REAL TIME
SIGNAL PROCESSING
with the
DSP563xx Signal Processor

Jean-Marie ORY

Université Henri Poincaré, Nancy, France

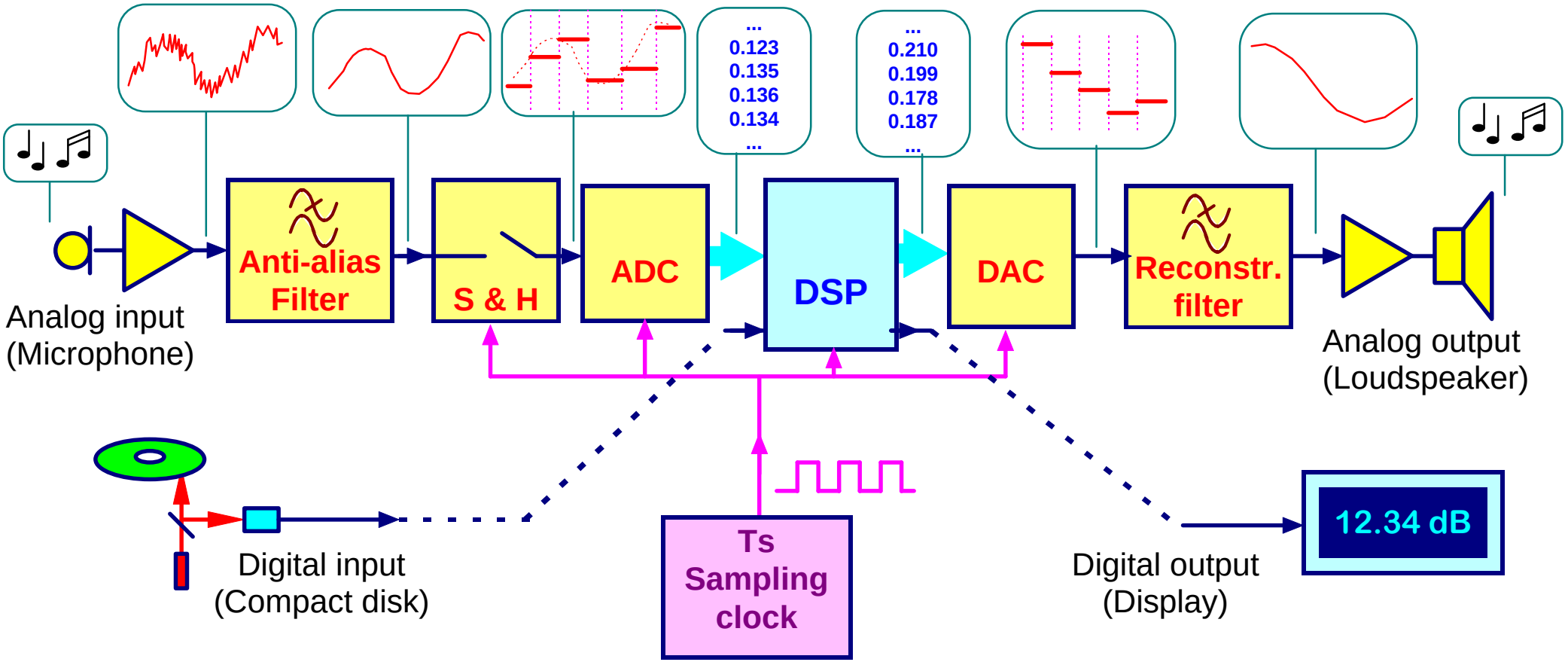
jean-marie.ory@esstin.uhp-nancy.fr

Sept. 2004

TABLE OF CONTENTS

1. DIGITAL SIGNAL PROCESSOR FEATURES

1.1 A typical DSP application (real time audio processing)



1.2 Digital Signal Processor (DSP) features

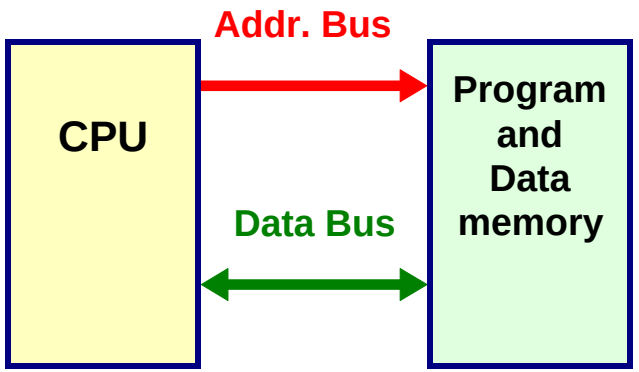
- Applications of DSPs: audio, video, telecommunications
- Machines designed for *real time* processing of fast data flows.
- Repetitive math intensive algorithms, most are sum of products (Convolution, Fourier, Matrix ...)
- Signals are sampled: processing is synchronous to sampling rate
- Execution must be deterministic time

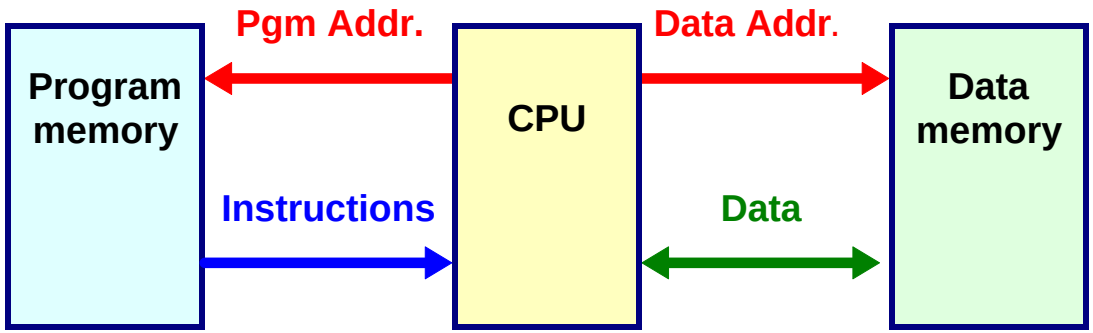
Example:
Processing Audio stereo at 48Ks/sec ➔ whole algorithm less than 20.8 µs

Goal: **process data as fast as possible**

1.3 Architecture

1.3.1 Harward vs. Von Neuman machine

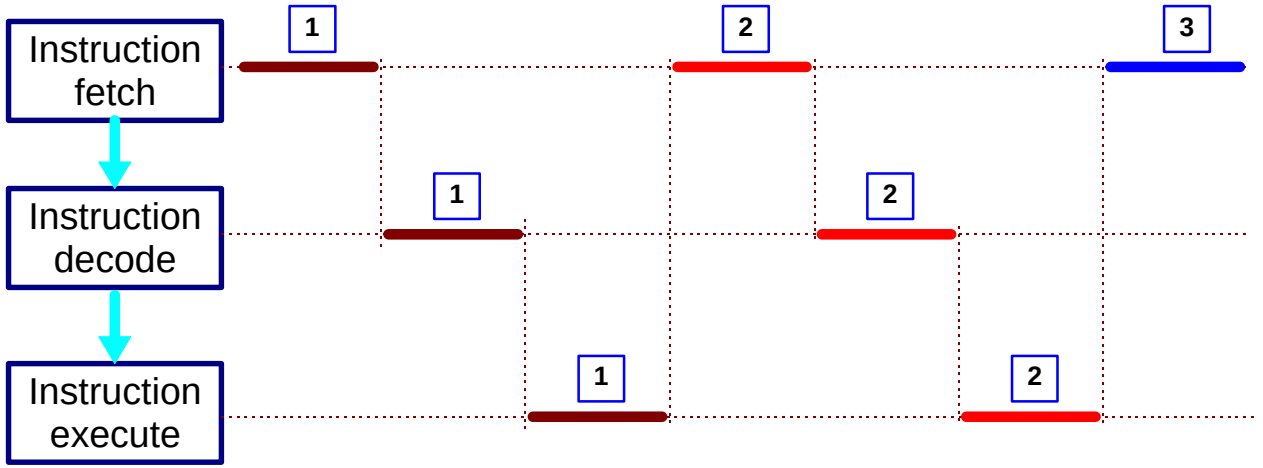




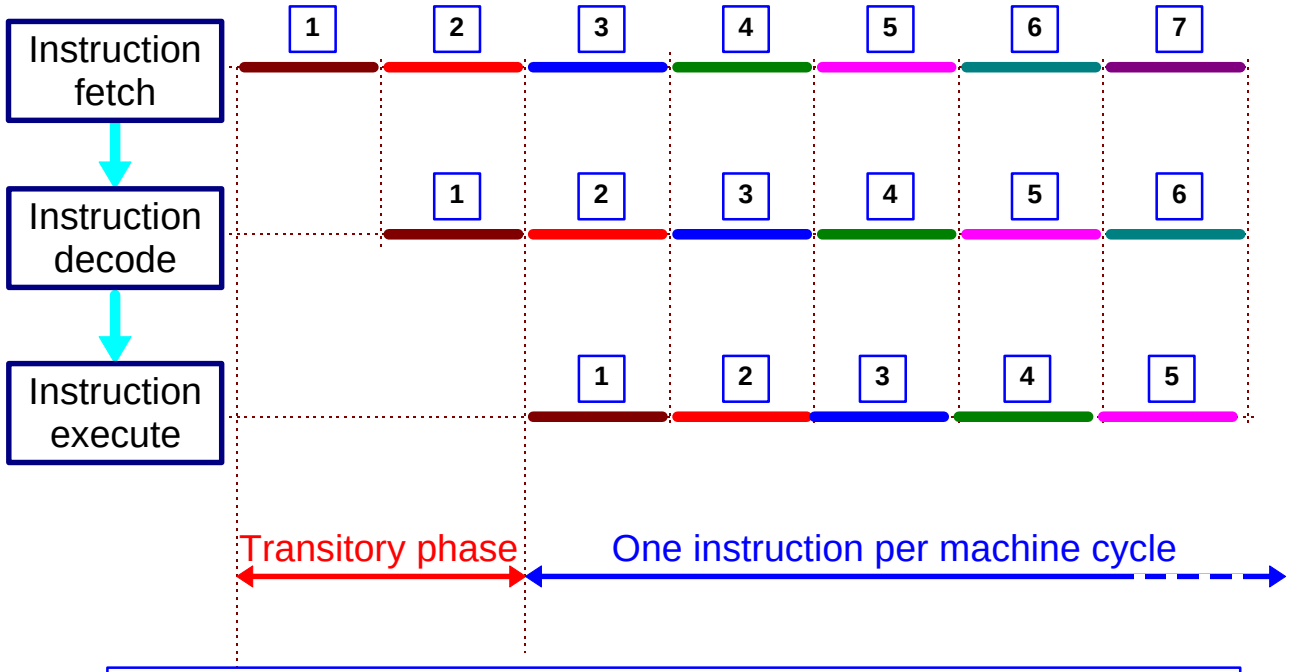
Von Neuman processors:
a) Fetch instruction
b) Fetch data operand(s)
➔ Data bus is the bottle-neck

Harward processors:
Fetch instr and data at once

1.3.2 Pipelined stages :



No pipeline:
Each instruction needs 3 machine cycles minimum



Pipelined processor:
After transitory phase, each instruction needs only 1 machine cycle

1.3.3 Parallel processing

Hardware
Several concurrent units:
CPU core, Address generators, DMA channels, smart I/O's

Software
Wide instruction words:

Examples:
DSP56000 24 bit arithmetic instruction format

WLIW Very Large Instruction Words (Texas Instruments C6xxx) → 1600 MIPS @ 200MHz

SIMD Single Instruction Multiple Data (Analog Devices Sharck)

- Specialized DSP features

Fixed point or floating point arithmetics.
Saturation limiting arithmetics
Fast multiply and Multiply-Accumulate (MAC) instructions
Barrel shifter
Hardware DO loops
Hardware stack
Modulo m addressing mode
Reversed binary addressing mode
Conditional execution

Most popular Digital Signal Processors (DSP) -- year 2000

Manufacturer	Family	Arith.	bits data	bits instr.	MIPS	Caractéristiques
Texas Instruments	TMS320C1x	fix	16	16	8,8	TMS320 -10 = first true DSP on market
	TMS320C2x	fix	16	16	12,5	
	TMS320C2xx	fix	16	16	40	Low cost
	TMS320C3x	float	32	32	30	
	TMS320C4x	float	32	32	30	Multi-processor design
	TMS320C5x	fix	16	16	50	Evolution of the C2x
	TMS320C54	fix	16	16	100	Specialized instructions
	TMS320C62	fix	32	256	1600	VLIW architecture
	TMS320C67	float	32	256	1336	VLIW architecture
	TMS320C8x	fix / float	8/16	32/64	300	2-4 DSP + RISC (video applications)
Motorola	DSP560xx	fix	24	24	47,5	56 bit accumulators audio, multimedia
	DSP563xx	fix	24	24	100	" "
	DSP566xx	fix	16	24	80	
	DSP568xx	fix	16	16	40	Programmable as a microcomputer
Analog Devices	ADSP-21xx	fix	16	24	50	
	ADSP-2106xx	float	32	48	40	Multi-processor design
Lucent	DSP16xx	fix	16	16	120	
	DSP16xxx	fix	16/32	16/32	100	2 multipliers
Hitachi	SH-DSP	fix	16/32	16/32	60	Hybrid DSP + Microcomputer
SGS Thomson	D950	fix	16	16	40	Processor core
DSP Group	Pine	fix	16	16	40	Processor core
	Oak	fix	16	16	40	Processor core

1.4 QUIZ 1

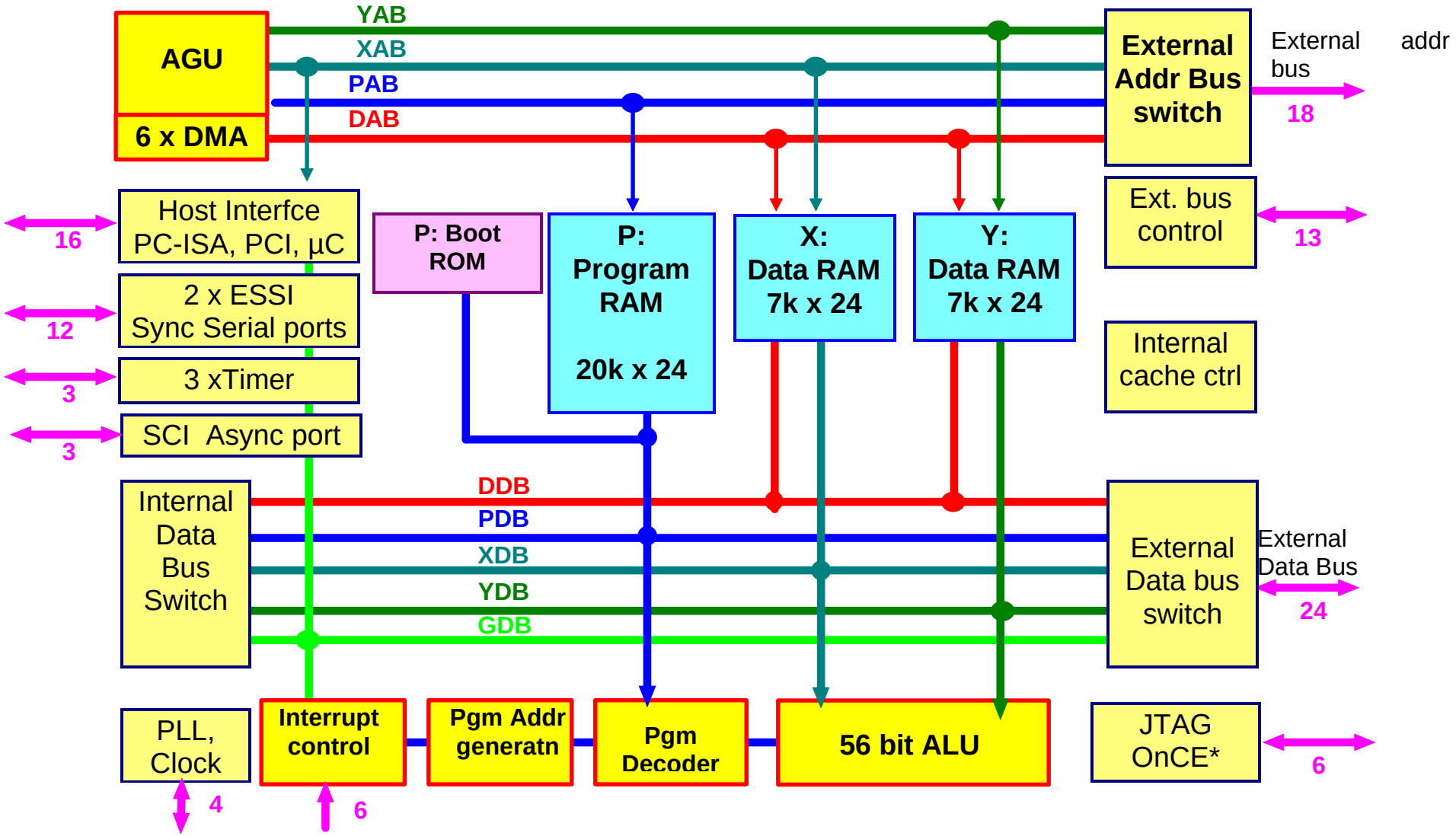
Digital Signal Processor features

1. What is most difficult when processing signals in real time ?
2. Give the names of most known DSP manufacturers
3. Which technologies are used in DSPs to achieve high processing speeds ?
4. What are the differences between fixed point and floating point DSPs ?
5. What is the difference between overflow and saturation ?
6. What is the most typical DSP instruction ?

---

2. THE MOTOROLA DSP563XX FAMILY

2.1 DSP56309 Internal architecture



2.2 DSP56300 family programming model

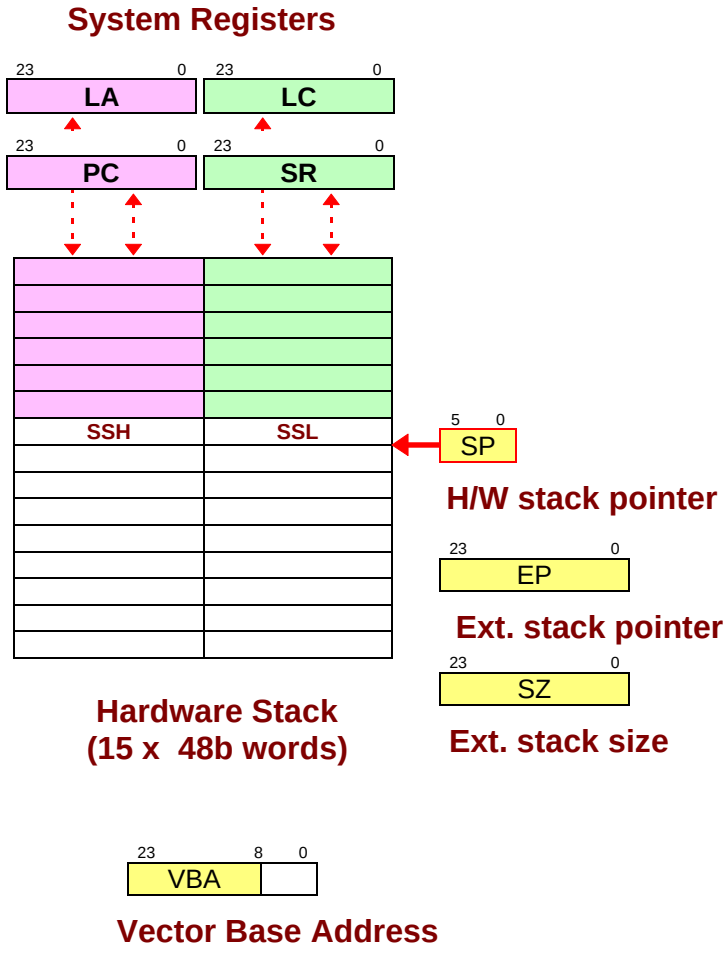
Accumulators

General purpose registers

Address Registers

23 0	23 0	23 0
R0	N0	M0
R1	N1	M1
R2	N2	M2
R3	N3	M3
R4	N4	M4
R5	N5	M5
R6	N6	M6
R7	N7	M7
Pointers	**Offsets**	**Modificators**

DSP56300 family programming model (continued)



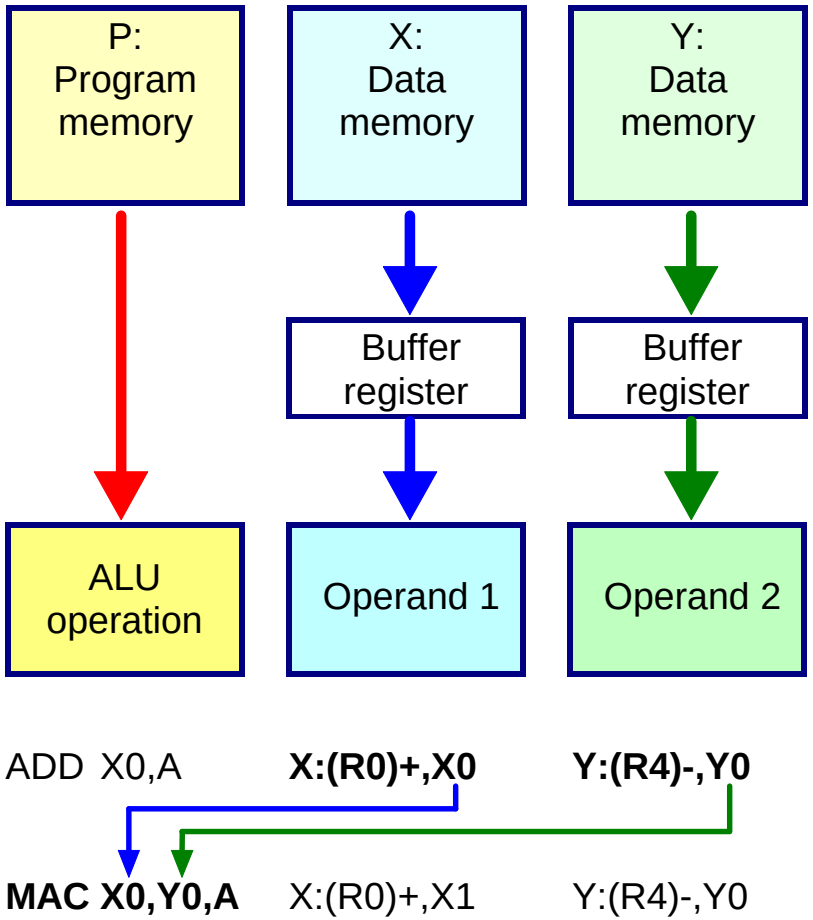
23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMR								MR								CCR							
CP		RM	SM	CE	-	SA	FV	LF	DM	SC	-	SCM		IM		S	L	E	U	N	Z	V	C

Status Register SR

EMR Extended Mode Register	MR Mode Register	CCR Condition Code Register
CP: Core priority vs DMA 00 lowest .... 11: highest	**LF**: DO Loop Flag	**S**: FFT scaling required (sticky bit)
	DM: Double precision Multiply mode (for 56002 compatibility)	**L** : Limit = overflow or limiting occured (sticky bit : cleared by user write or hw rst)
RM: Rounding Mode (0: Convgt 1: 2's Cmpt)	**SC** : 16 bit compatibility mode	**E** : Extension in use b55..b47 not identic.
SM: Saturation Mode (0: 56b 1: 48b)	**SCM** = Scaling mode: 00 wr (Acc) 01 wr (Acc/2) 10 wr (Acc*2)	**U** : Unnormalized 2 MSB's identical
CE: Cache Enable		**N** : Negative b55 = 1
SA: 16 bit Arithmetic mode	IM = Interrupt mask level: 00 IPL0,1,2,3 01 IPL1,2,3 10 IPL 2,3 11 IPL3	**Z** : Zero (b55 .. b0) = 0
FV: "DO-Forever" flag		**V** : oVerflow b55 <> sgn (result)
		C : Carry (Add / Sub / Shift / Rot) > 56b

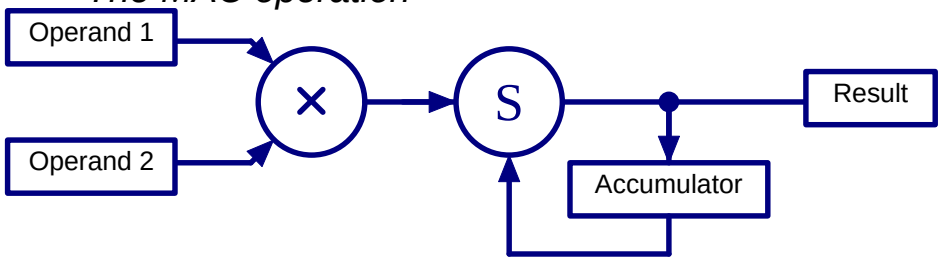
2.3 Utility of the dual data memory and buffer registers

DSP56xxx dual data fields allow repeated single cycle MAC instructions



➔ One MAC instruction per machine cycle

The MAC operation



2.4 DSP56300 Instruction set

2.4.1 **Arithmetic instructions**

They apply to a full 56 bit accumulator A or B.

ABS A	Absolute value A = |A|
ADC X,A	Add long w. carry A=A+X+C (48b)
ADD X0,A	Add A= A+X0 (24b)
ADDL X0,A	Shift left and add A=2A+X0
ADDR X0,A	Shift right and add A=A/2 + X0
ASL A	Arithmetic shift left A=2A
ASL #5,A,B	ASL multi-bit B=2^5A
ASR A	Arithmetic shift right A=A/2
ASR #5,A,B	ASR multi-bit B=2^{-5}A
CLR A	Clear accumulator A=0
CMP X0,A	Compare (A-X0) ?
CMPM X0,A	Compare magnitude (|A|-|x0|) ?
CMPU X0,A	Compare unsigned (A-X0) ?
DEC A	Decrement accumulator A= A-1
DIV X0,A	Divide iteration (one bit at a time)
DMAC X0,Y0,A	Double mac A=A*2^{-24} + x0*y0
INC A	Increment accu A=A+1

MAC X0,Y0,A	Multiply-accum. A=A+x0.y0
MACR X0,Y0,A	Multiply accumulate and round
MAX A,B	if A > B then B = A
MAXM A,B	if |A| > |B| then B=A
MPY X0,Y0,A	Multiply A=X0.Y0
MPYR X0,Y0,A	Multiply and round
NEG A	Negate accumulator A= -A
NORM R0,A	Normalisation
NORMF B1,A	Fast normalisation
RND A	Round accumulator
SBC X,A	Substract long w. carry A=A-X-C
SUB X0,A	Substract from accum. A=A-X0
SUBL X0,A	Shift left and substract A=2A-X0
SUBR X0,A	Shift right and substract A=A/2-X0
Tcc X0,A R0,R1	Transfer on condition <**cc**> true
TST A	Test accumulator, update CCR

2.4.2 **Logical Instructions** : Apply on 24 bit A1 (B1) partial accu, or on a single bit.

AND X0,A	Logical AND A1 := A1 & X0
BCHG #0,X:$12	Bit test and toggle
BCLR #0,X:$12	Bit test and clear
BSET #0,X:$12	Bit test and set
BTST #0,X:$12	Bit test
CLR A	Clear accumulator
EOR X0,A	Exclusive OR A1:= A1 xor X0
LSL A	Logic shift left A1:= 2A1
LSL #10,a	LSL multi-bit A1:= A1<<10
LSR A	Logic shift right A1:= ½ A1
LSR #10,a	LSR multi-bit A1:= A1>>10
NOT A	Logical complement A1:= A1\
OR X0,A	Logical OR A1:= A1 v X0
ROL A	Rotate A1 left
ROR A	Rotate A1 right

2.4.3 **Data transfer instructions**:

Data are 8, 16, 24, 48, or 56 bit long.
Automatic saturation / dynamic scaling if a full accumulator is the source.
Sign extension / force LSBs to 0 for 24b source if a full accumulator is the destination.

LUA	(R0)+N0,R1	Load updated addr R1=R0+N0
LRA	label,R0	Load PC-relative addr R0:=Label
MOVE	sce,dest	Copy source to destination
MOVEC	#0,SP	Control register move
MOVEM	X0,P:(R0)+	Program memory move
MOVEP	Y:$FFFFFF,X:0	Peripheral I/O move
TFR	X0,A	Register to accu transfer

2.4.4 Program control instructions

Bcc label	Branch (rel) if condition cc is true
BRA label	Branch always (PC relative)
BRCLR #3,Y:$FFFF,label	Branch if bit clear
BRSET #3,Y:$FFFF,label	Branch if bit set
BRKcc	If cc true, exit current DO loop
BScc label	Branch to subroutine if cc is true
BSR label	Branch to subroutine
DEBUG	Enter JTAG debug mode
DEBUGcc	Enter debug mode if cc is true
DO X0,label	Start DO loop up to absolute label
DOR #n,label	Start DO loop up to relative label
DOFOREVER label	Start infinite DO loop, absolute
DORFOREVER label	Start infinite DO loop, PC rel.
ENDDO	Abort current DO loop
IFcc	(parallel field) Conditional execution
Jcc label	Absolute jump if condition cc is true
JCLR #1,X:0,label	Absolute jump if bit clear
JMP label	Absolute jump
JScc label	Absolute jump to SR if cc is true
JSCLR #1,X:0,label	Abs jump to subroutine if bit clear
JSET#1,X:0,label	Absolute jump if bit set
JSR label	Jump to subroutine
JSSET #1,X:0,label	Absolute jump to subroutine if bit set

NOP	No operation
PLOCK	**|**
PUNLOCK|	
PLOCKR	**|**
PUNLOCKR	Program cache managment
PFREE	**|**
PFLUSH	**|**
PFLUSHUN	**|**
REP #n	Repeat next instruction
RESET	Initialize peripherals
RTI	Return from interrupt
RTS	Return from subroutine
STOP	Low power standby, clocks halted
TRAP	Software interrupt
TRAPcc	Trap if condition cc true
WAIT	Wait for interrupt (low power stdby)

2.5 DSP56000 / 56300 addressing modes

Addressing modes	Assembler syntax
Register direct	
Accumulators	A, A2, A1, A0, A10, AB B, B2, B1, B0, B10
Input registers	X, X1, X0, Y, Y1, Y0
Control registers	PC, SR, LA, LC, SSH, SSL, SP, SC, SZ, OMR, VBA
Address register	Rn, Nn, Mn n = 0 .. 7
Address register indirect	
No update	(Rn)
Postincrement by 1	(Rn)+
Postdecrement by 1	(Rn)-
Postincrement by offset Nn	(Rn)+Nn
Postdecrement by offset Nn	(Rn)-Nn
Indexed by offset Nn	(Rn+Nn)
Predecrement by 1	-(Rn)
Short / long displacement	(Rn+displ)
PC relative	
Short / long displacement, PC relative	(PC+displ)
Address register	(PC+Rn)
Special	
Short immediate	#[<]dd
Long immediate	#>dddddd
Absolute address	{X|Y|L|P}:aaaaaa
Absolute short address	[<]{X|Y|L}:aa
Short jump address	[<]aaa
I / O short address	[<<]{X|Y}:aa
Implicit	

2.6 DSP56000/56300 Instruction syntax

[Label] Operator {Operands} [Parall. mv1] [Parall. mv2] [;Comments]

```
conv    MAC     X0,Y0,A     X:(R0)+,X0   Y:(R4)-,Y0   ; convolution
```

Notes:

- Labels must be unique within the label field, they must begin with an alphabetic character.
- There must be a white space before the operator.
- There must be no space between operands
- Not all instructions allow parallel moves
- Read the DSP563xx Family manual to know which register combinations may be parallel moved

2.7 Parallel moves:

Parallel move type	Examples
No parallel move	MPY X0,Y1,A
I Short immédiate	ABS B **#$18,R1**
R Register to register	MAC -X0,Y0,A **Y1,N5**
U Addr. register update	RND B **(R3)+N3**
X: X memory	ASL A **X0,X:(R2)-**
X:R X memory and register	CMP Y0,A **A,X:(R0)+ B,Y0**
I R Immediate and register	MAX A,B **#0.12345,X0 A,Y0**
Y: Y:memory	EOR X0,B **A1,Y:$10**
R Y: Register and Y:memory	ADD B,A **B,X1 Y:(R6)-N6,B**
L: Long memory (48b)	SUB X0,B **AB,L:(R0)**
X: Y: XY memory	MPYR X1,Y0,A **X1,X:(R0)+ Y0,Y:(R4)+N4**
IFcc Conditionally execute	MPY X0,X1,A **IFEQ**

EXAMPLE: Vector dot product U.V

U = [0.1 0.2 0.3 0.4] V = [0.5 0.6 0.7 0.8]

R0 pointing U in the X: field R4 pointing V in the Y: field

```
PROD   CLR      A        X:(R0)+,X0   Y:(R4)+,Y0
       REP      #3
       MAC      X0,Y0,A  X:(R0)+,X0   Y:(R4)+,Y0
       MACR     X0,Y0,A
```

2.8 Assembler functions (for generating constants)

1 Mathematical Functions

@ABS - Absolute value
@ACS - Arc cosine
@ASN - Arc sine
@AT2 - Arc tangent
@ATN - Arc tangent
@CEL - Ceiling function
@COH - Hyperbolic cosine
@COS - Cosine
@FLR - Floor function
@L10 - Log base 10
@LOG - Natural logarithm
@MAX - Maximum value
@MIN - Minimum value
@POW - Raise to a power
@RND - Random value
@SGN - Return sign
@SIN - Sine
@SNH - Hyperbolic sine
@SQT - Square root
@TAN - Tangent
@TNH - Hyperbolic tangent
@XPN - Exponential function

2 Conversion Functions

@CVF - Convert integer to floating point
@CVI - Convert floating point to integer
@CVS - Convert memory space
@FLD - Shift and mask operation
@FRC - Convert floating point to fractional
@LFR - Convert floating point to long fraction
@LNG - Concatenate to double word
@LUN - Convert long fractional to float
@RVB - Reverse bits in field
@UNF - Convert fractional to floating point

3 String Functions

@LEN - Length of string
@POS - Position of substring in string
@SCP - Compare strings

4 Macro Functions

@ARG - Macro argument function
@CNT - Macro argument count
@MAC - Macro definition function
@MXP - Macro expansion function

5 Assembler Mode Functions

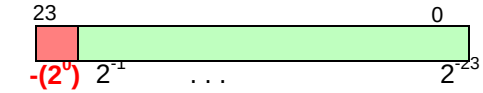
@CCC - Cumulative cycle count
@CHK - Current instruction/data checksum
@CTR - Location counter type
@DEF - Symbol definition function
@EXP - Expression check
@INT - Integer check
@LCV - Location counter value
@LST - LIST directive flag value
@MSP - Memory space
@REL - Relative mode function

2.9 Example: create a sine table in memory

```
; Example of using the assembler functions
; How to generate a 256 point   [0.. 2p[  sine table in Y: data memory


N       equ     256             ; define N
incr    equ     2.*pi/@cvf(N)   ; value of angle increment (N converted to floating value)
angle   set     0.0             ; angle = compilation variable

        org  y:                 ; current memory field
sine                            ; label at begin of table
        dup     256             ; duplicate following lines 256 times (preprocessor operation)
        dc      @sin(angle)     ; create sine value in Y: memory
angle   set     angle+incr      ; update angle value
        endm                    ; end duplicating
```

2.10 DSP56k Fractional Arithmetic

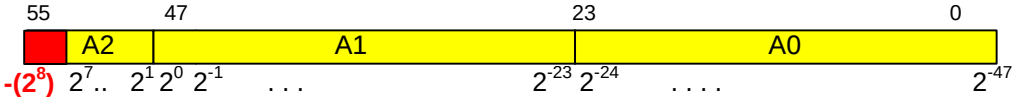
24b memory / register



Sign bit

m = -1.0 ... +(1.0 - 2^{-23})

Accumulator



Sign bit

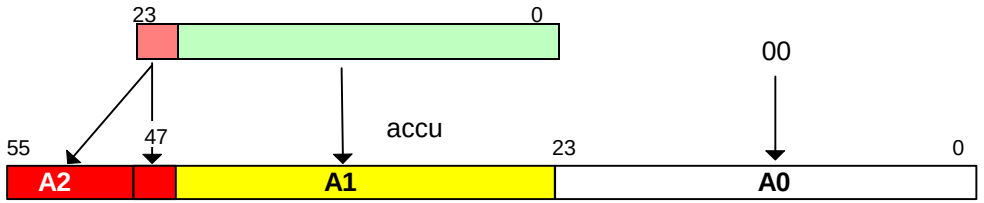
a = -256 ... +(256 - 2^{-47})

Transfer from memory to accumulator:

24 bit examples:

MOVE X:(R0),A
MOVE Y1,A

24b memory / register

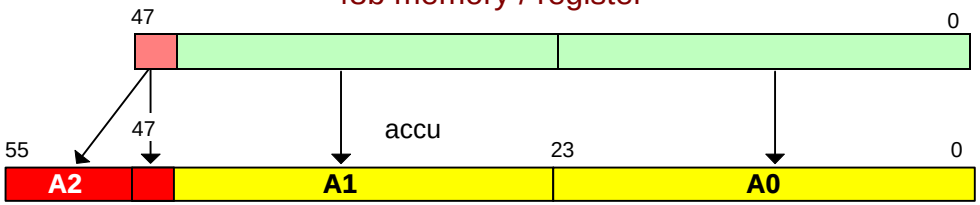


Sign extension

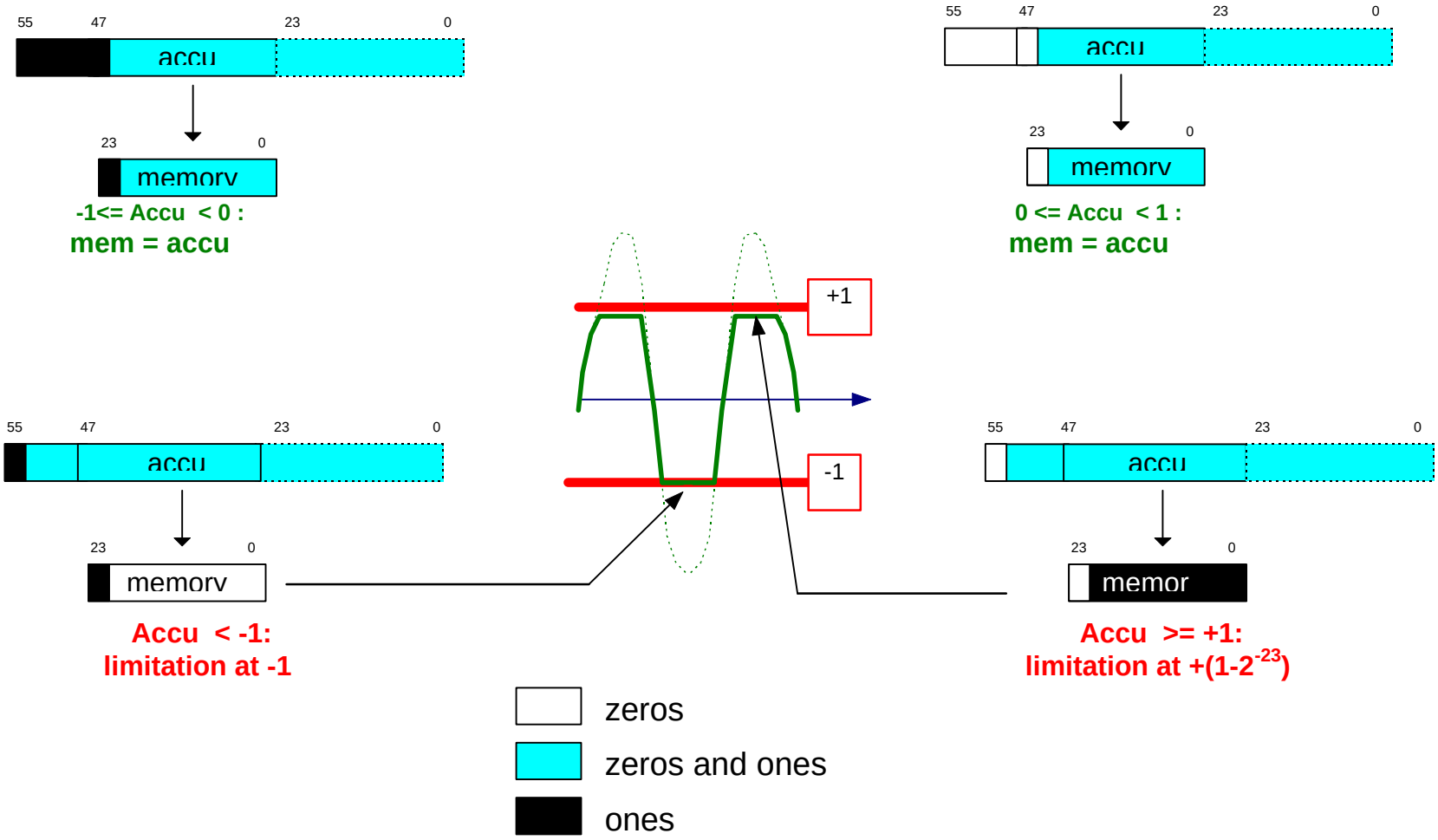
48 bits example:

MOVE L:(R0),A
MOVE Y,A

48b memory / register



2.11 Transfer from accumulator to memory (saturation logic)



2.12 DSP56000 / 56300 address modifier types

Mn	Modifier type
000000	Reverse carry mode. When incrementing Rn, carry bit propagates in the reverse direction. If Nn = 2^k then (Rn)+Nn generates mirror binary sequence
xx0001	Circular, modulo 2
xx0002	Circular, modulo 3
...	... < modulo any value > ...
xx7FFF	Circular, modulo 32768
xx8001	Circular, multiple wrap-around, modulo 2
xx8003	Circular, multiple wrap-around, modulo 4
...	... <only powers of 2's modulo> ....
xxBFFF	Circular, multiple wrap-around modulo 16384
xxFFFF	Linear (same as classical µP's)

2.13 Modulo addressing:

Buffer base address must be **p.2^{k}** with 2^{k} > Mn
Mn = **modulo value - 1** (= maximum buffer relative address)

Example:
Program R0 for **modulo 5** addressing
→ **M0 = 4**
→ Buffer base address must be 0, 8, 16, 24 ... 8p

(R0)+ sequence:

8 9 10 11 12 8 9 10 11 12 ...

(R0)+N0 sequence with **N0=3**

8 11 9 12 10 8 11 9 12 10 ...

2.14 QUIZ 2

The DSP563xx core

1. Why have DSP56xxx processors 56 bit wide accumulators ?

2. In accumulators A or B, give weights of following bits: 0, 24, 48, 54,55

3. Accumulator A = $00 123456 789ABC, and accumulator B = $FE DCBA98 765432
   give the values of concerned memory contents after following instructions:
   a) move a,x:0 b) move a,l:1 c) move b,x:2 d) move b1,x:3

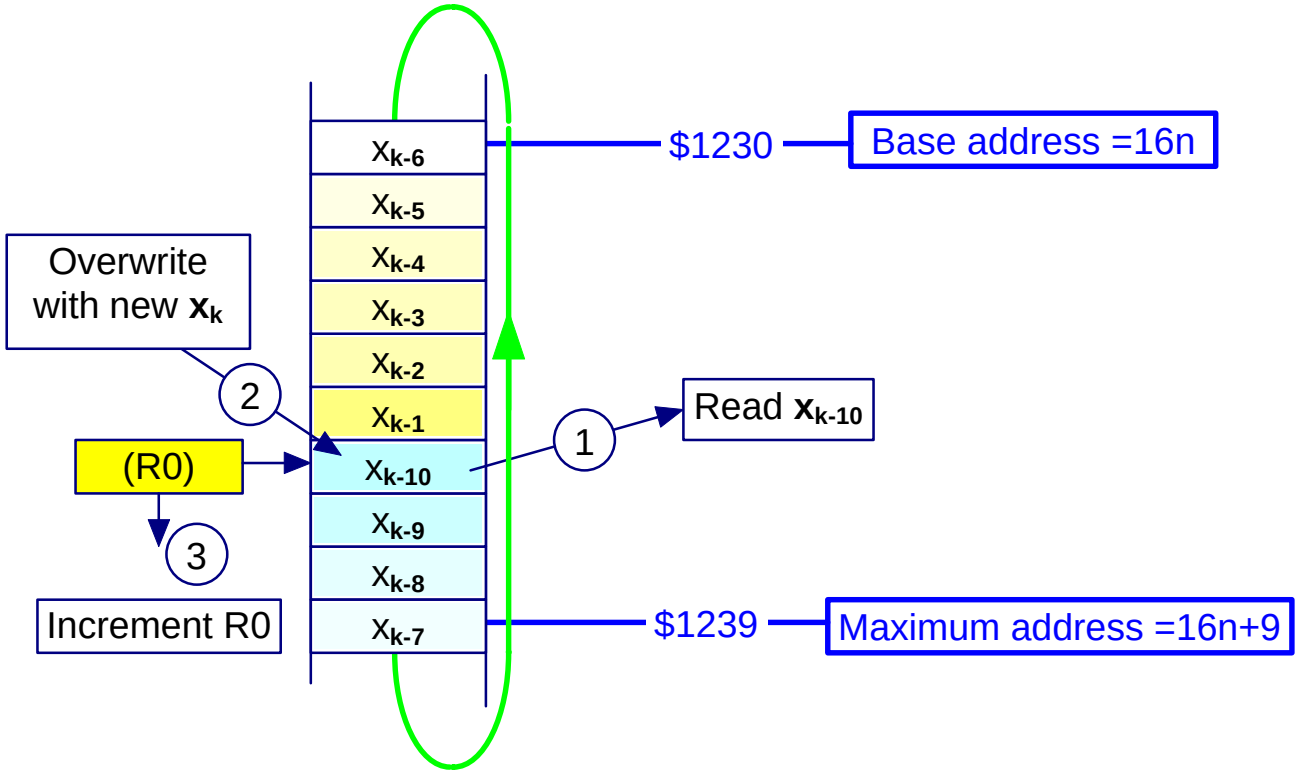
4. Address registers are initialized as following: R0=0 M0=$ffffff N0=5 R1=8 M1=4 N1=2.
   Give the address sequences generated by following addressing modes:
   a) (r0)+ b) (r0+n0) c) (r0)+n0 d) (r1)+ e) (r1)+n1

5. Optimize execution speed of following code segment:

```
        do      #100,end
        move    x:(r0)+,x0
        move    y:(r4)-,y0
        mac     x0,y0,a
end                                  (present execution time: 303 cycles)
```

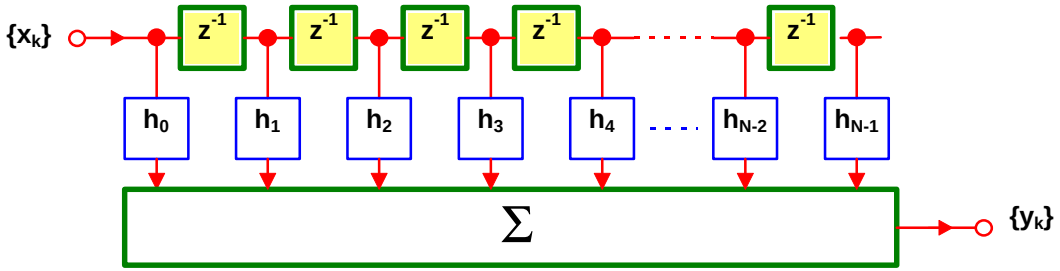
3. REAL TIME ALGORITHM IMPLEMENTATION

3.1 Delay line implementation

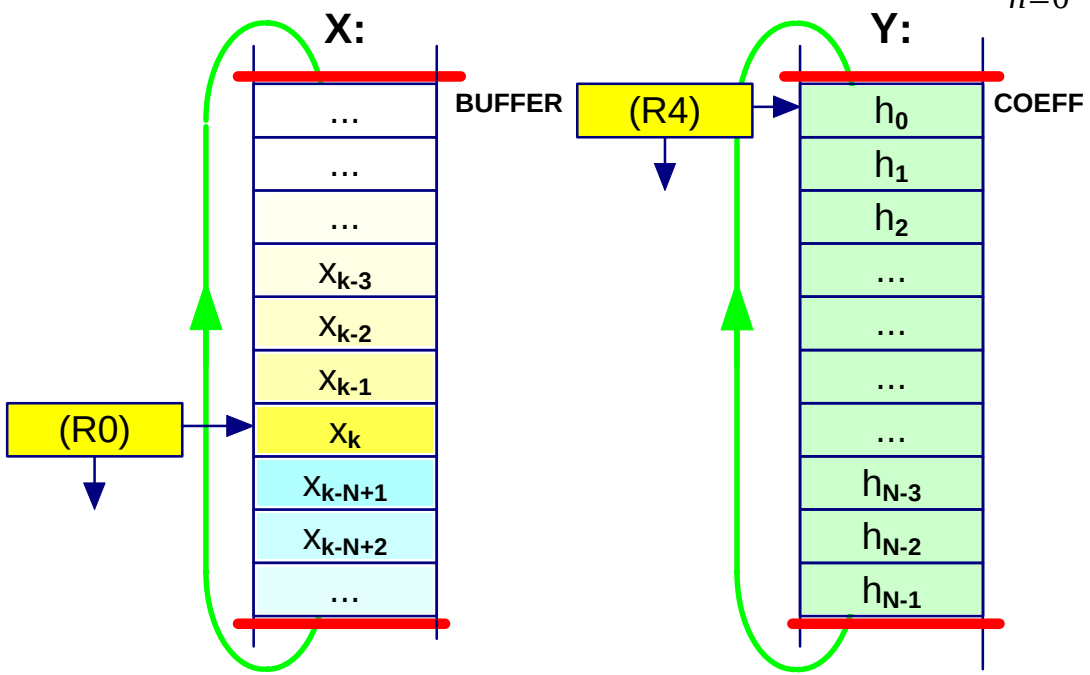


```
;       z-10 delay line routine  (a is line input, b is line output):


        org     x:          ; origin = current address in the X: data field

buffer  dsm     10          ; reserve N=10 words from the next 16n boundary
                            ; ( Define Storage Modulo 10 )

        org     p:          ; origin = current address in the P: program field
init    move    # buffer,r0 ; Init routine (to be called once at the begining)
        move    # 9,m0       ; r0 points to buffer base address, m0 has value N-1
        rts                 ; end of init routine

delay   move    x:(r0),b    ; read buffer's oldest sample
        move    a,x:(r0)+   ; write new sample over the oldest one, increment r0
        rts                 ; end of delay routine
```

3.2 Finite Impulse Response filters (digital convolution)



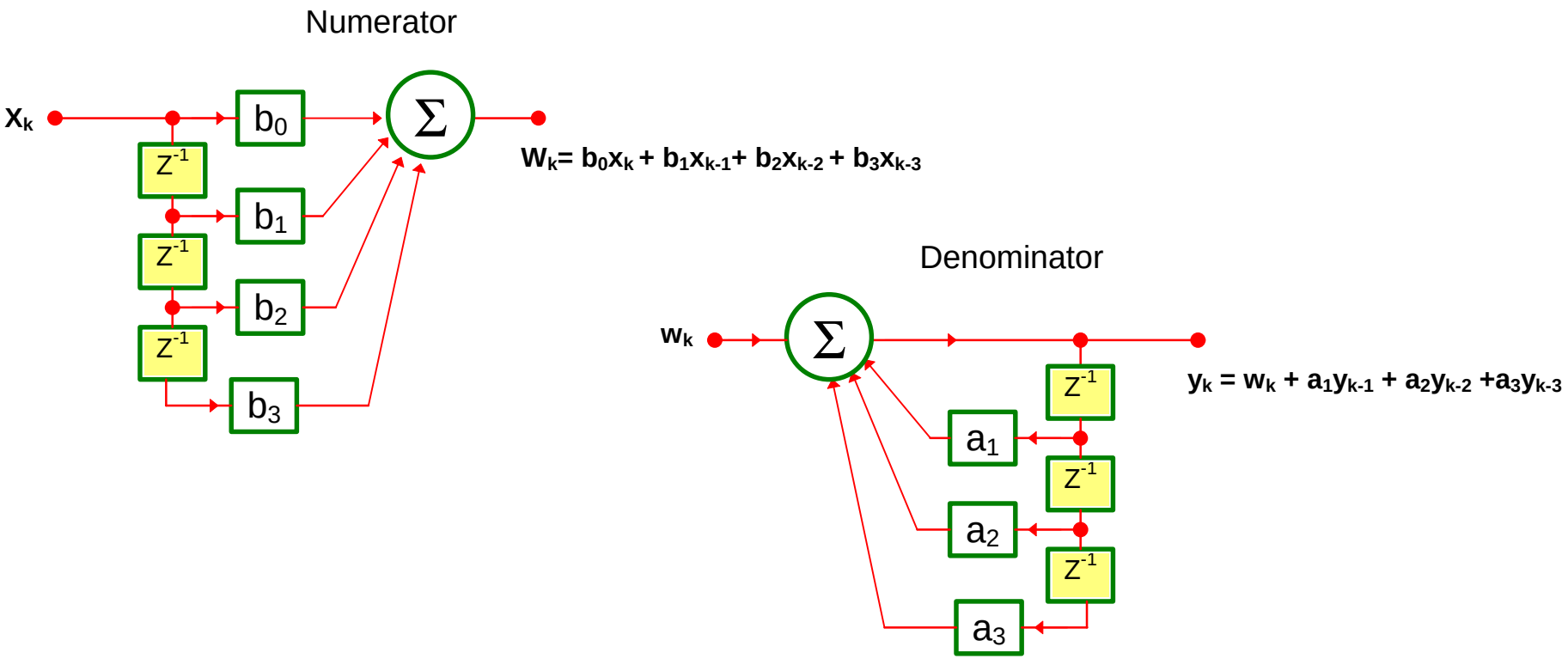
$$y_k = \sum_{n=0}^{N-1} h_n x_{k-n}$$



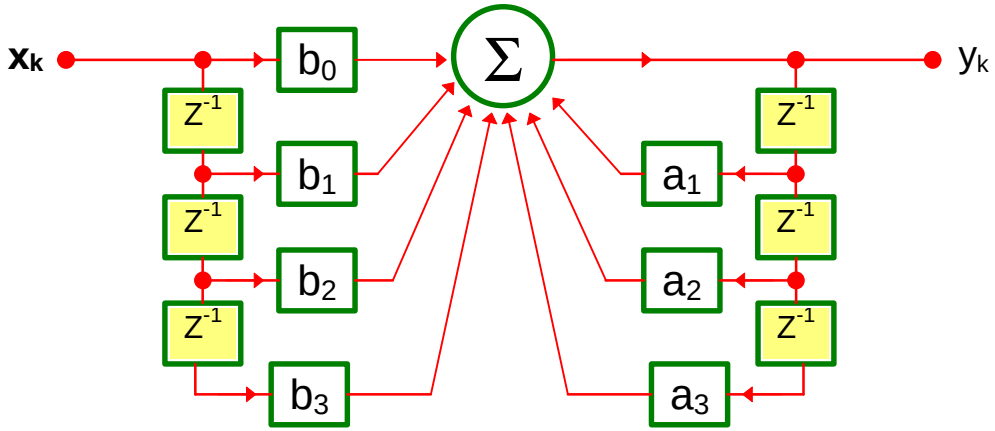

```
FIR   move    #buffer,r0
      move    #(N-1),m0
      move    #coeff,r4
      move    # m0,m4

loop  jsr     get_input
      move    a,x:(r0)
      clr     a           x:(r0)-,x0      y:(r4)+,y0
      rep     #(N-1)
      mac     x0,y0,a     x:(r0)-,x0      y:(r4)+,y0
      macr    x0,y0,a     (r0) +
      jsr     write_output
      jmp     loop
```

3.3 Recursive filters (Infinite Impulse Response)

$$\frac{Y(z)}{X(z)} = H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 - a_1 z^{-1} - a_2 z^{-2} - a_3 z^{-3}} = \frac{N(z)}{D(z)}$$



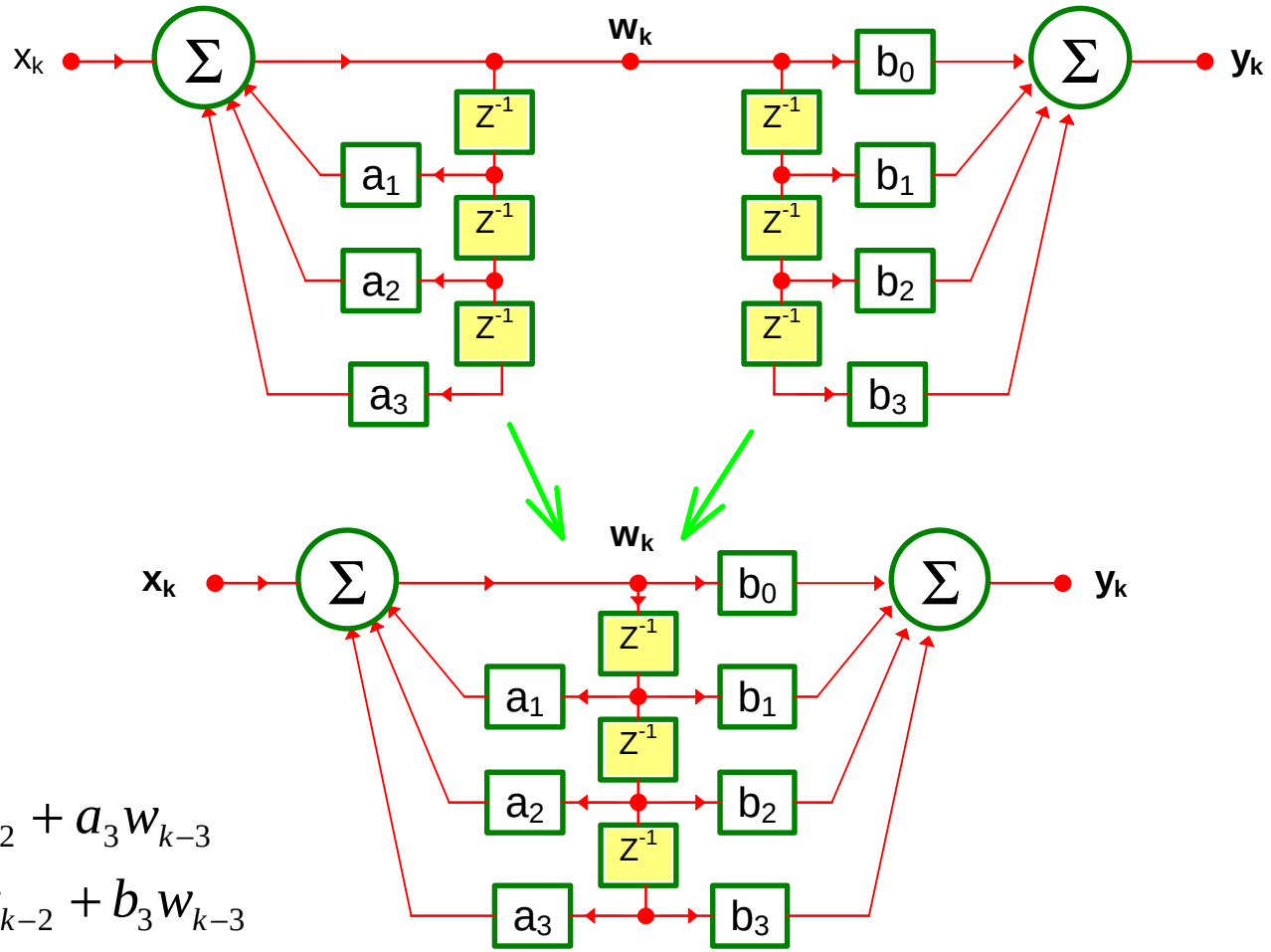
IIR, direct form



$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 - a_1 z^{-1} - a_2 z^{-2} - a_3 z^{-3}}$$

$$y_k = b_0 x_k + b_1 x_{k-1} + b_2 x_{k-2} + b_3 x_{k-3} + a_1 y_{k-1} + a_2 y_{k-2} + a_3 y_{k-3}$$

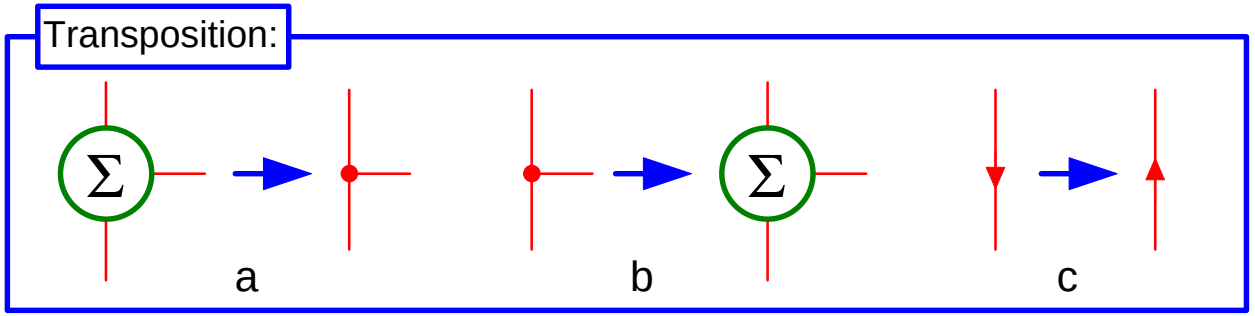
IIR, canonic form



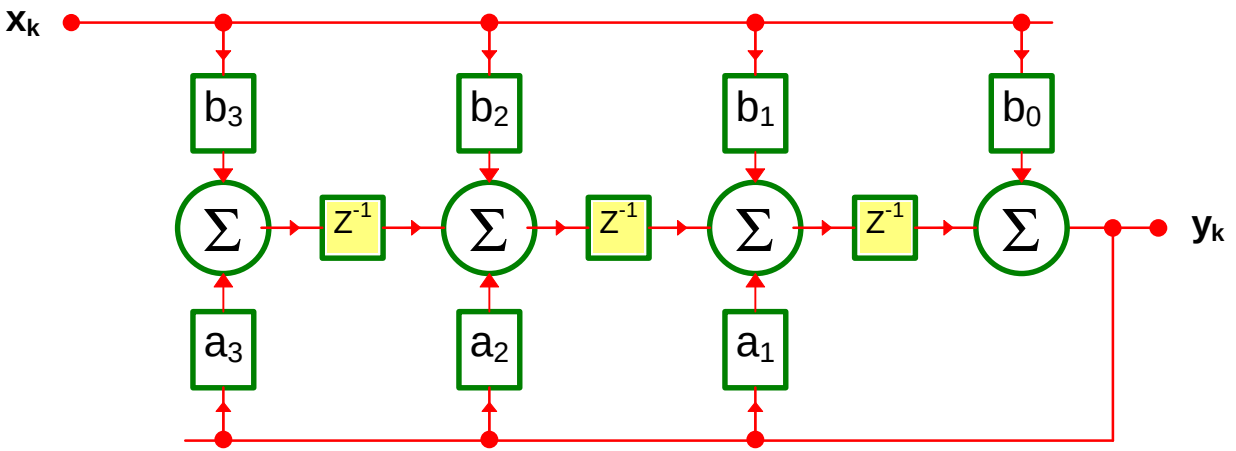
$$w_k = x_k + a_1 w_{k-1} + a_2 w_{k-2} + a_3 w_{k-3}$$

$$y_k = b_0 w_k + b_1 w_{k-1} + b_2 w_{k-2} + b_3 w_{k-3}$$

IIR, transposed form



Transposing the canonic form:



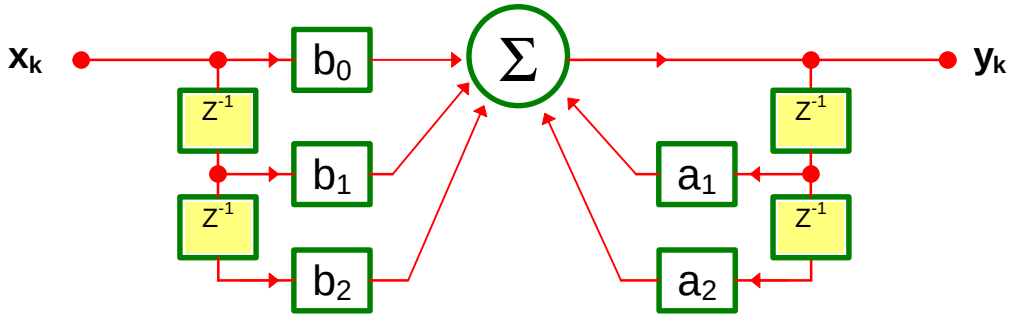
$$Y(z) = b_0.X + z^{-1}(b_1.X + a_1.Y + z^{-1}(b_2.X + a_2.Y + z^{-1}(b_3.X + a_3.Y)))$$

$$u_k = b_3 x_{k-1} + a_3 y_{k-1} \qquad v_k = u_{k-1} + b_2 x_{k-1} + a_2 y_{k-1}$$

$$w_k = v_{k-1} + b_1 x_{k-1} + a_1 y_{k-1} \qquad y_k = w_k + b_0 x_k$$

3.4 2nd order IIR direct form DSP56k implementation

Coefficients greater than 1



Filter type ($f_0 = F_s / 8$)	**a1**	**a2**	**b0**	**b1**	**b2**
Low-pass, Q=1.2	1.056	-0.493	0.109	0.218	0.109
High-pass, Q=1.2	1.056	-0.493	0.637	-1.27	0.637
Band-pass, Q=100	1.409	-0.992	0.004	0.000	-0.004
Band-stop, Q=100	1.409	-0.992	0.996	-1.409	0.996

Coefficients which are out of fractional range

Dynamic scaling allows handling coefficients in the [-2 ..+2[range:

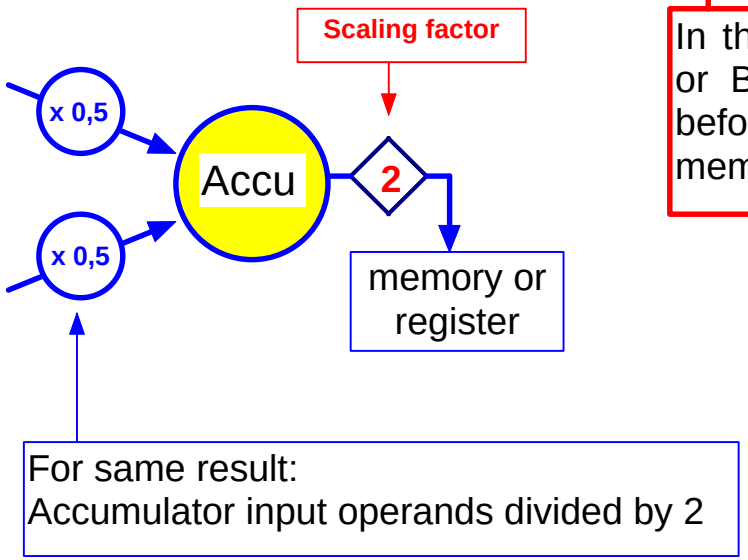
Solution:
Dynamic scaling

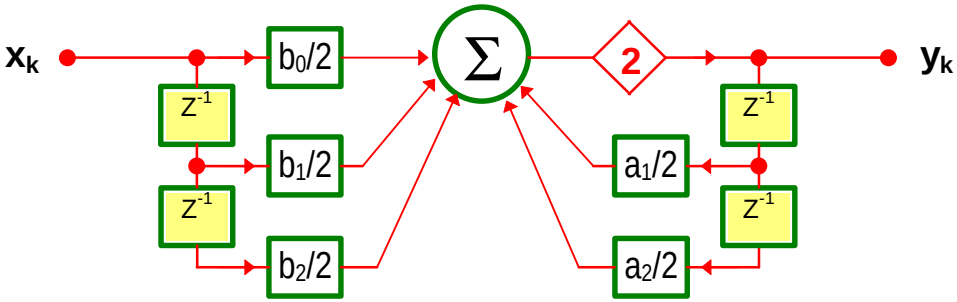
Status Register SR:

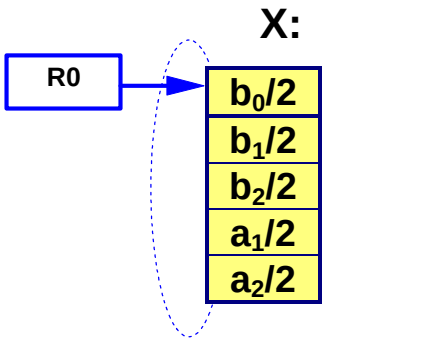
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LF	DM	T	-	S1	S0	I1	I0	S	L	E	U	N	Z	V	C

1 0

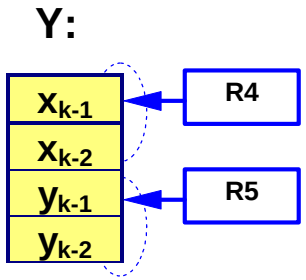
In this mode, accumulators A or B are shifted left one bit before being written into a memory or register.







R0: modulo 5 addressing



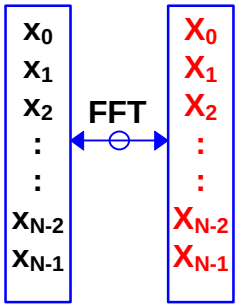
R4 and R5:
modulo 2 addressing

```
        ori     #8,mr                                   ; scaling mode on
        move    x:(r0)+,x0                              ; x0 = b0 / 2

loop    jsr     waitsample                              ; wait for new sample
        move    input,y1                                ; Y1 =  filter input
        mpy     x0,y1,a     x:(r0)+,x0  y:(r4)+,y0      ; A = b0/2.xk   X0=b1/2 Y0=xk-
        mac     x0,y0,a     x:(r0)+,x0  y:(r4),y0       ; A+=b1/2.xk-1 X0=b2/2 Y0=xk-2 R4→xk-2
        mac     x0,y0,a     x:(r0)+,x0  y:(r5)+,y0      ; A+=b2/2.xk-2 X0=a1/2 Y0=yk-1 R5→yk-2
        mac     x0,y0,a     x:(r0)+,x0  y:(r5),y0       ; A+=a1/2.yk-1 X0=a2/2 Y0=yk-2
        macr    x0,y0,a     x:(r0)+,x0  y1,y:(r4)       ; A+=a2/2.yk-2 X0=b0/2 xk-2 replaced by xk
        move    a,x1        a,y:(r5)                    ; x1 = 2A = yk    yk-2 replaced by yk
        move    x1,output                               ; x1 = filter output
        jmp     <loop
```

3.5 The Fast Fourier Transform (FFT)

3.5.1 Number of operations in DFT calculation

DFT: $\{x_k\}$ « $\{X_n\}$ k = 0 ... N-1 n = 0 ...N-1

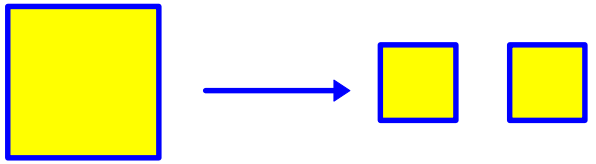


$$X_n = \sum_{k=0}^{N-1} x_k . W_N^{nk} \qquad \text{with} \qquad W_N = e^{\frac{-j2p}{N}}$$

N^2 complex multiplications + accumulations

Cooley & Tuckey's idea (1963):

Replace one N point DFT with two N/2 points DFT's
Hence,



N^2 operations $2.(N/2)^2$ operations

➔ gain = factor 2

Iteration until N one point DFTs feature $N.\log_2(N)$ operations. Note: *N must be a power of 2*

Example: N = 1024 N^2 = 1 048 576 $N.\log_2(N)$ = 10240 ➔ Gain: 102,4

3.5.2 The Decimation In Time (DIT) algorithm

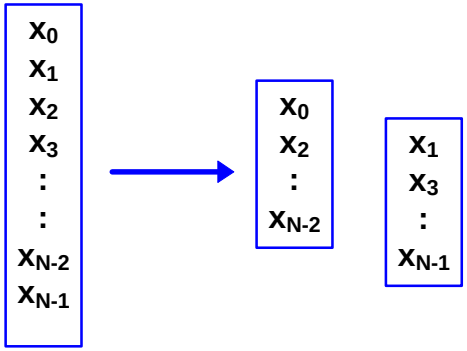
$$\{x_k\} = \{x_{2p}\} + \{x_{2p+1}\}$$

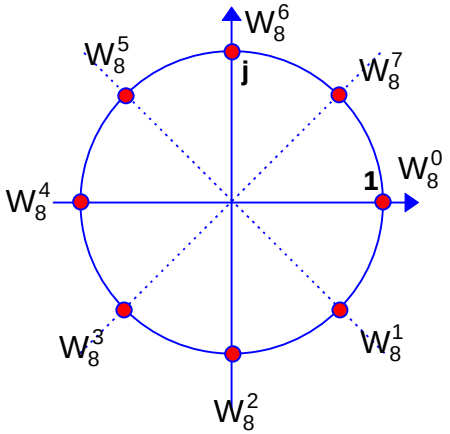
$$X_n = \sum x_{2p}.\,(W_N)^{2pn} + \sum x_{2p+1}.\,(W_N)^{(2p+1)n} \qquad p = 0 \ldots (N-1)/2$$

$$(W_N)^{2pn} = (W_{N/2})^{pn}$$

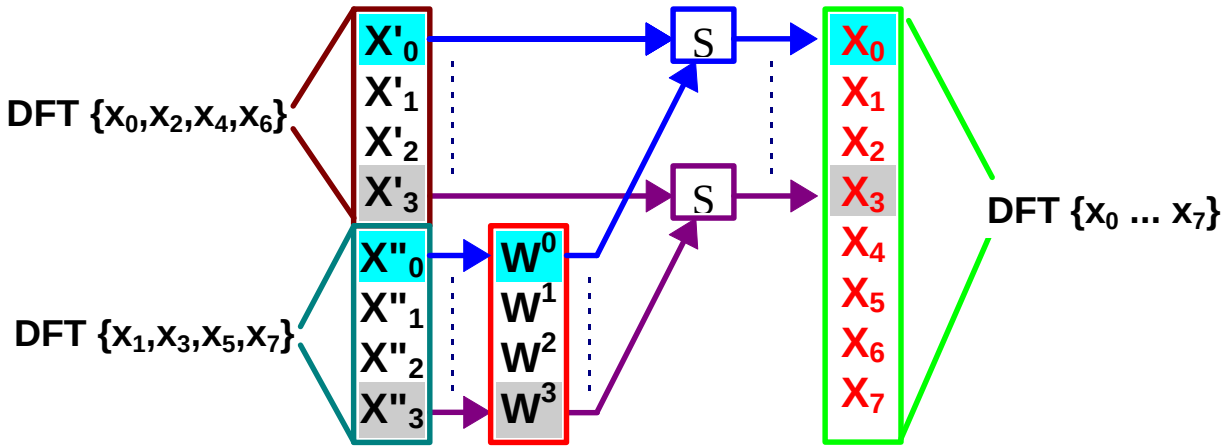
$$X_n = \sum x_{2p}.\,(W_{N/2})^{pn} + (W_N)^n.\sum x_{2p+1}.(W_{N/2})^{pn} \qquad p = 0 \,..\, (N-1)/2$$

➔ DFT $\{x_k\}$ = DFT $\{x_{2p}\}$ + W_N^n.DFT $\{x_{2p+1}\}$

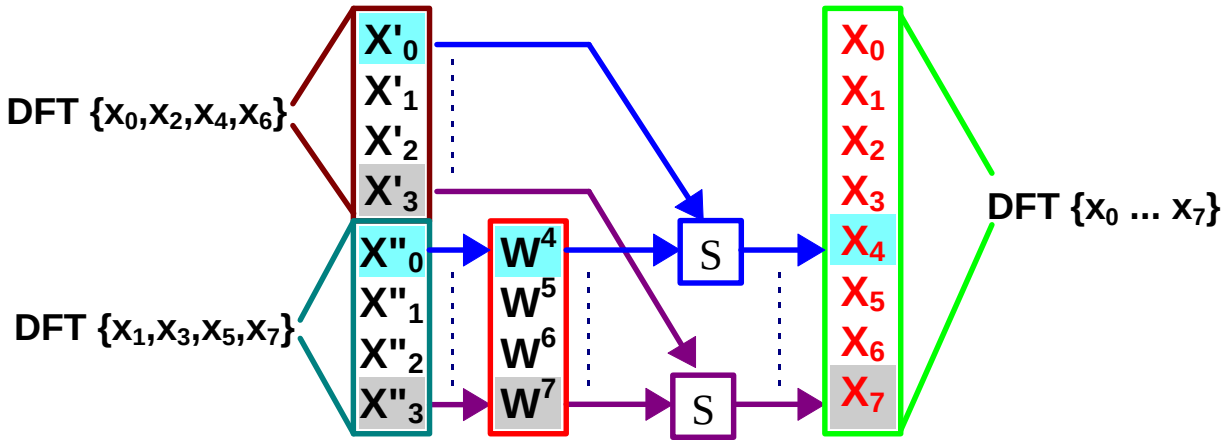




8 point DFT: getting $X_0..X_4$ from two 4 point DFTs



Getting next 4 points: [note that the 4 point DFTs are periodic]

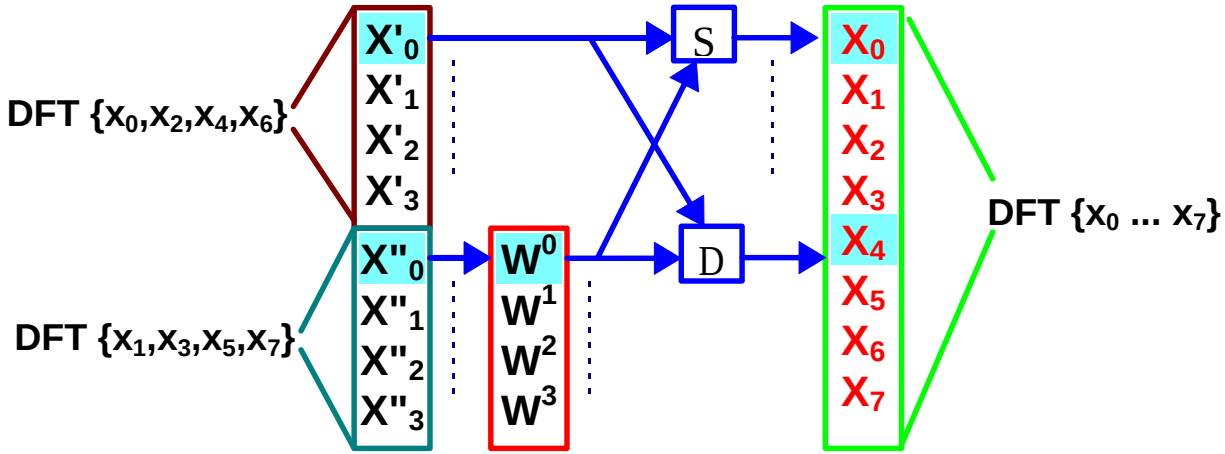


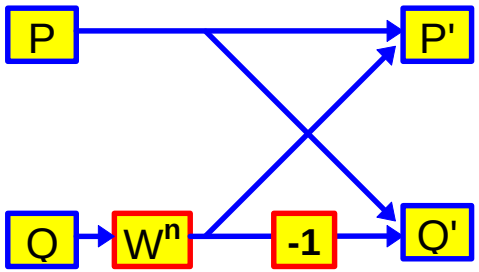
Twiddle factors symetry

$W^4 = -W^0$ $W^5 = -W^1$ $W^6 = -W^2$ $W^7 = -W^3$

➔ **Only N/2 factors are required**

The FFT butterfly



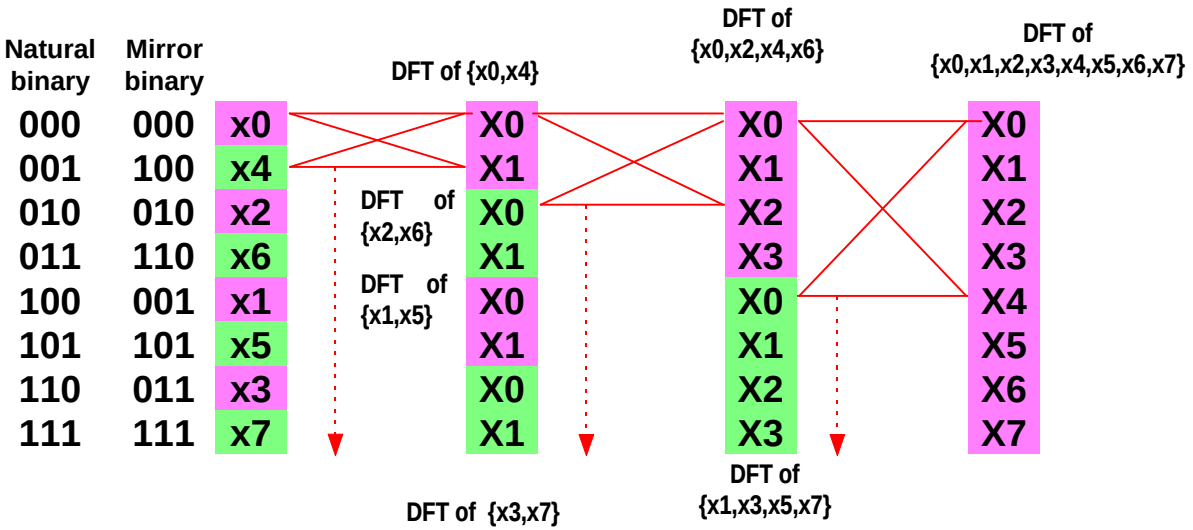


DIT butterfly:

$P' = P + W^n.Q$

$Q' = P - W^n.Q$

FFT recursion



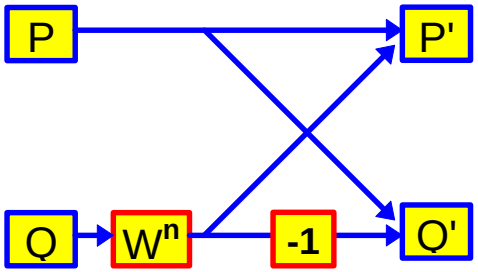
Recursion until the DFT's size is one

Mirror binary ordered time domain samples yield normally ordered frequency domain samples

or

Normally ordered time domain samples yield mirror binary ordered frequency domain samples.

3.6 DSP56k implementation of the DIT FFT butterfly



DIT butterfly:

$P' = P + W^n.Q$
$Q' = P - W^n.Q$

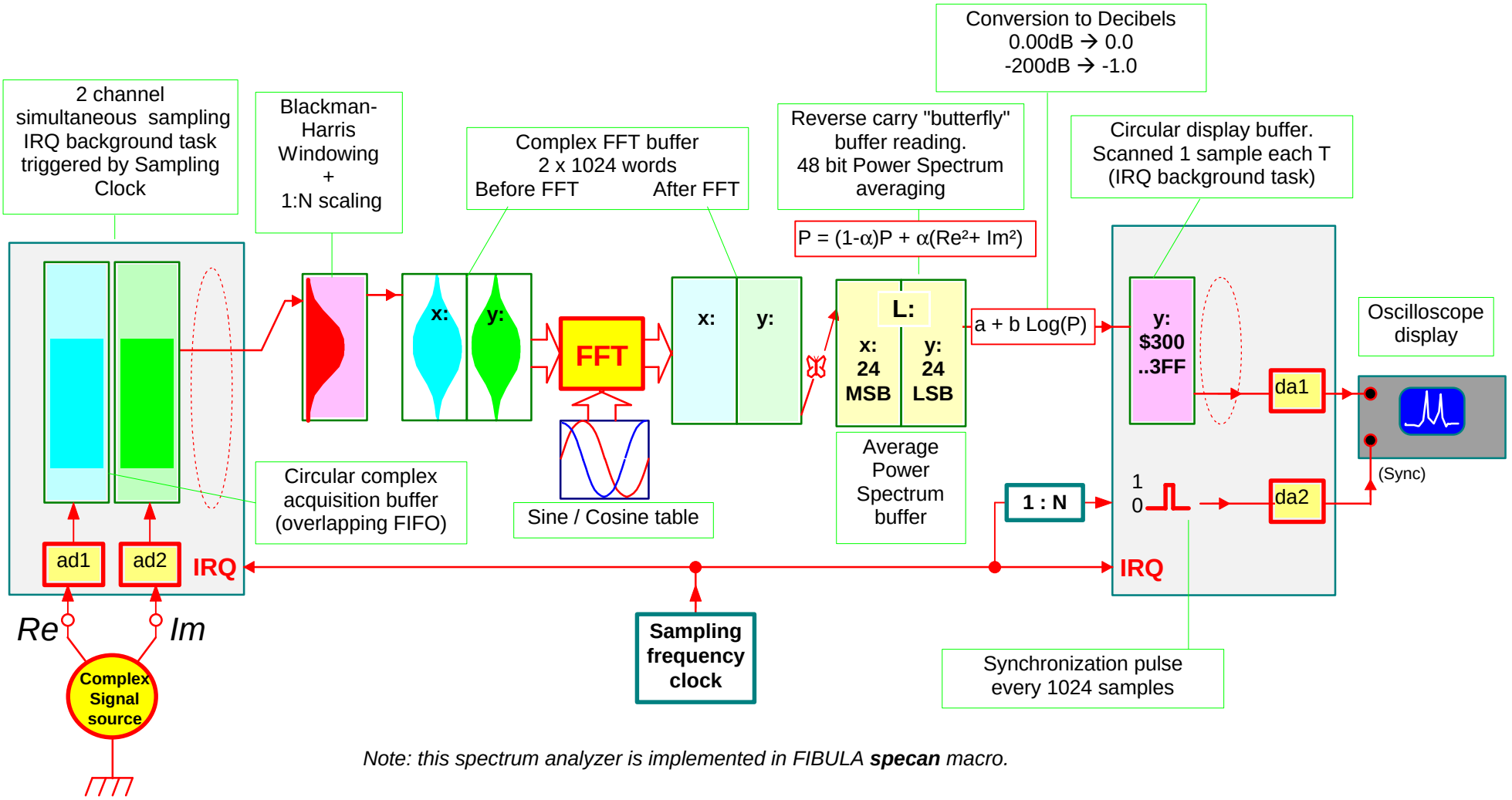
P'r = Pr + Wr.Qr - Wi.Qi
P'i = Pi + Wr.Qi + Wi.Qr
Q'r = Pr - Wr.Qr + Wi.Qi = 2Pr - P'r
Q'i = Pi - Wr.Qi - Wi.Qr = 2Pi - P'i

Pointer	X:	Y:
r0	Pr	Pi
r1	Qr	Qi
r6	wr	wi
r4	P'r	P'i
r5	Q'r	Q'i

Note:
In place FFT
P' and Q' replace P and Q
Normally r4, r5 are same as r0, r1
Due to pipeline programming, r0, r1 are preloaded while r4,r5 keep the previous address

```
; DSP56K   FFT  butterfly

        move        x:(r1),x1                       y:(r6),y0
        move        x:(r5),a                        y:(r0),b
        move        x:(r6)+n6,x0

        mac         x1,y0,b                         y:(r1)+,y1
        macr        -x0,y1,b        a,x:(r5)+       y:(r0),a
        subl        b,a             x:(r0),b        b,y:(r4)
        mac         -x1,x0,b        x:(r0)+,a       a,y:(r5)
        macr        -y1,y0,b        x:(r1),x1
        subl        b,a             b,x:(r4)+       y:(r0),b
```

3.7 A real time spectrum analyzer for complex signals



*Note: this spectrum analyzer is implemented in FIBULA **specan** macro.*

3.8 QUIZ 3

Algorithm Implementation

3.1 What is the approximate execution time (cycles) of a 100 sample delay line ?

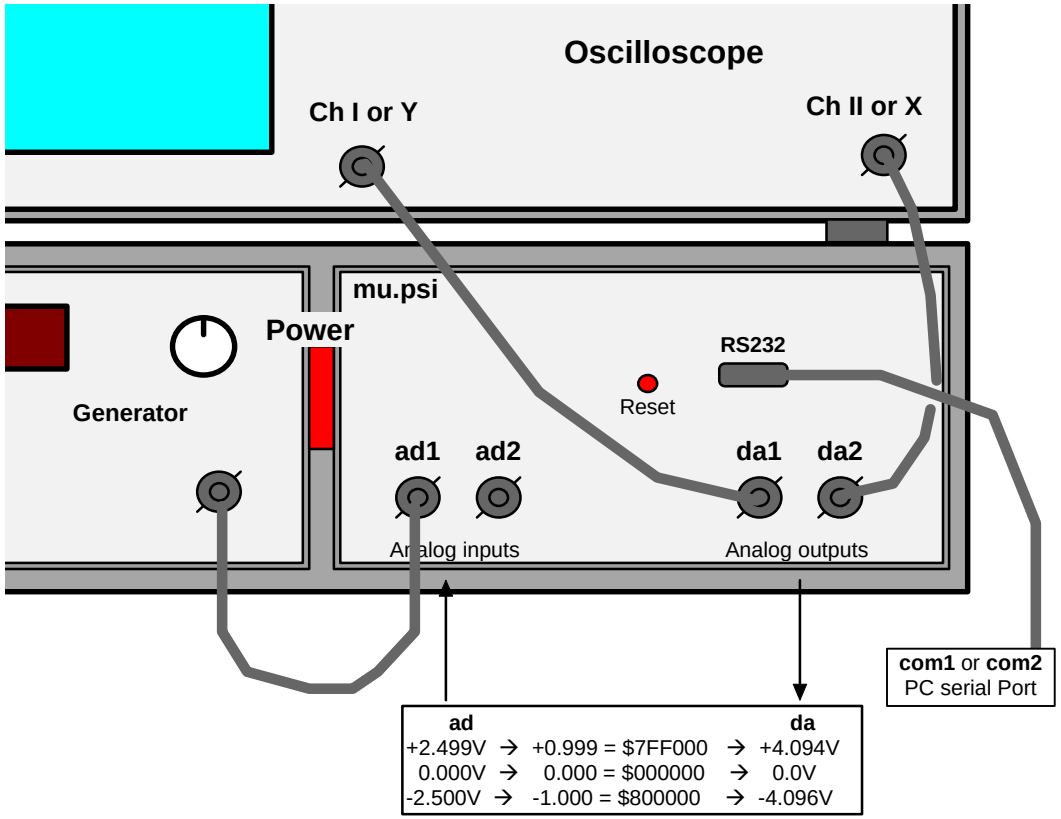
3.2 What are the advantages et drawbacks of non recursive (FIR) filters ?

3.3 How do you implement an N^{th} order IIR filter on a fixed point DSP ?

3.4 What tricks make FFT calculation faster on DSP563xx processors ?

4. LABORATORY EXPERIMENTS WITH MU.PSI AND FIBULA

4.1 Preparing the mu.psi and FIBULA DSP56309 pedagogic platform for ASSEMBLY LANGUAGE PROGRAMMING



Connect cables as represented.

Start **FIBULA**

Powerup the Hameg mainframe and the oscilloscope

Verify that FIBULA status LED turns GREEN
If not, verify connections and power, change serial port
hardware: displace cable, or:
software: **Settings| Serial port | Port |COM**x

For DSP56300 Assembly language programming, do following commands:

Settings | Compiler mode | ASM with startup

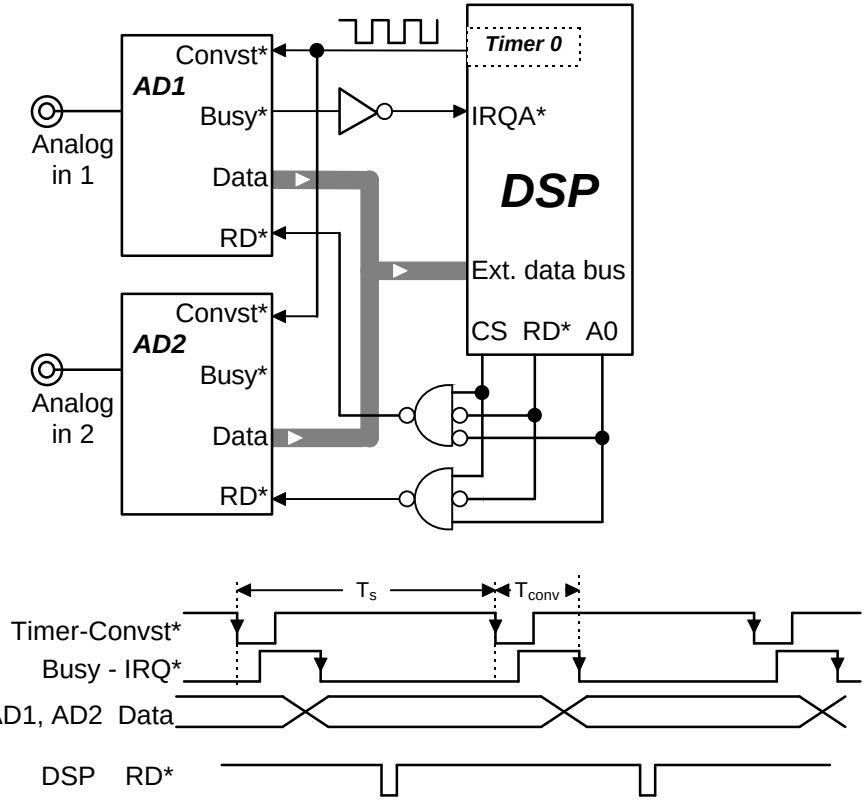
Help | View instructions | Assembly

Open the file **Fibula\asmlib\demo.asm**

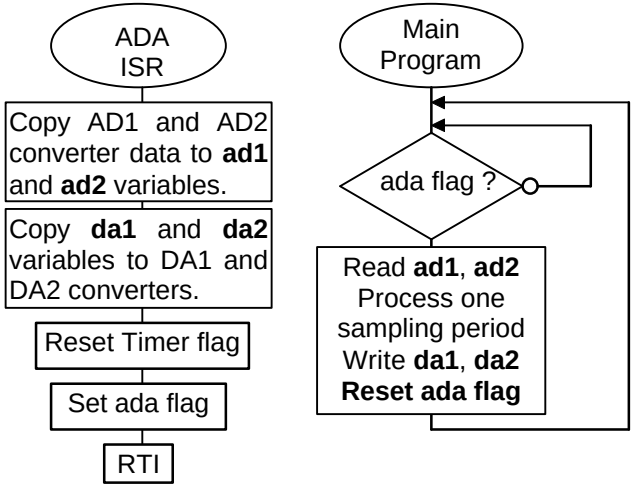
Run this demo to verify everything works.

4.2 The ADA macro for sampling analog I/O on Mu.Psi Card

The schematics hereunder describes the acquisition process. Either the ADC1 Busy signal connected to IRQA, or the Timer 0 interrupt can be used to call the ADA Interrupt Service Routine. This routine just copies H/W analog ports to/from S/W variables in memory, and sets a flag. The main program processes one sample each time a flag is set, and then resets this flag. Due to 1 level buffering, the processing time **Tp** available in main program for each sample may temporarily override the sampling period, provided the condition **Tp(k) + Tp(k-1)** £ **2(Ts-Tirq)** is respected.



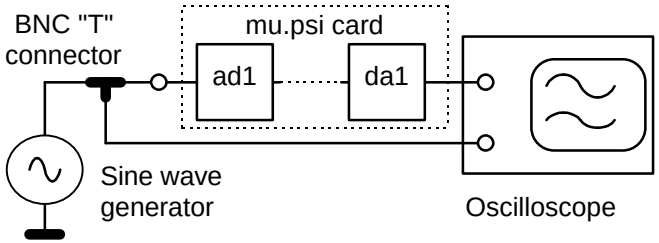
T_s Sampling period = 1µs .. 100ms
T_{conv} ADC conversion time = 0.9µs typ.
T_{IRQ} Interrupt execution time
Processing time available: **Tp(k) + Tp(k-1)** £ **2(Ts-Tirq)**




```
; USING THE ADA MACRO
; The macro installs inits, vector, and ISR
; The programmer just has to write the main program:

        org     p:          ; Don't force absolute address
loop    ada     1e5         ; Wait sample, Fs=100kHz
        move    x:ad1,a     ; Read ADC1 value
        ...
        <do some algorithm ...>
        ...
        move    a,x:da1     ; Write result to DAC1
        jmp     loop
```

Figure 4-1

4.3 Acquisition, scaling, quantification, aliasing




```
;       Simple AD to DA transfer
        org     p:              ; Current memory is program memory
loop    ada     1e5             ; Wait for Sample, Fs=100kHz
        move    x:ad1,a         ; Read ADC1 sample in A
        move    a,x:da1         ; Write sample to DAC1
        jmp     loop            ; Loop back
```

Figure 4-2

Connect a sine wave generator to Mu.Psi AD1 input, and to an oscilloscope input Y1, connect DA1 to oscilloscope input Y2, as represented above.
Connect the Mu.Psi serial port to a serial RS232 port on your PC.
Open FIBULA. Power on Mu.Psi or press the Reset button if power is on. The FIBULA status LED should turn green. If it doesn't, change serial port, either by changing the cable connection, or by doing a menu command such as: **Settings | Serial port | Port | COM2**
Set the compiler mode to Assembly by doing the menu command: **Settings | Compiler mode | ASM with startup**

Edit the small program represented above, then compile it, load it to the DSP, and run it by pressing this toolbar button 

The AD and DA 12 bits wide data busses are both connected to the 12 MSB of the DSP data bus, therefore, the DSP internal AD / DA data fractional representation is in the range [-1.0 ... +2047/2048].

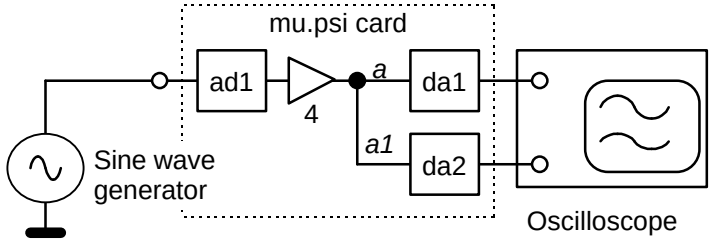
Choose a low frequency on the generator (100 .. 1000Hz). Set the signal level to minimum, then increase it until the Y2 oscilloscope trace begins to saturate.

- Determine the AD1 input voltage range ± Vin_max
- Determine the DA1 output voltage range ± Vout_max
- When processing input data, what represents a value of 0.3 ?
- What is the value of a quantification step *q* at the AD side, at the DA side ?
- For an input voltage of 1V RMS, calculate the (Signal power/Quantification noise power) ratio in decibel.

Choose a generator frequency of 101 kHz
- Are the 2 oscilloscope traces still similar ? Why ?
- What is the frequency observed on channel Y2 ? Explain.

4.4 Accumulator behavior

4.4.1 Saturation vs. overflow




```
;       Saturation vs. overflow
        org     p:              ; Current memory is program memory
loop    ada     1e5             ; Wait for Sample, Fs=100kHz
        move    x:ad1,a         ; Read ADC1 sample in A
        asl     #2,a,a          ; Shift A 2 bits left ⇔ multiply A by 4
        move    a,x:da1         ; Write full accumulator a to DAC1
        move    a1,x:da2        ; Write accumulator part a1 to DAC2
        jmp     loop            ; Loop back
```

Figure 4-3

When copying an entire accumulator (A or B), saturation logic applies, that is: an accumulator with value < -1.0 gives a value of -1.0, an accumulator with value >= +1 gives a value of $7FFFFF = 1.0 - ε.

When copying the A1 or B1 part of the accumulator, a bit-to-bit copy operation is performed, which provokes overflow if the accumulator value is <-1 or >= +1.

- Verify this behavior with the experiment described above.

4.4.2 Accumulator extension: integrate a sine wave over n full periods

If you integrate a sine wave for one period, you normally get 0, unless the signal has a DC offset. Given the sampling frequency Fs, the sine wave frequency F0, and it's amplitude 1.0, calculate the maximum value the integral takes during the integration process. Verify this, using this program: (Fs=100kHz, F0=1kHz, N=100)

```
;       Accumulator extension
        opt     dld                     ; Allow directives in do loops
        org     y:0
result  ds      1                       ; reserve memory in Y:
        org     p:                      ; Current memory is program
memory
        clr     a       #0,r0           ; A = 0
        clr     b
        #-1,m0
        do      #100,int1               ; Repeat 100 times until int1
        ada     1e5                     ; Wait for Sample, Fs=100kHz
        move    x:ad1,x0                ; Read ADC1 sample in x0
        add     x0,a                    ; Accumulate into A
        max     a,b                     ; get the max of A
int1
        move    a2,y:(r0)+              ; Write accumulators to memory
        move    a1,y:(r0)+
        move    b2,y:(r0)+
        move    b1,y:(r0)+
        bra     *
```

Figure 4-4

4.4.3 Fractional numbers vs. integers

The program represented right allows loading A, B, and X registers from the X: data memory on a bit to bit basis, then perform a 56 bit addition and a 24 bit * 24 bit multiplication. The results in A and B are written again in same memory locations.

Open the terminal  and use debugger commands to modify memory contents. For example to load A with $02 000123 000005 do:
x 0 <return>
2 <space> <space>
123 <space> <space>
5 <space> <space>

Then, run the program with
g 0 <return>
and stop it using the Reset button.
Observe the results in memory using :
x 0 <return> <space> <space> ....

Try these operations:
Fractional:
0.5 + 0.25 ($400000 + $200000)
0.5 * 0.25
-0.5 * 0.25 ($FF C00000 +$00 200000) Note: $FF in a2 is sign extension
Integer:
50 + 25 ($000032 + $000019)
50 * 25
-50 * 25
56 bit data:
$11 111111 111111 + $22 222222 222222
*Mixed (fractional * integer)*
50 * 0.5

```
;       Fractionals and integers

        org     x:0             ; X: data memory reservation, from address 0000
ah      ds      1               ; A2 value (NB never give a label a register name !)
am      ds      1               ; A1 value
al      ds      1               ; A0 value
bh      ds      1               ; B2 value
bm      ds      1               ; B1 value
bl      ds      1               ; B0 value
xh      ds      1               ; X1 value
xl      ds      1               ; X0 value

        org     p:              ; Current memory is P: program memory
        move    x:ah,a2
        move    x:am,a1
        move    x:al,a0
        move    x:bh,b2         ;Load registers A2,A1,A0,B2,B1,B0,X1,X0
        move    x:bm,b1         ; from X: data memory
        move    x:bl,b0
        move    x:xh,x1
        move    x:xl,x0

        add     a,b             ; Addition B = A + B
        mpy     x0,x1,a         ; Multiplication A = X0 * X1

        move    a2,x:ah
        move    a1,x:am         ; Copy accumulator registers to
        move    a0,x:al         ; X: data memory
        move    b2,x:bh
        move    b1,x:bm
        move    b0,x:bl
        bra     *               ; trap program here
```

Figure 4-5

Conclusions ? What is the difference between integer and fractional multiplication ?

4.5 Modulo addressing

4.5.1 Read a sine wave

Create a one period 100 point sine wave in Y: data memory as explained in 2.9 within a modulo 100 buffer. Initialize pointer R0 with sine wave begin address, and M0 with 99 to get modulo 100 addressing. Observe the sine wave on DA1. What is it's frequency ?

4.5.2 Increment by N0

How would you generate a frequency N times the previous frequency ?
Apply with N=3.

4.5.3 Generate a sine and a cosine

Use the offset by N0 (N0 representing pi/2 in the sine wave) to generate both sine and cosine values at each sample; output these results respectively on DA1 and DA2. Switch the scope in X-Y mode and verify you get a circular Lissajou figure.

4.5.4 Signed frequency

Now take a sampling frequency of 100 Hz, observe the slow spot rotation on the scope. Change post-increment by post-decrement, the rotation should reverse.

```
;       Modulo addressing: Simple sine wave generation
size    set     100             ; Points of sine wave
incr    set     2*pi/size       ; angular increment
angle   set     0.              ; initial value

        org     y:              ; In Y: data memory field
        buffer  m,size          ; reserve a modulo buffer
sine
        dup     size            ; duplicate size times until endm
        dc      @sin(angle)     ; create constant in memory
angle   set     angle+incr      ; new angle
        endm
        endbuf                  ; end of modulo buffer


        org     p:              ; Current memory is program memory
begin   move    #sine,r0        ; R0 points to sine wave
        move    #size-1,m0      ; R0 performs modulo size addressing
loop    ada     1e5             ; Wait for Sample, Fs=100kHz
        move    y:(r0)+,a       ; Read one sine point into A, post increment
        move    a,x:da1         ; Write A to DAC1
        jmp     loop            ; Loop back
```

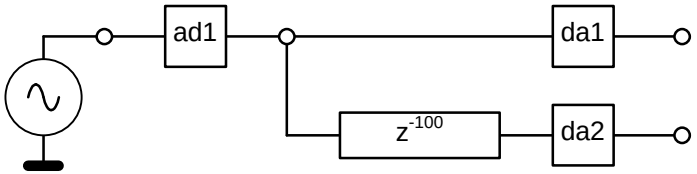
Figure 4-6

4.5.5 Some fun : draw flowers

Generate a sine and a cosine as in 4.5.2 . Use the ad1 signal to render the radius of the Lissajou figure variable (multiply sine and cosine signals by the ad1 value). Apply a periodic sine or triangle whose frequency is multiple of the sine-cosine frequency on ad1. You can add some DC at the function generator ...

4.6 Delay line

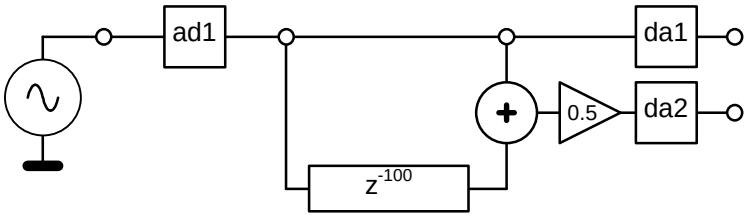
4.6.1 Delay line properties



Implement a 100 sample delay line as described in 3.1, and view both the sampled input signal and it's delayed version, as represented hereunder.
Give the expression of the phase between da2 and da1 as a function of frequency.

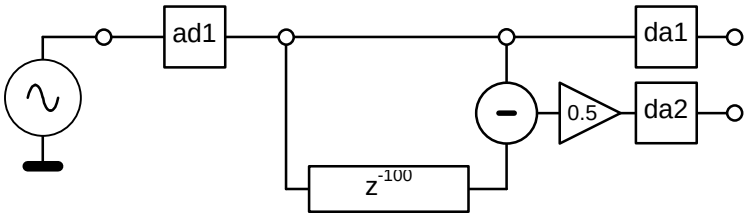
Figure 4-7 Delay line

4.6.2 Comb filters



Display on da2 the half sum of the signal and it's delayed version (fig4.2)
Sketch the frequency response |H(f)|
Where are the zeros of the system function located ?

Figure 4-8 Comb filter 1



Same questions as b, but with a half difference in place of a half sum:

Figure 4-9 Comb filter 2

4.7 FIR filter

4.7.1 Simple low-pass

Let's design a FIR low-pass filter cutting at $F_c = F_s/10$
The impulse response coefficients are given by:

$$h(n) = \frac{\sin(2\pi n F_c / F_s)}{\pi n}$$

Choose N=100
Reserve an N sample modulo buffer in X:
Initialize M0, and R0 pointing at the beginning of this buffer.
Create the N point impulse response h(n) in Y: memory, the same way you created a sine table (2.9). The impulse response must be symmetric, therefore, the N points P_0 .. P_{N-1} in memory should be:

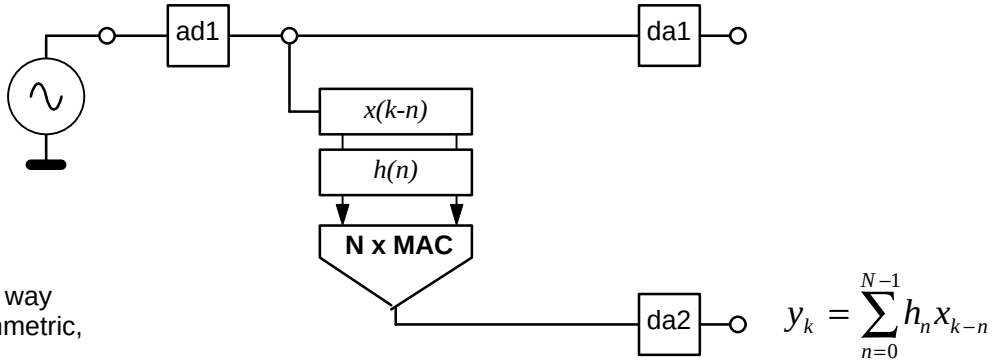


Figure 4-10

P_0 = h(-(N-1)/2) P_1 = h((-N+1)/2) ... P_{N-1} = h(N-1)/2
Note:
If N is odd, h(0) must be calculated separately in order to avoid division by 0.
If N is even, indices n in h(n) will be non integer.

Write a program similar to 3.2 to display both the original signal and the filtered signal. Try other values for N (

4.7.2 Ripple suppression

If you vary the frequency, you will observe some ripple both in the pass-band and in the stop-band.
This is due to the rectangular truncation of the h(n).
To attenuate this effect, weight the h(n) coefficients by a Hamming window : h'(n) = h(n)*w(n) with w(n) = 0.54 + 0.46Cos(2πn/N)

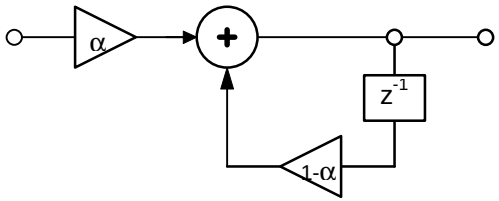
4.7.3 Adjusting gain to 1.0

Windowing slightly affects the gain of the filter in the pass-band. Show that the gain value in the pass-band is simply Σh'(n).

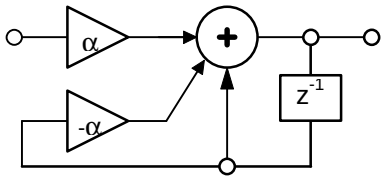
In order to adjust the gain to 1.0 in the pass-band, first calculate S = Σh'(n) (S is a compilation variable) ,
then create in memory the values h''(n) = h'(n) / S

4.8 IIR filters

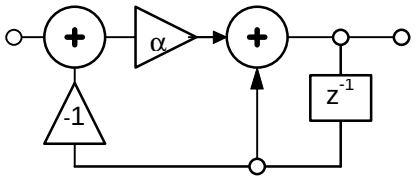
4.8.1 First order low-pass



(a)



(b)



(c)

Figure 4-11

The figures above are 3 implementations of the same filter. Give the frequency response of this filter as a function of α.
Implementation (b) uses a unique coefficient in memory, in place of 2 coefficients;
Implementation (c) allows 48 bit precision by using subtraction between accumulators followed by an arithmetic shift right (in this case, $\alpha = 2^{-n}$)

Write these 3 versions of the LP filter for Fc = 1Hz and Fs = 100kHz. Test their behavior with a sine generator connected on ad1.
Test their respective precision by applying a constant value (e.g. 0.5) at the input and reading the output in memory.
Conclusions ?
Note1 : considerable precision error arise when Fc << Fs, due to very small α coefficient, which is the case here.
Note2 : version (c) makes sense only if the state variable is 48 bit (thus reserve it in L:)

4.8.2 Second order band-pass

Implement the program given in section 3.4. Calculate the coefficients for a band-pass with Fc = 1kHz, Q=100, and Fs = 100kHz. (See the Motorola application note "Filters.pdf", section 2). Try other values for Q (1000, 10000). How is the filter output rise time related with Q ?

4.8.3 Single-pole complex filter, complex oscillator

Write a complex multiplication routine {X0=Re X1=Im} * {Y0=Re Y1=Im} → {A=Re B=Im}
If you implement figure 4-1 (a) with $\alpha = \rho \exp(j\theta)$ you get a single pole complex band-pass filter. Explain what the properties of such a filter are.
If the pole is exactly on the unit circle ($\rho = 1$), then, the filter becomes an oscillator with sine and cosine outputs.
Implement such an oscillator delivering a frequency of 1kHz, with Fs=100kHz.
Note: Initialize the output at {-1.0Re ; 0.0 Im }, choose ρ slightly greater than 1 to compensate truncation errors (e.g. 1.00001)

4.9 FFT

The purpose of this lab is to create a simplified power spectrum analyzer with oscilloscope display. The maximal number of points attainable with the resources of MuPsi is N=1024

Preliminary note: Since a DC input of 1.0 should give a FFT with a ray of amplitude 1.0 located at 0, and a sine wave of amplitude 1.0 should give 2 rays of amplitude 0.5 located at f0 and -f0, the 1/N factor has to be inserted in the *direct* Fourier Transform, not in the inverse FT:

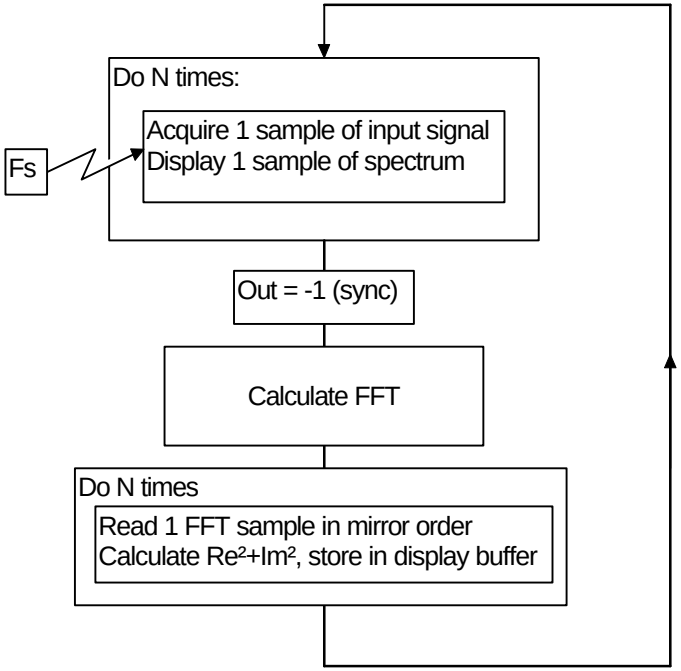


Figure 4-12

$$X_n = \frac{1}{N}.\sum_{k=0}^{N-1} x_k.e^{\frac{-j.2\pi.n.k}{N}}$$

The simplified spectrum analyzer algorithm comprises following steps:

1/ Acquire N samples of the signal in a complex FFT buffer (X:=Re; Y:=Im); multiply data by 1/N; if the signal is real, force the imaginary part to zero. One enhancement would be to weight the buffer data by a window such as Hamming etc. ...

2/ Calculate the FFT. Choose one version of a complex FFT macro from the Motorola FFT library (Help | Doc Motorola | Motorola Library | FFT). Read the associated help file. Include this macro at the beginning of your program.
To execute, you just have to invoke the macro with adequate parameters. Most macros use the same complex buffer for input and output.

3/ Unscramble the data. The FFT algorithm delivers a scrambled buffer which must be read using the mirror binary sequence. To do this, initialize R0=buffer, M0=0, N0=N/2, and use L:(R0)+N0 addressing mode to read complex data in the mirror sequence.

4/ Calculate the power spectrum ($Re^2 + Im^2$) of each complex sample, and store the results into the display buffer. One enhancement would be to average the spectrum in the same time by applying:
$Sn(k) = 0.9*Sn(k-1) + 0.1*(Re^2 + Im^2)$ where Sn(k) is the spectrum value at frequency n, refreshed for the k^{th} time.

5/ Read sequentially the display buffer. This operation must be timed, it is therefore advantageous to do it in the same time as the acquisition. The oscilloscope needs a sync signal for image stability. Since all data are positive or null, a -1.0 pulse will be perfect. To create it, you just have to set the DA output to -1.0 before the FFT calculation. Consequently, the sync pulse width represents the total spectrum calculation time.
Note:
The FFT modifies almost all pointers (R0-R7, M0-M7, N0-N7). The registers you use have to be initialized each time you start a loop.

4.10 A small project

In this lab, we will implement a filter bank spectrum analyzer which recognizes musical pitch. Each filter selects one musical note (halftone). Filter outputs are scanned during small periods (tic) to get the maximum amplitude value and the frequency associated with it. When the maximum amplitude observed is greater than Threshold1 then, the note value is written to the serial line, and the tic counter is reset to 0. When the maximum amplitude sinks under Threshold2, then the note duration expressed in tics is written to the serial line; hysteresis avoids multiple triggering around the detection level (Fig. 4.13).

4.10.1 Creating the main program

First, we have to implement the structure of the main program .
The diagram represented in Fig. 4-14 shows 2 states.

The simplest way to implement these two states is to make 2 loops inside the main program. Each loop samples the input signal, executes the filter bank and searches the maximum value and position. Switching from one loop to the other occurs when threshold levels are crossed.

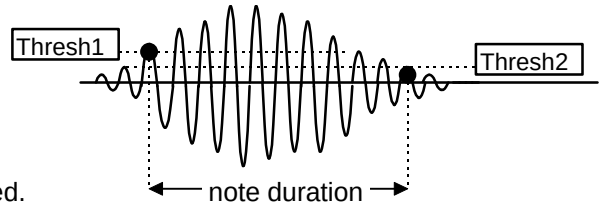


Figure 4-13 Hysteresis in note detection

Figure 4-15 shows the main program structure :

The filter bank and search maximum algorithm must be written as a subroutine.
The TIC software timer is also a subroutine. It produces an event every M samples.
It can be easily implemented by incrementing a reserved address register (R7 for example) in the modulo M mode (M7 = M-1).
The TIC event is true when R7 = 0 (copy R7 to A and test A).

Writing strings to the serial port can be done in 2 steps:
First create in memory the ASCII string to display terminated by a $FFFFFF
Initialize a reserved pointer (ex. R6) pointing to this string, and activate the SCI interrupt on TDRE mode. Write an Interrupt Service Routine which sends the pointed memory to the SCI transmit data low register, unless the MSB is 1, in which case the interrupt is deactivated. Create the vector pointing the ISR.

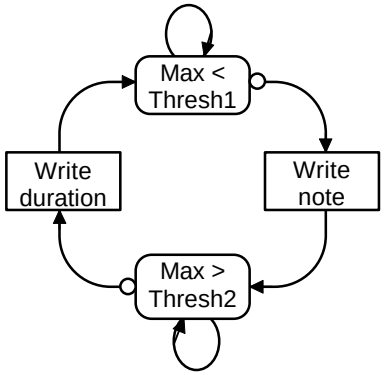


Figure 4-14 Program states

- **Write the main program, the Tic timer routine and the string output routine.**

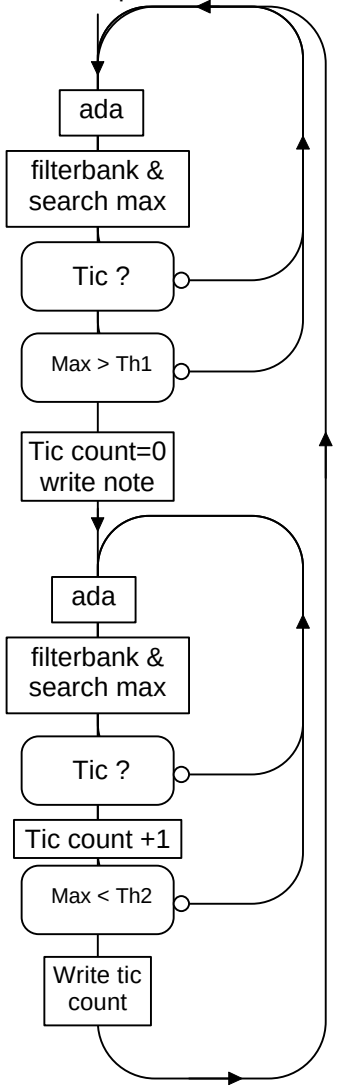


Figure 4-15 Main program

4.10.2 The filter bank

A musical tone frequency is defined by:

$$f_n = 2^{n/12} \times 440 \text{ Hz}$$

where *n* is the number of halftones from the reference LA or A to the current note.

- **Explain why we don't use a Fast Fourier Transform to get the pitch frequency.**

Filter frequencies and bandwidths are in a geometric progression of ratio $2^{1/12} = 1.0594631$.

- **Given we want adjacent filters to overlap at -3dB (Fig. 4.16), calculate the Q of the filters.**

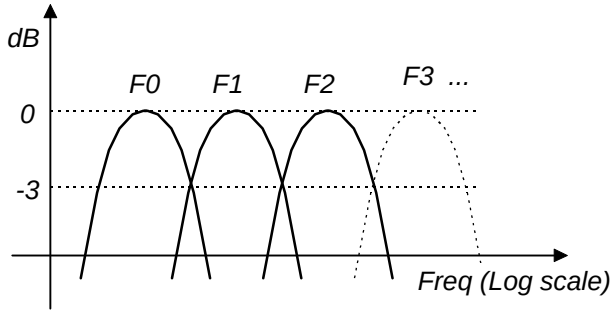


Figure 4-16 Filter bank

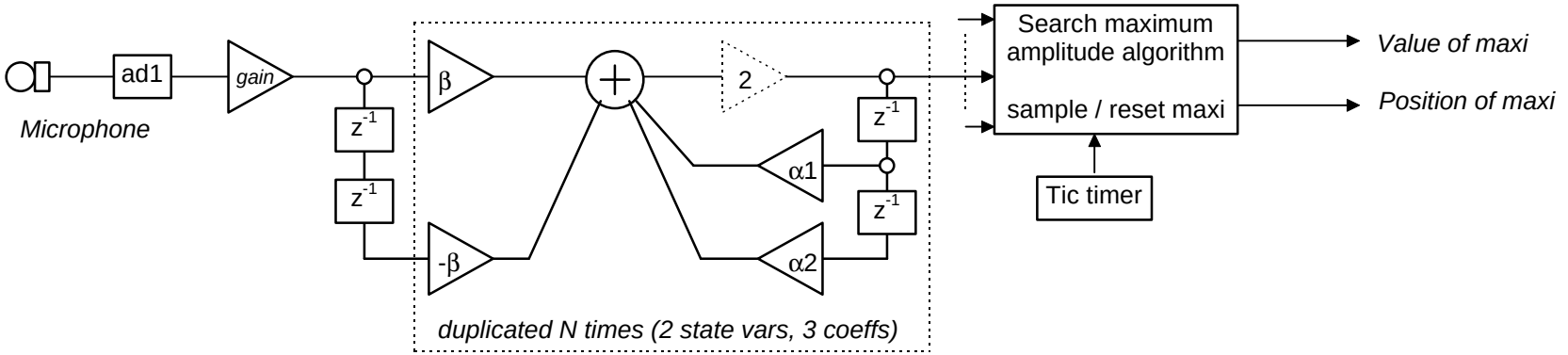


Figure 4-17 Band-pass filter bank implementation

Fig. 4-17 shows the direct form of a 2nd order band-pass filter implementation. Notice that the numerator requires only one coefficient β, and the two non recursive delay cells are common to all filters.

- **Consult the document Help | Doc Motorola | Application Notes | Filters.pdf to find how to calculate the a1, a2, and b coefficients.**
- **Create the data structure: 2N + 2 state variables in X:, 3N coefficients in Y:, with n = -30 ... +19 (N=50)**
- **Write the filter bank subroutine (use dynamic scaling to allow coefficients > 1)**

4.10.3 The maximum search algorithm

The N filter outputs are available in the state variable table (out0, delayed out0, out1, delayed out1, out2, ...).
To get the maximum, you scan the table at each sample. Current maximum value is reset to 0 at beginning of a tic period and sampled at end of this period.
Since tic period >> signal period >> sampling period, sampling the sine wave near the top is guaranteed during the tic period, which gives a good estimation of sine amplitude.
Reserve accumulator B for saving the current maximum value throughout a tic period.
Using the conditional transfer (tlt = transfer if less than) is the key of the maximum search algorithm:

```
        move    #states,r0              ; R0 -> begin of table
        move                x:(r0)+n0,x0  ; x0 = current filter output R0=current output address + 2
        do      #N,getmax1              ; Repeat for N filters
        cmp     x0,b        x:(r0)+n0,x0  ; (b - x0) ?  R0= current output address + 4
        tlt     x0,b        r0,r1       ; at end of do loop, b holds the maximum value, and R1, it's current address + 4
getmax1
```

The search algorithm represented above represents an autonomous routine.

- **However, you can optimize code efficiency by embedding it within the filter algorithm.**

4.10.4 Testing the application

Usually, when everything has been successfully compiled, there is still a strong probability that the application won't work at first !
So, you have to test independently each component of your software.
To test the filter bank, connect a sine generator at the ad1 input, an oscilloscope on da1 and da2 outputs, and write this program:

```
n       set     10                  ; filter to test
loop    ada     Fs                  ; wait for a sample
        move    x:ad1,x1            ; read sine generator input (x1 = filter bank input)
        move    x1,x:da1            ; output for test
        jsr     filterbank          ; execute N filters
        move    x:states+2*n,a      ; read one filter output
        move    a,x:da2             ; display output on scope
        jmp     loop
```

- **If some behavior seems incomprehensible don't hesitate to start the DSP software simulator to execute step by step your program.**

5. ANSWERS TO THE QUIZ

1.1 Executing a whole algorithm within a single sample time
1.2 Texas Instruments, Motorola, Analog Devices, Lucent
1.3 Harward Architecture, pipeline, parallelism, DMA, special instructions
1.4 Fixed point: limited dynamic requiring calibration to avoid saturation. Floating point: handles numbers from -∞ to +∞ , but more expensive and slower
1.5 Overflow: crossing maximum value limit produces minimum value Saturation: crossing maximum value limit produces maximum value
1.6 The MAC (multiply - accumulate) instruction: y = y + x1 * x2

2.1 DSP56xxx processors are designed for audio processing. Since digital audio is quantized with 16 bits, 8 additional guard bits are necessary to minimize truncation errors. Thus the basic data width is 24 bit. When multiplying two 24 bits words, the result is 48 bit. If a product is accumulated 256 times, then the untruncated result has 56 bit..
2.2 b0: 2-47 b24: 2-23 b48: 1 b54: 128 b55: -256
2.3 a) x:0=$123456 b) x:1 = $123456 y:1 = $789ABC c) x:2 = $800000 d) x:3=$DCBA98
2.4 a) 0, 1, 2, 3, 4, 5 ... b) 5 5 5 ... c) 0, 5, 10, 15... d) 8, 9, 10, 11, 12, 8, 9, ... e) 8, 10, 12, 9, 11, 8, 10, ...
2.5 Code optimization:

```
move    x:(r0)+,x0
move    y:(r4)-,y0
rep     #99
mac     x0,y0,a x:(r0)+,x0      y: (r4)-,y0
mac     x0,y0,a
```

New execution time: 105 cycles

3.1 Only 2 cycles
3.2 FIR Advantages: always stable, possible linear phase, low sensitivity to quantization errors, direct programming of the impulse response.
Drawbacks: High order necessary to get performance, thus execution time long.
3.3 Write the transfer function as a product of 2nd order sections (plus eventually one 1st order section). Dispatch the global gain at each section while caring never to get saturation in a section. In each 2nd order section, use dynamic scaling to handle coefficients > to 1.0.
3.4 Following DSP56k features accelerate FFT calculation:
Simultaneous P:, X:, Y: memory fields parallel accesses
Nested DO loops
Reverse Carry addressing mode (Mi = 0)
The "SUBL" instruction

6. REFERENCES

Craig Marven & Gillian Ewers
A simple approach to Digital Signal Processing
ISBN 0-904047-00-8 Texas Instruments

Mark McQuilken & James P. Leblanc
Digital Signal Processing and the Microcontroller
Motorola

Motorola Reference books (.pdf files)

DSP56300FM Family reference manual
DSP56309UM User Manual
DSPASM Assembler reference manual
DSPLNK Linker and utilities manual

Motorola Application notes (.pdf files)

Fractional and Integer Arithmetic using the DSP56000 family ...
Implementing IIR/FIR filters with Motorola's DSP5600x Digital Signal Processors
FFT Implementation on Motorola's Digital Signal Processors
Application optimization for the DSP56300/DSP56600 Digital Signal Processors