

---

## EEC 581 Computer Architecture

### Lec 7 – Instruction Level Parallelism (2.6 Hardware-based Speculation and 2.7 Static Scheduling/VLIW)

Chansu Yu  
Electrical and Computer Engineering  
Cleveland State University

---

### Acknowledgement ...

- **Part of class notes are from**
  - David Patterson
  - Electrical Engineering and Computer Sciences
  - University of California, Berkeley
  - <http://www.eecs.berkeley.edu/~pattsrn>
  - <http://www-inst.eecs.berkeley.edu/~cs252>
- **And, from**
  - Mary Jane Irwin
  - Computer Science and Engineering
  - Pennsylvania State University
  - <http://www.cse.psu.edu/~mji>
  - [www.cse.psu.edu/~cg431](http://www.cse.psu.edu/~cg431)

## Overview of Chap. 2 & 3 (again)

---

- Pipelined architecture allows multiple instructions run in parallel (ILP)
  - But, it has data and control hazard problems
  - How can we avoid or alleviate the hazard problems in pipelined architecture?
  - Key idea is to “reorder” the execution of instructions !!!
- 2.3 Branch prediction (branch history table) } 2.6 Speculative execution (“commit”)  
2.4 & 2.5 Multiple issue – dependency Dynamic scheduling (forwarding) }  
2.7 Multiple issue – dependency Static scheduling (“VLIW”)

10/12/2007

3

## Review of Chap. 2 (so far...)

---

- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
  - Works when can't know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another
  - Reservations stations: *renaming* to larger set of registers + buffering source operands
    - » Prevents registers as bottleneck
    - » Avoids WAR, WAW hazards
    - » Allows loop unrolling in HW
  - 360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, ...

10/12/2007

4

## Outline

---

- ILP (2.1)
- Compiler techniques to increase ILP (2.1)
- Loop Unrolling (2.2)
- Static Branch Prediction (2.3)
- Dynamic Branch Prediction (2.3)
- Overcoming Data Hazards with Dynamic Scheduling (2.4)
- Tomasulo Algorithm (2.5)
- Speculation, Speculative Tomasulo, Memory Aliases, Exceptions, Register Renaming vs. Reorder Buffer (2.6)
- VLIW, Increasing instruction bandwidth (2.7)
- Instruction Delivery (2.9)

10/12/2007

5

## Speculation to greater ILP

---

- A wide issue processor may execute a branch every cycle => overcoming control dependences is a big burden
- Dynamic scheduling => only fetches and issues those dependent instructions
- With speculation => fetch, issue, and execute those dependent instructions as if branch predictions were always correct
  - Need a mechanism to handle the situation where the speculation is incorrect

*Additional step: "Commit"*

10/12/2007

6

## 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**

10/12/2007

7

## 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 : reorder buffer (ROB)**

10/12/2007

8

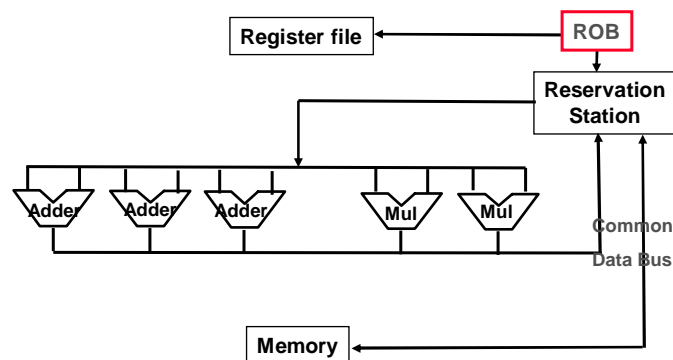
## Reorder Buffer (ROB)

- With speculation, the register file is not updated until the instruction commits
- 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 architected registers like RS

10/12/2007

9

## Tomasulo Organization with Speculation



10/12/2007

10

## Instruction Issue and Completion Policies

- **Instruction-issue – initiate execution**
  - Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- **Instruction-completion – complete execution**
  - Processor lookahead capability – complete issued instructions beyond the current instruction
- **Instruction-commit – write back results to the RegFile**

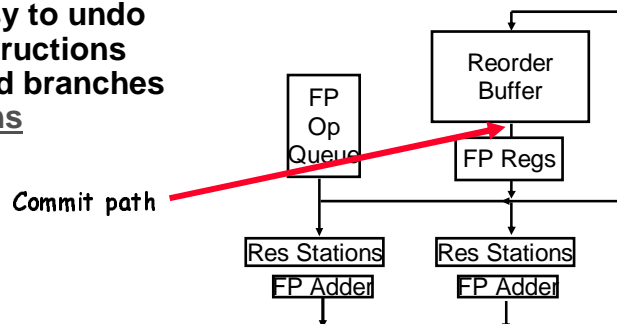
In-order issue with in-order completion  
In-order issue with out-of-order completion  
Out-of-order issue with out-of-order completion  
Out-of-order issue with out-of-order completion and in-order commit

10/12/2007

11

## Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
- Instructions commit  $\Rightarrow$  values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions



10/12/2007

12

## 4 Steps of Speculative Tomasulo Algorithm

### 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 ROB & result present, update register with result (or store to memory) and remove instr from ROB. Mispredicted branch flushes ROB (also called “graduation”)

10/12/2007

13

## 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

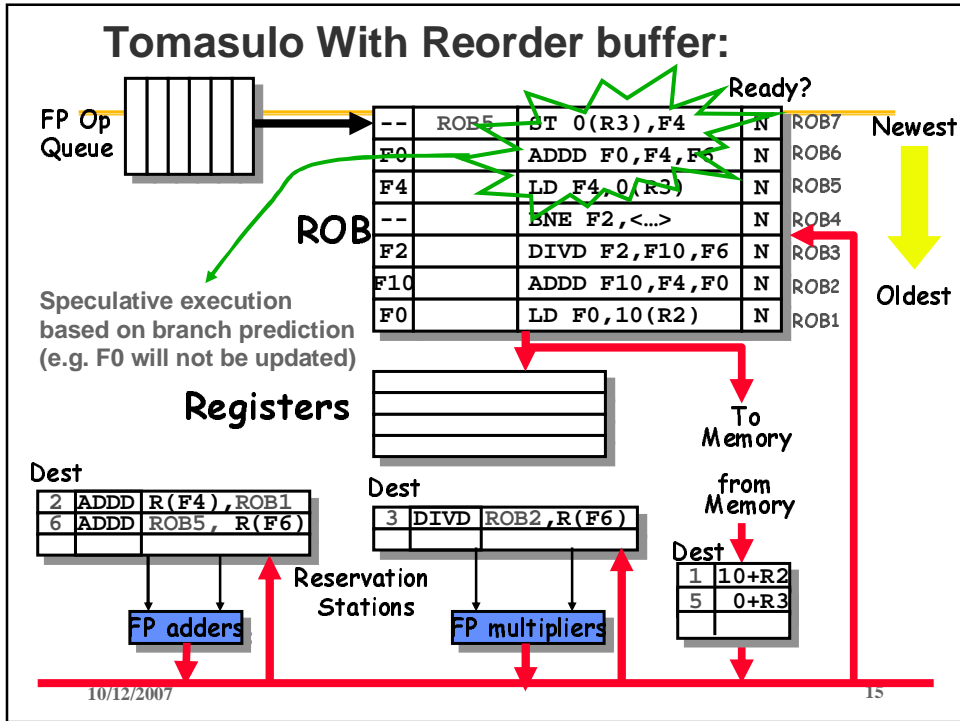
- Value of instruction result until the instruction commits

### 4. Ready

- Indicates that instruction has completed execution, and the value is ready

10/12/2007

14



- ## Exceptions and Interrupts
- IBM 360/91 invented “imprecise exceptions”
  - **Imprecise exceptions**
    - DIV F0, F2, F4
    - ADD F10, F10, F8
    - SUB F12, F12, F14
    - “ADD” and “SUB” can be executed earlier than “DIV”
    - What if “SUB” causes an exception (before “DIV” completes)?
  - **Imprecise exceptions**
    - The pipeline may have already completed instructions that are later than the offending instruction
    - The pipeline may have not yet completed instructions that are earlier than the offending instruction
- 10/12/2007 16

## Exceptions and Interrupts

---

- Speculation with in-order commit ensures “precise exceptions”
- Exceptions are handled by not recognizing the exception until the offending instruction is ready to commit in ROB
- If a speculated instruction raises an exception, the exception is recorded in the ROB
- If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
- This is why reorder buffers in all new processors

10/12/2007

17

## Outline

---

- ILP (2.1)
- Compiler techniques to increase ILP (2.1)
- Loop Unrolling (2.2)
- Static Branch Prediction (2.3)
- Dynamic Branch Prediction (2.3)
- Overcoming Data Hazards with Dynamic Scheduling (2.4)
- Tomasulo Algorithm (2.5)
- Speculation, Speculative Tomasulo, Memory Aliases, Exceptions, Register Renaming vs. Reorder Buffer (2.6)
- VLIW, Increasing instruction bandwidth (2.7)
- Instruction Delivery (2.9)

10/12/2007

18

## Extracting Yet *More* Performance: 2 Options

---

- **Increase the depth of the pipeline to increase the clock rate – superpipelining**
- **Fetch (and execute) more than one instructions at one time – multiple-issue**
  - **Dynamic multiple-issue processors (aka superscalar)**
    - » Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the hardware)
    - » E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500
  - **Static multiple-issue processors (aka VLIW)**
    - » Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
    - » E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)

10/12/2007

19

## History of VLIW Processors

---

- **Started with (horizontal) microprogramming**
  - Very wide microinstructions used to directly generate control signals in single-issue processors (e.g., IBM 360 series)
- **VLIW for multi-issue processors first appeared in the Multiflow and Cydrome (in the early 1980's)**
- **Current commercial VLIW processors**
  - Intel i860 RISC (dual mode: scalar and VLIW)
  - Intel I-64 (EPIC: Itanium and Itanium 2)
  - Transmeta Crusoe
  - Lucent/Motorola StarCore
  - ADI TigerSHARC
  - Infineon (Siemens) Carmel

10/12/2007

20

## VLIW (Very Long Instruction Word) Processors

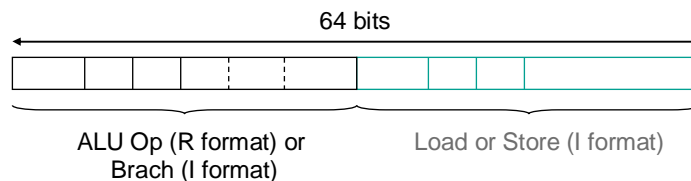
- VLIW processors use the compiler to decide which instructions to issue and execute simultaneously
  - Issue “bundle” (“packet” in IA-64, “molecule” in Transmeta) – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
  - The mix of instructions in the packet (bundle) is usually restricted
  - The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards
- VLIW’s have
  - Multiple functional units (like SS processors)
  - Multi-ported register files (again like SS processors)
  - Wide program bus

10/12/2007

21

## An Example: A VLIW MIPS

- Consider a 2-issue MIPS with a 2 instr bundle



**add \$1, \$2, \$3**

**lw \$4, 100(\$5)**

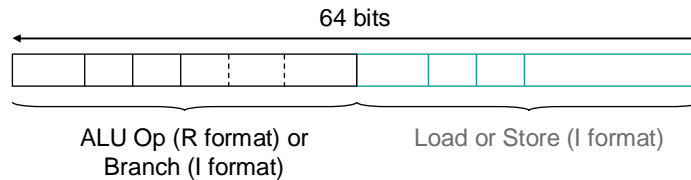
<b>IF</b>	--	--
<b>ID</b>	read \$2, \$3	read \$5
<b>EX</b>	calculate \$2+\$3	calculate 100+\$5
<b>MEM</b>	--	read M[100+\$5]
<b>WB</b>	write \$1	write \$4

10/12/2007

22

## An Example: A VLIW MIPS

- Consider a 2-issue MIPS with a 2 instr bundle

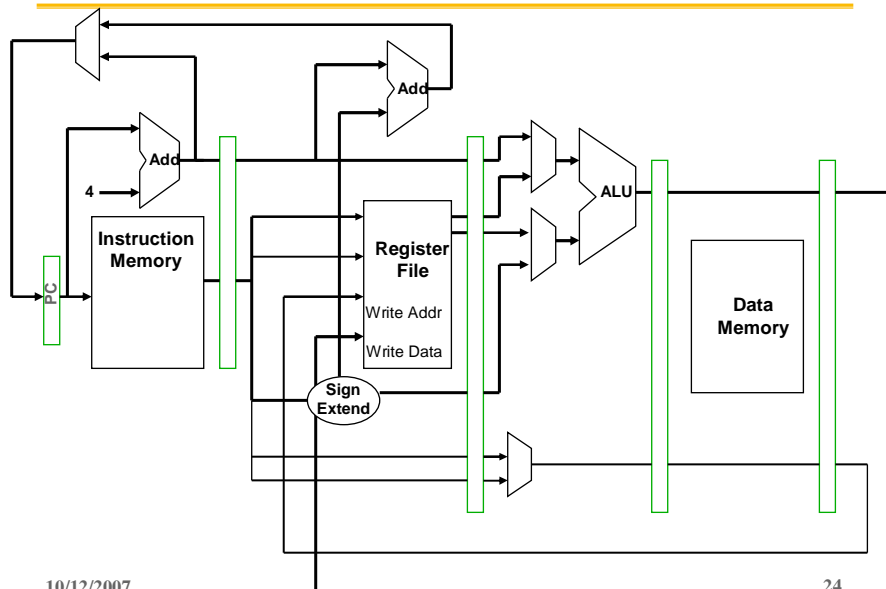


- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports (why?) and 2 write ports (why?) and a separate memory address adder
- Why isn't Load/Load or Load/Store allowed?
- Why isn't ALU/ALU allowed?

10/12/2007

23

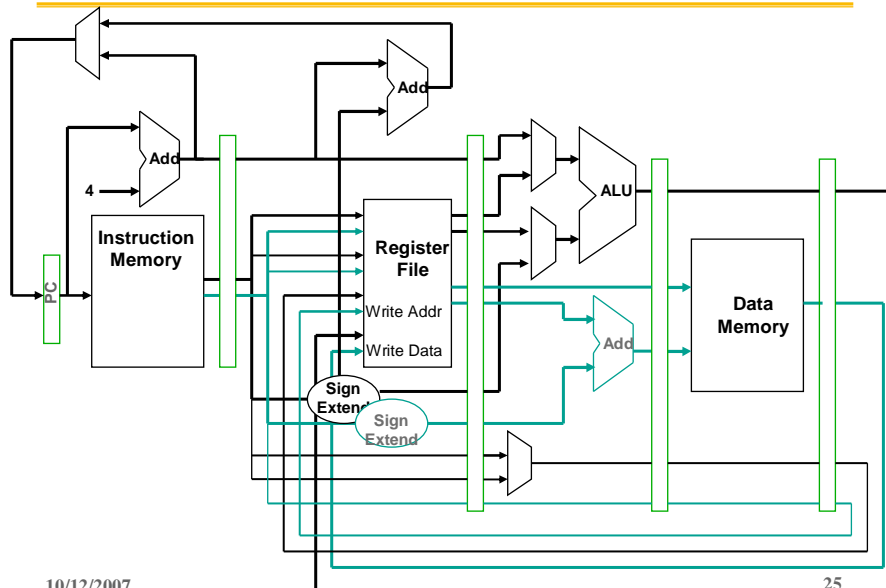
## MIPS (1-issue) Datapath



10/12/2007

24

## MIPS VLIW (2-issue) Datapath



## Code Scheduling Example

- Consider the following loop code

```
lp:  L.D  $F0,0($R1)  # $F0=array element
     ADD.D $F0,$F0,$F2 # add scalar in $F2
     S.D  $F0,0($R1) # store result
     DADDI $R1,$R1,-4 # decrement pointer
     BNEZ $R1,lp     # branch if $R1 != 0
```

- Must “schedule” the instructions to avoid pipeline stalls
  - Instructions in one bundle *must* be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load-use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

10/12/2007

26

## The Scheduled Code (Not Unrolled)

```
lp:  L.D  $F0,0($R1)  # $F0=array element
      ADD.D $F0,$F0,$F2 # add scalar in $F2
      S.D  $F0,0($R1) # store result
      DADDI $R1,$R1,-4 # decrement pointer
      BNEZ $R1,lp     # branch if $R1 != 0
```

	ALU or branch	Data transfer	CC
lp:			1
			2
			3
			4

10/12/2007

27

## The Scheduled Code (Not Unrolled)

```
lp:  L.D  $F0,0($R1)  # $F0=array element
      ADD.D $F0,$F0,$F2 # add scalar in $F2
      S.D  $F0,0($R1) # store result
      DADDI $R1,$R1,-4 # decrement pointer
      BNEZ $R1,lp     # branch if $R1 != 0
```

	ALU or branch	Data transfer	CC
lp:		L.D \$F0,0(\$R1)	1
			2
			3
			4

10/12/2007

28

## The Scheduled Code (Not Unrolled)

```
lp:  L.D  $F0,0($R1)  # $F0=array element
      ADD.D $F0,$F0,$F2 # add scalar in $F2
      S.D  $F0,0($R1) # store result
      DADDI $R1,$R1,-4 # decrement pointer
      BNEZ $R1,lp     # branch if $R1 != 0
```

	ALU or branch	Data transfer	CC
lp:		L.D \$F0,0(\$R1)	1
			2
	ADD.D \$F0,\$F0,\$F2		3
			4

10/12/2007

29

## The Scheduled Code (Not Unrolled)

```
lp:  L.D  $F0,0($R1)  # $F0=array element
      ADD.D $F0,$F0,$F2 # add scalar in $F2
      S.D  $F0,0($R1) # store result
      DADDI $R1,$R1,-4 # decrement pointer
      BNEZ $R1,lp     # branch if $R1 != 0
```

	ALU or branch	Data transfer	CC
lp:		L.D \$F0,0(\$R1)	1
			2
	ADD.D \$F0,\$F0,\$F2		3
		S.D \$F0,0(\$R1)	4

10/12/2007

30

## The Scheduled Code (Not Unrolled)

```
lp:  L.D  $F0,0($R1)  # $F0=array element
      ADD.D $F0,$F0,$F2 # add scalar in $F2
      S.D  $F0,0($R1)  # store result
      DADDI $R1,$R1,-4 # decrement pointer
      BNEZ $R1,lp     # branch if $R1 != 0
```

	ALU or branch	Data transfer	CC
lp:		L.D \$F0,0(\$R1)	1
	DADDI \$R1,\$R1,-4		2
	ADD.D \$F0,\$F0,\$F2		3
		S.D \$F0,4(\$R1)	4

10/12/2007

31

## The Scheduled Code (Not Unrolled)

```
lp:  L.D  $F0,0($R1)  # $F0=array element
      ADD.D $F0,$F0,$F2 # add scalar in $F2
      S.D  $F0,0($R1)  # store result
      DADDI $R1,$R1,-4 # decrement pointer
      BNEZ $R1,lp     # branch if $R1 != 0
```

	ALU or branch	Data transfer	CC
lp:		L.D \$F0,0(\$R1)	1
	DADDI \$R1,\$R1,-4		2
	ADD.D \$F0,\$F0,\$F2		3
	BNEZ \$R1, lp	S.D \$F0,4(\$R1)	4

10/12/2007

32

## The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:		L.D \$F0,0(\$R1)	1
	DADDI \$R1,\$R1,-4		2
	ADD.D \$F0,\$F0,\$F2		3
	BNEZ \$R1,lp	S.D \$F0,4(\$R1)	4

- **Four clock cycles to execute 5 instructions for a**
  - CPI of 0.8 (versus the best case of 0.5)
  - IPC of 1.25 (versus the best case of 2.0)

10/12/2007

33

## Loop Unrolling

- **Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP**
- **Apply loop unrolling (4 times for our example) and then *schedule* the resulting code**
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards
- **During unrolling the compiler applies *register renaming* to eliminate all data dependencies that are not true dependencies**

10/12/2007

34

## Unrolled Code Example

```

lp:  L.D  $F0,0($R1)  # $F0=array element
     L.D  $F1,-4($R1) # $F1=array element
     L.D  $F2,-8($R1) # $F2=array element
     L.D  $F3,-12($R1) # $F3=array element
     ADD.D $F0,$F0,$F2 # add scalar in $F2
     ADD.D $F1,$F1,$F2 # add scalar in $F2
     ADD.D $F3,$F3,$F2 # add scalar in $F2
     ADD.D $F4,$F4,$F2 # add scalar in $F2
     S.D  $F0,0($R1)  # store result
     S.D  $F1,-4($R1) # store result
     S.D  $F3,-8($R1) # store result
     S.D  $F4,-12($R1) # store result
     DADDI $R1,$R1,-16 # decrement pointer
     BNEZ $R1,lp      # branch if $R1 != 0
  
```

10/12/2007

35

## The Scheduled Code (Unrolled)

	ALU or branch	Data transfer	CC
lp:	DADDI \$R1,\$R1,-16	L.D \$F0,0(\$R1)	1
		L.D \$F1,12(\$R1)	2
	ADD.D \$F0,\$F0,\$F2	L.D \$F3,8(\$R1)	3
	ADD.D \$F1,\$F1,\$F2	L.D \$F4,4(\$R1)	4
	ADD.D \$F3,\$F3,\$F2	S.D \$F0,16(\$R1)	5
	ADD.D \$F4,\$F4,\$F2	S.D \$F1,12(\$R1)	6
		S.D \$F3,8(\$R1)	7
	BNEZ \$R1, lp	S.D \$F4,4(\$R1)	8

- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0)

10/12/2007

36

## The Scheduled Code (Unrolled)

Consider a 5-issue MIPS with 2 memory, 2 FP and one integer

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

CPI of 0.39, IPC of 2.55

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

10/12/2007

37

## Speculation & Predication

- **Speculation is used to allow execution of future instructions that (may) depend on the speculated instruction**
  - Speculate based on branch prediction
  - Results are stored in temporary store (reorder buffer)
  - “Commit” if the speculation is correct
- **Predication can be used to eliminate branches by making the execution of an instruction dependent on a “predicate”, e.g.,**

```
if (p) {statement 1} else {statement 2}
```

would normally compile using two branches.

With predication it would compile as

```
(p) statement 1
(~p) statement 2
```
- **The use of (condition) indicates that the instruction is committed only if condition is true** <sup>38</sup>

## Compiler Support for VLIW Processors

---

- The compiler packs groups of independent instructions into the bundle
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-then-else branch structures by replacing them with predicated instructions
- The compiler predicts memory bank references to help minimize memory bank conflicts

10/12/2007

39

## VLIW Advantages & Disadvantages

---

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
- Disadvantages
  - 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 predict 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

10/12/2007

40

## 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**
- **Hardware checks dependencies**
- **Predicated execution (select 1 out of 64 1-bit flags)**
  
- **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™ was 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

10/12/2007

41

## Outline

---

- **ILP (2.1)**
- **Compiler techniques to increase ILP (2.1)**
- **Loop Unrolling (2.2)**
- **Static Branch Prediction (2.3)**
- **Dynamic Branch Prediction (2.3)**
- **Overcoming Data Hazards with Dynamic Scheduling (2.4)**
- **Tomasulo Algorithm (2.5)**
- **Speculation, Speculative Tomasulo, Memory Aliases, Exceptions, Register Renaming vs. Reorder Buffer (2.6)**
- **VLIW, Increasing instruction bandwidth (2.7)**
- **Instruction Delivery (2.9)**

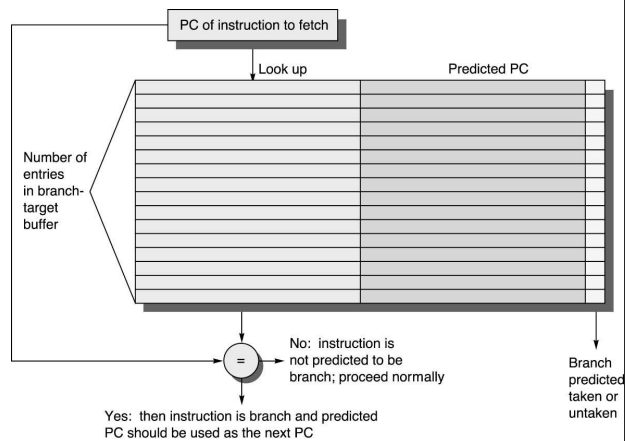
10/12/2007

42

## Increasing Instruction Fetch Bandwidth

- Predicts next instruct 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

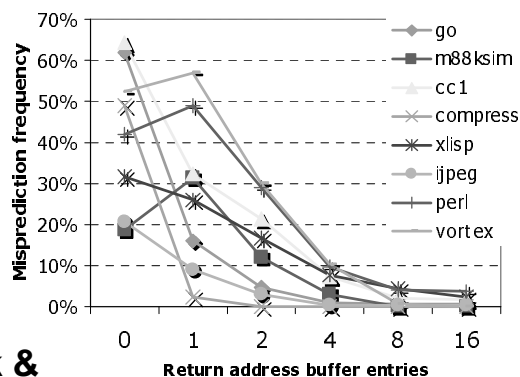
### Branch Target Buffer (BTB)



10/12/2007

## Return Address Predictor (Indirect Branch)

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



10/12/2007

44

## More Instruction Fetch Bandwidth

---

- **Integrated branch predictor is part of instruction fetch unit and is constantly predicting branches**
- **Instruction prefetch units prefetch to deliver multiple instructions 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**

10/12/2007

45

## 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**

10/12/2007

46

## Value Prediction

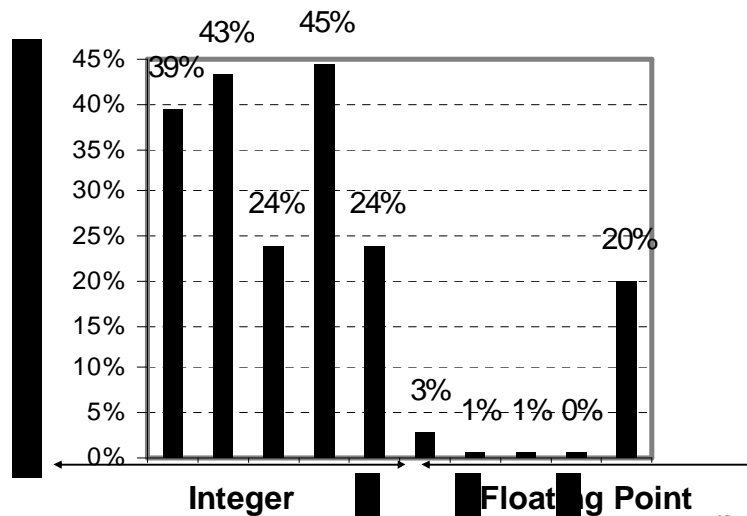
- Attempts to predict value produced by instruction
  - E.g., Loads a value that changes infrequently
- Value prediction is useful only if it significantly increases ILP
  - Focus of research has been on loads; so-so results, no processor uses value prediction
- Related topic is *address aliasing prediction*
  - 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

10/12/2007

47

## (Mis) Speculation on Pentium 4

- % of micro-ops not used



10/12/2007

48

## 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**
- **Conservative in ideas, just faster clock and bigger**
- **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**  
⇒ **performance 8 to 16X**
- **Peak v. delivered performance gap increasing**