CS252
Graduate Computer Architecture
Lecture 8

Instruction Level Parallelism: Getting the CPI < 1

September 27, 2000

Prof. John Kubiatowicz

Review: Reorder Buffer (ROB)

- Use of reorder buffer
  - In-order issue, Out-of-order execution, In-order commit
  - Holds results until they can be committed in order
    - » Serves as source of values until instructions committed
  - Provides support for precise exceptions/Speculation: simply throw out instructions later than excepted instruction.
  - Commits user-visible state in instruction order
  - Stores sent to memory system only when they reach head of buffer
- In-Order-Commit is important because:
  - Allows the generation of precise exceptions
  - Allows speculation across branches

Review: Memory Disambiguation

- Question: Given a load that follows a store in program order, are the two related?
  - Trying to detect RAW hazards through memory
  - Stores commit in order (ROB), so no WAR/WAW memory hazards.
- Implementation
  - Keep queue of stores, in prog order
  - Watch for position of new loads relative to existing stores
- When have address for load, check store queue:
  - If *any* store prior to load is waiting for its address, stall load.
  - If load address matches earlier store address (associative lookup), then we have a *memory-induced RAW hazard*:
    - » store value available ⇒ return value
    - » store value not available ⇒ return ROB number of source
  - Otherwise, send out request to memory
- Will relax exact dependency checking in later lecture

Review: Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA
- Key insight: Allocate a new physical destination register for every instruction that writes
  - Removes all chance of WAR or WAW hazards
  - Similar to compiler transformation called Static Single Assignment
    - » Like hardware-based dynamic compilation?
- Mechanism? Keep a translation table:
  - ISA register ⇒ physical register mapping
  - When register written, replace entry with new register from freelist.
  - Physical register free when not used by any active instructions
- Advantages of explicit renaming:
  - Decouples *renaming* from *scheduling:*
  - Allows data to be fetched from single register file
    - » No need to bypass values from reorder buffer
    - » Better utilization of register ports

Instruction Level Parallelism

- High speed execution based on *instruction level parallelism* (ilp): potential of short instruction sequences to execute in parallel
- High-speed microprocessors exploit ILP by:
  1) pipelined execution: overlap instructions
  2) Out-of-order execution (commit in-order)
  3) Multiple issue: issue and execute multiple instructions per clock cycle
  4) Vector instructions: many independent ops specified with a single instruction
- Memory accesses for high-speed microprocessor?
  - Data Cache possibly multiported, multiple levels

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
- (Very) Long Instruction Words (V)LIW: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
  - Joint HP/Intel agreement in 1999/2000?
  - Intel Architecture-64 (IA-64) 64-bit address
    - » Style: "Explicitly Parallel Instruction Computer (EPIC)"
  - New SUN MAJIC Architecture: VLIW for Java
- Vector Processing: Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors
- Anticipated success lead to use of Instructions Per Clock cycle (IPC) vs. CPI

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - 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

Review: Unrolled Loop that Minimizes Stalls for Scalar

```
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
```

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration

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 (1.5X)

Dynamic Scheduling in Superscalar The easy way

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point
- Issue 2X Clock Rate, so that issue remains in order
- Only FP loads might cause dependency between integer and FP issue:
  - Replace load reservation station with a load queue; operands must be read in the order they are fetched
  - Load checks addresses in Store Queue to avoid RAW violation
  - Store checks addresses in Load Queue to avoid WAR,WAW
  - Called "decoupled architecture": compare with Smith paper

Multiple Issue Challenges

- 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, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  - Register file: need 2x reads and 1x writes/cycle
  - Rename logic: must be able to rename same register multiple times in one cycle! For instance, consider 4-way issue:

```
add r1, r2, r3          add p11, p4, p7
sub r4, r1, r2    ⇒     sub p22, p11, p4
lw  r1, 4(r4)           lw  p23, 4(p22)
add r5, r1, r2          add p12, p23, p4
```

  Imagine doing this transformation in a single cycle!
  - Result buses: Need to complete multiple instructions/cycle
    - » So, need multiple buses with associated matching logic at every reservation station.
    - » Or, need multiple forwarding paths

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
  - In EPIC, 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: 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
```

LD to ADDD: 1 Cycle
ADDD to SD: 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
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

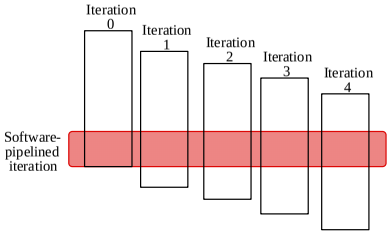
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

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)



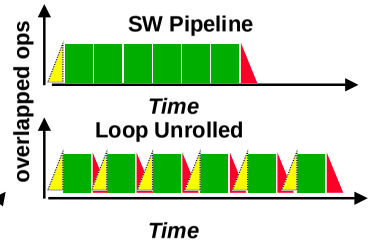
Recall: Software Pipelining Example

Before: Unrolled 3 times

```
1  LD    F0,0(R1)
2  ADDD  F4,F0,F2
3  SD    0(R1),F4
4  LD    F6,-8(R1)
5  ADDD  F8,F6,F2
6  SD    -8(R1),F8
7  LD    F10,-16(R1)
8  ADDD  F12,F10,F2
9  SD    -16(R1),F12
10 SUBI  R1,R1,#24
11 BNEZ  R1,LOOP
```

After: Software Pipelined

```
1  SD    0(R1),F4 ; Stores M[i]
2  ADDD  F4,F0,F2 ; Adds to M[i-1]
3  LD    F0,-16(R1);Loads M[i-2]
4  SUBI  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

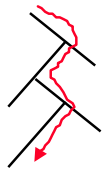
Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,-48(R1)	ST 0(R1),F4	ADDD F4,F0,F2			1
LD F6,-56(R1)	ST -8(R1),F8	ADDD F8,F6,F2		SUBI R1,R1,#24	2
LD F10,-40(R1)	ST 8(R1),F12	ADDD F12,F10,F2		BNEZ R1,LOOP	3

- 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 less 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 noops filing 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

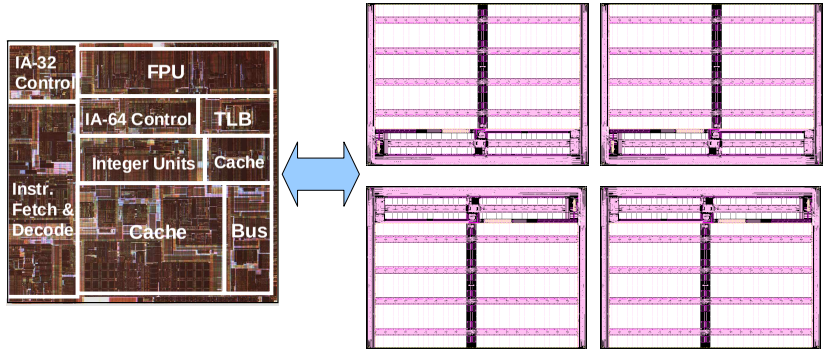
- Smaller code size
- Binary compatability across generations of hardware
- Simplified Hardware for decoding, issuing instructions
- No Interlock Hardware (compiler checks?)
- More registers, but simplified Hardware for Register Ports (multiple independent register files?)

Intel/HP "Explicitly Parallel Instruction Computer (EPIC)"

- 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
- 128 integer registers + 128 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: instruction set architecture; EPIC is type
  - VLIW = EPIC?
- Itanium™ is name of first implementation (2000/2001?)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process

Itanium™ Processor Silicon

(Copyright: Intel at Hotchips '00)



Core Processor Die

4 x 1MB L3 cache

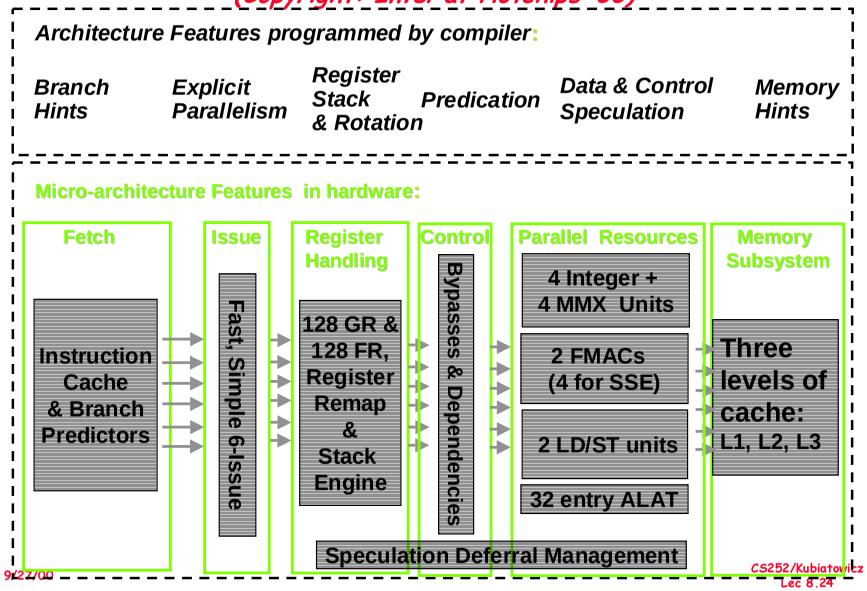
Itanium™ Machine Characteristics

(Copyright: Intel at Hotchips '00)

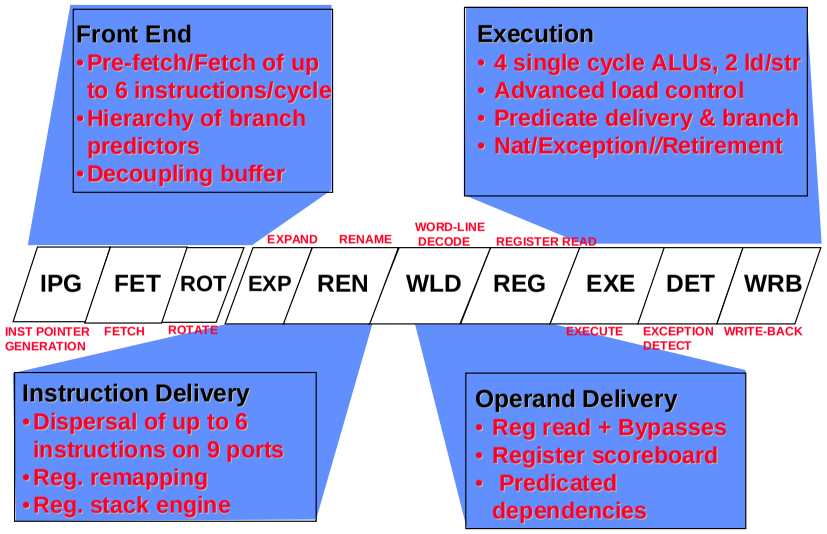
Frequency	800 MHz
Transistor Count	25.4M CPU; 295M L3
Process	0.18u CMOS, 6 metal layer
Package	Organic Land Grid Array
Machine Width	6 insts/clock (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)
Registers	14 ported 128 GR & 128 FR; 64 Predicates
Speculation	32 entry ALAT, Exception Deferral
Branch Prediction	Multilevel 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)



CS 252 Administrivia

- Exam: Wednesday 10/18
  Location: TBA
  TIME: 5:30 - 8:30
- This info is on the Lecture page (has been)
- Meet at LaVal's afterwards for Pizza and Beverages

- Assignment up tomorrow.
  - Done in pairs. Put both names on papers.
  - Make sure you have partners! Feel free to use mailing list for this.

Architecture in practice

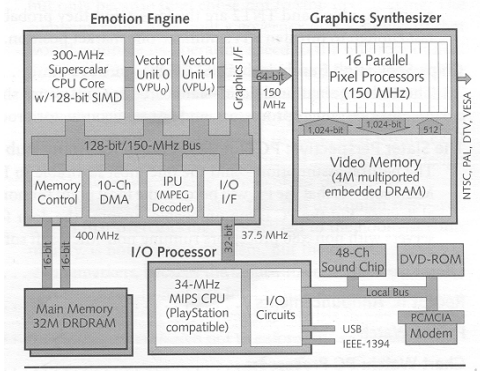


Figure 1. PlayStation 2000 employs an unprecedented level of parallelism to achieve workstation-class 3D performance.

Figure 2. PlayStation 2000 screenshot. (Source: Namco)

- (as reported in Microprocessor Report, Vol 13, No. 5)
  - Emotion Engine: 6.2 GFLOPS, 75 million polygons per second
  - Graphics Synthesizer: 2.4 Billion pixels per second
  - Claim: *Toy Story* realism brought to games!

Playstation 2000 Continued

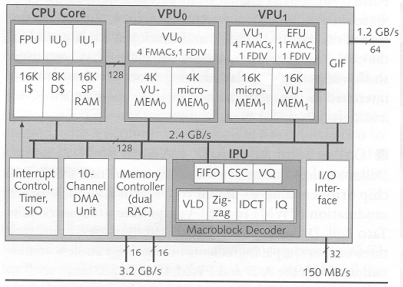


Figure 3. The PSX2's Emotion Engine provides ten floating-point multiplier-accumulators, four floating-point dividers, and an MPEG-2 decoder to deliver killer multimedia performance.

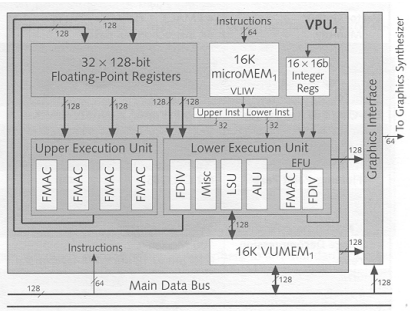


Figure 5. Each vector unit has enough parallelism to complete a vertex operation (19 mul-adds + 1 divide) every seven cycles.

- Emotion Engine:
  - Superscalar MIPS core
  - Vector Coprocessor Pipelines
  - RAMBUS DRAM interface
- Sample Vector Unit
  - 2-wide VLIW
  - Includes Microcode Memory
  - High-level instructions like matrix-multiply

Limits to Multi-Issue Machines

- Inherent limitations of ILP
  - 1 branch in 5: How to keep a 5-way VLIW busy?
  - Latencies of units: many operations must be scheduled
  - Need about Pipeline Depth x No. Functional Units of independent operations to keep all pipelines busy.
  - Difficulties in building HW
  - Easy: More instruction bandwidth
  - Easy: Duplicate FUs to get parallel execution
  - Hard: Increase ports to Register File (bandwidth)
    - » VLIW example needs 7 read and 3 write for Int. Reg. & 5 read and 3 write for FP reg
  - Harder: Increase ports to memory (bandwidth)
  - Decoding Superscalar and impact on clock rate, pipeline depth?

Limits to Multi-Issue Machines

- Limitations specific to either Superscalar or VLIW implementation
  - Decode issue in Superscalar: how wide practical?
  - VLIW code size: unroll loops + wasted fields in VLIW
    - » IA-64 compresses dependent instructions, but still larger
  - VLIW lock step => 1 hazard & all instructions stall
    - » IA-64 not lock step? Dynamic pipeline?
  - VLIW & binary compatibilityIA-64 promises binary compatibility

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 mechanims with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX
  - Motorola AltaVec
  - Supersparc Multimedia ops, etc.

Limits to ILP

Initial HW Model here; MIPS compilers.

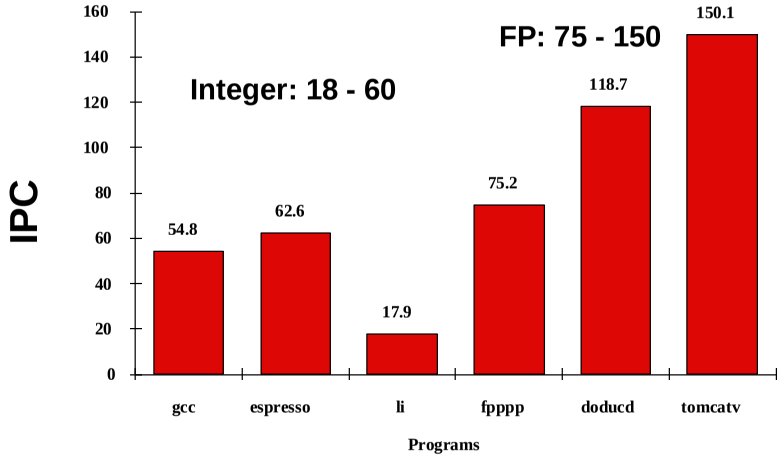
Assumptions for ideal/perfect machine to start:

1. *Register renaming*–infinite virtual registers and all WAW & WAR hazards are avoided
2. *Branch prediction*–perfect; no mispredictions
3. *Jump prediction*–all jumps perfectly predicted => machine with perfect speculation & an unbounded buffer of instructions available
4. *Memory-address alias analysis*–addresses are known & a store can be moved before a load provided addresses not equal

1 cycle latency for all instructions; unlimited number of instructions issued per clock cycle

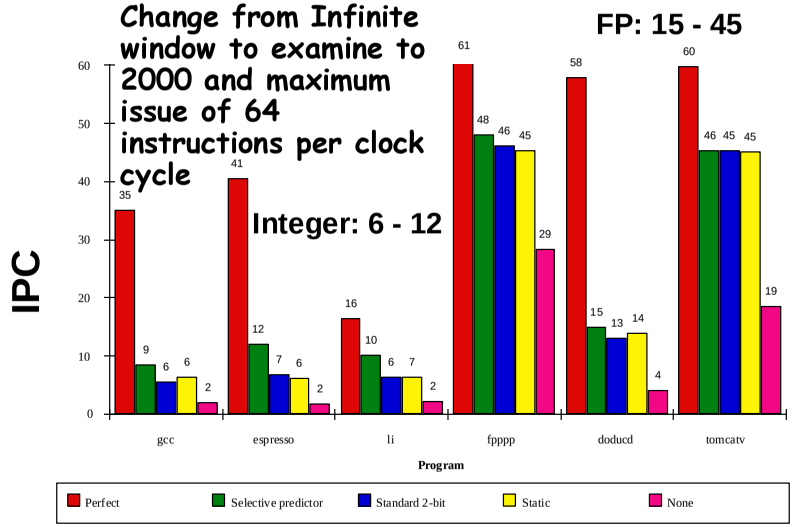
Upper Limit to ILP: Ideal Machine

(Figure 4.38, page 319)



More Realistic HW: Branch Impact

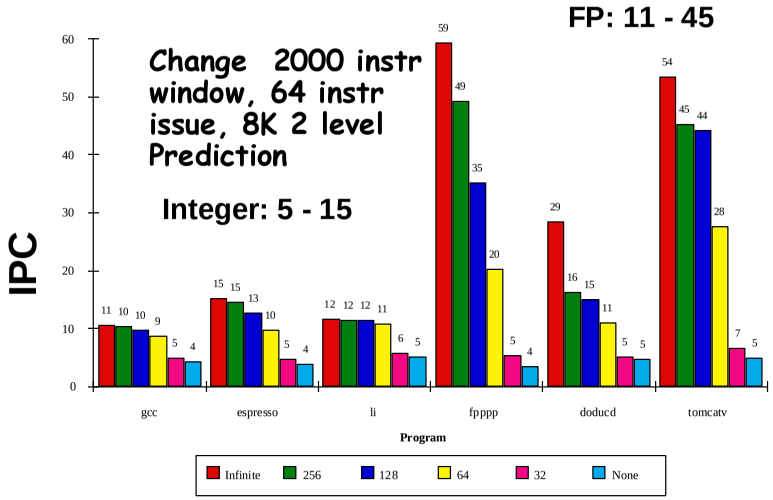
Figure 4.40, Page 323



Perfect Pick Cor. or BHT BHT (512) Profile No prediction

More Realistic HW: Register Impact

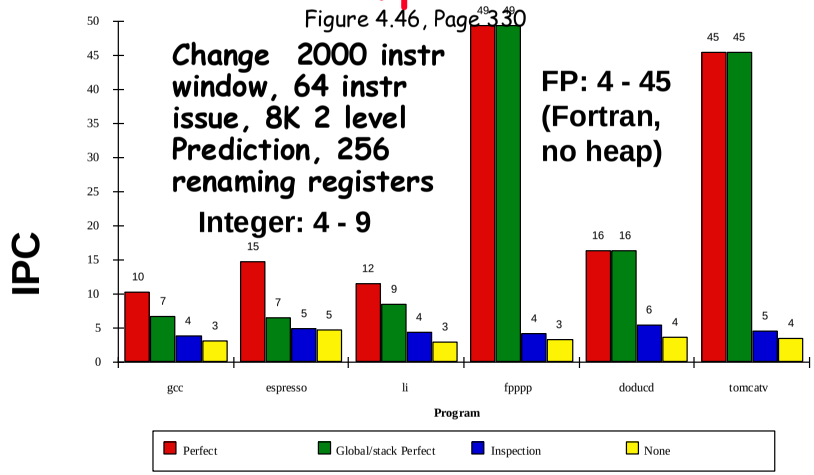
Figure 4.44, Page 328



Infinite 256 128 64 32 None

More Realistic HW: Alias Impact

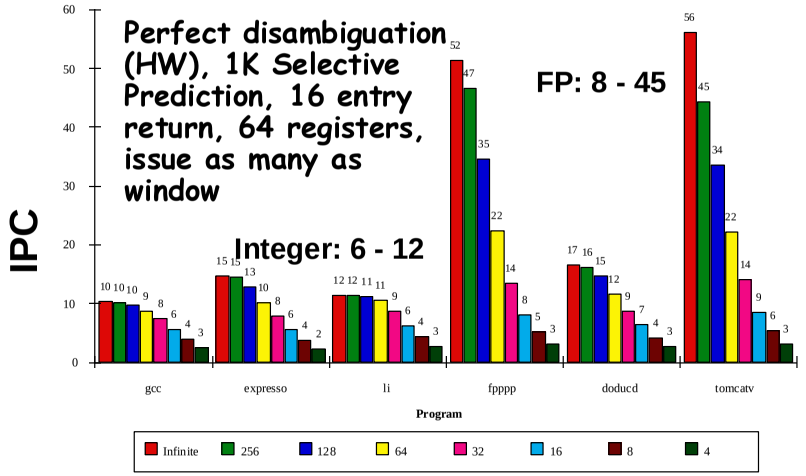
Figure 4.46, Page 330



Perfect Global/Stack perf; heap conflicts Inspec. Assem. None

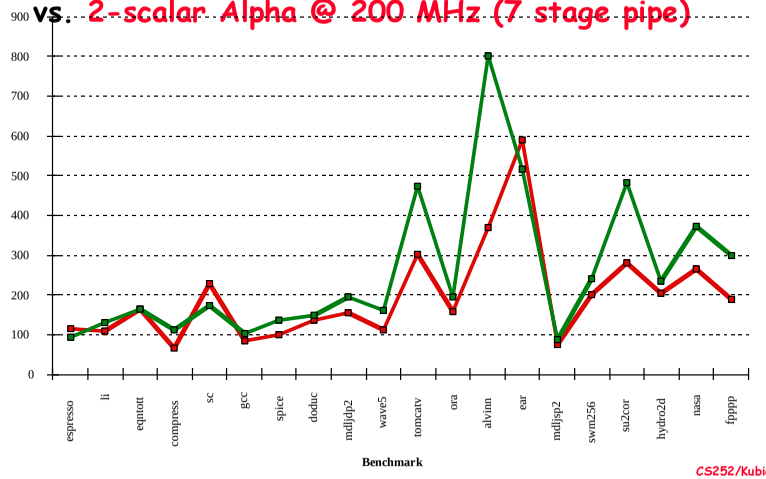
Realistic HW for '9X: Window Impact

(Figure 4.48, Page 332)



Braniac vs. Speed Demon(1993)

- 8-scalar IBM Power-2 @ 71.5 MHz (5 stage pipe)
- vs. 2-scalar Alpha @ 200 MHz (7 stage pipe)



Problems with scalar approach to ILP extraction

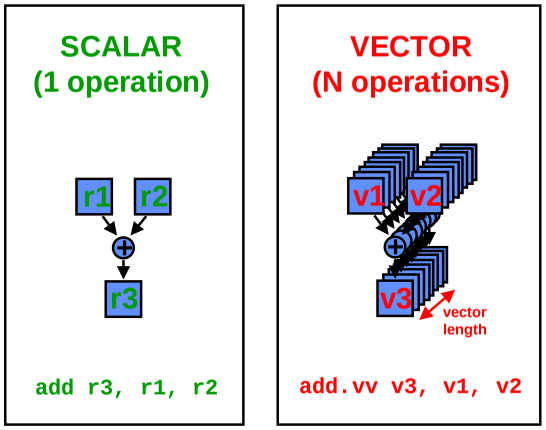
- Limits to conventional exploitation of ILP:
  - *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)
  - *branch prediction*: branches get in the way of wide issue. They are too unpredictable.
  - *instruction fetch and decode*: at some point, its hard to fetch and decode more instructions per clock cycle
  - *register renaming*: Rename logic gets really complicate for many instructions
  - *cache hit rate*: some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality

Cost-performance of simple vs. OOO

MIPS MPUs	R5000	R10000	10k/5k
• Clock Rate	200 MHz	195 MHz	1.0x
• On-Chip Caches	32K/32K	32K/32K	1.0x
• Instructions/Cycle	1(+ FP)	4	4.0x
• Pipe stages	5	5-7	1.2x
• Model	In-order	Out-of-order	---
• Die Size (mm^2)	84	298	3.5x
– without cache, TLB	32	205	6.3x
• Development (man yr.)	60	300	5.0x
• SPECint_base95	5.7	8.8	1.6x

Alternative Model: Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



Properties of Vector Processors

- Each result independent of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate
- Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency of over - 64 elements
  => no (data) caches required! (Do use instruction cache)
- Reduces branches and branch problems in pipelines
- Single vector instruction implies lots of work (- loop)
  => fewer instruction fetches

Operation & Instruction Count: RISC v. Vector Processor

(from F. Quintana, U. Barcelona.)

Spec92fp	Operations (Millions)			Instructions (M)		
Program	RISC	Vector	R / V	RISC	Vector	R / V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Styles of Vector Architectures

- *memory-memory vector processors*: all vector operations are memory to memory
- *vector-register processors*: all vector operations between vector registers (except load and store)
  - Vector equivalent of load-store architectures
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
  - We assume vector-register for rest of lectures

Components of Vector Processor

- *Vector Register*: fixed length bank holding a single vector
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 64-bit elements
- *Vector Functional Units (FUs)*: fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- *Vector Load-Store Units (LSUs)*: fully pipelined unit to load or store a vector; may have multiple LSUs
- *Scalar registers*: single element for FP scalar or address
- Cross-bar to connect FUs , LSUs, registers

"DLXV" Vector Instructions

	Instr.	Operands	Operation	Comment
•	ADDV	V1,V2,V3	V1=V2+V3	vector + vector
•	ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
•	MULTV	V1,V2,V3	V1=V2xV3	vector x vector
•	MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
•	LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
•	LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
•	LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indir.("gather")
•	CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
•	MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
•	MOV	VM,R1	Vec. Mask = R1	set vector mask

Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - » Fastest
  - Non-unit (constant) stride
  - Indexed (gather-scatter)
    - » Vector equivalent of register indirect
    - » Good for sparse arrays of data
    - » Increases number of programs that vectorize

DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector →

```
        LD      F0,a
        ADDI    R4,Rx,#512  ;last address to load
loop:   LD      F2, 0(Rx)   ;load X(i)
        MULTD   F2,F0,F2    ;a*X(i)
        LD      F4, 0(Ry)   ;load Y(i)
        ADDD    F4,F2,F4    ;a*X(i) + Y(i)
        SD      F4,0(Ry)    ;store into Y(i)
        ADDI    Rx,Rx,#8    ;increment index to X
        ADDI    Ry,Ry,#8    ;increment index to Y
        SUB     R20,R4,Rx   ;compute bound
        BNZ     R20,loop    ;check if done
```

```
LD      F0,a        ;load scalar a
LV      V1,Rx       ;load vector X
MULTS   V2,F0,V1    ;vector-scalar mult.
LV      V3,Ry       ;load vector Y
ADDV    V4,V2,V3    ;add
SV      Ry,V4       ;store the result
```

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Example Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2 L, 1 S
Cray YMP	1988	166 MHz	8	64	8	2 L, 1 S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Conv. C-1	1984	10 MHz	8	128	4	1
Conv. C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Vector Linpack Performance (MFLOPS)

Machine	Year	Clock	100x100	1kx1k	Peak(Procs)
Cray 1	1976	80 MHz	12	110	160(1)
Cray XMP	1983	120 MHz	121	218	940(4)
Cray YMP	1988	166 MHz	150	307	2,667(8)
Cray C-90	1991	240 MHz	387	902	15,238(16)
Cray T-90	1996	455 MHz	705	1603	57,600(32)
Conv. C-1	1984	10 MHz	3	--	20(1)
Conv. C-4	1994	135 MHz	160	2531	3240(4)
Fuj. VP200	1982	133 MHz	18	422	533(1)
NEC SX/2	1984	166 MHz	43	885	1300(1)
NEC SX/3	1995	400 MHz	368	2757	25,600(4)

Vector Pitfalls

- Pitfall: Concentrating on peak performance and ignoring start-up overhead:
  N_V (length faster than scalar) > 100!
- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
  - failure of Cray competitor (ETA) from his former company
- Pitfall: Good processor vector performance without providing good memory bandwidth
  - MMX?

Vector Advantages

- Easy to get high performance; N operations:
  - are independent
  - use same functional unit
  - access disjoint registers
  - access registers in same order as previous instructions
  - access contiguous memory words or known pattern
  - can exploit large memory bandwidth
  - hide memory latency (and any other latency)
- Scalable: (get higher performance by adding HW resources)
- Compact: Describe N operations with 1 short instruction
- Predictable: performance vs. statistical performance (cache)
- Multimedia ready: N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
- Mature, developed compiler technology
- Vector Disadvantage: Out of Fashion?
  - Hard to say. Many irregular loop structures seem to still be hard to vectorize automatically.
  - Theory of some researchers that SIMD model has great potential.

Summary #1

- Explicit Renaming: more physical registers than needed by ISA. Uses a translation table
- Precise exceptions/Speculation: Out-of-order execution, In-order commit (reorder buffer)
- 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 ⇒ "Flynn barrier"
- SW Pipelining
  - Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead
- Branch prediction is one of the most crucial factors to performance of superscalar execution.

Summary #2

- Vector model accommodates long memory latency, doesn't rely on caches as does Out-Of-Order, superscalar/VLIW designs
- Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer hazards, fewer branches, fewer mispredicted branches, ...
- What % of computation is vectorizable?
- Is vector a good match to new apps such as multimedia, DSP?