TMS320C54x, 55x Architecture and Programming

Wonyong Sung

School of Electrical Engineering
Seoul National University

Contents

1. Introduction
2. C54x Architecture and Instruction Set
3. Application Specific Instructions
4. C55x Architecture and Instructions
5. Compiler
6. Conclusion

1. Overview

- **C25, C50 DSP**
  - Optimized for FIR filtering, intended for 1 cycle operation for each tap of FIR filter
  - Poor performance for some special kind of DSP algorithms
    - Coefficient update for adaptive filtering
    - Viterbi decoding
    - FFT, linear phase FIR filter
  - Pipelining register between the multiplier and accumulator complicates the programming
  - One AGU and one data bus, which results in poor performance when not using ‘RPT’

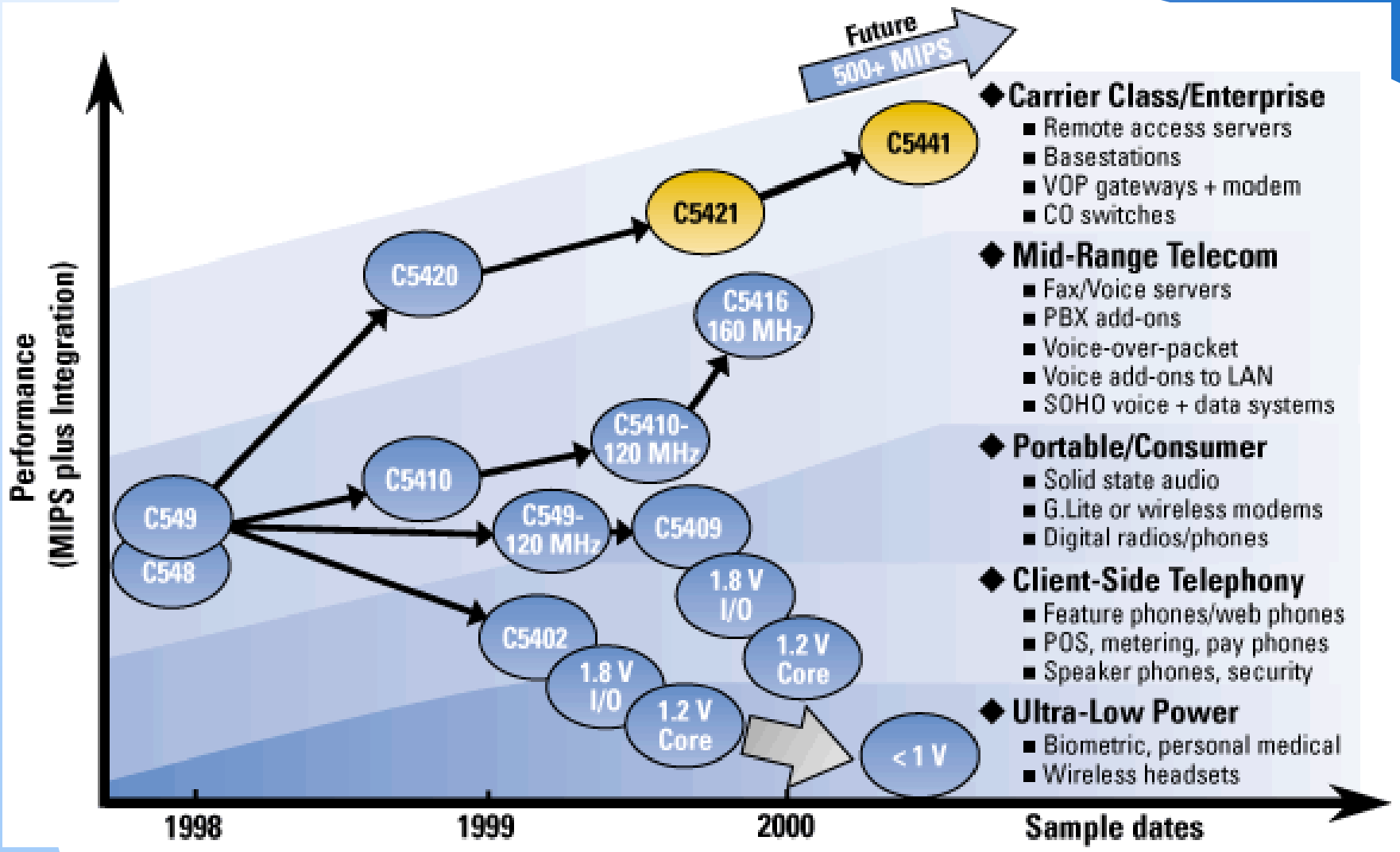
Enhancements in C54x

- **Multi-bus architecture: 3 data buses and one program bus**
- **One multiplier, two ALU (40bit), structure**
  - Conduct 2-tap linear-phase FIR filtering per cycle
- **Easier programming**
  - No pipelining register between the multiplier and adder
  - Guard bit for Accumulator (easier scaling)
- **Compare-select-store unit**
  - A special function unit for Viterbi decoding, pattern recognition (speech recognition)
  - N vector square distance : N cycle + alpha
  - Viterbi decoding : 4 cycle/bfy
- **Others**
  - Exponent encoder for block fixed-point arithmetic
  - Arithmetic instructions with parallel store and parallel load
- ***Although C54x architecture is more complex, its programming is easier than that using C25, 50 (less pipelining, enough data bus)**

Enhancements in C55x

- **32bit instruction bus and 24bit addr bus**
  - Mostly two parallel execution of 16-bit instructions (except for resource conflicts)
- **A unified program/data memory map. Program supports upto 16MB (24bit), data 8MB**
- **Two MAC (mul-acc) architecture (+40bit ALU, 16bit ALU, 40bit shifter)**
- **Three read bus and two write bus for data**
- **An instruction buffer and a separate fetch mechanism – decoupled instruction fetching**
- **Encouraging C based program developments**
  - C compiler friendly architecture (multiple accumulators,..)

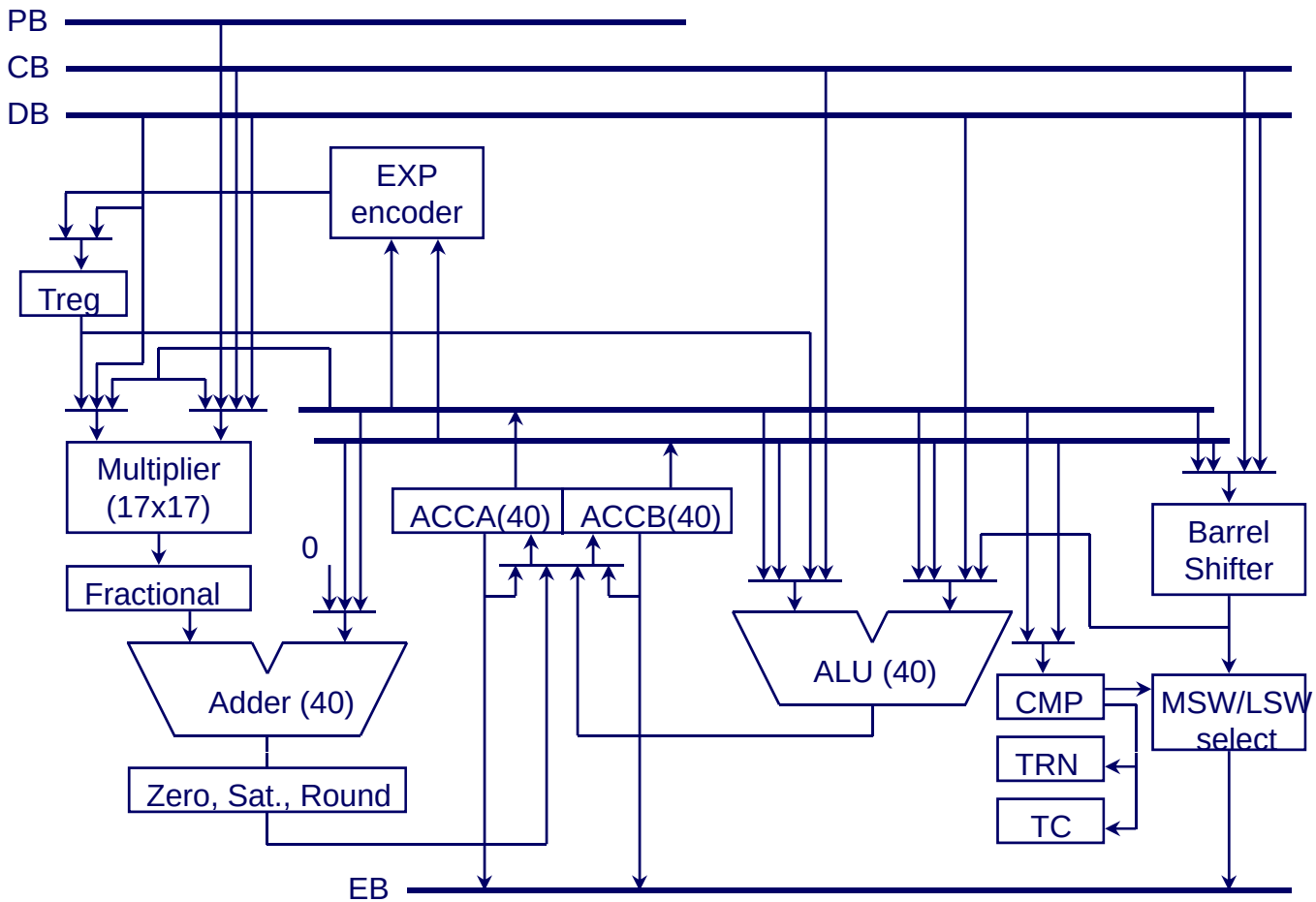
Many different versions



2. C54x Architecture and Instruction Set

- **16bit fixed-point DSP**
  - One 16bit multiplier
  - Two 40bit accumulators (A and B)
  - Accumulator based architecture
- **Four buses**
  - Three read buses: program, data, coefficient
  - One write bus: write back
- **Memory blocks (actual memory sizes differ ..)**
  - ROM in 4K blocks
  - Dual access RAM in 2k blocks
  - Single access RAM in 8k blocks
- **Two clock cycles per instruction cycle**

TMS320C54x data path



Notable C54 instructions

- Single and block repeat
- Vector instructions utilizing repeat
- Special instructions utilizing two ALU
- Special instructions utilizing CSS unit
- Instructions with 2- or 3-operand simultaneous reads
- Arithmetic instructions with parallel store and parallel load

C54x addressing modes (1)

- **Immediate addressing**
  - Operand is part of the instruction, and it can be short (3, 5, 8, 9 bits) or long 16bits (two words instruction)
  - LD #20, DP ; #20 -> DP
  - RPT #0FFFFh
- **Absolute addressing**
  - Address of the operand is part of the instruction. The address can be for a data memory (dmad), a program memory (pmad), a port (PA), or a location in the data space specified directly.
    - MVKD 1000h, *AR5; 1000h -> *AR5 (??)
    - LD *(1000h), A; *(1000h) -> A
- **Direct addressing**
  - Address of operand is part of the instruction (added to implied memory page). The 16 bit address of the data memory location is formed by combining the lower 7 bits of the data memory address contained in the instruction with a base address given by the data-page pointer (DP, when CPL=0) or stack pointer (SP, when CPL=1)
    - ADD 010h, A

Addressing modes (2)

- **Indirect addressing**
  - Address of operand is stored in a register
  - Use 16bit auxiliary registers (AR0 ~ AR7), two ARAU (ARAU0, ARAU1)
  - Offset addressing
    - ADD *AR1(10)
  - Register offset (AR1+AR0)
    - ADD *AR1+0
  - Auto increment and auto decrement
    - ADD *AR1+
  - Bit reversed addressing
    - ADD *AR1+B
  - Circular addressing
    - ADD *AR1+0%
  - Examples
    - *AR3+ (= addr<- AR3, then AR3 <- AR3 + 1)
    - *+AR3(-40h) (= AR3 <- AR3 - 40h,then addr<-AR3)
    - AR3 + % (= AR3 <- circ(AR3+1))
    - AR3 +0 % (= AR3 <- circ(AR3 + AR0))
    - *(lk) (= addr <- lk)
    - *+AR3(lk)% (=AR3<- circ(AR3+lk), addr<- AR3)

Addressing modes (3)

- **Circular buffer**
  - BK specifies the buffer size -1
  - A circular buffer must start on N-bit boundary; that is the N LSBs of the based address of the circular buffer must be 0.
    - e.g. a 48 word circular buffer must start at an address whose six LSBs are 0, BK<-47.
  - Example: Assume AR3=1020h, BK=40h, AR0=025h. What'll be the AR3 after the execution of LD *AR3 + 0%
    - Ans: circ (1020h + 25h)=1045h – 40h = 1005h (?? 1004h)

Bit reversed addressing for FFT

- **FFTs start or end with data in 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)
- **"Bit reverse" address addressing mode for use with autoincrement addressing**
  - AR0 specifies one half the size of the FFT
  - The next address is calculated by adding in a bit-reversed manner.

Addressing modes (examples)

- **MPY 13, B**
  - Multiply of T reg with data in data address 13, and the result is in ACC B.
- **MPY #01234, A**
  - Multiply of T reg with constant 1234, result is in ACC A.
- **MPY *AR2-, *AR4 + 0, B**
  - *(AR2)-> T || *(AR2)**(AR4) -> ACC B, then AR2 -1 -> AR2, AR4+AR0->AR4
- **MAS *AR3-, *AR4+, B, A**
  - *(AR3)->T || B - *(AR3)* *(AR4) -> A and then AR3 post dec, AR4 post inc.
  - This instruction is useful for FFT computation

Other features

- SSBX SSM

Program control

- **Conditional execution**
  - XC n, cond [,cond[,cond]]; 23possible conditions
  - Execute next n (1 or 2) words if conditions are met
  - Takes one cycle to execute
    - XC 1, ALEQ ; test for acc a<= 0
    - Mac *ar1+, *ar2+, a ; perform mac if a<= 0
    - Add #12, a, a ; always perform add

Program control (2)

- **Repeat single instruction or block**
  - Overhead: 1 cycle for RPT/RPTS and 4 cycles for RPTB
  - HW loop counters count down
  - Ex

```
rptz a, #39          ; zero acc a
Mac *ar2, *ar3, a  ; a+= a[n]*x[n]
```

Programming examples(1)

- **A = dmad(410h)+ dmad(411h) +..dmad(41fh)**

```
    STM #10H, AR2  ; AR2 = 10H
    STM #410H, AR1; AR1 = 410H
    LD #0H,A; ACC A = 0
    SSBX SXM; Select sign extension mode
Start:  ADD *AR1+, A;
    BANZ   Start, *AR2-
```

-
- **Homework, modify the code using RPT**

Programming examples(2)

- y[n] = h[0]*x[n]+h[1]*x[n-1]+h[2]*x[n-2]
  the lower 16bits is stored in y, and higher 16bits in y+1

- SSBX SXM
- STM #x, AR2
- STM #h, AR3
- LD #0H, A
- RPT #2
- MAC *AR2+, *AR3+, A
- STM #y, *AR2
- STL A, *AR2+
- STH A, *AR2+
- NOP
- .end

3. Application Specific Instructions

- **Vector arithmetic acceleration**
  - Each instruction operates on one element at a time
  - ABDIST absolute difference of vectors
  - SQDIST squared distance between vectors
  - SQURA sum of squares of vector elements
  - SQURS difference of squares of vector elements
  - Ex:
    - Rptz a, #39 ; zero accumulator a, repeat next instr. over 40 elements
    - Squra *ar2+, a ; a += x[n]**2

L_1 norm calculation (ABDST)

- ABDST Xmem, Ymem
- ABDST *AR3+, *AR4
- Operation: (B) + |A(32-16)| -> B
  ((Xmem) – (Ymem))<<16 -> A
- This instruction adds the absolute value of Acc A to B, while storing the difference of (Xmem) and (Ymem) to Acc A. 1word, 1cycle instruction.

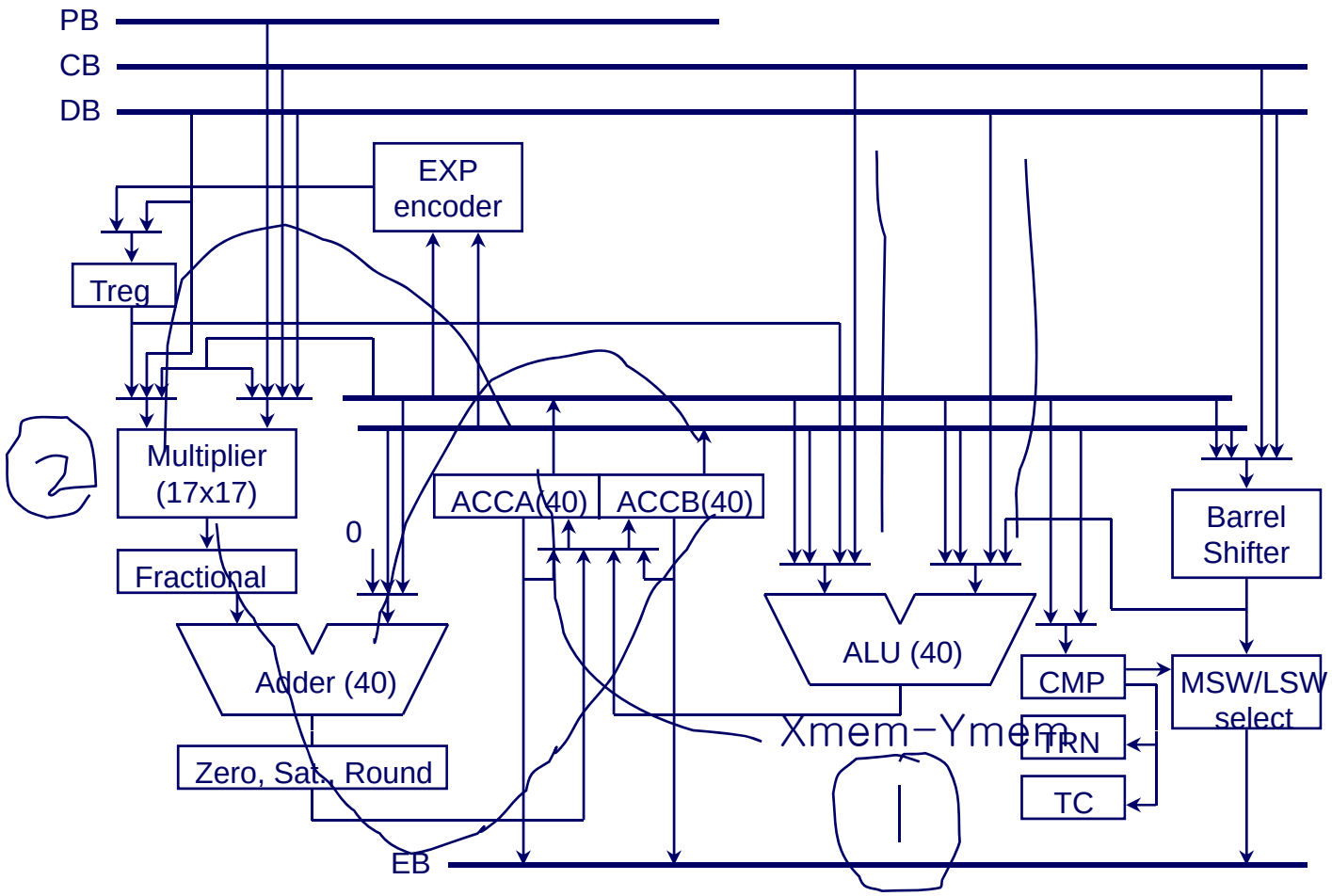
L_2 norm calculation (SQDST)

- **SQDST Xmem, Ymem**
- **SQDST *AR3+, *AR4+**
- **Operation:**

  (A(32-16))x(A(32-16))+B->B

  ((Xmem)-(Ymem)) << 16 -> A
- **Accumulates the square distance. 1 word, 1cycle.**

TMS320C54x data path



Symmetric FIR filtering (FIRS)

- **FIRS xmem, ymem, pmadd(16bit)**
  Ex: FIRS *AR3, *AR4, coeffs
- **Operation:**
  - pmad -> PAR
  - While (RC) .ne. 0
  - (B) + A(32-16)*(Pmem) -> B
    ((Xmem) + (Ymem) << 16 -> A
    (PAR)+1 -> PAR, (RC)-1 -> RC
  - This is a 2word, 3cycle instruction, but from 2nd repeat, this becomes a single cycle instr. **Program counter points the coefficient address.**

LMS: LMS coefficients update

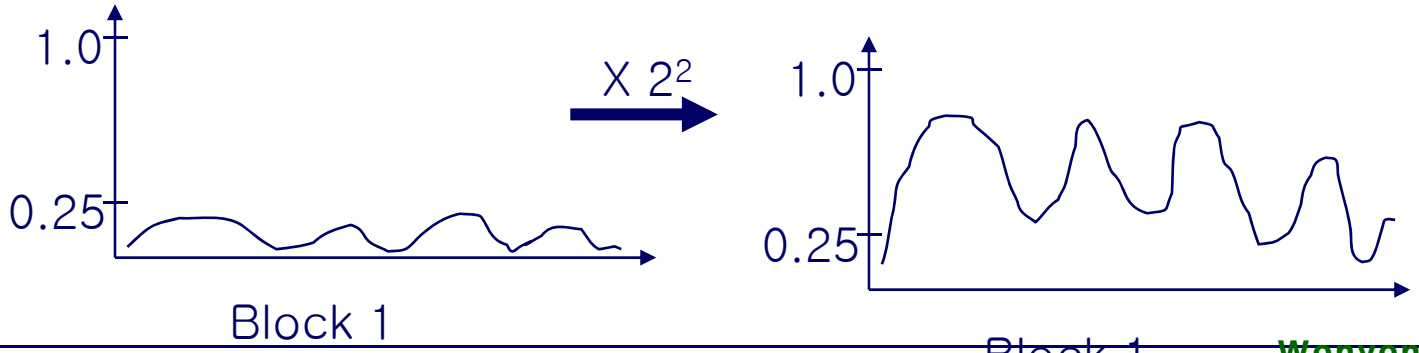
- **Lms(Xmem, Ymem, Acx, Acy)**
  - Performs a multiply-and-accumulate (MAC) operation and, in parallel, an addition with rounding (which would do the storage of updated coefficients)
  - Acy = Acy + (Xmem*Ymem),
    Acx = rnd (Acx + (Xmem << #16)

Instructions using CSSU

- **Dedicated to add/compare/select (ACS) operation in the Viterbi operation.**
- **CMPS B, *AR3**
  - If (B(31-16)>B(15-0)) then B(31-16)->*Ar3; TRN<<1; 0->TRN(0); 0->TC
  - ..

Block Floating-point Implementation

- Many signal processing algorithms are processed for a block of data
- Ex: A frame of 10 to 20ms is used for speech processing. 80~160samples per frame.
- Block floating-point arithmetic method normalizes this input data before processing, which reduces the effects of quantization noise when the signal level is low (block adaptive normalization).
- Can be understood as a block based AGC (automatic gain control).

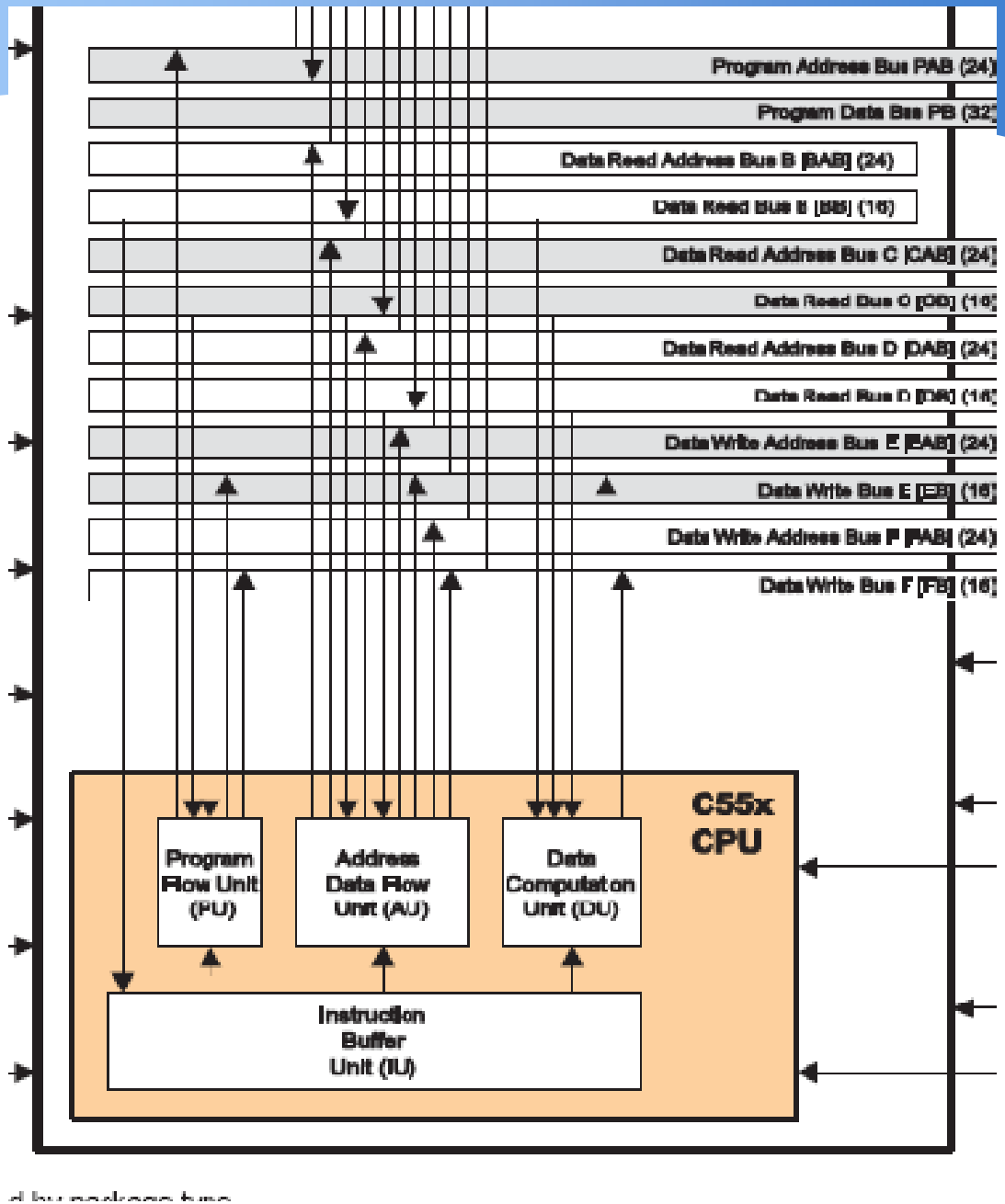


Block floating-point implementation procedure

- For a block of data, find out the maximum signal or its exponent (integer word-length)
- Determine the number of shifts for normalization (integer wordlength =0) from the integer wordlength of the maximum signal
- Normalize the signal by using barrel shifter, where the same number of shifts is applied for all the data in the block
- After processing, denormalize using the IWL previously determined.
- Supporting instruction: EXP src (A or B)
- number of leading bits of src -> T,
- When (src) = 0, T = 0
- Result: T becomes -8 to 31. (0 corresponds to normalized one, - corresponds to overflowed signal occupying the guard bit area)

4. C55x Architecture and Instructions

- **Architecture consists of**
  - Instruction buffer unit (IU)
    - Loads, parses, queues and decodes instructions
  - Program flow unit (PU)
    - Coordinates program actions among multiple parallel CPU functional units
    - Resource check, protected execution
  - Address data flow unit (AU)
    - Data address generation
  - Data computation unit (DU)
    - 40bit ALU, two MAC, a shifter, CSSU


Program Address Bus PAB (24)
Program Data Bus PB (32)
Data Read Address Bus B [BAB] (24)
Data Read Bus B [BB] (16)
Data Read Address Bus C [CAB] (24)
Data Read Bus C [CB] (16)
Data Read Address Bus D [DAB] (24)
Data Read Bus D [DB] (16)
Data Write Address Bus E [EAB] (24)
Data Write Bus E [EB] (16)
Data Write Address Bus F [FAB] (24)
Data Write Bus F [FB] (16)
C55x CPU
Program Flow Unit (PU)
Address Data Flow Unit (AU)
Data Computation Unit (DU)
Instruction Buffer Unit (IU)

Memory

- **Dual access RAM (DARAM) supporting two memory accesses per cycle**
- **Single access RAM (SARAM)**
- **ROM**
- **Configurable instruction cache**
- **EMIF (External Memory Interface)**
  - SRAM, SDRAM

Instruction buffer unit (IU)

- The CPU fetches 32bit packets from memory into the instruction buffer queue(IBQ)
- IBQ holds 64bytes of instructions to be decoded and provides 6 bytes (3 words) to the instruction decoder
- Speculative fetching of instructions while a condition is being tested for conditional goto, call, and return

Program flow unit (PU)

- Interpreting conditions for conditional instructions
- Determining branch (goto)instructions
- Initiating interrupt
- Managing single and block repeat operations
- Managing execution of parallel instructions

Address data flow unit (AU)

- Generating addresses for data read and writes
- Eight auxiliary registers and AGU's
- Circular, bit-reversed addressing..
- Also contains 16-bit ALU capable of performing arithmetical, logical, shift, and saturation operations.

Data computation unit (DU)

- **Two MAC units, a 40-bit ALU (can do dual 16-bit operations), a shifter.**
- **Four 40-bit accumulator**
  - source/destination for MAC and ALU
  - Optional 32/40 bit saturation.
- **Two transition registers (TRN0, TRN1) for Viterbi algorithm.**

Instruction pipeline (7 stages)

- Fetch stage – reads program
- Decode stage – decodes instructions and dispatches tasks to the other primary functional units
- Address stage – computes addresses for data accesses and branch addresses
- Access1 / Access2 stages – send data read addresses to memory
- Read stage – transfers operand data on B, C, D bus
- Execute stage – executes operation in the A, D unit and performs writes on the E and F bus

Tips for efficient memory allocation

- **Plan your SARAM vs DARAM data allocation**
- **For random access variables, use direct addressing and allocate them in the same 128-word page**
- **Reserve CPU resources for the exclusive use of interrupts**
  - Dedicated auxiliary registers are useful for servicing interrupts

Parallelism in C55x

- **Built-in parallelism (intra instruction parallelism)**
  - Perform two different operations using one instruction, separated by ::
  - EX: MPY … :: MPY, MAC
- **User-defined parallelism (inter instruction parallelism)**
  - Two instructions may be placed in parallel to have them both executed in a single cycle
  - Separated by “||”
- **Ex**
  - MPY *AR3+, *CDP+, AC0 :: MPY *AR4+, *CDP+, AC1 || RPT CSR

Restrictions in multiple instruction issues

- **Mainly resource conflicts**
  - Functional units, buses
- **Maximum instruction length**
  - The combined length of the instruction pair cannot exceed 6 bytes
- **Others**
  - Parallel enable bit is present for parallel execution

Parallelism tips

- **Place all load and store instructions in parallel.**
  - MOV *AR2, AC1 || MOV BRC0, *AR3
- **The A unit ALU can handle (un)saturated 16-bit processing in parallel with the D-unit (ALU, MAC, Shift)**
- **Accumulator shift, saturate, store operations can be placed in parallel with D-unit ALU or MAC**
- **Control operations (block repeat ..) can be placed in parallel with DSP operations.**
- **Instructions to be executed conditionally can be placed in parallel with the if instruction.**

Process for applying user-defined parallelism

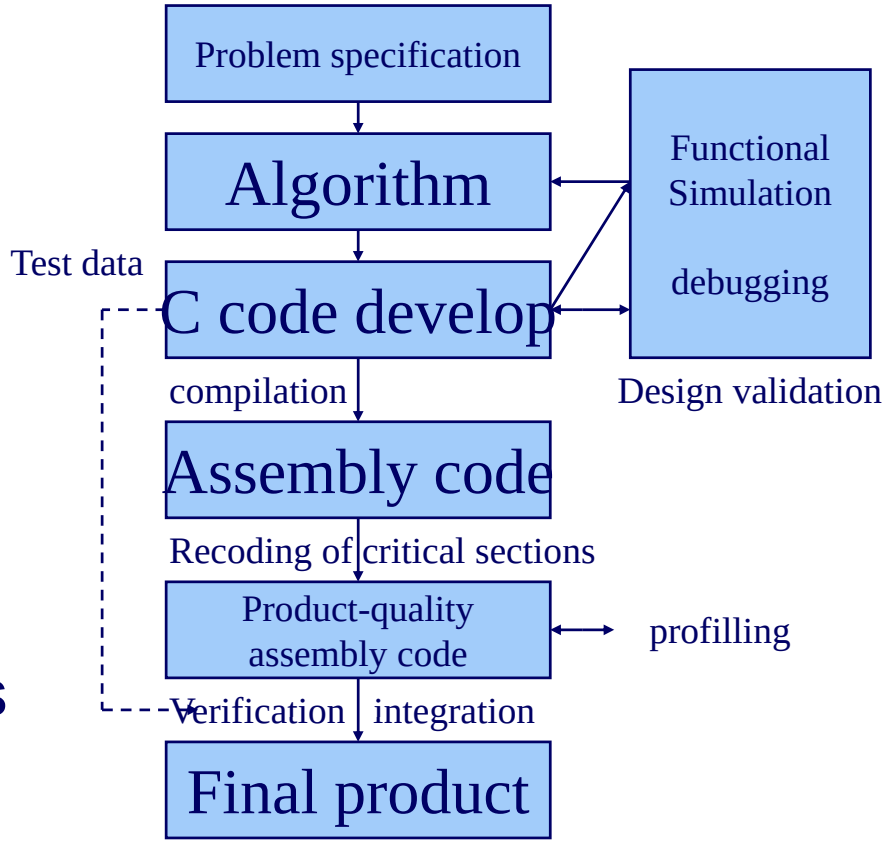
1. Write assembly code without parallel optimization, and test code for functional correctness (serial assembly code development)
2. Identify and place in parallel all potential parallel pairs
3. Assemble code and check assembler errors. If errors, goto step 2
4. Test code for functional correctness.

5. Compiler

- **Why compiler?**
  - Easy code development, good portability,..
- **Why are the compilers for programmable DSP's inefficient?**
  - Due to many parallel operations, distributed registers, different data-path structure (fixed-point DSP), application specific hardware
- **How is it worth trying?**
  - You can mix C and assembly codes.
  - 80%/20% rule. 20% of code (loops) takes 80% of execution time. In reality, it would be more!
    - So, 80% code use C code, 20% use optimized assembly language would be a compromise.
    - Many DSPLIB (very efficient) available for frequently used DSP kernels and algorithms (TI's very important asset)
    - Use intrinsics
  - Fast execution hardware speed, large memory
    - If the resources are not short, who cares for using inefficient programs (but fast delivery)?

Code development flow

- **C code development**
  - Functional verification
  - Profiling (CCS)
- **Assembly code development**
  - Loop or large execution time
  - Hardware dependent features



How to improve the C code in programmable DSP

- **Know the difference of data types and arithmetic operations in C and programmable DSP**
  - If you try saturated arithmetic, circular buffer in conventional C programming style, the results will be poor. (Portability is best.)
- **Try to utilize the special hw features, and see how the features are supported.**
  - Automatically supported (pointer increment..)
  - Needs to use macros or intrinsics
    - Saturated arithmetic, …
- **Try to tell as much as possible to the compiler using pragma (compiler directives) or memory models, the range of loop counts and so on.**
- **Needs to know how to integrate C and assembly codes**
  - Compile and assemble in different programs and link later
  - Use macro or inline assembly programs

Complier optimization

- **C compiler: parser, optimizer, code generator**
  - Assembler: generates a relocatable (COFF) object file
  - Linker: creates executable object file
- **The linker defines the memory map and allocate code and data into target memory (such as internal fast). You use linker to allocate global variables into the internal RAM and allocate executable code to external ROM. The compiler assumes nothing about the types of memory available, it just produces relocatable code.**

C compiler optimization options

- **Level 0:**
  - performs control flow graph simplification, eliminates unused code, expand inline function calls
- **Level 1:**
  - Performs local copy/constant propagation, removes unused assignments, eliminates local common expressions
- **Level 2:**
  - Performs loop optimization, loop unrolling, eliminates global common sub-expressions
- **Level 3:**
  - removes functions that are never called, performs file level optimization

Autoincrement and hardware repeat

- Auto increment addressing: for pointer expressions of the form *p++, the compiler uses efficient C55x autoincrement addressing modes. The loop optimization convert the array references to indirect references through autoincremented register variable pointers. Also hardware repeat is used.

```
int a[10], b[10]
void scale (int k)
{
   int i;
   for (i=0; i<10; ++i)
   a[i] = b[i]*k
}

MOV #9, BRC0
RPTBLOCAL L2-1
MPYM *AR2+, T0, AC0
MOV AC0, *AR3+
L2:
RET
```

Circular addressing in C

- **Original method is using the modulus operator (%) – but this is not efficiently supported yet, so need to use macros**
- **CIRC_UPDATE, CIRC_REF**
  - This macros utilize the hw circular addressing feature.

Data types

- **16bits – char, short, int**
- **32bits – long, float, double, long double**
- **40bits – long long**
- **Pointers – small memory mode 16bits**
  - Large memory mode 23bits
  - Function 24bits
- **C54x, C55x byte (minimum addressable size) is 16bits, sizeof(int)==1**
- **Saturated arithmetic: Conventional implementation in C is very inefficient**
  - Add, overflow check, (if overflow) max, min check, …
  - Use intrinsic

Saturated arithmetic(1)

- **Intrinsics are specified with a leading underscore, and are accessed by calling them as you do a function.**
- **The saturation is controlled by setting the saturation bit, ST1_SATD.**
- **int _sadd(int src1, int src2); -> ADD**
- **long _lsadd(long src1, long src2); -> ADD**

- **Int x1, x2, y;**
- **y = _sadd (x1, x2);**

- **_sadd:**
  - BSET ST3_SATA
  - ADD T1, T0
  - BCLR ST3_SATA
  - RETURN

- **ETSI FUNCTIONS**
  - #include <gsm.h>
  - Int sadd(int a, int b)
  - {
    - Return add(a, b)
    }

Rounded arithmetic

- Long _round(long src) -> ROUND
- returns the value of src rounded by adding 2**15 using unsaturating arithmetic and clearing the lower 16bits.

Other intrinsics

- long _lsadd(long src1, long src2); adds two 32-bit integers, producing a saturated 32bit result (SATD bit set)
- int _smpy(int src1, int src2); multiplies src1 and src2, and shifts the result left by 1, produces a saturated 16-bit result (SATD, FRCT bit set)
- Long _lsmpy(int src1, int src2); … produces a saturated 32bit result
- Long _smac(long src, int op1, int op2); multiplies op1 and op2, shifts the result left by 1, and adds it to src. Produces a saturated 32-bit result.

Efficient loop code

- **Encourage the use of the HW repeat (local repeat (only forward control flow insdie), blockrepeat, ..)**
  - Avoid function calls within the loop body
  - the loop count should be int, not long (16bit HW)
- **Keep loops small, this will result in local repeat code (less power consumption)**
- **Use MUST_ITERATE pragma**
  - No check whether the loop count is 0.

Multiplication and MAC in C54/55

- For single repeat, use local variables for the summation. Global variable needs an intervening storage. This prevents from a single repeat.
- Returning Q15 result for Multiply-accumulate

```
--;
long sum=0;
for (i=0; i<=n-1;i++)
      sum += (long) x[i] * y[i];
return (int) ((sum>>15) & 0x0000FFFFL);
```

Generating dual MAC operations for C55

- **The code must have two consequative MAC instructions and the two operations must not write their results to the same variable or location.**

```
long s1, s2
…
s1 = s1 + (*a++ * *c);
s2 = s2 + (*b++ *c++);
```

- **The memories are in the on-chip. Int onchip b[10]; …**

FIR filter

- **Single MAC based FIR**

```
short i, j; long y0;
for (j=0; j<m; j++){
y0 = 0;
  for (i=0; i<n; i++)
    y0 += (long) x[i+j]*h[i];
  y[j] = (short) (y0 >> 16);
 }
```

- **Dual MAC FIR – after unroll-and-jam transformation**

```
for (j=0; j<m; j+=2){
y0 = 0; y1=0;
  for (i=0; i<n; i++) {
    y0 += (long) x[i+j]*h[i];
    y1 += (long) x[i+j+1]*h[i];}
  y[j] = (short) (y0 >> 16);
  y[j+1] = (short) (y1 >> 16);
 }
```

Memory models

- Small memory model: compiler uses 16-bit data pointers. The upper 7bit of XARn is determined by the .bss page (reserves .
- Large memory model: data pointers are 23bits and occupy two words when stored in the memory

Summary of C/C++ code optimization techniques

Optimization tech.	Potential gain	Easy of implem.	Opportunities	Issues
Efficient loop code	high	Easy	Many	
Use MAC efficiently	High	Moderate	Many	
Intrinsics	High	Moderate	Many	Reduced portability
Avoid modulus in circular	Moderate	Easy	Some	
Use long for 16bit	Low	Moderate	Few	Two 16bit words using 32bit bus
Efficient control code	Low	Easy	Few	

6. Conclusion

- C54x/C55x contains application specific functional units and instructions.
- C54x/C55x has three to five data buses to increase the throughput
- C55x supports multiple execution of instructions (a kind of VLIW).
- C54x/C55x considers the easy of programming and compiler support in mind by eliminating the pipelining register in the MAC units, and increase the number of accumulators to 4 (C55)
- The use of C compiler is encouraged, and recent versions utilize the hardware architecture fairly well. The difference of data types and arithmetic operations (such as 40bit addition, 16*16bit -> 40bit, saturated arithmetic) would be something that needs careful consideration.