CS152 – Computer Architecture and Engineering
Lecture 15 – Advanced Pipelining

2003-10-16

Dave Patterson
(www.cs.berkeley.edu/~patterson)

www-inst.eecs.berkeley.edu/~cs152/

Road to Faster Processors

- Time = Instruction Count x CPI x Clock cycle time
- How get a shorter Clock Cycle Time?
- Can we get CPI < 1?
- Can we reduce pipeline stalls for cache misses, hazards, … ?

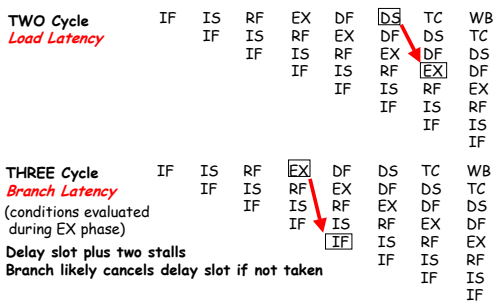
Fast Clock Cycle Time

- For a given technology, shorter clock cycle time
  => less work clock cycle
  => longer pipeline to accomplish task
- Deep pipelines ("superpipelined") to get high clock rate, low clock cycle times
- 5 pipeline stages MIPS 2000…
  => 8 pipeline stages MIPS 4000
  => 10 pipeline stages Pentium Pro
  => 20 pipeline stages Pentium 4

Case Study: MIPS R4000

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages:
  What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000



Recall: Compute CPI?

- Start with Base CPI
- Add stalls

$$CPI = CPI_{base} + CPI_{stall}$$

$$CPI_{stall} = STALL_{type-1} \times freq_{type-1} + STALL_{type-2} \times freq_{type-2}$$

- Suppose:
  - CPI_{base}=1
  - $Freq_{branch}$=20%, $freq_{load}$=30%
  - Suppose branches always cause 1 cycle stall
  - Loads cause a 100 cycle stall 1% of time
- Then: CPI = 1 + (1×0.20)+(100 × 0.30×0.01)=1.5
- Multicycle? Could treat as:
  CPI_{stall}=(CYCLES-CPI_{base}) × $freq_{inst}$

Case Study: MIPS R4000 (200 MHz)

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages:
  What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000

TWO Cycle *Load Latency*	IF	IS	RF	EX	DF	DS	TC	WB
		IF	IS	RF	EX	DF	DS	TC
			IF	IS	RF	EX	DF	DS
				IF	IS	RF	EX	DF
					IF	IS	RF	EX
						IF	IS	RF
							IF	IS
								IF
THREE Cycle *Branch Latency* (conditions evaluated during EX phase)	IF	IS	RF	EX	DF	DS	TC	WB
		IF	IS	RF	EX	DF	DS	TC
			IF	IS	RF	EX	DF	DS
				IF	IS	RF	EX	DF
					IF	IS	RF	EX
						IF	IS	RF
							IF	IS
								IF

Delay slot plus two stalls
Branch likely cancels delay slot if not taken

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP Pipe Stages

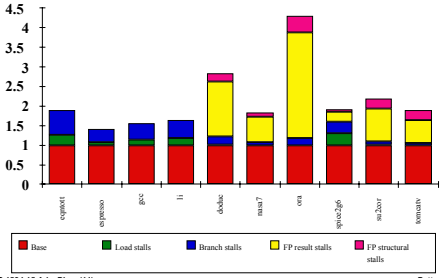
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	…
Add, Subtract	U	S+A	A+R	R+S					
Multiply	U	E+M	M	M	M	N	N+A	R	
Divide	U	A	R	D^{28}	…	D+A	D+R,	D+R,	D+A, D+R, A, R
Square root	U	E	$(A+R)^{108}$	…	A	R			
Negate	U	S							
Absolute value	U	S							
FP compare	U	A	R						

Stages:

M	*First stage of multiplier*	***A***	***Mantissa ADD stage***
N	*Second stage of multiplier*	***D***	***Divide pipeline stage***
R	*Rounding stage*	***E***	***Exception test stage***
S	*Operand shift stage*		
U	*Unpack FP numbers*		

R4000 Performance

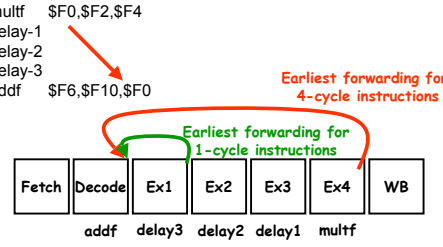
- Not ideal CPI of 1:
  - FP structural stalls: Not enough FP hardware (parallelism)
  - FP result stalls: RAW data hazard (latency)
  - Branch stalls (2 cycles + unfilled slots)
  - Load stalls (1 or 2 clock cycles)



Can we somehow make CPI closer to 1?

- Let's assume full pipelining:
  - If we have a 4-cycle instruction, then we need 3 instructions between a producing instruction and its use:

```
multf   $F0,$F2,$F4
delay-1
delay-2
delay-3
addf    $F6,$F10,$F0
```



FP Loop: Where are the Hazards?

```
Loop: LD   F0,0(R1) ;F0=vector element
      ADDD F4,F0,F2 ;add scalar from F2
      SD   0(R1),F4 ;store result
      SUBI R1,R1,8  ;decrement pointer 8B (DW)
      BNEZ R1,Loop  ;branch R1!=zero
      NOP           ;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
Load double	Store double	0
Integer op	Integer op	0

- Where are the stalls?

FP Loop Showing Stalls

```
1 Loop: LD   F0,0(R1)   ;F0=vector element
2       stall
3       ADDD F4,F0,F2   ;add scalar in F2
4       stall
5       stall
6       SD   0(R1),F4   ;store result
7       SUBI 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 (10 if SUBI/BNEZ is a stall): Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: LD   F0,0(R1)
2       stall
3       ADDD F4,F0,F2
4       SUBI R1,R1,8
5       BNEZ R1,Loop    ;delayed branch
6       SD   8(R1),F4   ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

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: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

```
1  Loop: LD   F0,0(R1)
2        ADDD F4,F0,F2
3        SD   0(R1),F4    ;drop SUBI & BNEZ
4        LD   F6,-8(R1)
5        ADDD F8,F6,F2
6        SD   -8(R1),F8   ;drop SUBI & BNEZ
7        LD   F10,-16(R1)
8        ADDD F12,F10,F2
9        SD   -16(R1),F12 ;drop SUBI & BNEZ
10       LD   F14,-24(R1)
11       ADDD F16,F14,F2
12       SD   -24(R1),F16
13       SUBI 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)
CPI = 27/15 = 1.8

Unrolled Loop That Minimizes Stalls

```
1  Loop: LD   F0,0(R1)
2        LD   F6,-8(R1)
3        LD   F10,-16(R1)
4        LD   F14,-24(R1)
5        ADDD F4,F0,F2
6        ADDD F8,F6,F2
7        ADDD F12,F10,F2
8        ADDD F16,F14,F2
9        SD   0(R1),F4
10       SD   -8(R1),F8
11       SD   -16(R1),F12
12       SUBI R1,R1,#32
13       BNEZ R1,LOOP
14       SD   8(R1),F16   ; 8-32 = -24
```

- What assumptions made when moved code?
  - OK to move store past SUBI even though SUBI changes register value
  - 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
CPI = 14/14 = 1

When safe to move instructions?

Administrivia

- Lab 5/6 Design Doc Due Sunday by 9 PM
  - Meet tomorrow with TA to go over plan Monday
- Mon 10/20: HW 3 due
- Design full cache, but only demo reads on Friday 10/24; demo writes on Friday 10/31
- Thurs 11/6: Design Doc for Final Project due
  - Deep pipeline? Superscalar? Out-of-order?
- Friday 11/14: Demo Project modules
- Monday 12/1: Demo Project to T.A.s
- Tuesday 12/2: 30 min oral presentation
- Wednesday 12/3: Processor racing

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two main variations: Superscalar and VLIW
- Superscalar: varying no. instructions/cycle (1 to 6)
  - Parallelism and dependencies determined/resolved by HW
  - Intel Pentium IV, IBM PowerPC G5, Sun UltraSparc,…
  - Very Long Instruction Words (VLIW): fixed number of instructions (16) parallelism determined by compiler
  - Pipeline is exposed; compiler must schedule delays to get right result
- Explicit Parallel Instruction Computer (EPIC)/ Intel Titanium
  - 128 bit packets containing 3 instructions (can execute sequentially)
  - Can link 128 bit packets together to allow more parallelism
  - Compiler determines parallelism, HW checks dependencies and fowards/stalls

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Simple Superscalar MIPS: 2 instructions, 1 FP & 1 anything
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration

Superscalar evolution

- 2 instructions ("2-scalar"): 1 FP + 1 everything
  - MIPS: 64-bit aligned in memory/cache
- 2 instructions: 1 anything + 1 anything but load/store (only 1 load/store per pair)
  - No alignment restrictions
- 3 - 4 instructions ("3 or 4-scalar"): 1 load/store + 3 anything else
- 3 - 6 instructions from a window of read to execute instructions: up to 2 load/store + rest anything else

Problems?

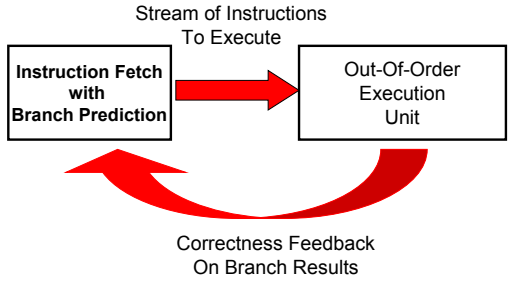
- How do we prevent WAR and WAW hazards?
- How do we deal with variable latency?
  - Forwarding for RAW hazards harder.

Instruction	Clock Cycle Number															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F6,34(R2)	IF	ID	EX	MEM	WB												
LD F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD F6,F8,F2						IF	ID	A1	A2	MEM	WB						

RAW

WAR

What about FETCH? Independent "Fetch" unit



- Instruction fetch decoupled from execution
- Often issue logic (+ rename) included with Fetch

Branches must be resolved quickly for loop overlap!

- In loop-unrolling example, we assumed branches were under control of "fast" integer unit in order to get overlap!

```
Loop:   LD       F0    0     R1
        MULTDF4  F0    F2
        SD       F4    0     R1
        SUBI     R1    R1    #8
        BNEZ     R1    Loop
```

- What happens if branch depends on result of multd??
  - We completely lose all of our advantages!
  - Need to be able to "predict" branch outcome.
  - If we were to predict that branch was taken, this would be right most of the time.
- Problem much worse for superscalar machines!

Prediction: Branches, Dependencies, Data

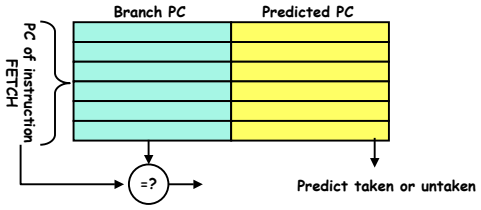
- Prediction has become essential to getting good performance from scalar instruction streams.
- We will discuss predicting branches. However, architects are now predicting everything:
  data dependencies, actual data, and *results of groups of instructions:*
  - At what point does computation become a probabilistic operation + verification?
  - We are pretty close with control hazards already…
- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.
- Prediction ⇒ Compressible information streams?

Dynamic Branch Prediction

- Prediction could be "Static" (at compile time) or "Dynamic" (at runtime)
  - For our example, if we were to statically predict "taken", we would only be wrong once each pass through loop
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be. Still some debate to this effect
  - Today, lots of hardware being devoted to dynamic branch predictors.
- Does branch prediction make sense for 5-stage, in-order pipeline? What about 8-stage pipeline?
  - Perhaps: eliminate branch delay slots/then predict branches

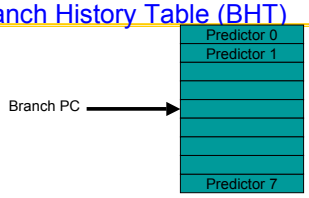
Simple dynamic prediction: Branch Target Buffer (BTB)

- Address of branch index to get prediction AND branch address (if taken)
  - Must check for branch match now, since can't use wrong branch address
  - Grab predicted PC from table since may take several cycles to compute



- Update predicted PC when branch is actually resolved
- Return instruction addresses predicted with stack

Branch History Table (BHT)



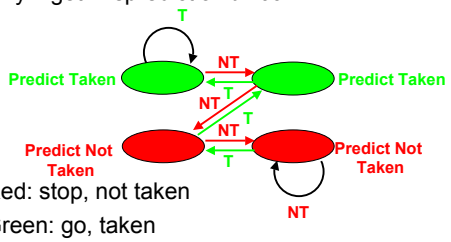
- BHT is a table of "Predictors"
  - Could be 1-bit, could be complete state machine
  - Indexed by PC address of Branch – without tags
- In Fetch state of branch:
  - BTB identifies branch
  - Predictor from BHT used to make prediction
- When branch completes
  - Update corresponding Predictor

Dynamic Branch Prediction: Usual Division

- Branch Target Buffer (BTB): identify branches and hold *taken addresses*
  - *Trick: identify branch before fetching instruction!*
- Branch History Table(BHT)
  - Table makes prediction by keeping long-term history
    - Example: Simple 1-bit BHT: keep last direction of branch
  - No address check: Can be good, can be bad….
- 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
- Performance = *ƒ*(accuracy, cost of misprediction)
  - Misprediction ⇒ Flush Reorder Buffer

Dynamic Branch Prediction: 2-bit predictor

- Solution: 2-bit scheme where change prediction only if get misprediction *twice:*



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%
- 4096 about as good as infinite table (in Alpha 21164)

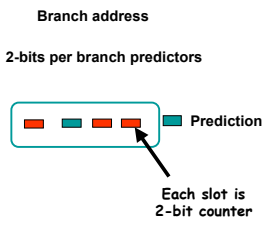
Correlating Branches

- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Two possibilities; Current branch depends on:
  - Last m most recently executed branches anywhere in program
    Produces a "GA" (for "global address") in the Yeh and Patt classification (e.g. GAg)
  - Last m most recent outcomes of same branch.
    Produces a "PA" (for "per address") in same classification (e.g. PAg)
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper branch history table entry
  - A single history table shared by all branches (appends a "g" at end), indexed by history value.
  - Address is used along with history to select table entry (appends a "p" at end of classification)
  - If only portion of address used, often appends an "s" to indicate "set-indexed" tables (I.e. GAs)

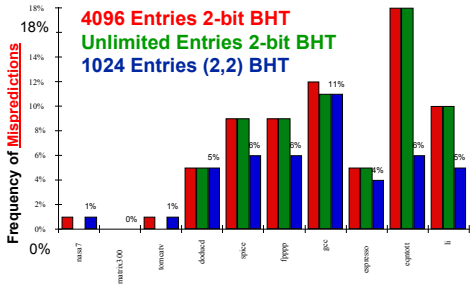
Correlating Branches

- For instance, consider global history, set-indexed BHT. That gives us a GAs history table.

(2,2) GAs predictor
- First 2 means that we keep two bits of history
- Second means that we have 2 bit counters in each slot.
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction
- Note that the original two-bit counter solution would be a (0,2) GAs predictor
- Note also that aliasing is possible here...



Accuracy of Different Schemes



Peer : Superpipelined (SP) v. Superscalar (SS)

- Which are true? Assume the same technology and design effort

A. SP likely has a higher clock rate than SS

B. SP likely has a higher CPI than SS

C. A 10-stage SP has the same peak instruction fetch bandwidth as a 5-stage SS

1. ABC: FFF
2. ABC: FFT
3. ABC: FTF
4. ABC: FTT
5. ABC: TFF
6. ABC: TFT
7. ABC: TTF
8. ABC: TTT

Peer: Superpipelined (SP) v. Superscalar (SS)

- Assume a 10-stage SP vs. as a 5-stage SS

A. You would expect the latency of the SP instruction cache to be half that of SS

B. The branch delay for SP is (likely) longer than the 1-instruction branch delay of SS

C. Although SP has a faster clock rate, SS is likely faster since it has fewer pipeline hazards

1.ABC: FFF 5. ABC: TFF
2.ABC: FFT 6. ABC: TFT
3.ABC: FTF 7. ABC: TTF
4.ABC: FTT 8. ABC: TTT

Peer: SP, SS, and Branch Prediction

- Assume a 10-stage SP vs. as a 5-stage SS

A. Branch prediction more important for SP v. SS

B. 2-bit branch predictor is useful for **`beq`**, **`bne`**

C. 2-bit branch predictor is useful for **`jr`**

1.ABC: FFF 5. ABC: TFF
2.ABC: FFT 6. ABC: TFT
3.ABC: FTF 7. ABC: TTF
4.ABC: FTT 8. ABC: TTT

Low CPI vs. Limits of Superscalar

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, harder to decode and issue
  - Even 2-scalar => compare 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

VLIW: Very Long Instruction Word

- 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 can 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 to have enough instructions

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?)

Loop Unrolling in VLIW

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

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration
Need more registers in VLIW(EPIC => 128int + 128FP)

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?

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

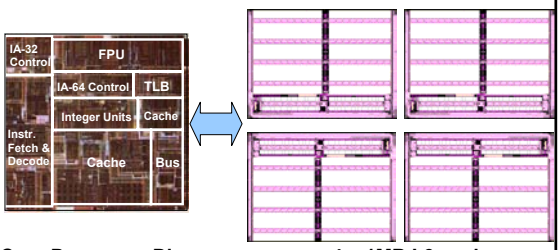
- **5-bit template field within each bundle describes both the presence of any stops associated with the bundle *and* the execution unit type required by each instruction within the bundle**

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

Itanium™ Processor Silicon

(Copyright: Intel at Hotchips '00)



Core Processor Die **4 x 1MB L3 cache**

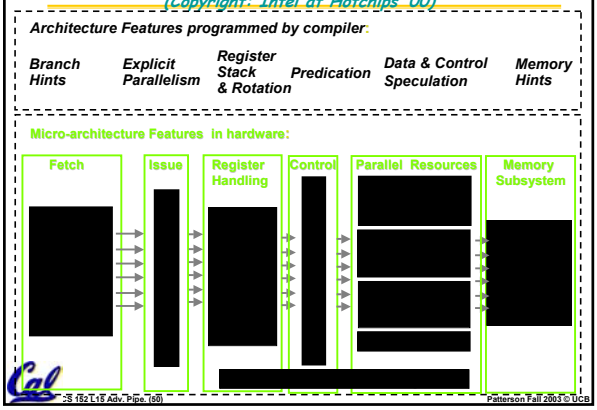
Itanium™ Machine Characteristics

(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

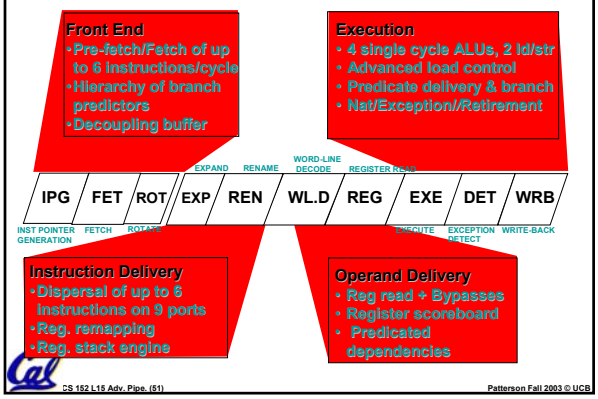
Itanium™ EPIC Design Maximizes SW-HW Synergy

(Copyright: Intel at Hotchips '00)



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!

Cost (Microprocessor Report, 8/25/03)

Processor	Alpha 21364	AMD Athlon XP	HP PA-8700	IBM Power4+	Intel Itanium 2	Intel XeonMP	Intel Xeon	MIPS R14000	Sun Ultra-III
Clock Rate	1.15GHz	2.17GHz	870MHz	1.45GHz	1.GHz	2.0GHz	3.06GHz	600MHz	1.05GHz
Cache (I/D/L2/L3)	64K/64K/ 1.75M	64K/64K/ 512K	750K/ 1.5M	64K/32K/ 1.5MB	16K/16K/ 256K/3M	12K/8K/ 512K/2M	12K/8K/ 512K	32K/32K	32K/64K
Issue Rate	4 issue	3 x86 instr	4 issue	8 issue	8 Issue	3 ROPs	3 ROPs	4 issue	4 issue
Pipeline Stages	7/9 stages	9/11 stages	7/9 stages	12/17 stages	8 stages	22/24 stages	22/24 stages	6 stages	14/15 stages
Out of Order	80 instr	72ROPs	56 instr	200 instr	None	126 ROPs	126 ROPs	48 instr	None
Rename Regs	48/41	36/36	56 total	48/40	328 total	128 total	128 total	32/32	None
BHT Entries	4K x 9-bit	4K x 2-bit	2K x 2-bit	3 x 16K x 1-bit	512 x 2-bit	4K x 2-bit	4K x 2-bit	2K x 2-bit	16K x 2-bit
TLB Entries	128/128	280/288	240 unified	1,024 unified	32L1I/32L1D/ 256L2D	128I/64D	128I/64D	64 unified	128I/512D
Memory B/W	12GB/s	2.7GB/s	1.54GB/s	12.8GB/s	6.4GB/s	3.2GB/s	4.3GB/s	1.6GB/s	4.8GB/s
Package	FC-LGA-1443	PGA-462	LGA-544	MCM	mPGA-700	mPGA-603	PGA-423	FCBGA-1153	FC-LGA 1368
IC Process	0.18µm 7M	0.13µm 6M	0.18µm 7M	0.13µm 7m	0.18µm 6M	0.13µm 6M	0.13µm 6M	0.15µm 7M	0.15µm 7M
Die Size	397mm²	101mm²	304mm²	267mm²**	418mm²*	211mm²	131mm²	142mm²	210mm²
Transistors	135 million	54.3 million	130 million	184 million**	221 million	160 million*	55 million	7.2 million	29 million
Est Die Cost	$180*	$46*	$96*	$144**	$166*	$64*	$55*	$68*	$72*
Power (Max)	110W*	76W(MTP)	75W*	85W**	130W	65W(Max)	82W(TDP)	16W*	75W*
Availability	1Q03	1Q03	3Q02	4Q02	3Q02	1Q03	4Q02	2Q02	1Q02

- 3X die size Pentium 4, 1/3 clock rate Pentium 4
- Cache size (KB): 16+16+256+3076 v. 12+8+512

Performance (Microprocessor Report, 8/25/03)

Processor	Alpha 21364	AMD Athlon XP	HP PA-8700	IBM Power 4+	Intel Itanium 2	Intel XeonMP	Intel Xeon	MIPS R14000	Sun UltraSPARC III
System or Motherboard	Alpha GS1280/7	ASUS A7N8X	HP9000 C3750	pSeries 650 6M2	HP RX2600	Dell PwrEdg 6650	Dell Prec. 350	SGI 3200	Sun Blade 2050
Clock Rate	1.15GHz	2.17GHz	870MHz	1.45GHz	1.0GHz	2.0GHz	3.06GHz	600MHz	1.05GHz
External Cache	None	None	None	16MB	None	None	None	8MB	8MB
164.gzip	583	1,026	588	673	583	758	1,138	322	433
175.vpr	822	653	688	902	704	625	606	572	460
176.gcc	859	755	906	914	1,014	1,100	1,236	445	577
181.mcf	712	420	494	1,391	834	599	773	783	659
186.crafty	982	1,292	751	884	781	712	1,179	502	558
197.parser	514	905	495	381	660	778	1,025	409	488
252.eon	958	1,483	592	1,150	1,004	920	1,387	507	527
253.perlbmk	768	1,306	619	712	815	952	1,381	367	540
254.gap	636	1,059	339	936	680	722	1,417	308	372
255.vortex	1,094	1,608	1,196	1,428	1,193	1,118	1,658	679	738
256.bzip2	824	840	534	965	759	712	856	493	629
300.twolf	1,018	887	911	1,198	880	1,009	900	645	570
SPECint_base2000	795	960	642	909	810	816	1,085	483	537
168.wupside	883	1,131	446	1,532	1,003	816	1,406	434	659
171.swim	3,590	1,006	931	1,417	3,205	848	1,837	529	980
172.mgrid	708	799	621	850	1,720	449	1,047	379	487
173.applu	1,518	654	702	979	2,033	496	1,168	381	310
177.mesa	928	1,103	694	737	642	814	1,165	425	543
178.galgel	2,105	738	1,603	3,186	2,505	1,200	1,536	1,398	1,713
179.art	2,014	495	670	1,864	4,226	1,147	716	1,436	9,389
183.equake	519	730	413	2,098	1,871	449	1,291	347	645
187.facerec	1,105	1,008	430	1,515	1,152	762	1,315	647	958
188.ammp	735	587	553	923	788	729	644	573	509
189.lucas	1,522	853	448	1,306	1,206	682	1,522	442	371
191.fma3d	1,019	850	404	898	747	551	1,089	306	400
200.sixtrack	469	538	471	621	894	376	564	298	366
301.aspi	1,242	705	696	966	678	695	833	406	471
SPECfp_base2000	1,124	776	600	1,221	1,356	677	1,092	499	701

Peformance of IA-64 Itanium?

- Whether this approach will result in significantly higher performance than other recent processors is unclear
- The clock rate of Itanium (733 MHz) and Itanium II (1.0 GHz) is competitive but slower than the clock rates of several dynamically-scheduled machines, which are already available, including the Intel Pentium 4 and AMD Operteron

Summary

- Loop unrolling ⇒ Multiple iterations of loop in SW:
  - Amortizes loop overhead over several iterations
  - Gives more opportunity for scheduling around stalls
- Very Long Instruction Word machines (VLIW) ⇒ Multiple operations coded in single, long instruction
  - Requires sophisticated compiler to decide which operations can be done in parallel
  - Trace scheduling ⇒ find common path and schedule code as if branches didn't exist (+ add "fixup code")
- Both require additional registers