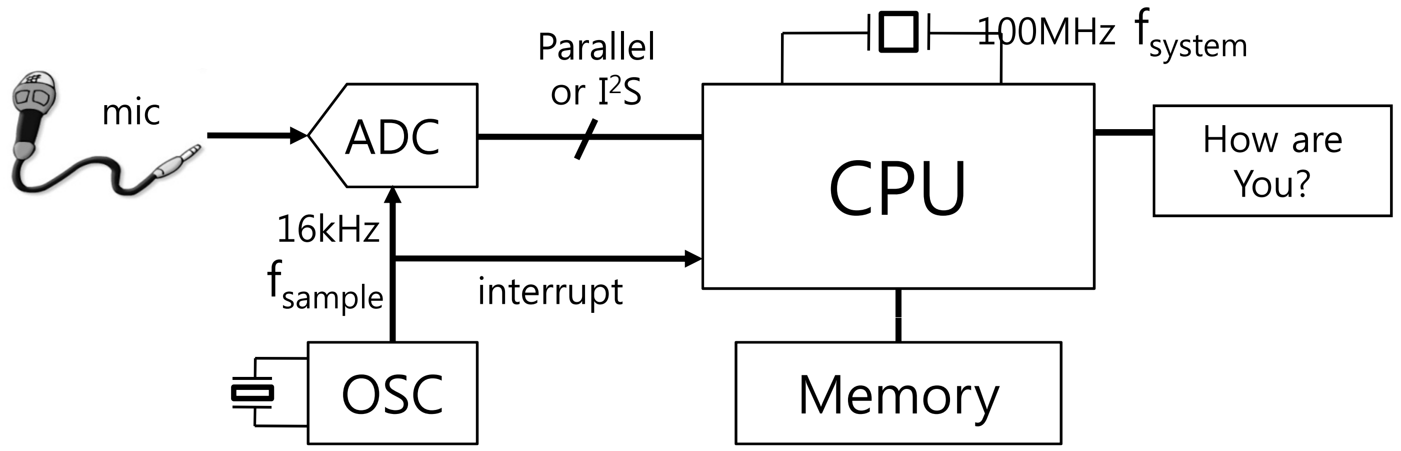
Embedded Real-time Signal Processing System Design Issues

Wonyong Sung

Contents

1. Introduction

- Hardware: CPU + memory + ADC/DAC
- Sampling clock(f_{sample}): clock for ADC/DAC
- System clock(f_{system}): that for CPU
- $f_{system} >> f_{sample}$ for SW based implementation
  - $f_{system:}$ 100MHz ~ 1GHz
  - $f_{sample:}$ 8KHz for telephony, 48KHz for MP3, 10MHz for video



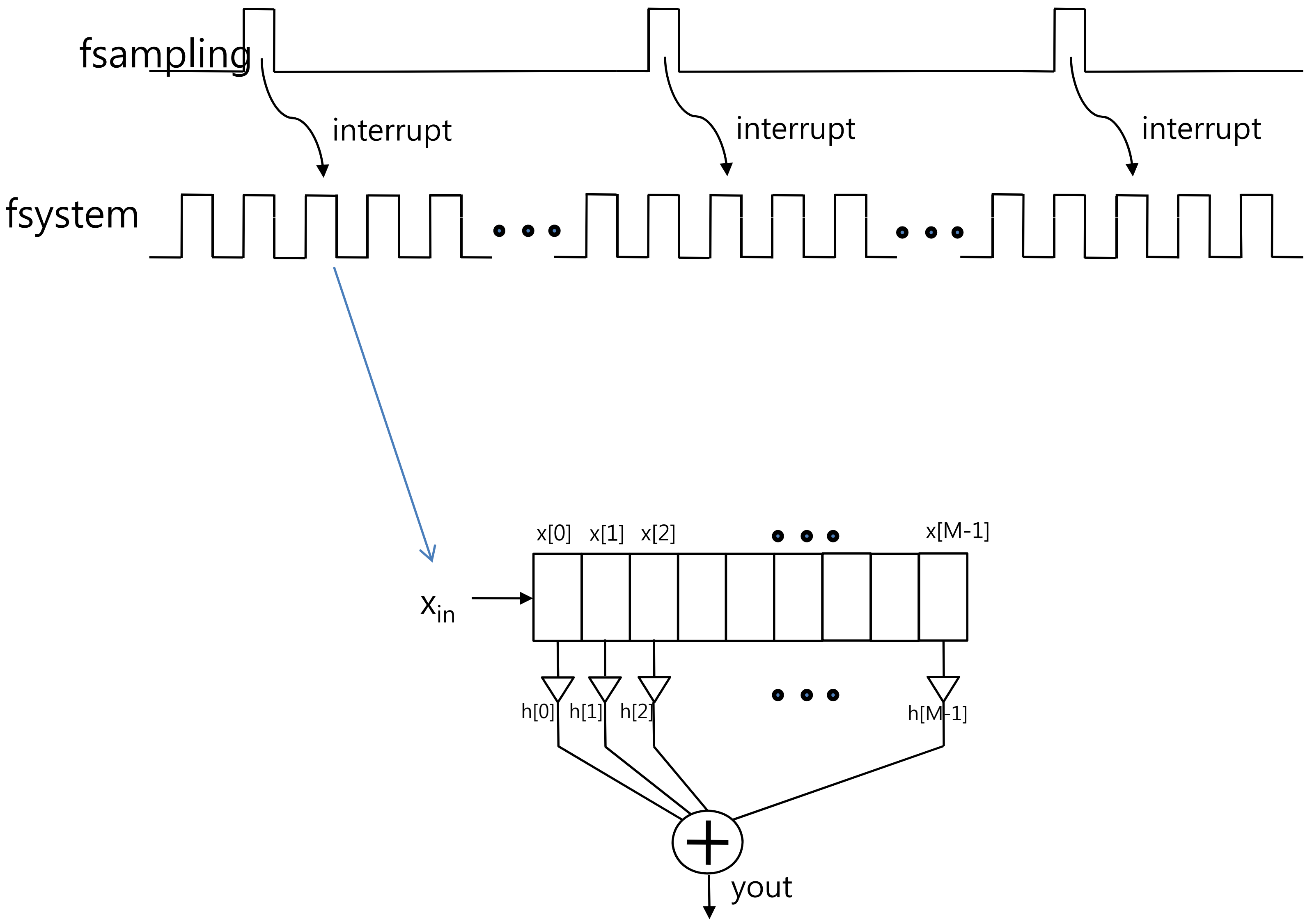
Speech recognition system

Simple FIR filtering flow

- Consider FIR (8tap) filtering
- For each interrupt
  - Get new input, xin, from the ADC buffer
  - Data move in the tap (new input arrived, so old one should move to the next tap)

    ```
    for (i=7; i>0; i++) x[i] = x[i-1];
    x[0] = xin;
    ```
  - Compute the output

    ```
    for (i=0; i<8; i++) yout += h[i]*x[i];
    ```
  - Put the new output, yout, to the DAC buffer


fsampling
interrupt
interrupt
interrupt
fsystem
x[0]
x[1]
x[2]
x[M-1]
x_{in}
h[0]
h[1]
h[2]
h[M-1]
yout

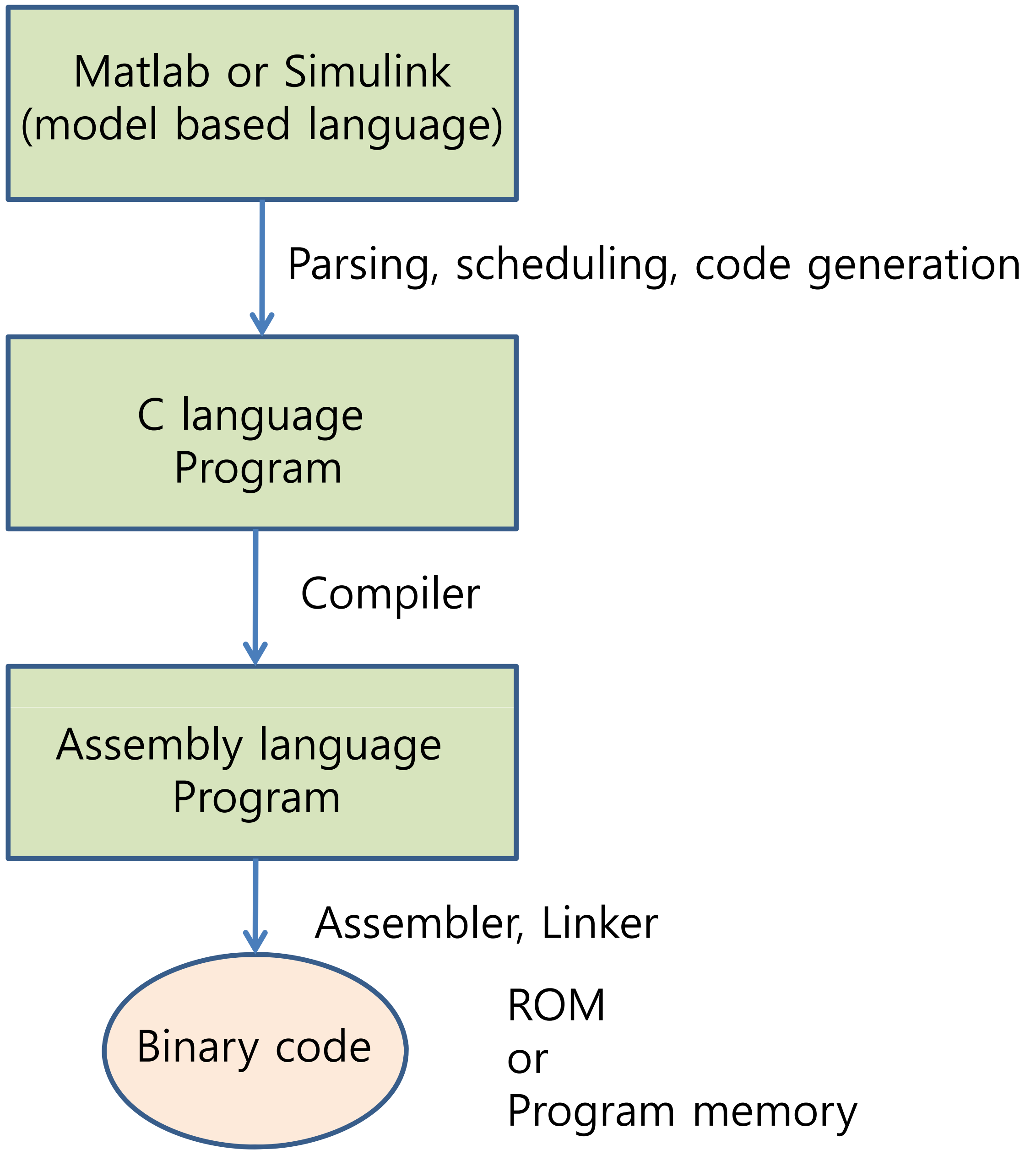
f_{sample} and f_{system}

- Sampling clock (for ADC) is the master of timing (not the CPU) in real-time systems.
- System clock (for CPU) is needed just for sequential operation of CPU. In most case, you can increase the fsystem without changing the system functionality (this increases idle time).
- fsystem/fclock means the number of CPU clock cycles assigned to each signal sample.

2. DSP Code Development

- By assembly programming
  - The code is quite efficient (use less cycles).
  - It would take time and hard to debug.
  - Not portable (different CPU, different programming)
- By C/C++ programming
  - The code can be fairly good.
  - C/C++ programming is easier than assembly programming, but still difficult to maintain and debug.
  - The program is portable (independent of CPU).
- By Matlab or Simulink
  - The code is very short (easy to develop and maintain) and portable. You don’t consider any HW.
  - Still very inefficient (good for simulation purpose)

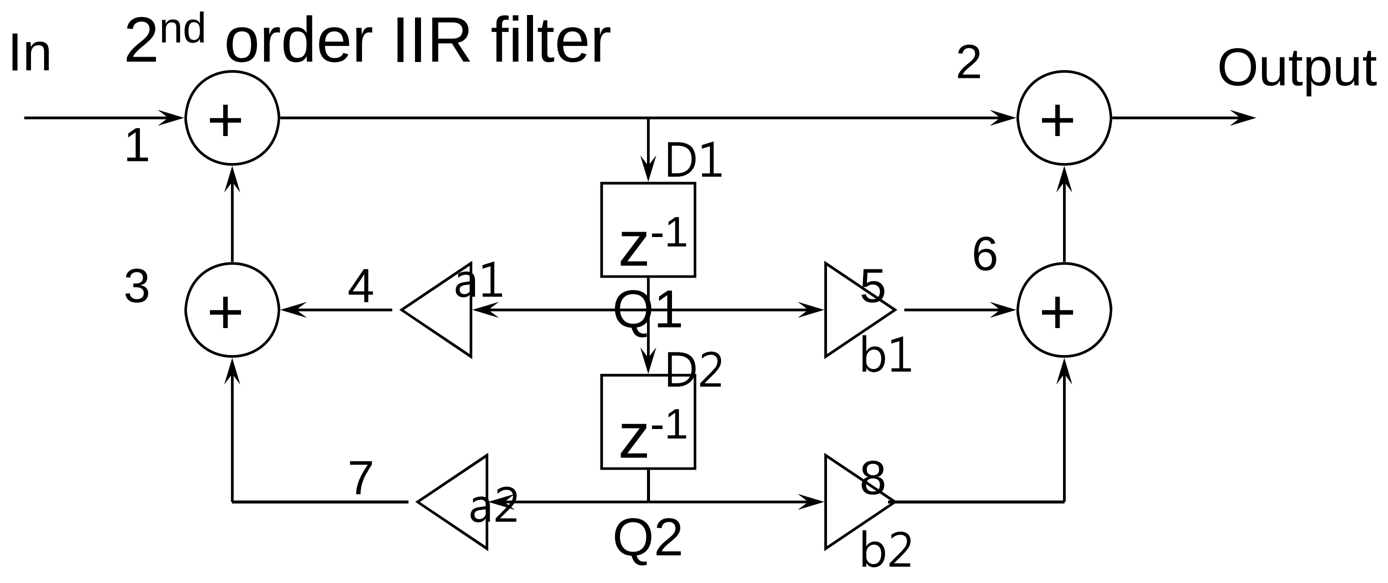
```
b=[h0, h1, ….]
y = filter( b, 1, x );
```



From SFG to a program

- SFG (Signal Flow Graph): shows the sequence and operations.
- SFG conducts the same computation with the period of sampling clock (single rate system).
  - The input of the computation: input sample, output of z-1.
  - The output of the computation: output sample, input to z-1
  - At the end of computation: update z-1 (input data is stored)
  - The sequence of computation is given by the directed graph in the SFG.

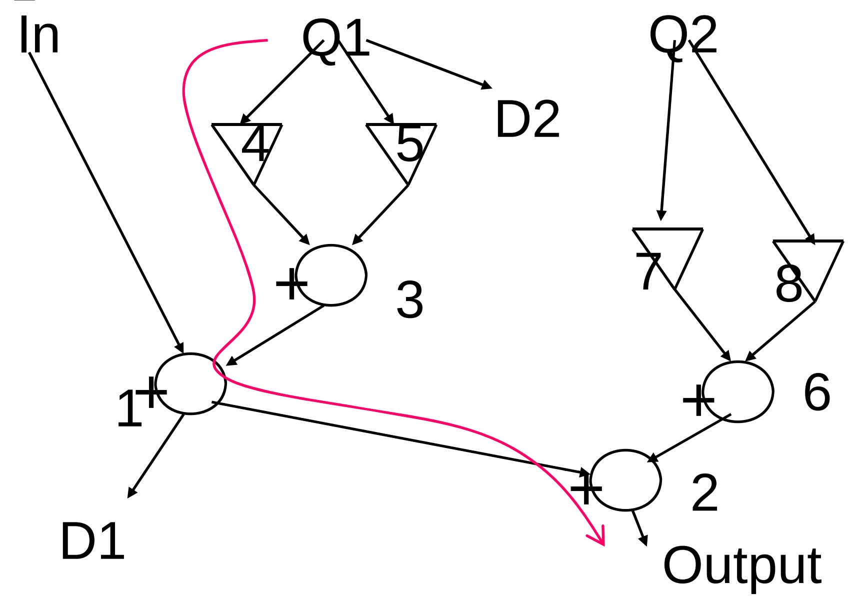
2nd order IIR filter



For each sampling period

f(In, Q1, Q2) -> Output, D1, D2

```
For (every new input sample) {
d1 = q1*a1 +q2*a2 + in;
output = q1*b1+q2*b2+d1;
d2 = q1;
/* now update the registers */
q2 = d2;
q1 = d1;
};
```



Adaptive lattice filter

3. Speeding-up

- Naïve implementation
  - ARM9 board with a codec (you can set the codec frequency in SW).
  - ARM compiler
  - Download the program into DRAM or flash memory

ARM9 CPU

- RISC architecture (pipelined instruction execution) with separate instruction and data cache
  - Only integer multipliers (32*8)
- Many SoC devices from Samsung, Telechips, …
  - In these days, for a real product implementation, choosing an adequate SoC that contains the I/O functions that you need is very important.

Good implementation

- You operate the system with a minimum CPU clock frequency.
  - It takes less power when lowering the frequency.
- Measure the number of cycles for FIR filter implementation (assume 1,000 taps, operating at 8KHz) and guess the system clock frequency needed.
- Think how you can improve the speed (or lower the needed CPU clock frequency).

Floating-point to integer conversion

- ARM9 or many programmable DSPs do not equip floating-point arithmetic units
  (do not need to care when using PC or GPU)
- 1 floating-point add or mult takes about 100 cycles when emulated using integer instructions
  - 2000 floating mult, add x 100 instr. x 8KHz
    - 1,600MHz minimum (too high!)
- float -> int, or short (16bit)
- What would be the problem?
  - Overflow, scaling, quantization error
- Now, 2,000 mult, add x 8KHz = 16 MHz, but too optimistic since we ignored data move, branch control overhead.

```
*output = 0;
for (j = 0; j < NTAPS; j++)
        *output += (*data--) * (*coef++);
```

(a) C code for FIR filtering

```
        st      %g0,[%o1]       ; *output = 0
        mov     0,%o7           ; j=0; initialize loop index
LM00001:
        ld      [%o3],%g2       ; load data
        add     %o7,1,%o7       ; j++; loop index++
        ld      [%o4],%i0       ; load coef
        cmp     %o7,255         ; j > 255? check if loop ends
        ld      [%o1],%g3       ; load *output
        smul    %g2,%i0,%g2     ; mul
        add     %g3,%g2,%g3     ; acc
        st      %g3,[%o1]       ; store *output
        add     %o4,4,%o4       ; coef++
        add     %o3,-4,%o3      ; data--
        ble     LM00001         ; branch
```

(b) Compiled code (assembly) for Sparc CPU

How many instructions for each tap? 11.
How many multiplications among 11 instructions: just 1
How many load, store instructions? 4

```
|L1.28|
        RSB     r1,r4,#0x64
        LDR     r0,[r6,r4,LSL #2]
        LDR     r1,[r7,r1,LSL #2]
        MLA     r5,r0,r1,r5
        ADR     r0,|L1.80|
        MOV     r1,r5
        STR     r5,[r8,r4,LSL #2]
        ADD     r4,r4,#1
        CMP     r4,#0x64
        BLT     |L1.28|
        ADD     sp,sp,#0x4b0
        POP     {r4-r8,pc}
        ENDP
```

CPU clock freq estimation with overhead instructions

- The former slide shows that there are 9 overhead instructions (for data move, branch) besides one mult, one add operations.
- So, the realistic estimation: 16MHz x 11/2 = 88MHz (it is still too optimistic). Why, it assumes the CPI of 1.

CPI issues

- CPI: cycle per instruction
  - Ideal pipelined RISC CPU: CPI = 1
  - Memory latency affects the CPI very much
  - When many cache misses, CPI goes very high.
  - RISC CPU with real cache: CPI = 1.5 ~ 10
  - VLIW CPU: CPI < 1.0
- ARM9 CPU equips cache memory, which speeds up in a probabilistic way
  - It is difficult to guarantee speed (of course you can do safe real-time implementation by allowing a large idle time).
  - Programmable DSPs usually do not have cache memory for this reason. But, it is costly to have a large fast memory.
- Assume CPI of 2 for this example:
  - Needed frequency: 2x88MHz = 176MHz CPU clock freq needed.

Algorithm or program optimization

- Algorithm opt: use symmetric filter structure to reduce the number of multiplications.
- You may use a recursive (IIR) filter because it usually needs a much lower order.

- Program optimization: use loop unrolling to remove the overhead of conditional branches.
  - This increases the code size.
- We can expect to lower the instruction cycles by a half: 88MHz CPU clock.

Use a parallel processor (SIMD, VLIW, ..)

- We can reduce the CPI below 1 (assume 0.2)
- The needed clock frequency becomes 8.8 MHz.
- Limitations of VLIW/SIMD approaches
  - Limited parallelism of DSP algorithm
    - Due to dependency problem
  - Difficult to increase the number of issues
    - Due to increasing complexity of interconnection
    - Due to increased number of ports for memory, registers

Use a better ISA (Instruction Set Architecture)

- Programmable DSP's support instructions that can do FIR filtering very efficiently.
  - 1 instruction/ tap
  - 1000 cycles x 8KHz + some overhead ~= 10MHz CPU clock frequency

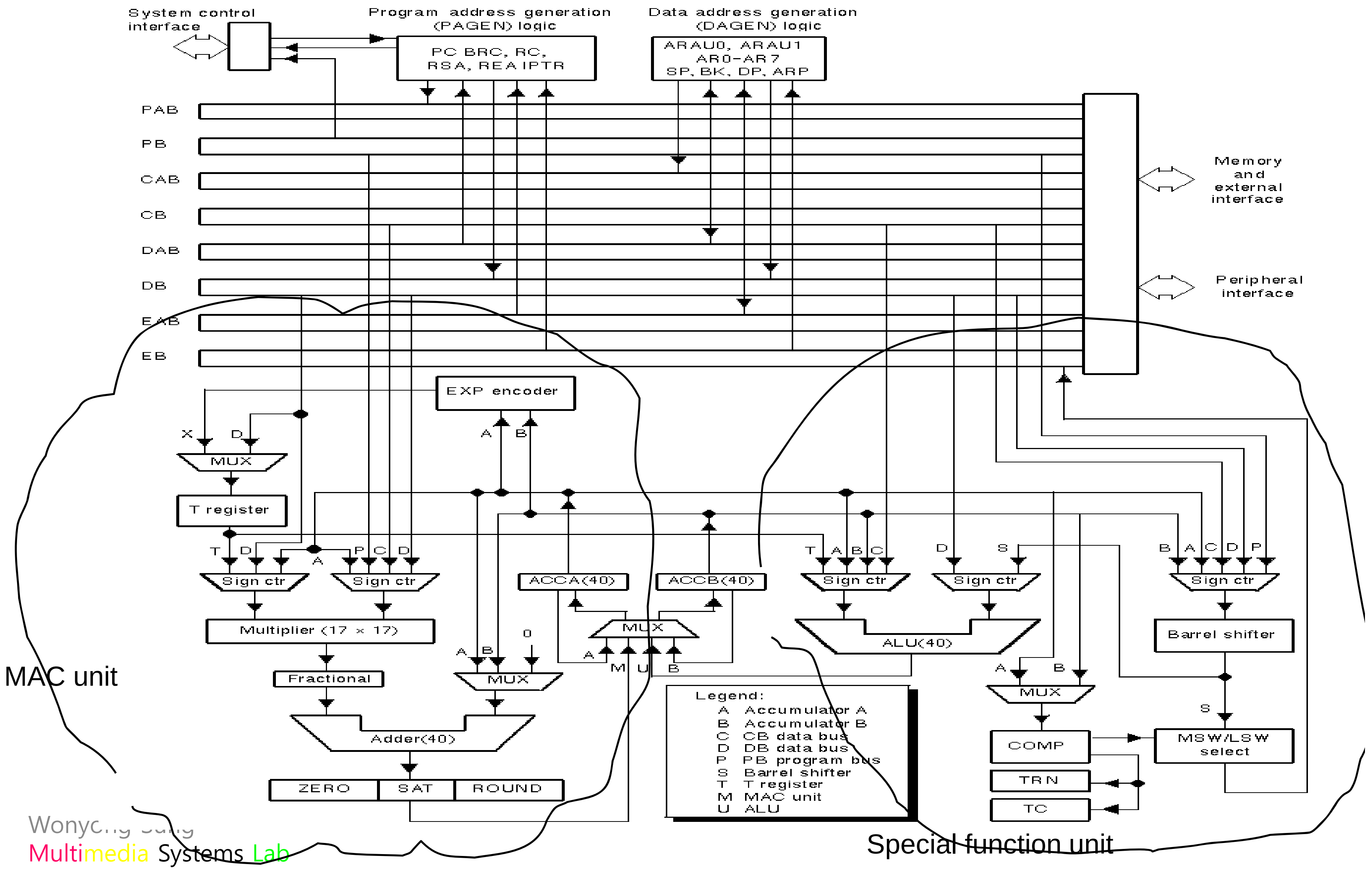
- Ex1: C2x, C5x FIR filter
```
zpr; zero product register
rpt  #FILT_ORDER-1
macd  #coef, *- ;ar3=&z[FILT_ORDER]
apac ; Add the P register to ACC
```

*To conduct macd instruction, we need a hardware that is much more complex than a RISC CPU (it should do concurrently multiplication, accumulation, coefficient read, data read, coefficient address update, data address update. Branch control is done is hardware.)

- Limitation: algorithm dependent speed-up

TMS320C54x Internal Block Diagram

TI Document



Difference between DSP and VLIW

- Special instructions in DSP are applicable to only some small kernels (filtering, FFT, Viterbi decoding). It may not be effective for other kernels.
- VLIW speed up is more universally applicable.
- VLIW or other parallel processors provide a better compiler and development tools.

4. Design Experience

- SpeakingPartner
  - A handheld English learning device for kids
  - Animation, MP3 play, speech recognition, voice recording (multi-tasking)

Programmable DSP based design

- DSP processor is efficient in conducting MP3, speech recognition, and so on. (x5 efficient).
- Problems:
  - It's memory space is too small (1Mbytes) for all the codes. We needed DRAM but TI DSP chip does not have DRAM controller. We had to develop FPGA based DRAM controller (expensive and consumes power)
  - It's C compiler is not good.
  - TI DSP has no LCD interface.

ARM7 SoC based design

- Samsung developed an ARM7 CPU based SoC for handheld PDA market.
  - It has DRAM and LCD interfaces.
  - The code development environment is very good.
  - ARM7 CPU takes many cycles for DSP processing.
- We adopt ARM7 SoC and try to develop efficient DSP codes on ARM CPU.
- System development was successful, but the power consumption was something large (due to small cache). (8 hrs with a battery. This needs rechargeable batteries.)

If SpeakingPartner is redesigned,

- ARM9 or ARM11 based SoC
  - Support higher clock frequency
  - Large cache (good for low-power)
  - Good compiler and debugging tools
  - Many suppliers
- TI OMAP based
  - ARM9 + TI DSP
    - ARM9 for animation, DSP for speech processing<- very efficient!
    - Use two different design environment
    - Only from TI