DSP Software Optimization Techniques 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

Why Optimize?

- DSP applications are extremely computationally demanding, but often require low power, low memory use
- Optimization yields a competitive advantage
  - **Execution speed**
    - Can use a slower, and less expensive, processor
    - Allows upgraded functionality using the same processor
    - More functionality, e.g., more channels
  - **Memory usage**
    - Memory usage contributes to system cost
  - **Power consumption**
    - Improved battery life or reduced power supply size

The Light and Dark Side of Optimization

- Speed-optimized code
  - May consume less energy
  - But often sacrifices memory
- Memory-optimized code
  - May reduce memory accesses and hence energy consumption
  - But generally slows down code
- Power-optimized code
  - Can mean optimizing for speed
  - But may just as well slow down code
- Performance advantage vs. development time

Optimization for Modern Processors

- 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?

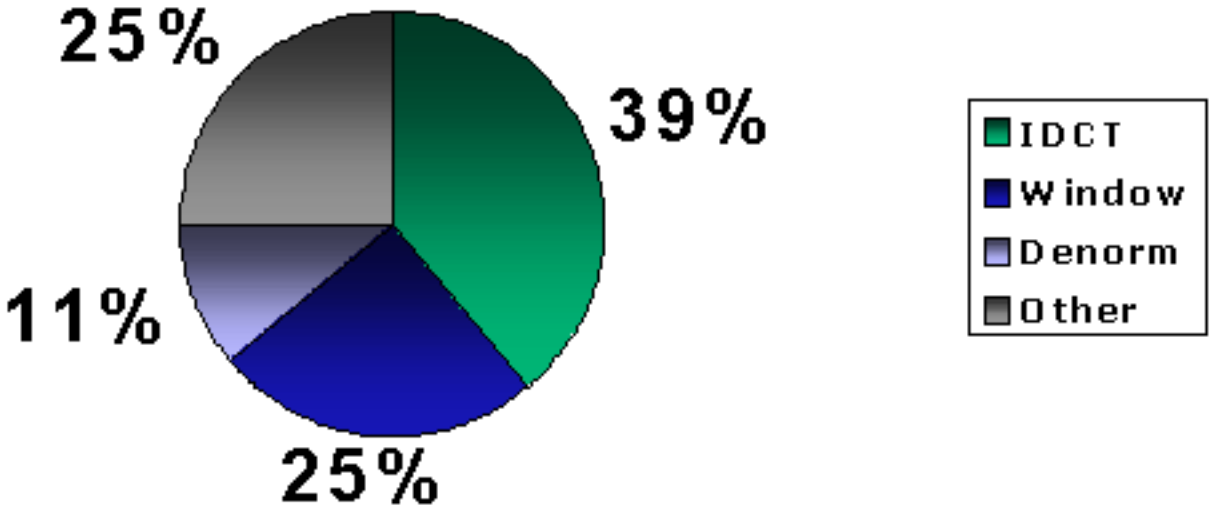
Where to Optimize?

- The 80/20 rule
  - 20% of the software uses 80% of the processing time
  - and 80% of the software uses 20% of the processing time
- DSP algorithm software spends most of its time in loops
  - This is where optimization for speed and power is most valuable
- Control software takes up the bulk of the program memory, but only a fraction of the processing time
  - Control software is the best candidate for memory optimization

Application Profiling

AC-3 Execution Time Profile

- Which modules take the longest time?
  - They're the best candidates for speed optimization



Application Profiling (cont'd)

- Similarly…
  - Which modules take the most memory?
    - They're the best candidates for memory optimization
  - Which modules are the most power-consuming?
    - They're the best candidates for power consumption optimization
    - Generally the same modules that take most time use the most power
- In a system with both a microcontroller and a DSP processor, a task may execute on either processor
  - Depending on which is faster, takes less memory, or less power

High-Level Language Optimizations

Or, "be smarter than the compiler"

Compiler Shortcomings

- Compilers typically don't generate sufficiently optimized DSP software
  - High-level language is sequential, but processors often aren't
  - Some DSP features aren't supported in the most common high-level languages
  - Memory layout is important, but compilers don't know that
  - Specialized addressing modes aren't supported
  - The best optimizer is still the human mind
- The programmer often must hand-optimize software
  - But then there's the development time...

High-Level Language Optimizations

- Simplify complex statements
- Simplify pointer arithmetic
- Arrange data in memory banks
- Help the compiler

Simplify Complex Statements

- One loop with complex conditions

```
for (A = 1; ((A-B < 4) && (A < 10)); A++)
        { . . . }
```

- becomes two loops with simple conditions

```
/* if you know that B+4 < 10 */
for (A = 1; A < B+4; A++)
        { . . . }
for (; A < 10; A++)
        { . . . }
```

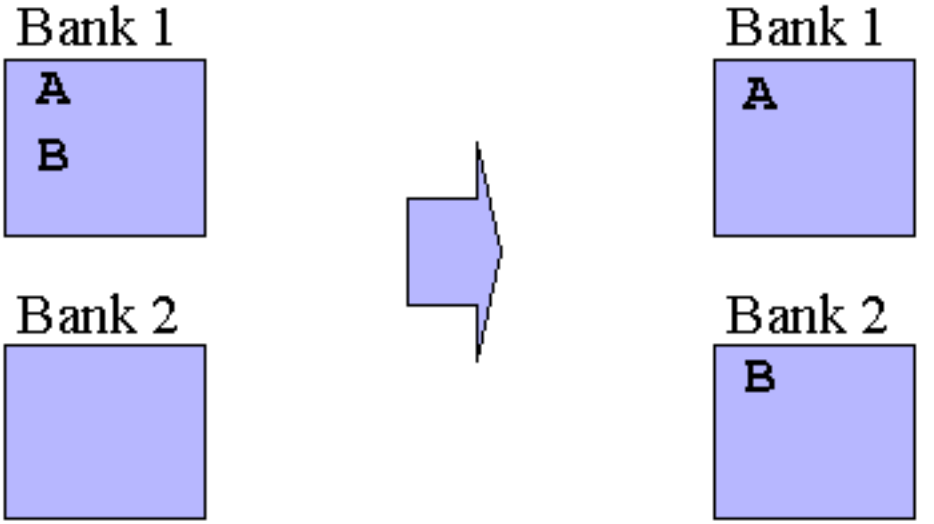
Simplify Pointers

- DSPs are very good at some kinds of pointer arithmetic...
- But not very good at other kinds
  - Vectorize matrix operations from forms of A[i][j] to B[k] or *B++
  - Avoid hierarchical pointers
  - Avoid pointer swapping
- GPPs usually handle pointers better than DSPs
  - But it generally pays to optimize pointers

Arrange Data in Memory Banks

- Virtually all DSP processors use multiple memory banks
  - Statements that require multiple data accesses per arithmetic operation can take advantage of multiple banks
  - Some GPPs also have multiple memory banks

```
sum += A[i] * B[i]
```



Help the Compiler

- Eliminate unnecessary function call overhead of small, often-called functions
- Rewrite expressions to help the compiler
  - The original code is

```
for(i=0; i<N; i++) sum += A[i] * B[i];
```

  - But the compiler may generate faster code if the code is

```
ptA = A;  ptB = B;
for(i=0; i<N; i++) sum += *ptA++ * *ptB++;
```

Loop Optimization

Or, "your errors will be repeated in your next iteration"

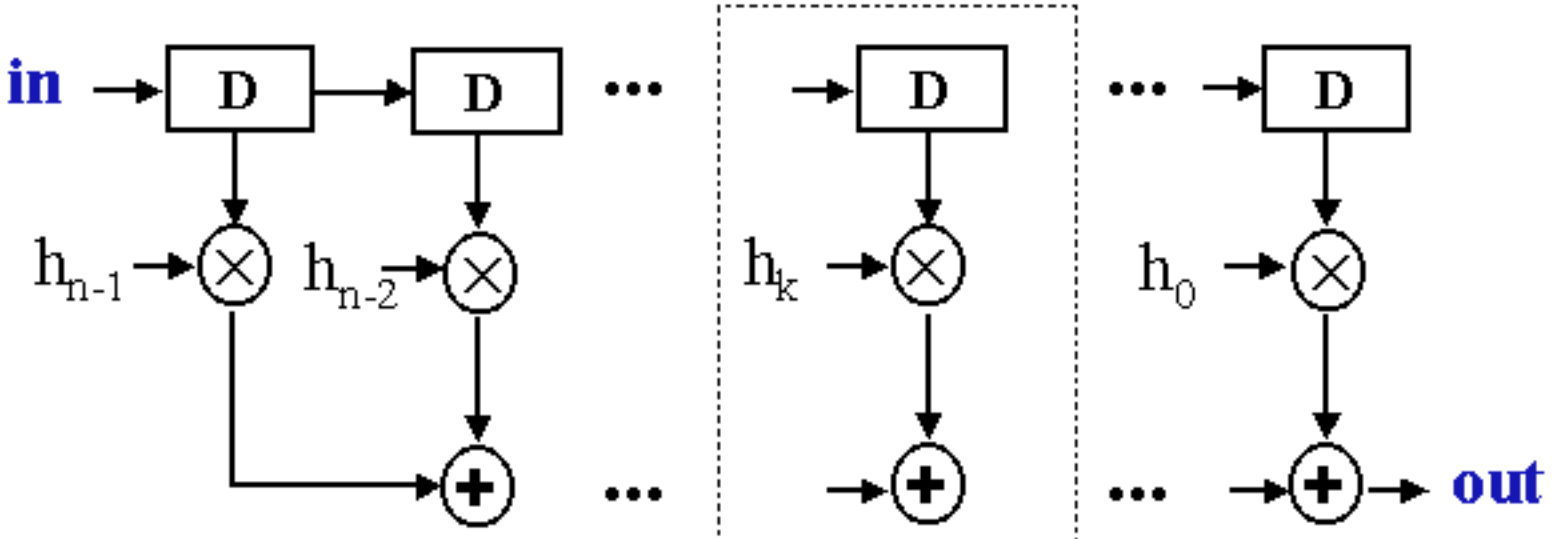
A Recipe for Loop Optimization

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:
- multiply
- add
- 2 loads
- 1 store

Resources:
- multiplier
- ALU
- 2 AGUs, 2 buses

Four Categories of Algorithmic Transformations, With Examples

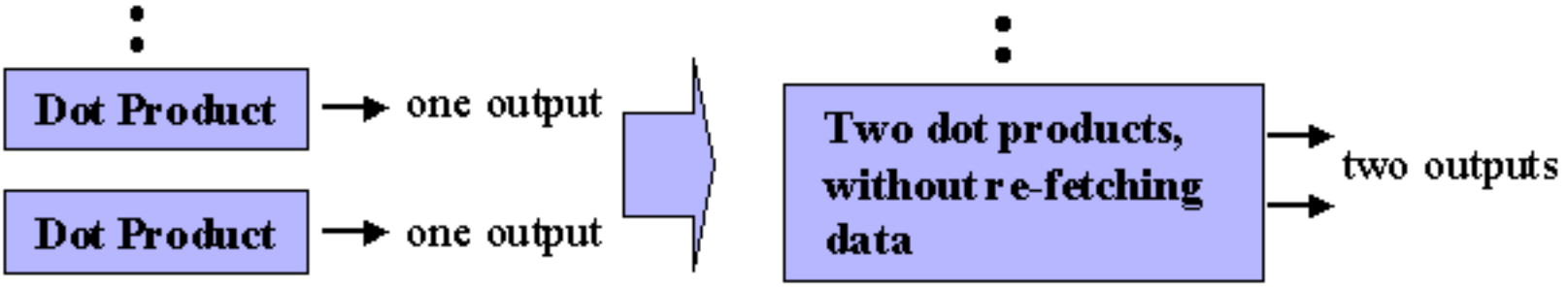
1. Unrolling across outer loops
2. Combining loops
3. Identifying operations that can be moved outside of a loop
4. Rearranging data in memory

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

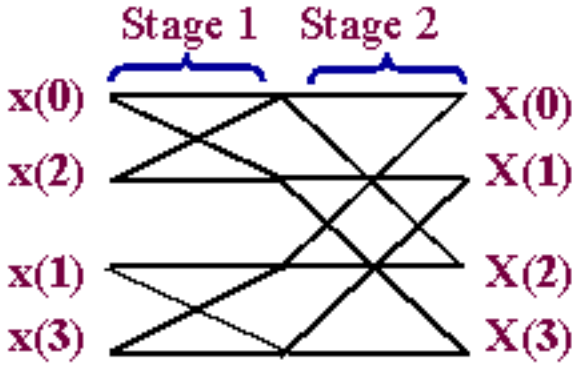
Block FIR Filter using "Zipping"




```
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 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 in R2, y0 in R3
```

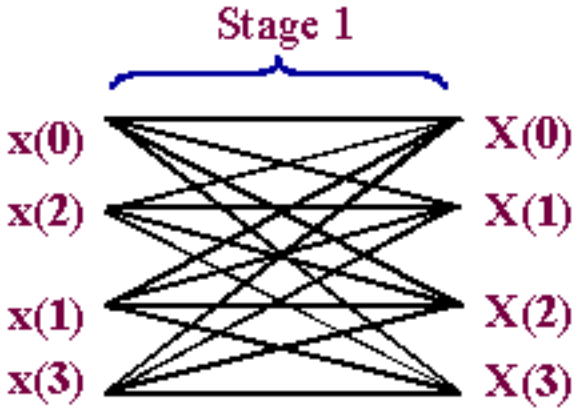
Radix-2 vs Radix-4 FFT Butterfly Structures



Radix-2:

Each butterfly requires:

- 8 Memory accesses
- 4 Multiplications
- 6 Additions



Radix-4:

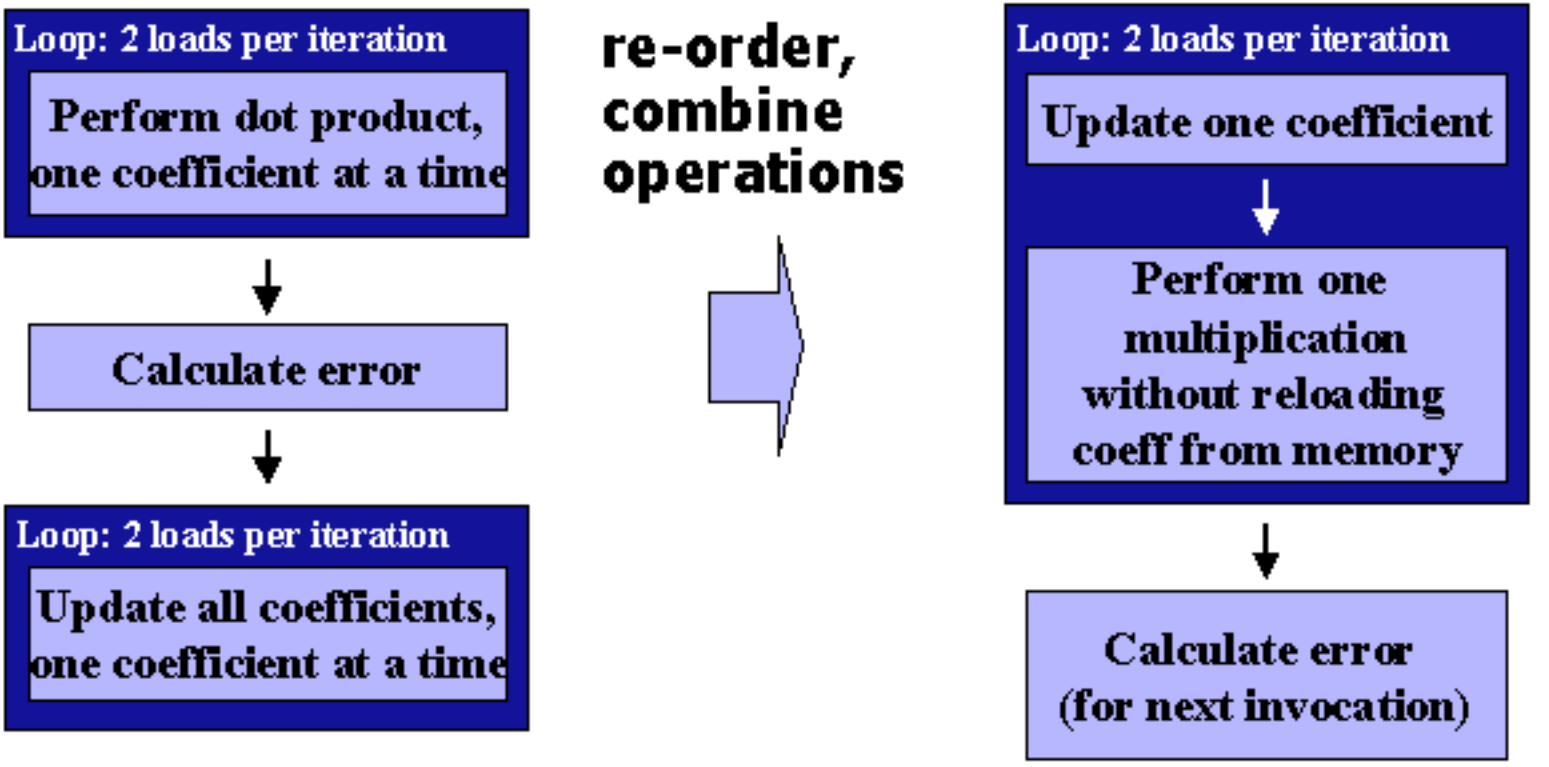
Each butterfly requires:

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

2. Combining Loops

- Goal: Avoid spending overhead on two separate loops
- Identify operations that are repeated in both loops
  - Combine the two loops so that those operations are only performed once
- Allows better re-use of intermediate results

LMS Adaptive FIR Filter

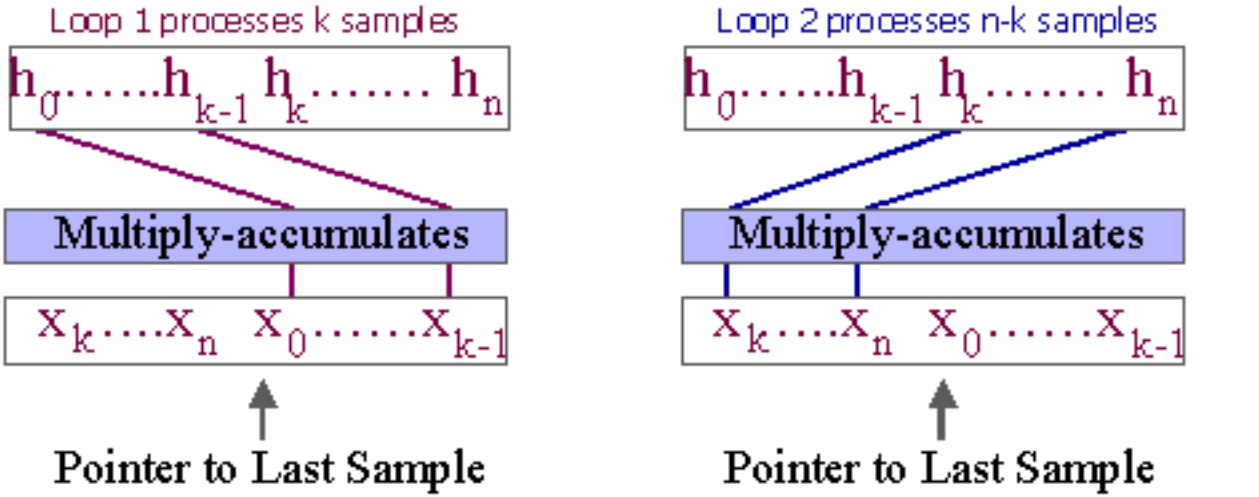


3. Moving Operations Outside Loop

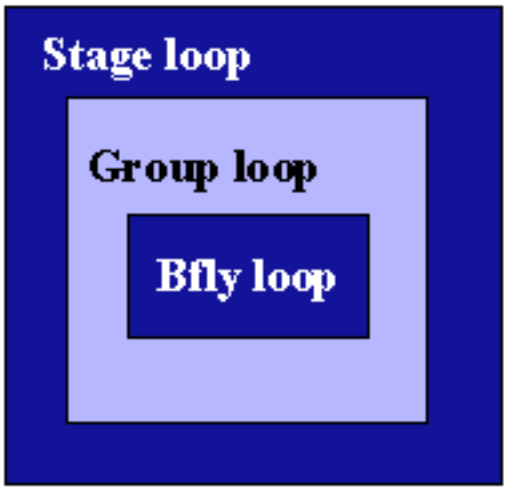
- Goal: Use *a priori* knowledge of the algorithm to avoid repeated operations
  - Identify calculations that produce constant results over the duration of the loop
- Move redundant work outside the loop

Circular Buffering for FIR Filter

- No support for modulo addressing. How to avoid checking for wraparound at each iteration?
  - Find wraparound point outside the loop since it is constant over the duration of the loop
  - Split the loop into two loops that don't wrap



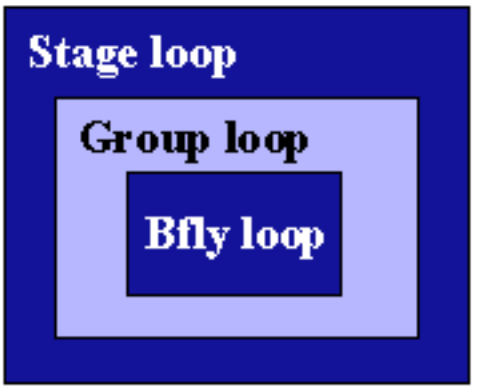
Radix-2 FFT



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

1st Stage

Subsequent Stages

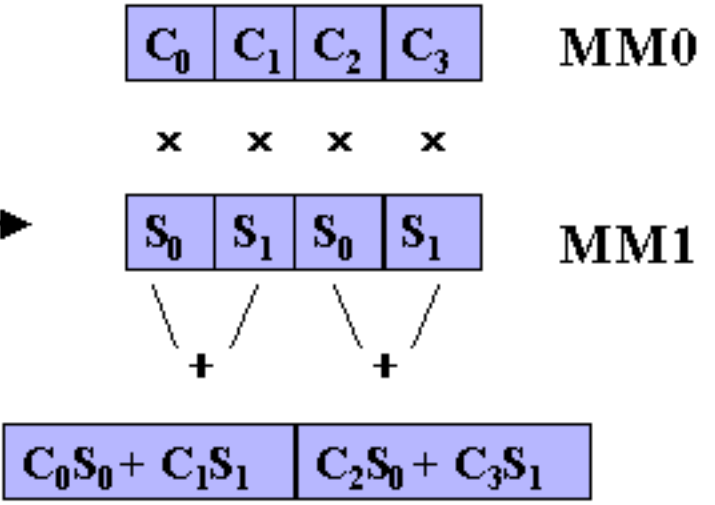


4. Arranging Data in Memory

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

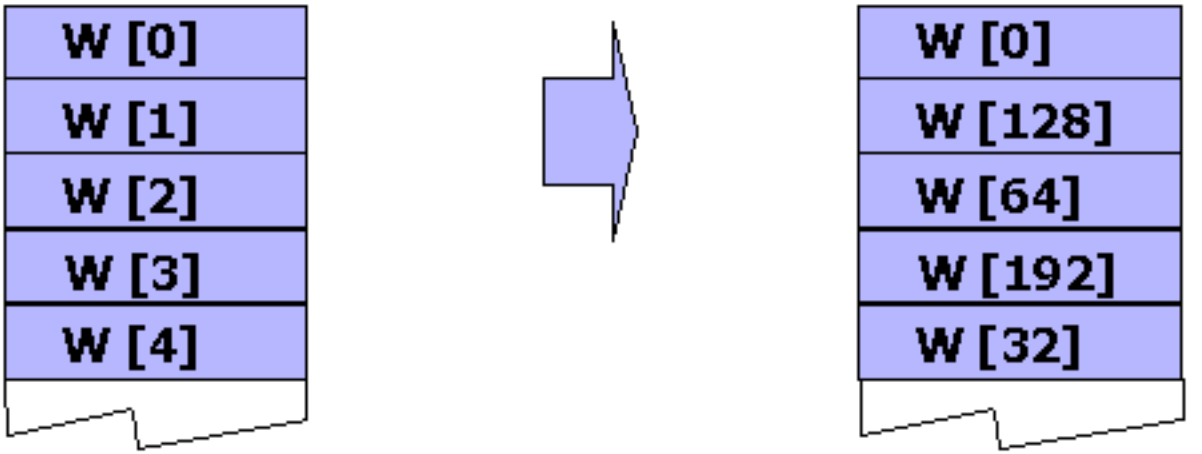
IIR Filter Biquad Section

Store two copies of filter state variables →



Radix-2 FFT

Arrange twiddle factors
in bit-reversed order



Scheduling Techniques

Or, "time is a measure of what *isn't* done"

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 must schedule operations to 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

Software Pipelining

- Main advantage
  - Increases performance by making use of instruction slots that would otherwise be wasted waiting for results
- Main disadvantage
  - Software pipelining makes programs more complicated, harder to read and maintain

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                  ; wait for loads to finish
   MPYHL .M1X A2,B2,A3     ; P0(i)=coef(2i)*state(2i)
|| MPYLH .M2X A2,B2,B7     ; P1(i) =coef(2i+1)*state(2i+1)
|| ADD  .S2  -1,B0,B0      ; Dec loop counter
    NOP 1                  ; wait for multiplies to finish
    ADD  .L1  A0,A3,A0     ; Sum0(i) += P0(i-2)
|| ADD  .L2  B1,B7,B1      ; Sum1(i) += P1(i-2)
||[B0] B  .S1 LOOP         ; Cond. Branch to LOOP
   NOP  5                  ; wait for branch to take effect

; Loop ends here
```

FIR Filter on 'C62xx

With SW Pipelining:
0.5 Cycles/Tap
35 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]
```

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

Vector Addition on TMS320C2700

- TMS320C2700 has high loop overhead
  - No multi-instruction hardware looping
  - Branches are costly

```
loop:   mov  ah,*ar2++              ; load element a0
        add  ah,*ar3++              ; add element b0
        mov  *ar4++,ah              ; store sum a0+b0
        ; repeat the loop body instructions
        mov  ah,*ar2++              ; load element a1
        add  ah,*ar3++              ; add element b1
        mov  *ar4++,ah              ; store sum a1+b1
        banz  loop,ar0--            ; branch to loop
```

Dot Product on TigerSHARC

No unrolling, no SW pipelining: 0.5 Cycles/Tap

```
loop:
   XR1:0 = Q[j10+=2]; YR1:0 = Q[k0+=2];;   // load 8
   samples
   XR3:2 = Q[j0+=2]; YR3:2 = Q[k0+=2];;    // load 8 coeffs

   // one cycle stall happens here

if NLC0E, jump loop; MR3:0 += R1:0*R3:2;;  // loop, 8 MACs
```

Dot Product on TigerSHARC

With unrolling, SW pipelining: 0.125 Cycles/Tap

```
loop:
  // load 8 coeffs, 8 samples, do 8 MACs
  YR7:4   = Q[j0+=4]; YR11:8  = Q[k0+=4]; MR3:0 += R7:6   * R11:10;;

  // load 8 coeffs, 8 samples, do 8 MACs
  XR7:4   = Q[j0+=4]; XR11:8  = Q[k0+=4]; MR3:0 += R13:12 * R17:16;;

  // load 8 coeffs, 8 samples, do 8 MACs
  YR15:12 = Q[j0+=4]; YR19:16 = Q[k0+=4]; MR3:0 += R15:14 * R19:18;;

  // branch, load 8 coeffs, 8 samples, do 8 MACs
  if NLCOE,jump loop;
       XR15:12 = Q[j0+=4]; XR19:16 = Q[k0+=4]; MR3:0 += R5:4 * R9:8;;
```

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 intermed results
   pmaddwd   mm1, COEFaddr[edi+8]   ; 4 multiplies, 2 adds
   paddd     mm7, mm3               ; accumulate intermed results
   movq      mm2, [esi+16]          ; load four new samples
   movq      mm3, [esi+24]          ; load four new samples
   paddd     mm7, mm0               ; accumulate intermed results
   pmaddwd   mm2, COEFaddr[edi+16]  ; 4 multiplies, 2 adds
   paddd     mm7, mm1               ; accumulate intermed 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
```

Don't Follow the Rules

Or, "rules are made to be broken"

Specialized Instructions

- Examples (not exhaustive):
  - **ADSP-2116x has specialized instruction for FFT**
    - One multiplication, and sum and difference of two operands
  - **TMS320C54x has several specialized instructions**
    - LMS, symmetrical FIR filter, polynomial evaluation, ...
  - **G4 includes instruction useful for LMS**
    - Eight multiplications, eight additions, eight roundings with single-cycle throughput
  - **DSP16xxx has application-specific operations**
    - Extended-precision multiplication specialized for enhanced full-rate GSM

Non-Conventional Use of Execution Units

- IS-54 convolutional encoder bit-interleaving on Pentium
  - Pentium can't do single-cycle

```
rcl     ecx, 1          ; rotate, insert prev carry bit
shl     eax, 1          ; shift, generate new carry bit
```

  - But it can do single-cycle

```
rcl     ecx, 1          ; rotate, insert prev carry bit
add     eax, eax        ; shift, generate new carry bit
```

- Vector maximum search on the ZSP ZSP164xx

```
loop: ldu r15,r14,1     ; new 16-bit value in
                        ; r15, address in r14
max.e  r4,r14           ; 32-bit max {r5,r4},
                        ;{r15,r14} includes address
agn0   loop             ; next value
```

Other Tweaks

- Complex multiplication (a+jb)*(c+jd)
  - Doesn't need to be 4 multiplications, 2 additions,
  - but can be 3 multiplications, 5 additions (why?)
  - and a complex dot product can be four real, partial dot products (why?)
- Look-up tables for FFT bit-reversal or Hamming distance
  - Often allows significant speed optimization, but can be costly in terms of memory usage
- ... and much, much more!

If Algorithmic Transformations Don't Help...

- Choose a different algorithm
  - A lower-order IIR filter may be used instead of a higher-order FIR filter
  - Gradient search (LMS) adaptive filter algorithm is less compute intensive than recursive least squares (RLS) algorithm
  - Different algorithms may cause other problems
    - For example, an IIR filter isn't unconditionally stable
- Trade quality of sound or video for faster processing
  - Product may become less expensive...
  - but with poorer quality

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 the processor, the algorithm, and the application
- Be aware of trade-offs between speed, memory usage, and power consumption

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