EEL 5764 Graduate Computer Architecture

Chapter 2 - Instruction Level Parallelism

Ann Gordon-Ross
Electrical and Computer Engineering
University of Florida

http://www.ann.ece.ufl.edu/

These slides are provided by:
David Patterson
Electrical Engineering and Computer Sciences, University of California, Berkeley
Modifications/additions have been made from the originals

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm

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)

Instruction Level Parallelism

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

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP 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];
```

Loop-Level Parallelism

- Exploit loop-level parallelism to parallelism by “unrolling loop” either by

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

- 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

Data Dependence and Hazards

- Instr_J is data dependent (aka true dependence) on Instr_I:

1. Instr_J tries to read operand before Instr_I writes it

```
I: add r1,r2,r3          I: add r1, r2, r3
J: sub r4,r1,r3          K: add r3, r2, r1
                         J: sub, r4, r5, r3
```

2. or Instr_J is data dependent on Instr_K which is dependent on Instr_I

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence ⇒ data dependence in source code ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard
- *Dependencies are independent of the pipeline, hazards are dependent on the pipeline*

ILP and Data Dependencies,Hazards

- HW/SW must preserve program order:
  - Must have same outcome as if executed sequentially as determined by the original source code
  - Dependences 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 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
  - As long as the results are the same, execute in any order

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
- Bad if Instr_J writes operand *before* Instr_I reads it

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

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

- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- Bad if Instr_J writes operand *before* Instr_I writes it.

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

- Called an “output dependence” by compiler writers This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline, 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 regs
  - Either by compiler or by HW

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`.

Control Dependence Ignored

- Control dependence need not be preserved, but results must be correct
  - 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 critical to program correctness are

  1) exception behavior and

  2) data flow

Exception Behavior

- Preserving exception behavior
  ⇒ any changes in instruction execution order must not change how exceptions are raised in program
  (⇒ no new exceptions)
- Example:

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

  – (Assume branches not delayed)
- Problem with moving `LW` before `BEQZ`?

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

Computers in the News

Who said this?

A. Jimmy Carter, 1979
B. Bill Clinton, 1996
C. Al Gore, 2000
D. George W. Bush, 2006

"Again, I'd repeat to you that if we can remain the most competitive nation in the world, it will benefit the worker here in America. People have got to understand, when we talk about spending your taxpayers' money on research and development, there is a correlating benefit, particularly to your children. See, it takes a while for some of the investments that are being made with government dollars to come to market. I don't know if people realize this, but the Internet began as the Defense Department project to improve military communications. In other words, we were trying to figure out how to better communicate, here was research money spent, and as a result of this sound investment, the Internet came to be.

The Internet has changed us. It's changed the whole world."

FEBRUARY 13, 2006
TIME
FURY OVER MUHAMMAD
IS AMERICA FLUNKING SCIENCE?
Our superiority was once the envy of the world. But we are slacking off just as other countries are getting stronger. What's the formula for a comeback?
BY MICHAEL D. LEMONICK
www.time.com AOL Keyword: TIME

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm

Software Techniques - Loop Unrolling 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 Cycles	Stalls between in cycles
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0
ALU op	Branch	1	1

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  - -To simplify, assume 8 is lowest address
  - R1 is loop counter initialized to 8000

```
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
```

Double precision so decrement by 8 (instead of 4)

FP Loop - Where are the stalls?

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

Assumption: no cache misses

Instruction producing result	Instruction using result	Latency in cycles	Stalls between in cycles
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
ALU op	Branch	1	1

- 9 clock cycles per element: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
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 DADDUI
7           BNEZ     R1, Loop
```

Move up add to hide stall..

And remove stall

Instruction producing result	Instruction using result	Latency in cycles	Stalls between in cycles
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
ALU op	Branch	1	1

7 clock cycles per element, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?

But this is hard for the compiler to do!!

Unroll Loop Four Times (straightforward way) - Loop Speedup

```
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, F10, F2
24          S.D      -24(R1), F16    ;drop DSUBUI & BNEZ
25          DADDUI   R1, R1, #-32    ;alter to 4*8
27          BNEZ     R1, Loop
```

Rewrite loop to minimize stalls?

1 cycle stall

2 cycles stall

27 clock cycles or 6.75 per element (dropped instructions, not stalls) (Assumes R1 is a multiple of 4)

But we have made the basic block bigger…more ILP!!

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         BNEZ      R1,LOOP
```

Group instructions to remove stalls

14 clock cycles, or 3.5 per element due to unrolling and rescheduling

Unrolled Loop Detail

- Assumption: Upper bound is known - not realistic
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Solution - 2 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

5 Loop Unrolling Decisions

- Hard for compiler - easy for humans. Compilers must be sophiticated:

1. Is loop unrolling useful? Are iterations independent
2. Are there enough registers? Need to avoid added data hazards by using the 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 from different iterations are independent
   - Memory analysis to determine that they do not refer to same address pointers make things more difficult.
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling - How Much Benefit Do We Get???

1. Diminishing returns as unrolling gets larger
   - How much more benefit going from 4 to 8?
   - Not much - Amdahl's Law
2. Growth in code size
   - Increase I-cache miss rate with larger loops
3. Register pressure: not enough registers for aggressive unrolling and scheduling
   - May need to store live values in memory

- But…..Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

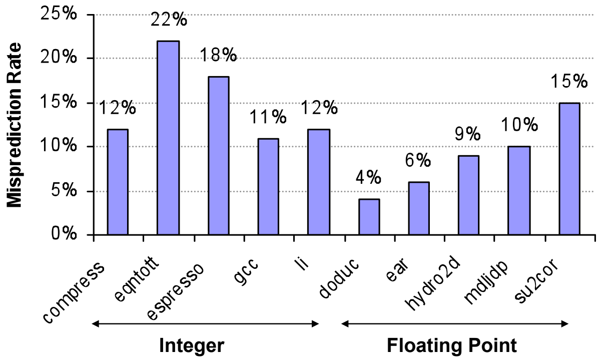
Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm

Static Branch Prediction

- Earlier lecture showed scheduling code around delayed branch - Where do we get instructions?
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC

- More accurate schemes use profile information



Dynamic Branch Prediction

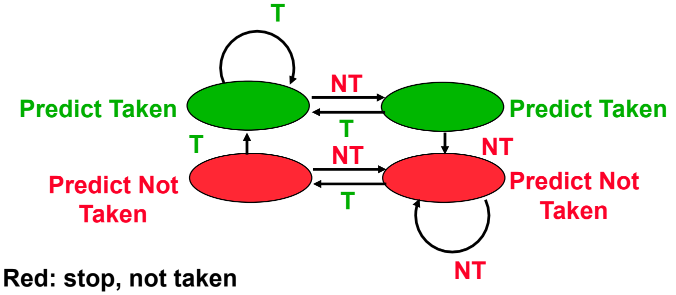
- Better approach
  - Hard to get accurate profile for static prediction
- Why does prediction work?
  - Regularities
    - » Underlying algorithm
    - » Data that is being operated
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs which have dynamic behavior

Dynamic Branch Prediction

- Performance is based on a function of accuracy and cost of misprediction
- Simple scheme - Branch History Table
  - Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check, just hint
  - Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 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
    - » Worse than always predicting taken

Dynamic Branch Prediction

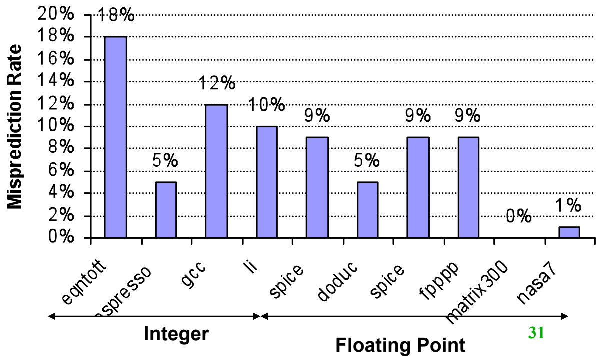
- How do we make dynamic branch prediction better?
  - Solution: 2-bit scheme where change prediction only if get misprediction *twice*



- Red: stop, not taken
- Green: go, taken
- Adds ***history*** to decision making process
- Simple but quite effective

BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Address conflicts - got branch history of wrong branch when index the table
- 4096 entry table:



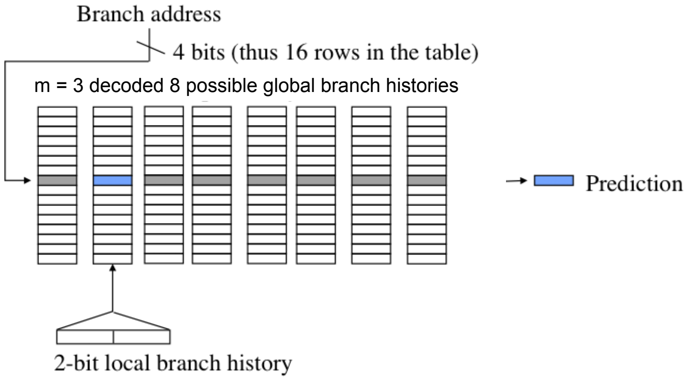
Correlated Branch Prediction

- Idea: correlate prediction based on recent branch history of previous branches
  - 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 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 (branch address) has *m* *n*-bit predictors.

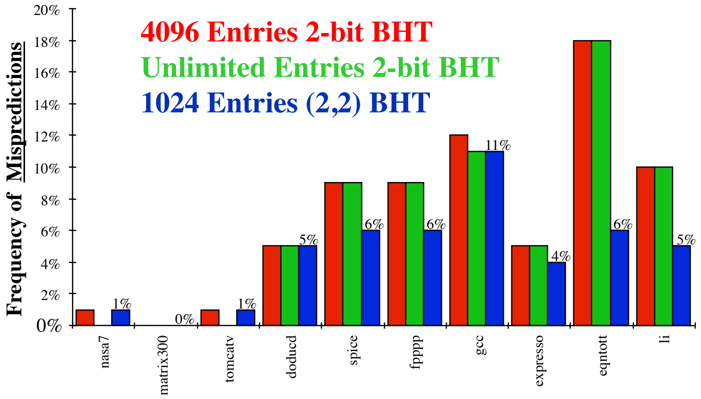
Correlating Branches

(3,2) predictor

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

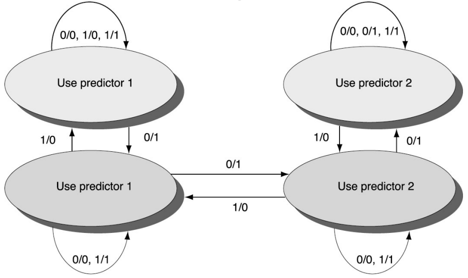

Branch address
4 bits (thus 16 rows in the table)
m = 3 decoded 8 possible global branch histories
Prediction
2-bit local branch history

Accuracy of Different Schemes

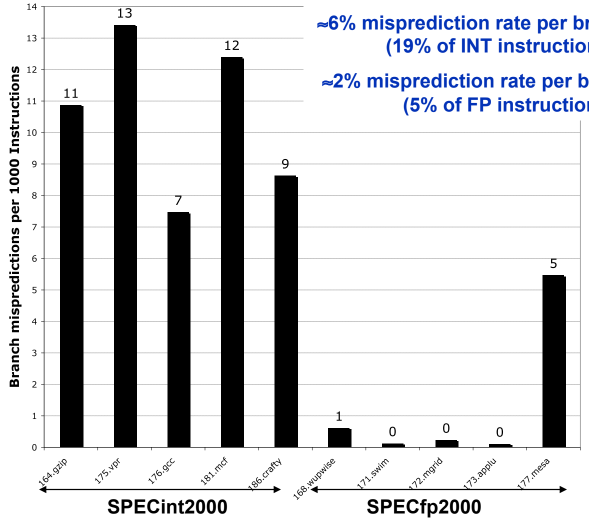


Tournament Predictors

- **Success of correlating branch prediction lead to tournament predictors**
  - **Multilevel branch predictor**
  - **Use *n*-bit saturating counter to choose between competing predictors - may the best predictor win**
- **Usual choice between global and local predictors**



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



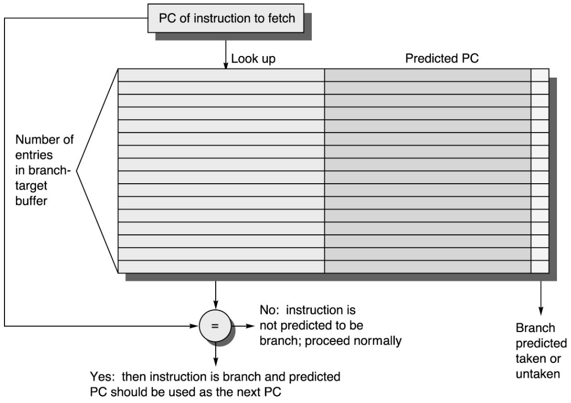
≈6% misprediction rate per branch SPECint (19% of INT instructions are branch)

≈2% misprediction rate per branch SPECfp (5% of FP instructions are branch)

Branch Target Buffers (BTB)

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

Branch Target Buffers


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

Dynamic Branch Prediction Summary

- Prediction becoming important part of execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
  - Either different branches (GA)
  - Or different executions of same branches (PA)
- 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
  - In 2006, tournament predictors using ≈ 30K bits are in processors like the Power5 and Pentium 4

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm

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
  - Hide cache misses by executing other code while waiting for the miss to resolve
- No recompiling - It allows code that compiled for one pipeline to run efficiently on a different pipeline
- It simplifies the compiler
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed

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

Division is slow, addd must wait but subd doesn't have to

- Enables out-of-order execution and allows out-of-order completion (e.g., SUBD)
  - Issue stage in order (in-order issue)
- Three instruction phases
  - *begins execution*
  - *completes execution*
  - *in execution* - between above 2 stages
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder

Dynamic Scheduling Step 1

- Instruction Decode (ID), also called Instruction Issue
- 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

A Dynamic Algorithm: Tomasulo's

- For IBM 360/91 (before caches!)
  - ⇒ Long memory latency
- Goal: High Performance without special compilers
  - Same code for many different models
- BIG LIMITATION - 4 floating point registers limited compiler ILP
  - Need more effective registers — renaming in hardware!
- Original algorithm focused on FP, but applicable to integer instructions
  - FP were slow, so wanted int instructions to go ahead
- Why Study 1966 Computer?
- The descendants of this have flourished!
  - Alpha 21264, Pentium 4, AMD Opteron, Power 5, …

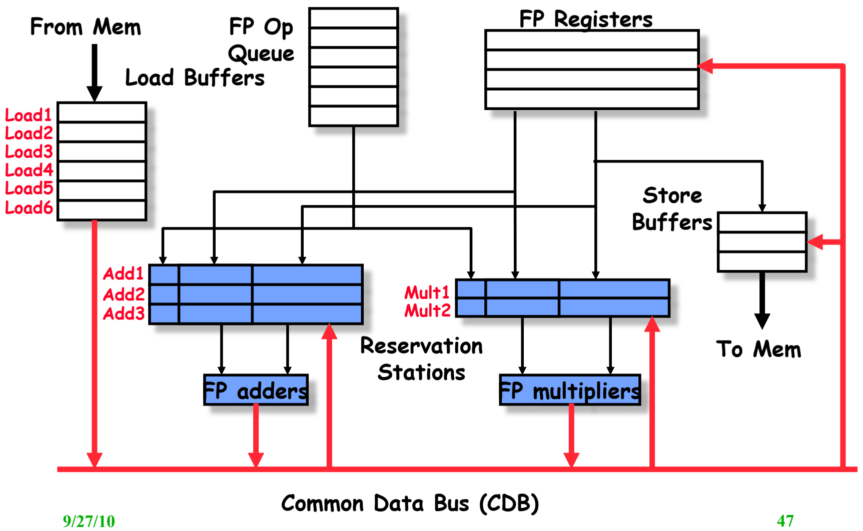
Tomasulo Algorithm

- Control & buffers distributed with Function Units (FU)
  - Instead of centralized register file, shift data to a buffer at each FU
  - FU buffers called "reservation stations"; hold operands for pending operations and the instruction
- Registers in instructions (held in the buffers) replaced by actual values or a pointer to reservation stations (RS) that will eventually hold the value - called register renaming
  - Register file only accessed once, then wait on RS values
  - Renaming avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't

Tomasulo Algorithm

- Results go directly to FU through RS, not through register file, over Common Data Bus (CDB) that broadcasts results to all FU RSs
  - Avoids RAW hazards by executing an instruction only when its operands are available
  - Register file not a bottleneck
- Load and Stores treated as FUs with RSs as well

Tomasulo Organization



Reservation Station Components

Op: Operation to perform in the unit (e.g., + or –)

Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue

   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).

2. **Execute**—operate on operands (EX)

   When both operands ready then execute; if not ready, watch Common Data Bus for result

3. **Write result**—finish execution (WB)

   Write on Common Data Bus to all awaiting units; mark reservation station available

   Difference between:

   – Normal data bus: data + destination ("go to" bus)

   – Common data bus: data + source ("come from" bus)

   » Write if matches expected Functional Unit (produces result)

   » Does the broadcast

- Example speed: 2 clocks for Fl .pt. +,-; 2 for load/store; 10 for * ; 40 clks for /

Tomasulo Example

Instruction stream

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

3 Load/Buffers

Reservation Stations:

Time	Name	Busy	OP	S1 Vj	S2 Vk	RS Qj	RS Qk	Status
	Add1							
	Add2							
	Add3							
	Mult1							
	Mult2							

FU count down

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

Clock

0

Clock cycle counter

Register Result Status:

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

Tomasulo Example – Cycle 1

Assuming R2 = 6

Instruction Status:

Inst		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	Status
Load1	Yes	40	Issue
Load2	No		
Load3	No		

Reservation Stations:

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

Clock

1

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU			2	Load 1					

Tomasulo Example – Cycle 2

Assuming R3 = 1

Instruction Status:

Inst		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	Status
Load1	Yes	40	Exe 1
Load2	Yes	46	Issue
Load3	No		

Reservation Stations:

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

Clock

2

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU		Load 2	2	Load 1					

Note: Can have multiple loads outstanding

Tomasulo Example – Cycle 3

Assuming R3 = 1

Instruction Status:

					Exec	Write
Inst		j	k	Issue	Comp	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	Status
Load1	Yes	40	Exe 2
Load2	Yes	46	Exe 1
Load3	No		

Note: registers names are removed ("renamed") in Reservation Stations; MULT issued

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	Yes	MULTD		2	Load2		Issue
	Mult2	No						

Clock

3

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Load 2	2	Load 1					

Load1 completing; what is waiting for Load1?

Tomasulo Example – Cycle 4

Assuming M[40] = 10

Instruction Status:

					Exec	Write
Inst		j	k	Issue	Comp	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	Status
Load1	Yes	40	Commit
Load2	Yes	46	Exe 2
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	Yes	SUBD	10			Load2	Issue
	Add2	No						
	Add3	No						
	Mult1	Yes	MULTD		2	Load2		Waiting
	Mult2	No						

Clock

4

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Load 2	2	10	Add1				

- Load2 completing; what is waiting for Load2?

Tomasulo Example – Cycle 5

Assuming M[46] = 3

Instruction Status:

					Exec	Write
Inst		j	k	Issue	Comp	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	Status
Load1	No		
Load2	Yes	46	Commit
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
2	Add1	Yes	SUBD	10	3			Ready
	Add2	No						
	Add3	No						
10	Mult1	Yes	MULTD	3	2			Ready
	Mult2	Yes	DIVD		10	Mult1		Issue

Clock

5

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	3	2	10	Add1	Mult2			

- Timer starts down for Add1, Mult1

Tomasulo Example – Cycle 6

Instruction Status:

					Exec	Write
Inst		j	k	Issue	Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
1	Add1	Yes	SUBD	10	3			Exe1
	Add2	Yes	ADDD		3	Add1		Issue
	Add3	No						
9	Mult1	Yes	MULTD	3	2			Exe1
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

6

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	3	2	Add2	Add1	Mult2			

- Issue ADDD here despite name dependency on F6?

Tomasulo Example – Cycle 7

Instruction Status:

					Exec	Write
Inst	j	k		Issue	Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
0	Add1	Yes	SUBD	10	3			Exe2
	Add2	Yes	ADDD		3	Add1		Waiting
	Add3	No						
8	Mult1	Yes	MULTD	3	2			Exe2
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

7

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	Add2	Add1	Mult2			

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

Tomasulo Example – Cycle 8

Instruction Status:

					Exec	Write
Inst	j	k		Issue	Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	Yes	SUBD	10	3			Commit
2	Add2	Yes	ADDD	7	3			Ready
	Add3	No						
7	Mult1	Yes	MULTD	3	2			Exe3
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

8

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	Add2	7	Mult2			

Tomasulo Example – Cycle 9

Instruction Status:

					Exec	Write
Inst	j	k		Issue	Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
1	Add2	Yes	ADDD	7	3			Exe1
	Add3	No						
6	Mult1	Yes	MULTD	3	2			Exe4
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

9

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	Add2	7	Mult2			

Tomasulo Example – Cycle 10

Instruction Status:

					Exec	Write
Inst	j	k		Issue	Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
0	Add2	Yes	ADDD	7	3			Exe2
	Add3	No						
5	Mult1	Yes	MULTD	3	2			Exe5
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

10

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	Add2	7	Mult2			

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

Tomasulo Example – Cycle 11

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

Time	Name	Busy	OP	S1 Vj	S2 Vk	RS Qj	RS Qk	Status
	Add1	No						
	Add2	Yes	ADDD	7	3			Commit
	Add3	No						
4	Mult1	Yes	MULTD	3	2			Exe6
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

11

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	10	7	Mult2			

Tomasulo Example – Cycle 12

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

Time	Name	Busy	OP	S1 Vj	S2 Vk	RS Qj	RS Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
3	Mult1	Yes	MULTD	3	2			Exe7
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

12

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	10	7	Mult2			

Tomasulo Example – Cycle 13

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

Time	Name	Busy	OP	S1 Vj	S2 Vk	RS Qj	RS Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
2	Mult1	Yes	MULTD	3	2			Exe8
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

13

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	10	7	Mult2			

Tomasulo Example – Cycle 14

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

Time	Name	Busy	OP	S1 Vj	S2 Vk	RS Qj	RS Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
1	Mult1	Yes	MULTD	3	2			Exe9
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

14

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	3	2	10	7	Mult2			

Tomasulo Example – Cycle 15

Instruction Status:

Inst		j	k	Issue	Exec	Write
					Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
0	Mult1	Yes	MULTD	3	2			Exe10
	Mult2	Yes	DIVD		10	Mult1		Waiting

Clock

15

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	3	2	10	7	Mult2			

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

Tomasulo Example – Cycle 16

Instruction Status:

Inst		j	k	Issue	Exec	Write
					Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	Yes	MULTD	3	2			Commit
40	Mult2	Yes	DIVD	6	10			Ready

Clock

16

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	6	3	2	10	7	Mult2			

- Just waiting for Mult2 (DIVD) to complete

Faster than light computation (skip a couple of cycles)

Tomasulo Example – Cycle 55

Instruction Status:

Inst		j	k	Issue	Exec	Write
					Comp	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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	No						
1	Mult2	Yes	DIVD	6	10			Exe39

Clock

55

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	6	3	2	10	7	Mult2			

Tomasulo Example – Cycle 56

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	No						
0	Mult2	Yes	DIVD	6	10			Exe40

Clock

56

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	6	3	2	10	7	Mult2			

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

Tomasulo Example – Cycle 57

Instruction Status:

Inst		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	Status
Load1	No		
Load2	No		
Load3	No		

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

Reservation Stations:

				S1	S2	RS	RS	
Time	Name	Busy	OP	Vj	Vk	Qj	Qk	Status
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	No						
	Mult2	Yes	DIVD	6	10			Commit

Clock

57

Register Result Status:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	6	3	2	10	7	0.6			

Why 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**

Tomasulo's scheme offers 2 major advantages

1. Distribution of the hazard detection logic
   - distributed reservation stations and the CDB
   - Simultaneous instruction release - If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
   - Don't have to wait on centralized register file
     - » 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

Tomasulo Drawbacks

- 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

Outline

- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Speculation to greater ILP

- How do we get *greater* ILP:
  - Overcome control dependence by hw speculating outcome of branches
    - » Execute program as if guesses were correct
  - 2 methods:
    - » Dynamic scheduling ⇒ only fetches and issues instructions
    - » Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
- Essentially a data flow execution model: Operations execute as soon as their operands are available

Speculation to greater ILP

- What do we need?
  - 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

Outline

- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Adding Speculation to Tomasulo

- Separate execution from finishing
  - This additional step called instruction commit
- Update register file/memory only when instruction is no longer speculative
- Additional requirements - reorder buffer (ROB)
  - Set of buffers to hold results of instructions that have finished execution but have not committed
  - Also used to pass results among instructions that may be speculated

Reorder Buffer (ROB)

- In Tomasulo's algorithm, results are written to the register file after an instruction is finished
- With speculation, the register file is not updated until the instruction commits
  - (we know definitively that the instruction should execute)
- But instruction cannot commit until it is no longer speculative
- ROB stores results while instruction is still speculative
  - Like reservation stations, ROB is a source of operands
  - ROB extends architectural registers like RS

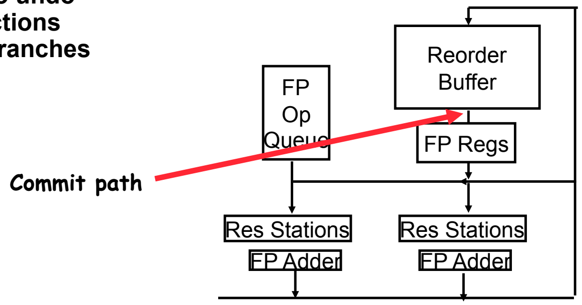
Reorder Buffer Entry

- 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

Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
  - Must have notion of time for in-order commit
- When instructions complete, results placed into ROB
  - Supplies operands ⇒ more registers like RS
  - Waiting operands tagged with ROB buffer number instead of RS
- Instructions commit ⇒ values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

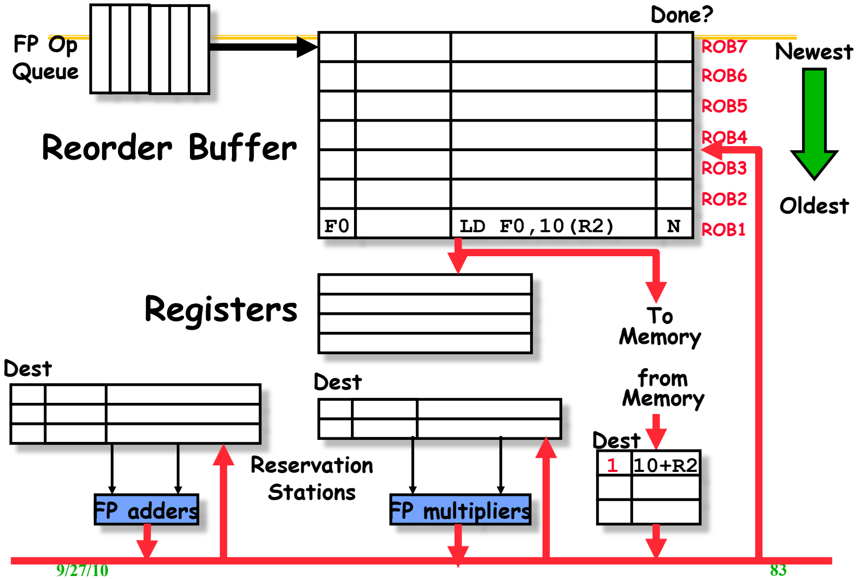


Recall: 4 Steps of Speculative Tomasulo Algorithm

New stuff is in blue

1. Issue—get instruction from FP Op 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")

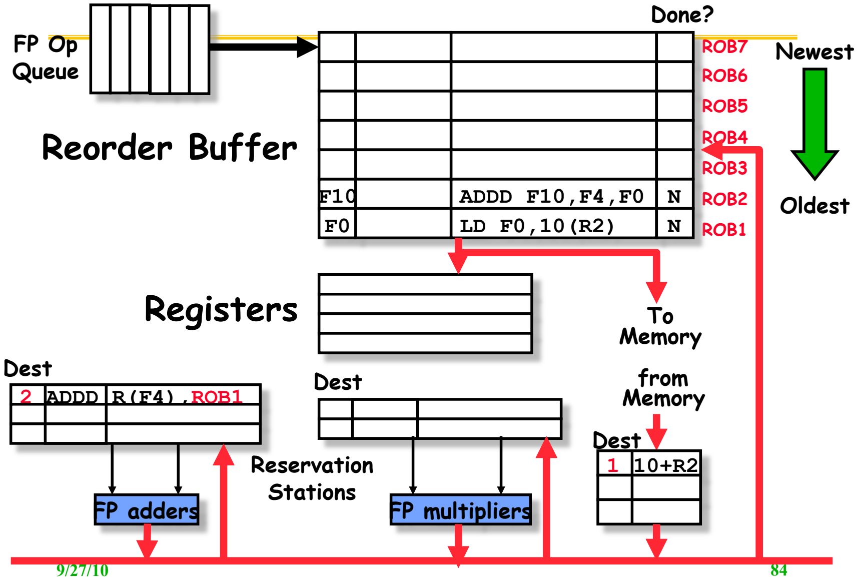
Tomasulo With Reorder buffer:



F0		LD F0,10(R2)	N

1	10+R2

Tomasulo With Reorder buffer:

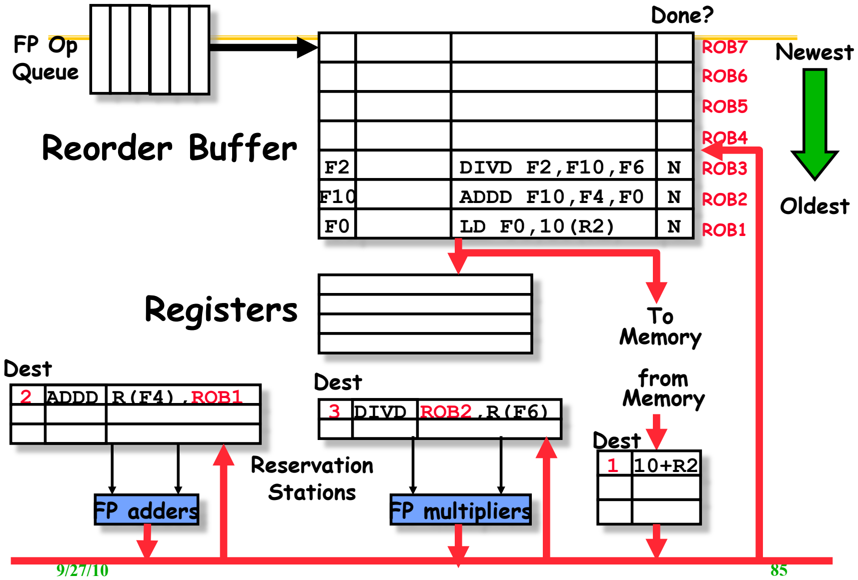


F10		ADDD F10,F4,F0	N
F0		LD F0,10(R2)	N

2	ADDD	R(F4),ROB1

1	10+R2

Tomasulo With Reorder buffer:



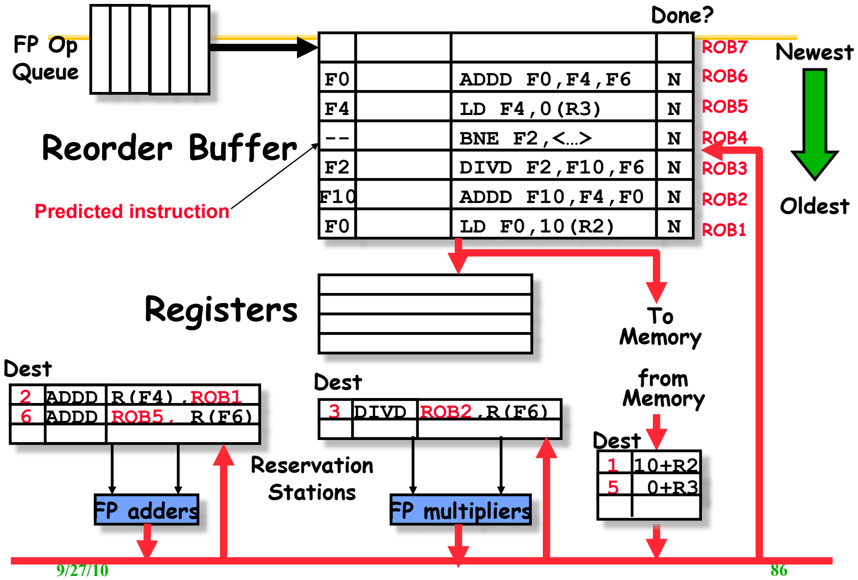
			Done?	
				ROB7
				ROB6
				ROB5
				ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2

Tomasulo With Reorder buffer:



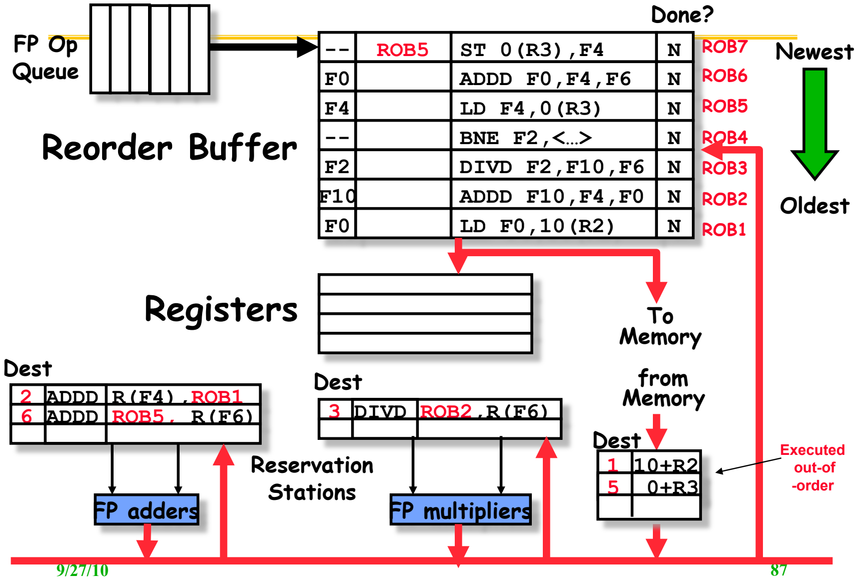
			Done?	
				ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1
6	ADDD	ROB5, R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2
5	0+R3

Tomasulo With Reorder buffer:



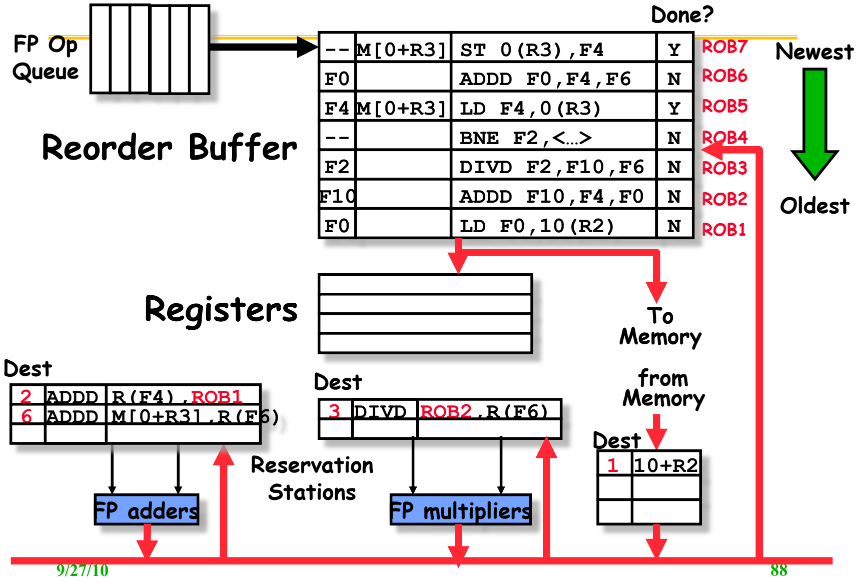
			Done?	
--	ROB5	ST 0(R3),F4	N	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4		LD F4,0(R3)	N	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1
6	ADDD	ROB5, R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2
5	0+R3

Tomasulo With Reorder buffer:



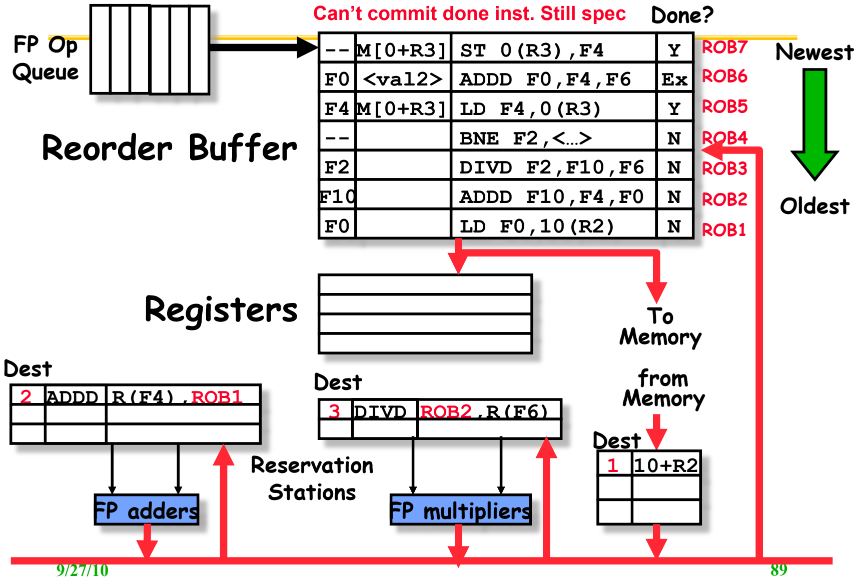
			Done?	
--	M[0+R3]	ST 0(R3),F4	Y	ROB7
F0		ADDD F0,F4,F6	N	ROB6
F4	M[0+R3]	LD F4,0(R3)	Y	ROB5
--		BNE F2,<…>	N	ROB4
F2		DIVD F2,F10,F6	N	ROB3
F10		ADDD F10,F4,F0	N	ROB2
F0		LD F0,10(R2)	N	ROB1

Dest		
2	ADDD	R(F4),ROB1
6	ADDD	M[0+R3],R(F6)

Dest		
3	DIVD	ROB2,R(F6)

Dest	
1	10+R2

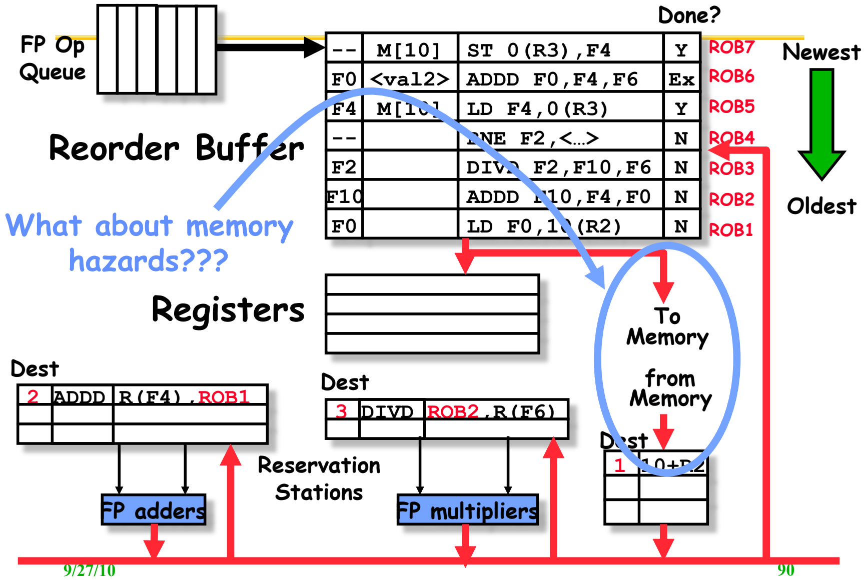
Tomasulo With Reorder buffer:



Can't commit done inst. Still spec

			Done?	
--	M[0+R3]	ST 0(R3),F4	Y	ROB7
F0	<val2>	ADDD F0,F4,F6	Ex	ROB6
F4	M[0+R3]	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

Tomasulo With 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

What about memory hazards???

Avoiding Memory Hazards

- How does hardware handle out-of-order memory accesses?
  - 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
  - Problem only if we commit out-of-order so we commit sequentially
- 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

Outline

- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Exceptions and Interrupts

- IBM 360/91 invented "imprecise interrupts"
  - Just a guess
  - Computer stopped at this PC; its likely close to this address
  - Due to out-of-order commit
  - Not so popular with programmers - hard to find bugs
  - Bad for page faults, which instruction caused it
- 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
  - Branch speculation is the same as precise exceptions
- Only recognize exception when ROB is ready to commit
  - If a speculated instruction raises an exception, the exception is recorded in the ROB
  - This is why reorder buffers in all new processors

Outline

- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Getting CPI below 1

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- How do we get CPI <= 1?
  - Multiple-issue processors come in 3 flavors:
    1. Compiler - statically-scheduled superscalar processors
       - use in-order execution if they are statically scheduled
    2. Runtime - dynamically-scheduled superscalar processors
       - out-of-order execution if they are dynamically scheduled
    3. Compiler - VLIW (very long instruction word) processors
       - 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)

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
  - Fixed size instruction like in RISC
    - » The long instruction word has room for many operations
  - All operations in each instruction 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
  - Assume compiler can figure out the parallelism and assume that it is correct - no hardware checks

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

Loop Unrolling in VLIW

Mem Ref 1	Mem Ref 2	FP Op 2	FP Op 2	Int op/ branch	clk
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 - more than before

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)

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
  - Assume that “compiler knows best” - no hardware checking
  - a stall in any functional unit pipeline caused entire processor and all operations in the instruction 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

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
  - First attempt, next would be better….
- 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

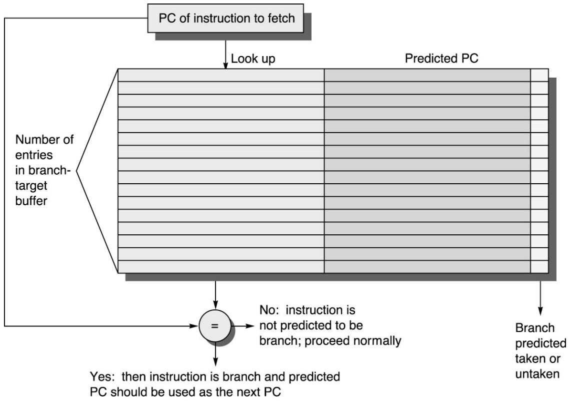
Outline

- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Increasing Instruction Fetch Bandwidth

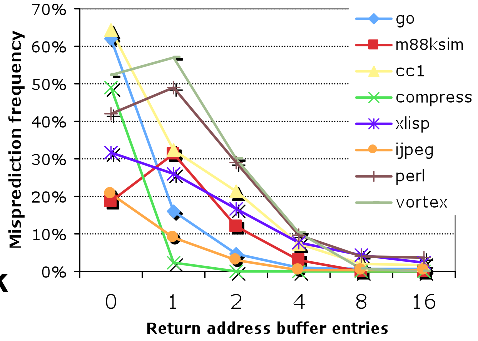
- Predicts next address, sends it out *before* decoding instruction
- PC of branch sent to BTB
- When match is found, Predicted PC is returned
- If branch predicted taken, instruction fetch continues at Predicted PC
- Allows fetching back-to-back instructions

Branch Target Buffer (BTB)


PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be branch; proceed normally
Yes: then instruction is branch and predicted PC should be used as the next PC
Branch predicted taken or untaken

IF BW: Return Address Predictor

- Small buffer of return addresses acts as a stack
- Caches most recent return addresses
- Call ⇒ Push a return address on stack
- Return ⇒ Pop an address off stack & predict as new PC



More Instruction Fetch Bandwidth

- **Integrated branch prediction**
  - branch predictor is part of instruction fetch unit and is constantly predicting branches
- **Instruction prefetch**
  - Instruction fetch units prefetch to deliver multiple instruct. per clock, integrating it with branch prediction
- **Instruction memory access and buffering**
  - Fetching multiple instructions per cycle:
    - » May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks)
    - » Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed

Outline

- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Speculation: Register Renaming vs. ROB

- Just have a larger physical set of registers combined with runtime register renaming
  – replace 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
- Allows binary compatibility

Outline

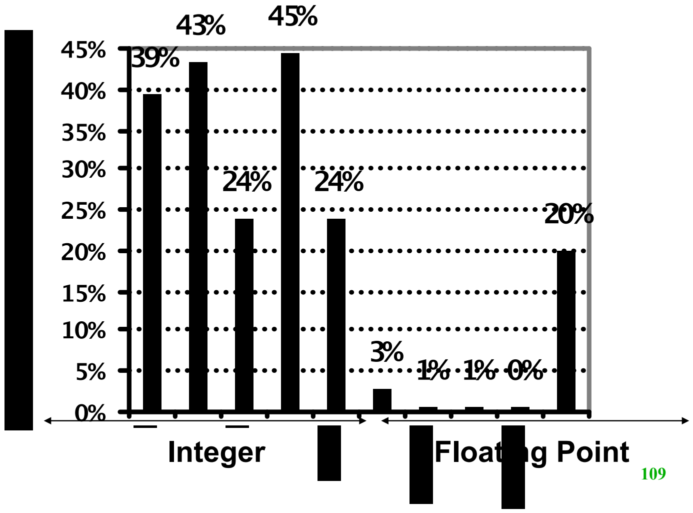
- Speculation
- Adding Speculation to Tomasulo
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

Value Prediction

- Value prediction
  – Attempts to predict value produced by instruction
    » E.g., Loads a value that changes infrequently
  – Value prediction is useful only if it significantly increases ILP
    » Hard to get good accuracy ≈ 50%
- Related topic is *address aliasing prediction*
  – Do two registers point to the same memory location
  – RAW for load and store or WAW for 2 stores
  – Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
  – Has been used by a few processors

(Mis) Speculation on Pentium 4

- % of micro-ops not used


Integer
Floating Point