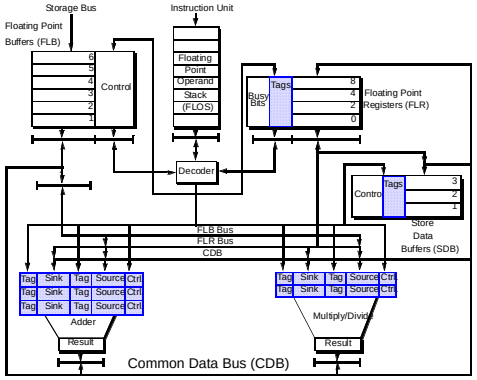


## Tomasulo's Algorithm

- Control & buffers distributed with Functional Units (FU)
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - Renaming avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Result forwarding via a Common Data Bus that broadcasts results to all FUs
  - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue



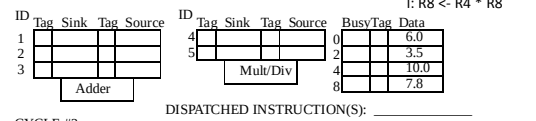
## IBM 360/91 FPU



## Example 3

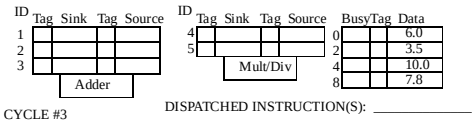
i: R4 <- R0 * R8
j: R2 <- R0 + R4
k: R4 <- R0 + R8
l: R8 <- R4 * R8

CYCLE #1



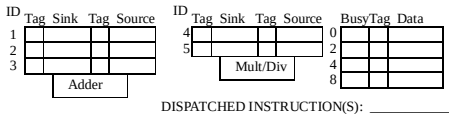

DISPATCHED INSTRUCTION(S): _____

CYCLE #2




DISPATCHED INSTRUCTION(S): _____

CYCLE #3




DISPATCHED INSTRUCTION(S): _____

53

## Performance Enhancement—Better Branch Prediction

- Accurate Branch Prediction becomes more important with dynamic scheduling
  - Dynamic scheduling may stall if it can't look past branch points
  - Cost of misprediction may be high

## Dynamic (Run-time) Branch Prediction

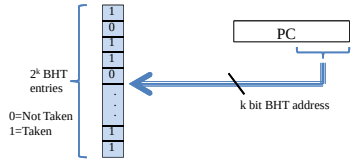
- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be (most modern processor use it)
  - There are a small number of important branches in programs which have dynamic behavior

55

## Dynamic Branch Prediction

Simplest Dynamic Predictor:
Branch History Table: Lower bits of PC address index table of 1-bit values

Keeps track of whether or not branch taken last time
No address check

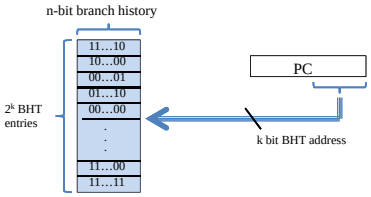



Problem: in a loop, 1-bit BHT will cause two mispredictions (average loop has only 9 iterations before exit):

End of loop case, when it exits instead of looping as before
First time through loop on *next* time through code, when it predicts exit instead of looping

56

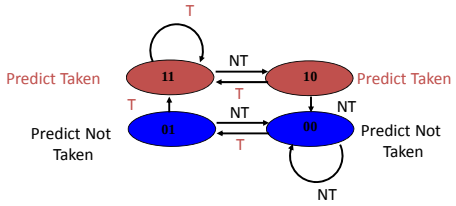
## Multi-bit Branch History




In general, there is little performance improvement Beyond n=2

## A two-bit branch predictor

- Change prediction only if get misprediction *twice*

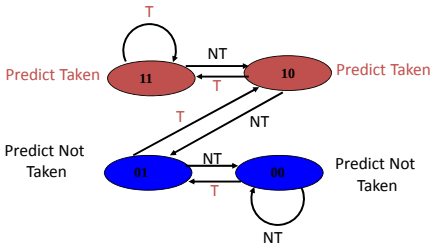



- Adds *hysteresis* to decision making process
- Many other two-bit prediction schemes are possible

58

## Another two-bit branch predictor

- Two-bit saturating counter (Smith Predictor)




59

## Correlated Branch Prediction

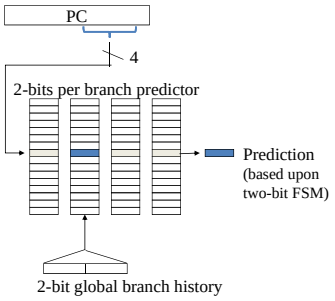
- Idea: track the outcome of the *m* most recently executed branches (globally), and use that pattern to select the proper *n*-bit branch history table
- In general, (*m*,*n*) predictor means use last *m (global)* branch outcomes to select between $2^m$ history tables, each with *n*-bit counters
  - Thus, old 2-bit BHT is a (0,2) predictor
- Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.
- Each entry in table has *m* *n*-bit predictors (local branch history).

60

## Example: A (2,2) Branch Predictor

(2,2) predictor

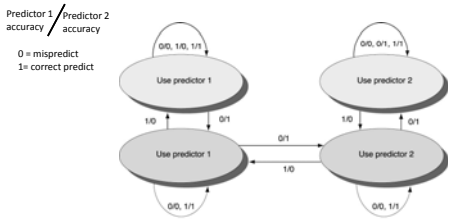
- Behavior of recent branches selects between four predictions of next branch, updating just that prediction




61

## Tournament Branch Predictor

- Multilevel branch predictor
- Use *n*-bit saturating counter to choose between predictors
- Usual choice between global and local predictors





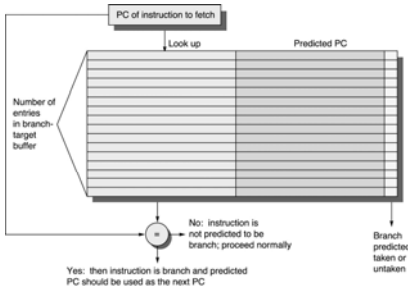

62

## Branch Prediction—What about the Branch Target Address(BTA)?

- Branch Prediction is of no value unless we know the BTA
- Branch target calculation is costly and stalls the instruction fetch.
- A Branch Target Buffer (BTB) can store previously computed BTAs
- The BTA of a taken branch is stored in the BTB
- For subsequent executions of this branch, the BTA can be "looked up" in the BTB
- If the branch was predicted taken, instruction fetch continues at the predicted PC

63

## Branch Target Buffer (BTB)

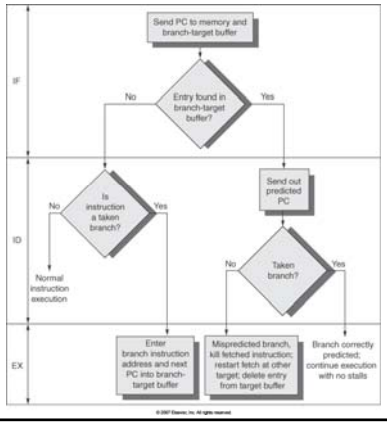


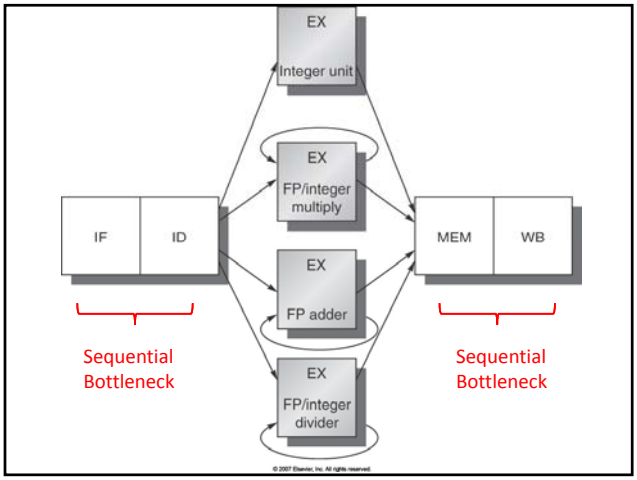

Often, BTB is used in conjunction with Dynamic Prediction

- BTB provides fast prediction and BTA in fetch stage
- Dynamic Predictor provides more accurate prediction in decode stage

64

## BTB Flowchart

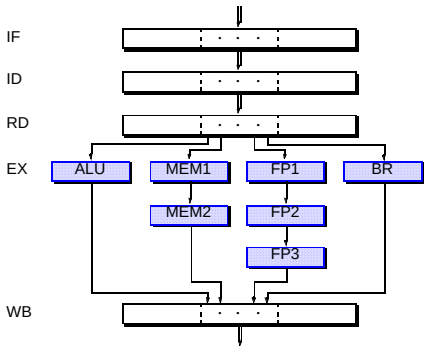



## Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
- Inefficient unified pipeline
  - Long latency for each instruction
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Tomasulo's algorithm alleviated this problem

## Superscalar Pipeline



## Challenge for Superscalar Pipes

- How to keep the pipeline operating at or near full capacity?
  - Wide instruction fetch gobbles up instructions at a high rate
  - Branches are encountered frequently
  - Cost of stalls is much higher than for scalar pipelines
- Branches pose the biggest challenge to exploiting Instruction Level Parallelism (ILP)

## Superscalar Pipelines—Exploiting ILP

- To maintain a steady stream of instructions to feed functional units it is necessary to maintain instruction fetch and execution beyond branch points
- This leads to "speculative execution" of instructions
  - Accurate branch prediction is essential
  - Must insure that wrong guesses don't lead to incorrect behavior

## Speculation for greater ILP

- Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  - Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
  - Dynamic scheduling ⇒ only fetches and issues instructions
- Essentially a data flow execution model: Operations execute as soon as their operands are available



## Speculation for greater ILP

- 3 components of HW-based speculation:

1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow execution of instructions before control dependences are resolved
   + ability to undo effects of incorrectly speculated sequence
3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks

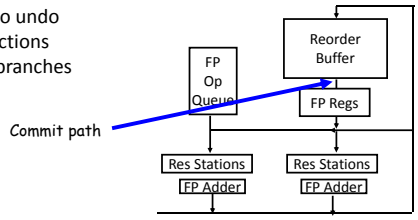
## Adding Speculation to Tomasulo's Algorithm

- Must separate execution from instruction completion or "commit"
- This additional step called instruction commit
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This reorder buffer (ROB) is also used to pass results among instructions that may be speculated

73

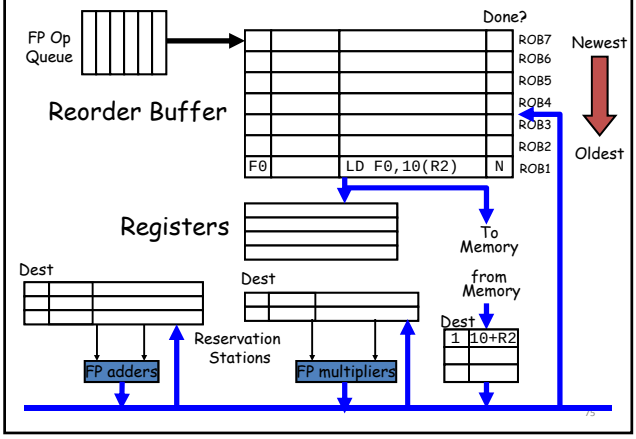
## Reorder Buffer operation

- Holds instructions in FIFO order, exactly as dispatched
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions




74

## Tomasulo With Reorder buffer:



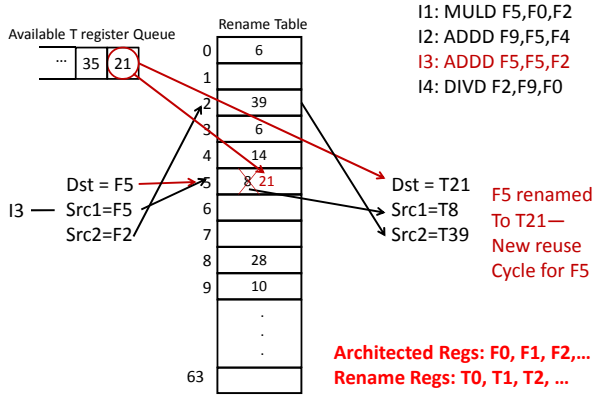

75

## Speculation: Register Renaming vs. ROB

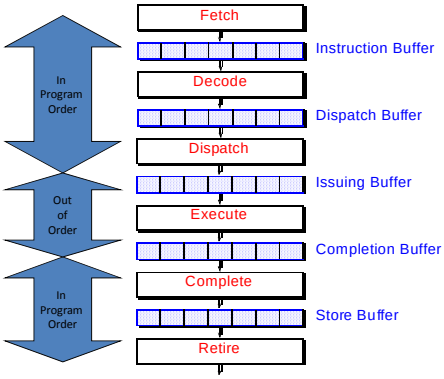
- Alternative to ROB is a larger physical set of registers combined with register renaming
  - Extended registers replace function of both ROB and reservation stations
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits
- Most Out-of-Order processors today use extended registers with renaming

76

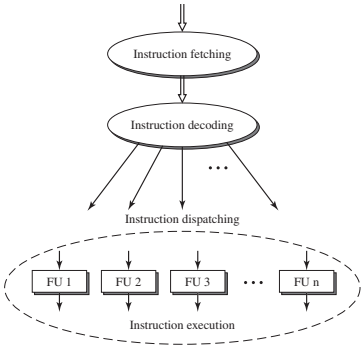
## Register Renaming Example



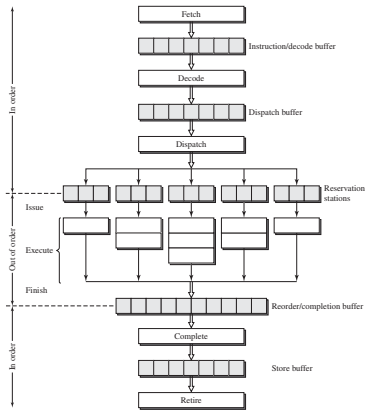

## Superscalar Pipeline Stages




## Necessity of Instruction Dispatch




## A Dynamic Superscalar Processor



## Avoiding Memory Hazards

- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
- RAW hazards through memory are maintained by two restrictions:
  1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
  2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
- these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

81

## Memory Data Dependences

- **"Memory Aliasing" = Two memory references involving the same memory location (collision of two memory addresses).**
- **"Memory Disambiguation" = Determining whether two memory references will alias or not (whether there is a dependence or not).**
- **Memory Dependency Detection:**
  - Must compute effective addresses of both memory references
  - Effective addresses can depend on run-time data and other instructions
  - Comparison of addresses require much wider comparators

**Example code:**

```
(1)  STORE  V
(2)  ADD
(3)  LOAD   W
(4)  LOAD   X
(5)  LOAD   V
(6)  ADD
(7)  STORE  W
```

82

## Conservative Approach: Maintain Total Order of Loads and Stores

- **Keep all loads and stores totally in order with respect to each other.**
- **However, loads and stores can execute out of order with respect to other types of instructions.**
- **Consequently, stores are held for all previous instructions, and loads are held for stores.**
  - **I.e. stores performed at commit point**
  - **Sufficient to prevent wrong branch path stores since all prior branches now resolved**

83

## Load Bypassing

- **Loads can be allowed to bypass stores (if no aliasing).**
- **Two separate reservation stations and address generation units are employed for loads and stores.**
- **Store addresses still need to be computed before loads can be issued to allow checking for load dependences. If dependence cannot be checked, e.g. store address cannot be determined, then all subsequent loads are held until address is valid (conservative).**
- **Stores are kept in ROB until all previous instructions complete; and kept in the store buffer until gaining access to cache port.**

84

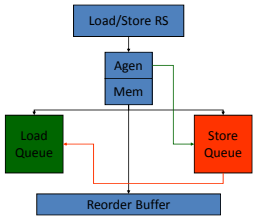
## Load Forwarding

- **If a subsequent load has a dependence on a store still in the store buffer, it need not wait till the store is issued to the data cache.**
- **The load can be directly satisfied from the store buffer if the address is valid and the data is available in the store buffer.**
- **This avoids the latency of accessing the data cache.**

85

## Speculative Disambiguation

- What if aliases are rare?
  1. Loads don't wait for addresses of all prior stores
  2. Full address comparison of stores that are ready
  3. Bypass if no match, forward if match
  4. Check all store addresses when they commit
     - No matching loads – speculation was correct
     - Matching unbypassed load – incorrect speculation
  5. Replay starting from incorrect load




86