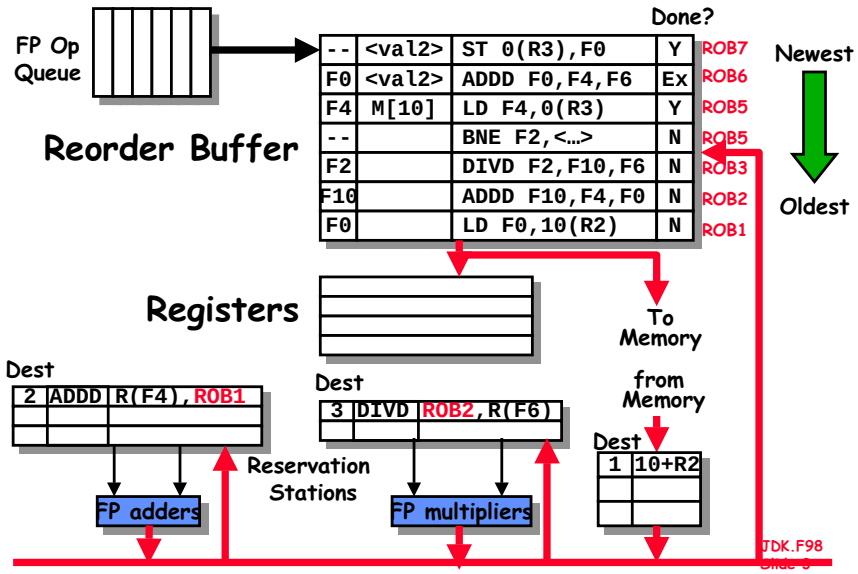
Lecture 7: Instruction Level Parallelism 2: Getting the CPI < 1

Prof. John Kubiatowicz
Computer Science 252
Fall 1998

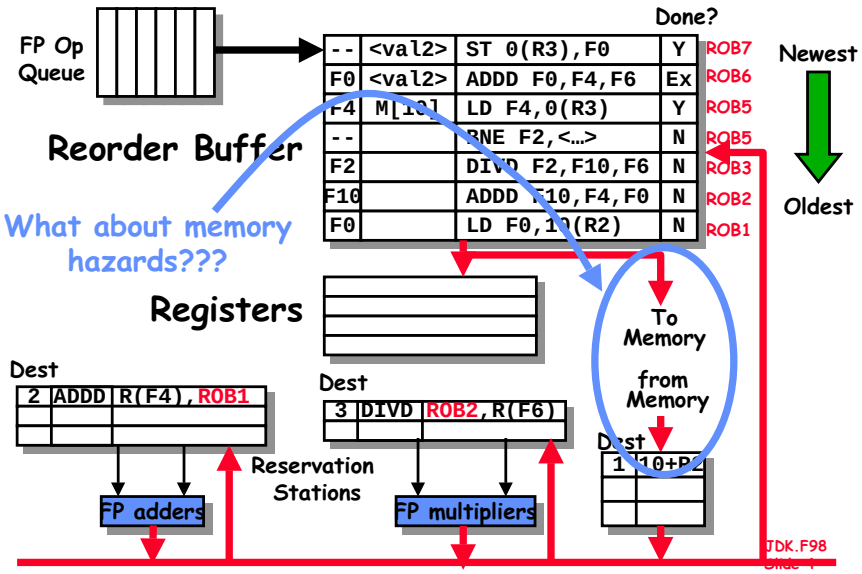
Review: Hardware unrolling, in-order commit, and explicit register renaming

- Machines that use hardware techniques with register renaming (such as tomasulo) can unroll loops automatically in hardware
- In-Order-Commit is important because:
  - Allows the generation of precise exceptions
  - Allows speculation across branches
- Use of reorder buffer
  - Commits user-visible state in instruction order
- Explicit register renaming uses a rename table and large bank of physical registers

Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Memory Disambiguation: Sorting out RAW Hazards in memory

- Question: Given a load that follows a store in program order, are the two related?
  - (Alternatively: is there a RAW hazard between the store and the load)?

```
Eg:   st    0(R2),R5
      ld    R6,0(R3)
```

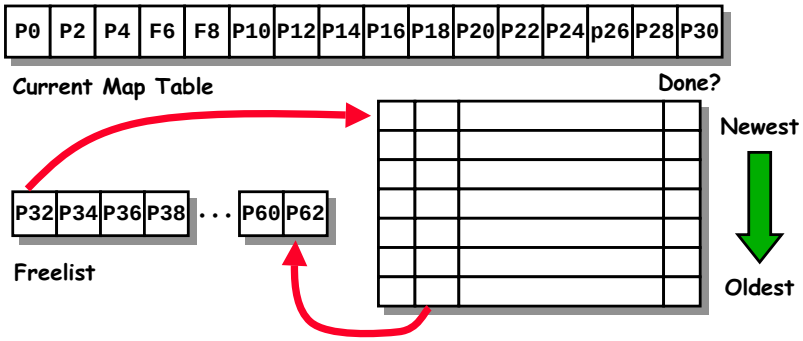
- Can we go ahead and start the load early?
  - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - We might want to issue/begin execution of both operations in same cycle.
  - Today: Answer is that we are not allowed to start load until we know that address 0(R2) ≠ 0(R3)
  - Next Week: We might guess at whether or not they are dependent (called "dependence speculation") and use reorder buffer to fixup if we are wrong.

Hardware Support for Memory Disambiguation

- Need buffer to keep track of all outstanding stores to memory, in program order.
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: will retire stores from this buffer in program order
- When issuing a load, record current head of store queue (know which stores are ahead of you).
- 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
- Actual stores commit in order, so no worry about WAR/WAW hazards through memory.

Explicit register renaming:

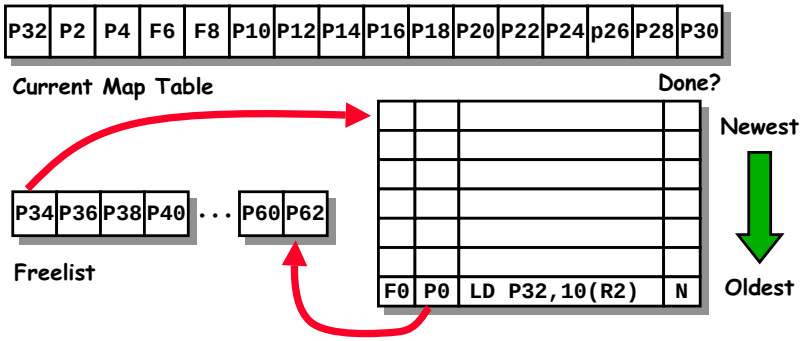
Hardware equivalent of static, single-assignment (SSA) compiler form



- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist
- Used on: R10000, Alpha 21264, HP PA8000

Explicit register renaming:

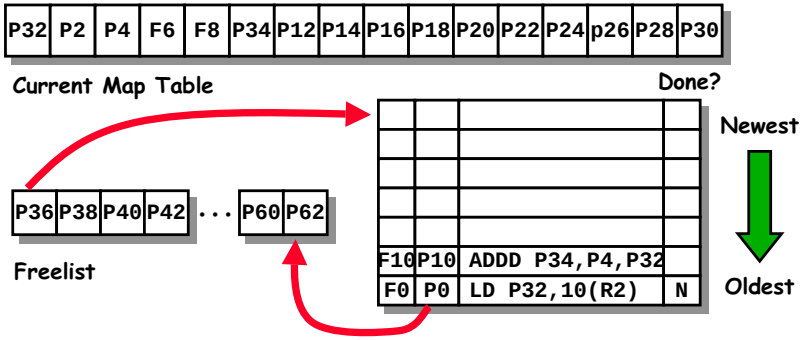
Hardware equivalent of static, single-assignment (SSA) compiler form



- Note that physical register P0 is "dead" (or not "live") past the point of this load.
  - When we go to commit the load, we free up

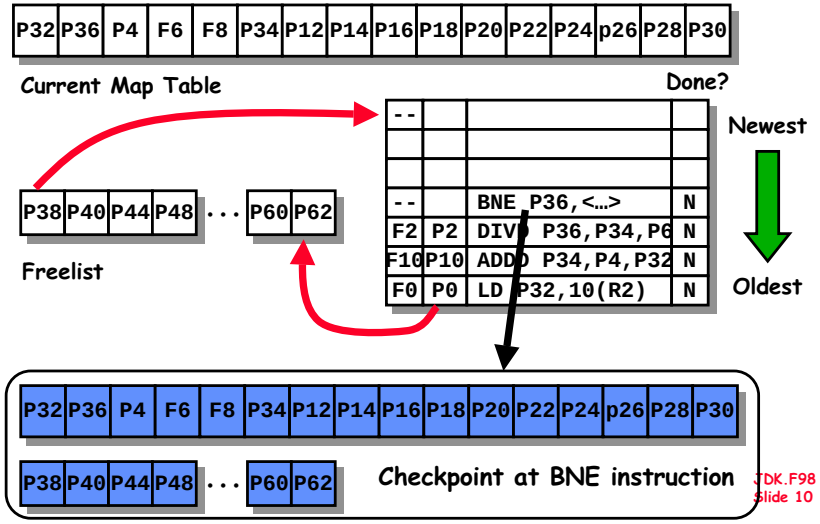
Explicit register renaming:

Hardware equivalent of static, single-assignment (SSA) compiler form



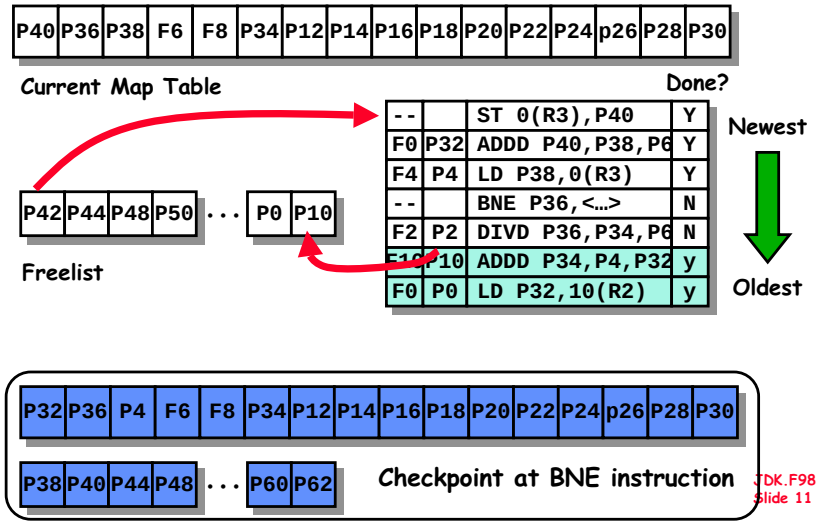
Explicit register renaming:

Hardware equivalent of static, single-assignment (SSA) compiler form



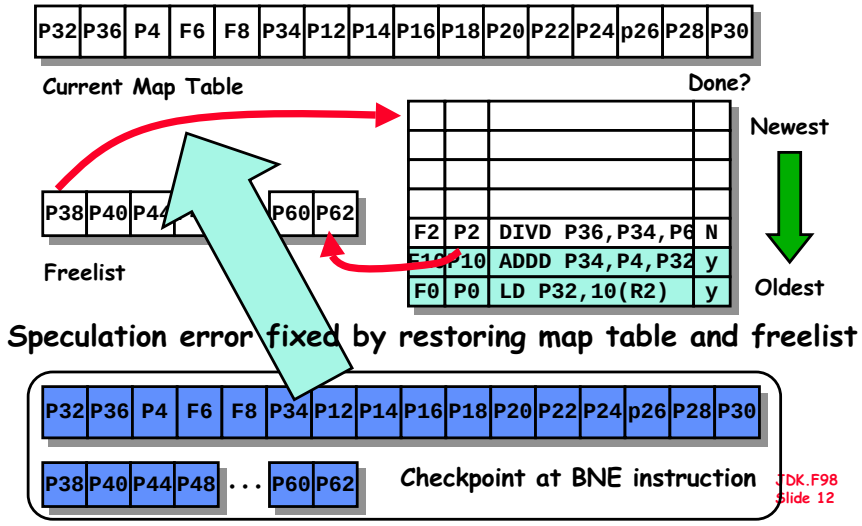
Explicit register renaming:

Hardware equivalent of static, single-assignment (SSA) compiler form



Explicit register renaming:

Hardware equivalent of static, single-assignment (SSA) compiler form



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

- Two variations
- 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)"
- 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

- 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"

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
  - Multiported rename logic: must be able to rename same register multiple times in one cycle!
- VLIW: 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

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

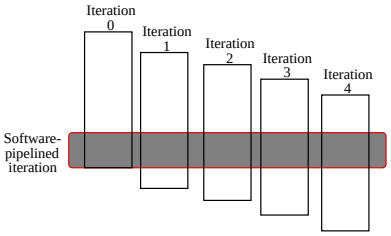
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

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

Software Pipelining

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



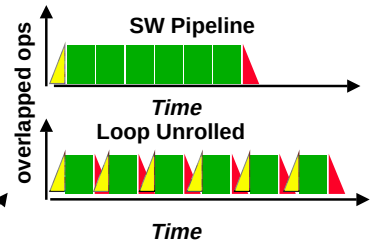
Software Pipelining Example

Before: Unrolled 3 times

```
1  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

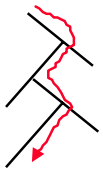
Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,-48(R1)	ST 0(R1),F4	ADDD F4,F0,F2			1
LD F6,-56(R1)	ST -8(R1),F8	ADDD F8,F6,F2		SUBI R1,R1,#24	2
LD F10,-40(R1)	ST 8(R1),F12	ADDD F12,F10,F2		BNEZ R1,LOOP	3

- 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 (*<u>trace</u>*) 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



- Compiler undoes bad guess (discards values in registers)
- Subtle compiler bugs mean wrong answer vs. pooer performance; no hardware interlocks

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

- HW determines address conflicts
- HW better branch prediction
- HW maintains precise exception model
- HW does not execute bookkeeping instructions
- Works across multiple implementations
- SW speculation is much easier for HW design

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
- 64 integer registers + 64 floating point registers
  - Not separate filesper funcitonal 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
- Merced is name of first implementation (1999/2000?)
- LIW = EPIC?

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.

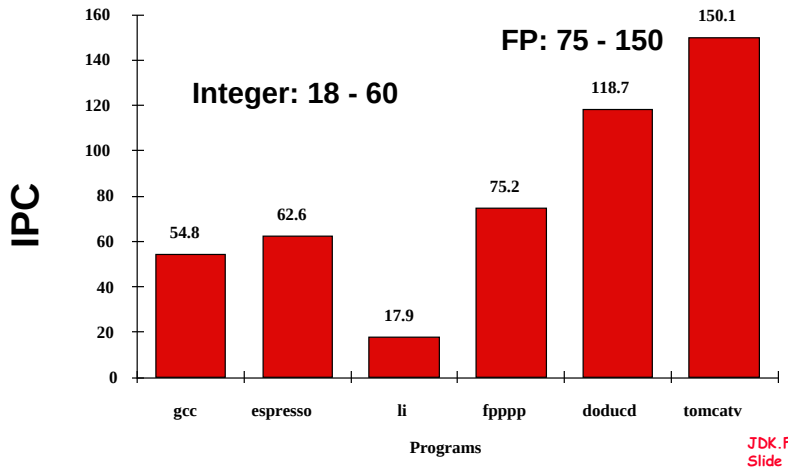
Assumptions for ideal/perfect machine to start:

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

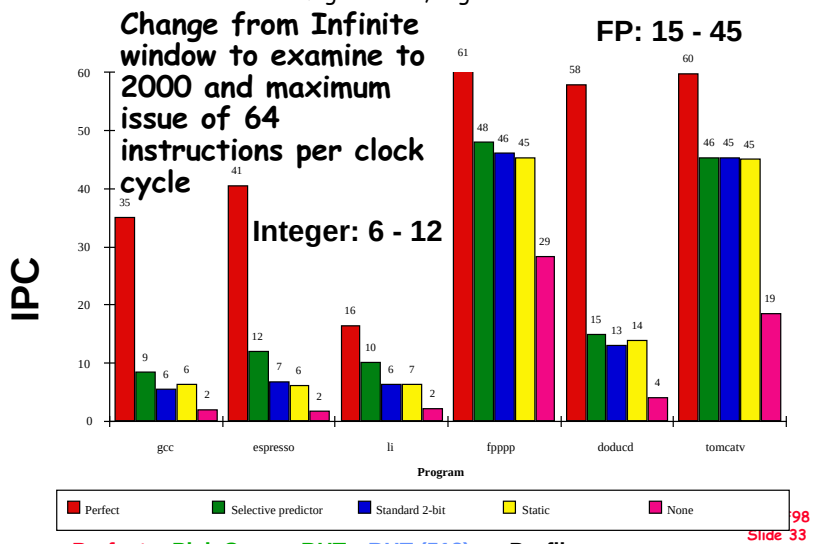
Upper Limit to ILP: Ideal Machine

(Figure 4.38, page 319)



More Realistic HW: Branch Impact

Figure 4.40, Page 323

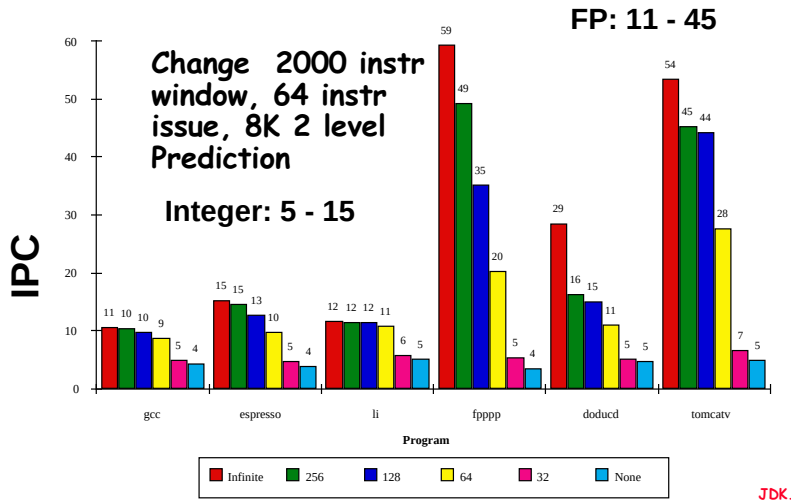


Perfect | Pick Cor. or BHT | BHT (512) | Profile | No prediction

JDK.F98 Slide 33

More Realistic HW: Register Impact

Figure 4.44, Page 328

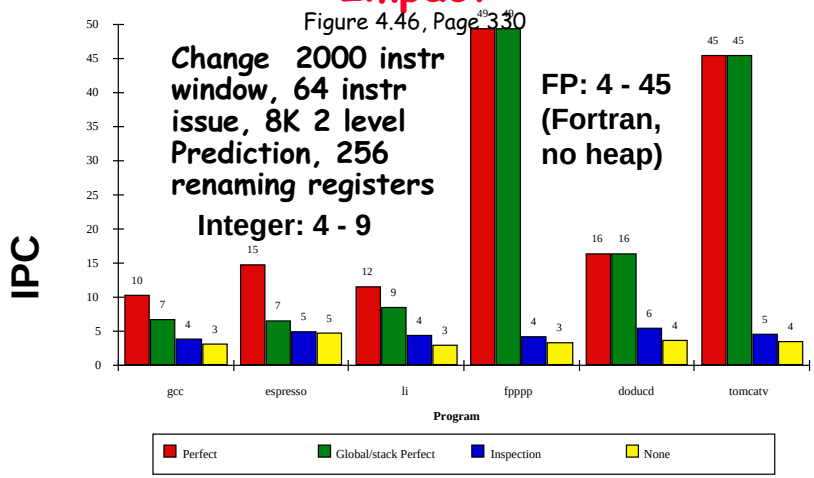


Infinite | 256 | 128 | 64 | 32 | None

JDK.F98 Slide 34

More Realistic HW: Alias Impact

Figure 4.46, Page 330

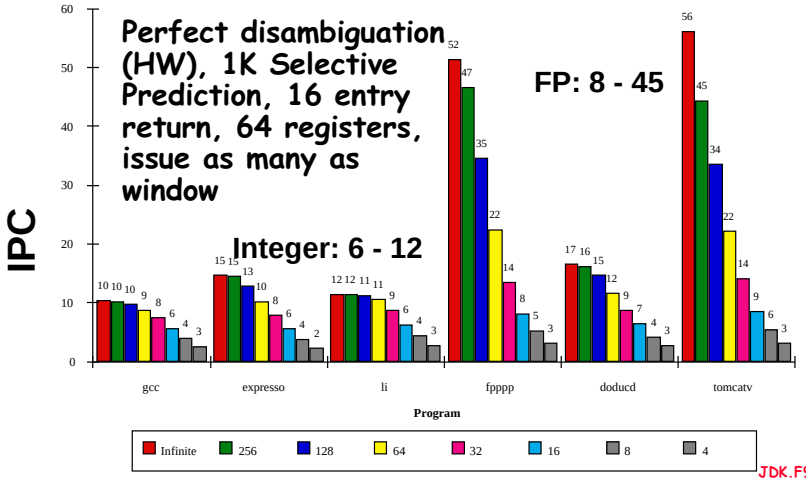


Perfect | Global/Stack perf; heap conflicts | Inspec. Assem. | None

JDK.F98 Slide 35

Realistic HW for '9X: Window Impact

(Figure 4.48, Page 332)



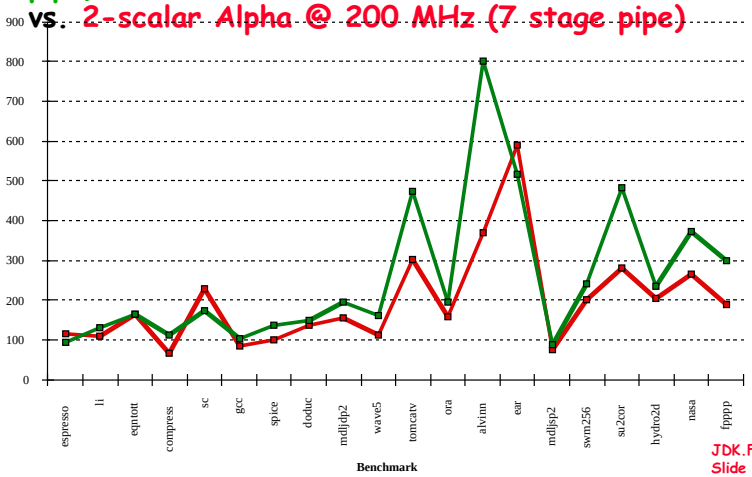
Infinite | 256 | 128 | 64 | 32 | 16 | 8 | 4

JDK.F98 Slide 36

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)



CS 252 Administrivia

- Exercises for Lectures 3 to 7
  - 4.2, 4.10, 4.19, 4.24, 4.14 parts c) and d) only, B.2
  - Due Wednesday Sept 23rd in class.
  - Extension available to people doing prelims
    - » Friday in class. Send mail to Aaron, cc me.
  - Done in pairs, but both need to understand whole assignment; Anyone need a partner?
  - Study groups encouraged, but pairs do own work
  - Turn in (copy of) photo with name on it (phonetic spelling, if useful)

Problems with scalar approach to ILP extraction

- Limits to conventional exploitation of ILP:

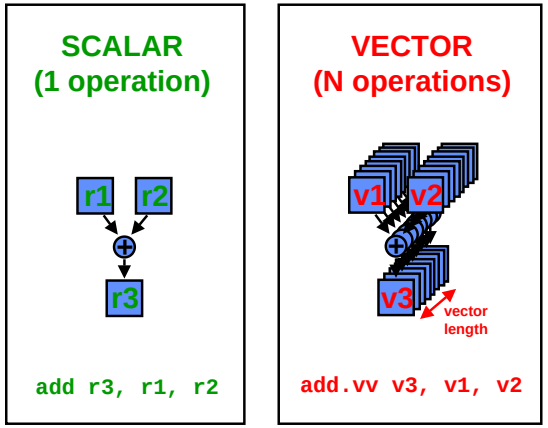
  1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)

  2) *instruction fetch and decode*: at some point, its hard to fetch and decode more instructions per clock cycle

  3) *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

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          ;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
```

```
      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
```

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)

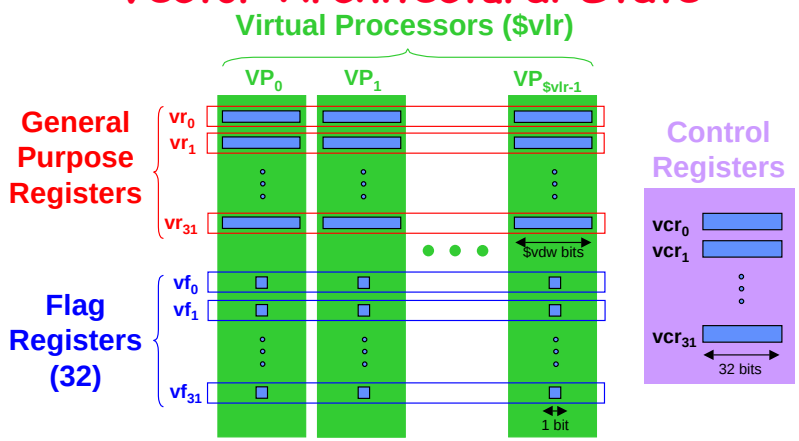
Vector Surprise

- Use vectors for inner loop parallelism (no surprise)
  - One dimension of array: A[0, 0], A[0, 1], A[0, 2], ...
  - think of machine as, say, 32 vector regs each with 64 elements
  - 1 instruction updates 64 elements of 1 vector register
- and for outer loop parallelism!
  - 1 element from each column: A[0,0], A[1,0], A[2,0], ...
  - think of machine as 64 "virtual processors" (VPs) each with 32 scalar registers! (- multithreaded processor)
  - 1 instruction updates 1 scalar register in 64 VPs
- Hardware identical, just 2 compiler perspectives

Virtial Processor Vector Model

- Vector operations are SIMD (single instruction multiple data)operations
- Each element is computed by a virtual processor (VP)
- Number of VPs given by vector length
  - vector control register

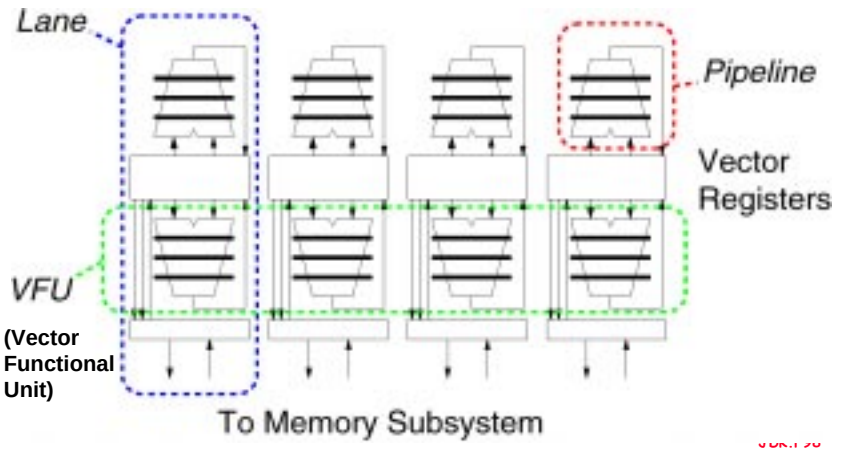
Vector Architectural State



Vector Implementation

- Vector register file
  - Each register is an array of elements
  - Size of each register determines maximum vector length
  - Vector length register determines vector length for a particular operation
- Multiple parallel execution units = "lanes" (sometimes called "pipelines" or "pipes")

Vector Terminology: 4 lanes, 2 vector functional units



Vector Execution Time

- Time = f(vector length, data dependicies, struct. hazards)
- *Initiation rate*: rate that FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- *Convoy*: set of vector instructions that can begin execution in same clock (no struct. or data hazards)
- *Chime*: approx. time for a vector operation
- *m* convoys take *m* chimes; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead; good approximization for long vectors)

```
1:  LV    V1,Rx     ;load vector X
2:  MULV  V2,F0,V1  ;vector-scalar mult.
    LV    V3,Ry     ;load vector Y
3:  ADDV  V4,V2,V3  ;add
4:  SV    Ry,V4     ;store the result
```

4 conveys, 1 lane, VL=64
=> 4 x 64 - 256 clocks
(or 4 clocks per result)

DLXV Start-up Time

- *Start-up time*: pipeline latency time (depth of FU pipeline); another sources of overhead
- Operation Start-up penalty (from CRAY-1)
- Vector load/store 12
- Vector multply 7
- Vector add 6

Assume convoys don't overlap; vector length = n:

Convoy	*Start*	*1st result*	*last result*	
1. LV	0	12	11+n (12+n-1)	
2. MULV, LV	12+n	12+n+12	23+2n	*Load start-up*
3. ADDV	24+2n	24+2n+6	29+3n	*Wait convoy 2*
4. SV	30+3n	30+3n+12	41+4n	*Wait convoy 3*

Why startup time for each vector instruction?

- Why not overlap startup time of back-to-back vector instructions?
- Cray machines built from many ECL chips operating at high clock rates; hard to do?
- Berkeley vector design ("TO") didn't know it wasn't supposed to do overlap, so no startup times for functional units (except load)

Vector Load/Store Units & Memories

- Start-up overheads usually longer fo LSUs
- Memory system must sustain (# lanes x word) /clock cycle
- Many Vector Procs. use banks (vs. simple interleaving):

  1) support multiple loads/stores per cycle
  => multiple banks & address banks independently

  2) support non-sequential accesses (see soon)
- Note: No. memory banks > memory latency to avoid stalls
  - *m* banks => *m* words per memory lantecy *l* clocks
  - if *m* < *l*, then gap in memory pipeline:

clock:	0	…	*l*	*l*+1	*l*+2	…	*l*+*m*- 1	*l*+*m*	…	2 *l*
word:	--	…	0	1	2	…	*m*-1	--	…	*m*

  - may have 1024 banks in SRAM

Vector Length

- What to do when vector length is not exactly 64?
- *vector-length register* (VLR) controls the length of any vector operation, including a vector load or store. (cannot be > the length of vector registers)

```
    do 10 i = 1, n
10  Y(i) = a * X(i) + Y(i)
```

- Don't know n until runtime!
  n > Max. Vector Length (MVL)?

Strip Mining

- Suppose Vector Length > Max. Vector Length (MVL)?
- *Strip mining*: generation of code such that each vector operation is done for a size S to the MVL
- 1st loop do short piece (n mod MVL), rest VL = MVL

```
    low = 1
    VL = (n mod MVL)  /*find the odd size piece*/
    do 1 j = 0,(n / MVL)  /*outer loop*/
      do 10 i = low,low+VL-1  /*runs for length VL*/
            Y(i) = a*X(i) + Y(i)  /*main operation*/
10  continue
    low = low+VL  /*start of next vector*/
    VL = MVL  /*reset the length to max*/
1   continue
```

Common Vector Metrics

- R_∞: MFLOPS rate on an infinite-length vector
  - vector "speed of light"
  - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  - (R_n is the MFLOPS rate for a vector of length n)
- $N_{1/2}$: The vector length needed to reach one-half of R_∞
  - a good measure of the impact of start-up
- N_V: The vector length needed to make vector mode faster than scalar mode
  - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

Vector Stride

- Suppose adjacent elements not sequential in memory

```
do 10 i = 1,100
    do 10 j = 1,100
        A(i,j) = 0.0
        do 10 k = 1,100
10          A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- Either B or C accesses not adjacent (800 bytes between)
- *stride*: distance separating elements that are to be merged into a single vector (caches do unit stride) => LVWS (load vector with stride) instruction
- Strides => can cause bank conflicts (e.g., stride = 32 and 16 banks)

Compiler Vectorization on Cray XMP

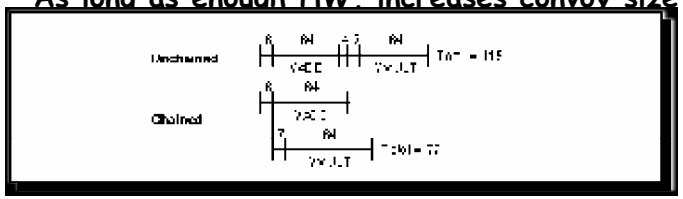
Benchmark	%FP	%FP in vector	
ADM	23%	68%	
DYFESM	26%	95%	
FLO52	41%	100%	
MDG	28%	27%	
MG3D	31%	86%	
OCEAN	28%	58%	
QCD	14%	1%	
SPICE	16%	7%	(1% overall)
TRACK	9%	23%	
TRFD	22%	10%	

Vector Opt #1: Chaining

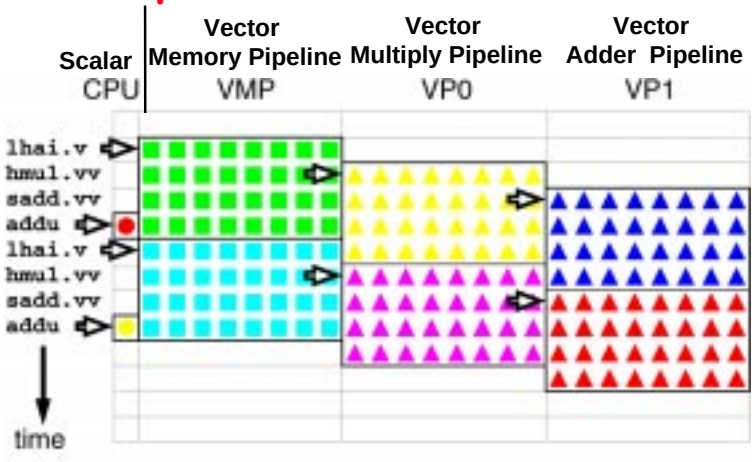
- Suppose:

```
MULV    V1,V2,V3
ADDV    V4,V1,V5    ; separate convoy?
```

- *chaining*: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector
- *Flexible chaining*: allow vector to chain to any other active vector operation => more read/write port
- As long as enough HW, increases convoy size



Example Execution of Vector Code



8 lanes, vector length 32, chaining

Vector Opt #2: Conditional Execution

- Suppose:

```
do 100 i = 1, 64
      if (A(i) .ne. 0) then
            A(i) = A(i) – B(i)
      endif
100 continue
```

- *vector-mask control* takes a Boolean vector: when *vector-mask register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.
- Still requires clock even if result not stored; if still performs operation, what about divide by 0?

Vector Opt #3: Sparse Matrices

- Suppose:

```
        do    100 i = 1,n
100     A(K(i)) = A(K(i)) + C(M(i))
```

- *gather* (LVI) operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector => a nonsparse vector in a vector register
- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store (SVI), using the same index vector
- Can't be done by compiler since can't know Ki elements distinct, no dependencies; by compiler directive
- Use CVI to create index 0, 1xm, 2xm, ..., 63xm

Sparse Matrix Example

- Cache (1993) vs. Vector (1988)

	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

- Cache: 1 address per cache block (32B to 64B)
- Vector: 1 address per element (4B)

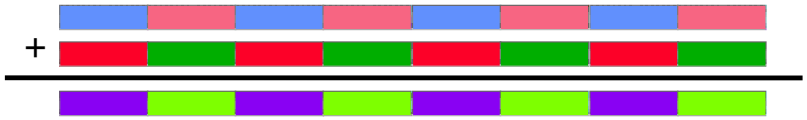
Applications

Limited to scientific computing?

- Multimedia Processing (compress., graphics, audio synth, image proc.)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems/Networking (memcpy, memset, parity, checksum)
- Databases (hash/join, data mining, image/video serving)
- Language run-time support (stdlib, garbage collection)
- even SPECint95

Vector for Multimedia?

- Intel MMX: 57 new 80x86 instructions (1st since 386)
  - similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC
- 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
  - reuse 8 FP registers (FP and MMX cannot mix)
- - short vector: load, add, store 8 8-bit operands



- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler

MMX Instructions

- Move 32b, 64b
- Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- Shifts (sll,srl, sra), And, And Not, Or, Xor in parallel: 8 8b, 4 16b, 2 32b
- Multiply, Multiply-Add in parallel: 4 16b
- Compare = , > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches
- Pack/Unpack
  - Convert 32b<-> 16b, 16b <-> 8b
  - Pack saturates (set to max) if number is too large

Vectors and Variable Data Width

- Programmer thinks in terms of vectors of data of some width (8, 16, 32, or 64 bits)
- Good for multimedia; More elegant than MMX-style extensions
- Don't have to worry about how data stored in hardware
  - No need for explicit pack/unpack operations
- Just think of more virtual processors operating on narrow data
- Expand Maximum Vector Length with decreasing data width:
  64 x 64bit, 128 x 32 bit, 256 x 16 bit, 512 x 8 bit

Mediaprocesing: Vectorizable? Vector Lengths?

Kernel	Vector length
Matrix transpose/multiply	# vertices at once
DCT (video, communication)	image width
FFT (audio)	256-1024
Motion estimation (video)	image width, iw/16
Gamma correction (video)	image width
Haar transform (media mining)	image width
Median filter (image processing)	image width
Separable convolution (img. proc.)	image width

(from Pradeep Dubey - IBM, http://www.research.ibm.com/people/p/pradeep/tutor.html)

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 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 as more HW resources available)
- Compact: Describe N operations with 1 short instruction (v. VLIW)
- Predictable (real-time) performance vs. statistical performance (cache)
- Multimedia ready: choose N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
- Mature, developed compiler technology
- Vector Disadvantage: Out of Fashion?

Summary #1

- Dynamic hardware schemes can unroll loops dynamically in hardware
- 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

Summary #2

- Vector model accomodates 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 harzards, fewer branches, fewer mispredicted branches, ...
- What % of computation is vectorizable?
- Is vector a good match to new apps such as multidemia, DSP?