CS252
Graduate Computer Architecture

Lecture 19:
Intro to Static Pipelining

April 6, 2001

Prof. David A. Patterson

Computer Science 252

Spring 2001

Review: Dynamic Examples

- P6 (Pentium Pro, II, III) successful micro-architecture, even with imitator (AMD Athlon)
  - Translate most 80x86 instructions to micro-operations
    - » Longer pipeline than RISC instructions
  - Dynamically execute micro-operations
- "Netburst" (Pentium 4, …) success not clear
  - Much longer pipeline, higher clock rate in same technology as P6
  - Trace Cache to capture micro-operations, avoid hardware translation
- Multithreading to increase performance for servers, parallel programs written to use threads
  - Extra copies of PCs, Registers per thread; e.g., IBM AS/400
- Simultaneous Multithreading (SMT) exploit underutilized Dynamic Execution HW to get higher throughput at low extra cost?

Overview

- Last 3 lectures: binary compatibility and exploiting ILP in hardware: BTB, ROB, Reservation Stations, ...
- How far can you go in compiler?
- What if you can also change instruction set architecture?
- Will see multi billion dollar gamble by two Bay Area firms for the future of computer architecture: HP and Intel to produce IA-64
  - 7 years in the making?

Static Branch Prediction

- Simplest: Predict taken
  - average misprediction rate = untaken branch frequency, which for the SPEC programs is 34%.
  - Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%)
- Predict on the basis of branch direction?
  - choosing backward-going branches to be taken (loop)
  - forward-going branches to be not taken (if)
  - SPEC programs, however, most forward-going branches are taken => predict taken is better
- Predict branches on the basis of profile information collected from earlier runs
  - Misprediction varies from 5% to 22%

Running Example

- This code, a scalar to a vector:

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

- Assume following latency all examples

Instruction producing result	*Instruction using result*	*Execution in cycles*	*Latency 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

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  - To simplify, assume 8 is lowest address

```
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
       DSUBUI  R1,R1,8  ;decrement pointer 8B (DW)
       BNEZ    R1,Loop  ;branch R1!=zero
       NOP              ;delayed branch slot
```

Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1) ;F0=vector element
2       stall
3       ADD.D  F4,F0,F2 ;add scalar in F2
4       stall
5       stall
6       S.D    0(R1),F4 ;store result
7       DSUBUI R1,R1,8  ;decrement pointer 8B (DW)
8       BNEZ   R1,Loop  ;branch R1!=zero
9       stall           ;delayed branch slot
```

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

- 9 clocks: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2       stall
3       ADD.D  F4,F0,F2
4       DSUBUI R1,R1,8
5       BNEZ   R1,Loop   ;delayed branch
6       S.D    8(R1),F4 ;altered when move past DSUBUI
```

Swap BNEZ and S.D by changing address of S.D

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

6 clocks, but just 3 for execution, 3 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

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

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop Detail

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

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

- What assumptions made when moved code?
  - OK to move store past DSUBUI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

Compiler Perspectives on Code Movement

- Compiler concerned about dependencies in program
- Whether or not a HW hazard depends on pipeline
- Try to schedule to avoid hazards that cause performance losses
- (True) Data dependencies (RAW if a hazard for HW)
  - Instruction i produces a result used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.
- If dependent, can't execute in parallel
- Easy to determine for registers (fixed names)
- Hard for memory ("memory disambiguation" problem):
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?

Where are the name dependencies?

```
1 Loop:L.D    F0,0(R1)
2      ADD.D  F4,F0,F2
3      S.D    0(R1),F4      ;drop DSUBUI & BNEZ
4      L.D    F0,-8(R1)
5      ADD.D  F4,F0,F2
6      S.D    -8(R1),F4     ;drop DSUBUI & BNEZ
7      L.D    F0,-16(R1)
8      ADD.D  F4,F0,F2
9      S.D    -16(R1),F4    ;drop DSUBUI & BNEZ
10     L.D    F0,-24(R1)
11     ADD.D  F4,F0,F2
12     S.D    -24(R1),F4
13     DSUBUI R1,R1,#32     ;alter to 4*8
14     BNEZ   R1,LOOP
15     NOP
```

How can remove them?

Where are the name dependencies?

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

The Orginal"register renaming"

Compiler Perspectives on Code Movement

- Name Dependencies are Hard to discover for Memory Accesses
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- Our example required compiler to know that if R1 doesn't change then:

  `0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)`

  There were no dependencies between some loads and stores so they could be moved by each other

Steps Compiler Performed to Unroll

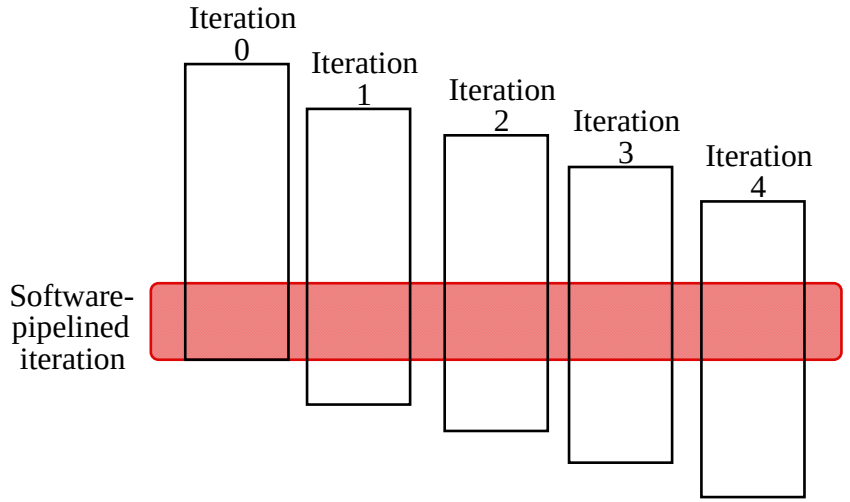
- Check OK to move the S.D after DSUBUI and BNEZ, and find amount to adjust S.D offset
- Determine unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

Administratrivia

- 3rd (last) Homework on Ch 3 due Saturday
- 3rd project meetings 4/11: signup today
- Project Summary due Monday night
- Quiz #2 4/18 310 Soda at 5:30

Another possibility: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)



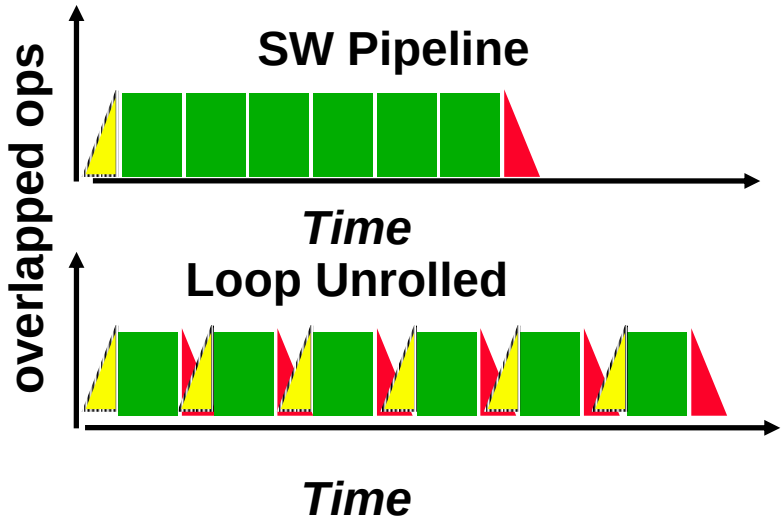
Software Pipelining Example

Before: Unrolled 3 times

```
1   L.D    F0,0(R1)
2   ADD.D  F4,F0,F2
3   S.D    0(R1),F4
4   L.D    F6,-8(R1)
5   ADD.D  F8,F6,F2
6   S.D    -8(R1),F8
7   L.D    F10,-16(R1)
8   ADD.D  F12,F10,F2
9   S.D    -16(R1),F12
10  DSUBUI    R1,R1,#24
11  BNEZ   R1,LOOP
```

After: Software Pipelined

```
1   S.D    0(R1),F4 ; Stores M[i]
2   ADD.D  F4,F0,F2 ; Adds to M[i-1]
3   L.D    F0,-16(R1);Loads M[i-2]
4   DSUBUI R1,R1,#8
5   BNEZ   R1,LOOP
```



- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop
    vs. once per each unrolled iteration in loop unrolling

5 cycles per iteration

When Safe to Unroll Loop?

- Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];    /* S1 */
      B[i+1] = B[i] + A[i+1];  /* S2 */
}
```

1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
This is a "loop-carried dependence": between iterations

- For our prior example, each iteration was distinct
- Implies that iterations can't be executed in parallel, Right????

Does a loop-carried dependence mean there is no parallelism???

- Consider:

```
for (i=0; i< 8; i=i+1) {
      A = A + C[i];    /* S1 */
}
```

Could compute:

```
"Cycle 1":  temp0 = C[0] + C[1];
            temp1 = C[2] + C[3];
            temp2 = C[4] + C[5];
            temp3 = C[6] + C[7];
"Cycle 2":  temp4 = temp0 + temp1;
            temp5 = temp2 + temp3;
"Cycle 3":  A = temp4 + temp5;
```

- Relies on associative nature of "+".
- See "Parallelizing Complex Scans and Reductions" by Allan Fisher and Anwar Ghuloum (handed out next week)

Hardware Support for Exposing More Parallelism at Compile-Time

- Conditional or Predicated Instructions
  - Discussed before in context of branch prediction
  - Conditional instruction execution
- First instruction slot Second instruction slot

```
LW    R1,40(R2)          ADD R3,R4,R5
                         ADD R6,R3,R7
BEQZ  R10,L
LW    R8,0(R10)
LW    R9,0(R8)
```

- Waste slot since 3rd LW dependent on result of 2nd LW

Hardware Support for Exposing More Parallelism at Compile-Time

- Use predicated version load word (LWC)?
  - load occurs unless the third operand is 0
- First instruction slot Second instruction slot

```
LW    R1,40(R2)            ADD R3,R4,R5
LWC     R8,20(R10),R10     ADD R6,R3,R7
BEQZ  R10,L
LW    R9,0(R8)
```

- If the sequence following the branch were short, the entire block of code might be converted to predicated execution, and the branch eliminated

Exception Behavior Support

- Several mechanisms to ensure that speculation by compiler does not violate exception behavior
  - For example, cannot raise exceptions in predicated code if annulled
  - Prefetch does not cause exceptions

Hardware Support for Memory Reference Speculation

- To compiler to move loads across stores, when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture
  - The special instruction is left at the original location of the load and the load is moved up across stores
  - When a speculated load is executed, the hardware saves the address of the accessed memory location
  - If a subsequent store changes the location before the check instruction, then the speculation has failed
  - If only load instruction was speculated, then it suffices to redo the load at the point of the check instruction

What if Can Chance Instruction Set?

- Superscalar processors decide on the fly how many instructions to issue
  - HW complexity of Number of instructions to issue $O(n^2)$
- Why not allow compiler to schedule instruction level parallelism explicitly?
- Format the instructions in a potential issue packet so that HW need not check explicitly for dependences

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)
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

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

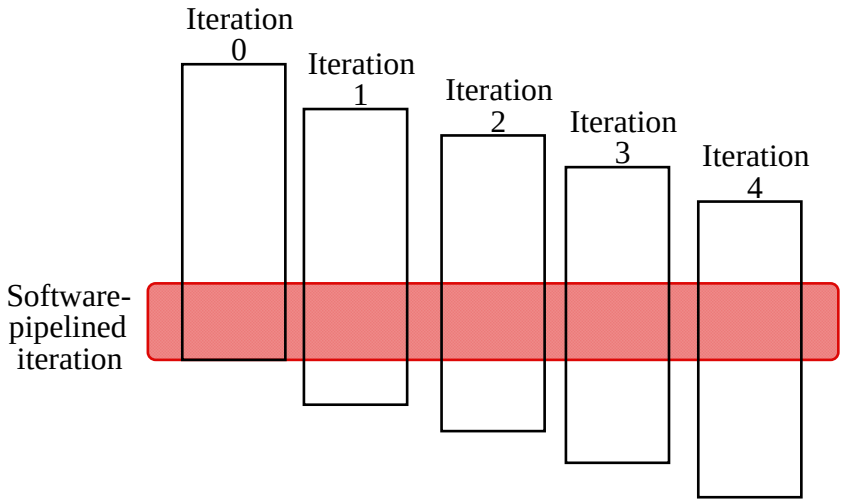
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

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

Recall: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)



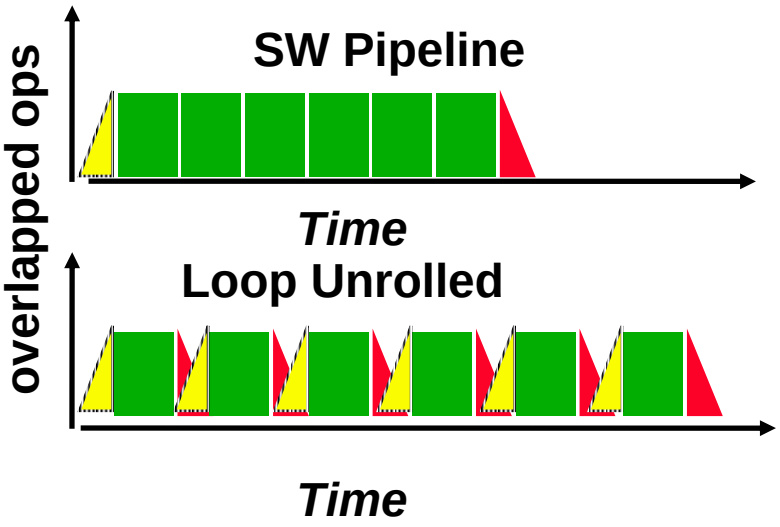
Recall: Software Pipelining Example

Before: Unrolled 3 times

```
1  L.D    F0,0(R1)
2  ADD.D  F4,F0,F2
3  S.D    0(R1),F4
4  L.D    F6,-8(R1)
5  ADD.D  F8,F6,F2
6  S.D    -8(R1),F8
7  L.D    F10,-16(R1)
8  ADD.D  F12,F10,F2
9  S.D    -16(R1),F12
10 DSUBUI R1,R1,#24
11 BNEZ   R1,LOOP
```

After: Software Pipelined

```
1  S.D    0(R1),F4 ; Stores M[i]
2  ADD.D  F4,F0,F2 ; Adds to M[i-1]
3  L.D    F0,-16(R1); Loads M[i-2]
4  DSUBUI R1,R1,#8
5  BNEZ   R1,LOOP
```



- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling

Software Pipelining with Loop Unrolling in VLIW

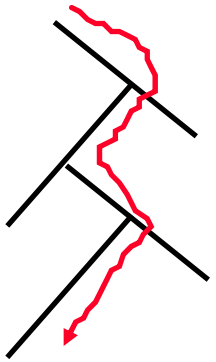
Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F4,F0,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F8,F6,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST 8(R1),F12	ADD.D F12,F10,F2		BNEZ R1,LOOP	3

- Software pipelined across 9 iterations of original loop
  - In each iteration of above loop, we:
    - » Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
    - » Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
    - » Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)
- 9 results in 9 cycles, or 1 clock per iteration
- Average: 3.3 ops per clock, 66% efficiency

Note: Need fewer registers for software pipelining (only using 7 registers here, was using 15)

Trace Scheduling

- Parallelism across IF branches vs. LOOP branches?
- Two steps:
  - *Trace Selection*
    - » Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - *Trace Compaction*
    - » Squeeze trace into few VLIW instructions
    - » Need bookkeeping code in case prediction is wrong



- This is a form of compiler-generated speculation
  - Compiler must generate "fixup" code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding
- Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks

Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

- HW advantages:
  - HW better at memory disambiguation since knows actual addresses
  - HW better at branch prediction since lower overhead
  - HW maintains precise exception model
  - HW does not execute bookkeeping instructions
  - Same software works across multiple implementations
  - Smaller code size (not as many nops filling blank instructions)
- SW advantages:
  - Window of instructions that is examined for parallelism much higher
  - Much less hardware involved in VLIW (unless you are Intel…!)
  - More involved types of speculation can be done more easily
  - Speculation can be based on large-scale program behavior, not just local information

Superscalar v. VLIW

- Smaller code size
- Binary compatibility across generations of hardware

- Simplified Hardware for decoding, issuing instructions
- No Interlock Hardware (compiler checks?)
- More registers, but simplified Hardware for Register Ports (multiple independent register files?)

Problems with First 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; EPIC is type
  - EPIC = 2nd generation VLIW?
- Itanium™ is name of first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process
- 128 64-bit integer registers + 128 82-bit floating point registers
  - 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?

IA-64 Registers

- The integer registers are configured to help accelerate procedure calls using a register stack
  - mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture.
  - Registers 0-31 are always accessible and addressed as 0-31
  - Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96)
  - The new register stack frame is created for a called procedure by renaming the registers in hardware;
  - a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure
- 8 64-bit Branch registers used to hold branch destination addresses for indirect branches
- 64 1-bit predict registers

IA-64 Registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation is designed to ease the task of allocating of registers in software pipelined loops
- When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop
  - makes the SW-pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages

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

- Instruction group: a sequence of consecutive instructions with no register data dependences
  - All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved
  - An instruction group can be arbitrarily long, but the compiler must explicitly indicate the boundary between one instruction group and another by placing a stop between 2 instructions that belong to different groups
- IA-64 instructions are encoded in bundles, which are 128 bits wide.
  - Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length
- 3 Instructions in 128 bit "groups"; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr

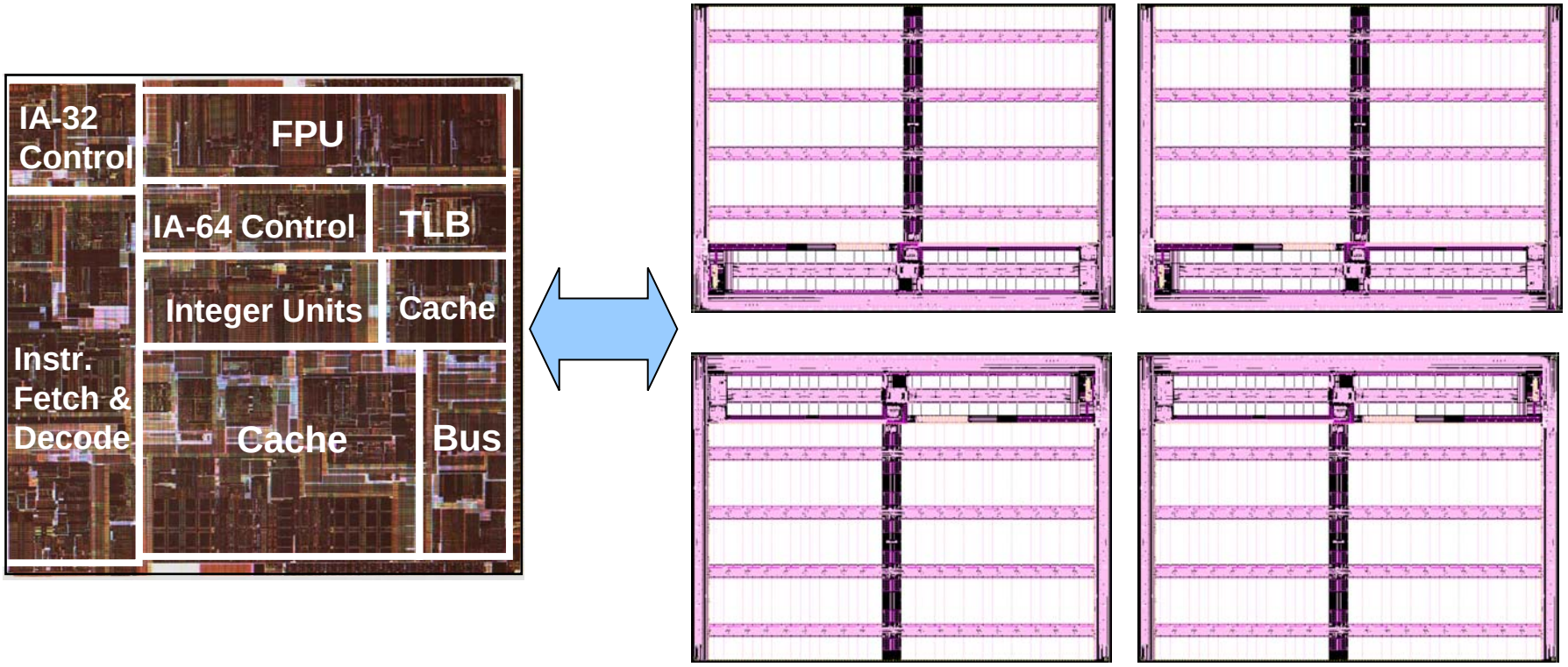
5 Types of Execution in Bundle

Execution Unit Slot	*Instruction type*	*Instruction Description*	*Example Instructions*
I-unit	A	Integer ALU	add, subtract, and, or, cmp
	I	Non-ALU Int	shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, cmp
	M	Memory access	Loads, stores for int/FP regs
F-unit	F	Floating point	Floating point instructions
B-unit	B	Branches	Conditional branches, calls
L+X	L+X	Extended	Extended immediates, stops

- 5-bit template field within each bundle describes both the presence of any stops associated with the bundle *and* the execution unit type required by each instruction within the bundle (see Fig 4.12 page 271)

Itanium™ Processor Silicon

(Copyright: Intel at Hotchips '00)



Core Processor Die

4 x 1MB L3 cache

Itanium™ Machine Characteristics

(Copyright: Intel at Hotchips '00)

Frequency	800 MHz
Transistor Count	25.4M CPU; 295M L3
Process	0.18u CMOS, 6 metal layer
Package	Organic Land Grid Array
Machine Width	6 insts/clock (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)
Registers	14 ported 128 GR & 128 FR; 64 Predicates
Speculation	32 entry ALAT, Exception Deferral
Branch Prediction	Multilevel 4-stage Prediction Hierarchy
FP Compute Bandwidth	3.2 GFlops (DP/EP); 6.4 GFlops (SP)
Memory -> FP Bandwidth	4 DP (8 SP) operands/clock
Virtual Memory Support	64 entry ITLB, 32/96 2-level DTLB, VHPT
L2/L1 Cache	Dual ported 96K Unified & 16KD; 16KI
L2/L1 Latency	6 / 2 clocks
L3 Cache	4MB, 4-way s.a., BW of 12.8 GB/sec;
System Bus	2.1 GB/sec; 4-way Glueless MP Scalable to large (512+ proc) systems

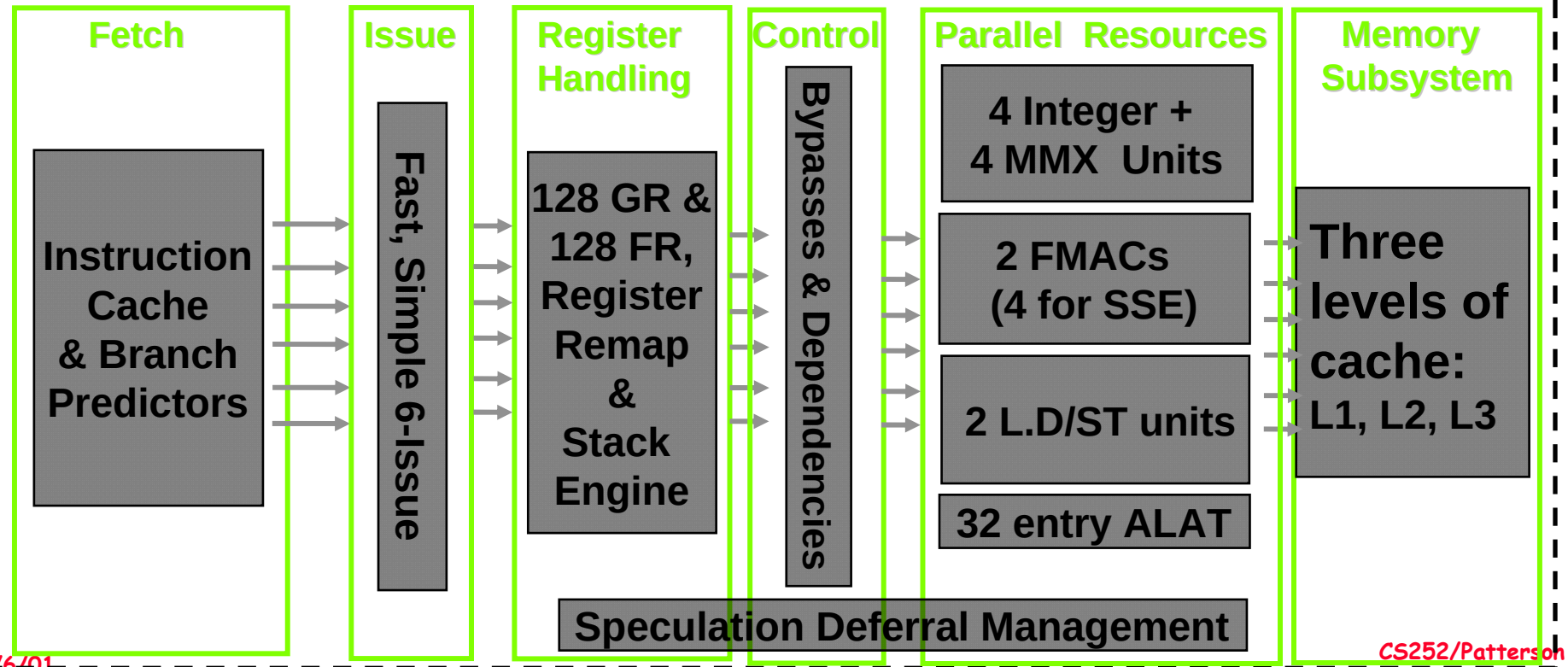
Itanium™ EPIC Design Maximizes SW-HW Synergy

(Copyright: Intel at Hotchips '00)

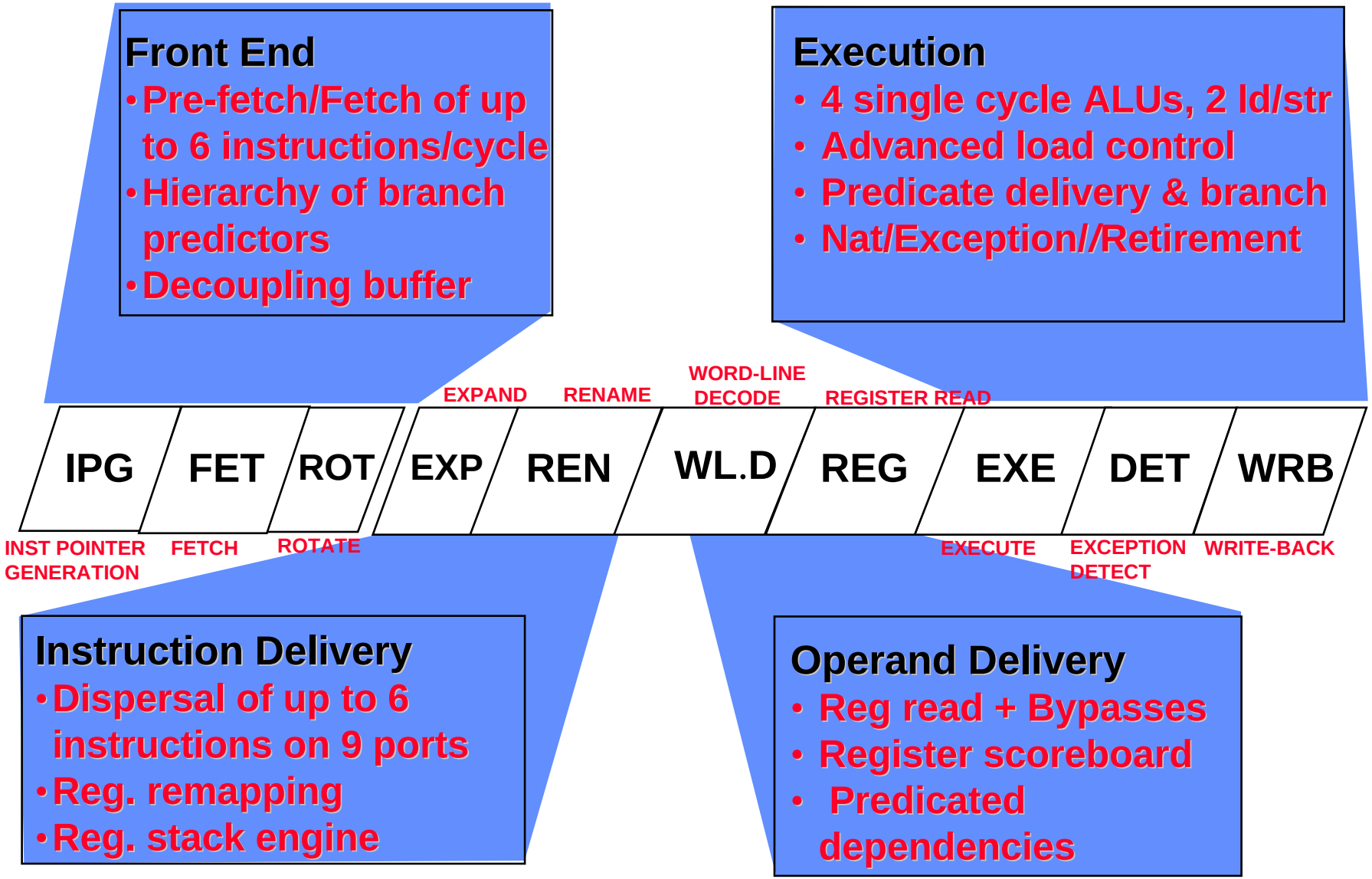
Architecture Features programmed by compiler:

Branch Hints | ***Explicit Parallelism*** | ***Register Stack & Rotation*** | ***Predication*** | ***Data & Control Speculation*** | ***Memory Hints***

Micro-architecture Features in hardware:



10 Stage In-Order Core Pipeline
(Copyright: Intel at Hotchips '00)



Itanium processor 10-stage pipeline

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to 8 bundles (24 instructions)
  - Branch prediction is done using a multilevel adaptive predictor like P6 microarchitecture
- Instruction delivery (stages EXP and REN): distributes up to 6 instructions to the 9 functional units
  - Implements registers renaming for both rotation and register stacking.

Itanium processor 10-stage pipeline

- Operand delivery (WLD and REG): accesses register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences.
  - Scoreboard used to detect when individual instructions can proceed, so that a stall of 1 instruction in a bundle need not cause the entire bundle to stall
- Execution (EXE, DET, and WRB): executes instructions through ALUs and load/store units, detects exceptions and posts NaTs, retires instructions and performs write-back
  - Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits, called NaTs for Not a Thing, for the GPRs (which makes the GPRs effectively 65 bits wide), and NaT Val (Not a Thing Value) for FPRs (already 82 bits wides)

Comments on Itanium

- Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines
  - strong emphasis on branch prediction, register renaming, scoreboarding, a deep pipeline with many stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection
- Surprising that an approach whose goal is to rely on compiler technology and simpler HW seems to be at least as complex as dynamically scheduled processors!

Peformance of IA-64 Itanium?

- Despite the existence of silicon, no significant standard benchmark results are available for the Itanium
- Whether this approach will result in significantly higher performance than other recent processors is unclear
- The clock rate of Itanium (733 MHz) is competitive but slower than the clock rates of several dynamically-scheduled machines, which are already available, including the Pentium III, Pentium 4 and AMD Athlon

Summary#1: Hardware versus Software Speculation Mechanisms

- To speculate extensively, must be able to disambiguate memory references
  - Much easier in HW than in SW for code with pointers
- HW-based speculation works better when control flow is unpredictable, and when HW-based branch prediction is superior to SW-based branch prediction done at compile time
  - Mispredictions mean wasted speculation
- HW-based speculation maintains precise exception model even for speculated instructions
- HW-based speculation does not require compensation or bookkeeping code

Summary#2: Hardware versus Software Speculation Mechanisms cont'd

- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling
- HW-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture
  - may be the most important in the long run?

Summary #3: Software Scheduling

- Instruction Level Parallelism (ILP) found either by compiler or hardware.
- Loop level parallelism is easiest to see
  - SW dependencies/compiler sophistication determine if compiler can unroll loops
  - Memory dependencies hardest to determine => Memory disambiguation
  - Very sophisticated transformations available
- Trace Sceduling to Parallelize If statements
- Superscalar and VLIW: CPI < 1 (IPC > 1)
  - Dynamic issue vs. Static issue
  - More instructions issue at same time => larger hazard penalty
  - Limitation is often number of instructions that you can successfully fetch and decode per cycle