

---

## Lecture 5 – Instruction Level Parallelism (cont'd)

Slides were used during lectures by David Patterson, Berkeley, spring 2006

---

### Review from Last Time (1)

- 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

---

### Review from Last Time (2)

- Reservations stations: *renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, ...

---

### Outline

- ILP
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction
- Limits to ILP

---

### 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

---

### 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

## 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

## 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 architected registers like RS

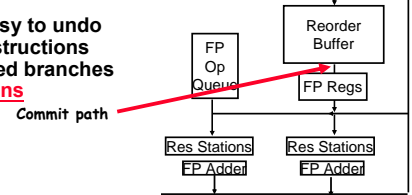
## Reorder Buffer Entry

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

## Reorder Buffer operation

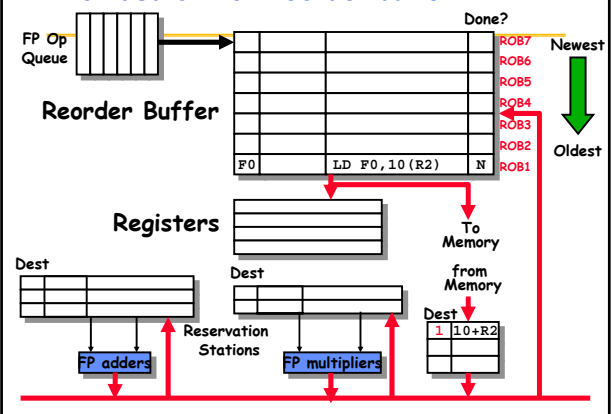
- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit  $\Rightarrow$  more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions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

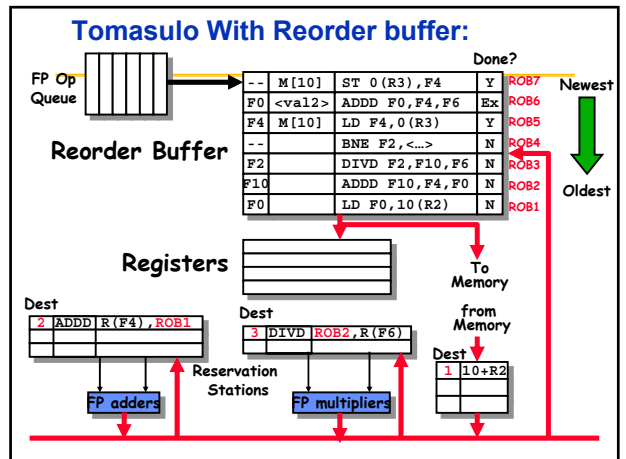
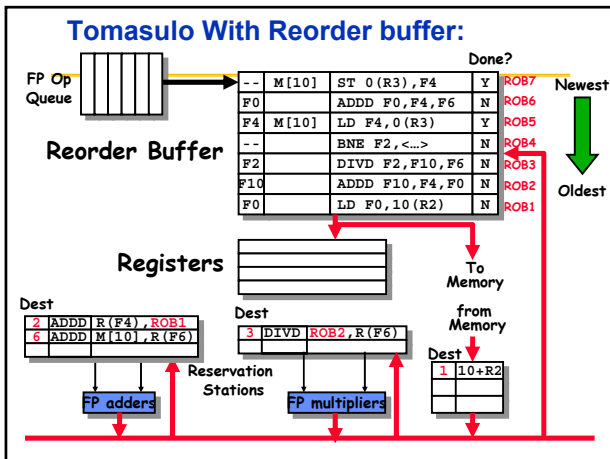
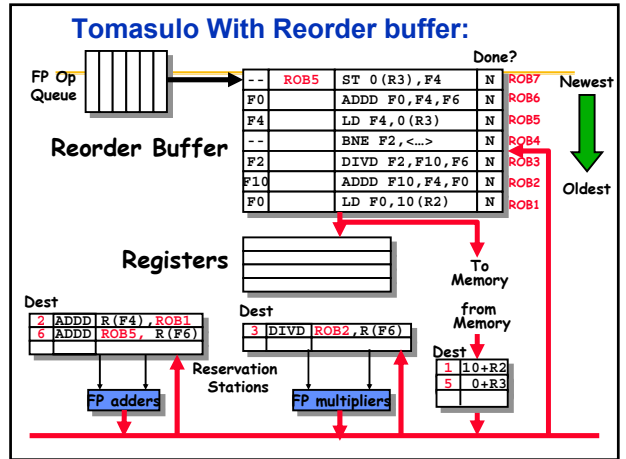
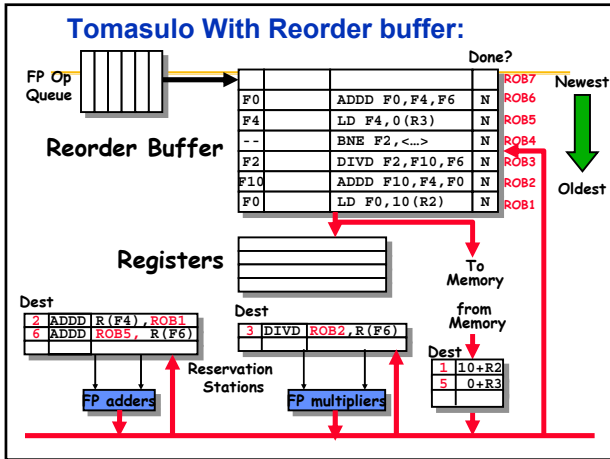
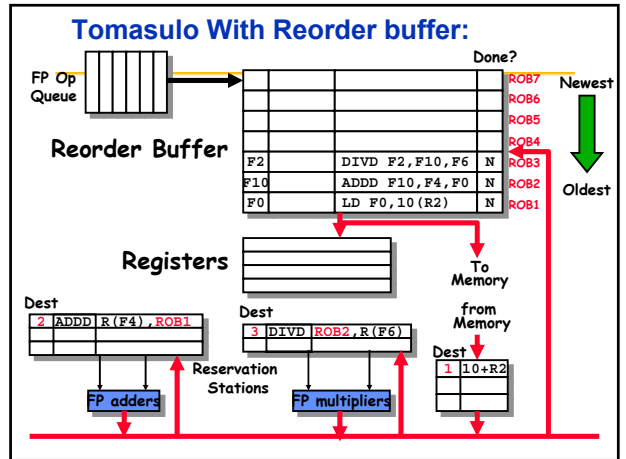
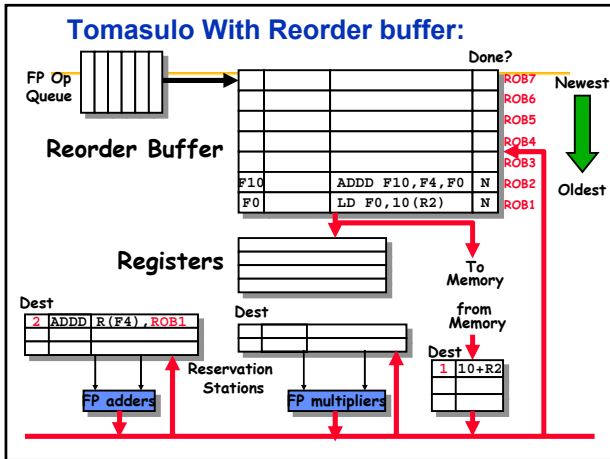


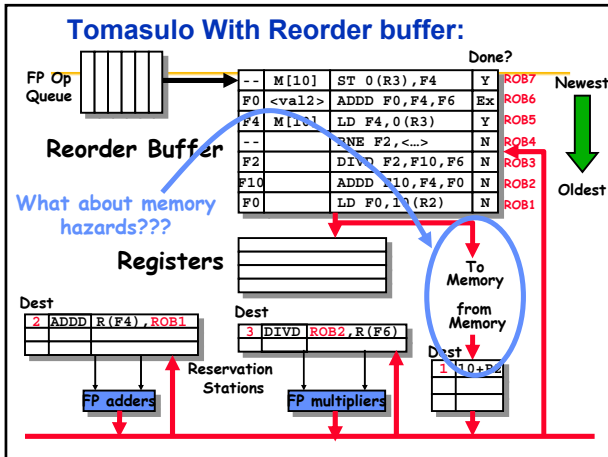
## Recall: 4 Steps of Speculative Tomasulo Algorithm

- 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 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:







## 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:
  - 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
  - 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

## 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

## Getting CPI below 1

- CPI  $\geq 1$  if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  - statically-scheduled superscalar processors,
  - dynamically-scheduled superscalar processors, and
  - 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)

## 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

## Recall: Unrolled Loop that Minimizes Stalls for Scalar

```

1 Loop: L.D   F0, 0(R1)           LD to ADD.D: 1 Cycle
2         L.D   F6, -8(R1)        ADD.D to S.D: 2 Cycles
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
  
```

14 clock cycles, or 3.5 per iteration

## 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)

## Problems with 1st Generation VLIW

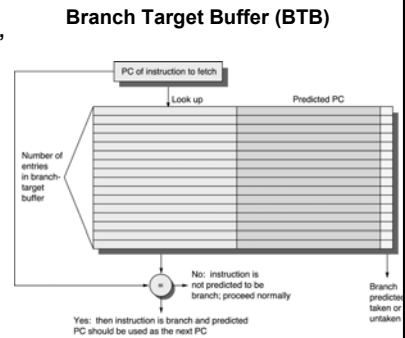
- **Increase in code size**
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- **Operated in lock-step; no hazard detection HW**
  - 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

## 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

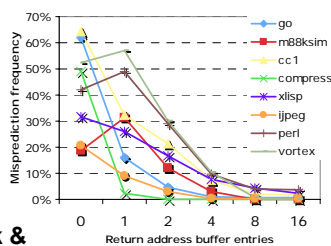
## Increasing Instruction Fetch Bandwidth

- 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



## 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

## Speculation: Register Renaming vs. ROB

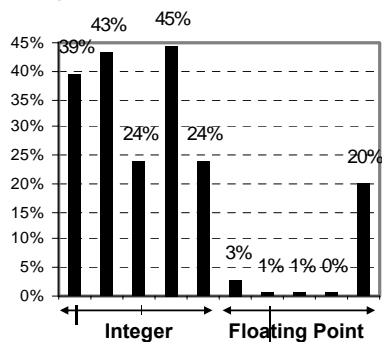
- Alternative to ROB is a larger physical set of registers combined with register renaming
  - Extended registers replace function of both ROB and reservation stations
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
  - On issue, allocates 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

## 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

## (Mis) Speculation on Pentium 4

% of micro-ops not used



## 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

## In Conclusion ...

- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
  - Possibly large quantities of state must be saved before interrupting
- Machines with **precise exceptions** provide one single point in the program to restart execution
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  - Important enabling factor for out-of-order execution

break

## Limits to ILP

- **Conflicting studies of amount**
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication
- **How much ILP is available using existing mechanisms with increasing HW budgets?**
- **Do we need to invent new HW/SW mechanisms to keep on processor performance curve?**
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola AltaVec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.

## Overcoming Limits

- **Advances in compiler technology + significantly new and different hardware techniques *may* be able to overcome limitations assumed in studies**
- **However, unlikely such advances when coupled *with realistic hardware* will overcome these limits in near future**

## Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. **Register renaming** – infinite virtual registers => all register WAW & WAR hazards are avoided
  2. **Branch prediction** – perfect; no mispredictions
  3. **Jump prediction** – all jumps perfectly predicted (returns, case statements)
- 2 & 3 => no control dependencies; perfect speculation & an unbounded buffer of instructions available
4. **Memory-address alias analysis** – addresses known & a load can be moved before a store provided addresses not equal; 1&4 eliminates all but RAW

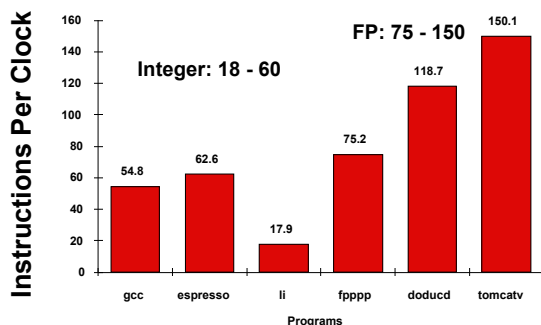
Also: perfect caches; 1 cycle latency for all instructions (FP \*,/); unlimited instructions issued/clock cycle;

## Limits to ILP HW Model comparison

	Model	Power 5
Instructions Issued per clock	Infinite	4
Instruction Window Size	Infinite	200
Renaming Registers	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias Analysis	Perfect	??

## Upper Limit to ILP: Ideal Machine

Figure 3.1

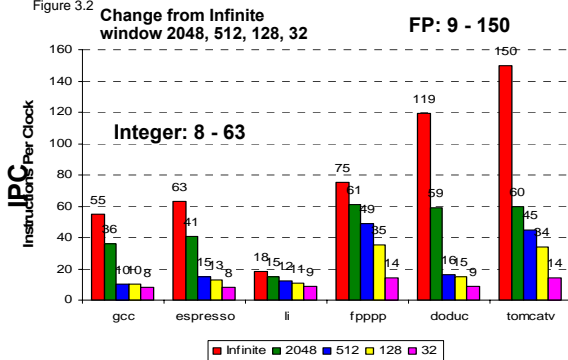


## Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	Infinite	Infinite	4
Instruction Window Size	Infinite, 2K, 512, 128, 32	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??

### More Realistic HW: Window Impact

Figure 3.2

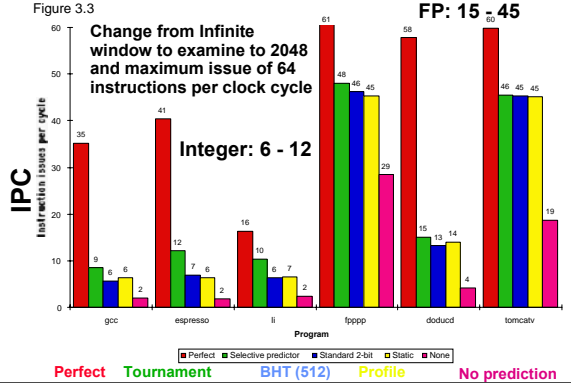


### Limits to ILP HW Model comparison

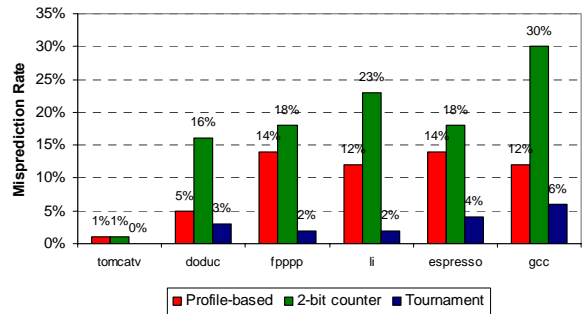
	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??

### More Realistic HW: Branch Impact

Figure 3.3



### Misprediction Rates

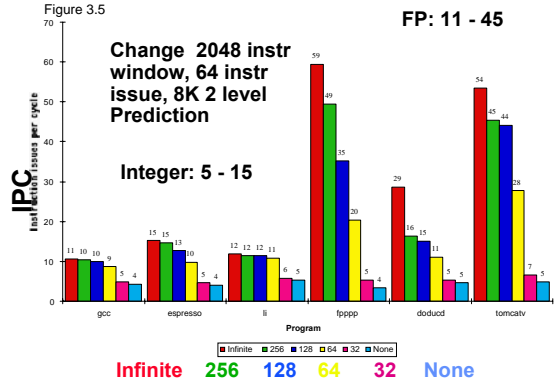


### Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite v. 256, 128, 64, 32, none	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament Branch Predictor
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	Perfect

### More Realistic HW: Renaming Register Impact (N int + N fp)

Figure 3.5



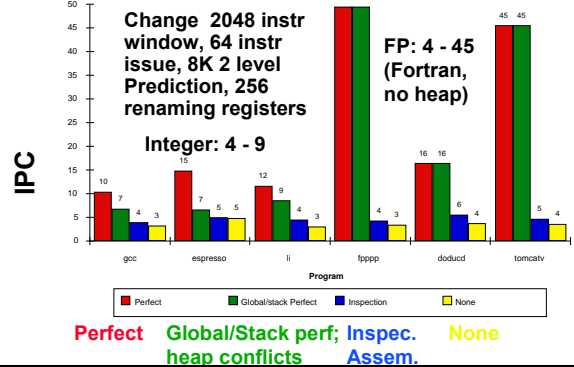


## Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	256 Int + 256 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect v. Stack v. Inspec. v. none	Perfect	Perfect

## More Realistic HW: Memory Address Alias Impact

Figure 3.6

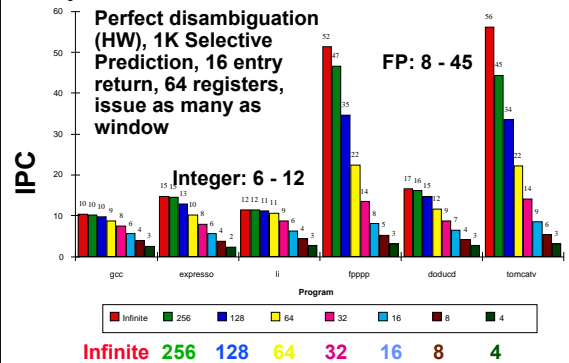


## Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64 (no restrictions)	Infinite	4
Instruction Window Size	Infinite vs. 256, 128, 64, 32	Infinite	200
Renaming Registers	64 Int + 64 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	1K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	HW disambiguation	Perfect	Perfect

## Realistic HW: Window Impact

Figure 3.7



## Limits to ILP (1)

- Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to
  - issue 3 or 4 data memory accesses per cycle,
  - resolve 2 or 3 branches per cycle,
  - rename and access more than 20 registers per cycle, and
  - fetch 12 to 24 instructions per cycle.
- The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate
  - E.g., widest issue processor is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power!

## Limits to ILP (2)

- Most techniques for increasing performance increase power consumption
- The key question is whether a technique is *energy efficient*: does it increase power consumption faster than it increases performance?
- Multiple issue processors techniques all are energy inefficient:
  1. Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
  2. Growing gap between peak issue rates and sustained performance
- Number of transistors switching =  $f(\text{peak issue rate})$ , and performance =  $f(\text{sustained rate})$ , growing gap between peak and sustained performance
  - ⇒ increasing energy per unit of performance

## Reading

---

- **This lecture:**
  - chapter 2 *ILP*
  - chapter 3: 3.1-3.4 *Limits to ILP*
- **Next lecture:**
  - chapter 3: 3.5-3.8 *Simultaneous Multithreading (SMT)*
- **No class on Wed Oct 31<sup>st</sup>**
- **Wed Nov 14<sup>th</sup> 11.15-13.00h & 13.45-15.30h, room 402**