SIGNAL PROCESSING ON THE TMS320C6X VLIW DSP

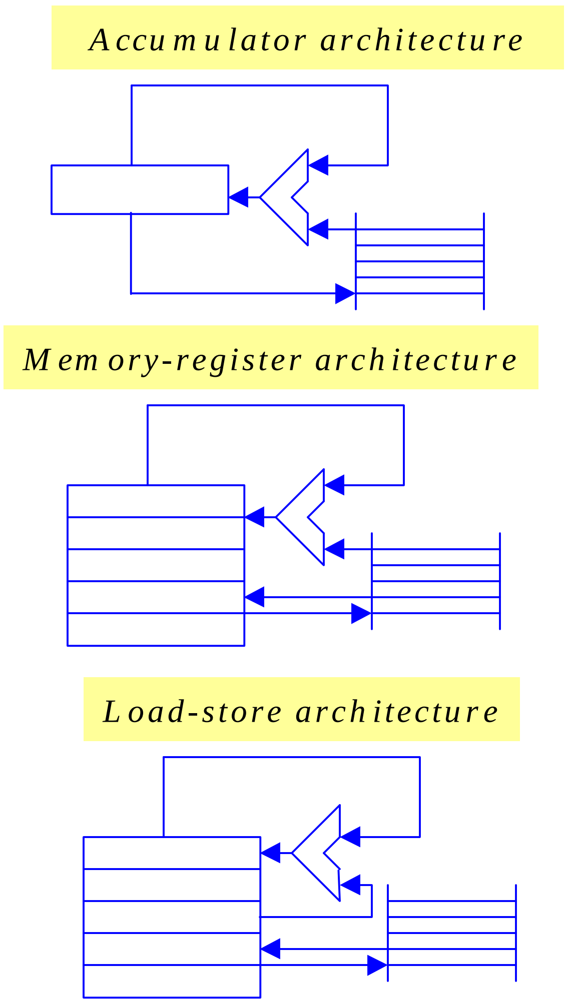
Prof. Brian L. Evans

in collaboration with

Niranjan Damera-Venkata and
Magesh Valliappan

Embedded Signal Processing Laboratory
The University of Texas at Austin
Austin, TX 78712-1084

http://signal.ece.utexas.edu/



Outline

- Introduction
- FIR filters
- Discrete cosine transform
- Lookup tables
- Assembler, C compiler, and programming hints
- Software pipelining
- Compiler efficiency
- Conclusion

TMS320C6x Processor

- Architecture
  - 8-way VLIW DSP processor
  - RISC instruction set
  - 2 16-bit multiplier units
  - Byte addressing
  - Modulo addressing
  - Non-interlocked pipelines
  - Load-store architecture
  - 2 multiplications/cycle
  - 32-bit packed data type
  - No bit reversed addressing
- Applications
  - Wireless base stations
  - xDSL modems
  - Videoconferencing
  - Document processing

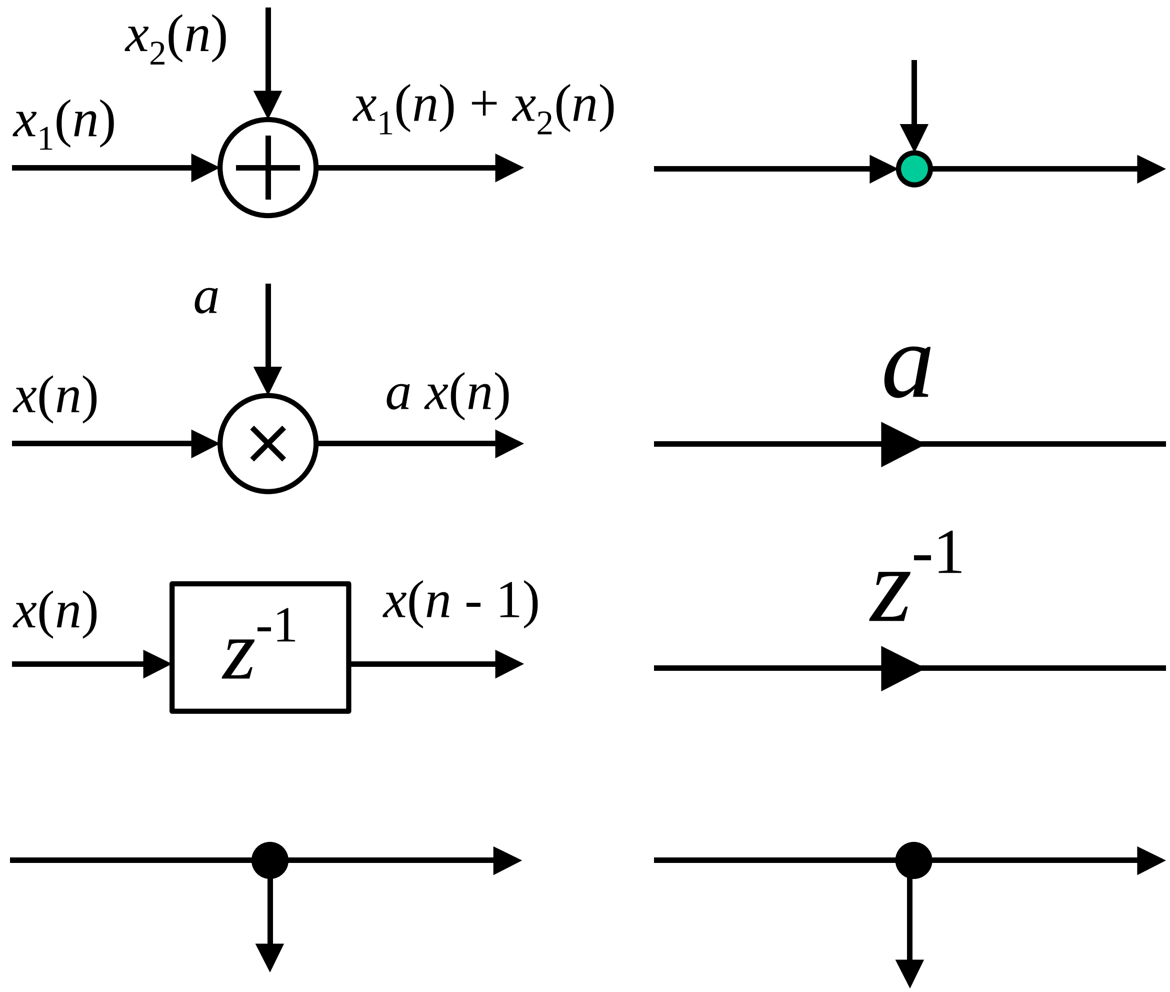
Signal Flow Graph Notation

Addition
(adder)

Multiplication
(multiplier)

Delays
(register or memory)

Branch



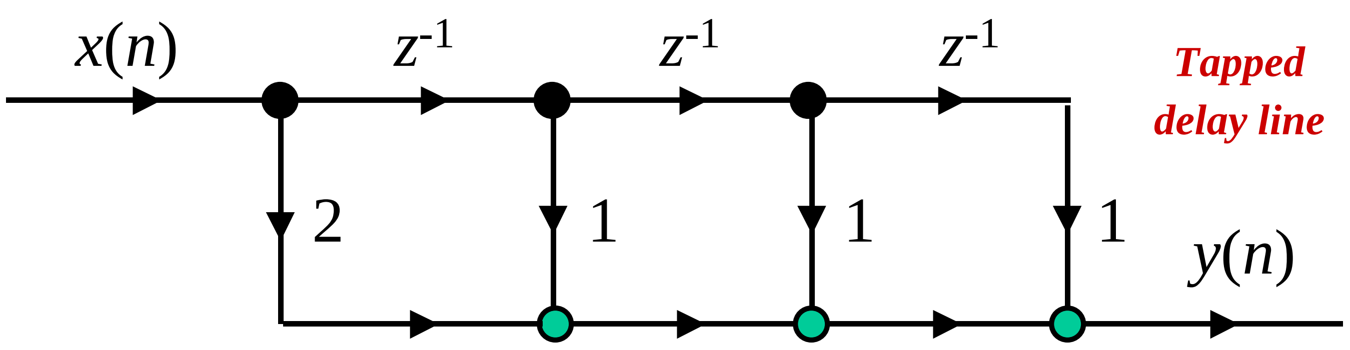
FIR Filter

- Difference equation (inner product)

$$y(n) = 2\,x(n) + x(n-1) + x(n-2) + x(n-3)$$

- Signal flow graph

$$y(n) = \sum_{i=0}^{N-1} a(i)\; x(n-i)$$



- Vector dot product plus circularly buffer input

Optimized Vector Dot Product on the C6x

- Prologue
  - Retime dot product to compute two terms per cycle
  - Initialize pointers: A5 for $a(n)$, B6 for $x(n)$, A7 for $y(n)$
  - Move number of times to loop (N) divided by 2 into A2
- Inner loop
  - Put $a(n)$ and $a(n+1)$ in A0 and $x(n)$ and $x(n+1)$ in A1 (***packed data***)
  - Multiply $a(n)\,x(n)$ and $a(n+1)\,x(n+1)$
  - Accumulate even (odd) indexed terms in A4 (B4)
  - Decrement loop counter (A2)
- Store result

Reg	Meaning
A0	$a(n)\ \|\|\ a(n+1)$
B1	$x(n)\ \|\|\ x(n+1)$
A2	$(N-n)/2$
A3	$a(n)\,x(n)$
B3	$a(n+1)\,x(n+1)$
A4	$y_{even}(n)$
B4	$y_{odd}(n)$
A5	$\&a$
B6	$\&x$
A7	$\&y$

FIR Filter Implementation on the C6x

```
        MVK   .S1 0x0001,AMR ; modulo block size 2^2
        MVKH  .S1 0x4000,AMR ; modulo addr register B6
        MVK   .S2 2,A2       ; A2 = 2 (four-tap filter)
        ZERO  .L1 A4         ; initialize accumulators
        ZERO  .L2 B4
; initialize pointers A5, B6, and A7
fir     LDW   .D1  *A5++,A0  ; load a(n) and a(n+1)
        LDW   .D2  *B6++,B1  ; load x(n) and x(n+1)
        MPY   .M1X A0,B1,A3  ; A3 = a(n) * x(n)
        MPYH  .M2X A0,B1,B3  ; B3 = a(n+1) * x(n+1)
        ADD   .L1  A3,A4,A4  ; yeven(n) += A3
        ADD   .L2  B3,B4,B4  ; yodd(n) += B3
[A2]    SUB   .S1  A2,1,A2   ; decrement loop counter
[A2]    B     .S2  fir       ; if A2 != 0, then branch
        ADD   .L1  A4,B4,A4  ; Y = Yodd + Yeven
        STH   .D1  A4,*A7    ; *A7 = Y
```

Throughput of two multiply-accumulates per cycle

Discrete Cosine Transform (DCT)

- DCT of sequence $x(n)$ defined on n in $[0, N\text{-}1]$

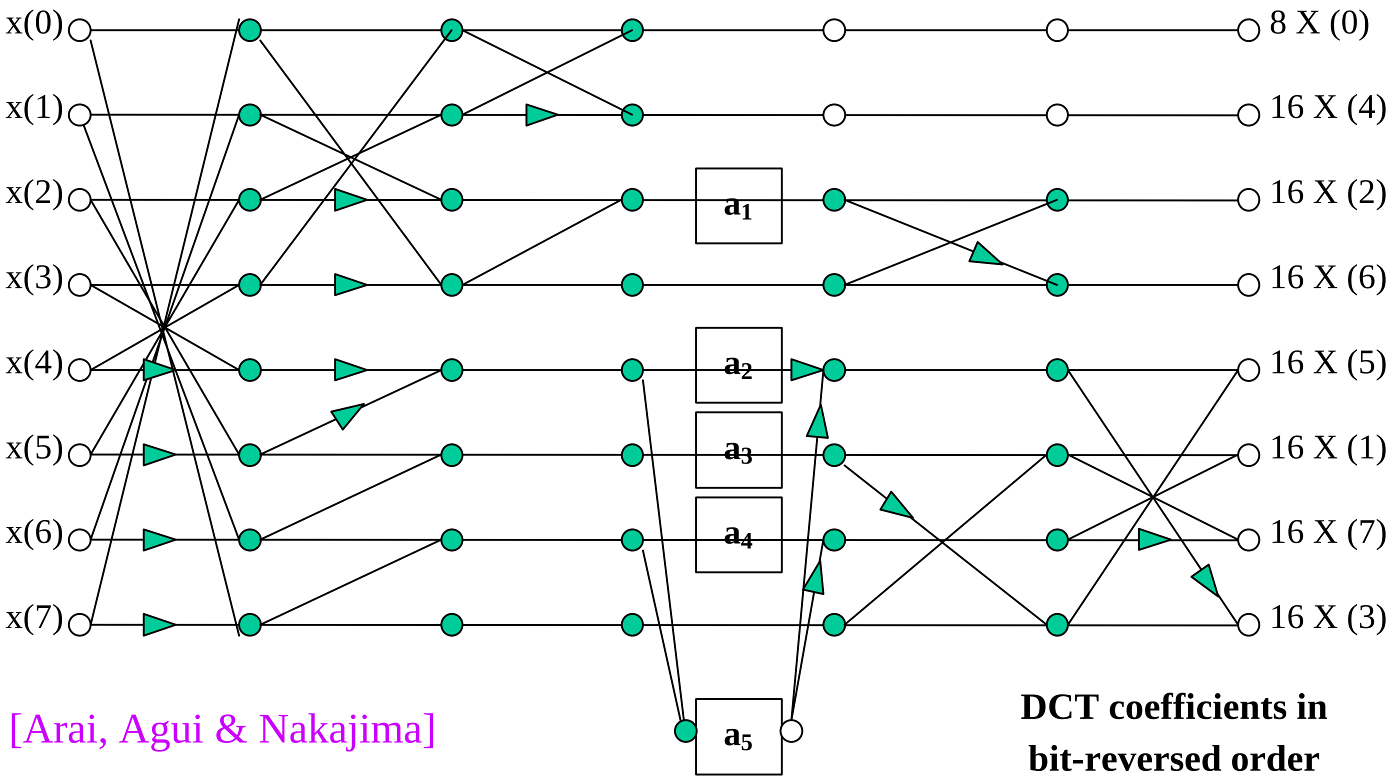
$$X_{DCT}(k) = \sum_{n=0}^{N-1} x(n)\, C_k \cos\left(\frac{2n+1}{2N} kp\right)$$

$$x(n) = \sum_{k=0}^{N-1} X_{DCT}(k)\, C_k \cos\left(\frac{2n+1}{2N} kp\right)$$

$$C_k = \begin{cases} \dfrac{1}{\sqrt{N}}, & k = 0 \\ \dfrac{2}{\sqrt{N}}, & k > 0 \end{cases}$$

A Fast DCT Implementation

- Arrows represent multiplication by -1
- $a_1=0.707$, $a_2=0.541$, $a_3=0.707$, $a_4=1.307$, $a_5=0.383$



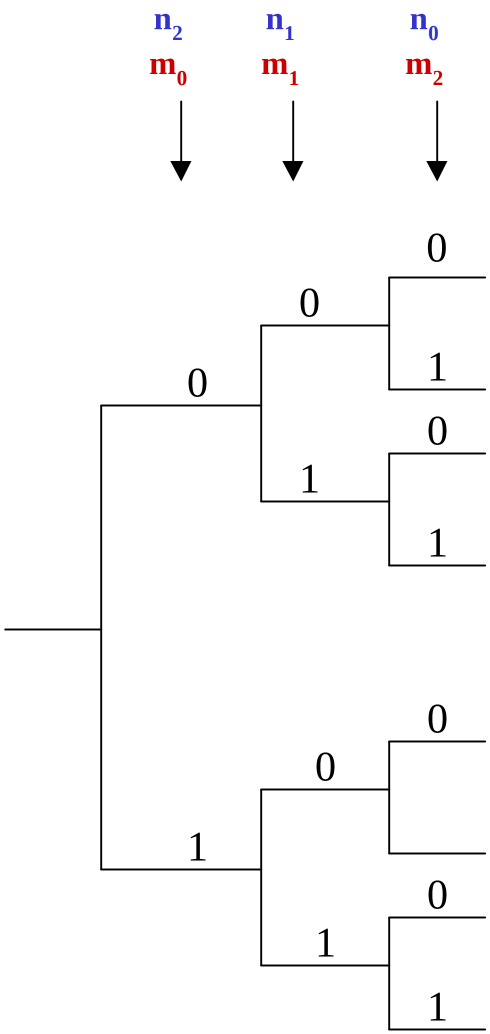
[Arai, Agui & Nakajima]

DCT coefficients in bit-reversed order

Bit Reversed Sorting on the C6x

- In-place computations using discrete transforms
  - Input or output value at index 1010_2 at index 0101_2
  - Emulate bit-reversed addressing on C6x: in transform-domain filtering, avoid by permuting filter coefficients
- Linear-time constant-space algorithm
  - Chad Courtney, "Bit-Reverse and Digit-Reverse: Linear-Time Small Lookup Table Implementation for the TMS320C6000," *TI Application Note SPRA440*, 5/98
  - Higher radix transforms use digit-reversed addressing
  - Divide-and-conquer approach augmented by lookup tables for short bit lengths
  - Avoid swapping values twice

Linear-Time Bit-Reversed Sorting



Normal order	*Bit-reversed order*
$x[n_2\ n_1\ n_0]$	**$X[m_2\ m_1\ m_0]$**
x[0 0 0]	**X[0 0 0]**
x[0 0 1]	**X[1 0 0]**
•	•
•	•
•	•
x[1 1 1]	**X[1 1 1]**

C6x bit operations

Inst	Meaning
clr	*clear bit field*
ext	*extract bit field*
lmbd	*give position of leftmost bit*
norm	*Normalize integer; give # redundant sign bits*
set	*Set bit field to ones*

Lookup Table Bit-Reversed Sorting

- Store pre-computed bit-reversed indices in table
- Goals for hand-coded implementation
  - Minimize accesses to memory (equal to array length)
  - Minimize execution time
- Limitations on C6x architecture
  - Five conditional registers: A0, A1, A2, B0, and B1
  - Delay of 5 cycles for branch and 4 cycles for load/store
  - No more than four reads per register per cycle
  - One read of register file on another data path: maintain copy of loop counter and array pointer in each data path
- Example: Assume transform of length 256
  - Array indices fit into a byte (lookup table is 256 bytes)
  - Data array is a 256-word array (16 bits per coefficient)

Lookup Table Bit-Reversed Sorting

```
; A3 256-word array, B5 256-byte bit-rev index lut
        MVK  .S1  255,A2        ; index to swap 0 … 255
||      MVK  .S2  255,B2        ; 255 bit reversed is 255
||      ZERO .L1  A1            ; don’t swap first index
||      MV   .L2  A3,B3         ; B3 also points to data
        SUB  .S1  A2,1,B1       ; B1=A2-1
sort    .trip 255               ; tell assembler loop 255X
  [A2]  LDBU .D2 *B5[B1],B7     ; B7=next bit-rev index
  [A2]  SUB  .S1  A2,1,A2       ; decrement loop counter
||[B1]  SUB  .S2  B1,1,B1       ; B1=A2-1
||[A1]  MV   .L1  B2,A4         ; A4=B2 for swapping
||[A1]  MV   .L2  A2,B4         ; B4=A2 for swapping
||[A1]  LDW  .D1 *A3[A2],A6     ; A6=data at index
||[A1]  LDW  .D2 *B3[B2],B6     ; B6=data at bit-rev index
        CMPGT .L1 A2,B7,A1      ; A1=switch next values
||      MV   .L2  B7,B2         ; B2=bit-rev index
||[A1]  STW  .D1  A6,*A3[A4]    ; swap data
||[A1]  STW  .D2  B6,*B3[B4]
||[A2]  B    .S2  sort          ; if A2 != 0, then branch
```

Throughput of 3 cycles/ coefficient

Better Lookup Table Bit-Reversed Sorting

- Improve execution time by 53%
  - For a 256-length data array, only 120 swaps occur
  - Use 2 120-element arrays: index and bit-reversed index

```
; A5 and B5 120-byte index and bit-reversed index lut
        MVK  .S1  120,A2     ; loop counter
||      MV   .S2  A3,B3      ; A3/B3 point to array data
sort    .trip 120            ; tell assembler loop 120X
        LDBU .D1 *A5++,A4    ; A4=index
||      LDBU .D2 *B5++,B4    ; B4=bit-reversed index
        MV   .S1 B4,A7       ; swap indices to swap vals
||      MV   .S2 A4,B7
||      LDW  .D1 *A3[A4],A6
||      LDW  .D2 *B3[B4],B6
  [A2]  SUB  .S1 A2,1,A2     ; decrement loop counter
||[A2]  B    .S2 sort        ; if A2 != 0, then branch
||      STW  .D1 A6,*A3[A7]
||      STW  .D2 B6,*B3[B7]
```

Throughput of 1.4 cycles/ coefficient

Assembly Optimizations

- Hand coding optimizations
  - Use instructions in parallel

```
   add .L1 A1,A2,A2
|| sub .L2 B1,B2,B1    ; parallel instruction
```

  - Fill NOP delay slots with useful instructions
  - Manual loop unrolling
  - Pack two 16-bit numbers in a 32-bit register: replace two LDH instructions with LDW instruction
- Assembler optimizations
  - Assigns functional units when not specified
  - Pack and parallelize linear assembly language code
  - Software pipelining

C6x C Compiler

- Software development in a high-level language
  - Initialization and resource allocation
  - Call time-critical loops in assembly
  - C++ compilers are under development
- Compiler optimization
  - Disable symbolic debugging to enable optimization
  - Optimize registers, local instructions, global program flow, software pipelining, and across multiple files
  - Use **volatile** keyword to prevent removal of wait loops (dead code) and unused variables (shared resource)

Efficient Use of C Data Types

- *int* is 32 bits (width of CPU and register busses)
- 16 bit x 16 bit multiplication in hardware
  - multiplying *short* is 4x faster than multiplying *int*
  - adding packed *shorts* is 2x faster than adding *int*
- 32-bit byte addressing (access to 4 Gbyte range)
- *long* is 40 bits
  - useful for extended precision arithmetic (8 guard bits)
  - performance penalty
  - in assembler, *.long* means 32 bits
- C67x adds support for *float* and *double*

Volatile Declarations

- Optimizer avoids memory accesses when possible
  - Code which reads from memory locations outside the scope of C (such as a hardware register) may be optimized out by the compiler
  - To prevent this, use the **volatile** keyword
- Example: wait for location to have value 0xFFFF

```
         unsigned short *ctrl;    /* wait loop */
         while(*ctrl != 0xFFFF);  /* loop would be removed */
```

```
volatile unsigned short *ctrl;    /* safe declaration */
         while(*ctrl != 0xFFFF);
```

Software Pipelining

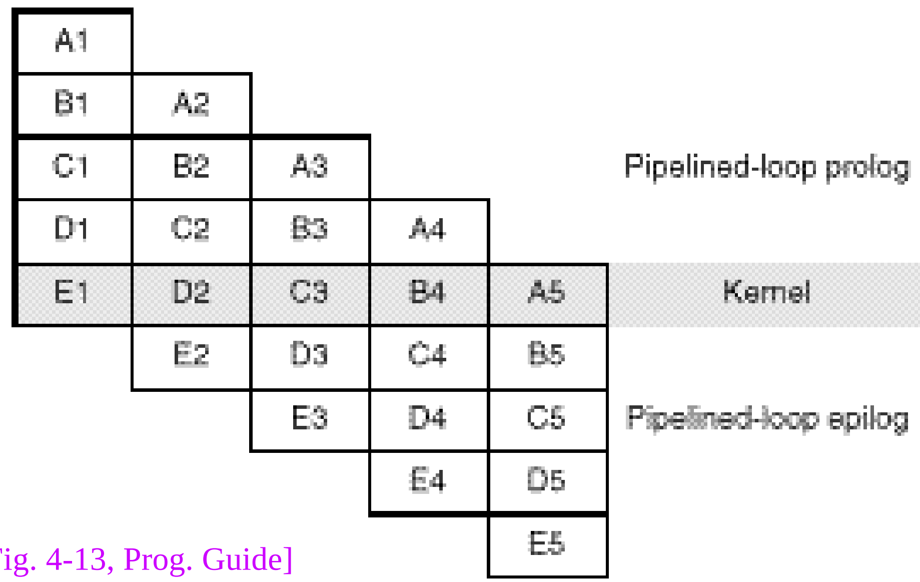
- Enabled with -o2 and -o3 compiler options
- Example
  - Stages of the loop are A, B, C, D, and E
  - A maximum of five stages execute at the same time

Trip count

Redundant loops

Loop unrolling

Speculative execution (*epilog removal*)



[Fig. 4-13, Prog. Guide]

Trip Count and Redundant Loops

- Trip count is minimum number of times a loop executes
  - Must be a constant
  - Used in software pipelining by assembler optimizer if loop counts down
  - Compiler can transform some loops to count down
  - If compiler cannot determine that a loop will always execute for the minimum trip count, then it generates a redundant unpipelined loop
- Communicating trip count information in C
  - Use -o3 and -pm compiler options
  - Use _nassert intrinsic

```
_nassert(N >= 10);
```

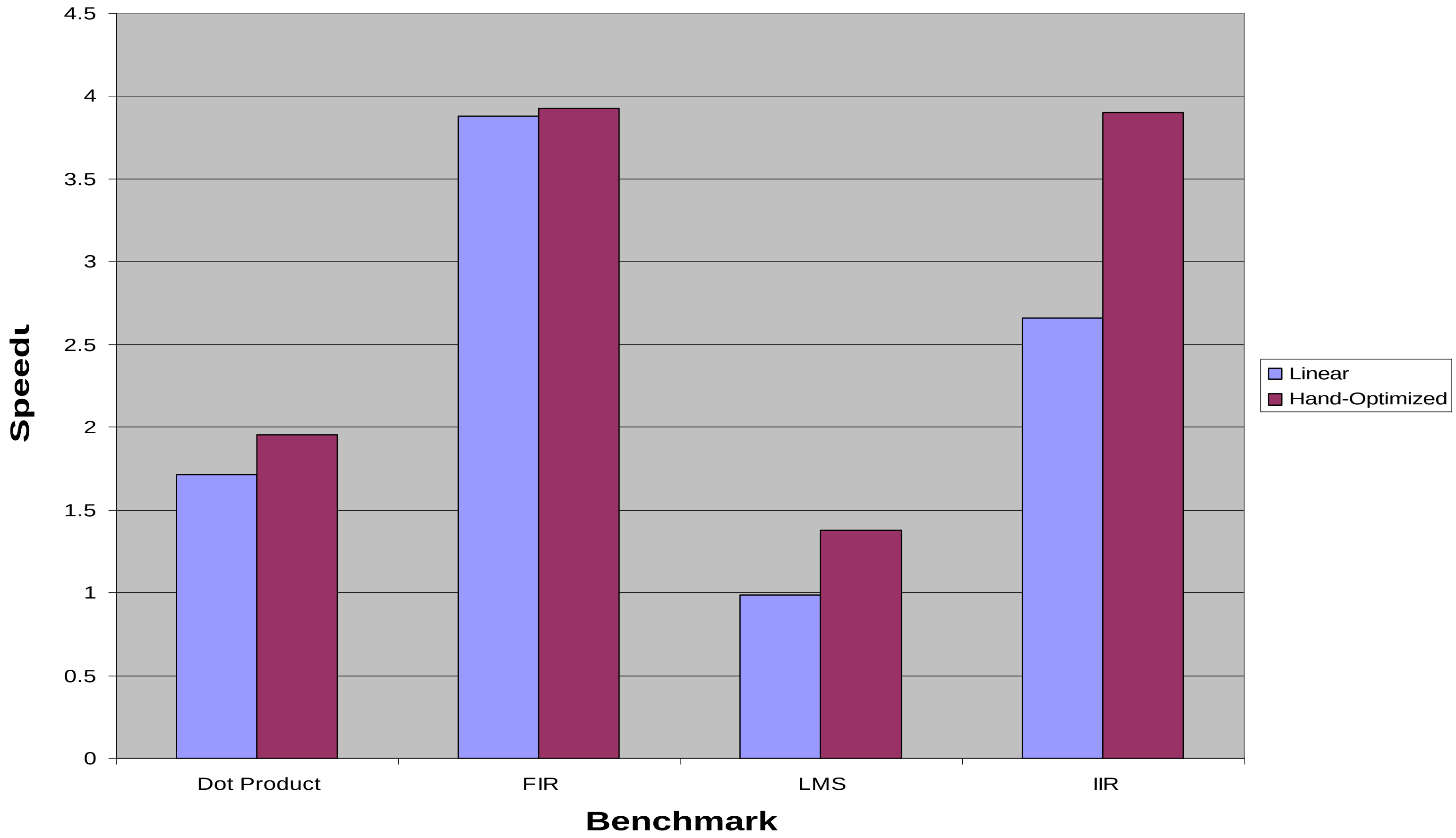
Specifying Minimum Iteration Count

```
; Procedure Dotp with 3 arguments placed in a4,b4,a6
Dotp:  .proc a4, b4, a6      ; beginning of procedure
       .reg p_m, m, p_n, n, prod, sum, len
       mv  a4, p_m           ; pointer to vector m
       mv  b4, p_n           ; pointer to vector n
       mv  a6, len           ; vector length
       zero sum
loop:  .trip 40              ; minimum iteration count
       ldh *p_m++, m
       ldh *p_n++, n
       mpy m, n, prod
       add prod, sum, sum
[len]  sub len, 1, len
[len]  b   loop
       mv sum, a4
       .endproc a4           ; return a4
```

Software Pipelining Limitations

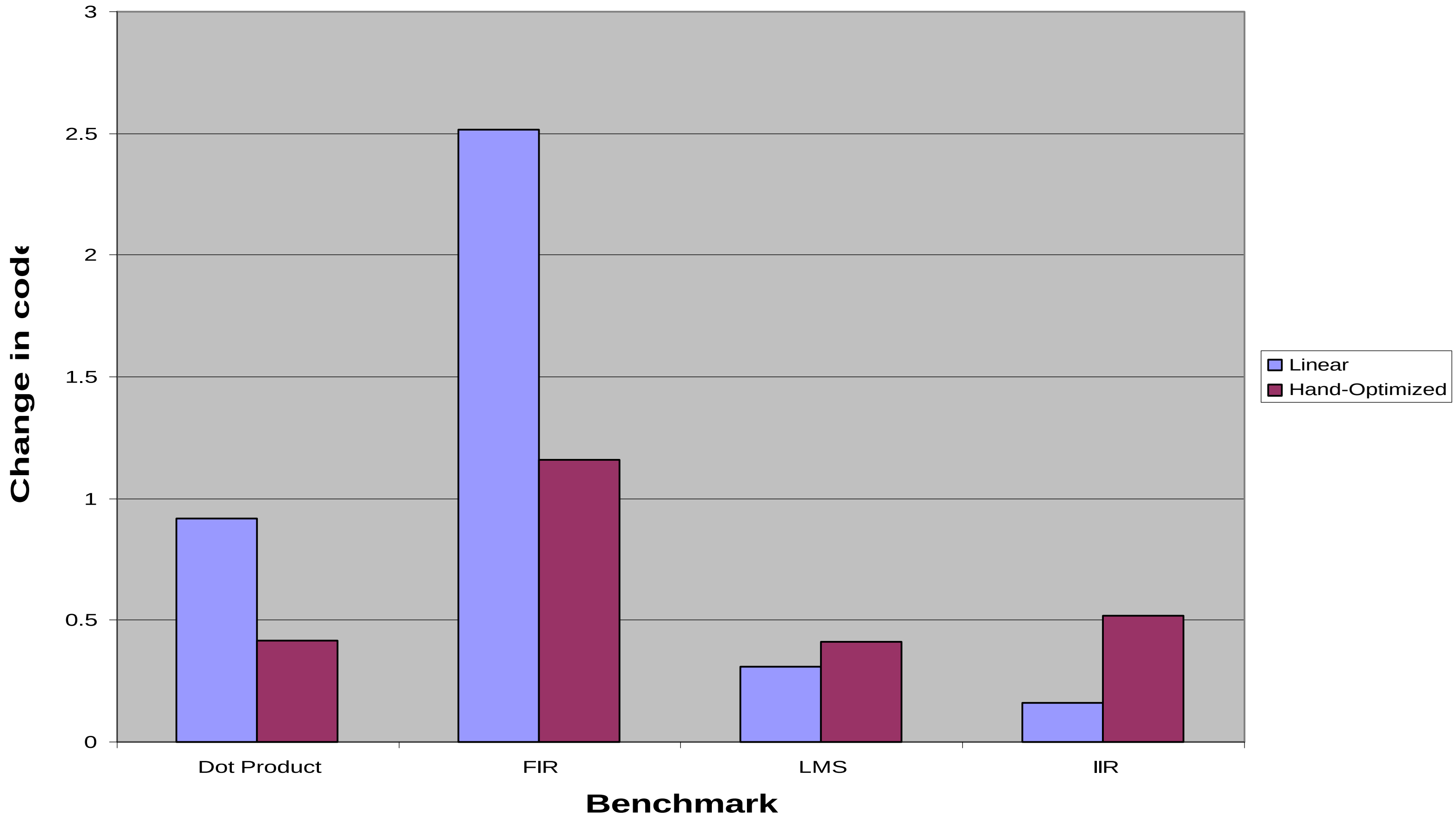
- Only innermost loop may be pipelined
- Any of the following inside a loop prevents software pipelining [Prog. Guide, Section 4.3.3]
  - Function calls (intrinsics are okay)
  - Conditional break (early exit)
  - Alteration of loop index (conditional or unconditional)
  - Requires more than 32 registers
  - Requires more than 5 conditional registers
- C intrinsics allow explicit access to special architectural features such as packed data types

C Compiler Efficiency



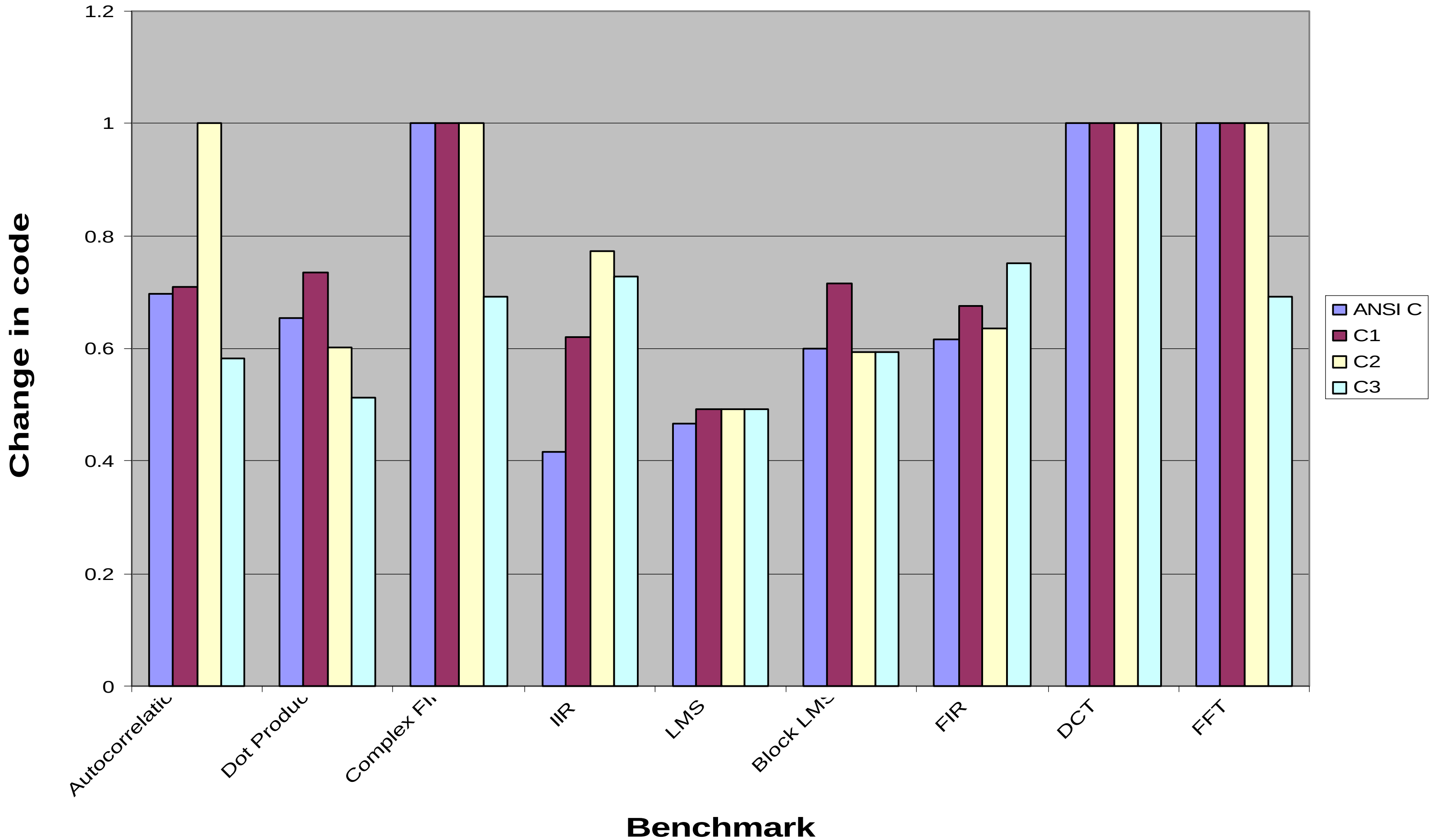
Speedup of assembly versions over ANSI C versions

C Compiler Efficiency



Change in code size of assembly versions over ANSI C versions

C Compiler Efficiency



Effect of epilog removal on compiled C code

C Compiler Efficiency

- Different C compiler optimizations for FIR filter
  - M outputs and N filter coefficients
  - Each achieves a throughput of 2 MACs/cycle
  - Least overhead in #2 (still 25% overhead)

Redundant Load Elimination	*Outer Loop Unrolling Factor*	*Inner Loop Unrolling Factor*	*Intrinsics*	*Clock cycles*	*Overhead*
	8	2	×	M(N+11)/2 + 22	11M/2+22
	4	2	×	M(N+9)/2 + 47	9M/2 + 47
	2	2	×	M(N+19)/2 + 19	19M/2+19
´	2	4		M(N+24)/2 + 24	12M+24
´	2	2		M(N+19)/2 + 19	19M/2 + 19

Conclusion

Arithmetic	Logical	Data Management
ABS	AND	LD
ADD	CMPEQ	MV
ADDA	CMPGT	MVC
ADDK	CMPLT	MVK
ADD2	NOT	MVKH
MPY	OR	ST
MPYH	SHL	
NEG	SHR	
SMPY	SSHL	
SMPYH	XOR	
SADD		
SAT		
SSUB		
SUB		
SUBA		
SUBC		
SUB2		
ZERO		

Bit Management	Program Control
CLR	B
EXT	IDLE
LMBD	NOP
NORM	
SET	

C6x Instruction Set by Category

(un)signed multiplication

saturation/packed arithmetic

Conclusion

.S Unit	
ADD	NEG
ADDK	NOT
ADD2	OR
AND	SET
B	SHL
CLR	SHR
EXT	SSHL
MV	SUB
MVC	SUB2
MVK	XOR
MVKH	ZERO

.L Unit	
ABS	NOT
ADD	OR
AND	SADD
CMPEQ	SAT
CMPGT	SSUB
CMPLT	SUB
LMBD	SUBC
MV	XOR
NEG	ZERO
NORM	

.D Unit	
ADD	ST
ADDA	SUB
LD	SUBA
MV	ZERO
NEG	

.M Unit	
MPY	SMPY
MPYH	SMPYH

Other	
NOP	IDLE

C6x Instruction Set by Category

Six of the eight functional units can perform add, subtract, and move operations

Conclusion

- C compiler's performance with ANSI C code far from optimal (average of 2.4 times slower)
- Manual C code optimization reduces execution time (by 50%, i.e. average of 1.2 times slower)
- C code optimizations are difficult
  - Numerous possibilities
  - Significant re-organization of code required
  - No generic algorithm for optimization
- C62x assembly code from TI: Arithmetic, filters, FFT/DCT, Viterbi decoders, matrices

  http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm

  http://www.ti.com/sc/docs/dsps/hotline/techbits/c6xfiles.htm

Conclusion

- Web resources
  - comp.dsp newsgroup: FAQ www.bdti.com/faq/dsp_faq.html
  - embedded processors and systems: www.eg3.com
  - on-line courses and DSP boards: www.techonline.com
- References

  - R. Bhargava, R. Radhakrishnan, B. L. Evans, and L. K. John, "Evaluating MMX Technology Using DSP and Multimedia Applications," *Proc. IEEE Sym. Microarchitecture*, pp. 37-46, 1998. http://www.ece.utexas.edu/~ravib/mmxdsp/
  - B. L. Evans, "EE379K-17 Real-Time DSP Laboratory," UT Austin. http://www.ece.utexas.edu/~bevans/courses/realtime/
  - B. L. Evans, "EE382C Embedded Software Systems," UT Austin. http://www.ece.utexas.edu/~bevans/courses/ee382c/