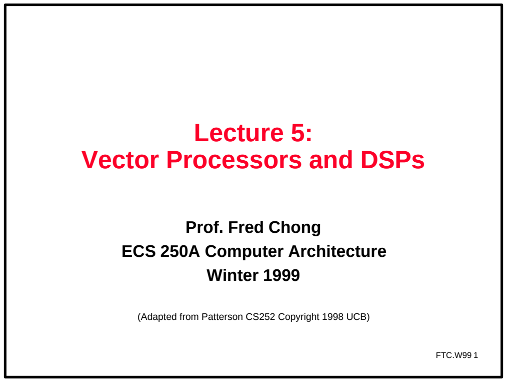
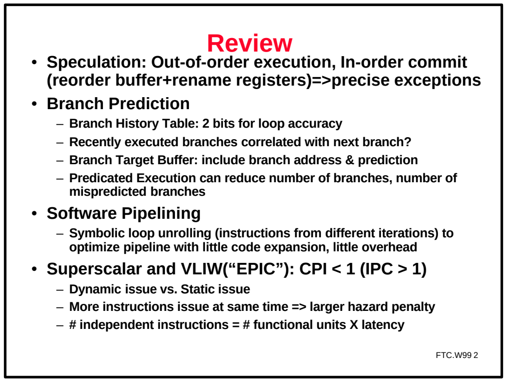


Lecture 5:
Vector Processors and DSPs

Prof. Fred Chong
ECS 250A Computer Architecture
Winter 1999

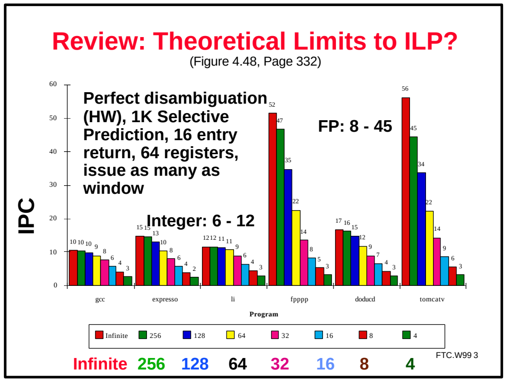


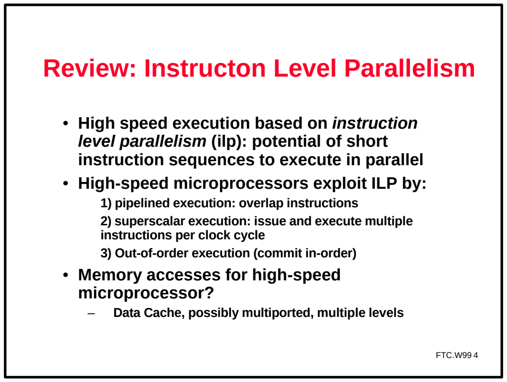
Review

- Speculation: Out-of-order execution, In-order commit (reorder buffer+rename registers)=>precise exceptions
- Branch Prediction
  - Branch History Table: 2 bits for loop accuracy
  - Recently executed branches correlated with next branch?
  - Branch Target Buffer: include branch address & prediction
  - Predicated Execution can reduce number of branches, number of mispredicted branches
- Software Pipelining
  - Symbolic loop unrolling (instructions from different iterations) to optimize pipeline with little code expansion, little overhead
- Superscalar and VLIW(“EPIC”): CPI < 1 (IPC > 1)
  - Dynamic issue vs. Static issue
  - More instructions issue at same time => larger hazard penalty
  - # independent instructions = # functional units X latency

Review: Theoretical Limits to ILP?

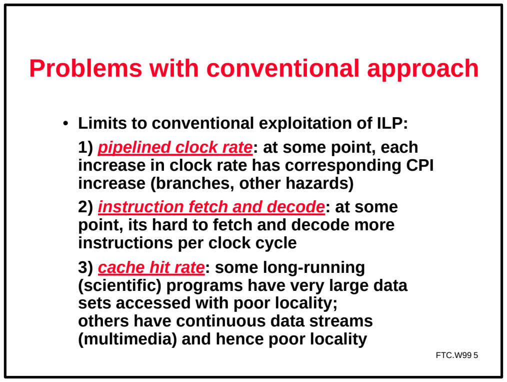
(Figure 4.48, Page 332)





Review: Instructon 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) superscalar execution: issue and execute multiple instructions per clock cycle
  - 3) Out-of-order execution (commit in-order)
- Memory accesses for high-speed microprocessor?
  - Data Cache, possibly multiported, multiple levels



Problems with conventional approach

- Limits to conventional exploitation of ILP:

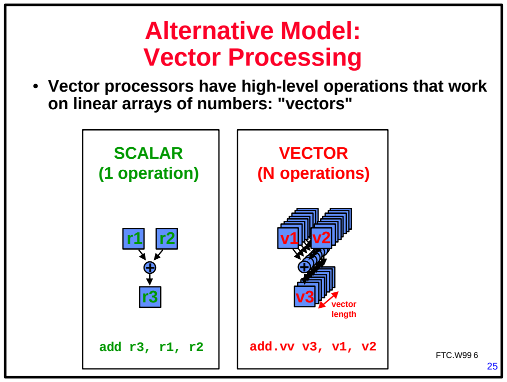
  1) *pipelined clock rate*: at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)

  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"



`add r3, r1, r2`

`add.vv v3, v1, v2`

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)

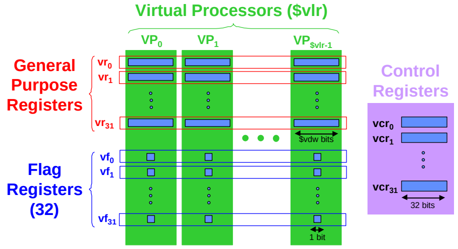
Vector Surprise

- 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

Virtual 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

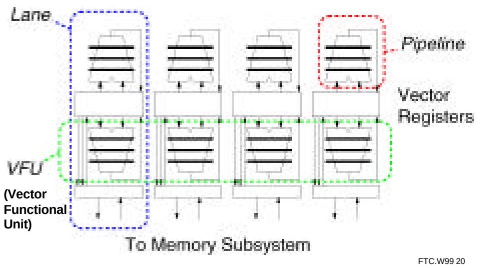
Vector Architectural State



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 ("T0") 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 Š 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_{inf}: MFLOPS rate on an infinite-length vector
  - upper bound
  - 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_{inf}
  - 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)
- Think of address per vector element

Compiler Vectorization on Cray XMP

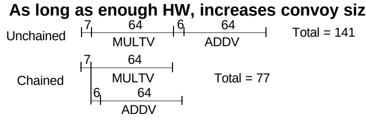
Benchmark	%FP	%FP in vector	
ADM	23%	68%	
DYFESM	26%	95%	
FLO52	41%	100%	
MDG	28%	27%	
MG3D	31%	86%	
OCEAN	28%	58%	
QCD	14%	1%	
SPICE	16%	7%	(1% overall)
TRACK	9%	23%	
TRFD	22%	10%	

Vector Opt #1: Chaining

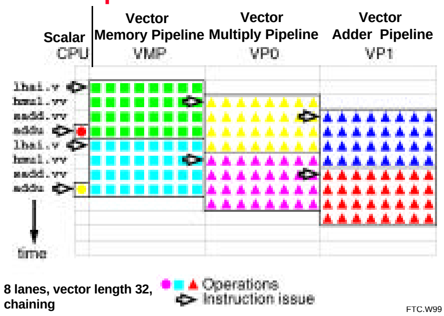
- Suppose:

  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)

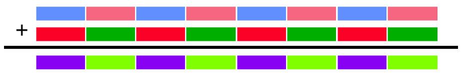
Applications

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

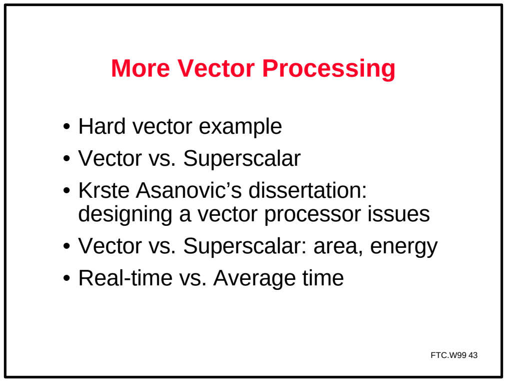
- Pitfall: Concentrating on peak performance and ignoring start-up overhead: N_V (length faster than scalar) > 100!
- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
  - failure of Cray competitor from his former company
- Pitfall: Good processor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

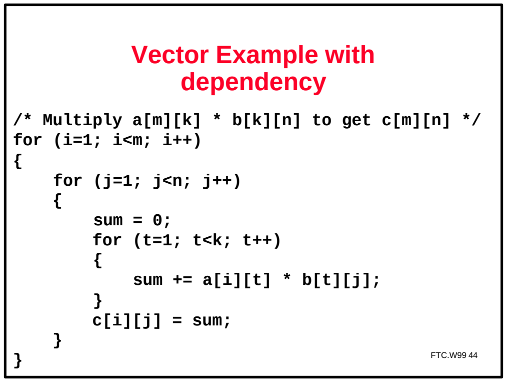
Vector Summary

- Alternate 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?



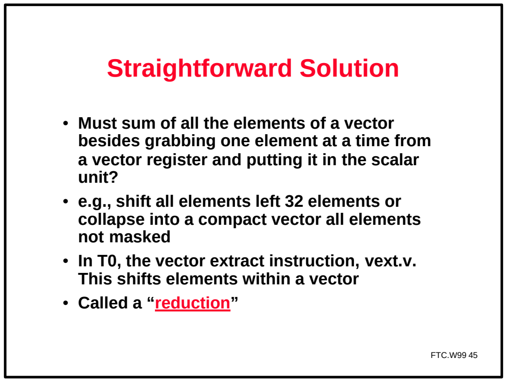
More Vector Processing

- Hard vector example
- Vector vs. Superscalar
- Krste Asanovic’s dissertation: designing a vector processor issues
- Vector vs. Superscalar: area, energy
- Real-time vs. Average time



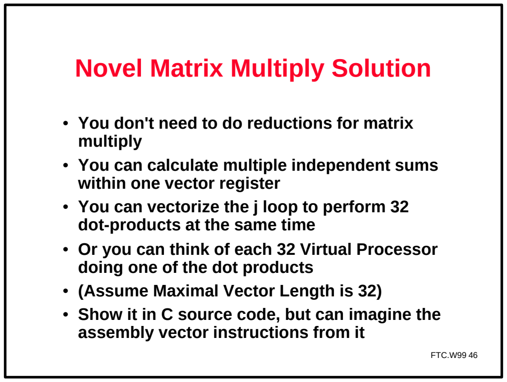
Vector Example with dependency

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```



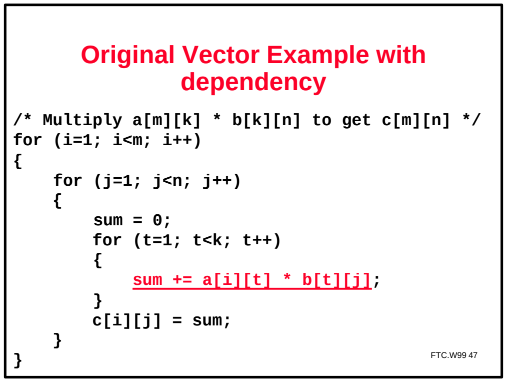
Straightforward Solution

- Must sum of all the elements of a vector besides grabbing one element at a time from a vector register and putting it in the scalar unit?
- e.g., shift all elements left 32 elements or collapse into a compact vector all elements not masked
- In T0, the vector extract instruction, vext.v. This shifts elements within a vector
- Called a “reduction”



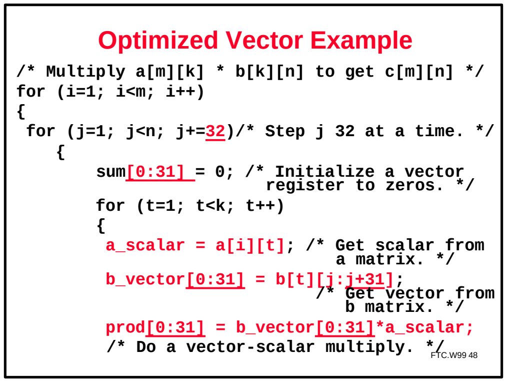
Novel Matrix Multiply Solution

- You don't need to do reductions for matrix multiply
- You can calculate multiple independent sums within one vector register
- You can vectorize the j loop to perform 32 dot-products at the same time
- Or you can think of each 32 Virtual Processor doing one of the dot products
- (Assume Maximal Vector Length is 32)
- Show it in C source code, but can imagine the assembly vector instructions from it



Original Vector Example with dependency

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```



Optimized Vector Example

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
 for (j=1; j<n; j+=32)/* Step j 32 at a time. */
    {
        sum[0:31] = 0; /* Initialize a vector
                          register to zeros. */
        for (t=1; t<k; t++)
        {
         a_scalar = a[i][t]; /* Get scalar from
                                a matrix. */
         b_vector[0:31] = b[t][j:j+31];
                              /* Get vector from
                                 b matrix. */
         prod[0:31] = b_vector[0:31]*a_scalar;
         /* Do a vector-scalar multiply. */
```

Optimized Vector Example cont’d

```
            /* Vector-vector add into results. */
            sum[0:31] += prod[0:31];
         }

         /* Unit-stride store of vector of
            results. */
         c[i][j:j+31] = sum[0:31];
     }
}
```

Novel, Step #2

- It's actually better to interchange the i and j loops, so that you only change vector length once during the whole matrix multiply
- To get the absolute fastest code you have to do a little register blocking of the innermost loop.

Designing a Vector Processor

- Changes to scalar
- How Pick Vector Length?
- How Pick Number of Vector Registers?
- Context switch overhead
- Exception handling
- Masking and Flag Instructions

Changes to scalar processor to run vector instructions

- Decode vector instructions
- Send scalar registers to vector unit (vector-scalar ops)
- Synchronization for results back from vector register, including exceptions
- Things that don’t run in vector don’t have high ILP, so can make scalar CPU simple

How Pick Vector Length?

- Vector length => Keep all VFUs busy:

vector length •

$$\frac{\text{(\# lanes) X (\# VFUs)}}{\text{\# Vector instructions/cycle}}$$

How Pick Vector Length?

- Longer good because:
  - 1) Hide vector startup
  - 2) lower instruction bandwidth
  - 3) tiled access to memory reduce scalar processor memory bandwidth needs
  - 4) if know max length of app. is < max vector length, no strip mining overhead
  - 5) Better spatial locality for memory access
- Longer not much help because:
  - 1) diminishing returns on overhead savings as keep doubling number of element
  - 2) need natural app. vector length to match physical register length, or no help

How Pick Number of Vector Registers?

- More Vector Registers:

1) Reduces vector register “spills” (save/restore)
  - 20% reduction to 16 registers for su2cor and tomcatv
  - 40% reduction to 32 registers for tomcatv
  - others 10%-15%

2) aggressive scheduling of vector instructinons: better compiling to take advantage of ILP

- Fewer:

Fewer bits in instruction format (usually 3 fields)

Context switch overhead

- Extra dirty bit per processor
  - If vector registers not written, don’t need to save on context switch
- Extra valid bit per vector register, cleared on process start
  - Don’t need to restore on context switch until needed

Exception handling: External

- If external exception, can just put pseudo-op into pipeline and wait for all vector ops to complete
  - Alternatively, can wait for scalar unit to complete and begin working on exception code assuming that vector unit will not cause exception and interrupt code does not use vector unit

Exception handling: Arithmetic

- Arithmetic traps harder
- Precise interrupts => large performance loss
- Alternative model: arithmetic exceptions set vector flag registers, 1 flag bit per element
- Software inserts trap barrier instructions from SW to check the flag bits as needed
- IEEE Floating Point requires 5 flag bits

Exception handling: Page Faults

- Page Faults must be precise
- Instruction Page Faults not a problem
- Data Page Faults harder
- Option 1: Save/restore internal vector unit state
  - Freeze pipeline, dump vector state
  - perform needed ops
  - Restore state and continue vector pipeline

Exception handling: Page Faults

- Option 2: expand memory pipeline to check addresses before send to memory + memory buffer between address check and registers
- multiple queues to transfer from memory buffer to registers; check last address in queues before load 1st element from buffer.
- Pre Address Iinstruction Queue (PAIQ) which sends to TLB and memory while in parallel go to Address Check Instruction Queue (ACIQ)
- When passes checks, instruction goes to Committed Instruction Queue (CIQ) to be there when data returns.
- On page fault, only save instructions in PAIQ and ACIQ

Masking and Flag Instructions

- Flag have multiple uses (conditional, arithmetic exceptions)
- Alternative is conditional move/merge
- Clear that fully masked is much more efficient that with conditional moves
  - Not perform extra instructions, avoid exceptions
- Downside is:

1) extra bits in instruction to specify the flag register

2) extra interlock early in the pipeline for RAW hazards on Flag registers

Flag Instruction Ops

- Do in scalar processor vs. in vector unit with vector ops?
- Disadvantages to using scalar processor to do flag calculations (as in Cray):

1) if MVL > word size => multiple instructions; also limits MVL in future

2) scalar exposes memory latency

3) vector produces flag bits 1/clock, but scalar consumes at 64 per clock, so cannot chain together

- Proposal: separate Vector Flag Functional Units and instructions in VU

Vectors Are Inexpensive

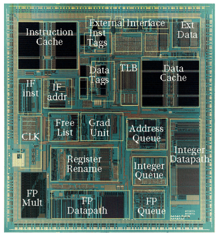
Scalar

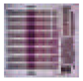
- N ops per cycle
  ⇒ $O(N^2)$ circuitry
- HP PA-8000
  - 4-way issue
  - reorder buffer: 850K transistors
    - incl. 6,720 5-bit register number comparators

Vector

- N ops per cycle
  ⇒ $O(N + \varepsilon N^2)$ circuitry
- T0 vector micro
  - 24 ops per cycle
  - 730K transistors total
    - only 23 5-bit register number comparators
  - No floating point

MIPS R10000 vs. T0





*See `http://www.icsi.berkeley.edu/real/spert/t0-intro.html`

Vectors Lower Power

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access in whole cache lines

Vector

- One instruction fetch,decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary

Superscalar Energy Efficiency Even Worse

Superscalar

- Control logic grows quadratically with issue width
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

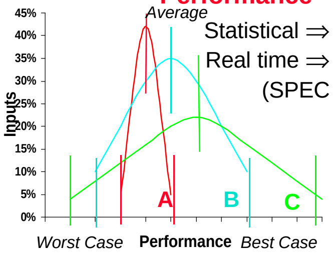
Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals

New Architecture Directions

- “…media processing will become the dominant force in computer arch. & microprocessor design.”
- “... new media-rich applications... involve significant real-time processing of continuous media streams, and make heavy use of vectors of packed 8-, 16-, and 32-bit integer and Fl. Pt.”
- Needs include high memory BW, high network BW, continuous media data types, real-time response, fine grain parallelism
  - “How Multimedia Workloads Will Change Processor Design”, Diefendorff & Dubey, *IEEE Computer* (9/97)

Which is Faster? Statistical v. Real time v. SPEC Performance

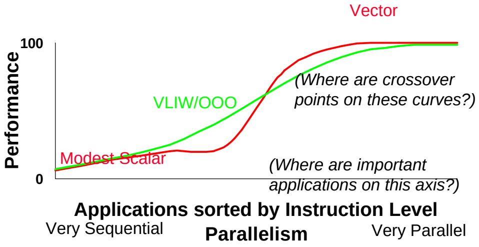


Statistical ⇒ Average ⇒ C

Real time ⇒ Worst ⇒ A

(SPEC ⇒ Best ⇒ C)

VLIW/Out-of-Order vs. Modest Scalar+Vector



(Where are crossover points on these curves?)

(Where are important applications on this axis?)

Cost-performance of simple vs. OOO

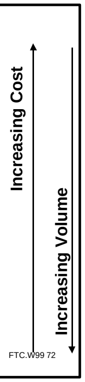
MIPS MPUs	R5000	R10000	10k/5k
• Clock Rate	200 MHz	195 MHz	1.0x
• On-Chip Caches	32K/32K	32K/32K	1.0x
• Instructions/Cycle	1(+ FP)	4	4.0x
• Pipe stages	5	5-7	1.2x
• Model	In-order	Out-of-order	---
• Die Size (mm^2)	84	298	3.5x
– without cache, TLB	32	205	6.3x
• Development (man yr.)	60	300	5.0x
• SPECint_base95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

Summary

- Vector is alternative model for exploiting ILP
- If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
- Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Will multimedia popularity revive vector architectures?

Review: Processor Classes

- General Purpose - high performance
  - Pentiums, Alpha's, SPARC
  - Used for general purpose software
  - Heavy weight OS - UNIX, NT
  - Workstations, PC's
- Embedded processors and processor cores
  - ARM, 486SX, Hitachi SH7000, NEC V800
  - Single program
  - Lightweight, often realtime OS
  - DSP support
  - Cellular phones, consumer electronics (e. g. CD players)
- Microcontrollers
  - Extremely cost sensitive
  - Small word size - 8 bit common
  - Highest volume processors by far
  - Automobiles, toasters, thermostats, ...



DSP Outline

- Intro
- Sampled Data Processing and Filters
- Evolution of DSP
- DSP vs. GP Processor

DSP Introduction

- Digital Signal Processing: application of mathematical operations to digitally represented signals
- Signals represented digitally as sequences of samples
- Digital signals obtained from physical signals via tranducers (e.g., microphones) and analog-to-digital converters (ADC)
- Digital signals converted back to physical signals via digital-to-analog converters (DAC)
- Digital Signal Processor (DSP): electronic system that processes digital signals

Common DSP algorithms and applications

- Applications – Instrumentation and measurement
  - Communications
  - Audio and video processing
  - Graphics, image enhancement, 3- D rendering
  - Navigation, radar, GPS
  - Control - robotics, machine vision, guidance
- Algorithms
  - Frequency domain filtering - FIR and IIR
  - Frequency- time transformations - FFT
  - Correlation

What Do DSPs Need to Do Well?

- Most DSP tasks require:
  - Repetitive numeric computations
  - Attention to numeric fidelity
  - High memory bandwidth, mostly via array accesses
  - Real-time processing
- DSPs must perform these tasks efficiently while minimizing:
  - Cost
  - Power
  - Memory use
  - Development time

Who Cares?

- DSP is a key enabling technology for many types of electronic products
- DSP-intensive tasks are the performance bottleneck in many computer applications today
- Computational demands of DSP-intensive tasks are increasing very rapidly
- In many embedded applications, general-purpose microprocessors are not competitive with DSP-oriented processors today
- 1997 market for DSP processors: $3 billion

A Tale of Two Cultures

- General Purpose Microprocessor traces roots back to Eckert, Mauchly, Von Neumann (ENIAC)
- DSP evolved from Analog Signal Processors, using analog hardware to transform phyical signals (classical electrical engineering)
- ASP to DSP because
  - DSP insensitive to environment (e.g., same response in snow or desert if it works at all)
  - DSP performance identical even with variations in components; 2 analog systems behavior varies even if built with same components with 1% variation
- Different history and different applications led to different terms, different metrics, some new inventions
- Increasing markets leading to cultural warfare

DSP vs. General Purpose MPU

- DSPs tend to be written for 1 program, not many programs.
  - Hence OSes are much simpler, there is no virtual memory or protection, ...
- DSPs sometimes run hard real-time apps
  - You must account for anything that could happen in a time slot
  - All possible interrupts or exceptions must be accounted for and their collective time be subtracted from the time interval.
  - Therefore, exceptions are BAD!
- DSPs have an infinite continuous data stream

Today's DSP "Killer Apps"

- In terms of dollar volume, the biggest markets for DSP processors today include:
  - Digital cellular telephony
  - Pagers and other wireless systems
  - Modems
  - Disk drive servo control
- Most demand good performance
- All demand low cost
- Many demand high energy efficiency
- Trends are towards better support for these (and similar) major applications.

Digital Signal Processing in General Purpose Microprocessors

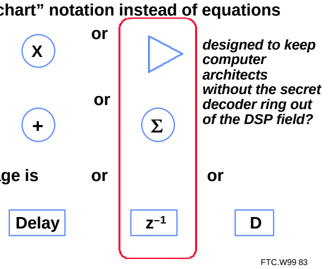
- Speech and audio compression
- Filtering
- Modulation and demodulation
- Error correction coding and decoding
- Servo control
- Audio processing (e.g., surround sound, noise reduction, equalization, sample rate conversion)
- Signaling (e.g., DTMF detection)
- Speech recognition
- Signal synthesis (e.g., music, speech synthesis)

Decoding DSP Lingo

- DSP culture has a graphical format to represent formulas.
- Like a flowchart for formulas, inner loops, not programs.
- Some seem natural: Σ is add, X is multiply
- Others are obtuse: z^{-1} means take variable from earlier iteration.
- These graphs are trivial to decode

Decoding DSP Lingo

- Uses "flowchart" notation instead of equations
- Multiply is or
- Add is or
- Delay/Storage is or or



designed to keep computer architects without the secret decoder ring out of the DSP field?

FIR Filtering: A Motivating Problem

- M most recent samples in the delay line (Xi)
- New sample moves data down delay line
- "Tap" is a multiply-add
- Each tap (M+1 taps total) nominally requires:
  - Two data fetches
  - Multiply
  - Accumulate
  - Memory write-back to update delay line
- Goal: 1 FIR Tap / DSP instruction cycle

DSP Assumptions of the World

- Machines issue/execute/complete in order
- Machines issue 1 instruction per clock
- Each line of assembly code = 1 instruction
- Clocks per Instruction = 1.000
- Floating Point is slow, expensive

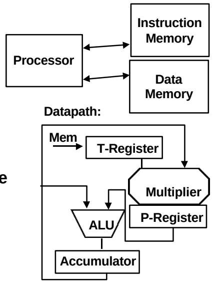
FIR filter on (simple) General Purpose Processor

```
loop:
   lw   x0, 0(r0)
   lw   y0, 0(r1)
   mul  a, x0,y0
   add  y0,a,b
   sw   y0,(r2)
   inc  r0
   inc  r1
   inc  r2
   dec ctr
   tst ctr
   jnz loop
```

- Problems: Bus / memory bandwidth bottleneck, control code overhead

First Generation DSP (1982): Texas Instruments TMS32010

- 16-bit fixed-point
- “Harvard architecture”
  - separate instruction, data memories
- Accumulator
- Specialized instruction set
  - Load and Accumulate
- 390 ns Multiple-Accumulate (MAC) time; 228 ns today



TMS32010 FIR Filter Code

- Here X4, H4, ... are direct (absolute) memory addresses:

```
LT X4   ; Load T with x(n-4)
MPY H4  ; P = H4*X4
LTD X3  ; Load T with x(n-3); x(n-4) = x(n-3);
        ; Acc = Acc + P
MPY H3  ; P = H3*X3
LTD X2
MPY H2
...
```

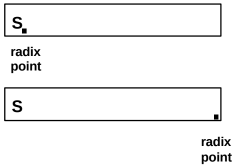
- Two instructions per tap, but requires unrolling

Features Common to Most DSP Processors

- Data path configured for DSP
- Specialized instruction set
- Multiple memory banks and buses
- Specialized addressing modes
- Specialized execution control
- Specialized peripherals for DSP

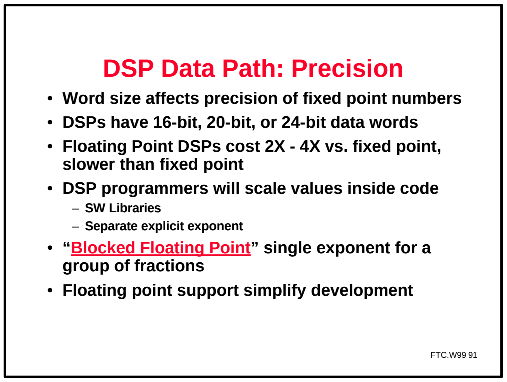
DSP Data Path: Arithmetic

- DSPs dealing with numbers representing real world => Want “reals”/ fractions
- DSPs dealing with numbers for addresses => Want integers
- Support “fixed point” as well as integers



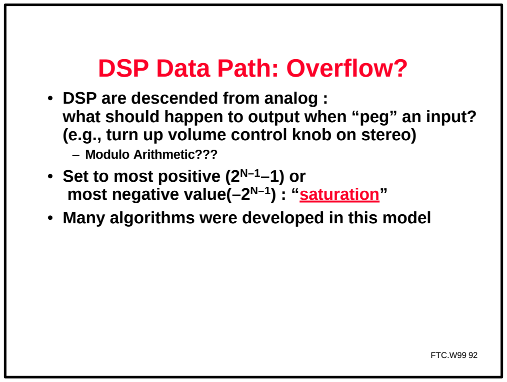
-1 Š x < 1

-2^{N-1} Š x < 2^{N-1}



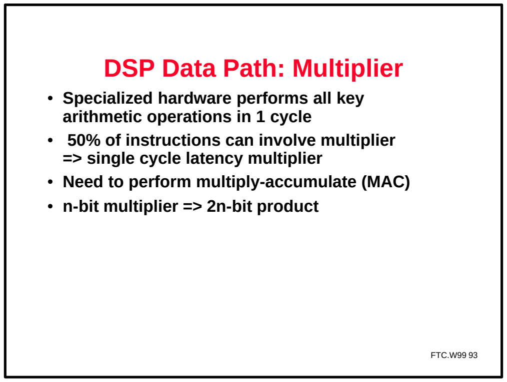
DSP Data Path: Precision

- Word size affects precision of fixed point numbers
- DSPs have 16-bit, 20-bit, or 24-bit data words
- Floating Point DSPs cost 2X - 4X vs. fixed point, slower than fixed point
- DSP programmers will scale values inside code
  - SW Libraries
  - Separate explicit exponent
- "Blocked Floating Point" single exponent for a group of fractions
- Floating point support simplify development



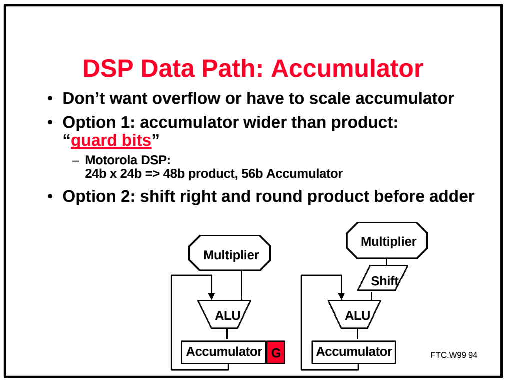
DSP Data Path: Overflow?

- DSP are descended from analog : what should happen to output when "peg" an input? (e.g., turn up volume control knob on stereo)
  - Modulo Arithmetic???
- Set to most positive (2^{N-1}–1) or most negative value(–2^{N-1}) : "saturation"
- Many algorithms were developed in this model



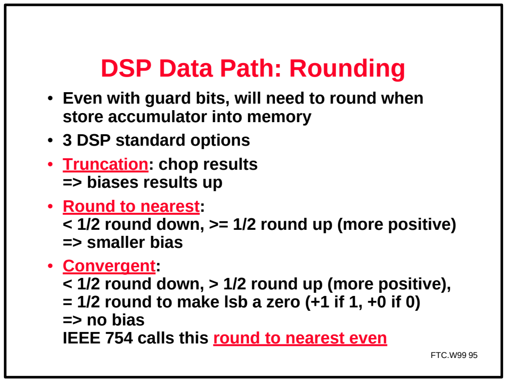
DSP Data Path: Multiplier

- Specialized hardware performs all key arithmetic operations in 1 cycle
- 50% of instructions can involve multiplier => single cycle latency multiplier
- Need to perform multiply-accumulate (MAC)
- n-bit multiplier => 2n-bit product



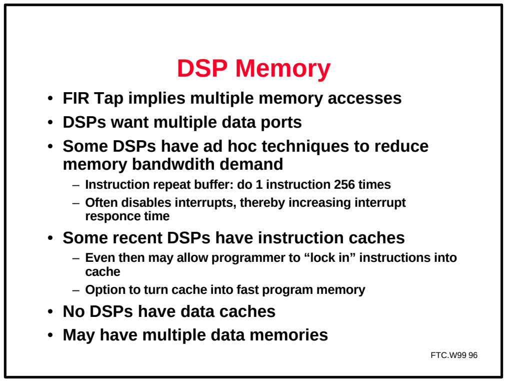
DSP Data Path: Accumulator

- Don't want overflow or have to scale accumulator
- Option 1: accumulator wider than product: "guard bits"
  - Motorola DSP: 24b x 24b => 48b product, 56b Accumulator
- Option 2: shift right and round product before adder



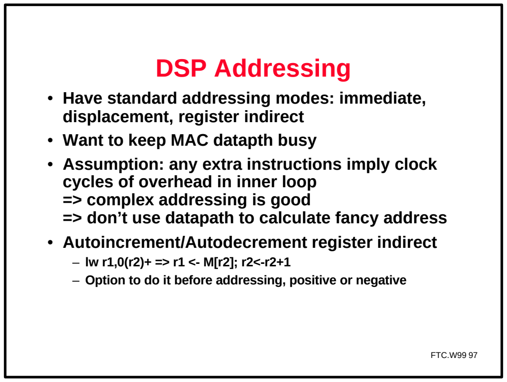
DSP Data Path: Rounding

- Even with guard bits, will need to round when store accumulator into memory
- 3 DSP standard options
- Truncation: chop results
  => biases results up
- Round to nearest:
  < 1/2 round down, >= 1/2 round up (more positive)
  => smaller bias
- Convergent:
  < 1/2 round down, > 1/2 round up (more positive),
  = 1/2 round to make lsb a zero (+1 if 1, +0 if 0)
  => no bias
  IEEE 754 calls this round to nearest even



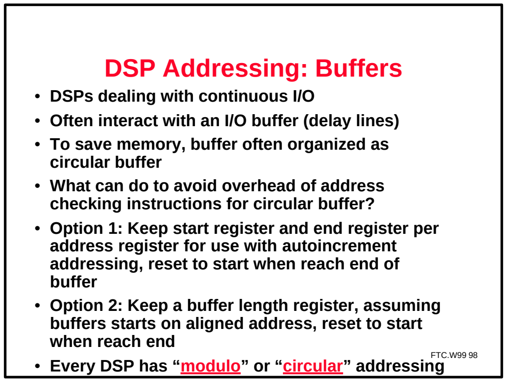
DSP Memory

- FIR Tap implies multiple memory accesses
- DSPs want multiple data ports
- Some DSPs have ad hoc techniques to reduce memory bandwdith demand
  - Instruction repeat buffer: do 1 instruction 256 times
  - Often disables interrupts, thereby increasing interrupt responce time
- Some recent DSPs have instruction caches
  - Even then may allow programmer to "lock in" instructions into cache
  - Option to turn cache into fast program memory
- No DSPs have data caches
- May have multiple data memories



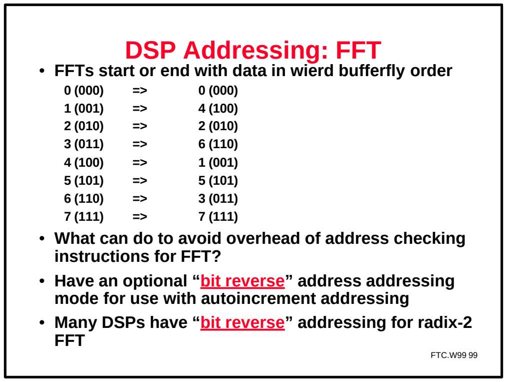
DSP Addressing

- Have standard addressing modes: immediate, displacement, register indirect
- Want to keep MAC datapth busy
- Assumption: any extra instructions imply clock cycles of overhead in inner loop
  => complex addressing is good
  => don't use datapath to calculate fancy address
- Autoincrement/Autodecrement register indirect
  - lw r1,0(r2)+ => r1 <- M[r2]; r2<-r2+1
  - Option to do it before addressing, positive or negative



DSP Addressing: Buffers

- DSPs dealing with continuous I/O
- Often interact with an I/O buffer (delay lines)
- To save memory, buffer often organized as circular buffer
- What can do to avoid overhead of address checking instructions for circular buffer?
- Option 1: Keep start register and end register per address register for use with autoincrement addressing, reset to start when reach end of buffer
- Option 2: Keep a buffer length register, assuming buffers starts on aligned address, reset to start when reach end
- Every DSP has "modulo" or "circular" addressing

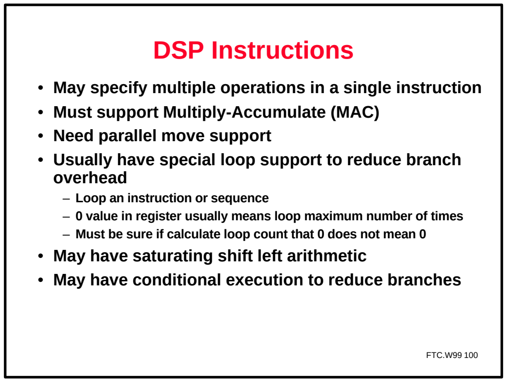


DSP Addressing: FFT

- FFTs start or end with data in wierd bufferfly order

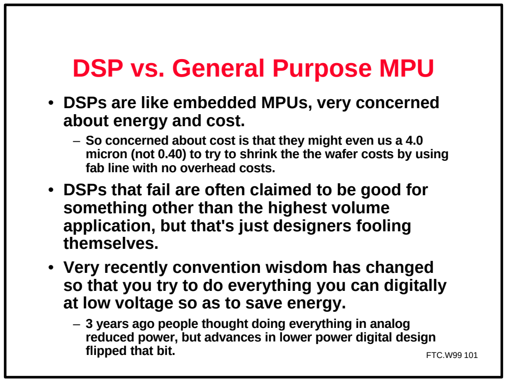
0 (000)	=>	0 (000)
1 (001)	=>	4 (100)
2 (010)	=>	2 (010)
3 (011)	=>	6 (110)
4 (100)	=>	1 (001)
5 (101)	=>	5 (101)
6 (110)	=>	3 (011)
7 (111)	=>	7 (111)

- What can do to avoid overhead of address checking instructions for FFT?
- Have an optional "bit reverse" address addressing mode for use with autoincrement addressing
- Many DSPs have "bit reverse" addressing for radix-2 FFT



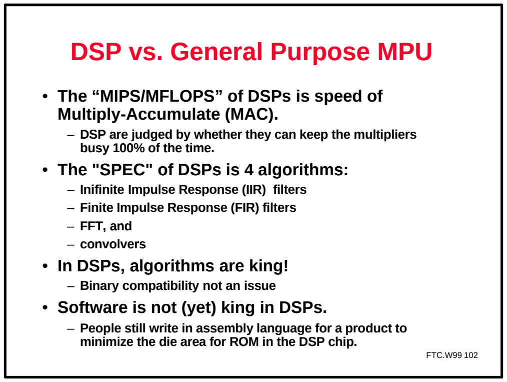
DSP Instructions

- May specify multiple operations in a single instruction
- Must support Multiply-Accumulate (MAC)
- Need parallel move support
- Usually have special loop support to reduce branch overhead
  - Loop an instruction or sequence
  - 0 value in register usually means loop maximum number of times
  - Must be sure if calculate loop count that 0 does not mean 0
- May have saturating shift left arithmetic
- May have conditional execution to reduce branches



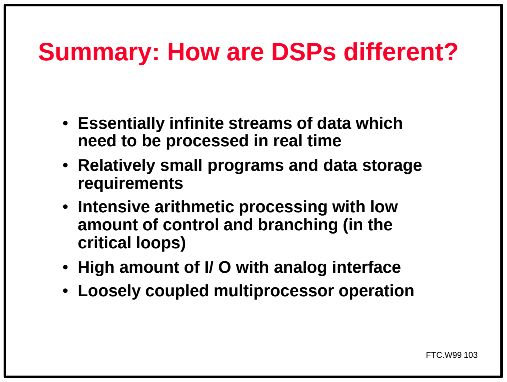
DSP vs. General Purpose MPU

- DSPs are like embedded MPUs, very concerned about energy and cost.
  - So concerned about cost is that they might even us a 4.0 micron (not 0.40) to try to shrink the the wafer costs by using fab line with no overhead costs.
- DSPs that fail are often claimed to be good for something other than the highest volume application, but that's just designers fooling themselves.
- Very recently convention wisdom has changed so that you try to do everything you can digitally at low voltage so as to save energy.
  - 3 years ago people thought doing everything in analog reduced power, but advances in lower power digital design flipped that bit.



DSP vs. General Purpose MPU

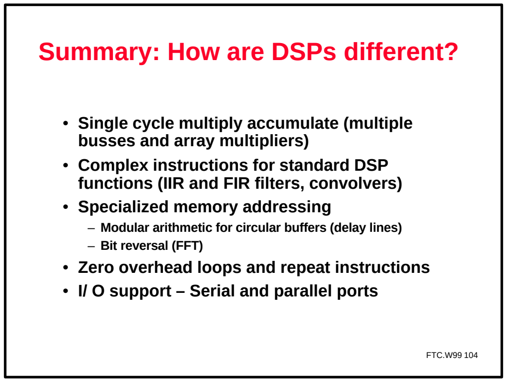
- The "MIPS/MFLOPS" of DSPs is speed of Multiply-Accumulate (MAC).
  - DSP are judged by whether they can keep the multipliers busy 100% of the time.
- The "SPEC" of DSPs is 4 algorithms:
  - Inifinite Impulse Response (IIR) filters
  - Finite Impulse Response (FIR) filters
  - FFT, and
  - convolvers
- In DSPs, algorithms are king!
  - Binary compatibility not an issue
- Software is not (yet) king in DSPs.
  - People still write in assembly language for a product to minimize the die area for ROM in the DSP chip.



Summary: How are DSPs different?

- Essentially infinite streams of data which need to be processed in real time
- Relatively small programs and data storage requirements
- Intensive arithmetic processing with low amount of control and branching (in the critical loops)
- High amount of I/ O with analog interface
- Loosely coupled multiprocessor operation

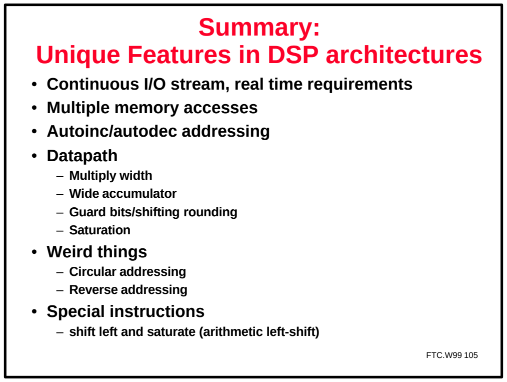
FTC.W99 103



Summary: How are DSPs different?

- Single cycle multiply accumulate (multiple busses and array multipliers)
- Complex instructions for standard DSP functions (IIR and FIR filters, convolvers)
- Specialized memory addressing
  - Modular arithmetic for circular buffers (delay lines)
  - Bit reversal (FFT)
- Zero overhead loops and repeat instructions
- I/ O support – Serial and parallel ports

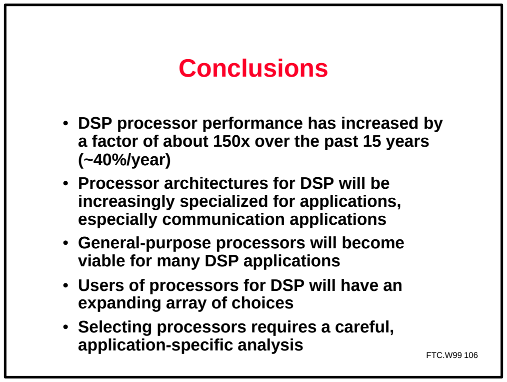
FTC.W99 104



Summary: Unique Features in DSP architectures

- Continuous I/O stream, real time requirements
- Multiple memory accesses
- Autoinc/autodec addressing
- Datapath
  - Multiply width
  - Wide accumulator
  - Guard bits/shifting rounding
  - Saturation
- Weird things
  - Circular addressing
  - Reverse addressing
- Special instructions
  - shift left and saturate (arithmetic left-shift)

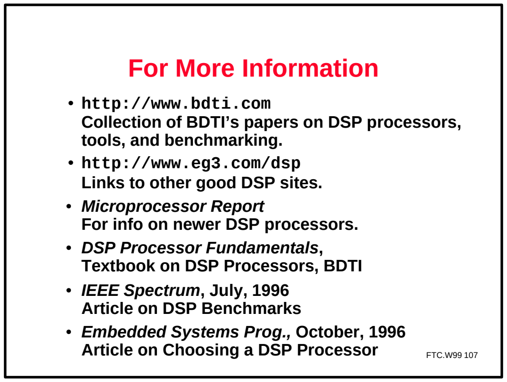
FTC.W99 105



Conclusions

- DSP processor performance has increased by a factor of about 150x over the past 15 years (~40%/year)
- Processor architectures for DSP will be increasingly specialized for applications, especially communication applications
- General-purpose processors will become viable for many DSP applications
- Users of processors for DSP will have an expanding array of choices
- Selecting processors requires a careful, application-specific analysis

FTC.W99 106



For More Information

- `http://www.bdti.com`
  Collection of BDTI's papers on DSP processors, tools, and benchmarking.
- `http://www.eg3.com/dsp`
  Links to other good DSP sites.
- *Microprocessor Report*
  For info on newer DSP processors.
- *DSP Processor Fundamentals*,
  Textbook on DSP Processors, BDTI
- *IEEE Spectrum*, July, 1996
  Article on DSP Benchmarks
- *Embedded Systems Prog.*, October, 1996
  Article on Choosing a DSP Processor

FTC.W99 107