RISC CPU Based Implementation of Digital Signal Processing Algorithms

Wonyong Sung

School of Electrical Engineering
Seoul National University

Contents

1. **Introduction**
   - Comparison of RISC CPU and DSP
2. **DSP Algorithm Optimization using ARM CPU**
   - Loop fusion
   - Loop unrolling
   - Merging arrays
   - Circular addressing
   - Other methods
   - CELP decoder and ADPCM recorder
3. **Performance Comparison of DSP and ARM9**
4. **MP3 Implementation using ARM7 CPU**
5. **RISC CPU with SIMD Support**
6. **Conclusion**

1. Introduction

- **Why DSP processing using RISC CPU?**
  - Performance increase of RISC CPU's
    - By clock speed increase
    - By including multiple buses (separate cache and data buses)
    - Multiplier included
- **Diversified applications**
  - Many applications need some amount of control style programs, which are not efficient in DSP
    - Large memory size needed
    - Good compiler needed
- **Many embedded SOC using RISC core (ARM or MIPS)**
    - ARM based SOC from Intel, Samsung, Motorola
    - MIPS based SOC from Toshiba, NEC

❖ Disadvantages of RISC CPU

- Limited number of buses (one unified bus or only one data bus) -> Harvard architecture, 64 bit buses
- General purpose orthogonal instructions
  - Code density is usually poor.
- No specific data format for signal processing (no saturation or rounding…)
- No direct memory accesses (except for load, store instr.)
- Large delay in context switching (interrupt service)
- Large power consumption (unified register file, large instruction width…)

Comparison of RISC and DSP CPU's

RISC CPU

- Small number of instructions.
- Simple instruction format.
- Small number of addressing modes
- Mostly single cycle instructions (>75%)
- Load/store machine
- Hardwired controller
- Good compiler

Programmable DSP

- Many and application specific instructions

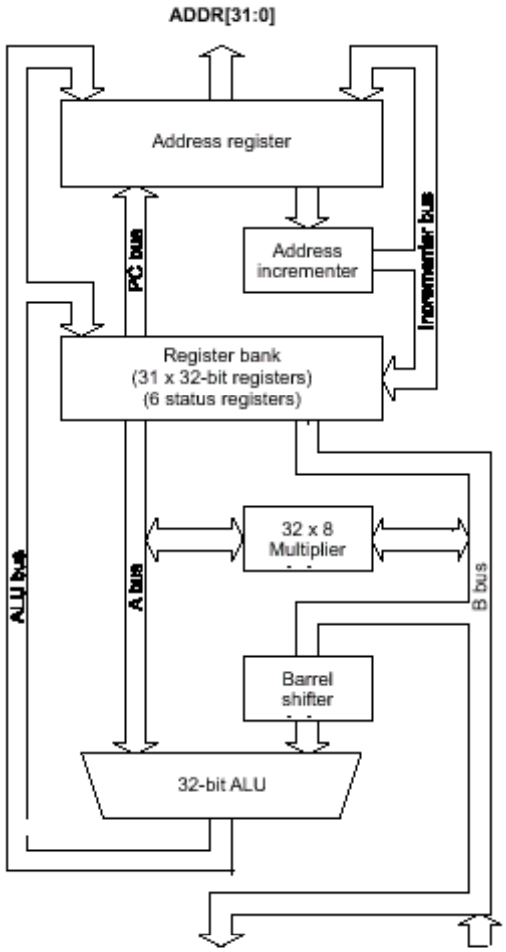
  ex) FFT (bit reversed addressing.

  Viterbi (CSSU)
- Zero-loop overhead

  ex) RPT,RPB
- Address generation units

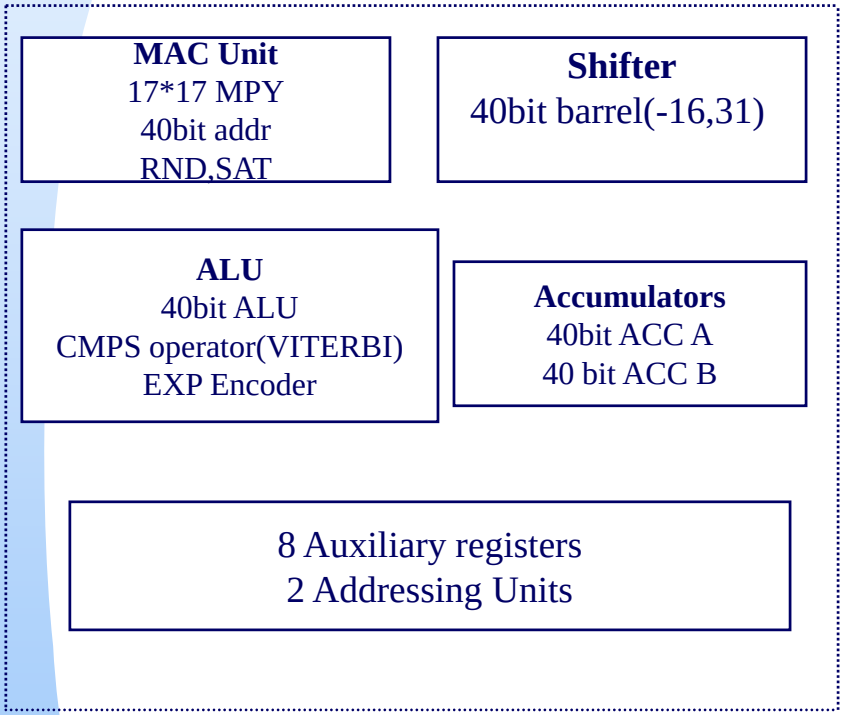
  ex) circular addressing
- MAC
- Distributed registers and special functional units

ARM 7TDMI CPU block diagram



- Von Neumann Architecture
(Unified Instruction/Data BUS)
- Multiplier (32*8)
- MAC(Multiply and Add :MLA)
- Barrel Shift
- 32bit ARM/16bit Thumb Instruction
- 31 32-bit general purpose register

TMS320C54x block diagram



54x block diagram

- Harvard Architecture
  (Separate Instruction /Data Bus)
- Multiplier (17*17)
- two independent 40 bit accumulators
- 40 barrel shift
- single cycle MAC(Multiply and Add)
- CSSU (compare, select and store unit) for Viterbi
- single-instruction repeat and block-repeat operations
  (RPT, RPTB)
- circular addressing (BK: block size)

2. DSP Algorithm Optimization using RISC CPU

- **For FIR filtering with RISC CPU, What are needed?**
  - Reducing the loop overhead
    - By loop unrolling
  - Reduce memory accesses
    - Use register data as much as possible to reduce the number of loads and stores
    - Loop fusion and array merges
  - Multiply minimization
    - Now many RISC contain HW multipliers
  - Word-length optimization (16bit if possible)
    - multiply cycle reduction according to the coefficients accuracy
  - Use RISC CPU specific instructions
    - LDM, STM (load multiple, store multiple)

2.1 Loop unrolling

- **Conduct multiple iterations in one loop cycle to reduce the number of loop repeats -> less loop overhead (less number of conditional jumps, counter decrease)**
- **Less number of loads and stores possible when the bus width is bigger than the data width, especially for MMX based implementation**

Tv : short (2bytes)
int(4bytes)

```
for ( i = 0 ; i < SubFrLen ; i ++ )
    TmpVect[i] = Tv[i] ;
```

→

```
for ( i = 0 ; i < SubFrLen/4 ; i ++ ){
    TmpVect[I*4] = Tv[i] ;
    TmpVect[I*4+1] = Tv[I*4+1] ;
    TmpVect[I*4+2] = Tv[I*4+2] ;
    TmpVect[I*4+3] = Tv[I*4+3] ;
}
```

2.2 Loop fusion

- **Merges multiple loops having the same loop length, and (possible) reusing the loaded variables.**
- **DSP implementation does not care about the number of loads and stores.**
- **RISC CPU needs to reduce the number of loads and stores – register based implementation**
- **PrevLsp : load (3->1), store (2->1)**
- **LspDcTable: load (3->1), Lsp : load (2->1) , store(2->1)**

```
for ( j = 0 ; j < LpcOrder ; j ++ )
  PrevLsp[j] = sub(PrevLsp[j], LspDcTable[j] ) ;
for ( j = 0 ; j < LpcOrder ; j ++ ){
  Tmp = mult_r( PrevLsp[j], Lprd ) ;
  Lsp[j] = add( Lsp[j], Tmp ) ;}
for ( j = 0 ; j < LpcOrder ; j ++ ) {
  PrevLsp[j]=add(PrevLsp[j],LspDcTable[j] );
  Lsp[j] = add( Lsp[j], LspDcTable[j] ) ;}
```

→

```
for ( j = 0 ; j < LpcOrder ; j ++ ){
    z=LspDcTable[j];
    x = sub(PrevLsp[j], z ) ;
    Tmp = mult_r( x, Lprd ) ;
    y = add( Lsp[j], Tmp ) ;
    PrevLsp[j]= add( x, z ) ;
    Lsp[j] = add( y, z ) ;  }
```

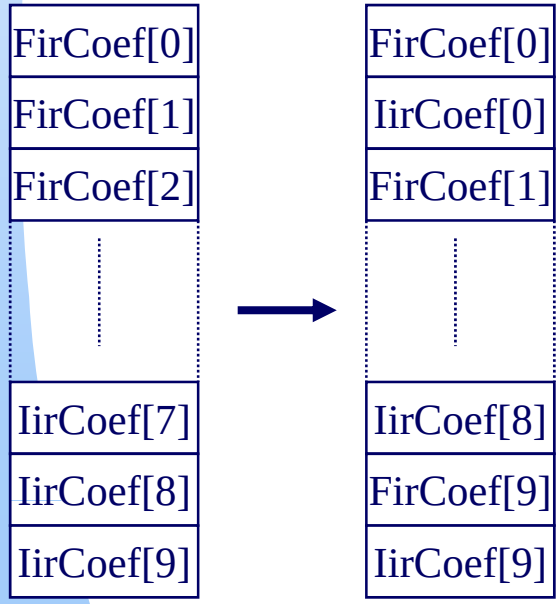
* Good for memory addressing based implementation

*Good for register based implementation

2.3 Merging arrays

- **Rearrange the data in the order of accesses**
- **Reducing cache misses by increasing the spatial locality (reduce the number of working sets)**

FirCoef[10],IirCoef[10] -> Fir_IirCoef[20]

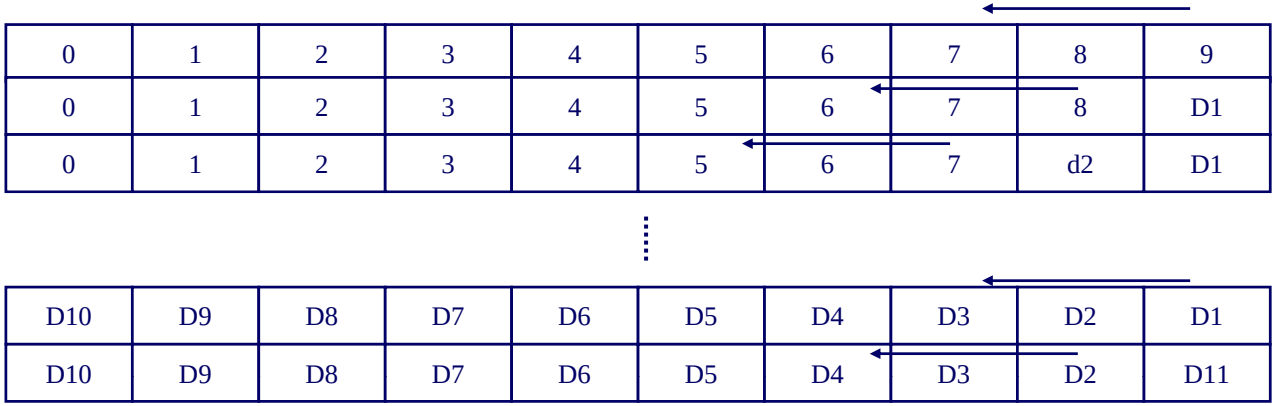



```
for ( i = 0 ; i < LpcOrder ; i ++ ) {
    FirCoef[i] = mult_r( Lpc[i], PostfirFiltTable[i] ) ;
    IirCoef[i] = mult_r( Lpc[i], PostiirFiltTable[i] ) ;
    }
```

↓

```
for ( i = 0 ; i < LpcOrder ; i ++ ) {
    Fir_IirCoef[i*2] = mult_r( Lpc[i], PostFiltTable[i*2] ) ;
    Fir_IirCoef[i*2+1] = mult_r( Lpc[i], PostFiltTable[i*2+1] ) ;
    }
```

2.4 Circular addressing method



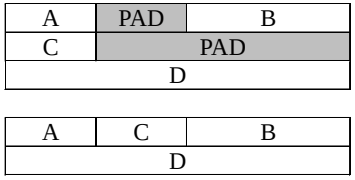

```
for ( j = 0 ; j < LpcOrder ; j ++ )
    Acc0 = L_mac( Acc0, Lpc[j], SyntIirDl[j] ) ;
for ( j = LpcOrder-1 ; j > 0 ; j -- )
    SyntIirDl[j] = SyntIirDl[j-1] ;
    SyntIirDl[0] = round( Acc0 ) ;
```

```
for ( j =LpcOrder-1 ; j >= 0 ; j-- )
   Acc0 = L_mac( Acc0, Lpc[j],SyntIirDl[j] ) ;
SyntIirDl[0] = round( Acc0 ) ;
```

2.5 Other methods

A. Data Alignment
- efficient memory use

```
char A;          char A;
short B;   →     char C;
char C;          short B;
int  D;          int  D;
```



B. Multiply by constant

- Multiply by 6

```
ADD Ra,Ra,Ra,LSL #1 ;Ra*3
MOV Ra,Ra,LSL #1    ;Ra*2
```

-Multiply by 10 and Add(Rc)

```
ADD Ra,Ra,Ra,LSL #2 ;Ra*5
ADD Ra,Rc,Ra,LSL #1 ;Ra*2+Rc
```

Example 1: FIR-IIR filtering(C)

- Formant postfilter (ARMA)

```
for(i=0;i < SubFrLen;i++)
{
      sum = In_data[i];
/* Fir Part */
      for(j=0;j<LpcOrder;j++)
            sum -= FirCoef[j]*PostFir[j]; MAC operation
      for(j=LpcOrder-1;j>0;j--)
            PostFir[j]=PostFir[j-1]; Data delay
      PostFir[0] = In_data[i];
/* Iir part */
      for(j=0;j<LpcOrder;j++)
            sum += IirCoef[j]*PostIir[j];
      for(j=LpcOrder-1;j>0;j--)
            PostIir[j]=PostIir[j-1];
       PostIir[0] = sum;
}
```

-Loop fusion,
-Merging arrays
-Loop unrolling
-Circular addressing

Combined method

(Loop fusion,circular addressing,loop unrolling,merging array)

```
/* FIR part */
    for ( j = 0 ; j < LpcOrder ; j ++ )
      Acc0 = L_msu( Acc0, FirCoef[j], PostFirDl[j] ) ;
    for ( j = LpcOrder-1 ; j > 0 ; j -- )
      PostFirDl[j] = PostFirDl[j-1] ;
/* IIR part */
    for ( j = 0 ; j < LpcOrder ; j ++ )
      Acc0 = L_mac( Acc0, IirCoef[j], PostIirDl[j] ) ;
    for ( j = LpcOrder-1 ; j > 0 ; j -- )
      PostIirDl[j] = PostIirDl[j-1] ;
```

```
for ( j = LpcOrder-1 ; j >= 0 ; j -- ) {
      Acc0=L_msu( Acc0, Fir_IirCoef[j*2], PostFir_IirDl[j*2] ) ;
      Acc0= L_mac( Acc0, Fir_IirCoef[j*2+1], PostFir_IirDl[j*2+1] ) ;
}
```

Example1 : FIR -IIR filtering(ASM)

° ARM7TDMI

```
Spf5
      LDRSH    a2,[v8]
Spf6
      LDR          a3,[v6],#-4
      MOV          a4,a3,ASR #16
      MOV          a3,a3,LSL #16
      MOV          a3,a3,ASR #16
      LDR      v1,[v3],#-4
      MOV          v2,v1,ASR #16
      MOV          v1,v1,LSL #16
      MOV          v1,v1,ASR #16
      MUL          v1,a3,v1
     SUB           a2,a2,v1
     MLA           a2,v2,a4,a2
     CMP           v4,v6
     ADDEQ         v6,v6,#0x28
     LDR           a3,[v6],#-4
      MOV          a4,a3,ASR #16
      MOV          a3,a3,LSL #16
      MOV          a3,a3,ASR #16
      LDR      v1,[v3],#-4
      MOV          v2,v1,ASR #16
      MOV          v1,v1,LSL #16
      MOV          v1,v1,ASR #16
     MUL           v1,a3,v1
     SUB           a2,a2,v1
     MLA           a2,v2,a4,a2
     CMP           v4,v6
     ADDEQ         v6,v6,#0x28
     CMP           lr,v3
     BLT           Spf6
     ADD           v3,lr,#0x28
;***** Load DecStat.PostIir[1]
     MOV           v1,v6
     ADD           v2,v4,#0x28
     CMP           v2,v1
     SUBEQ         v2,v2,#0x22
     ADDNE         v2,v6,#0x6
     LDRSH         a4,[v2]
     MLA           a3,a4,v5,a3
     MOV           a3,a3,LSL #1
;***** Store PostFir[0],PostIir[0]
     MOV           a2,a2,LSL #16
     BIC           ip,ip,#0xFF000000
     BIC           ip,ip,#0x00FF0000
     ORR           a2,a2,ip
     STR           a2,[v6],#-4
     CMP           v4,v6
     ADDEQ         v6,v6,#0x28
     STRH          a4,[v8],#2
     CMP           v8,a1
     BLT           Spf5
```

-Loop Fusion,
-Merging Array,
-Loop Unrolling (x2)
-Circular Addressing

◦ TMS320C54x

```
;AR3=&iir_coef,AR4=&Fir_coef,AR5=&Post_Fir,AR6=&Post_iir

        STM     #_DecStat+190,AR5
        STM     #_DecStat+200,AR6
        STM     #(LpcOrder-1),BK
        STM     #SubFrLen,BRC
        RPTB    L25-1
        RPT     #LpcOrder-1
        MAS     *AR4+%, *AR5+%,A
        RPT     #LpcOrder-1
        MAC     *AR3+%, *AR6+%,A
L25:
        STH     B,*(_DecStat+181)
          STM       #_DecStat+182,AR3
        MVMM    SP,AR4
        MAR     *+AR4(#12)
          STH   A,*AR2+
```

RPT,RPTB,
circular addressing

Example : ADPCM
(Quantizer Scale factor Adaptation Part)

```
if(ap > (1 << 13)) al = (1 << 14);
else al = (ap << 1) ;
y = (al * yu + ((1 << 14) - al) * yl) >> 14;
```

° ARM7TDMI

```
;a2=ap , a3=al,v1=yu,v2=yl
MOV       a1,#0x2000          ;1 <<13
CMP       a2,a1               ;if(ap>(1<<13)) ??
MOVGT     a3,#0x4000          ;if(ap>al) al =1<<14
MOVLE     a3,a2,LSL #1        ;else al=ap <<1
SUB       a2,v1,v2            ;a2=yu-yl
MUL       a1,a2,a1            ;a1=al*(yu-yl)
ADD       a1,a1,v2,LSL #14 ;a1=a1+yl*(1<<14)
MOV       a1,a1,ASR #14     ;a1>>14
```

° TMS320C54x

```
;AR1=ap,AR2=al,AR3=yu,AR4=yl
 LD   #0x2000,A
 SUB  *AR1,A
 BC     L1,AGEQ
 ST     #0x4000,*AR5
 B      L2
L1:
  ADD  *AR1,1,B
  ST     BL,*AR5
L2:
  LD *AR3,A
  MPY *AR2,*AR3,A
  MPY  *AR2,*AR4,B
  ADD   B,A
  SUB    #0x4000,A
 ADD    A,14,B
```

Comparison of ARM CPU and TI DSP

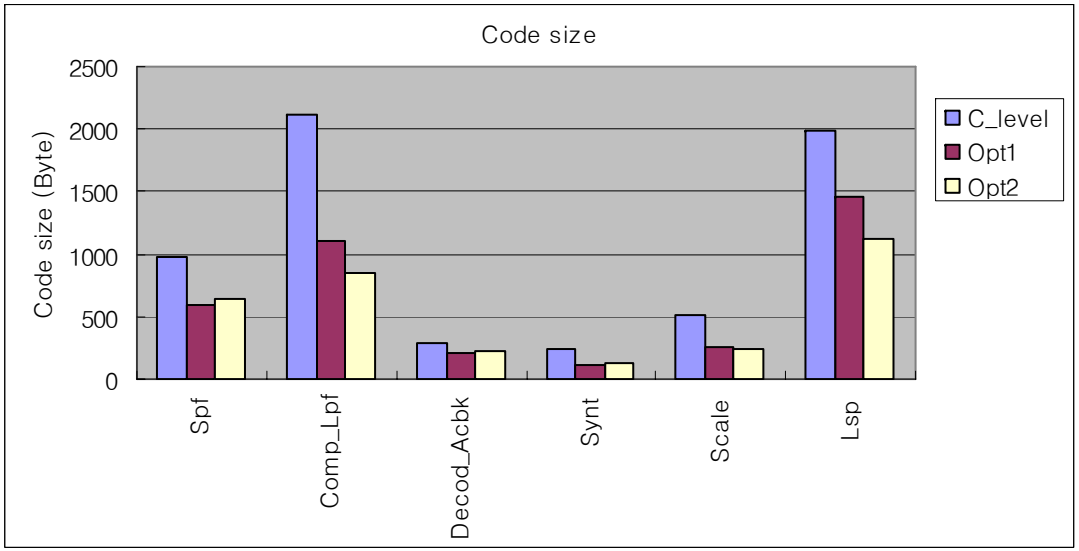
		TI DSP(Tms320c54x)		ARM(7TDMI)	
		C Code	Assembly Code	C Code	Assembly Code
ADPCM	Code size(b)	3.2k	3.1k	5k	2k
	Cycle/ Sample	12000	3200	4369	799
FIR	Code size*	22	16	60	60
	Cycle	5.06	1.11	6.09	5.13
FFT	Code size	7.1k	2.5k	11.8k	4.6k
	Cycle**	226134	15485	126883	45683
Viterbi	Code size	9.2k	5.4k	9.0k	14k
	Cycle/frame	338727	11845	220526	94298
DCT	Code size	1.5k	0.7k	1.3k	1k
	Cycle***	10656	4513	6607	4615
Synthesis (IIR)	Code size	136	64	248	136
	Cycle	420	55	520	140

* kernel code size
** 256 point complex FFT
*** 8*8 matrix

CELP decoder implementation and performance comparison

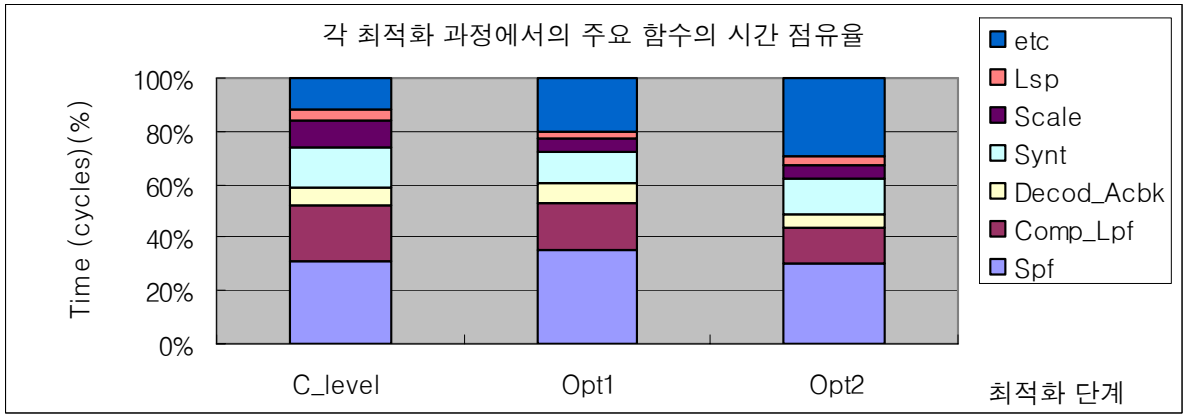
-Opt1 : loop fusion, Indirect addressing,
-Opt2 : loop unrolling,circular addressing, merging arrays

A. Code size of the blocks

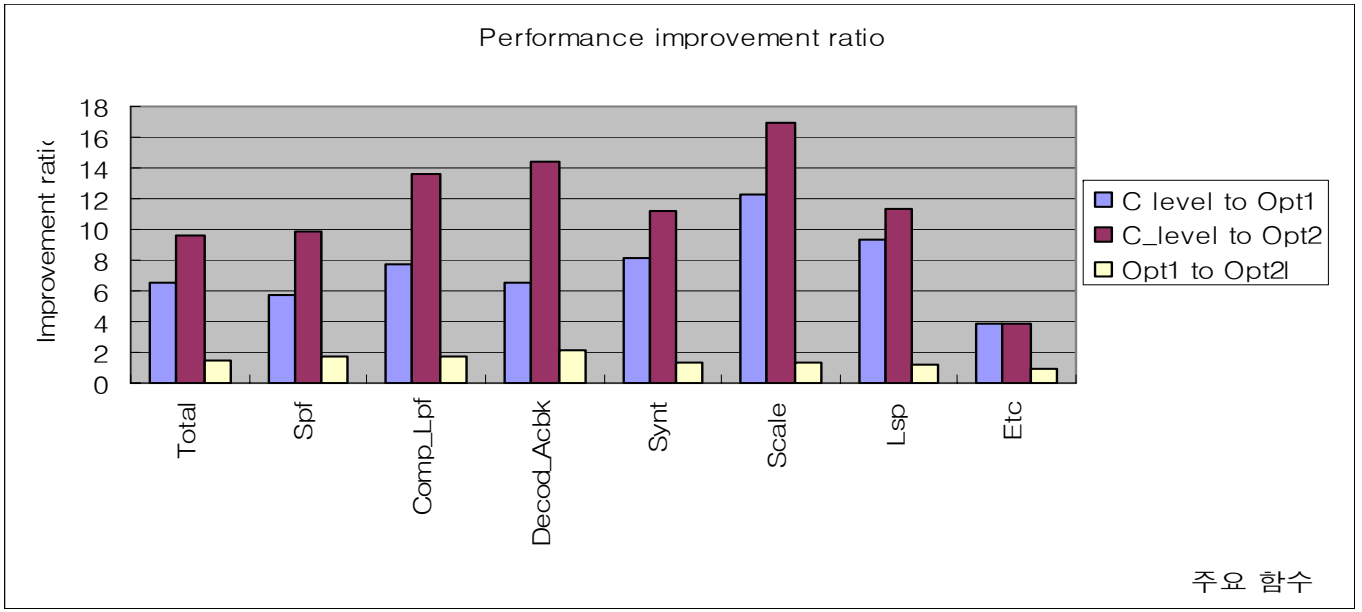


* C_level: 12.5kbytes
* Opt1 : 8.3kbytes
* Opt2 : 7.5kbytes

B. Ratio of execution time taken for the blocks



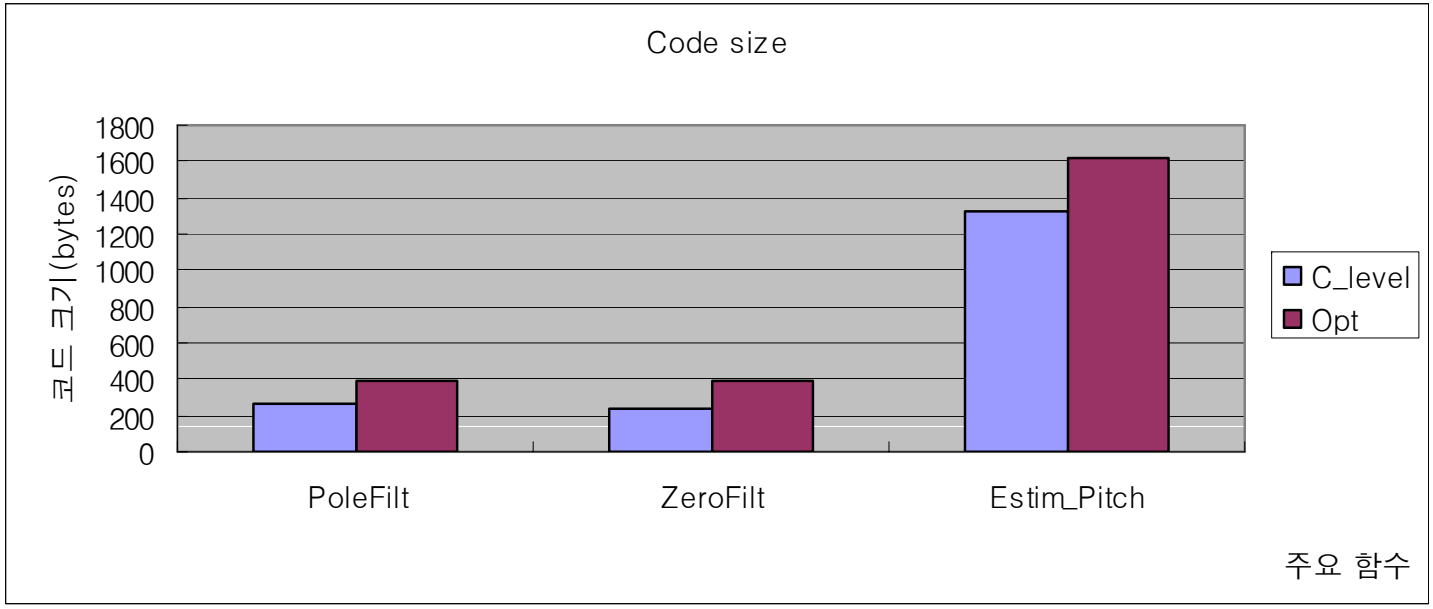
C. Performance improvement ratio



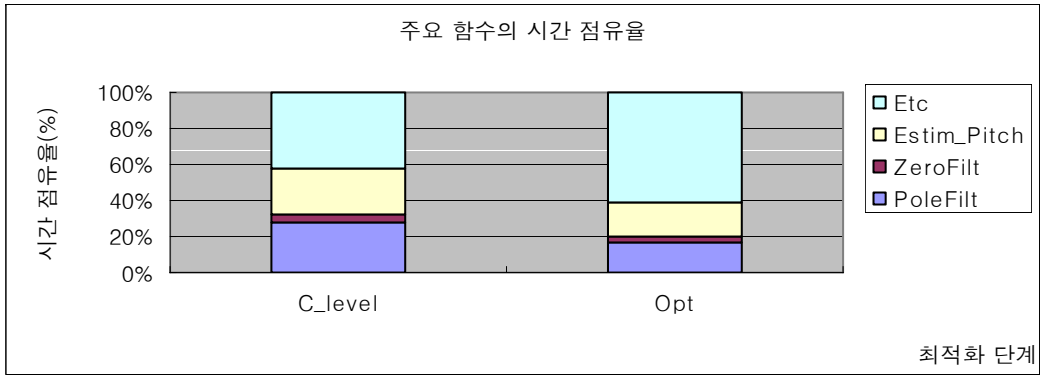
ADPCM Recorder

3.9.1 Code size comparison

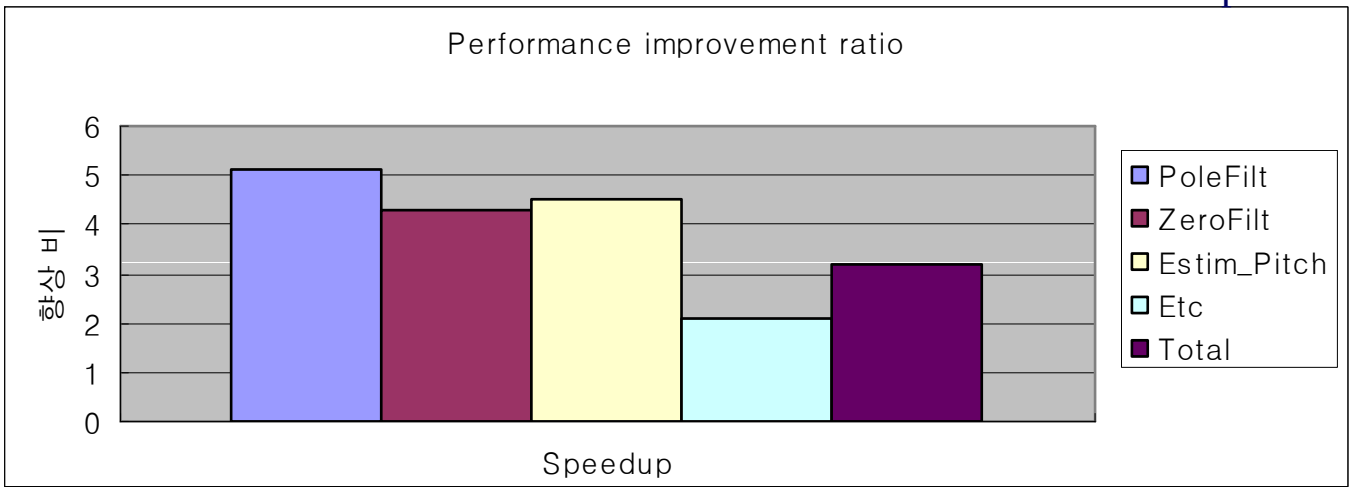
- Total code size : C_level :18.3Kbytes
  Opt :15.2Kbytes



3.9.2 Ratio of time taken before and after optimization



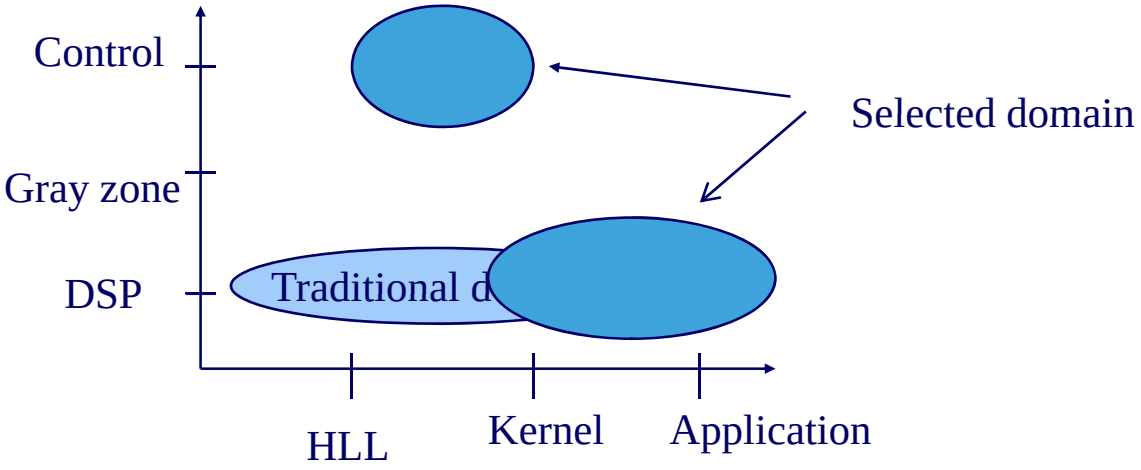
-Sample size : 68kbytes
- Sampling Freq : 12Khz
- 소요시간 : 2.84sec
- C_level : 6.82sec
- Opt : 2.2ses



3. Performance Comparison of DSP and ARM9 based Implementations

- **Title: A platform-based comparison between a digital signal processor and a general-purpose processor from an embedded systems perspective**
  - D. L. Cuadrado and et. Al. Aalborg Univ and Nokia
- **C55x and ARM9E-S**
- **Platform is defined as a processor and a compiler - compiled and evaluated (not assembly programming)**
- **C55: MAC/Dual MAC, specialized addressing, Viterbi**
- **ARM9E-s (ARMv5TE): includes DSP extensions to improve 16-bit fixed-point performance using a single cycle MAC. Support ARM (32bit) and Thumb (16bit) instruction set**

Selected case studies for comparison



- **HLL: high level construct:**
  - Simple code segments for demonstrating pointer addressing, function call, ...
- **Kernel: FFT, FIR, LMS, small state machines**
- **Application:**
  - GSM, CVSD

Case studies and types

Case study	Type	Size
Matrix functions (AC, FS)	DSP	Kernel
Dhrystone (DM)	Control	Kernel
CVSD (CE, CD)	DSP	Application
Viterbi (VD) algorithm	Control	Kernel

Matrix funcitons: autocorrelation, forward substitution

Dhrystone benchmark: a synthetic benchmark (1988)

assignments (51%), control statements (23%), procedure, call (17%)

Cycle count comparison in compiler environments

better than the C55x. Finally, the ARM9E-S performs for Viterbi decoding around 45% better. Even though the C55x has hardware support for Viterbi decoding, the compiler is unable to take advantage of it.
The greatest improvements of the ARM over C55x are observed in the control oriented applications.

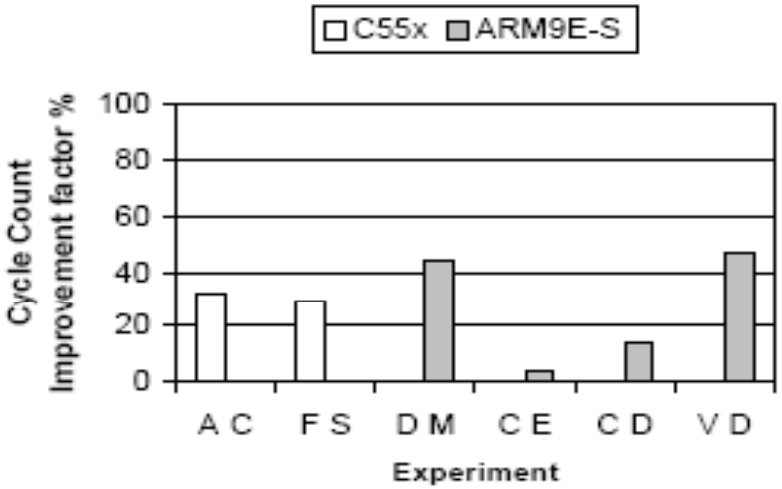


Figure 3. Cycle count improvement factors.

Code size comparison

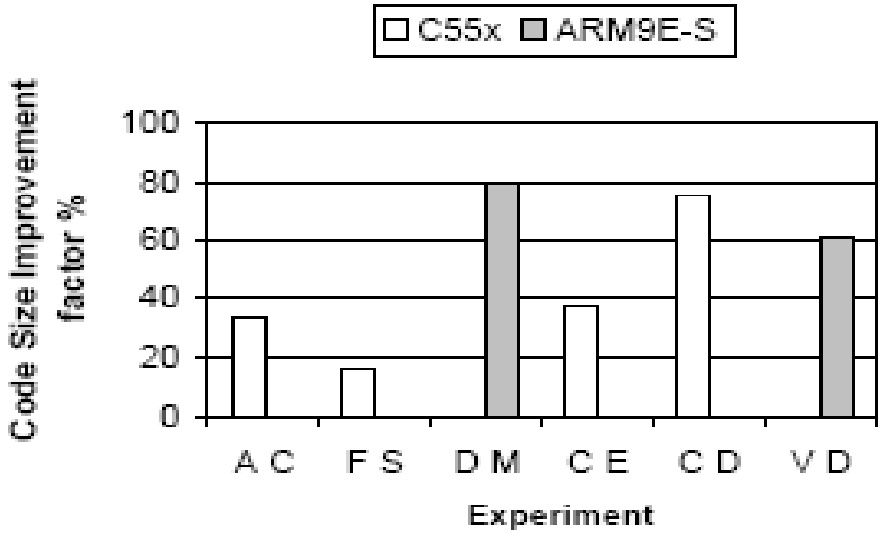


Figure 4. Code size improvement factors.

On the CVSD, the ARM9 outperforms the C55x in cycle count. An investigation of the assembly code generated by the compilers reveals that both compilers take advantage of the processors' conditional instruction execution capabilities.

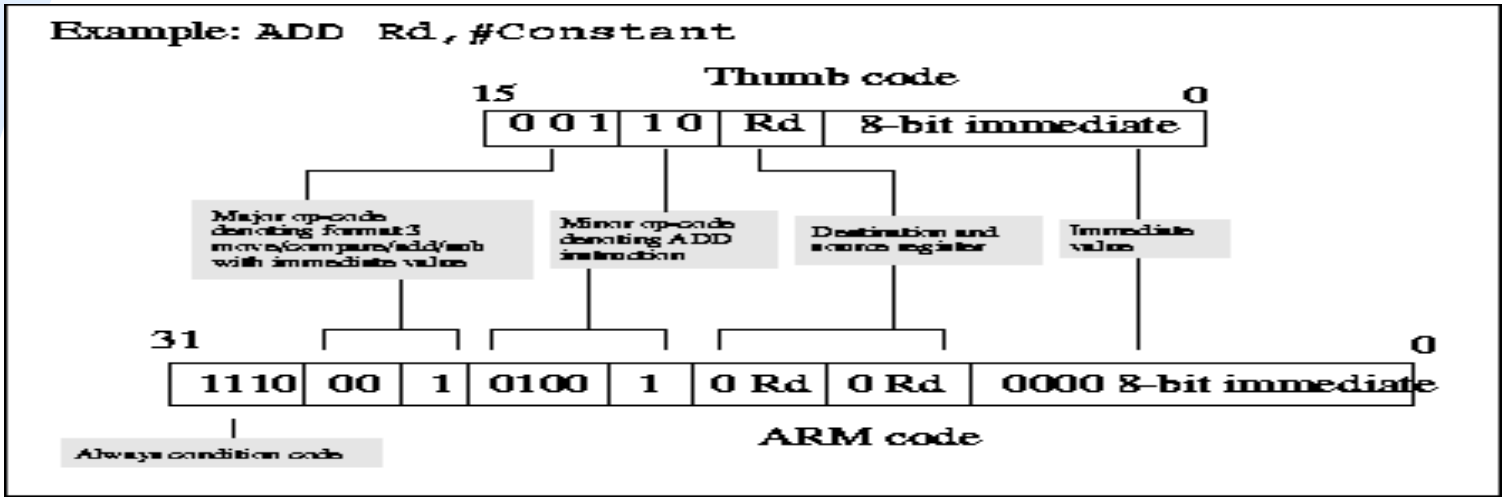
The C55x compiler does not take advantage of the C55x's parallel instruction execution ability on this benchmark.

The ARM9 benefits strongly from its large homogeneous register file characteristic of GPPs, avoiding the C55x's problem of burning cycles on restoration of its register contents caused by its small, heterogeneous register file.

The ARM9 assembly code contains more branches than the C55x, which also takes advantage of its block repeat feature. However, the number of cycles spent on branches on the two processors is about evened out, because the C55x incurs much stiffer branch penalties than the ARM9.

In conclusion, the C55x spends more cycles than the ARM9 primarily because the C55x must perform housekeeping tasks restoring register contents. Features that give the C55x potential to outperform the ARM9, such as parallel instruction execution, are not exploited by the compiler.

ARM vs Thumb instruction set



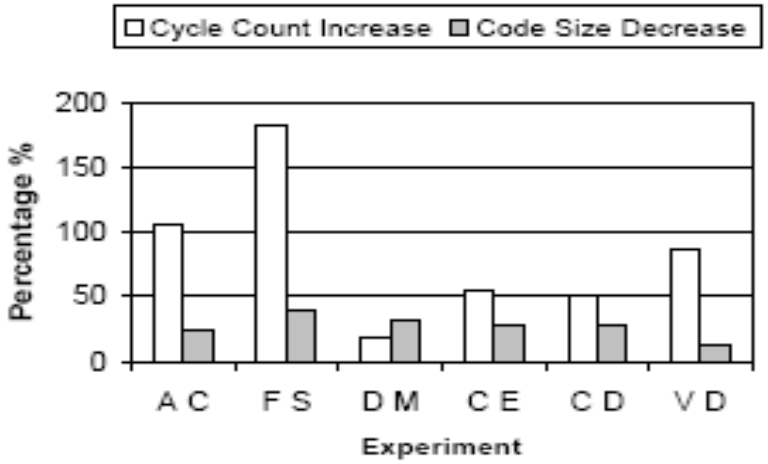


Figure 5: ARM vs. Thumb instruction sets comparison.

- Branches are more limited in the Thumb
- Data processing instructions are fewer (only one multiplication in Thumb, while 14 in ARM)
- Limited access to 8 of the 16 registers
- Single and multiple loads can access only 8 registers

4. MP3 Implementation Using ARM7 CPU

- **MP3 market**
  - Sigmatel: DSP56000 based SOC
    - Sigmatel is good at analog technology (codec, DC-DC)
  - Cirrus Logic, Telechips and et al: ARM7 or ARM9 based SOC
    - Integrates codec, internal memory (64 KB or so), NOR flash for code
  - Need to support multiple audio standards in these days
    - MP3, WMA, ..
- **Wonchul Lee, Kisun You and Wonyong Sung, "Software Optimization of MPEG Audio Layer-III for a 32bit RISC Processor," *in Proc. IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS),* 2002, vol. 1, pp. 435-438.**

Floating-point profiling results

- The profiling results in a PC show that the Subband synthesis and IMDCT parts take about 84%.
- These parts have DSP kernel like characteristics.

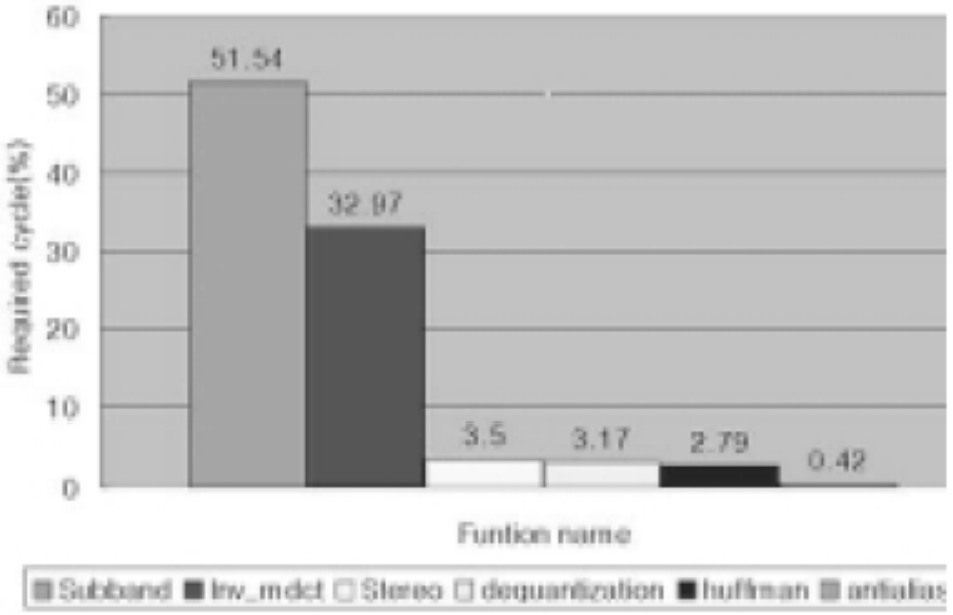
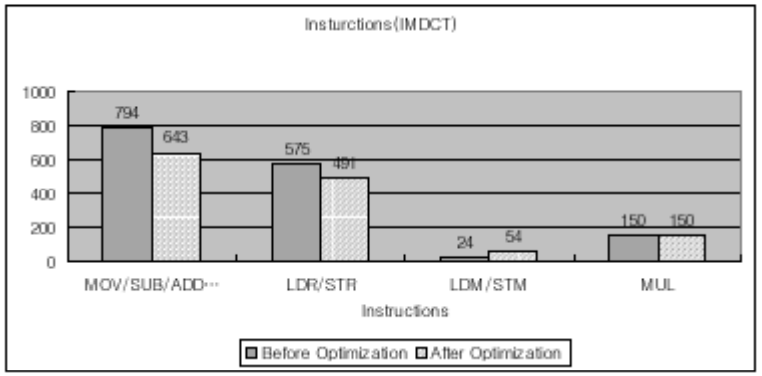


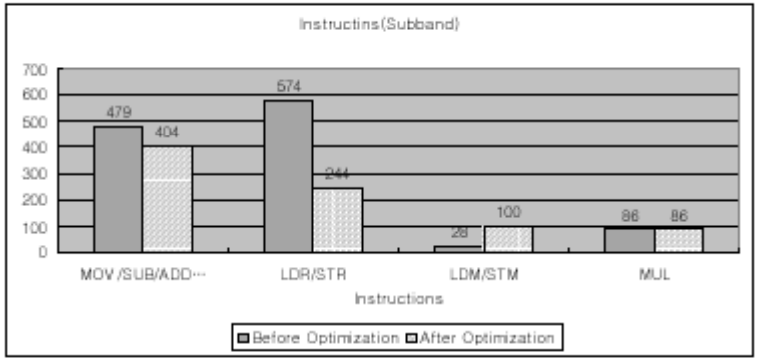
Fig.1. Floating-point profiling of MPEG audio Layer-III

Assembly program based optimization

- **LDM/STM: block transfer of upto 15 registers**
  - Compiler do not use these instructions excep for context switching
  - Takes 15 sequential, 1 non-seq, 1 internal cycle
  - Repetitive 15 execution 15*2N cycles
  - 21%, 25% decrease in the number of clock cycles for IMDCT and Subband synthesis



(a) IMDCT



(b) Subband Synthesis

Fig 2. Instruction types in (a) IMDCT, (b) Subband synthesis

Effects of multiply accuracy reduction

- **ARM7TDMI has 32*8bit multiplier**
  - 32*16 takes 4 cycles
  - 32*32 takes 6 cycles
- **32*16 implementation requires 8% less cycles**
- **The SQNR is 82dB while 32*32bit yields 91dB.**

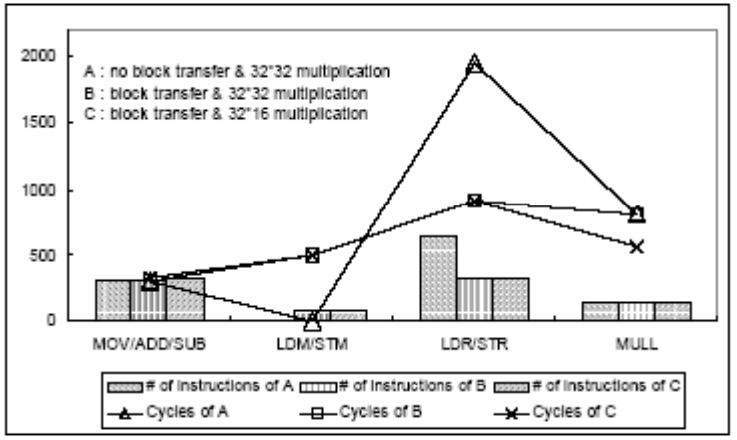


Fig. 3. Number of instructions and cycles in IMDCT function.

8.1% lower than 'B'. Comparing 'A', 'B', and 'C' of Fig. 3, we can find that the effect of reducing the data memory movement is greater than that of reducing the precision for multiply. This means that the optimized MPEG audio program is not a multiplication intensive problem, but rather an access bound program. As expected, the SNR of

Cache misses

- Data cache misses are dominant because program behavior is very predictable in MP3 program
- MP3 SOC usually uses internal ROM for code and RAM for data, instead of cache.

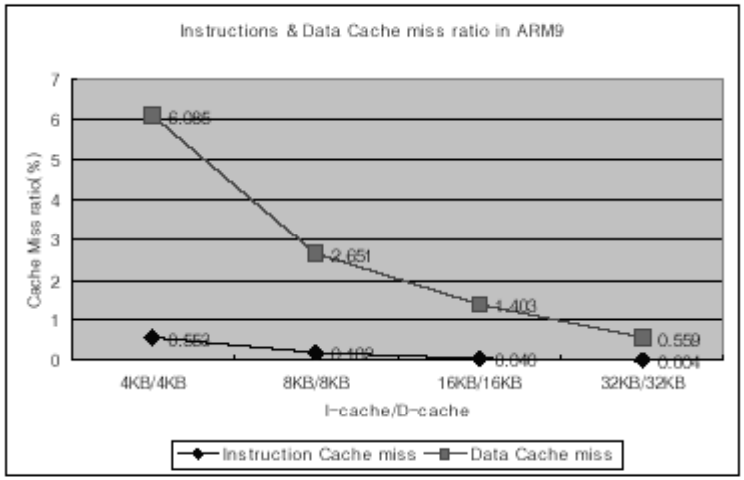
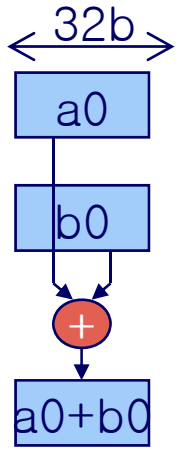


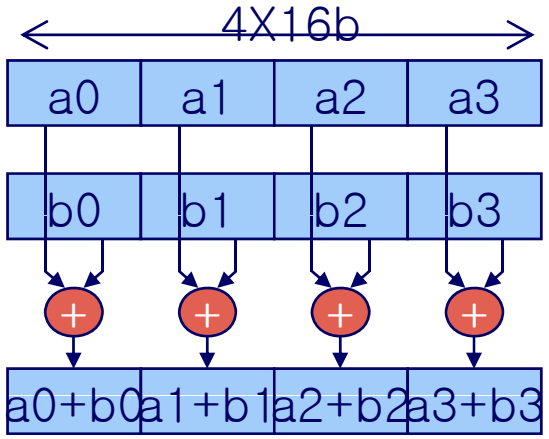
Fig. 3 instruction and data cache miss varying the cache size in ARM9

5. RISC CPU with SIMD Support

- **SIMD architecture**
  - Single instruction multiple data
  - Implementations
    - Intel Pentium MMX/SSE, Intel Xscale WMMX, ARM 11, AMD 3DNOW, PowerPC AltiVec, SUN SPARC VIS, TI C6000, ...
  - Exploit data parallelism
    - same operation with multiple data



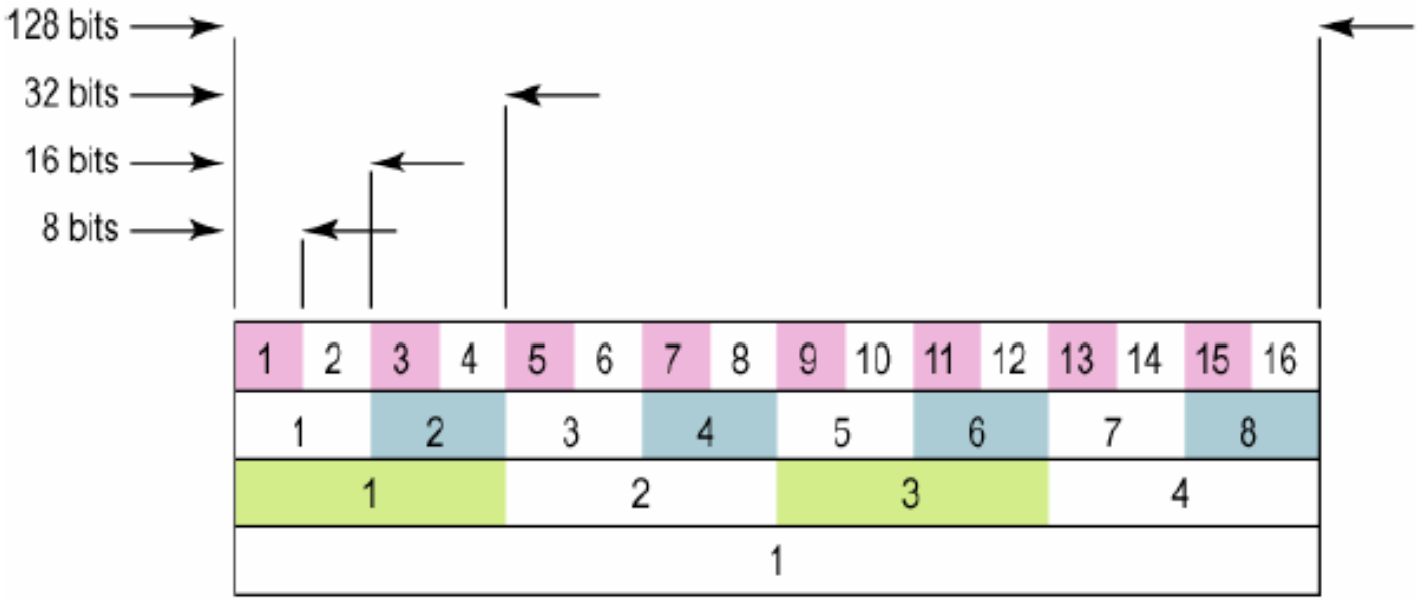
Conventional ALU structure



Partitioned ALU structure

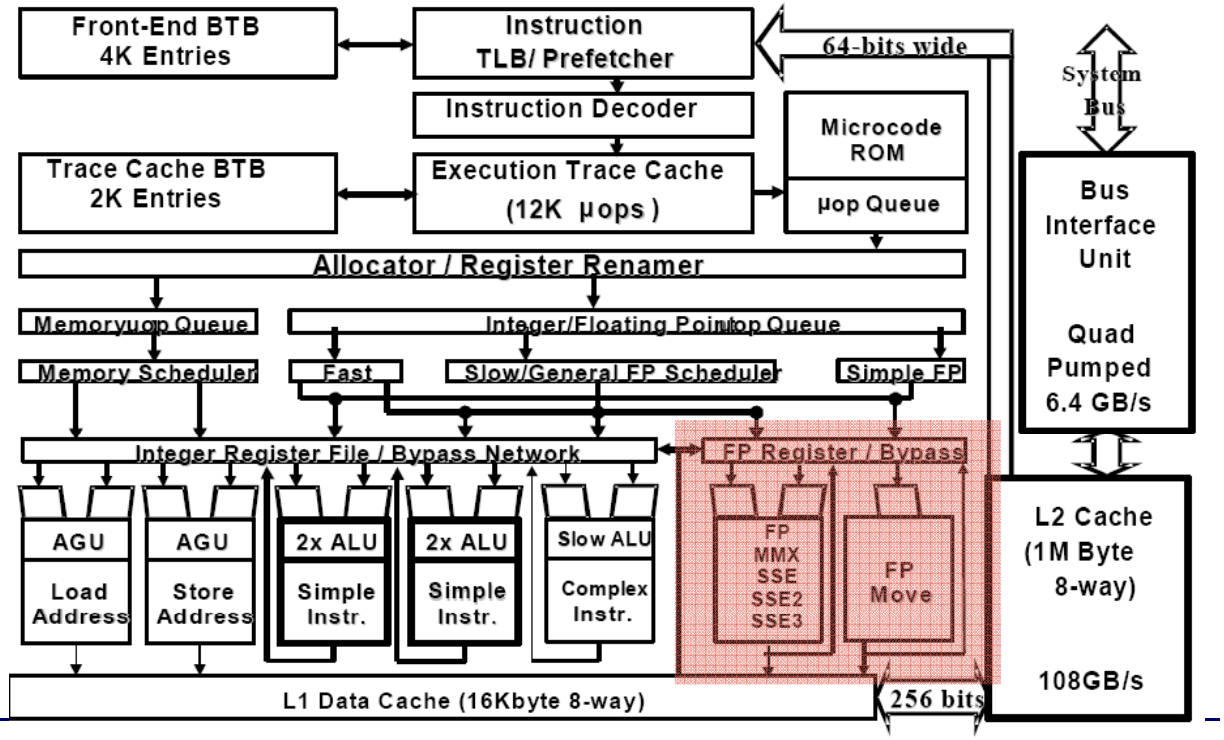
SIMD Introduction

- Operation with packed data
  - A wide SIMD register holds multiple data
    - Compatible with existing data-path
  - 2X64b, 4X32b, 8X16b, 16X8b



SIMD architecture example (Intel Pentium 4 SSE3)

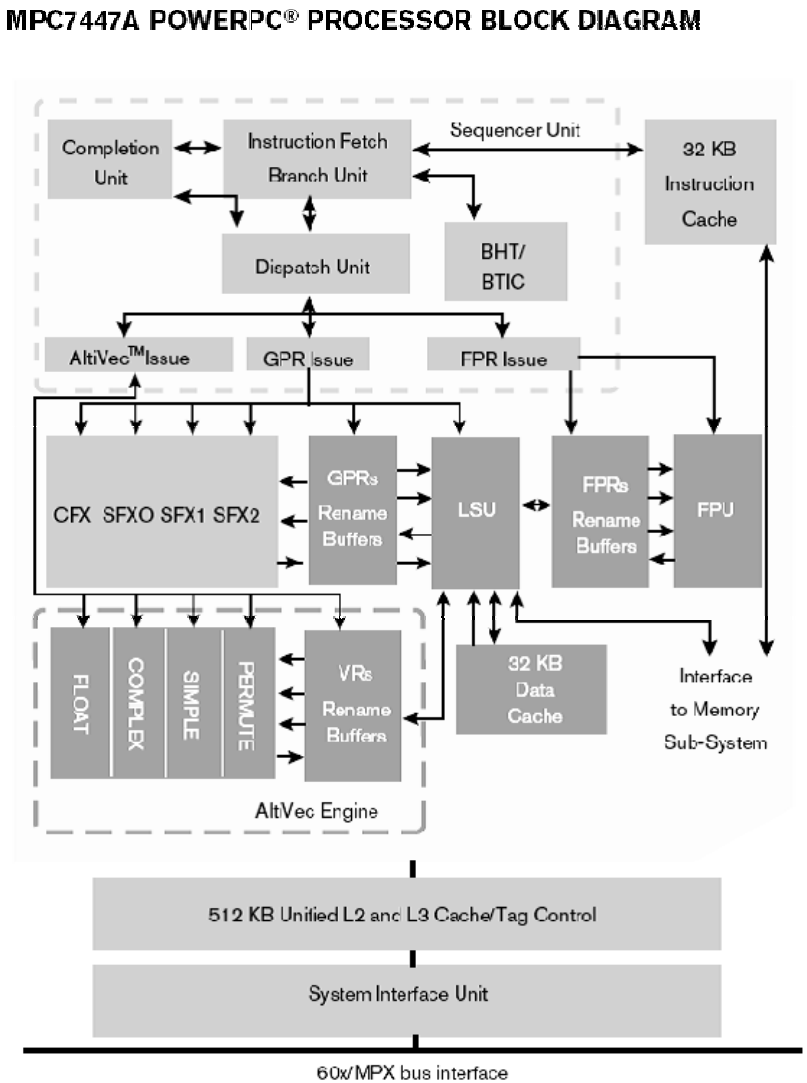
- 128b partitioned ALU, 8 128b registers
- H/W Prefetch unit, S/W prefetch inst.
- Unaligned load/store inst.



SIMD architecture example (Motorola PowerPC AltiVec)

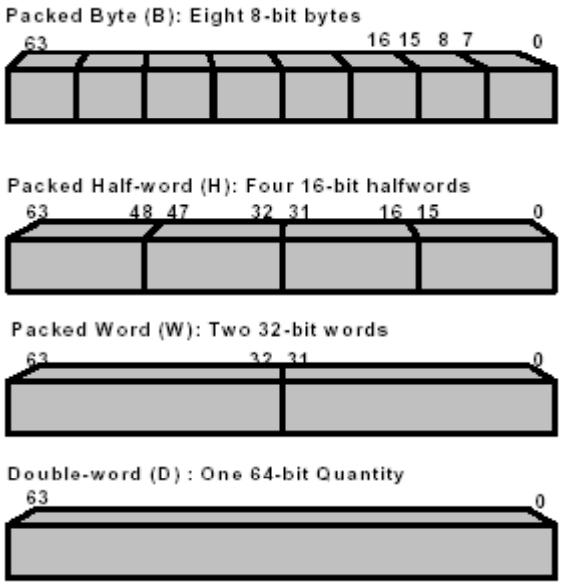
- 128b partitioned ALU
- 32 128b registers

- Sum-across inst.: sum all element in vector
- Stride-N access prefetch
- Unaligned memory access by aligned load/store and permute inst.



Intel® Wireless MMX™

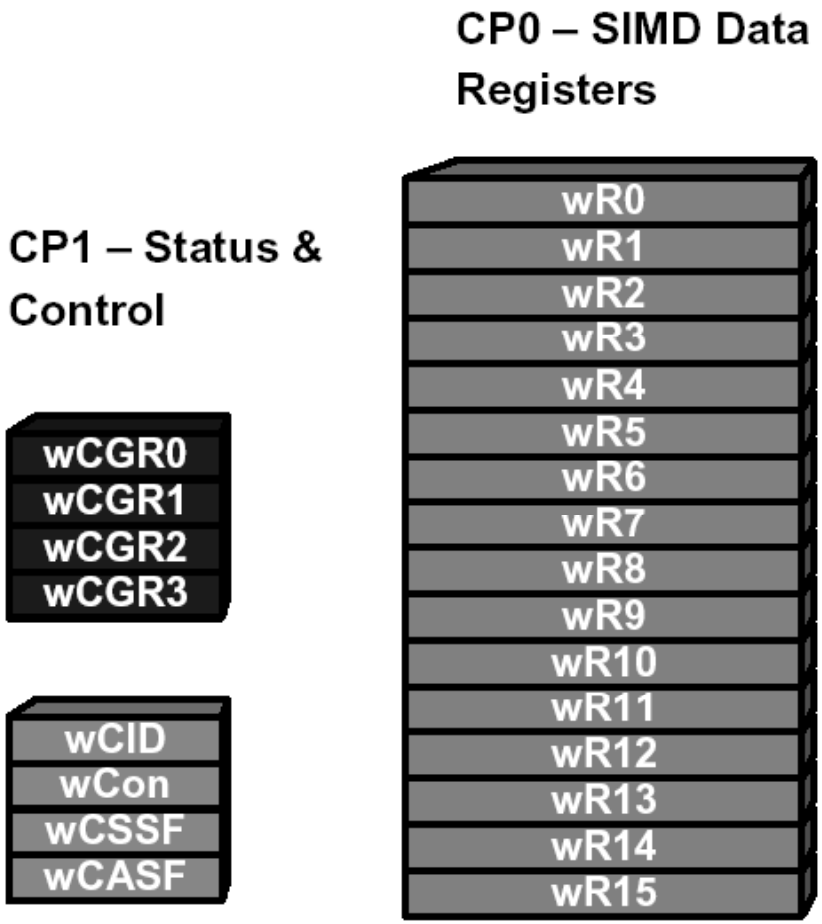
- **Wireless MMX™ Technology Mechanism**
  - It exploit the data parallelism by executing the same operation on different data elements in parallel. This is accomplished by packing data elements into a single register and introducing new types of instruction to operate on packed data.
  - Wireless MMX™ Technology Data Types



Intel® Wireless MMX™

- **Mapping Wireless MMX™ technology onto the ARM architecture**
  - The Wireless MMX™ technology utilize two ARM coprocessors; coprocessor 0 and coprocessor 1.
  - These coprocessors support Wireless MMX™ technology data and control registers using standard coprocessor transfer instructions.
  - Two coprocessor space is mapped onto two register files.
    - A main register file, mapped onto coprocessor 0 space, is provided for holding 16X64-bit packed data.
    - 32-bit control register file, mapped onto coprocessor 1 space, is provided for auxiliary support functions.

Wireless MMX™ Register File Organization



Wireless MMX™ Instruction

- **Compatibility Instructions**
  - The main class of instructions in the Wireless MMX™ technology are the compatibility instructions.
  - Wireless MMX™ technology provides equivalent functionality to all the Intel® MMX™ instructions and integer instructions from SSE instruction group.
  - In particular they provide equivalent functionality to:
    - MMX™ technology
    - Integer Intel Streaming SIMD Extensions (SSE)
    - Intel® XScale™ microarchitecture media instructions

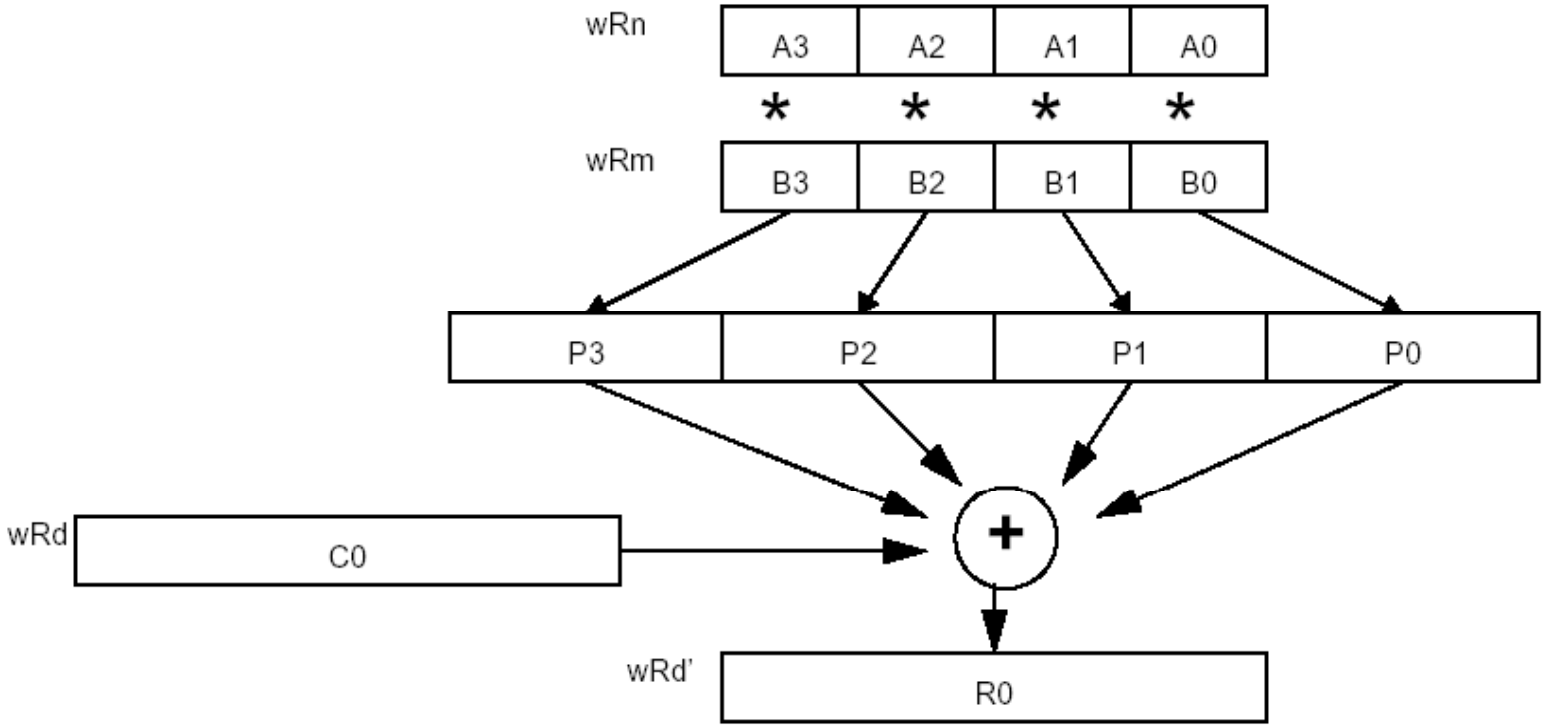
Wireless MMX™ Instruction

❖ New Wireless MMX™ Instructions

Instruction	Comments
WMAC	Multiply four signed or unsigned 16-bit half-words in parallel and accumulate with a 64-bit register.
WACC	Unsigned addition of eight bytes, four half-words, or two words.
WALIGN WALIGNI	Performs alignment on byte boundaries between two registers
TANDC TORC	Performs logical operations across the fields of the SIMD PSR (wCASF) and sends the result to the ARM* CPSR
TEXTRC	Extracts a 4-, 8-, 16-bit field specified by the 3-bit Immediate field data from the SIMD PSR register (wCASF)
TBCST	Broadcasts a value from the ARM* source register, Rn, or to every SIMD position in the Shortened Product Name Destination register, wRd; can operate on 8-, 16-, and 32-bit data values

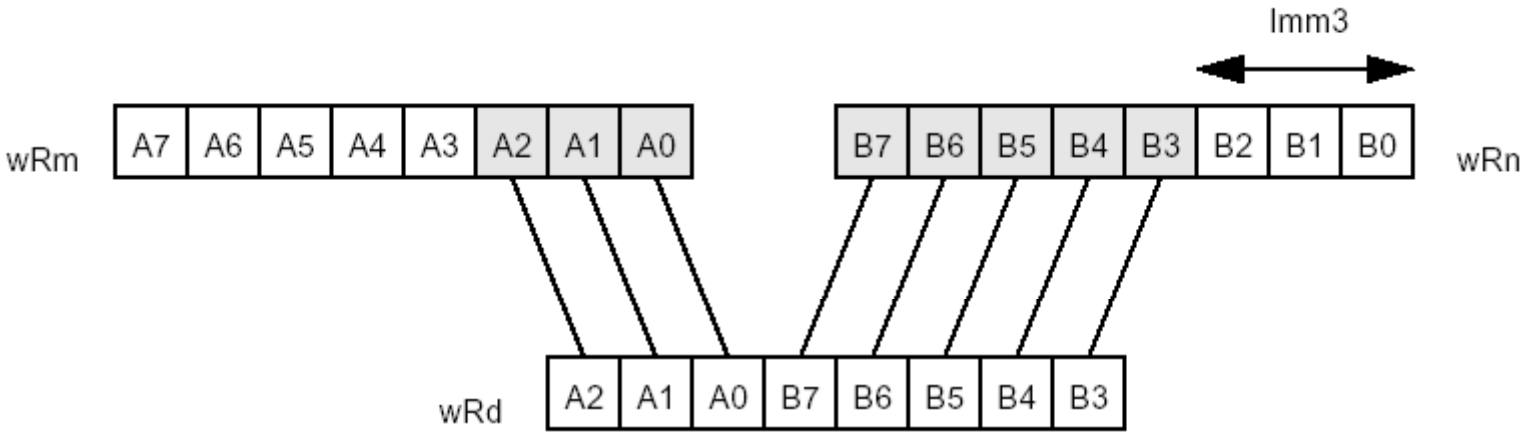
Wireless MMX™ Instruction

- **WMAC<U,S>{Z}{Cond} wRd, wRn, wRm**
  - Performs a vector multiplication of wRn and wRm and can accumulate the result with wRd on vectors of 16-bit data only.



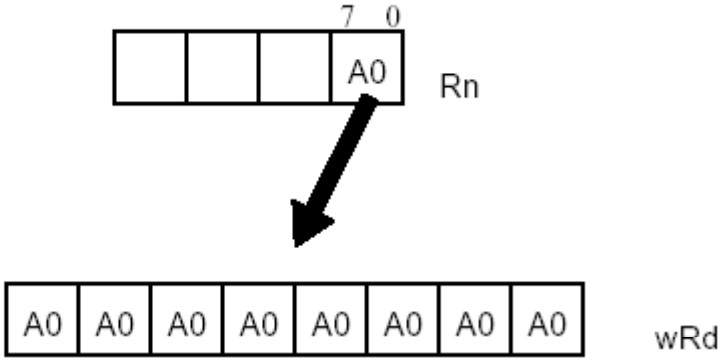
Wireless MMX™ Instruction

- **WALIGNI{cond} wRd, wRn, wRm, #Imm3**
  - Extracts an 64-bit value from the two 64-bit Source registers (wRn, wRm), and place the result in the Destination register, wRd.



Wireless MMX™ Instruction

- **TBCST<B,H,W>{Cond} wRd, Rn**
  - Broadcasts a value from the ARM* Source register, Rn, or to every SIMD position in the Wireless MMX™ technology Destination register, wRd; can operate on 8-, 16-, and 32.`TBCSTB wRd,Rn`



Wireless MMX™ Instruction

- **Transfer to and from Coprocessor Register**
  - The transfer instructions for moving data between the Wireless MMX™ technology control and data registers and the Intel® XScale™ microarchitecture registers.

Instruction	Comments
TMRRC	MRRC transfers the contents of the wRn 64-bit Wireless MMX™ technology data register to two ARM* core Destination registers (RdHi, RdLo)
TMCRR	TMCRR transfers the contents of the two ARM* registers (RdHi and RdLo) to wRd (the 64-bit Wireless MMX™ technology destination register.)
TMRC	TMRC transfers the contents of the 32-bit Wireless MMX™ technology control register (wCx) to the ARM* core Destination register
TMCR	TMCR transfers the contents of the Rn ARM* Core registers to a 32-bit wCx Wireless MMX™ technology control register
WLDR/WSTR	Data Load Store operation

Wireless MMX™ Instruction

- **Intrinsic support**
  - Most Wireless MMX™ instructions have a corresponding C intrinsic that implement that instruction directly.
  - Intrinsic function use a new C data type, the __m64 data type. The __m64 data type is used to represent the contents of Wireless MMX™ technology register.

Optimization Techniques

- **Instruction Scheduling**
  - Increasing Load Throughput
    - The buffering in the Memory pipeline allows two Load Double transactions to be outstanding without incurring a penalty(stall)
    - Back-to-Back WLDRD instructions will incur a stall, Back-to-Back WLDR(BHW) instructions will not incur a stall.
    - The WLDRD requires 4 cycles to return the DWORD assuming a cache hit, Back-to-Back WLDR(BHW) require 3 cycles to return the data.

Optimization Techniques

- Interleave other operation to avoid the penalty with successive WLDRD instructions.

```
WLDRD wR3,[r4],#8
WLDRD wR5,[r4],#8        @ STALL
WLDRD wR4,[r4],#8        @ STALL
WADDB wR0,wR1,wR2
WADDB wR0,wR0,wR6
WADDB wR0,wR0,wR7
```



```
WLDRD wR3,[r4],#8
WADDB wR0,wR1,wR2
WLDRD wR4,[r4],#8
WADDB wR0,wR0,wR6
WLDRD wR5,[r4],#8
WADDB wR0,wR0,wR7
```

- Always try to separate 3 consecutive WLDRD instructions so that only 2 are outstanding at any one time and the loads are always interleaved with other instructions.

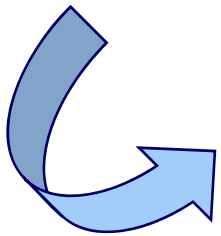
```
WLDRD wR0, [r2] , #8
WZERO wR15
WLDRD wR1, [r4] , #8
SUBS  r3, r3, #8
WLDRD wR3, [r4] , #8
```

Optimization Techniques

❖ SIMD Optimization Techniques

- Software Pipelining

```
for (i = 0; i < N; i++) {
    s = 0;
    for (j = 0; j < T; j++) {
    s += a[j]*x[i-j]);
    }
    y[i] = round (s);
}
```



```
    @Pointers r0 -> val , r1 -> pResult, r2 -> pTapsQ15 r3 -> tapsLen

        WLDRD     wR0, [r0], #8    @ Load first 4 input samples
        WLDRD     wR1, [r2], #8    @ Load first 4 coefficients
        WZERO     wR15
        WLDRD     wR2, [r0], #8    @ Load next 4 input samples
Loop_Begin:
        WLDRD     wR3, [r2], #8    @ Load next 4 coefficients
        WMACS     wR15,wR0,wR1
        WLDRDNE   wR0, [r0], #8    @ Load 4 input samples
        SUBS      r3,r3,#8
        WLDRDHS   wR2, [r2], #8
        WMACS     wr8, wr1, wr3
        WLDRDHS   wr1, [r1], #8
        BHS       Loop_Begin
```

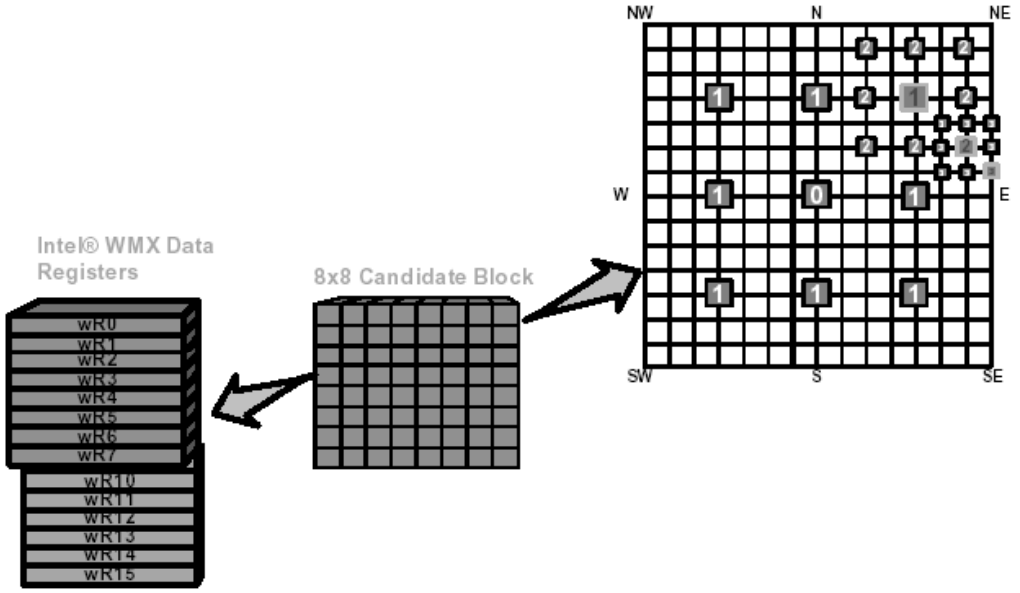
Optimization Techniques

- Multi-Sample Technique
  - Calculating multiple outputs with each loop iteration similar to loop unrolling.
  - C code for FIR filter with Multiple Samples for 8-taps per iteration

```
for (i = 0; i < N; i++4) {
   s0=s1=s2=s3=0;
   for (j = 0; j < T/4; j++4) {
      s0 += a[j]*x[i-j];
      s1 += a[j]*x[i-j+1];
      s2 += a[j]*x[i-j+2];
      s3 += a[j]*x[i-j+3]);
   }
   y[i] = round (s0);
   y[i+1] = round (s1);
   y[i+2] = round (s2);
   y[i+3] = round (s3);
}
```

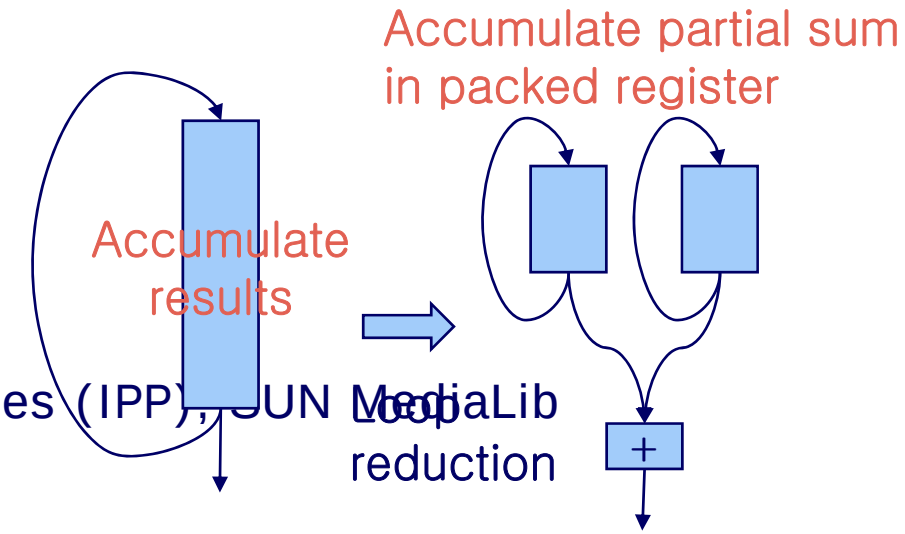
Optimization Techniques

- Register File Usage
  - With the large register file of the Wireless MMX™ technology it is possible to store large data structures in the register file and reduce memory load traffic.
  - Example: the register file is used to store the 8X8 pixel macroblock during a video encode motion sea



SIMD programming method

- **SIMD library**
  - Various signal processing kernels
  - Provided by processor makers or self-made
  - Various optimization techniques applied
    - Software pipelining
    - Data layout modification
    - Loop reduction
    - …



Ex) Intel performance primitives (IPP), SUN MediaLib

SIMD programming method

- **Intrinsic functions**
  - A function known by compiler
  - directly map to instruction

Ex) **`c = _mm_mulhi_pi16(a, b)`**

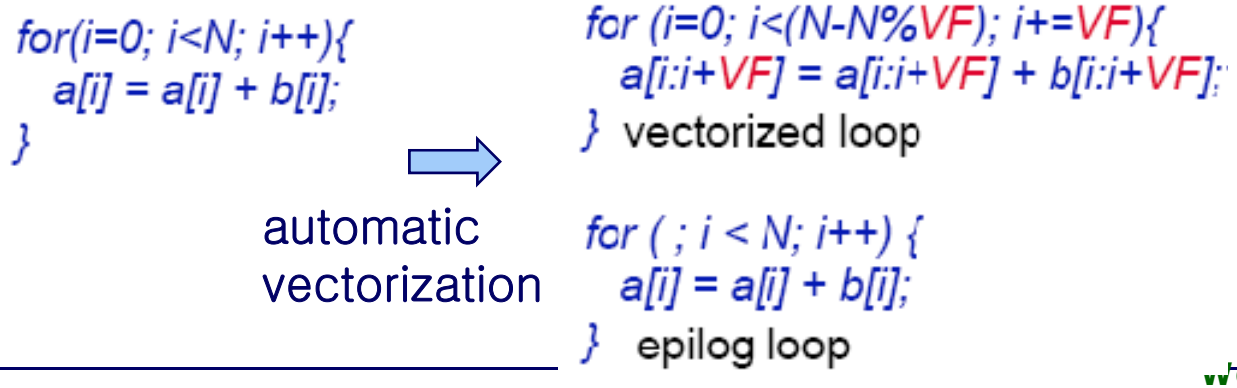
➔ **`PMULHW c, a, b`**

`c = _mm_unpackhi_pi32(a, b)`

➔ **`UNPCKHDQ c, a, b`**

SIMD programming method

- **C compiler**
  - Autovectorization : automatically transform serial code to vector code.
    - Identify loop
    - Memory reference analysis (access pattern, dependency, alignment)
    - Vectorize (loop unrolling, peeling, reduction, idioms)
  - Intel compiler, gcc v4.0



Performance of SIMD

- **Speed-up by SIMD extension**
  - Packed arithmetic inst. reduce FU inst.
  - Unrolled loop, saturation mode reduce branch inst.
  - Packed memory inst. reduce memory inst.

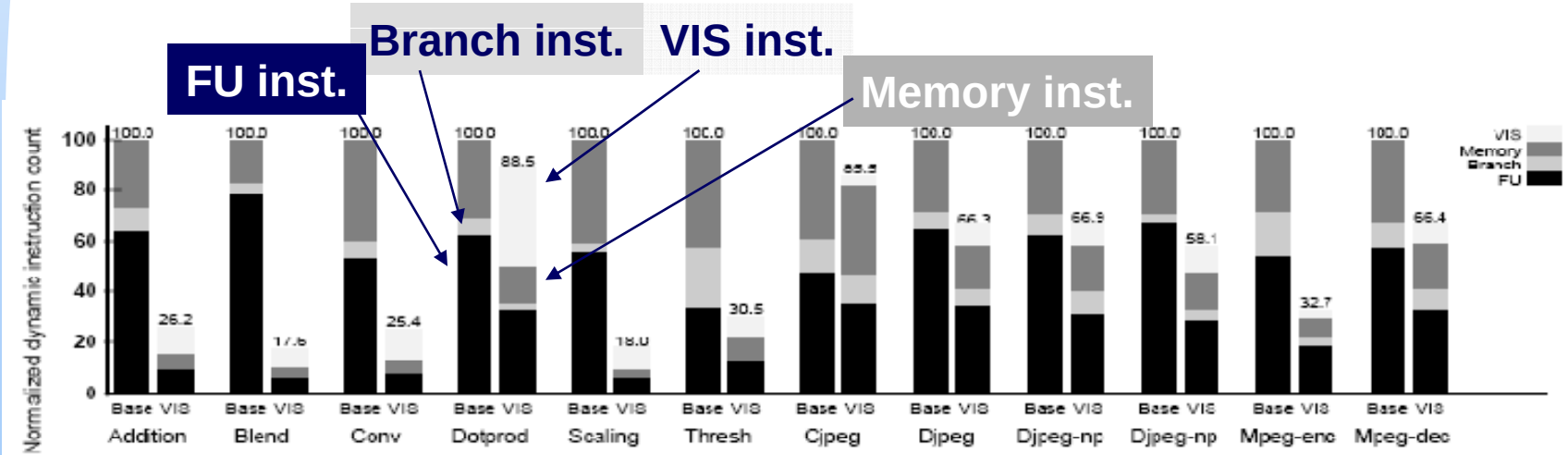
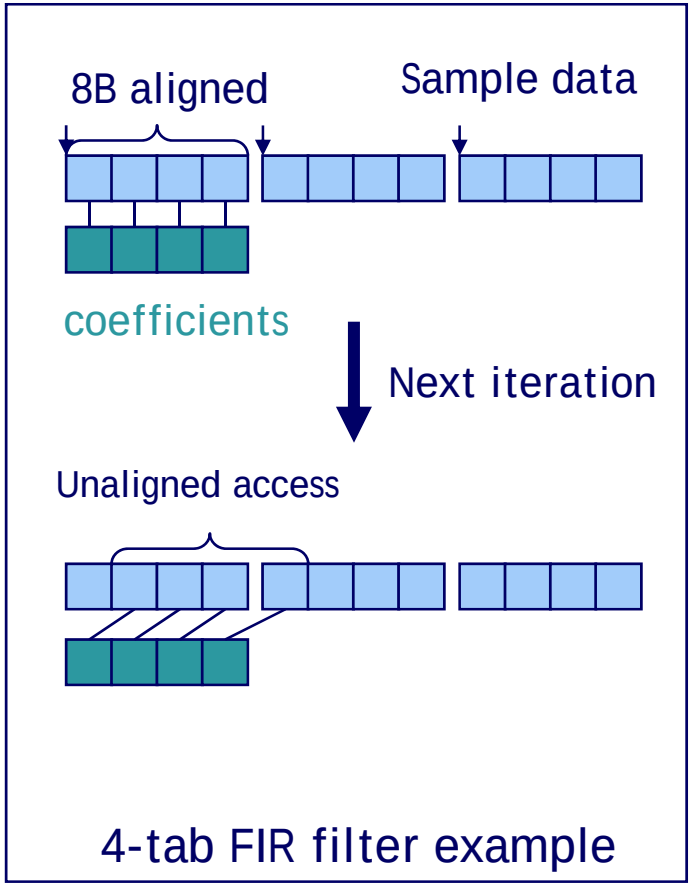


Figure 2. Impact of VIS on dynamic (retired) instruction count.

P. Ranganathan, S.V. Adve, N.P. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," *proc. ISCA*, 1999.

Performance bottlenecks

- **Not suitable to control-oriented software**
  - Ex) Huffman coding: inherently sequential, variable data size
- **Data rearrangement overhead**
  - Misaligned data
  - Stride > 1
  - Reordering
  - Ex1) FIR filter
  - Ex2) accessing RGB interleaved pixel data
  - Ex3) bit-reverse addressing of FFT



4-tab FIR filter example

6. Conclusion

- 1. Many DSP applications are nowadays implemented by RISC CPUs partly because of the performance increase of RISC processors (dual cache, wide bandwidth data bus, hardware multiplier)
- 2. RISC processors are advantageous for implementing control (branches) intensive, large memory size, and complex applications requiring compiler based development.
- 3. RISC CPU specific optimization methods can increase the implementation performance very much. DSP and RISC CPU needs different program optimization strategies.
- Example: CELP decoder

  - code size : 13K -> 7.5kbytes

  - speedup : Upto x5.

4. SIMD support closes the gap between RISC and DSP.