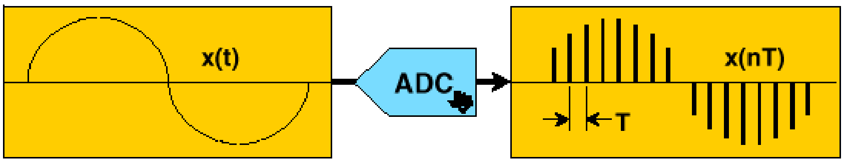
Part 5
Programming and Architecture for Digital Signal Processor

- Real-time processing on a DSP processor
- DSP selection
- TMS320C55x programming
- TMS320C55x architecture

Real-Time Processing on a DSP Processor

Sampling Frequency Consideration



$$\text{Sampling Period (T)} = \frac{1}{\text{Sampling Frequency (} f_s \text{)}}$$

Example:

Fs = 8 kHz = 8,000 cycles per second

T = 1 / 8 kHz = 125 µ s or 0.000125 seconds

Sampling Period (T) / Instruction Cycle Time = # of Instructions per Sample

125 µ s / 10 ns = 12,500 Instructions per Sample

125 µ s / 5 ns = 25,000 Instructions per Sample

125 µ s / 3 ns = 41,600 Instructions per Sample

Sample-by-sample processing

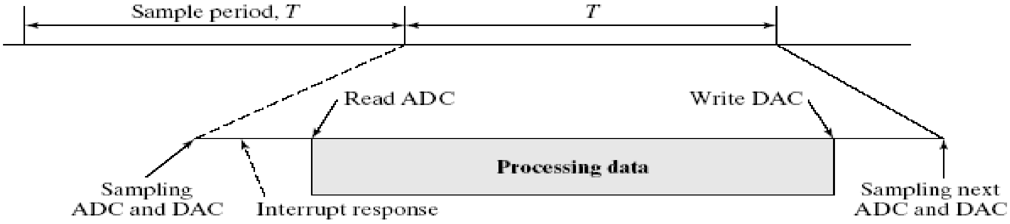


Figure 3.16 Detailed interrupt timing at each sampling interval for an I/O operation

Here we show the detailed sampling interval for sample-by-sample processing mode.

Real-Time processing refers to the digital processing of data within the sampling interval.

DSP must finish processing within 1 sample interval, T_S

Require 1 word FIFO, memory or register to hold incoming data

Block processing

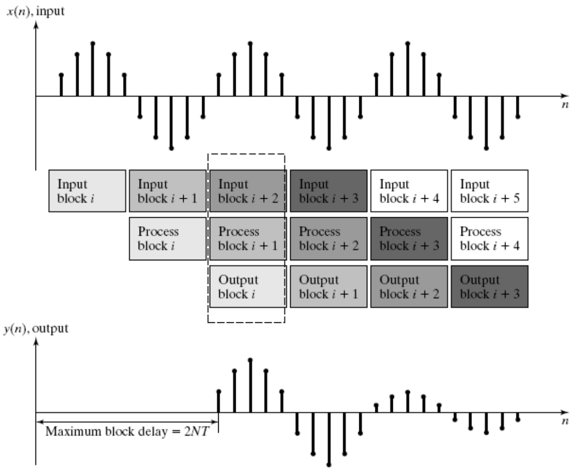
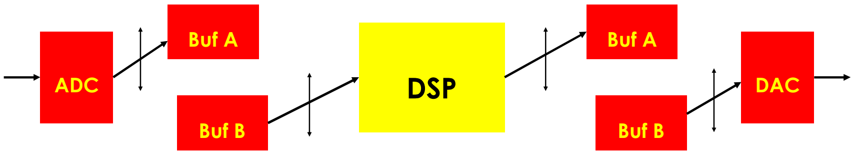


Figure 3.17 Block processing of an input signal in a block of five samples

- Collect N samples at a time
- Processing must finish within N sample period, NT
- Normally used in FFT, Compression
- Requires double or triple buffering to perform acquisition and processing
- $T_p < N^*T_s$
- A maximum block delay of $2NT_s$

Double buffering



- Advantage:
  - Can process more data for some given T_s due to the reduced setup time
  - Or use a higher F_s compared to the sampling approach
- Disadvantage:
  - More memory required
  - More effort in programming
  - Processing latency

DSP Performance Ratings

DSP processors may be rated at:

- Low (instruction) levels
- Mid (algorithm kernel) levels
- High (DSP application) levels

MIPS Rating

- MIPS = number of million MAC (multiply-accumulate) instructions per second
- TMS320C55x example: 120 MHz clock, 1 cycle/MAC (when fully pipelined), 2 parallel multiply-accumulator = 240 MIPS
- Other alternatives:
  BIPS = billion instructions per second
  MOPS/BOPS = … operations …
  MFLOPS = … floating point operations …
- May not be suitable for signal processing algorithms where multiplication is not the limiting factor

Algorithm Kernels for Ratings

- Performance of an algorithm kernel such as FIR filtering, IIR filtering, FFT, etc. is measured.
- Relatively simple to optimize.
- Good benchmark.

Library functions of individual DSP processors, hand-optimized to run with the least CPU cycles, show the performance of the processor.

TMS320C55x example:

Function	Description	Benchmarks
fir2	Finite Impulse Response Filter	Core: nx * (3+nh/2); Overhead: 25
dlms	Adaptive Delayed LMS Filter	Core: nx * (7+2*(nh-1)) Overhead: 26
convol	Convolution	Core: nr * (1+nh) Overhead: 44
cfft	Complex FFT	(FFT Size,Cycle) = (8,208) (16,358) ... (512,11848) (1024,25954)
rand16, rand16init	Random Number Generation (16-bit) and Its Initialization	rand16: Core: 13 + nr*2, Overhead:10; rand16init: 6

nx/nr = input length, nh = filter length

DSP Applications for Ratings

- Performance of a complete DSP application, such as V.90 modem, mp3 decoder, etc. is measured.
- Complex and costly to implement and optimize.
- Even then, the performance may depend on the programmer’s ability.

DSP Selection

Hardware platforms

- General purpose microprocessor (µP)
- General prupose DSPs
- Digital building blocks (DBB) such as multiplier, adder, program controller
- Field programmable gate array (FPGA) and application-specific integrated circuits (ASIC)

Hardware platforms

	µP	DSP	DBB	ASIC
Chip count	1	1	>1	1
Flexibility	Programmable	Programmable	Limited	None
Design time	Short	Short	Medium	Long
Power consumption	Medium	Low-medium	Medium-high	Low
Processing speed	Low-medium	Medium-high	High	High
Reliability	High	High	Low-medium	High
Development cost	Low	Low	Medium	High
Production cost	Low-medium	Low-medium	High	Low

Selection of DSP chips

The objective is to select the device that

- meets the time
- is the most cost-effective

Factors to be considered are

- computational power, resolution
- cost
- I/O
- development environment and tools

Selection is affected by the volume:

- High volume = cost
- Low volume = tradeoff between development time and cost

Fixed point versus floating point

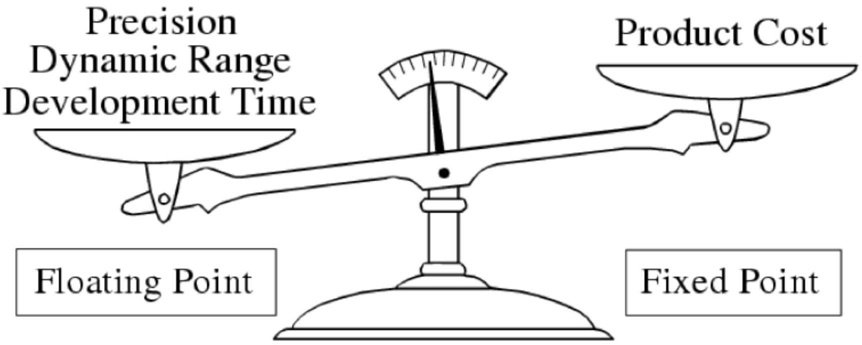
16 bit fixed point processors

- Texas Instruments TMS320C1X, TMS320C2X
- Motorola 56000
- Analog Devices ADSP2100
- AT&T DSP16

32 bit floating point processors

- Texas Instruments TMS320C3X, TMS320C4X
- Motorola 96000
- AT&T DSP32C

Fixed point versus floating point



Fixed point versus floating point

Fixed point processors	Floating point processors
16 or 24 bit	32 bit
Limited dynamic range	Large dynamic range
Overflow and quantization errors must be resolved	Easier to program since no scaling is required
Poorer C compiler efficiency; usually programmed in assembly	Better C compiler efficiency; can be developed in C
Long product development time	Quick time to market
Faster clock rate	Slower clock rate
Less silicon area since functional units are simpler	More silicon area since functional units are complex
Cheaper	More expensive
Lower power consumption	Higher power consumption

Fixed point versus floating point: Applications

Fixed point processors	Floating point processors
• Disk drive • Motor control • Consumer audio applications such as MP3 player • Multimedia gaming • Digital camera • Speech coding/decoding • Channel coding • Communication devices such as modem and cellular phone	• Image processing in radar, sonar, and seismic applications • High-end audio applications such as ambient acoustics simulator, professional audio encoding/decoding, and audio mixing • Sound synthesis in professional audio • Video coding/decoding • Prototyping

TMS family

- **C1X, C2X**: fixed point, 16 bit, used in toys, hard disk drives, modems, active car suspensions

- **C3X**: floating point, 32 bit, used in hi-fi systems, voice mail systems, 3D graphic processing

- **C4X**: floating point, 32 bit, designed for parallel processing, optimized on-chip communication channel enables several devices to be connected, used in virtual reality, recognition, parallel processing systems

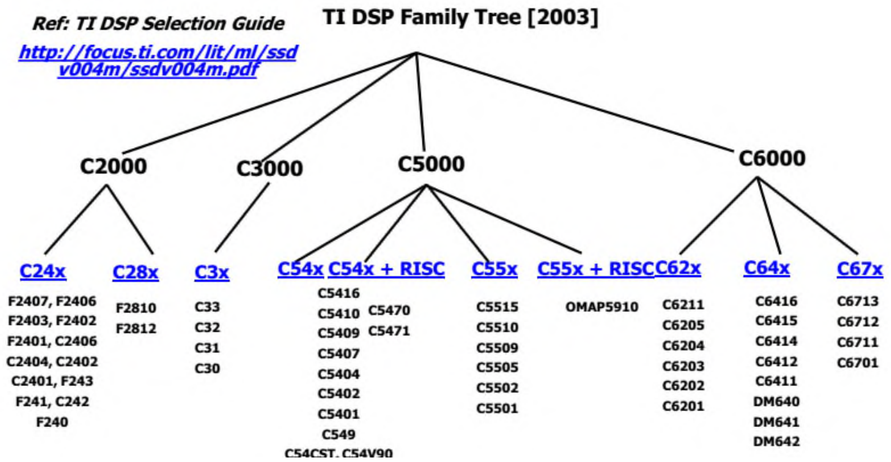
TMS family

- **C5X**: fixed point, low power (0.25mW/MIP), used in personal and portable electronics such as cell phones, digital music players, digital cameras

- **C6X**: both fixed and floating point, high performance DSP with speeds up to 1GHz, used in wired and wireless broadband networks, imaging applications, professional audio

- **C8X**: multimedia processor, multicore with parallel processing on a single chip, includes a controlling RISC processor, used in high performance telephony, 3D computer graphics, virtual reality

TMS family



TMS320C55x Programming

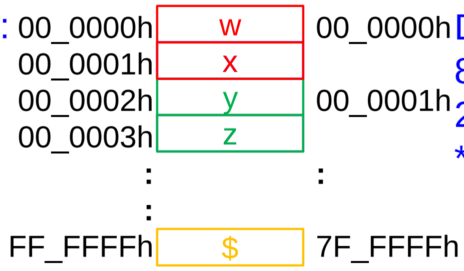
TMS320C55x

- 16-bit Fixed-Point Processor, Up to 120 MHz Clock
- Two Multipliers (up to 240M MAC operations)
- Two arithmetic logic units (ALU)
- Internal Data/Address Busses: 3 reads, 2 writes
- Separate Program Data/Address Bus
- 64KB Dual-Access RAM, 256KB Single-Access RAM
- Multiple Peripherals supports

C55x Unified Memory Map

Program and data share the same memory

Program memory:
16M x 8 bit
24-bit address
*(00_0001h) = x

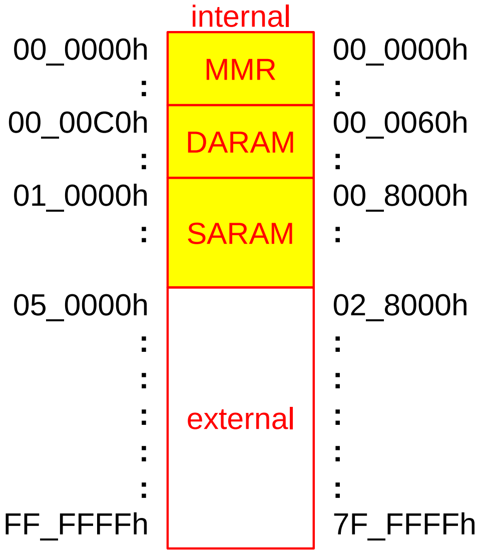


Data memory:
8M x 16 bit
23-bit address
*(00_0001h)=yz

segmented into 128 pages (7 msb bits)

page = 64K words (16 lsb bits)

Internal (on-chip) and external memory:



96 memory mapped registers

about 32K on-chip dual-access program/data memory

128K on-chip single-access program/data memory

external memory

C55x Registers

- Memory mapped registers (MMR): 00_0000h to 00_005Fh
- Accumulators: AC0 to AC3
- Transition registers: for logical branching
- Temporary: T0 to T3

(Accumulator, auxiliary register, and temporary register will be described latter)

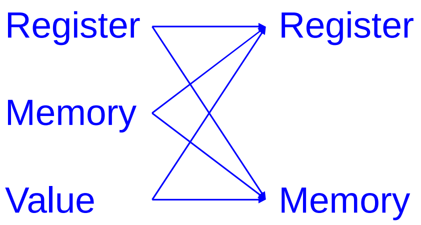
- Registers to address data memory and I/O space: normally 16 bit (address within page), with X (extended) 23 bit (includes page address)
  - Auxiliary: AR0 to AR7
  - Coefficient data pointer (CDP), XCDP, used to address coefficient memory
  - Circular buffer registers
  - Data page (DP), XDP
  - Peripheral data
  - Stack pointer registers

- Registers for program flow such as program counter, return address during subroutine call
- Interrupt registers
- Registers to control single-repeat/block-repeat loops
- Status registers, having control and flag bits such as overflow, carry, etc.

Assembly Instructions on C55x

- To appreciate the processor architecture and how it achieves parallel operations, assembly instructions are helpful
- Mnemonic form (algebraic form is also possible)
- 2 example instructions: MOV, MAC (only some cases)
- MOV is used in any general purpose processor
- MAC is unique to DSP architecture

MOV src, dst



Registers

Accumulators: AC0 to AC3
Located in D unit, used for ALU/MAC/shifter output,
each portion can be accessed individually or together

guard=bits 39-32	high = bits 31-16	low = bits 15-0

Examples:

AC0	40 bits
HI(AC0)	16 bits (31-16)
LO(AC0)	16 bits (15-0)

MOV AC0, AC1   move 40 bits from AC0 to AC1

Before		After	
AC0	01_E590_0030h	AC0	01_E590_0030h
AC1	00_0000_0000h	AC1	01_E590_0030h

Auxiliary registers: AR0 to AR7
Located in A unit, used to address data memory,
ARxH = 7-bit memory page, ARx = 16 lsb bits of address

ARxH = bits 22-16	ARx = bits 15-0

XARx = bits 22-0

Examples:

AR0	16 bits (15-0)
XAR0	23 bits

MOV AR0, AR1	move 16 bits from AR0 to AR1
MOV XAR0, XAR1	move 23 bits from XAR0 to XAR1

MOV HI(AC0), AR0 move 16 high bits of AC0

Before		After	
AC0	01_E590_0030h	AC0	01_E590_0030h
XAR0	00_FF24h	XAR0	00_E590h

MOV AR0, HI(AC0) move only to 16 high bits of AC0

Before		After	
AC0	01_E590_0030h	AC0	01_0024_0030h
XAR0	02_0024h	XAR0	02_0024h

MOV can be x bits → y bits, where
x = y
x > y move y lsb bits

MOV AC0, AR0 move 16 low bits of AC0

Before		After	
AC0	01_E590_0030h	AC0	01_E590_0030h
XAR0	02_FF24h	XAR0	02_0030h

MOV AC0, XAR0 move bits 22-0 of AC0

Before		After	
AC0	01_E590_0030h	AC0	01_E590_0030h
XAR0	02_FF24h	XAR0	10_0030h

x < y x bits sign extended to y bits,
except when x = 23 bits, x bits zero filled to y bits

MOV AR0, AC0 16 bits of AR0 sign extended to 40 bits

Before		After	
AC0	02_E590_0030h	AC0	00_0000_0024h
XAR0	01_0024h	XAR0	01_0024h

MOV XAR0, AC0 23 bits of XAR0 zero filled to 40 bits

Before		After	
AC0	02_E590_0030h	AC0	00_0001_0024h
XAR0	01_0024h	XAR0	01_0024h

Temporary registers: T0 to T3
Located in A unit, used for multiplicand, shift count, pointer value, transition matrix, etc.

Tx = bits 15-0

Example:
T0 16 bits

Tx works in the exact same way as ARx shown before

Pair of registers: AR0, AR2, AR4, AR6, T0, T2
Refers to 2 registers such as AR0 and AR1

Example:
Pair(AR0) 2 x 16 bits

Value

Denoted by “#”
Can be in decimal or hex (denoted by “h”)

k4/**-k4**: 4-bit unsigned constant (with “-” if negative)
Example:
(k4) #11 [in mnemonic, 11 means $(11)_{10}$]
(k4) #Bh = 11 [in mnemonic, Bh means $(B)_{hex}$]
(-k4) #-11 = –11

MOV #Bh, AC0 sign extended to 40 bits (even though Bh = $(1011)_2$, since it is known to be unsigned/positive, sign extended by 0’s)

Before		After	
AC0	02_E590_0030h	AC0	00_0000_000Bh

MOV #-1, AR0 sign extended (by 1’s) to 16 bits

Before		After	
AR0	0024h	AR0	FFFFh

K8: 8-bit signed constant
Example:
#20
#14h = 20

K16: 16-bit signed constant
Example:
#248
#FC18h = –1000

MOV #248, AC0 sign extended to 40 bits

Before		After	
AC0	02_E590_0030h	AC0	00_0000_00F8h

Value → register uses k4, -k4, K16
Value → memory uses K8, K16

Memory Addressing Modes in C55x

- Absolute (constant) addressing
- Direct (offset) addressing
- Indirect (pointer) addressing

k23 absolute addressing:

Denoted by “*”

*(23-bit unsigned constant) = 16-bit data at this address

constant = bits 22-0

Example:

*(#010501h) 16-bit data at 01_0501

k16 absolute addressing:

Denoted by “*”

*abs16(16-bit unsigned constant) = use 7-bit DPH (high part of XDP) + 16-bit constant, to obtain a 23-bit address, then 16-bit data at this address

DPH = bits 22-16	constant = bits 15-0

Example:

*abs16(#0501h) 16-bit data at (DPH)_0501

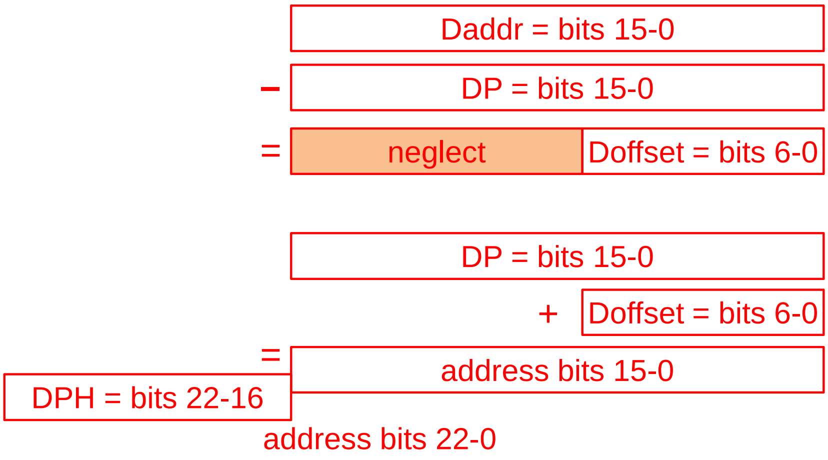
MOV #256, *abs16(#0501h) move value 256

Before		After	
DPH	00h	DPH	00h
00_0501	FC00h	00_0501	0100h

MOV AC0, *abs16(#0E10h) move 16 bits

Before		After	
AC0	01_4500_0030h	AC0	01_4500_0030h
DPH	00h	DPH	00h
00_0E10	0000h	00_0E10	0030h

Direct addressing:
Denoted by “@”
(assume compiler mode bit CPL=1 in status register)
@Daddr = Use 7-bit DPH + 16-bit (DP + Doffset), then 16-bit data at this address. Doffset = 7 lsb bits of (Daddr – DP).



Example: (assume DPH = 03, DP = FFF0)

@65524	Doffset = 65524 – FFF0h = 4h address = 03_FFF4, 16-bit data at this address
@FFF4h	same as above

MOV @FFF4h, AR0 move data at 03_FFF4 to AR0

Before		After	
XDP	03_FFF0h	XDP	03_FFF0h
XAR0	01_0000h	XAR0	01_3400h
03_FFF4	3400h	03_FFF4	3400h

If DP value is not known, use:

.dp x	directive to provide compile-time base address for assembler to calculate Doffset

MOV @(x+4), AR0

Auxiliary register (AR) indirect addressing:
Denoted by “*”
*ARx = 16-bit data at XARx

XARx = bits 22-0

Example:

*AR0	16-bit data at XAR0
low_byte(*AR0)	8-bit (bit 7-0) data at XAR0
high_byte(*AR0)	8-bit (bit 15-8) data at XAR0
*AR0, *AR1	2x16-bit data, bit 15-0 at XAR0, bit 31-16 at XAR1
dbl(*AR0)	32-bit data. If XAR0 = even, then bit 31-16 at XAR0 and bit 15-0 at XAR0+1. If XAR0 = odd, then bit 31-16 at XAR0 and bit 15-0 at XAR0-1.

MOV *AR0, *AR1 move data at XAR0 to location addressed by XAR1

Before		After	
XAR0	01_0300h	XAR0	01_0300h
XAR1	01_0400h	XAR1	01_0400h
01_0300	3400h	01_0300	3400h
01_0400	0000h	01_0400	3400h

MOV *AR0, AR1 move data at XAR0 to AR1

Before		After	
XAR0	01_0300h	XAR0	01_0300h
XAR1	01_0400h	XAR1	01_3400h
01_0300	3400h	01_0300	3400h

MOV *AR0, AC0 move 16 bit data at XAR0, after sign extension to 40 bit, to AC0

Before		After	
XAR0	01_0300h	XAR0	01_0300h
AC0	01_1111_0000h	AC0	00_0000_3400h
01_0300	3400h	01_0300	3400h

MOV AR0, *AR1 move 16 bits from AR0 to location addressed by XAR1

Before		After	
XAR0	01_0300h	XAR0	01_0300h
XAR1	01_0400h	XAR1	01_0400h
01_0400	3400h	01_0400	0300h

MOV AC0, high_byte(*AR0) move bits 7-0 of AC0 to bits 15-8 of location at XAR0

Before		After	
XAR0	01_0200h	XAR0	01_0200h
AC0	20_FC00_6788h	AC0	20_FC00_6788h
01_0200	6903h	01_0200	8803h

MOV AR0, low_byte(*AR1) move bits 7-0 of AR0 to bits 7-0 of location at XAR1

Before		After	
XAR0	01_6788h	XAR0	01_6788h
XAR1	02_0300h	XAR1	02_0300h
02_0300	6903h	02_0300	6988h

MOV *AR0, *AR1, AC0

Move 16-bit data at XAR0 to bits 15-0 of AC0. Move 16-bit data at XAR1, sign extended to 24-bit, to bits 39-16 of AC0.

Before		After	
XAR0	01_0300h	XAR0	01_0300h
XAR1	01_0400h	XAR1	01_0400h
AC0	01_1111_0000h	AC0	00_0300_FF00h
01_0300	FF00h	01_0300	FF00h
01_0400	0300h	01_0400	0300h

MOV AC0, *AR0, *AR1

Move bits 15-0 of AC0 to location at XAR0. Move bits 31-16 of AC0 to location at XAR1.

Before		After	
AC0	01_4500_0030h	AC0	01_4500_0030h
XAR0	00_0200h	XAR0	00_0200h
XAR1	00_0201h	XAR1	00_0201h
00_0200	3400h	00_0200	0030h
00_0201	0FD3h	00_0201	4500h

MOV dbl(*AR0), dbl(*AR1)

move 2 consecutive data

Before		After	
XAR0	01_0300h	XAR0	01_0300h
XAR1	01_0400h	XAR1	01_0400h
01_0300	3400h	01_0300	3400h
01_0301	0FD3h	01_0301	0FD3h
01_0400	0000h	01_0400	3400h
01_0401	0000h	01_0401	0FD3h

MOV dbl(*AR0), pair(T0)

move bits 31-16 to T0 and bits 15-0 to T1

MOV dbl(*AR0), AC0

move 32-bit data

Before		After	
XAR0	01_0300h	XAR0	01_0300h
AC0	01_1111_0000h	AC0	00_3400_0FD3h
01_0300	3400h	01_0300	3400h
01_0301	0FD3h	01_0301	0FD3h

MOV dbl(*AR0), XAR1 move bits 22-0 to XAR1

Before		After	
XAR0	01_0200h	XAR0	01_0200h
XAR1	00_0000h	XAR1	12_0FD3h
01_0200	3492h	01_0200	3492h
01_0201	0FD3h	01_0201	0FD3h

MOV AC0, dbl(*AR0) move 32 bits of AC0

MOV XAR1, dbl(*AR0) move 23 bits of XAR1 zero filled to 32 bits

Before		After	
XAR0	01_0200h	XAR0	01_0200h
XAR1	7F_3492h	XAR1	7F_3492h
01_0200	3765h	01_0200	007Fh
01_0201	0FD3h	01_0201	3492h

MOV pair(T0), dbl(*AR0) move T0 to location at XAR0
move T1 to location at XAR0+1

Operands

Operands for AR indirect addressing:

*ARx	no modification
*ARx±	increase/decrease ARx after addressing
*±ARx	increase/decrease ARx before addressing

MOV *AR0+, AR1 move data at XAR0 to AR1, AR0=AR0+1

Before		After	
XAR0	01_0300h	XAR0	01_0301h
XAR1	01_0400h	XAR1	01_3400h
01_0300	3400h	01_0300	3400h

MOV *+AR0, AR1 AR0=AR0+1, move data at XAR0 to AR1

Before		After	
XAR0	01_0300h	XAR0	01_0301h
XAR1	01_0400h	XAR1	01_0701h
01_0301	0701h	01_0301	0701h

*(ARx±AR0) add/subtract AR0 to ARx after addressing
*(ARx±Ty) add/subtract Ty to ARx after addressing

MOV *(AR0+T0), AR1 move data at XAR0 to AR1, AR0=AR0+T0

Before		After	
XAR0	01_0308h	XAR0	01_030Ch
XAR1	01_0400h	XAR1	01_3228h
T0	0004h	T0	0004h
01_0308	3228h	01_0308	3228h

*ARx(offset) no modification to ARx, but offset is used while addressing. offset = AR0, Ty, #K16

MOV *AR0(T0), AR1 move data at XAR0+T0 to AR1

Before		After	
XAR0	01_0304h	XAR0	01_0304h
XAR1	01_0400h	XAR1	01_3228h
T0	0004h	T0	0004h
01_0308	3228h	01_0308	3228h

*+ARx(#K16) add #K16 to ARx before addressing

MOV *+AR0(#4), AR1 AR0=AR0+4, move data at XAR0 to AR1

Before		After	
XAR0	01_0304h	XAR0	01_0308h
XAR1	01_0400h	XAR1	01_3228h
01_0308	3228h	01_0308	3228h

Example:

Given the “before” state

XDP 02-0106h	T0 0002h	02_0105 0021h
AC0 00_1111_2222h	T1 0000h	02_0106 0030h
XAR1 02_0106h	02_0206 0060h	02_0107 0040h
		02_0108 0050h

find the state after the instruction(s).

MOV *(#020106h), AC0	AC0 00_0000_0030h
.dp x MOV @(x+1h), AC0	AC0 00_0000_0040h
.dp x MOV #4, @(x+128)	02_0106 0004h

MOV T0, *AR1+	XAR1 02_0107h	02_0106 0002h
MOV *(AR1+T0), T1	XAR1 02_0108h	T1 0030h
MOV *AR1(T0), AC0	AC0 00_0000_0050h	
MOV *AR1(#0100h), T1	T1 0060h	
MOV *+AR1(#-1), AC0	XAR1 02_0105h	AC0 00_0000_0021h

MAC m1, m2, sum

sum = sum + (m1 * m2)

- sum = ACx
- m1, m2 of shorter length are sign extended to 17 bits
- 32-bit multiplication result is sign extended to 40 bits and added to ACx. If an overflow is detected, accumulator overflow status bit is set.
- Rounding and/or shifting may be performed if needed.

There are 4 types of MAC based on the choice of m1, m2

1) **MAC** with registers: m1 = ACx, m2 = Ty
ACx: only bits 32-16 are multiplied
Ty: sign extended to 17 bits

Example:
MAC AC0, T1, AC2 AC2 = AC2 + ($AC0_{32\text{-}16}$ * T1)

2) **MACK** with value: m1 = Tx, m2 = K8/K16
K8/K16: 8/16-bit signed value, sign extended to 17 bits

Examples:
MACK T0, #FFh, AC0 AC0 = AC0 + (T0 * FFh)
MACK T0, #FFFFh, AC0 AC0 = AC0 + (T0 * FFFFh)

3) **MACM** with memory: m1 = memory, m2 = ACx/Tx/memory
memory: sign extended to 17 bits

Examples:

MACM *AR0, AC0, AC1 AC1 = AC1 + (data * $AC0_{32\text{-}16}$)

MACM *AR0, *AR1, AC0 AC0 = AC0 + (data1 * data2)

Before		After	
AC0	00_2300_EC00h	AC0	00_5A20_EC00h
XAR0	01_0302h	XAR0	01_0302h
XAR1	01_0202h	XAR1	01_0202h
01_0202	7000h	01_0202	7000h
01_0302	7E00h	01_0302	7E00h

working:
7Eh * 7h = 126 * 7 = 882 = 372h
So 7E00h * 7000h = 3720_0000h
2300_EC00h + 3720_0000h = 5A20_EC00h

memory can be on-chip coefficient memory, addressed by the coefficient data pointer CDP

MACM *AR0, *CDP, AC0 AC0 = AC0 + (data * coeff)

Before		After	
AC0	00_EC00_0000h	AC0	00_EBFF_8000h
XAR0	02_0302h	XAR0	02_0302h
CDP	01_0202h	CDP	01_0202h
01_0202	0040h	01_0202	0040h
02_0302	FE00h	02_0302	FE00h

working:
FEh * 4h = –2 * 4 = –8 = F8h
So FE00h * 0040h = FFFF_8000h
EC00_0000h + FFFF_8000h = EBFF_8000h

4) **MACMK** with memory and value: m1 = memory, m2 = K8
Example:
MACMK *AR0, #FFh, AC0 AC0 = AC0 + (data * FFh)

Parallel Instructions in C55x

- Built-in parallelism – two instructions are defined to operate parallely
- User-defined parallelism – if certain constraints are satisfied, then the programmer/compiler may execute two instructions parallely

Built-in parallelism example: **MAC::MAC**
MAC memory, coeff_memory, accumulator :: MAC memory, coeff_memory, accumulator

```
MAC *AR0, *CDP, AC0 :: MAC *AR1, *CDP, AC1
                                      AC0 = AC0 + (data1 * coeff)
                                      AC1 = AC1 + (data2 * coeff)
```

User-defined parallelism example: **MOV || MOV**
MOV register, register || MOV register, register

```
MOV AC0, AC1 || MOV AC2, AC3      AC1 = AC0, AC3 = AC2
```

The following operations may be executed parallely:

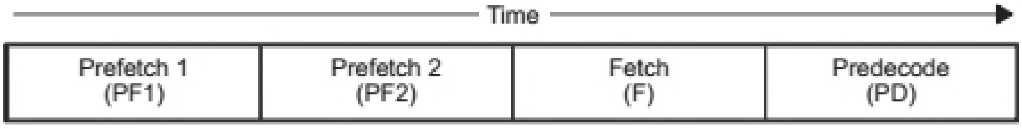
- Add/subtract, add/subtract
- Add/subtract, read/write data memory
- Multiply, multiply
- Multiply, read/write data memory
- MAC, MAC
- MAC, multiply
- MAC, read/write data memory

Pipeline in C55x

Fetch phases

Instructions are fetched from memory in 4 phases:

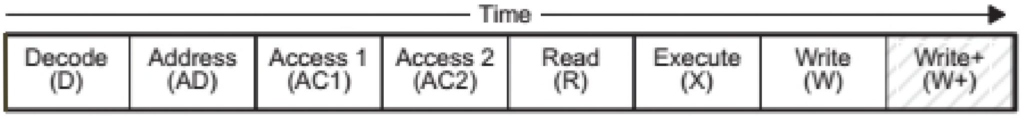
1. Prefetch1 (program address presented to memory)
2. Prefetch2 (wait for memory to respond)
3. Fetch (fetch instruction bits from memory to instruction buffer queue)
4. Predecode (identify entry and exit phases, parallelism)



Execution phases

Instructions are executed in typically 7 phases:

1. Decode (decode instruction, dispatch to other units)
2. Address (read/modify with A-unit address ALU)
3. Access1 (place address on bus)
4. Access2 (read access done to the memory)
5. Read (read data delivered to bus)
6. Execute (perform modify/read registers)
7. Write (write data to memory)
8. Write+ (memory confirmation)



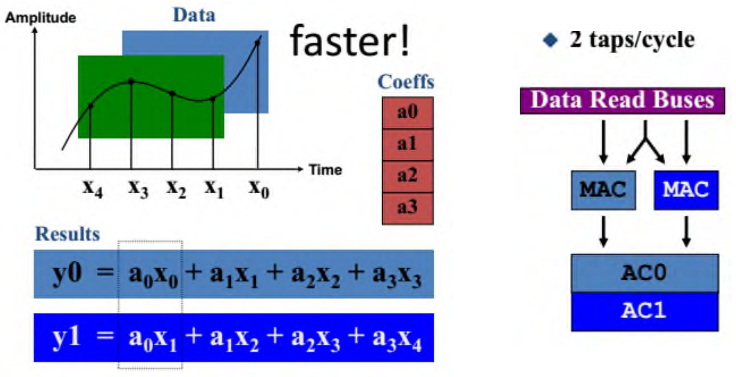
Note: ▨ Only for memory write operations.

To improve cycle utilization, different phases of different instructions can be simultaneously executed.

D	AD	AC1	AC2	R	X	W	Cycle
I1							1
I2	I1						2
I3	I2	I1					3
I4	I3	I2	I1				4
I5	I4	I3	I2	I1			5
I6	I5	I4	I3	I2	I1		6
I7	I6	I5	I4	I3	I2	I1	7

Optimum, or fully interlocked, pipeline is achieved when 7 instructions **I1** to **I7** are simultaneously executed, such as in cycle 7

FIR Filtering in C55x

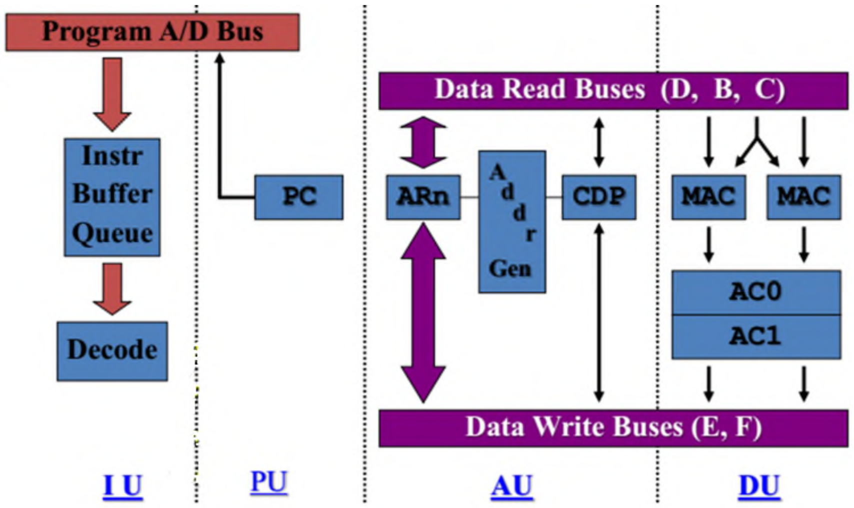


$$y0 = a_0x_0 + a_1x_1 + a_2x_2 + a_3x_3$$

$$y1 = a_0x_1 + a_1x_2 + a_2x_3 + a_3x_4$$

C55x: MAC *AR2+, *CDP+, AC0 :: MAC *AR3+, *CDP+, AC1

Achieves 2 taps/cycle using two MACs in parallel



```
MAC *AR2+, *CDP+, AC0 :: MAC *AR3+, *CDP+, AC1
```

This is possible because of multiple buses in C55x (described later)

Circular buffer addressing

C55x permits circular buffer for filtering

Linear buffer

				x_0	x_1
			x_0	x_1	x_2
		x_0	x_1	x_2	x_3
	x_0	x_1	x_2	x_3	x_4
▶ x_0	▶ x_1	▶ x_2	▶ x_3	▶ x_4	▶ x_5

Shift all past values

Circular buffer

▶ x_0	x_0	x_0	x_0	x_0	▶ x_5
	▶ x_1	x_1	x_1	x_1	x_1
		▶ x_2	x_2	x_2	x_2
			▶ x_3	x_3	x_3
				▶ x_4	x_4

Shift the pointer

Pointer is incremented modulo the buffer size

A linear/circular configuration bit in status register controls whether a pointer is linear or circular.

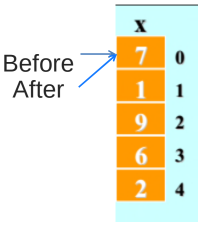
A circular buffer consists of:

- One buffer start address register
- One buffer size register
- Pointer offset stored in ARx or CDP

It is possible to have up to 3 simultaneous circular buffers of different sizes.

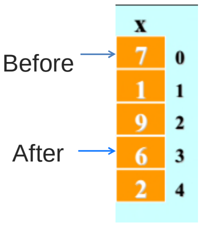
Example:
AR4 is used as the pointer to circular buffer **x** of size 5, with data as shown. Initial value of AC0=0 and AR4=0. Find the values of AC0 and AR4 after each step.

MOV #3, T0



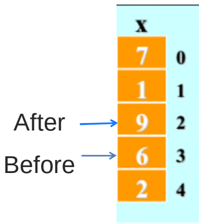
AC0 = 0, AR4 = 0

MOV *(AR4+T0), AC0



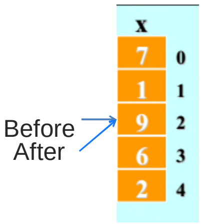
AC0 = 7, AR4 = 3

MOV *+AR4(#4h), AC0



AC0 = 9, AR4 = 2

MOV *AR4(T0), AC0



AC0 = 7, AR4 = 2

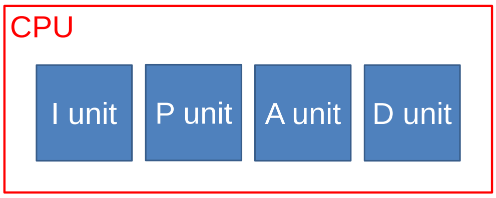
TMS320C55x Architecture

TMS320C55x

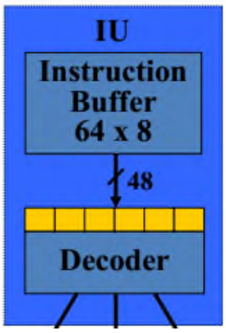
- 16-bit Fixed-Point Processor, Up to 120 MHz Clock
- Two Multipliers (up to 240M MAC operations)
- Two arithmetic logic units (ALU)
- Internal Data/Address Busses: 3 reads, 2 writes
- Separate Program Data/Address Bus
- 64KB Dual-Access RAM, 256KB Single-Access RAM
- Multiple Peripherals supports

CPU

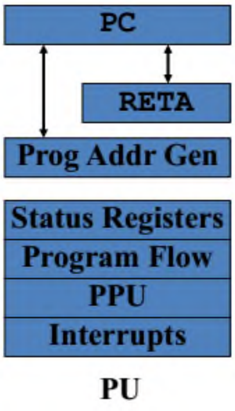
- Instruction buffer (I) unit
- Program flow (P) unit
- Address-data flow (A) unit
- Data computation (D) unit



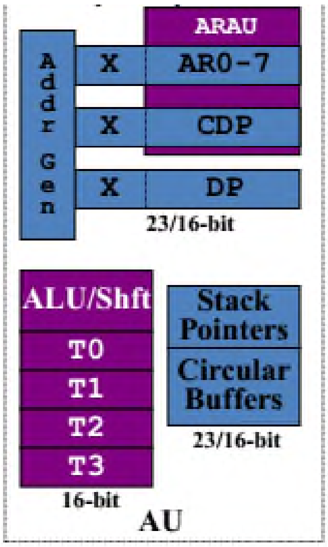
- I (instruction buffer) unit
  - Instruction buffer queue allows block fetching of instructions
  - Instruction decoder supports multiple length instructions



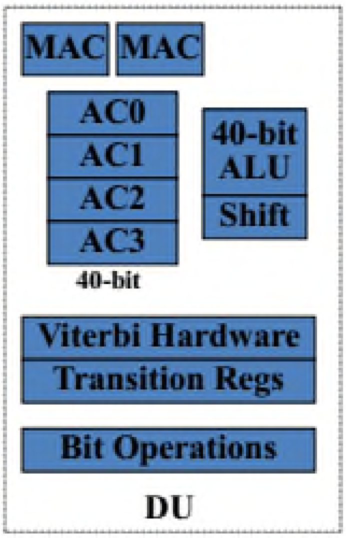
- P (program flow) unit
  - Program address generator and control logic
  - Registers: program flow registers, registers to control block repeat and single repeat, interrupt registers, status registers

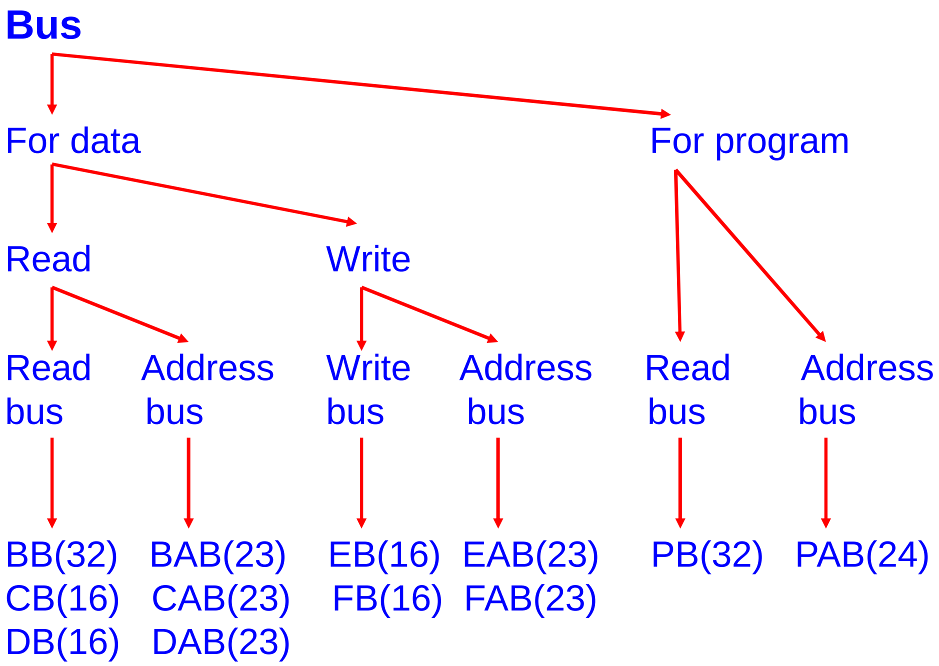


- A (address-data flow) unit
  - Data address generator and A-unit arithmetic logic unit (ALU)
  - Registers: data page register (DP), auxiliary registers (AR), pointers such as coefficient data pointer (CDP), circular buffer registers, temporary registers (T)



- D (data computation) unit
  - Shifter (bit shift, rotate, round, truncate) and D-unit special bit manipulation (BIT)
  - D-unit ALU and two multiply-accumulators (MAC)
  - Registers: accumulators (AC), transition registers





Bus

For data

For program

Read

Write

Read bus

Address bus

Write bus

Address bus

Read bus

Address bus

BB(32)
CB(16)
DB(16)

BAB(23)
CAB(23)
DAB(23)

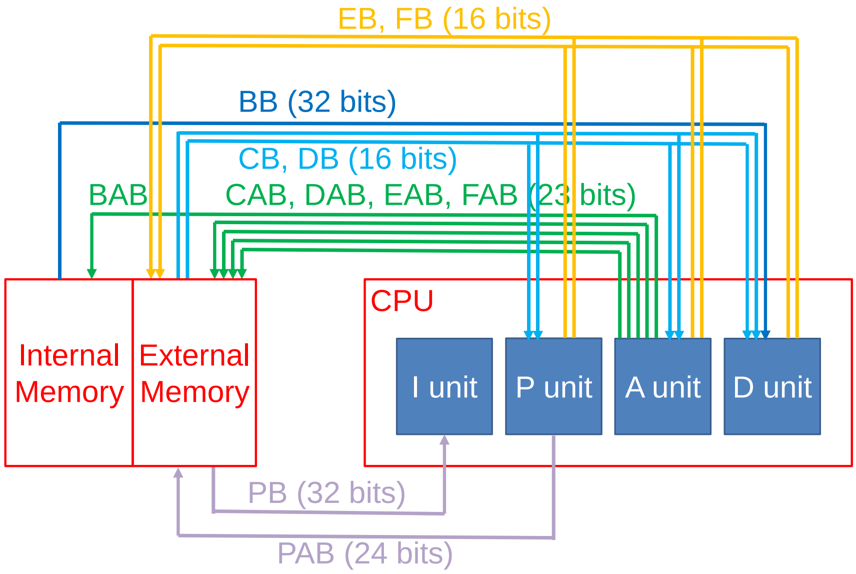
EB(16)
FB(16)

EAB(23)
FAB(23)

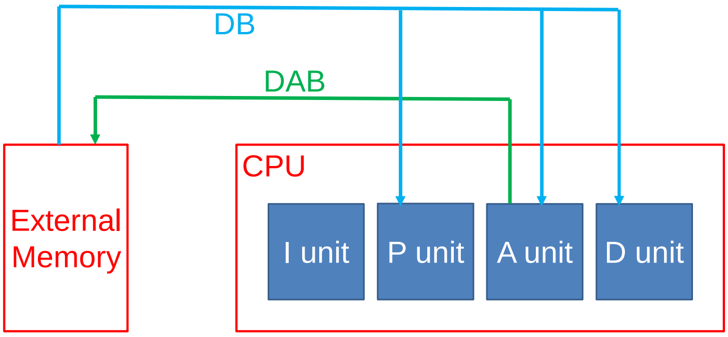
PB(32)

PAB(24)

Data address, data read, data write
Program address, program read

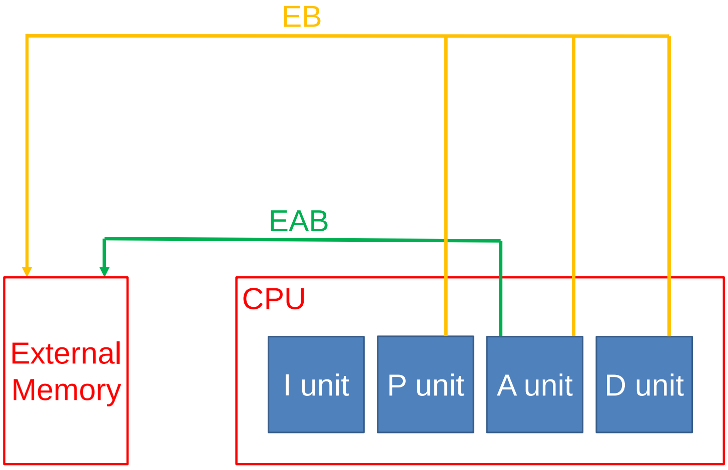


Single (16-bit) data / memory-mapped register / IO read



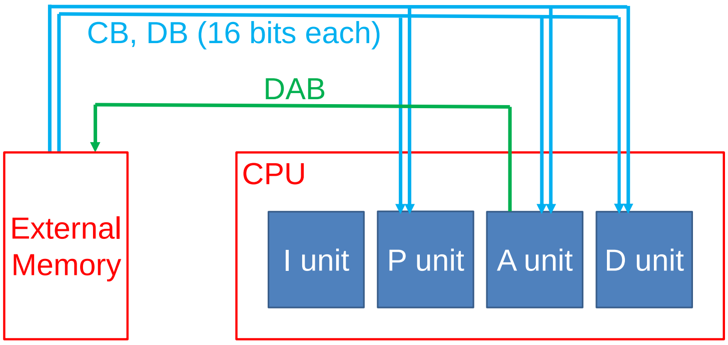
Example: MOV *AR0, AR1 (to A unit)

Single (16-bit) data / memory-mapped register / IO write



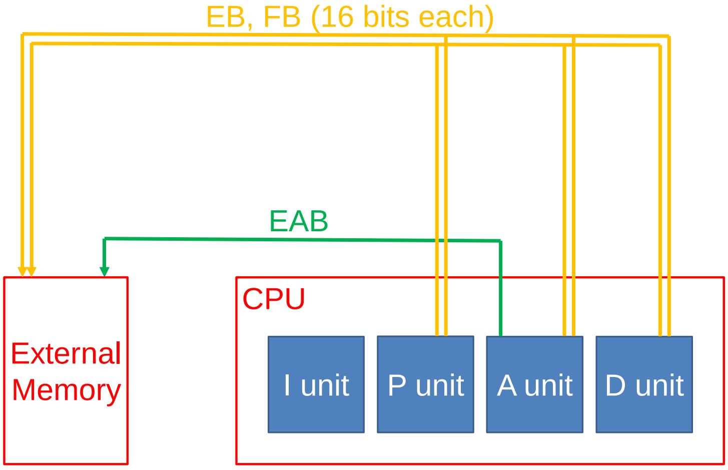
Example: MOV AR0, *AR1 (from A unit)

Long (32-bit) data / memory-mapped register read (reads one 32-bit register or two adjacent 16-bit registers)



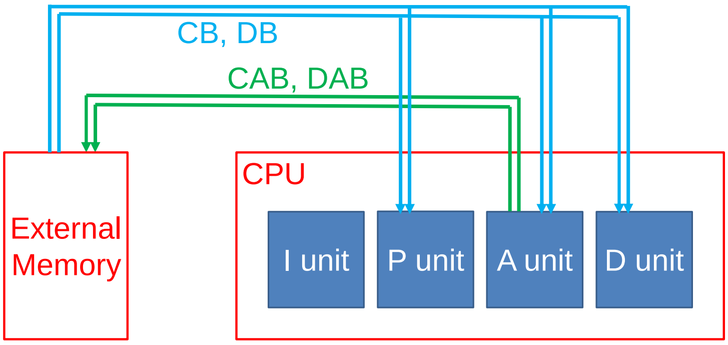
Example: MOV dbl(*AR0), AC0 (to D unit)

Long (32-bit) data / memory-mapped register write (writes one 32-bit register or two adjacent 16-bit registers)



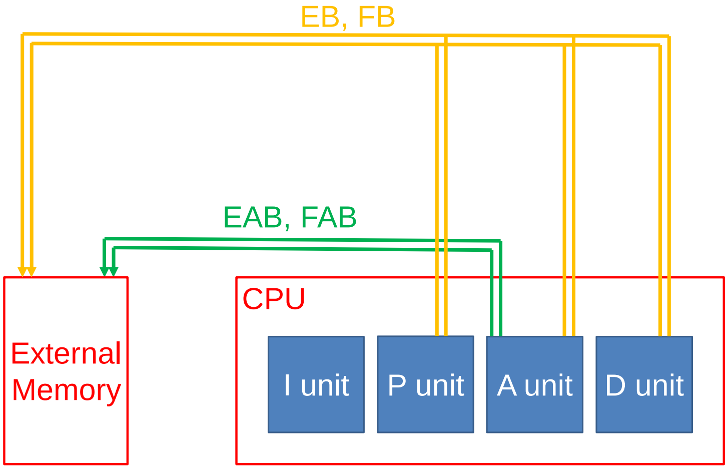
Example: MOV AC0, dbl(*AR0) (from D unit)

Dual (two simultaneous 16-bit) data read
(first operand uses DAB and DB, can be from MMR/IO)
(second operand uses CAB and CB, must be data)



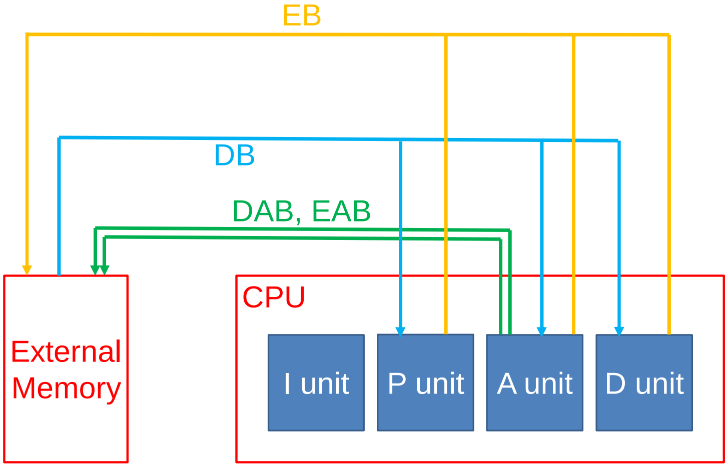
Example: MOV *AR0, *AR1, AC0 (to D unit)

Dual write
(first operand uses FAB and FB, must be data)
(second operand uses EAB and EB, can be to MMR/IO)



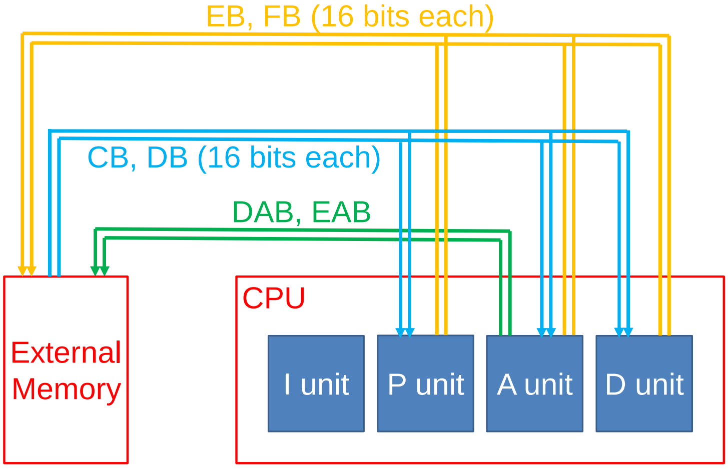
Example: MOV AC0, *AR0, *AR1 (from D unit)

Single data read || single data write
(read uses DAB and DB, parallelly write uses EAB and EB)



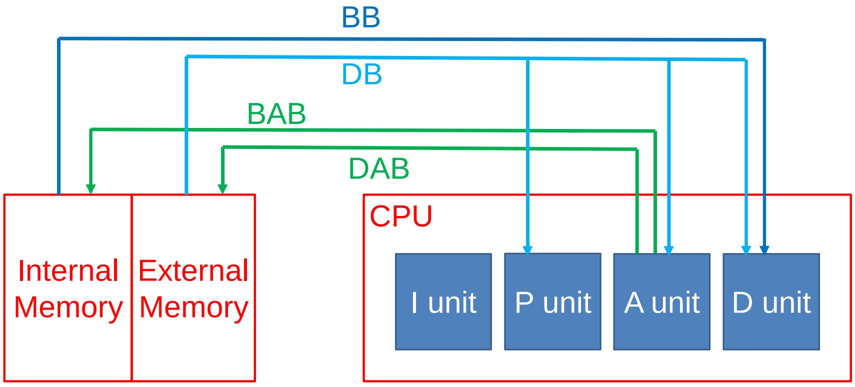
Example: user-defined parallel instructions

Long data read || long data write
(read uses DAB, CB, DB, parallelly write uses EAB, EB, FB)



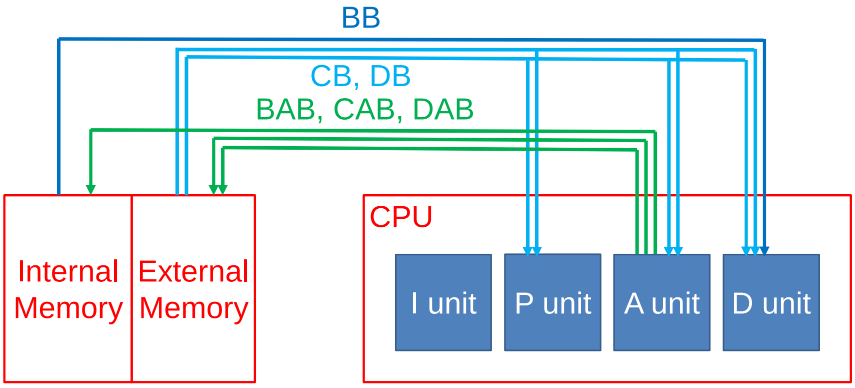
Example: user-defined parallel instructions

Single data read || single/dual coefficient data read
(data read uses DAB,DB from data memory; one or two 16-bit coefficients are read using BAB,BB from internal memory)



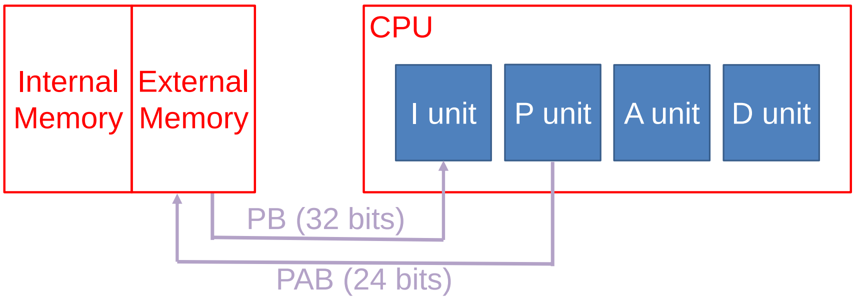
Example: MACM *AR0, *CDP, AC0 uses BAB and BB bus to fetch the coefficient (to D unit)

Dual data read || single/dual coefficient data read
(data read uses CAB,DAB,CB,DB from data memory; coefficient read uses BAB, BB from internal memory)



Example: MAC *AR0, *CDP, AC0 :: MAC *AR1, *CDP, AC1 (to D unit)

Instruction (32-bit) fetch



Arithmetic Operations with Conditions

Operations

- Upto 40 bit add/subtract in D unit ALU
- 16 bit add/subtract in A unit ALU
- 17-bit x 17-bit multipliers in D unit MACs

Overflow

- In D unit ALU/MAC, overflow is detected typically at bit 31
- Accumulator overflow status bit (ACOVx) is set to 1

Example:
Find if there is overflow in 16-bit addition of the following numbers (decimal values for Q0.15 format).

4000h + 3000h = 0.5 + 0.375 = 7000h = 0.875, no overflow

5000h + 3000h = 0.625+0.375 = 8000h = –1, overflow

```
 0 0101 0000 0000 0000
 0 0011 0000 0000 0000
----------------------
 0 1000 0000 0000 0000
```

C000h + 7000h = –0.5+0.875 = 3000h = 0.375, no overflow

C000h + B000h = –0.5–0.625 = 7000h = 0.875, overflow

Saturation
During overflow, the result may be saturated.

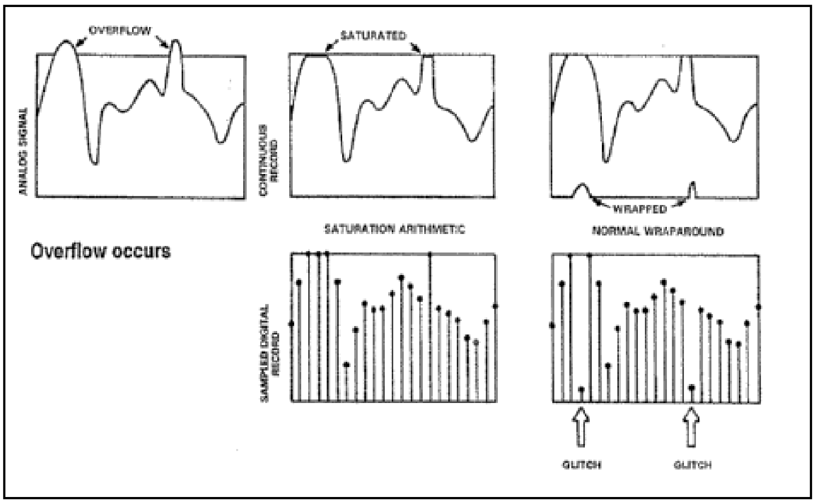
- If saturation mode bit for D unit (SATD) is 1, then positive result is typically saturated to 00_7FFF_FFFFh and negative result is saturated to FF_8000_0000h

- If saturation mode bit for A unit (SATA) is 1, then positive saturated to 7FFFh and negative to 8000h

- If saturation on multiplication bit (SMUL) is 1, then same as SATD for multiplication result

In the previous example, with saturation,
5000h + 3000h = 7FFFh = 0.99997
C000h + B000h = 8000h = –1

- Overflow gives wrap around effect for 2’s complement numbers
- Saturation clips the result, introducing less error



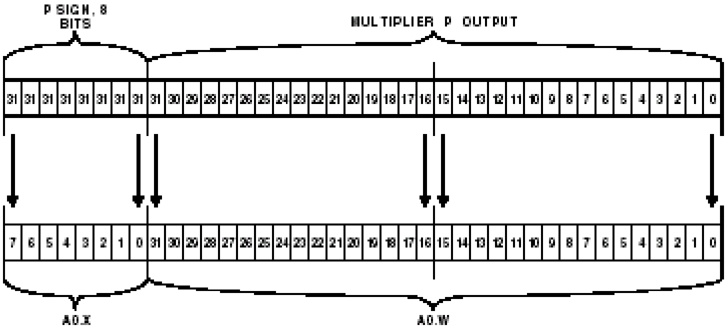
Fractional mode multiplication

In signal processing, typically fractional numbers are multiplied.

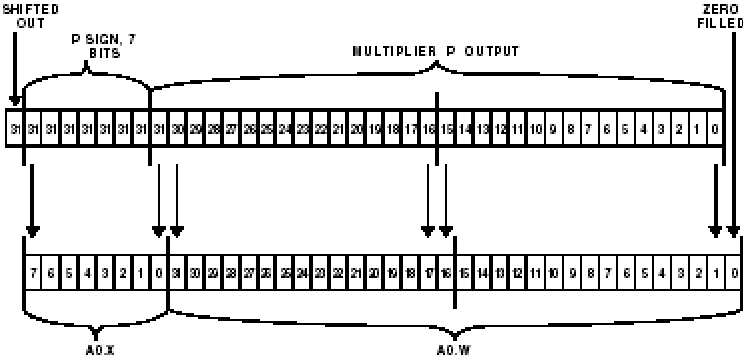
- Let 16-bit Q0.15 inputs be
  0.100 0000 0000 0000 = 0.5 and
  0.010 0000 0000 0000 = 0.25
- Sign extend to 17-bit Q1.15 format:
  0 0.100 0000 0000 0000
  0 0.010 0000 0000 0000
- Multiply to obtain 32-bit Q1.30 product:
  00.00 1000 0000 0000 0000 0000 0000 0000 = 0.125
- Need to shift left by 1 bit to obtain 32-bit Q0.31 fraction:
  0.001 0000 0000 0000 0000 0000 0000 0000

If fractional mode status bit (FRCT) is 1, products are automatically shifted left by 1 bit.

Integer mode (bit = 0)



Fractional mode (bit = 1)



Example:
Find the product of 7000h and 6000h without the fractional mode (decimal values for Q15.0 format) and with the fractional mode (decimal values for Q0.15 format).

Without the fractional mode,
7000h becomes 0 0111 0000 0000 0000 = 28672
6000h becomes 0 0110 0000 0000 0000 = 24576
Product is 0010 1010 0000 0000 0000 0000 0000 0000
= 2A00_0000h = 704643072

With the fractional mode,
7000h is same as above = 0.875
6000h is same as above = 0.75
Product is same as above. After left shift, it becomes
0101 0100 0000 0000 0000 0000 0000 0000
= 5400_0000h = 0.65625