Instruction-Level Parallelism
CS 282 – KAUST – Spring 2010
Muhamed Mudawar

Original slides by: David Patterson

1

Outline

- **ILP**
- Loop unrolling
- Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1

2

Recall from Pipelining Review

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

3

Instruction Level Parallelism

- Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP:
  1) Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power) , and
  2) Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2)

4

Instruction-Level Parallelism (ILP)

- Basic Block (BB) is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25% => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g.,

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

5

Loop-Level Parallelism

- Exploit loop-level parallelism by "unrolling loop" either by

1. dynamic via branch prediction or
2. static via loop unrolling by compiler

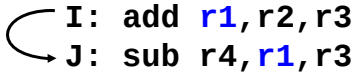
   (Another way is vectors, to be covered later)

- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are
  - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

6

Data Dependence and Hazards

- $Instr_J$ is data dependent (aka true dependence) on $Instr_I$:
  1. $Instr_I$ produces a result that is used by $Instr_J$



  2. or $Instr_J$ is data dependent on $Instr_K$ which is data dependent on $Instr_I$
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence

  $\Rightarrow$ effect of original data dependence must be preserved
- If data dependence causes a hazard in pipeline, then it is called a Read After Write (RAW) data hazard

7

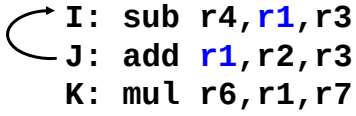
ILP and Data Dependencies, Hazards

- HW/SW must preserve program order:
  order that instructions would execute in if executed sequentially as determined by original source program
  - Dependencies are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1) indicates the possibility of a data hazard
  2) determines order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

8

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- Anti-dependence between Instruction I and J
- $Instr_I$ reads an operand, which is later written by $Instr_J$



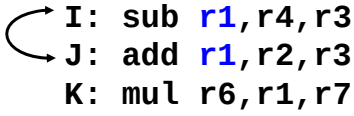
This results from reuse of the name "r1"

- If anti-dependence causes a hazard in the pipeline, then it is called a Write After Read (WAR) hazard

9

Name Dependence #2: Output dependence

- Output dependence between instructions I and J
- $Instr_I$ and $Instr_J$ write the same register or memory.



- This also results from the reuse of name "r1"
- If output-dependence causes a hazard in the pipeline, then it is called a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for registers
  - Either by compiler or by HW

10

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {
  S1;
};
if p2 {
  S2;
}
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.

11

Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties are critical to program correctness
  1) Preserving exception behavior and
  2) Preserving data flow

12

Preserving Exception Behavior

- ⇒ any changes in instruction execution order must not change how exceptions are raised in program (⇒ no new exceptions)
- Example: (assume branches are not delayed)

```
    DADDU       R2,R3,R4
    BEQZ        R2,L1
    LW          R1,0(R2)
L1:
```

- Is LW data dependent on BEQZ? (NO)
- Is LW control dependent BEQZ? (YES)
- Problem with moving LW before BEQZ?

13

Preserving Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:

```
    DADDU     R1,R2,R3
    BEQZ      R4,L
    DSUBU     R1,R5,R6
L:  . . .
    OR        R7,R1,R8
```

- **OR** depends on **DADDU** or **DSUBU**?
  Must preserve data flow on execution

14

Outline

- ILP
- **Loop unrolling**
- Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1

15

Software Techniques - Example

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i–1)
   x[i] = x[i] + s;
```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

Instruction producing result	Instruction using result	Latency in stalls cycles
FP operation	Another FP operation	3
FP operation	Store	2
Load	FP operation	1
Load	Store	0
Integer ALU op	Integer ALU op	0
Integer ALU op	Branch	1

16

FP Loop: Where are the Hazards?

- **First translate into MIPS code:**
- **Register R2 contains address of x[0]**
- **Register R1 is used as a pointer to x[i]**
- **Initially, R1 point to last element x[1000]**

```
Loop: L.D    F0,0(R1)    ;F0=vector element
      ADD.D  F4,F0,F2    ;add scalar in F2
      S.D    0(R1),F4    ;store result
      DADDUI R1,R1,-8    ;decrement pointer
      BNE    R1,R2,Loop  ;branch R1!=R2
```

17

FP Loop Showing Stalls

```
1 Loop: L.D    F0, 0(R1)     ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2      ;add scalar in F2
4       stall
5       stall
6       S.D    F4, 0(R1)     ;store result
7       DADDUI R1,R1,-8      ;decrement pointer
8       stall                ;can't forward to branch
9       BNE    R1,R2,Loop    ;branch R1!=R2
```

Instruction producing result	Instruction using result	Latency in stalls cycles
FP operation	Store	2
Load	FP operation	1
Integer ALU op	Branch	1

Without any scheduling, each loop iterate will take 9 cycles

Branch delay is assumed to be zero

18

Revised FP Loop Minimizing 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    F4, 8(R1)     ;altered address to 8(R1)
7       BNE    R1, R2, Loop
```

Reduced to 7 clock cycles per loop iterate

3 cycles for execution (L.D, ADD.D, S.D)

4 cycles for loop overhead and stall cycles

How to make it run faster?

Unroll Loop Four Times

(assume number of loop iterations is multiple of 4)

```
1 Loop:L.D    F0,0(R1)
3      ADD.D  F4,F0,F2
6      S.D    0(R1),F4      ;drop DSUBUI & BNEZ
7      L.D    F6,-8(R1)
9      ADD.D  F8,F6,F2
12     S.D    -8(R1),F8     ;drop DSUBUI & BNEZ
13     L.D    F10,-16(R1)
15     ADD.D  F12,F10,F2
18     S.D    -16(R1),F12   ;drop DSUBUI & 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 -8*4
27     BNE    R1,R2,LOOP
```

1 stall cycle

2 stall cycles

27 clock cycles, or 6.75 cycles per iteration

Rewrite loop to minimize stalls?

Unrolled Loop That Minimizes Stalls

```
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       DADDUI  R1,R1,#-32
13       S.D     8(R1),F16     ; 8-32 = -24
14       BNE     R1,R2,LOOP
```

14 clock cycles, or 3.5 cycles per iteration

21

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop

22

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent (requires analyzing memory address and finding that they do not refer to the same address)
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

23

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling (Amdahl's Law)
   - Overhead is ½ cycle per iteration when loop is unrolled 4 times. It is ¼ cycle per iteration if loop is unrolled 8 times.
2. Growth in code size
   - For larger loops, it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline
- Another way is branch prediction

24

Outline

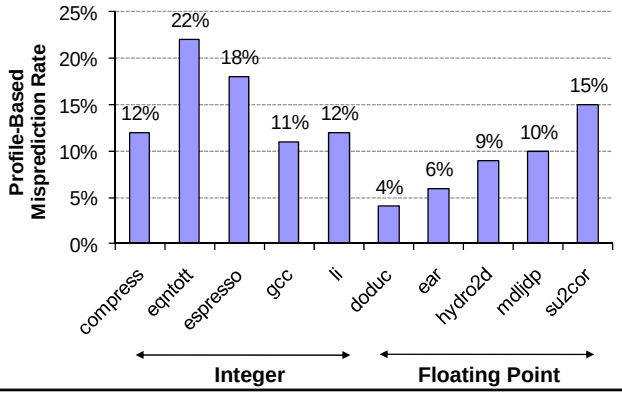
- ILP
- Loop unrolling
- **Branch Prediction**
- Dynamic Scheduling
- Tomasulo Algorithm
- Reorder Buffer
- CPI less than 1

25

Static Branch Prediction

- Predict branch statically when we compile the program
- Simplest scheme is to predict all branches as taken
  - Untaken branch frequency = 34% of all branch instructions (SPEC programs)
  - Or predict backward branches as taken and forward branches as not taken
- Some processors allow branch prediction hints to be inserted in code

More accurate static scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run



Dynamic Branch Prediction

- Main advantages:
  - Learn branch behavior autonomously
    - No compiler analysis, heuristics, or profiling
  - Adapt to changing branch behavior
    - Program phase changes branch behavior
- First proposed in 1979-1980
  - US Patent #4,370,711, Branch predictor using random access memory, James. E. Smith
- Continually refined since then

Dynamic 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
  - There are some important branches in programs which have dynamic behavior

28

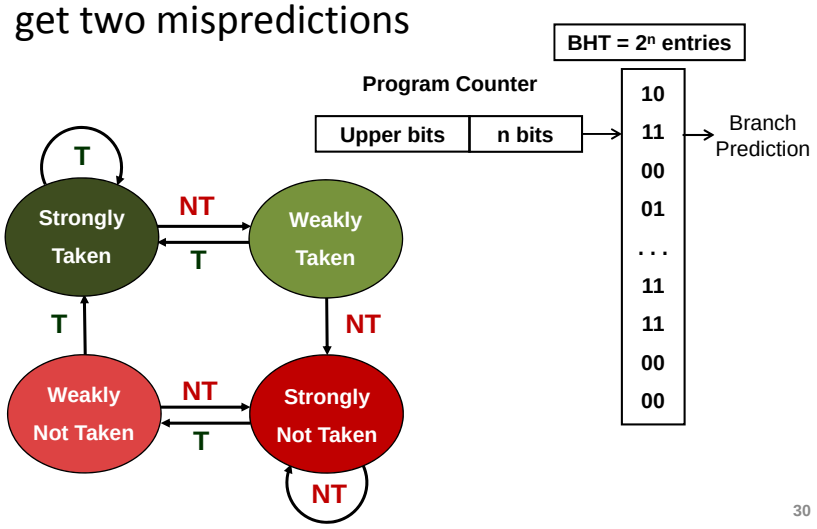
Dynamic Branch Prediction

- Performance = ƒ(accuracy, cost of misprediction)
- Branch History Table (BHT)
- Lower bits of PC address = index to BHT table
  - Each entry consists of few bits
  - Says whether or not branch is predicted to be taken
  - No address check
- 1-bit BHT is simplest to implement
  - Record last branch outcome and uses it to predict future
  - Problem: in a loop, 1-bit BHT will cause two mispredictions
  - 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

29

2-bit Predictors

- 2-bit scheme change prediction only if we get two mispredictions

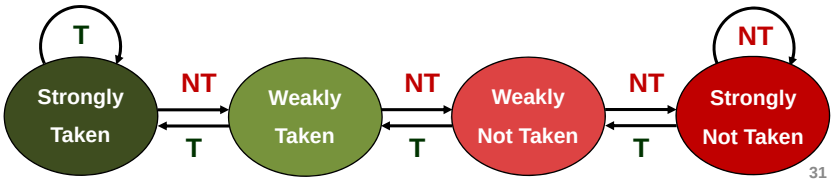


30

2-bit Saturating Counter

- Slightly different state diagram
  - Saturates at strongly not taken = 00 and strongly taken = 11
- Example on branch prediction

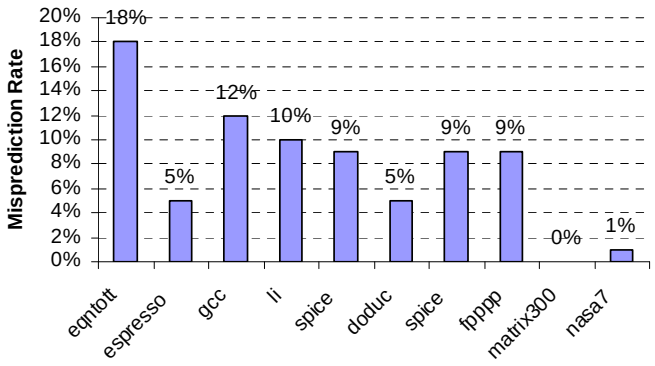
Outcome:	T	T	T	N	T	N	N	N	T	N
State:	WT	ST	ST	WT	ST	WT	WN	SN	WN	SN
Predict:	T	T	T	T	T	T	N	N	N	N
Correct?	C	C	C	X	C	X	C	C	X	C



31

BHT Accuracy

- Branch misprediction because either:
  - Wrong guess for that branch
  - Got history of wrong branch when indexing the table
- 4096-entry Branch History Table:



32

Correlating Predictors

- Branches from different instructions may be correlated

```
if (aa < 0) aa = 0;
if (bb < 0) bb = 0;
if (aa != bb) { . . . }
```

- If the first two conditions are true, then the third one will be false
- Save history of all recent branch outcomes
- Global Branch History Register is a *m*-bit shift register
  - Holds most recent *m* branch outcomes
  - Approximation to path followed

33

Correlated Branch Prediction

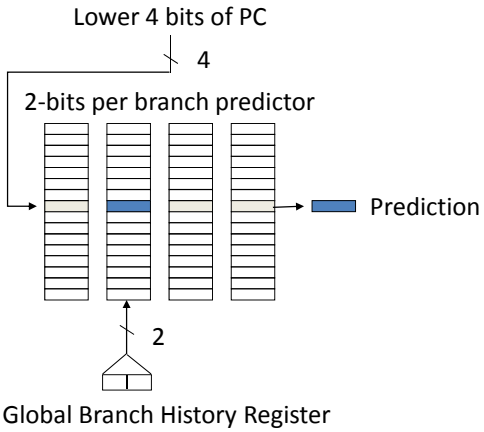
- Idea: record *m* most recently executed branches as taken or not taken, and use that pattern to select the proper *n*-bit branch history table
- In general, (*m*,*n*) predictor means record last *m* branches to select between 2^m branch history tables, each with *n*-bit counters
  - Thus, local 2-bit BHT is a (0,2) predictor
- Global Branch History Register: *m*-bit shift register keeping status of last *m* branches
- Use both PC and GBHR to access prediction table

34

Correlating Branches

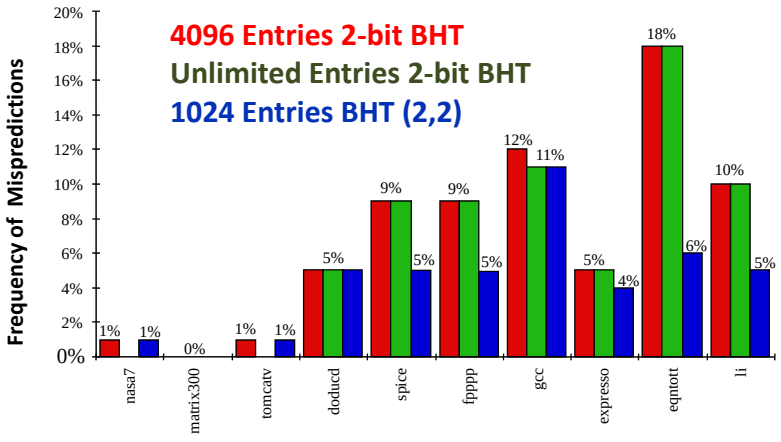
Example: (2,2) predictor

Behavior of recent branches selects between $2^2 = 4$ predictions of next branch, updating just that prediction



35

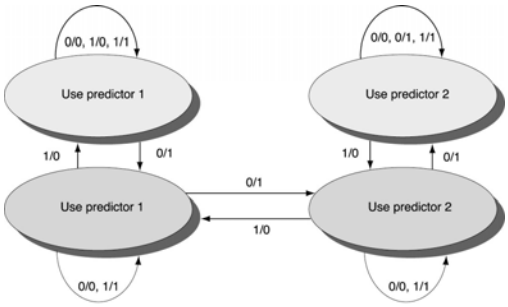
Accuracy of Different Schemes



36

Tournament Predictors

- Multilevel branch predictor
- Use *n*-bit saturating counter to select between predictors
- Usual choice between global and local predictors
- Ability to select the right predictor for a particular branch



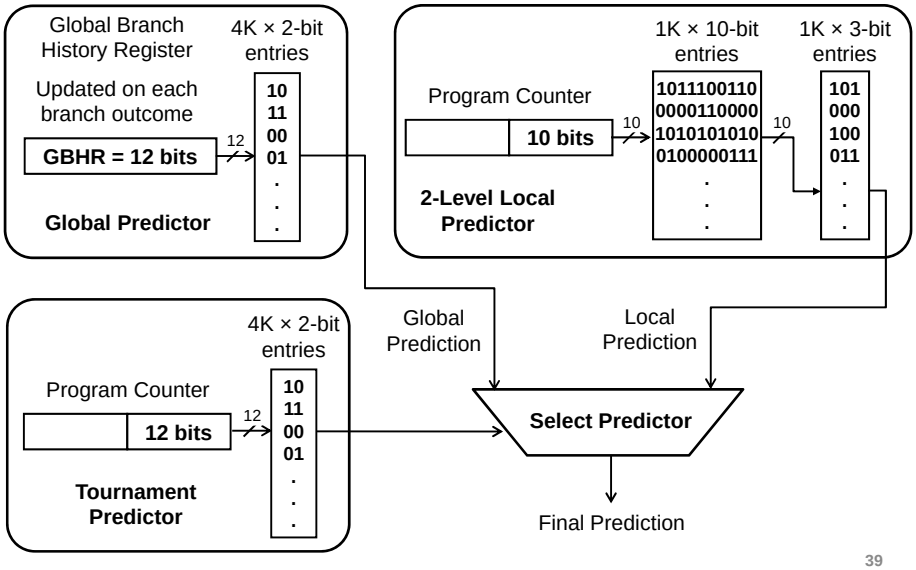
37

Tournament Predictors

- Example of a tournament predictor using 29K bits and used in Alpha 21264, Pentium 4, and Power 5.
- Uses 4K 2-bit counters indexed by local branch address to select between:
- Global predictor
  - 4K entries indexed by history of last 12 branches (2^{12} = 4K)
  - Each entry is a standard 2-bit predictor
- Local predictor is a 2-level predictor
  - Local history table: 1024 10-bit entries recording last 10 branch outcomes, indexed by branch address
  - The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters

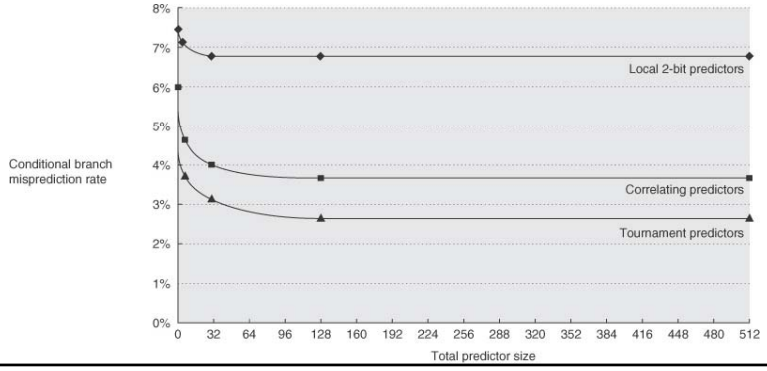
38

Example of a Tournament Predictor

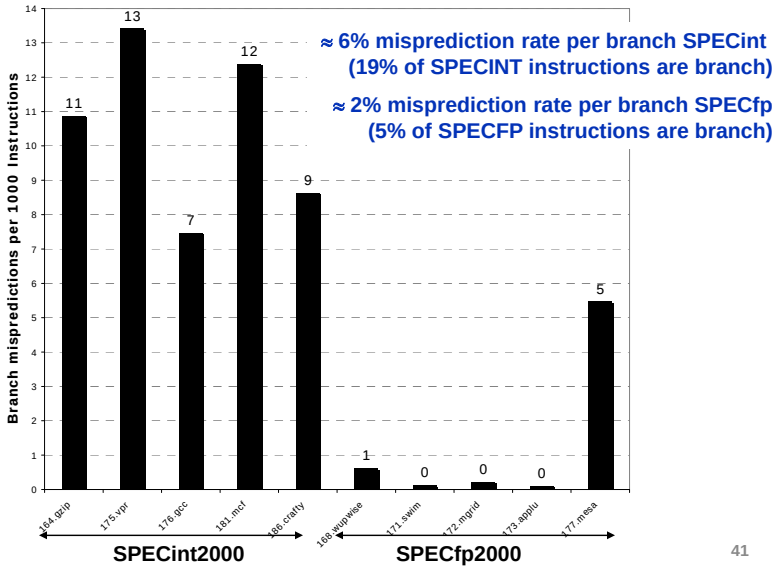


Comparing Predictors

- Advantage of tournament predictor is ability to select the right predictor for a particular branch
  - Particularly crucial for integer benchmarks.
  - A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks



Pentium 4 Misprediction Rate (per 1000 instructions, not per branch)



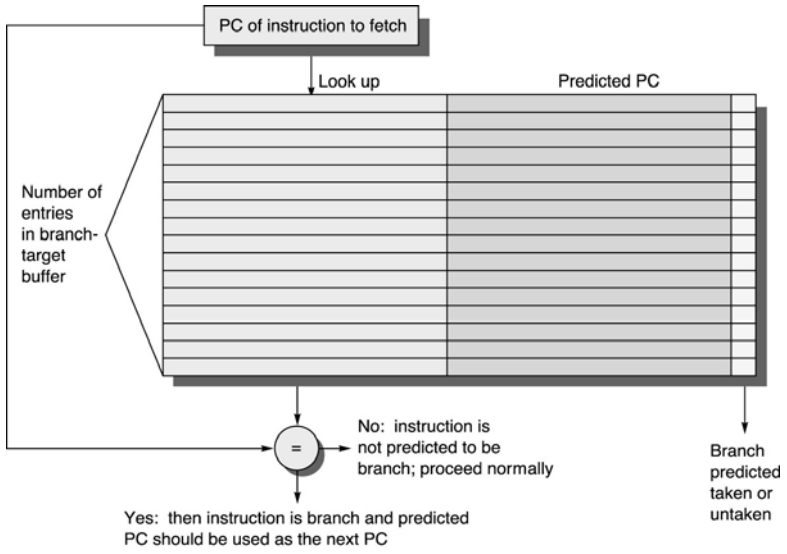
41

Branch Target Buffer (BTB)

- Branch target calculation is costly and stalls the instruction fetch.
- BTB stores Branch Address and Branch Target Address the same way as a cache
- The PC of a branch is sent to the BTB
- When a match is found the corresponding Predicted Branch Target Address is returned
- If the branch was predicted taken, instruction fetch continues at the Branch Target Address

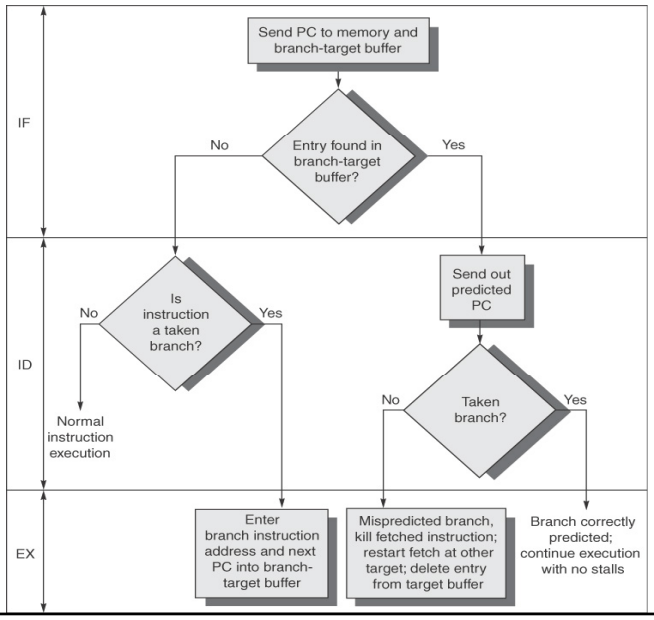
42

Branch Target Buffer



43

Handling an Instruction with a BTB



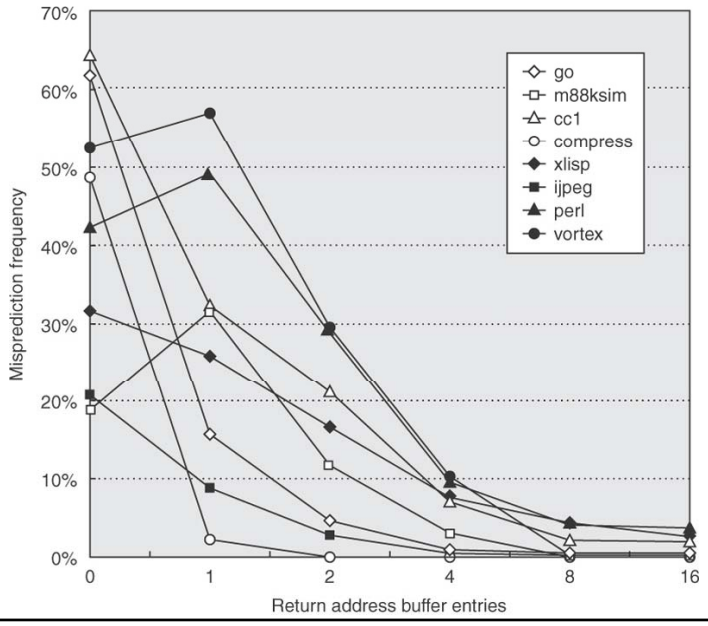
44

Return Address Prediction

- Vast majority of indirect jumps come from procedure returns
- For SPEC benchmarks, procedure returns account about 15% of the branches (most frequent in object oriented languages)
- Predicting procedure return with a BTB is not appropriate
  - One procedure can have many callers
  - Return address can vary from one call to another
- Best to predict return addresses using a return address stack
- Return address stack
  - Caches the most recent return addresses
  - Push a return address on the stack at a call and pop at a return
  - If the cache buffer is sufficiently large, it will predict returns perfectly

45

Prediction Accuracy of Return Addresses



46

Dynamic Branch Prediction Summary

- Prediction becoming important part of execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: predicting a branch based on recently executed branches
  - Global Branch History Register (record last m outcomes)
  - Or even different executions of the same branch
- Tournament predictors take insight to next level, by using multiple predictors
  - usually one based on global information and one based on local information, and combining them with a selector
- Branch Target Buffer to predict branch target address
- Return address stack to predict return address

47

Outline

- ILP
- Loop unrolling
- Branch Prediction
- **Dynamic Scheduling – Tomasulo's Algorithm**
- Reorder Buffer
- CPI less than 1

48

Advantages of Dynamic Scheduling

- Dynamic scheduling - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
- It handles cases when dependences unknown at compile time
  - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
- It allows code that compiled for one pipeline to run efficiently on a different pipeline
- It simplifies the compiler
- Hardware speculation is a technique with significant performance advantages that builds on dynamic scheduling

49

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

- Enables out-of-order execution and allows out-of-order completion (e.g., **SUB.D**)
  - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (in-order issue)
- Distinguish when an instruction *begins execution* and when it *completes execution*; between 2 times, the instruction is *in execution*
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder

50

Example on WAR and WAW Hazards

```
DIV.D     F0, F2, F4
ADD.D     F6, F0, F8
S.D       F6, 0(R1)
SUB.D     F8, F10, F14
MUL.D     F6, F10, F8
```

Anti-dependence between ADD.D and SUB.D

WAR hazard on F8

Output dependence between ADD.D and MUL.D

WAW hazard on F6

WAR and RAW hazards are caused by out-of-order execution, but can be eliminated with register renaming

```
DIV.D     F0, F2, F4
ADD.D     F6, F0, F8
S.D       F6, 0(R1)
SUB.D     T1, F10, F14
MUL.D     T2, F10, T1
```

Use temporary registers T1 and T2 to eliminate name dependences

Register renaming can be done statically by the compiler or dynamically by the pipeline

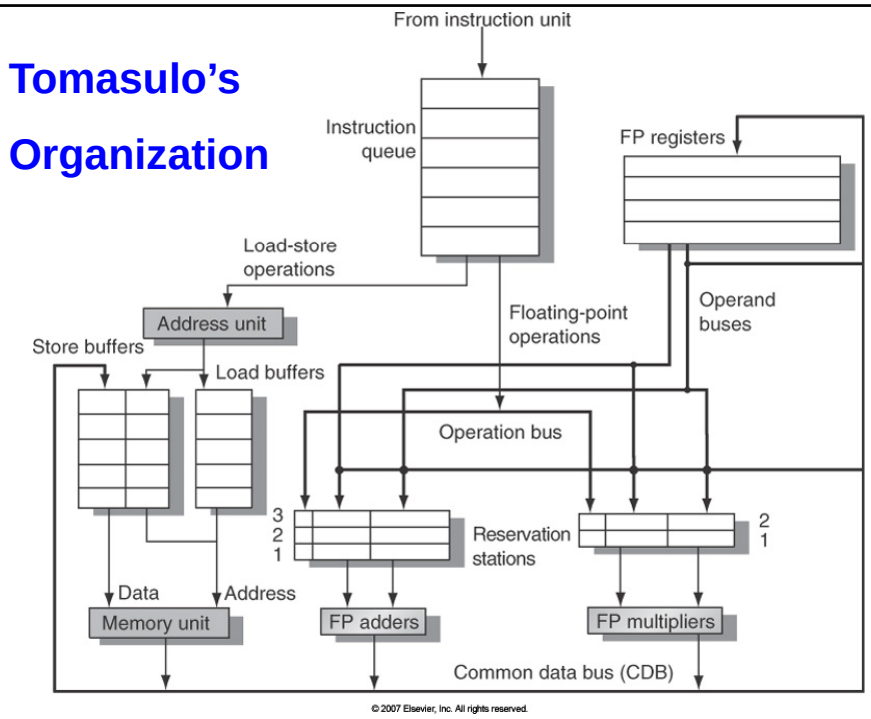
51

Dynamic Scheduling: Tomasulo

- Fast IBM 360/91 for scientific code
  - Completed in 1967
  - Before cache memories
  - Implemented complex memory system
- Pipelined floating point units
  - FP Adder
  - FP Multiplier (Divide done in multiplier)
- Dynamic scheduling in FP unit (Tomasulo)
- The descendants of Tomasulo are found in
  - Alpha 21264, Pentium 4, AMD Opteron, Power 5, etc

52

Tomasulo's Organization



Tomasulo Algorithm

- Buffers & Control distributed with Function Units
  - FU buffers are called "reservation stations" and have pending operands
- Registers in instructions are replaced by values or by pointers to reservation stations, called tags
  - Reservation stations provide renaming to avoid WAR & WAW hazards
- Results are broadcast on the Common Data Bus to all reservation stations, not through registers
  - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as Function Units (FU) with Reservation Stations (RS) as well

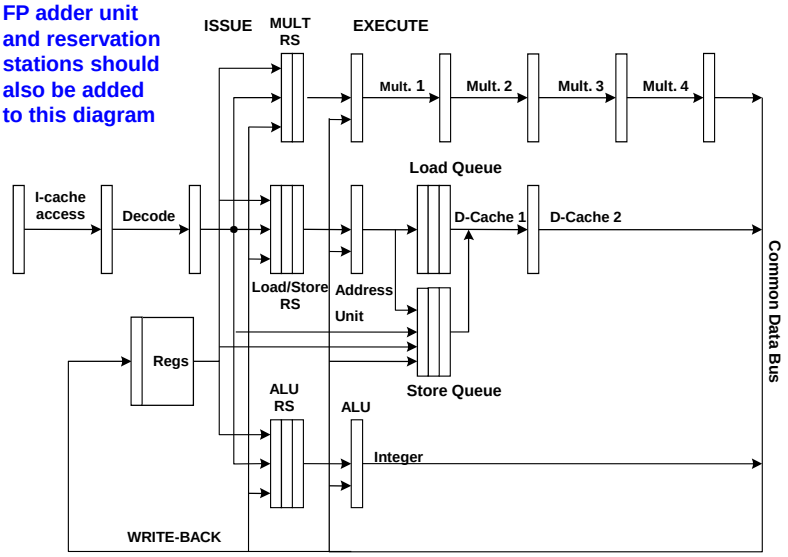
54

Generalized Tomasulo's Organization

- Enhance parallel pipeline architecture
- Apply Tomasulo's algorithm to all pipelined units
  - Not just floating point
  - Biggest difference: handle loads/stores like other instructions
- Use "tags" to identify data values
  - Both tags and register designators can name data
- Reservation Stations (RS) distribute control
  - Set of Reservation Stations per functional unit
  - Tag identifies result of instruction in RS
- Common Data Bus (CDB) broadcasts all results

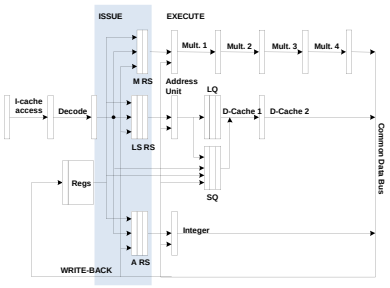
Generalized Tomasulo's Organization

FP adder unit and reservation stations should also be added to this diagram



Three Stages of Tomasulo's Algorithm

1. ISSUE
   - Get next instruction from fetch unit
   - Check for available reservation station
   - If not available, stall due to structural hazard
   - If RS available, *issue*
     - Copy ready registers to RS
     - Copy tags for all non-ready registers to RS

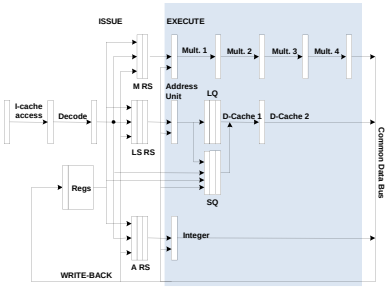


57

Three Stages of Tomasulo's Algorithm

2. Execute
   - If input operands available, issue and begin execution
   - If not, monitor CDB for necessary input operands

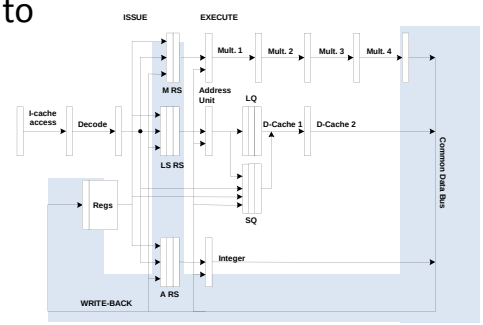
If several instructions become ready for the same functional unit in the same clock cycle, one of them is chosen for execution



58

Three Stages of Tomasulo's Algorithm

3. Write Back
   - If CDB available, write result on CDB
     - All destinations with matching tags receive data
     - CDB broadcasts results to all reservation stations
   - If not, wait for CDB to become available



Reservation Station Components

Op: Operation to perform (e.g., ADD.D or MUL.D)

Vj, Vk: Value of Source operands
- Store buffers has Vj field, data to be stored
- Load and store buffers have A field = memory Address

Qj, Qk (tags): Reservation stations producing result
- If (Qj == 0) then Vj value is present
- If (Qk==0) then Vk value is present
- Store buffers only have Qj for RS producing result

Busy: Indicates reservation station and FU are busy

Register file has a field Qi: Indicates which functional unit will write each register. If (Qi == 0) then no active instruction is computing a result for that register.

Tomasulo Example

Instruction stream

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

FU count down

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
0	FU								

Clock cycle counter

61

Tomasulo Example Cycle 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU			Load1					

62

Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Note: Can have multiple loads outstanding

63

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU	Mult1	Load2		Load1					

- **Note: registers names are removed ("renamed") in Reservation Stations; MULT issued**
- **Load1 completing; what is waiting for Load1?**

64

Tomasulo Example Cycle 4

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1				

- **Load2 completing; what is waiting for Load2?**

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

- **Timer starts down for Add1, Mult1**

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- **Issue ADDD here despite name dependency on F6?**

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- **Add1 (SUBD) completing; what is waiting for it?**

Tomasulo Example Cycle 8

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- **Add2 (ADDD) completing; what is waiting for it?**

Tomasulo Example Cycle 11

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Write result of ADDD here?**
- **All quick instructions complete in this cycle!**

Tomasulo Example Cycle 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
14	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult1 (MULTD) completing; what is waiting for it?**

Tomasulo Example Cycle 16

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Just waiting for Mult2 (DIVD) to complete**

77

Faster than light computation (skip a couple of cycles)

78

Tomasulo Example Cycle 55

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
55	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Mult2 (DIVD) is completing; what is waiting for it?**

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- **Once again: In-order issue, out-of-order execution and out-of-order completion.**

81

How can Tomasulo overlap iterations of loops?

- Register renaming
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- Reservation stations
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall
- Other perspective: Tomasulo building data flow dependency graph on the fly

82

Tomasulo's scheme offers 2 major advantages

1. Distribution of the hazard detection logic
   - distributed reservation stations and the CDB
   - If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
   - If a centralized register file were used, the units would have to read their results from the registers when register buses are available
2. Elimination of stalls for WAW and WAR hazards

83

Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units ⇒high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs ⇒ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

84

Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- **Reorder Buffer**
- CPI less than 1

85

Speculation to 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

86

Speculation to 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

87

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish 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

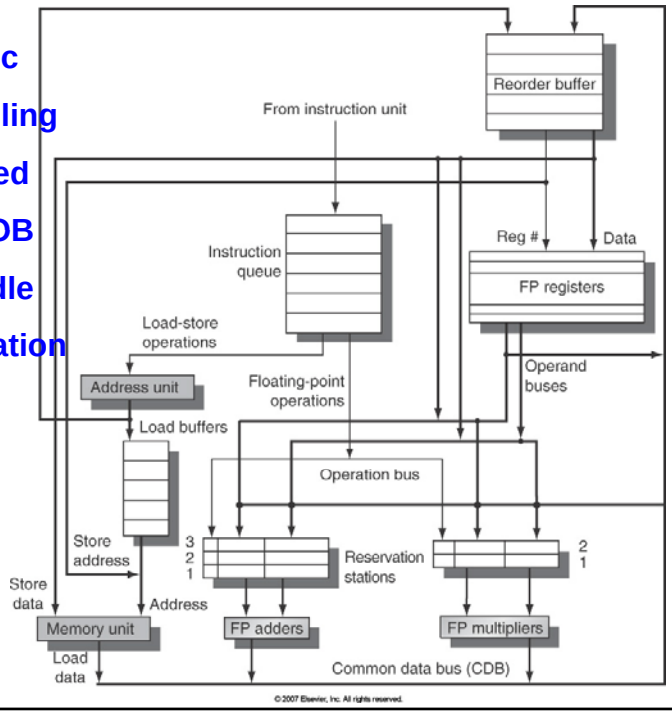
88

Reorder Buffer (ROB)

- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
  - (we know definitively that the instruction should execute)
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
  - ROB extends architectured registers like RS

89

Dynamic Scheduling Extended With ROB To handle Speculation



Reorder Buffer Entry

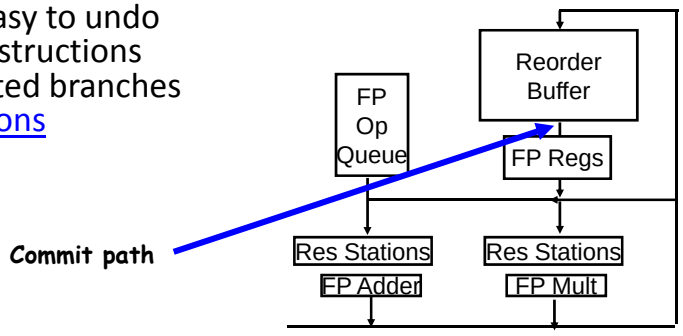
- Each entry in the ROB contains four fields:

1. Instruction type
   - a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
2. Destination
   - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
3. Value
   - Value of instruction result until the instruction commits
4. Ready
   - Indicates that instruction has completed execution, and the value is ready

91

Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
- 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



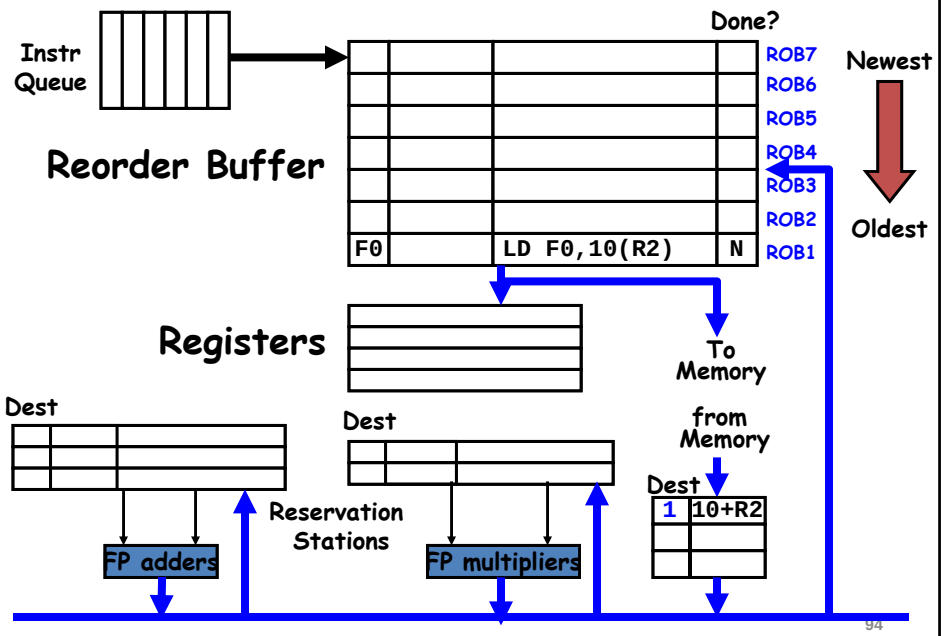
92

4 Steps of Speculative Tomasulo Algorithm

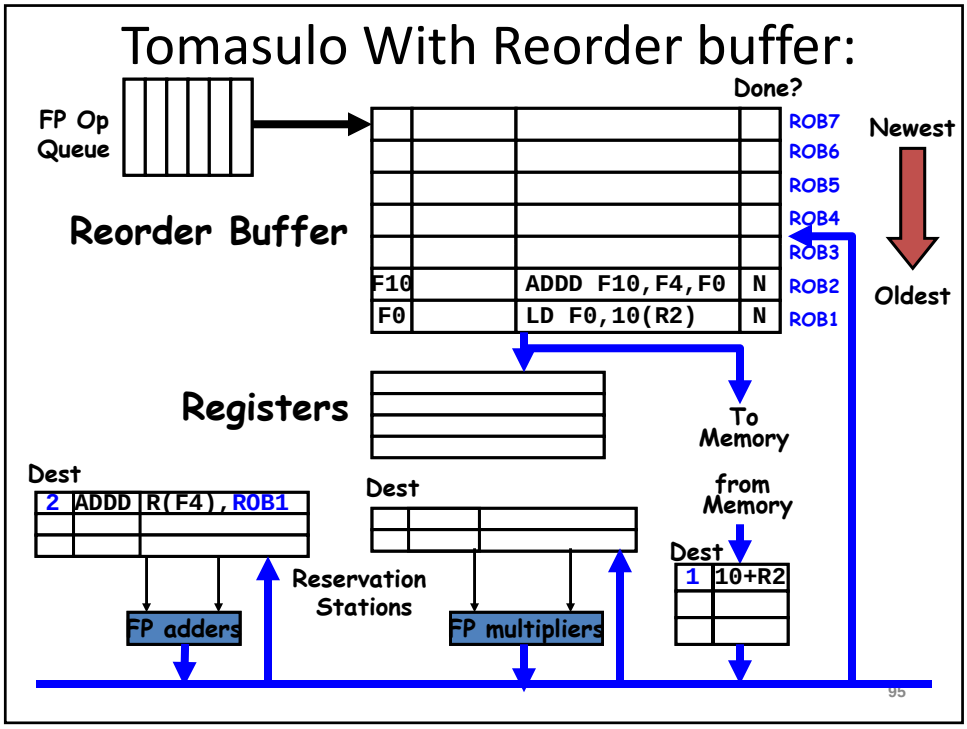
1. Issue—get instruction from Instruction Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. Commit—update register with reorder result
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

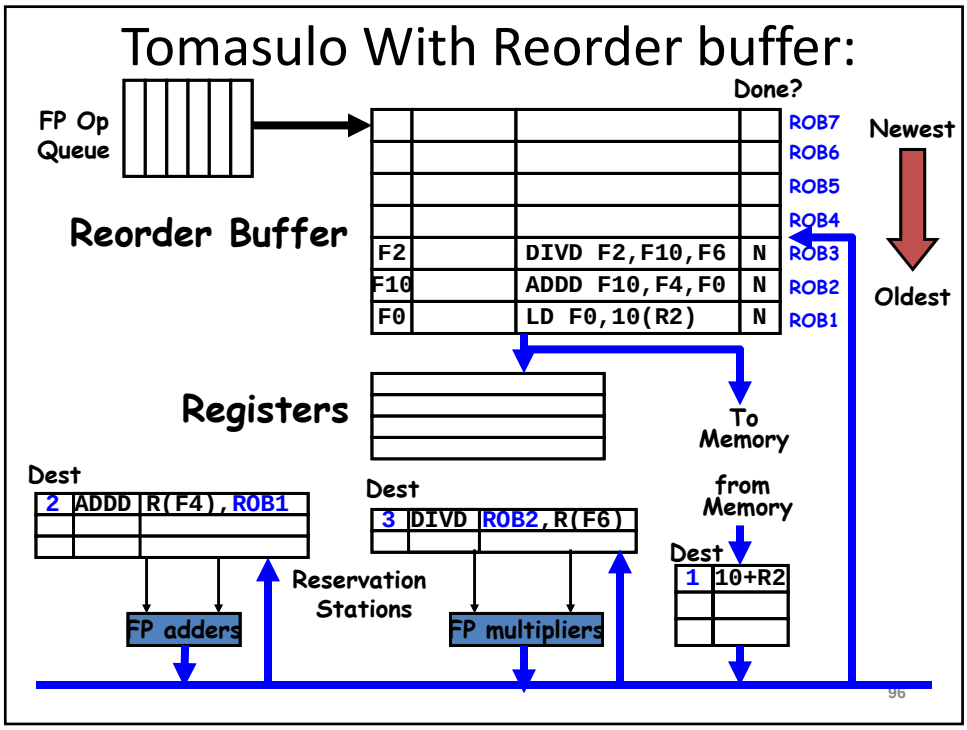
93

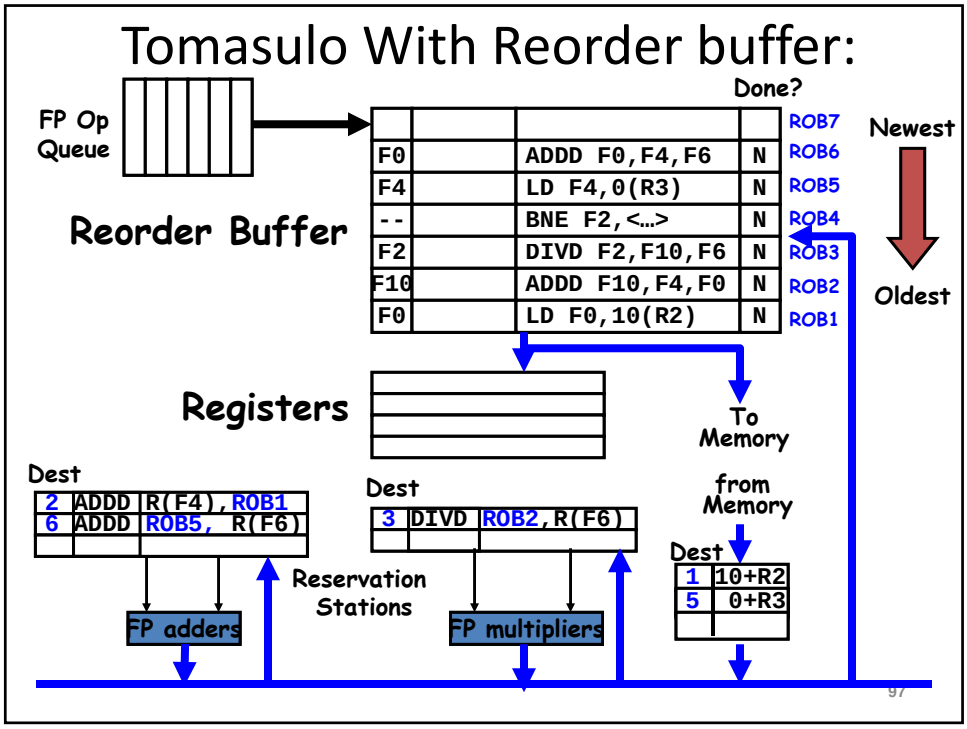
Tomasulo With Reorder buffer:

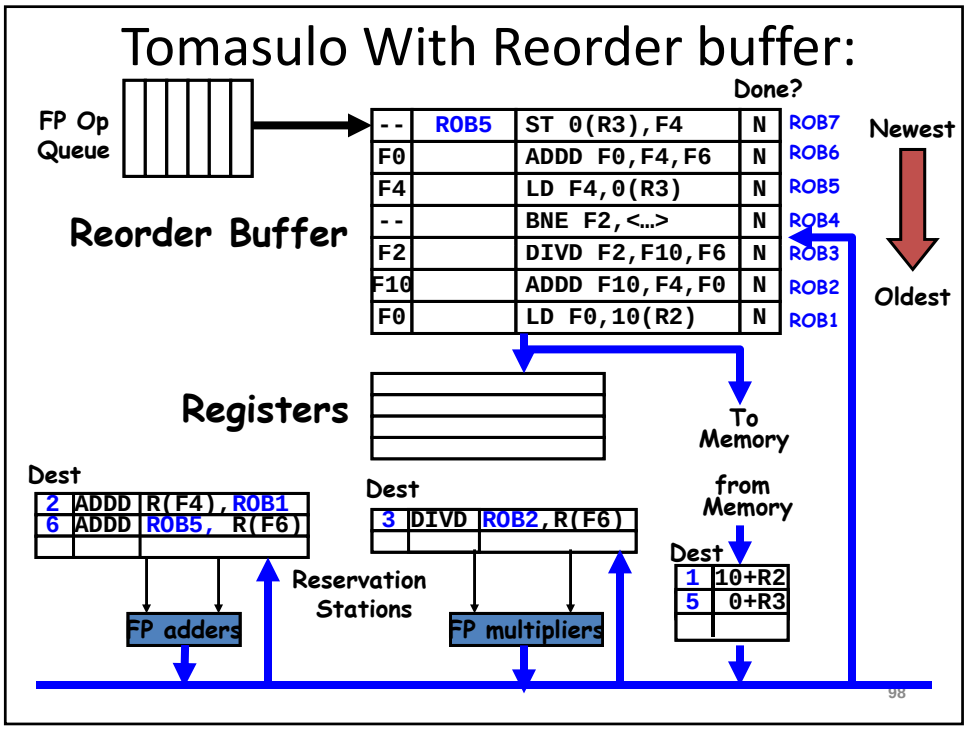


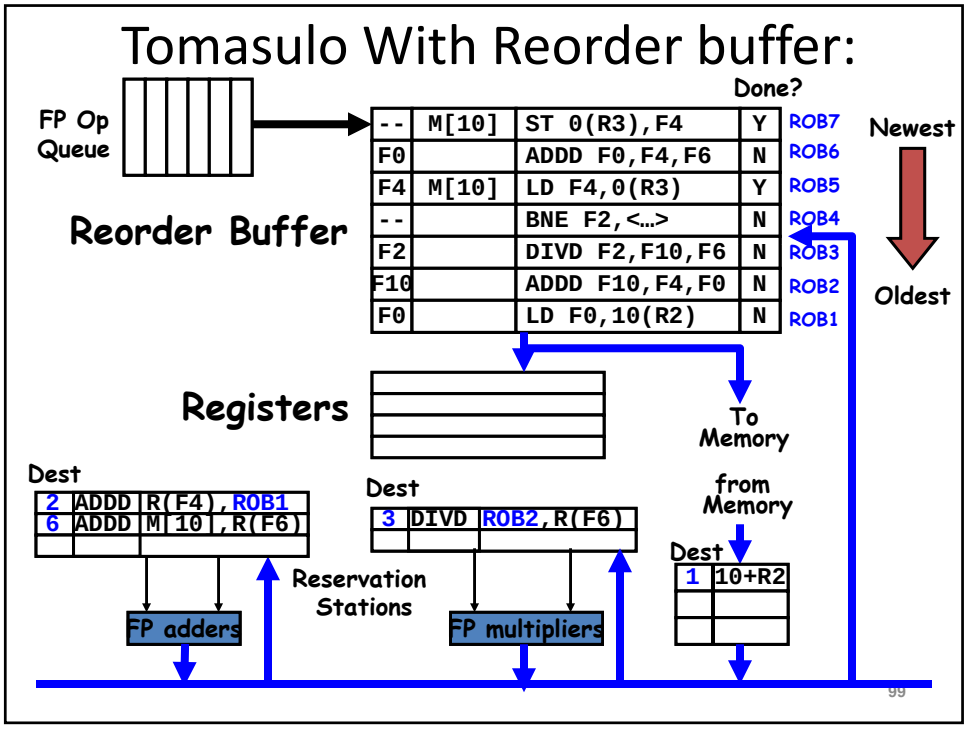
94

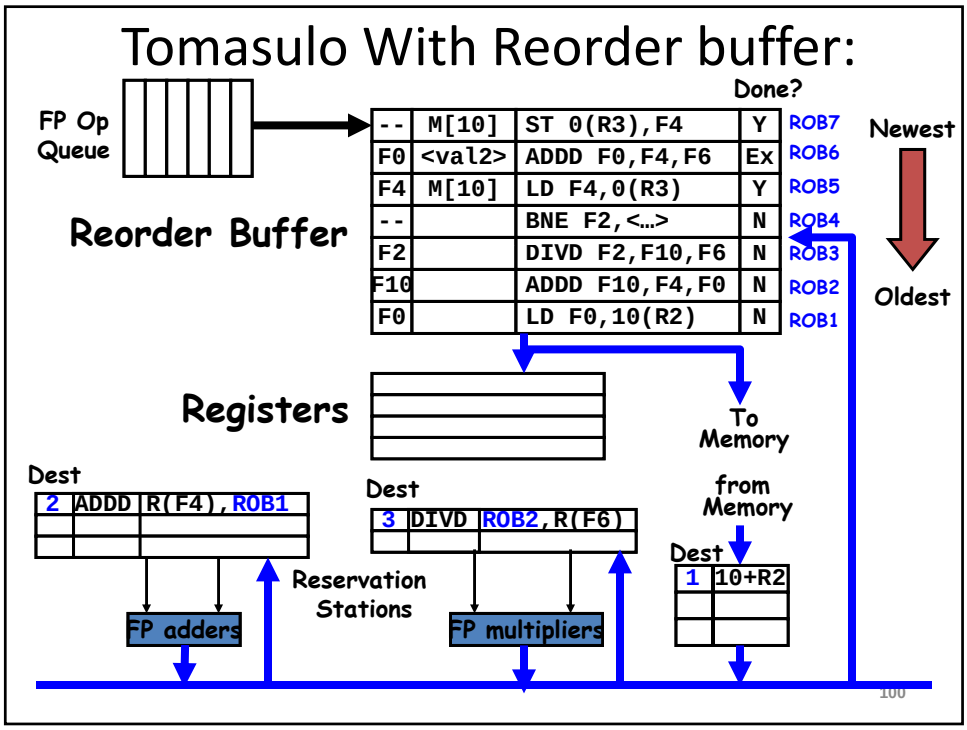

Tomasulo With Reorder buffer:
FP Op Queue
Done?
Newest
Oldest
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Reorder Buffer
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
Reservation Stations
FP adders
FP multipliers
1 10+R2
95

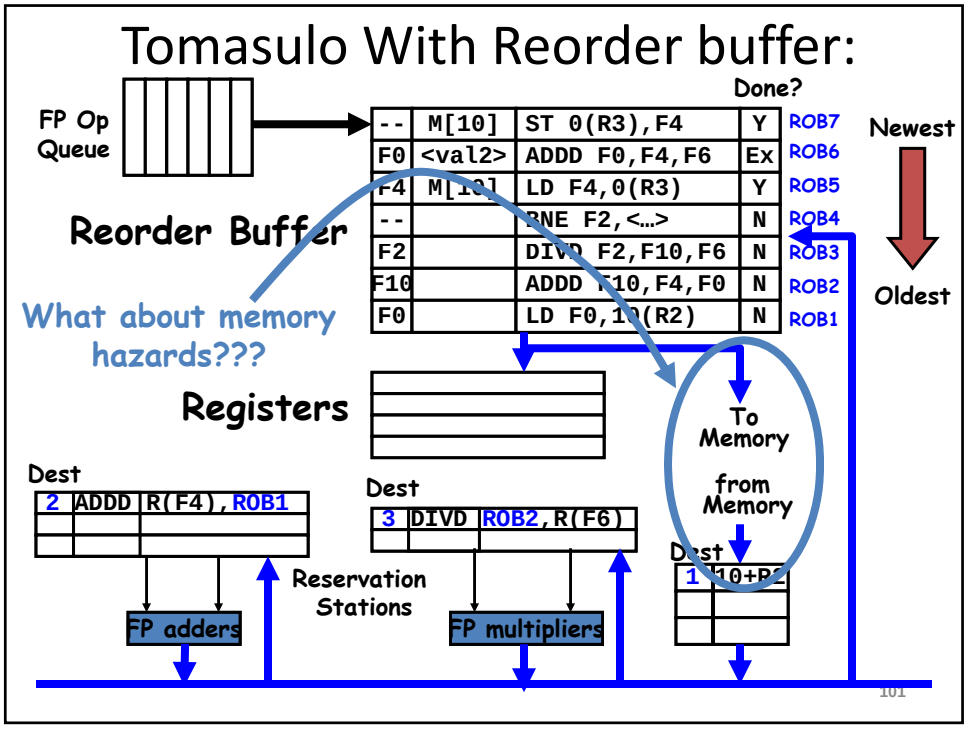

Tomasulo With Reorder buffer:
FP Op Queue
Done?
Newest
Oldest
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
Reorder Buffer
F2 DIVD F2,F10,F6 N
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
3 DIVD ROB2,R(F6)
Reservation Stations
FP adders
FP multipliers
1 10+R2
96


Tomasulo With Reorder buffer:
FP Op Queue
Reorder Buffer
Done?
Newest
Oldest
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
F0 ADDD F0,F4,F6 N
F4 LD F4,0(R3) N
-- BNE F2,<…> N
F2 DIVD F2,F10,F6 N
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
6 ADDD ROB5, R(F6)
3 DIVD ROB2,R(F6)
1 10+R2
5 0+R3
Reservation Stations
FP adders
FP multipliers
97


Tomasulo With Reorder buffer:
FP Op Queue
Reorder Buffer
Done?
Newest
Oldest
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1
-- ROB5 ST 0(R3),F4 N
F0 ADDD F0,F4,F6 N
F4 LD F4,0(R3) N
-- BNE F2,<…> N
F2 DIVD F2,F10,F6 N
F10 ADDD F10,F4,F0 N
F0 LD F0,10(R2) N
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
6 ADDD ROB5, R(F6)
3 DIVD ROB2,R(F6)
1 10+R2
5 0+R3
Reservation Stations
FP adders
FP multipliers
98


Tomasulo With Reorder buffer:
FP Op Queue
Reorder Buffer
Done?
-- M[10] ST 0(R3),F4 Y ROB7
F0 ADDD F0,F4,F6 N ROB6
F4 M[10] LD F4,0(R3) Y ROB5
-- BNE F2,<…> N ROB4
F2 DIVD F2,F10,F6 N ROB3
F10 ADDD F10,F4,F0 N ROB2
F0 LD F0,10(R2) N ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
6 ADDD M[10],R(F6)
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers
99


Tomasulo With Reorder buffer:
FP Op Queue
Reorder Buffer
Done?
-- M[10] ST 0(R3),F4 Y ROB7
F0 <val2> ADDD F0,F4,F6 Ex ROB6
F4 M[10] LD F4,0(R3) Y ROB5
-- BNE F2,<…> N ROB4
F2 DIVD F2,F10,F6 N ROB3
F10 ADDD F10,F4,F0 N ROB2
F0 LD F0,10(R2) N ROB1
Newest
Oldest
Registers
To Memory
from Memory
Dest
2 ADDD R(F4),ROB1
Dest
3 DIVD ROB2,R(F6)
Dest
1 10+R2
Reservation Stations
FP adders
FP multipliers
100



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

102

Exceptions and Interrupts

- IBM 360/91 invented "imprecise interrupts"
  - Computer stopped at this PC; its likely close to this address
  - Not so popular with programmers
  - Also, what about Virtual Memory? (Not in IBM 360)
- Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
  - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  - This is exactly same as need to do with precise exceptions
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
  - If a speculated instruction raises an exception, the exception is recorded in the ROB
  - This is why reorder buffers in all new processors

103

Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- **CPI less than 1**

104

Getting CPI below 1

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  1. statically-scheduled superscalar processors,
  2. dynamically-scheduled superscalar processors, and
  3. VLIW (very long instruction word) processors
- 2 types of superscalar processors issue varying numbers of instructions per clock
  - use in-order execution if they are statically scheduled, or
  - out-of-order execution if they are dynamically scheduled
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)

105

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

106

Recall: Unrolled Loop that Minimizes Stalls for Scalar

```
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      BNEZ   R1,LOOP
14      S.D    8(R1),F16    ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

14 clock cycles, or 3.5 per iteration

107

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				7
S.D -32(R1),F20	S.D -40(R1),F24			DSUBUI R1,R1,#48	8
S.D -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

108

Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

109

Intel/HP IA-64 "Explicitly Parallel Instruction Computer (EPIC)"

- IA-64: instruction set architecture
- 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) => 40% fewer mispredictions?
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

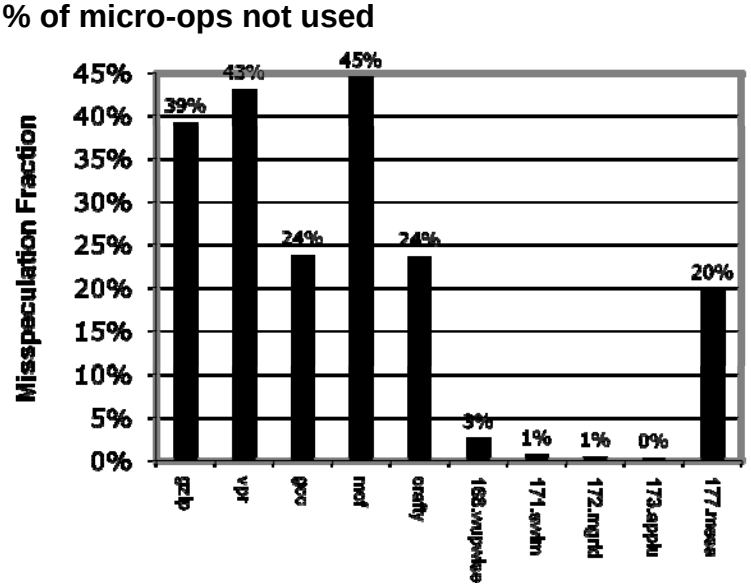
110

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

111

(Mis) Speculation on Pentium 4



112

Perspective

- Interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Processors of last 5 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units $\Rightarrow$ performance 8 to 16X
- Peak v. delivered performance gap increasing

113