Optimizing DSP Software for the Latest Processors

Berkeley Design Technology, Inc.
2107 Dwight Way, Second Floor
Berkeley, California U.S.A.

+1 (510) 665-1600
info@bdti.com
http://www.bdti.com

Optimization

Definition: A procedure used in the design of a system to maximize (or minimize) some performance index.

Possible performance indices:

- Execution speed
- Memory usage (code size and data size)
- Power consumption

◆ DSP applications require optimized software to be competitive

◆ Compilers typically don't generate sufficiently optimized software; the programmer often must hand-optimize inner loops in assembly language

Characteristics of Modern Processors

- Modern processors use increased parallelism to get high performance on DSP tasks
- Several different paths to achieve this goal:
  - Allowing many parallel operations to be encoded in each instruction
  - Issuing multiple instructions per cycle--superscalar, VLIW (very long instruction word) architectures
  - Adding SIMD (single instruction, multiple data) capabilities
- Given the current architectural landscape, what optimization techniques are effective?

BDTi

Optimization Considerations

- Optimizations often involve trade-offs between speed, memory usage, and power consumption
- The best optimization technique depends on the processor and the application
- Fortunately, there are some general techniques that apply to nearly any processor
- A key observation is that DSP applications spend most of their time in loops -- this is where optimization for speed and power is most valuable

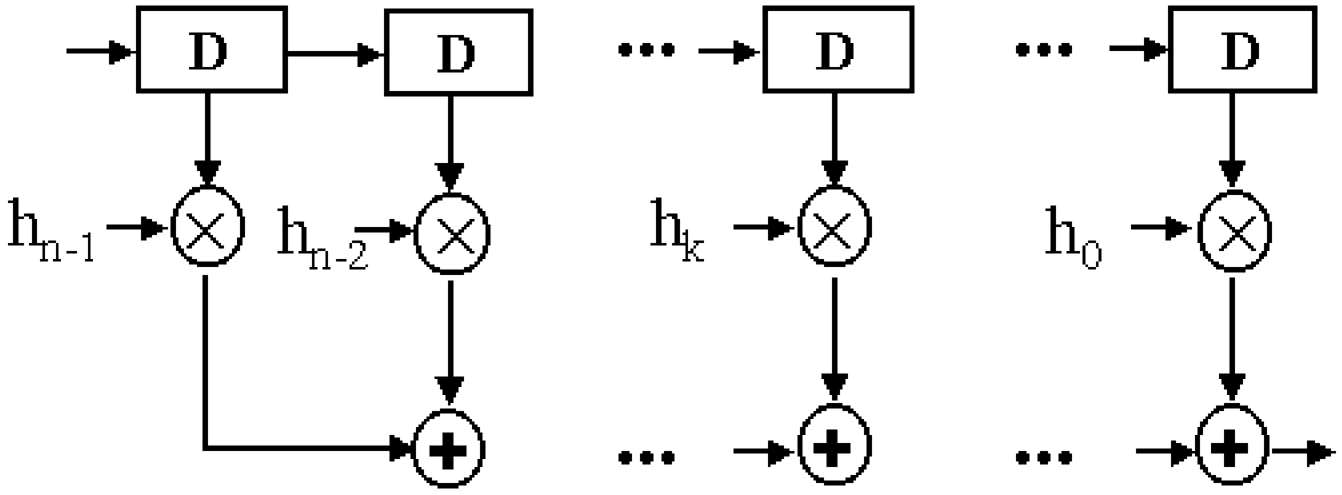
A Recipe for Loop Optimization

1. Identify the best approach to implementing the algorithm:

- Profile the loop to identify bottlenecks. For example, are bottlenecks caused by
  - a particular execution unit?
  - accesses to memory?
- Re-structure the algorithm to alleviate these bottlenecks ("algorithmic transformation")

2. Implement this approach efficiently using scheduling techniques

Profiling an FIR Filter on a DSP



Requirements:	Resources:
• multiply	• multiplier
• add	• ALU
• 2 loads	• 2 AGUs, 2 buses
• 1 store	

Three Categories of Algorithmic Transformations, With Examples

1. Unrolling across outer loops
2. Identifying operations that can be moved outside of a loop
3. Rearranging data in memory

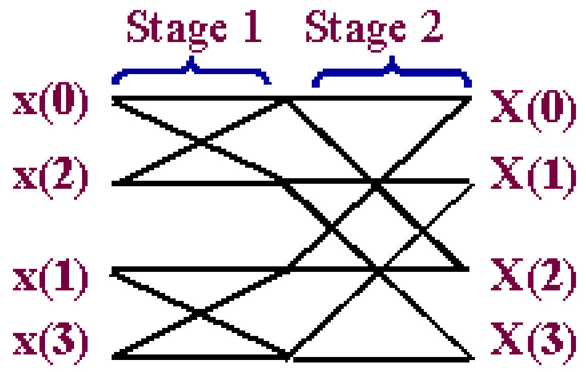
Examples we present here are not exhaustive, just illustrative of the concepts of each type of algorithmic transformation

1. Unrolling Across Outer Loops

- Useful in algorithms that use nested loops
- The goal: combine work from consecutive iterations of outer loop in inner loop
- Allows better re-use of intermediate results

BDTi

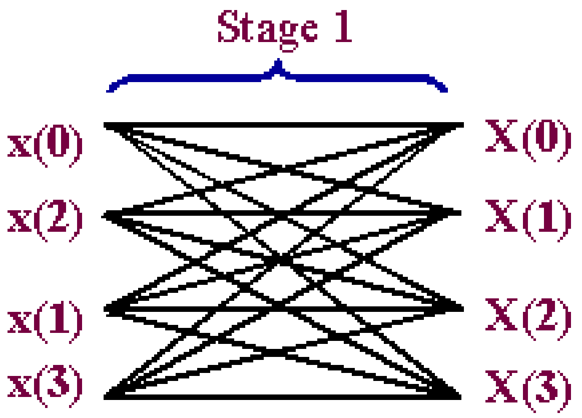
Radix-2 vs Radix-4 FFT Butterfly Structures



Radix-2:

Each butterfly requires:

- 8 Memory accesses
- 4 Multiplications
- 6 Additions

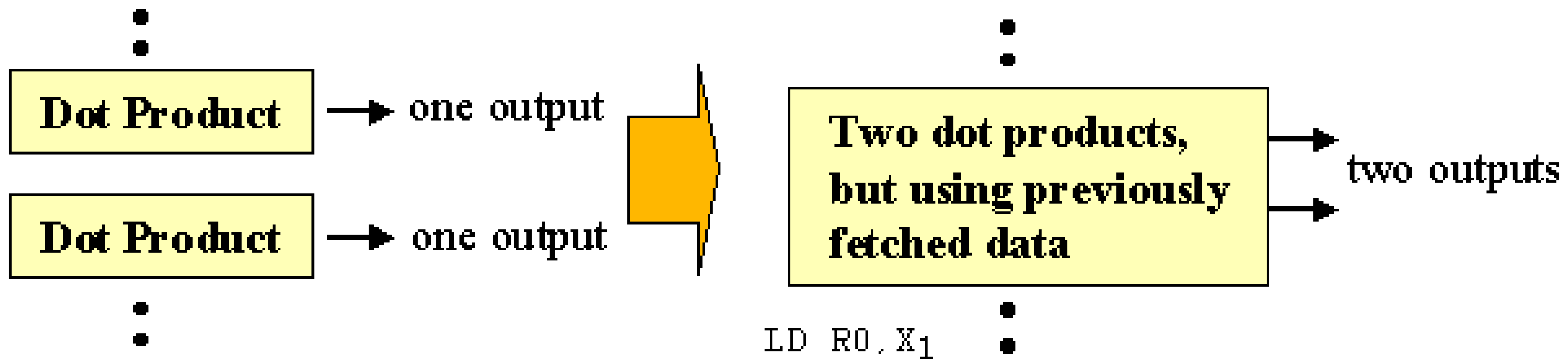


Radix-4:

Each butterfly requires:

- 16 Memory accesses (4 / R-2 bfly)
- 12 Multiplications (3 / R-2 bfly)
- 22 Additions (5.5 / R-2 bfly)

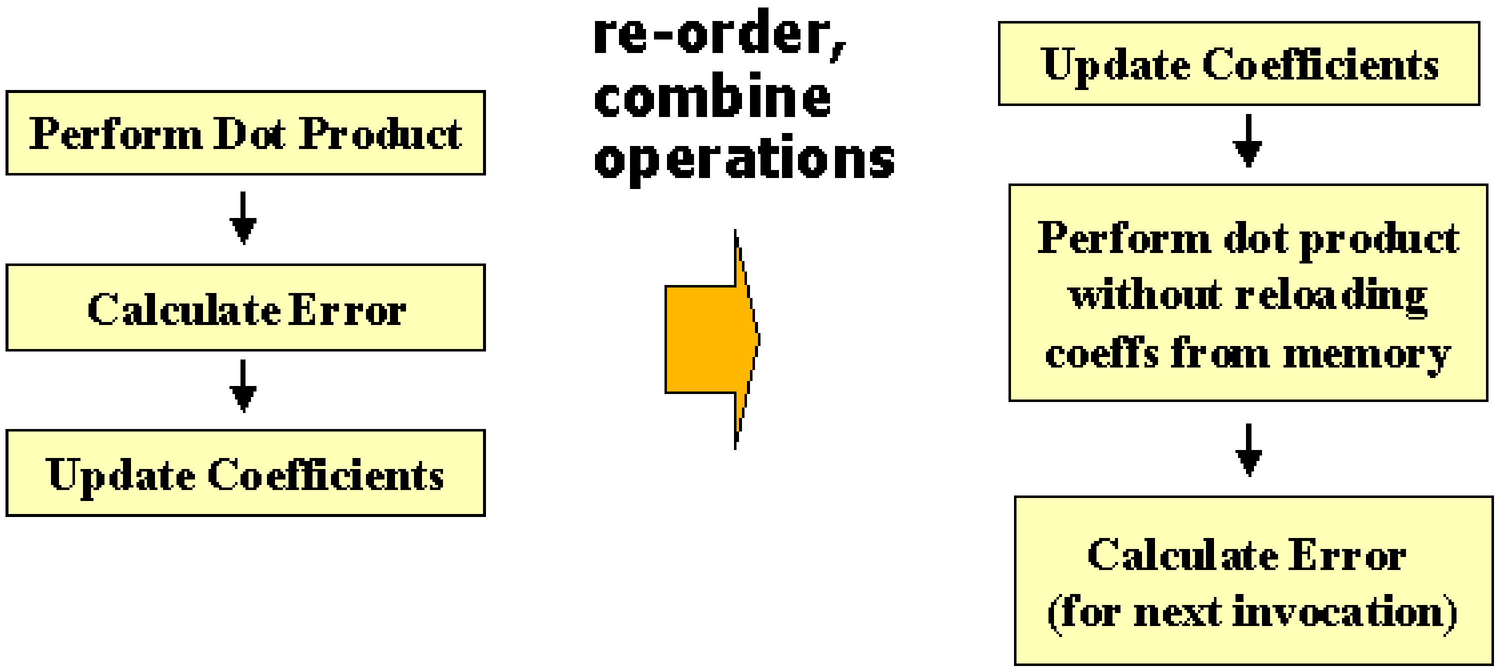
Block FIR Filter using "Zipping"



```
LD R0,X0
LD R1,C0
R2 = R0*R1, LD R0,X(-1)
LD R1,C1
R2 = R2+R0*R1, LD R0,X(-2)
LD R1,C2
R2 = R2+R0*R1, LD R0,X(-3)
LD R1,C3
R2 = R2+R0*R1
;y0 is in R2
```

```
LD R0,X1
LD R1,C0
R2 = R0*R1, LD R0,X0
R3 = R0*R1, LD R1,C1
R2 = R2+R0*R1, LD R0,X(-1)
R3 = R3+R0*R1, LD R1,C2
R2 = R2+R0*R1, LD R0,X(-2)
R3 = R3+R0*R1, LD R1,C3
R2 = R2+R0*R1, LD R0,X(-3)
R3 = R3+R0*R1
; y1 is in R2
; y0 is in R3
```

LMS Adaptive FIR Filter



Perform Dot Product
Calculate Error
Update Coefficients
re-order, combine operations
Update Coefficients
Perform dot product without reloading coeffs from memory
Calculate Error (for next invocation)

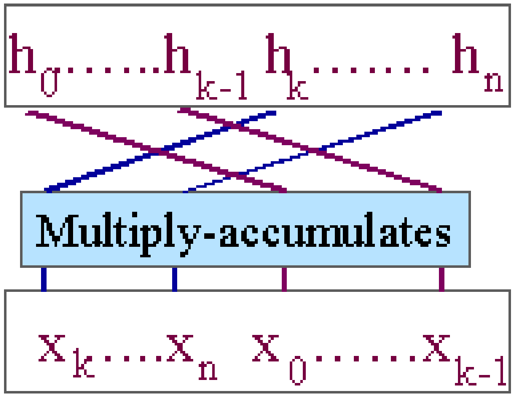
2. Moving Operations Outside Loop

- Goal: Use *a priori* knowledge of the algorithm to avoid repeated operations
- Identify calculations that produce constant results over the duration of the loop
  - Move those calculations outside the loop

BDTi

Circular Buffering for FIR Filter

- Implementing a circular buffer without support for modulo addressing. How to avoid checking for wraparound at each iteration?



$h_0 \ldots\ldots h_{k-1}\ h_k \ldots\ldots\ h_n$
Multiply-accumulates
$x_k \ldots x_n\ \ x_0 \ldots\ldots x_{k-1}$

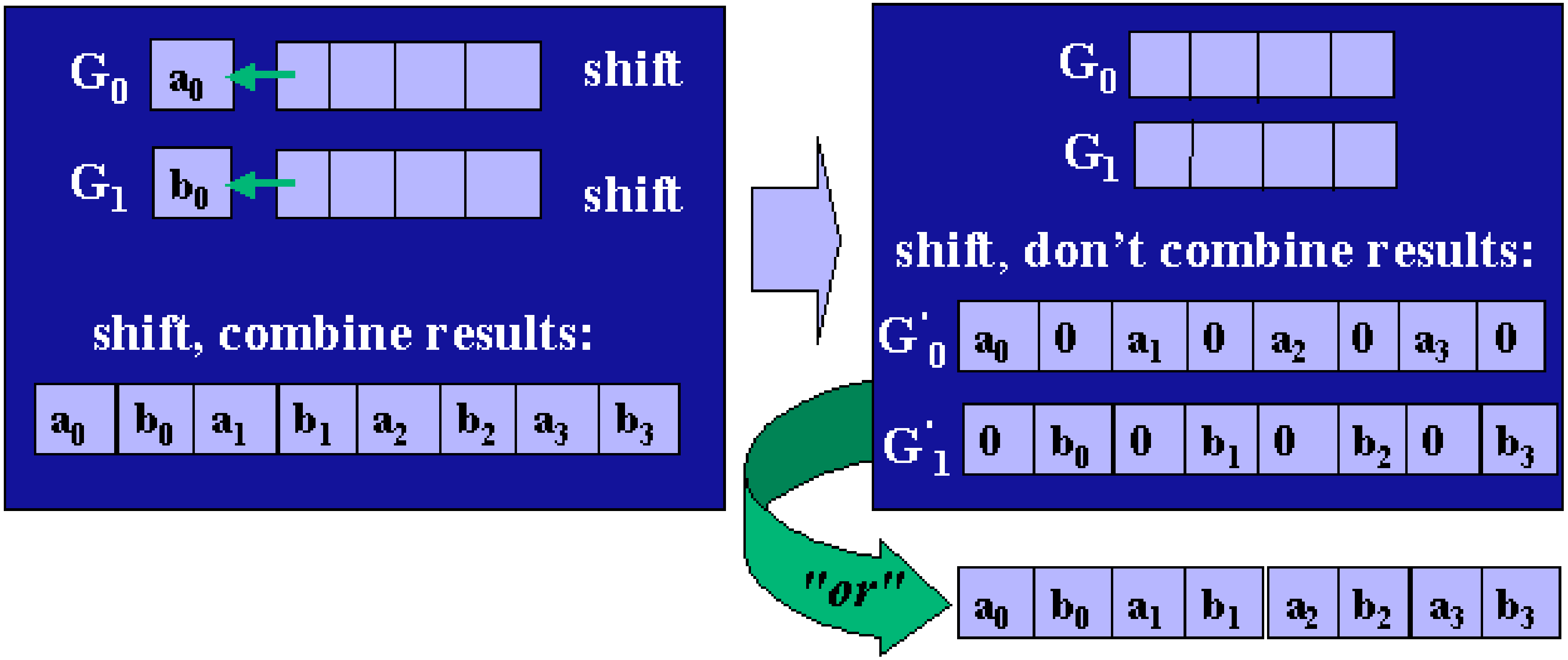
Pointer to Location of Last Sample

Loop 1 processes k samples
Loop 2 processes n-k samples

Find wraparound point outside of loop since it is constant over the duration of the loop

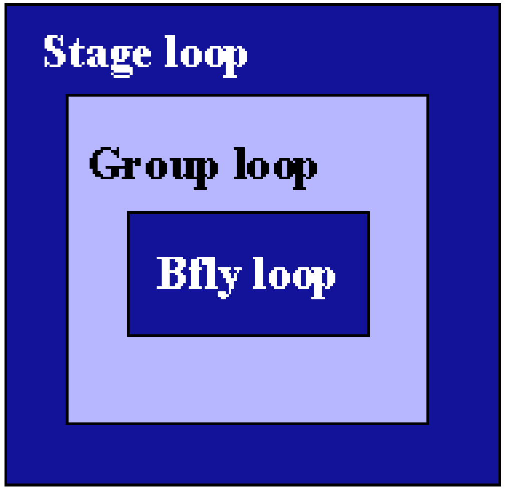
BDTi

Convolutional Encoder

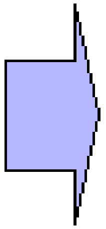


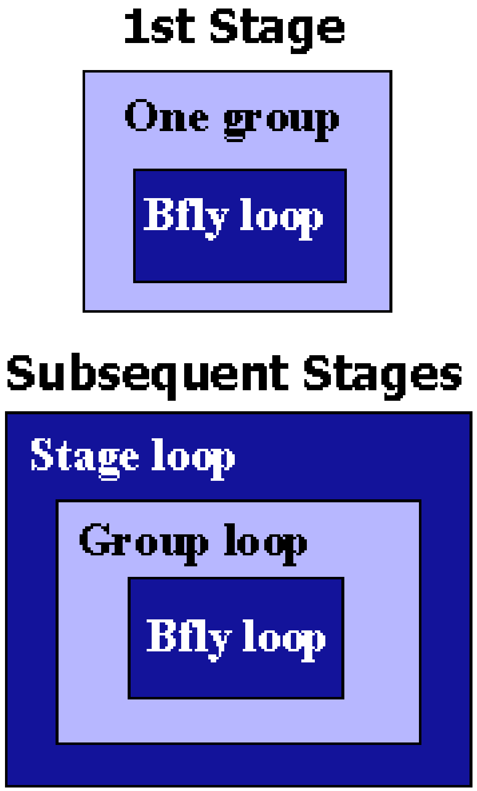
G_0 a_0 shift
G_1 b_0 shift
shift, combine results:
a_0 b_0 a_1 b_1 a_2 b_2 a_3 b_3
G_0
G_1
shift, don't combine results:
G'_0 a_0 0 a_1 0 a_2 0 a_3 0
G'_1 0 b_0 0 b_1 0 b_2 0 b_3
"or"
a_0 b_0 a_1 b_1 a_2 b_2 a_3 b_3

Radix-2 FFT



Twiddle factors for 1st stage are 1 and 0; can eliminate multiplications in 1st stage.



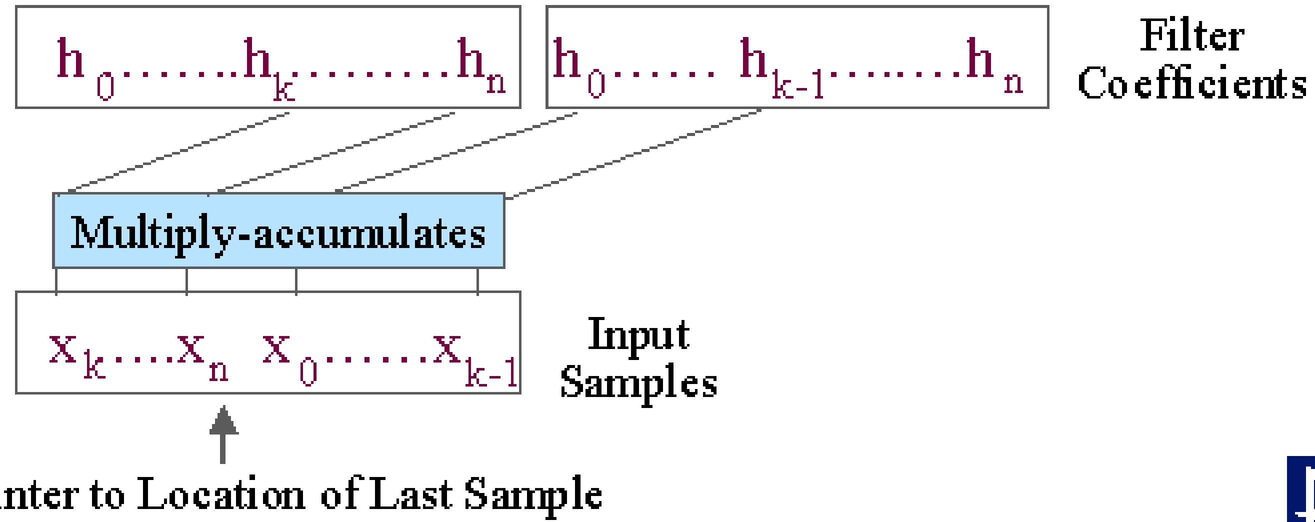


3. Arranging Data in Memory

- Goals:
  - Simplify addressing to save cycles on address calculations
  - Enable use of SIMD or other parallel operations
  - Reduce number of repeated loads

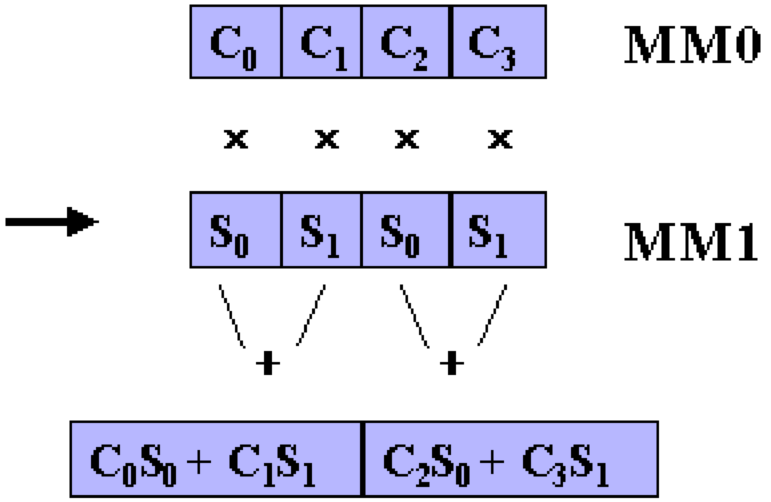
Circular Buffering for FIR Filter

- How to avoid checking for wraparound at every iteration of the inner loop?
- Maintain two copies of filter coefficients in memory


$h_0 \ldots h_k \ldots h_n$
$h_0 \ldots h_{k-1} \ldots h_n$
Filter Coefficients
Multiply-accumulates
$x_k \ldots x_n \; x_0 \ldots x_{k-1}$
Input Samples
Pointer to Location of Last Sample

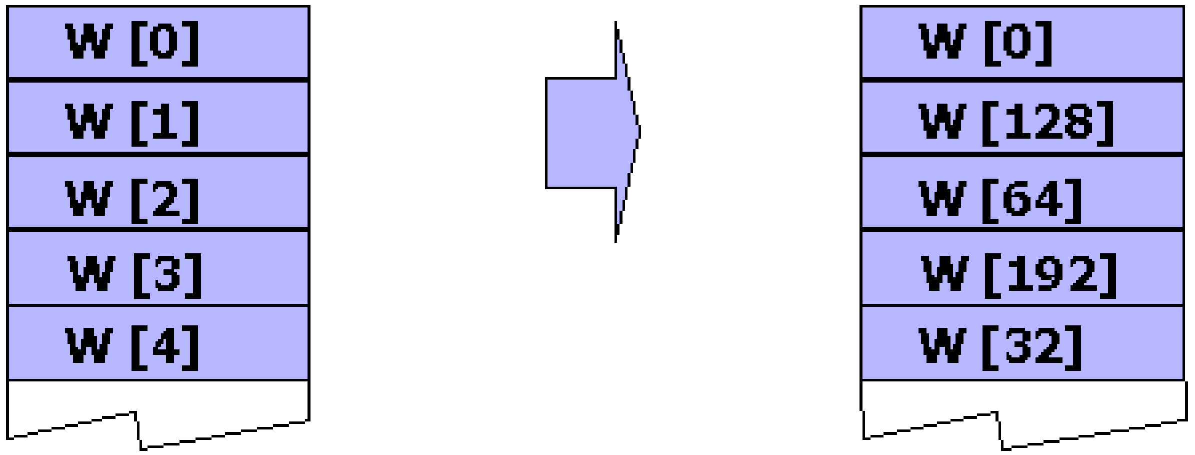
IIR Filter Biquad Section

Store two copies of filter state variables



Radix-2 FFT

Arrange twiddle factors
in bit-reversed order



Scheduling Techniques

Now that you've found the best general approach for the algorithm, you need to create an efficient implementation.

The programmer or compiler needs to schedule operations so that the program can take full advantage of the processor's parallelism. How?

- Software pipelining
- Loop unrolling

Software Pipelining

What is software pipelining?

- Execution of operations from different iterations of the (non-software-pipelined) loop in parallel
  - In each loop iteration, use intermediate results generated by the previous iteration and perform operations whose intermediate results will be used in the next iteration
- The deeper the hardware pipeline, the more likely it is that software pipelining will be necessary

FIR Filter on 'C62xx

No SW Pipelining:
6.5 Cycles/Tap
11 instructions

```
LOOP:
   LDW  .D2 *B4++,B2      ; load coef(0) & coef(1)
|| LDW  .D1 *A7--,A2      ; load state(0) & state(1)
   NOP  4
   MPYHL .M1X A2,B2,A3    ; P0(i)=coef(2i)*state(2i)
|| MPYLH .M2X A2,B2,B7    ; P1(i) =coef(2i+1)*state(2i+1)
    NOP 1
    ADD .L1 A0,A3,A0       ; Sum0(i) += P0(i-2)
|| ADD .L2 B1,B7,B1       ; Sum1(i) += P1(i-2)
|| ADD .S2 -1,B0,B0       ; Dec loop counter
||[B0] B .S1 LOOP          ; Cond. Branch to LOOP
   NOP  5

; Loop ends here
```

BDTi

FIR Filter on 'C62xx

With SW Pipelining:
0.5 Cycles/Tap
36 instructions

```
[Not shown: 24 instructions to
   prime pipeline, set up registers before loop start]

LOOP:
  ADD .L1 A0,A3,A0         ; Sum0(i) += P0(i-2)
||ADD .L2 B1,B7,B1         ; Sum1(i) += P1(i-2)
||MPYHL .M1X A2,B2,A3      ; P0(i) = coef(2i)*state(2i)
||MPYLH .M2X A2,B2,B7      ; P1(i) = coef(2i+1)*state(2i+1)
||LDW .D2 *B4++,B2         ; load coef(2i+10) & coef(2i+11)
||LDW .D1 *A7--,A2         ; load state(2i+10) & state(2i+11)
||[B0] ADD .S2 -1,B0,B0    ; Cond. dec loop counter
||[B0] B .S1 LOOP          ; Cond. Branch to LOOP
; LOOP ends here

  [Not shown: 3 instructions for final calculations]
```

FFT Butterfly on DSP16000

Software Pipelined Loop

```
j=4
do cloop {
a4=a0+p0-p1                                 *r0++j=a4_5h
a2=a0-p0+p1    p0=xh*yl    p1=xl*yh        *r1++j=a2_3h
a5=a1+p0+p1                                  y=*r1--
a3=a1-p0-p1    p0=xh*yh    p1=xl*yl          a0_1h=*r0--
 }
```

Loop Unrolling

- Repetition of loop-body instructions several times within a single loop iteration
- Main advantages:
  - Reduces relative loop overhead
  - May facilitate software pipelining by enabling operations from different loop iterations to execute in parallel
- Main disadvantages:
  - Increased memory usage
  - Loss of generality

FIR Filter on MMX Pentium

No Unrolling, no SW pipelining 1.75 Cycles/Tap

```
loop1:
   movq mm0, [esi]    ; load four samples
   pmaddwd    mm0, COEFaddr[edi]   ; 4 multiplies, 2 adds

/* two cycle stall happens here */

  paddd mm7, mm0      ; accumulate intermed results
  add  edi, 8         ; update coefficient index
  add  esi, 8         ; update delay line pointer
  dec  ecx            ; decrement loop count
  jnz  loop1
```

FIR on MMX Pentium

With Unrolling & SW Pipelining: 0.625 Cycles/Tap

```
loop1:
  pmaddwd mm0, COEFaddr[edi]       ; 4 multiplies, 2 adds
  paddd    mm7, mm2                ; accumulate intermediate results
  pmaddwd mm1, COEFaddr[edi+8] ; 4 multiplies, 2 adds
  paddd    mm7, mm3                ; accumulate intermediate results
  movq     mm2, [esi+16]         ; load four new samples
  movq     mm3, [esi+24]         ; load four new samples
  paddd    mm7, mm0                ; accumulate intermediate results
  pmaddwd mm2, COEFaddr[edi+16] ; 4 multiplies, 2 adds
  paddd    mm7, mm1                 ; accumulate intermediate result
  pmaddwd mm3, COEFaddr[edi+24] ; 4 multiplies, 2 adds
  movq     mm0, [esi+32]         ; load four new samples
  movq    mm1, [esi+40]         ; load four new samples
  add      edi, 32                 ; update coefficient index
  add      esi, 32                 ; update delay line pointer
  dec      ecx                     ; decrement loop count
jnz loop1
```

Conclusions

- As architectures diversify and become more complicated, optimization gets harder
- Since compilers often do not generate sufficiently optimized code, it is incumbent upon programmers to optimize critical code by hand, usually in assembly
- Optimization requires strong knowledge of both the processor and the algorithm
- Be aware of trade-offs between speed, memory usage, and power consumption

BDTi

For More Information...

- These slides will be available at BDTI's web site:

  http://www.bdti.com

- *DSP Processor Fundamentals* (BDTI, 1996), a textbook on DSP processors