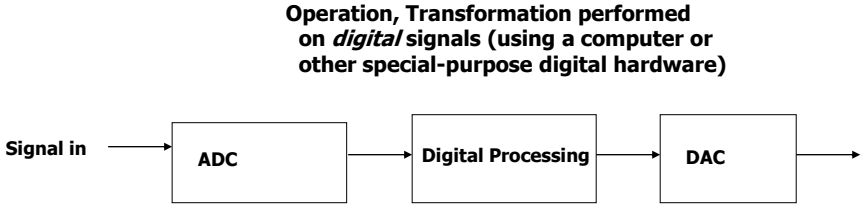
Real-Time Digital Signal Processing

Introduction

What is Digital Signal Processing?



What is Real-Time Digital Signal Processing?



Time-constrained Operation or Transformation performed on *digital* signals within a required period of time to maintain synchronization with occurring events.

Example:

- **Processor clocked at 120 MHz and can perform 120MIPS**
  - **Sampling rate = 48KHz (Digital Audio Tape - DAT) number of instructions per sample = $(120 \times 10^6)/(48 \times 10^3) = 2500$.**
  - **Sampling rate = 8KHz (voice-band, telephony) number of instructions per sample = 15000.**
  - **Sampling rate = 75MHz (CIF 360x288 Video at 30 frames per second) number of instructions per sample = 1.6.**

Real-Time Digital Signal Processing

Constraints:

- **real-time DSP applications limited to cases where the required sampling rate is sufficiently lower than the processor's instruction rate**

Challenge:

- **Produce working code.**
- **Produce sufficiently compact code to execute in real-time.**
- **A sufficient number of instructions need to be performed between sample periods.**

What is DSP?

DSP = Digital Signal Processing

OR

DSP = Digital Signal Processor?

DSP used to denote both

- **meaning can be deduced from the context in which the term DSP is used.**

What is a Digital Signal Processor (DSP)?

- **Microprocessor specifically designed to perform fast DSP operations (e.g., Fast Fourier Transforms, Multiply & Accumulate)**

Why Go Digital?

Programmability

- **One hardware can perform several tasks.**
- **Upgradeability and flexibility.**

Repeatability

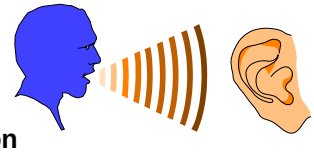
- **Identical performance from unit to unit.**
- **No drift in performance due to temperature or aging.**

Immune to noise

Offers higher performance: CD players versus phonographic turntable

DSP Applications

Speech processing



- Speech compression
- Speech recognition
- Speaker Identification, Verification
- Speech synthesis
- Speech enhancement, Echo cancellation

Audio Processing

- Compression
- 3-D reproduction

DSP Applications

Image Processing



- Image compression
- Pattern recognition
- Ghost cancellation
- Noise reduction
- Deblurring
- Object tracking
- Image fusion

Video Processing/compression, tracking...

DSP Applications

MODEM

- correlators (matched filters)
- echo cancellers
- equalizers

Cellular Telephony

- speech compression
- array processing

Software Radio

DSP Market

Important companies

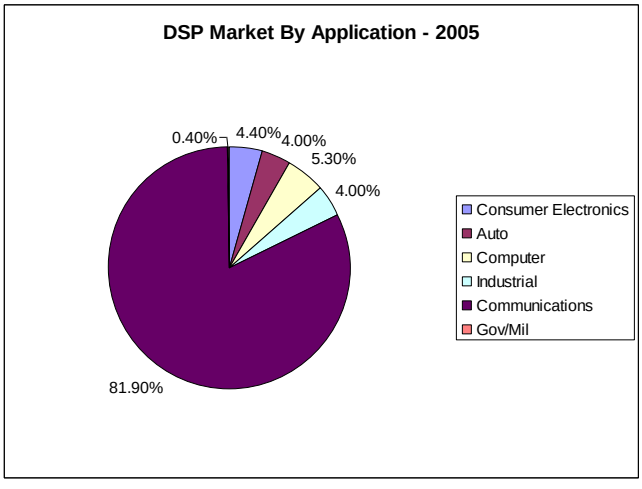
- Texas Instruments → **Kits available in the lab are from TI**
- Freescale Semiconductor
- Analog Devices
- Philips Semiconductors
- Agere Systems
- Toshiba
- DSP Group
- NEC Electronics

DSP Market – By Application

Communications applications (i.e: wireless) Jumped from 68.3% in 2003 to 82% in 2005.

Expectations:

1) **DSP market will increase by 9% in 2006**
2) **Followed by an 18% increase in 2007.**
3) **A boom of 27% in 2008**



Ref: IC Insights

http://www.icinsights.com/news/releases/press20051123.html

What is special about Signal Processing Applications?

Large number of samples being continuously fed to the system (samples or blocks).

Repetitive Operations:

- **The same operation being applied to different set of samples**
- **Parallel processing**

Vector and Matrix Operations

Real time operations

Example: Digital Filtering

The two most common real-time digital filters are:

- **Finite Impulse Filter (FIR)**
- **Infinite Impulse Filter (IIR)**

The basic FIR Filter equation is

$$y[n] = \sum h[k].x[n-k]$$

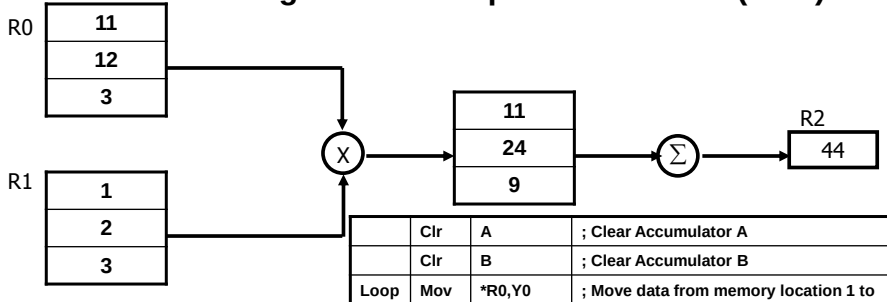
where h[k] is an array of constants

In C language

```
y[n]=0;
For (n=0; n<N;n++)
{
For (k = 0;k<N;k++)
//inner loop
        y[n] = y[n] + h[k]*x[n-k];}
```

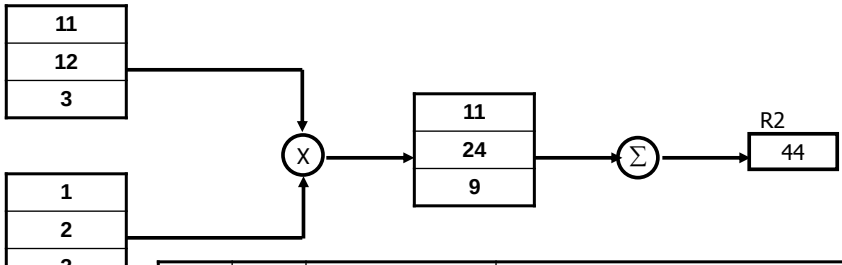
Only Multiply and Accumulate (MAC) is needed!

MAC using General Purpose Processor (GPP)



	Clr	A	; Clear Accumulator A
	Clr	B	; Clear Accumulator B
Loop	Mov	*R0,Y0	; Move data from memory location 1 to register Y0
	Mov	*R1,X0	; Move data from memory location 2 to register X0
	Mpy	X0,Y0,A	;X0*Y0 ->A
	Add	A,B	;A + B -> B
	Inc	R0	;R0 + 1 -> R0
	Inc	R1	;R1 + 1 -> R1
	Dec	N	;Dec N (initially equals to 3)
	Tst	N	;Test for the value

MAC using DSP



	Clr	A	;Clear Accumulator A
	Rep	N	; Rep N times the next instruction
	MAC	*(R0)+, *(R1)+, A	; Fetch the two memory locations pointed by R0 and R1, multiply them together and add the result to A, the final result is stored back in A
	Mov	A, *R2	; Move result to memory

GPP Drawbacks

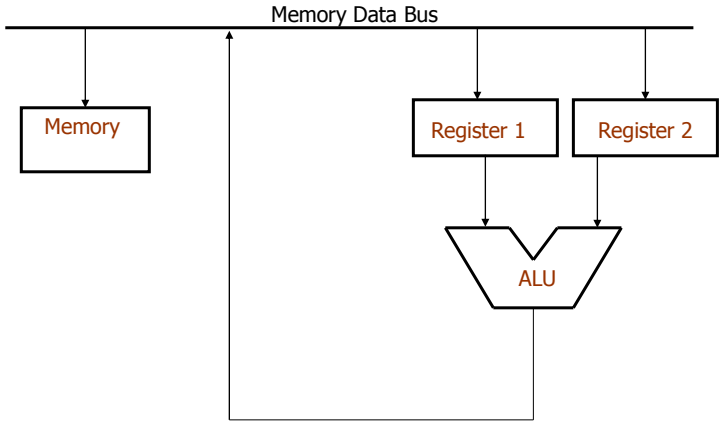
More instructions/task

Common Memory for data and program

- **Limited bus/memory bandwidth**

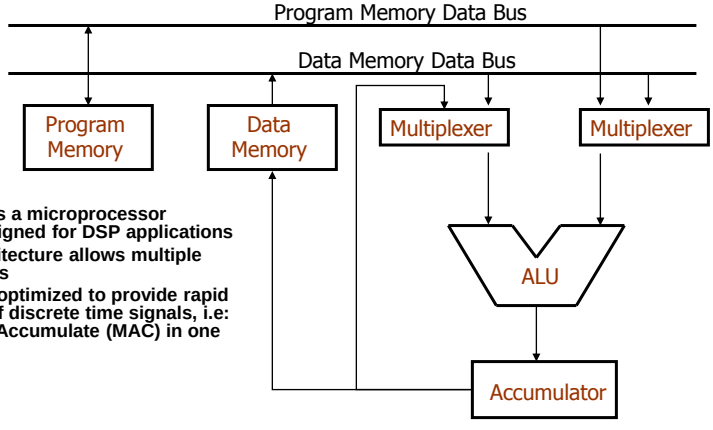
Solution : DSP Architectures

GPP – Data Path Only



Same memory for program and data

Digital Signal Processors – Data Path Only

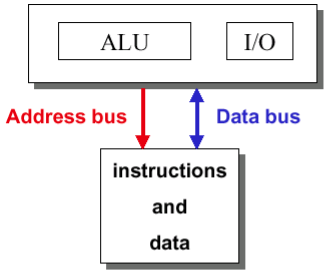


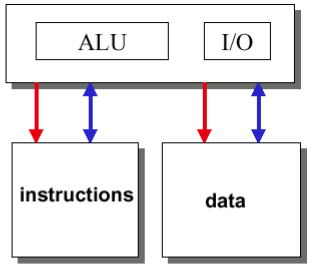
A DSP Chip is a microprocessor specially designed for DSP applications

Harvard architecture allows multiple memory reads

Architecture optimized to provide rapid processing of discrete time signals, i.e: Multiply and Accumulate (MAC) in one cycle

Memory structures





Von Neuman architecture **Area efficient but requires higher bus bandwidth because instructions and data must compete for memory.**

Harvard architecture **was coined to describe machines with separate memories. Speed efficient: Increased parallelism.**

DSP versus GPP

Multiple parallel units
- **multiply accumulate (possibly several units)**
- **address calculation in parallel to processing**
- **barrel shifter**

Memory Access
- **special ALU for address calculation**
- **Bit reversed addressing**
- **circular addressing**

Automatic loops
- **Software looping: writing assembly code to perform branching**
- **Hardware looping: dedicated hardware loop counter register**

Hardware support for managing arithmetic computation (in GPP it needs multiple cycles)
- **Shifters**
- **Guard bits**
- **Saturation**

Preventing Overflow!!

Enhancing DSP Architectures

More parallelism

- **Increase the number of operations that can be performed in each instruction**
  - **Adding More Executing units (i.e: Multipliers)**
- **Increase the number of instructions that can be issued and executed in every cycle**

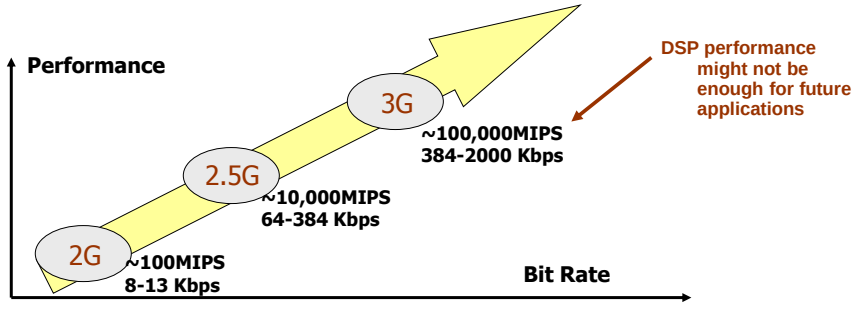
Highly specialized hardware in core

Co-processors

Multi-Core DSPs

Why Consider DSP Alternatives

Wireless Systems requires more and more high performance and higher bandwidth



What are the alternatives

High-performance GPPs with DSP enhancements.

- **Eliminating the need of a DSP and GPP for many products and thus reducing cost**
- **Example: Pentium 4**
  - **Single Instruction Multiple Data (SIMD) instructions allowing identical operations on multiple pieces of data in parallel.**
  - **144 new special instructions providing advanced capabilities for applications such as 3D graphics, video encoding/decoding, and speech recognition.**
  - **Several Data Types (floating/integer)**

Multi-Core DSPs

Application Specific Integrated Circuits (ASIC)

Field Programmable Gate Array (FPGA)

ASIC

Uses hard-wired logic with varied architectures according to the application (i.e: 256 point hardware implemented FFT)

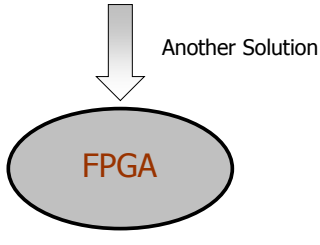
Sometimes includes proprietary processor cores (i.e: licensed Intellectual Properties – IP)

ASIC - Advantages

Speed
Reduced Power Consumption
Cost/performance
Design Flexibility

ASIC- Disadvantage

Large development costs
Lengthy development cycles
Inflexibility



What is FPGA

It is a network of reconfigurable hardware with reconfigurable interconnect controlled by a switching matrix

Historically used for prototyping

Recently includes DSP features

- **Major Companies DSP + FPGA: ALTERA (e.g.: Stratex) & XILINX (e.g.: Virtex II)**

FPGA - Advantages

More Flexible than ASIC

Huge Performance Gain in Some Applications

Re-use Hardware for different applications

FPGA - Disadvantages

Long Development Cycle
Expensive compared to DSP
Much higher power consumption compared to DSP
Slow time to market compared to DSP

Why Still use DSP?

Several applications are not suited to be implemented in FPGA

- **Parallelism is sometimes inherently limited**
- **Speed is not always the highest factor to consider**

FPGA is still very expensive for terminal products (i.e: cell phones)

Why Still use DSP?

Comparison: DSP, FPGA, ASIC (ref: Bill Dally, Stanford University, IEEE ICASSP04 Talk)

DSP
- < 10 MOPS/mW
- ~0.1 GOPS/$
- < 10 GOPS peak performance
- 1 M $ programming cost
- Programmable

ASIC
- 50-200 MOPS/mW
- 2-10 GOPS/$
- Up to 1000 GOPS peak performance
- 10M-15M $ design cost
- Fixed

FPGA
- 2-10 MOPS/mW
- ~1 GOPS/$
- Up to 500 GOPS peak performance
- ~5M $ design cost
- Reconfigurable

- New improved DSPs with more efficiency and parallelism (e.g., multi-core)

Types of DSP

Low End Fixed Point
- **TMS320C2XX, ADSP21XX, DSP56XXX**

High End Fixed Point
- **TMS320C55XX, DSP16XXX,**
- **ADSP215XX, DSP56800**

Floating Point
- **TMS320C3X, C67XX, ADSP210XX, DSP96000, DSP32XX**

Berkeley Design Tech. Inc. Pocket Guide to DSPs

http://www.bdti.com/pocket/pocket.htm

Fixed Point Vs Floating Point

Fixed Point/Floating Point

- **fixed point processor are :**
  - **cheaper**
  - **smaller**
  - **less power consuming**
  - **Harder to program**
    - **Watch for errors: truncation, overflow, rounding**
  - **Limited dynamic range**
  - **Used in 95% of consumer products**
- **floating point processors**
  - **have larger accuracy**
  - **are much easier to program**
  - **can access larger memory**
- **It is harder to create an efficient program in C on a fixed point processors than on floating point processors**

Fixed Point Vs Floating Point

TEXAS INSTRUMENTS

Floating Point

Applications

•Modems

•Digital Subscriber Line (DSL)

•Wireless Basestations

•Central Office Switches

•Private Branch Exchange (PBX)

•Digital Imaging

•3D Graphics

•Speech Recognition

•Voice over IP

Fixed Point

Applications

•Portable Products

•2G, 2.5G and 3G Cell Phones

•Digital Audio Players

•Digital Still Cameras

•Electronic Books

•Voice Recognition

•GPS Receivers

•Headsets

•Biometrics

•Fingerprint Recognition

What Chip will be used?

TI TMS320C5515

- **Family: TMS320C55xx**
- **Kit: TMS320C5515eZdsp**
- **Software: TI Code Composer Studio**

Software Coding

Write Code in C

Compile to create Assembly code

Assemble the code to create object code and link

Use simulator to test the speed of the code

If code is not fast enough - rewrite the C code and test again. If not fast enough yet, write in Assembly language

Why use Assembly?

Most C compilers for DSP chips produce code that does not fully utilize the capabilities of the DSP

- **Data Fetch parallel to execution**
- **Parallel execution**

The C code can be 3 to 30 times slower than the best assembly code possible. Especially in the signal processing parts of the code.

The problem is more acute with fixed-point DSPs

But I don't want to write Assembly



Have somebody else write assembly for you

- **use libraries**

Rewrite your C code to produce a better assembly code

Test and profile your code to see which parts of the software take most of the CPU time. Limit Assembly code to subroutines:

- **That the program spends a lot of time in them**
- **That benefit from the special functions of DSP such as MACS and parallel execution and fetch.**

How to Write a Better C Code

Use Simple Loops

Avoid if statements in loops

Avoid subroutine calls statements in loops

Use inline subroutines

- **Compiler inserts function directly into the caller's code stream (conceptually similar to what happens with a #define macro)**
- **Avoids the subroutine call overhead (saving volatile variables)**
- **Increases code size**

Avoid division and modulo operations

Use and (&) and shift when possible

Use 5%/80% rule

- **Program in Assembly the 5% of the lines of code of the project that take 80% of the CPU load.**
- **Try to change your code to fit existing assembly routines.**

DSP Processor Selection Criteria

Wide range of DSP processors are available, which one to select?

It depends about the application: what are the most important criteria?

- **Speed.**
- **Memory bandwidth.**
- **Cost.**
- **Ease of use of development tools.**
- **Packaging options.**
- **On-chip integration.**
- **Power consumption.**

DSP Processor Selection Criteria

Use of available benchmarks:

- **BDTI kernel benchmarks.**
- **BDTI application benchmarks.**

Use a hierarchical approach to pick a processor

- **List your requirements.**
- **Start with critical criteria; and prioritize the remaining ones.**
- **Trade-offs may be required.**

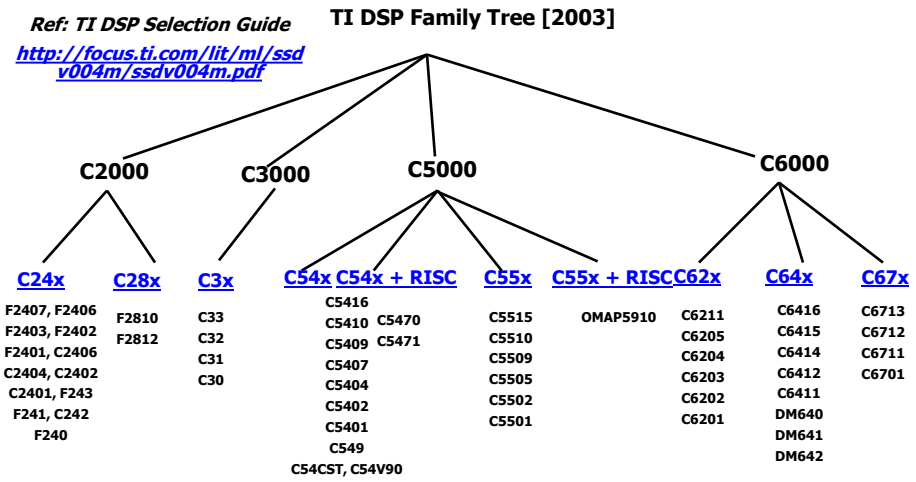
DSP C5000

Architecture Overview

Objectives

- TI DSP family tree
- Discuss pipeline phases
- List the key features of the C55x memory map and peripherals
- Give some details about some of the CPU registers

TI DSP Family Tree

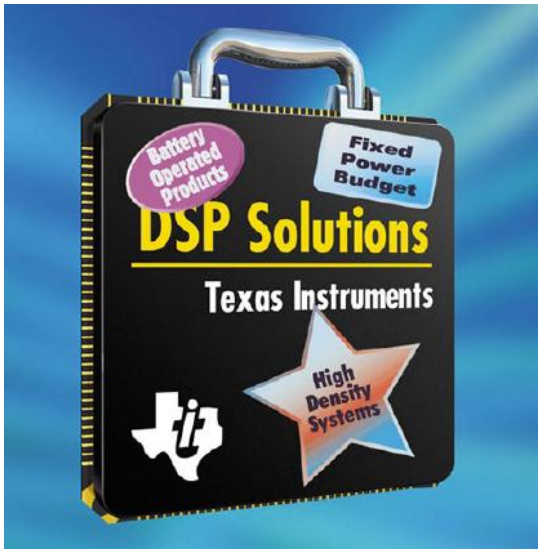


http://en.wikipedia.org/wiki/Texas_Instruments_TMS320

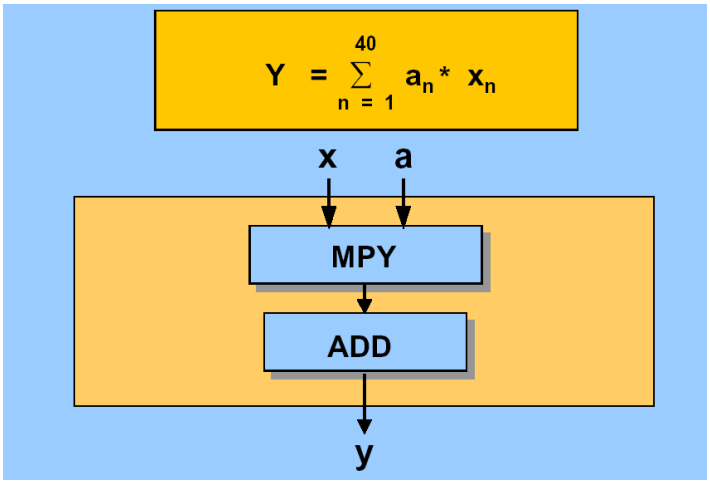
TI family tree : C5000

- TMS320C5000:
  - 16-bit fixed-point DSPs with performance up to 300MHz (600 MIPs).
  - Ultra low power consumption
  - High peripheral integration & large on-chip
  - Focus on TMS320C55x generation
    - Today, most C55x DSPs are sold as discrete chips
    - OMAP1 chips combine an ARM9 (ARMv5TEJ) with a C55x series DSP
    - OMAP2420 chips combine an ARM11 (ARMv6) with a C55x series DSP

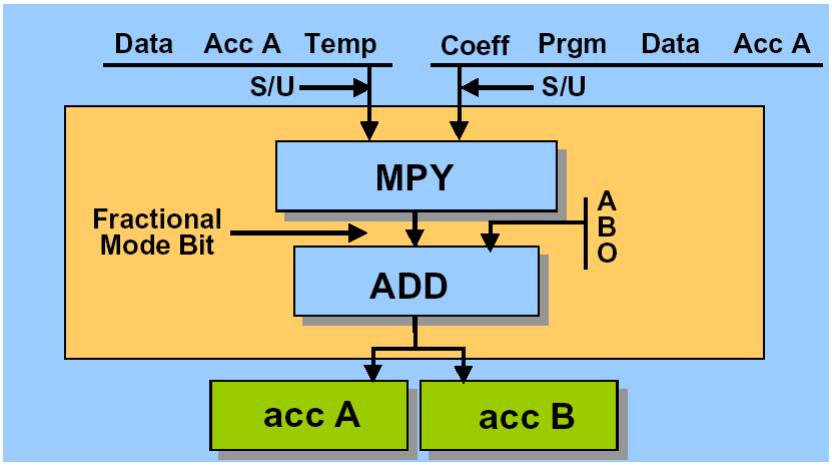
What Constitutes a Good DSP?



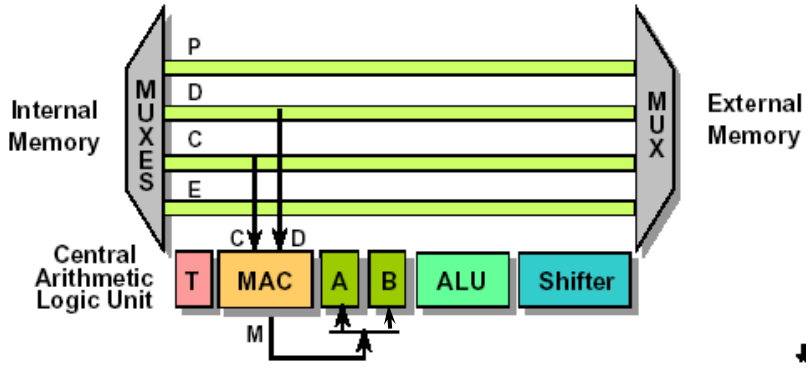
DSP Requires Multiply and Accumulate



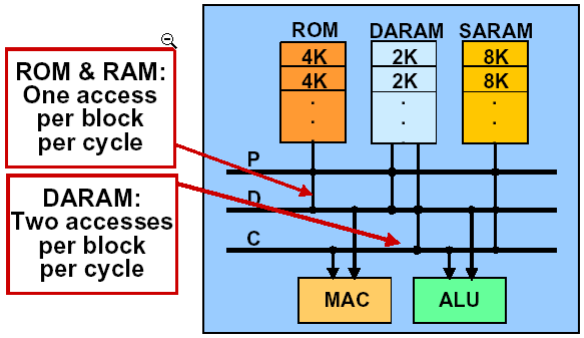
Multiply and Accumulate Unit



Multiple Buses

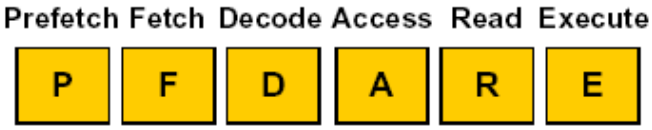


Internal Memory for Fast Access



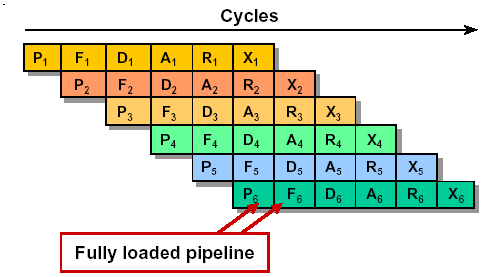
Instruction Pipeline for Fast Execution

Instruction is broken into smaller tasks that can be executed in parallel

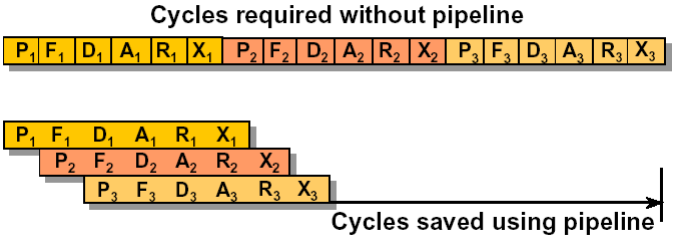


- Prefetch: Calculate address of instruction
- Fetch: Collect instruction
- Decode: Interpret instruction
- Access: Collect address of operand
- Read: Collect operand
- Execute: Perform operation

Sequential Processing of Instructions



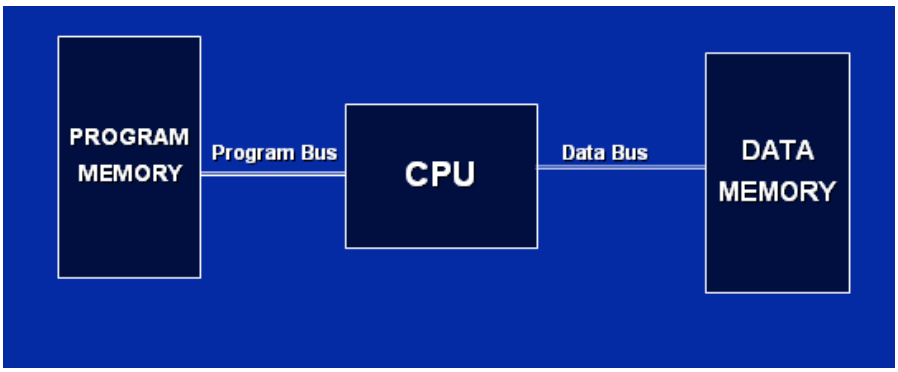
Less Cycles per Instruction
Less Power Consumption



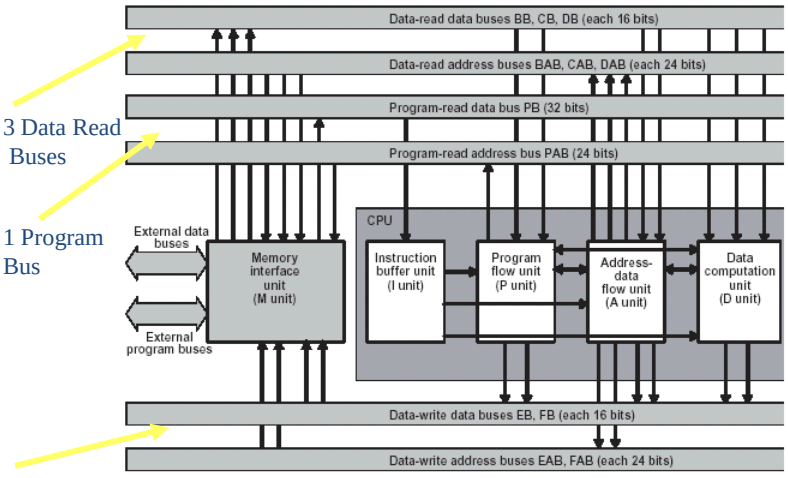
Texas Instruments
C5000 Solutions



Basic Harvard Architecture
1st DSP Generation



TMS320C55X DSP Block Diagram



TMS320C55X key features

- Instruction unit
  - 32 x 16-bit Instruction buffer queue (IBQ)
  - Fetches instructions from memory into CPU
  - Different instruction lengths. (1 byte to 8 bytes)
  - Can hold small instruction loop, even conditional program flow control can be implemented (more efficient and reduce in power)
- Program unit
  - Consists of program counter (PC), four status registers (ST0-3), address generator, pipeline protection unit.
  - 24-bit address bus (16Mb)
  - Branch, call, return, conditional execution, interrupt will cause nonsequential program execution which breaks down the pipeline.

TMS320C55X key features

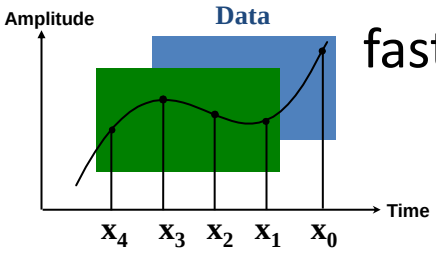
- Address unit
  - 8 x 23-bit XARS, 4 x 16-bit T, 23-bit XCDP, 23-bit XSP
  - 16-bit ALU for simple arithmetic instructions
  - 2 XAR and XCDP in parallel (1 clock cycle)
  - 5 circular buffers
- Data unit
  - 2 MAC, 40-bit ALU, 4x40bit AC, barrel shifter ($2^{-32} - 2^{31}$), rounding, saturation logic.
  - 2 data-read paths and coefficient path can be used simultaniously by dual MAC.
  - 17bit x 17bit multiplication and 40-bit addition in 1 cycle with saturation option



TMS320C55X key features

- Twelve independent buses:
  - Three data read buses
  - Two data write buses
  - Five data address buses
  - One program read bus
  - One program address bus

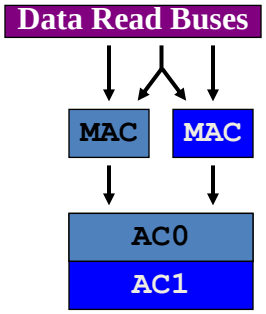
With the C55x it can be done faster!



Coeffs

a0
a1
a2
a3

- 2 taps/cycle



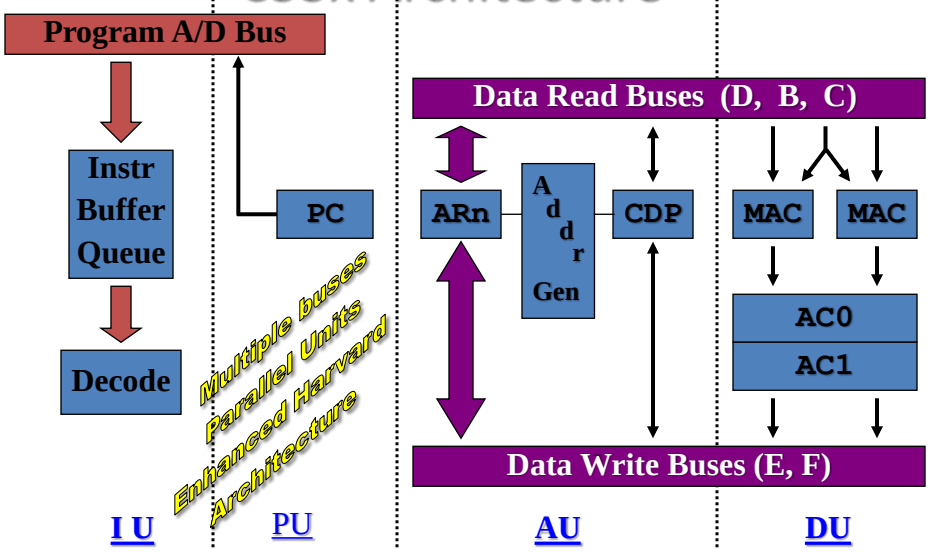
Results

$y0 = a_0x_0 + a_1x_1 + a_2x_2 + a_3x_3$

$y1 = a_0x_1 + a_1x_2 + a_2x_3 + a_3x_4$

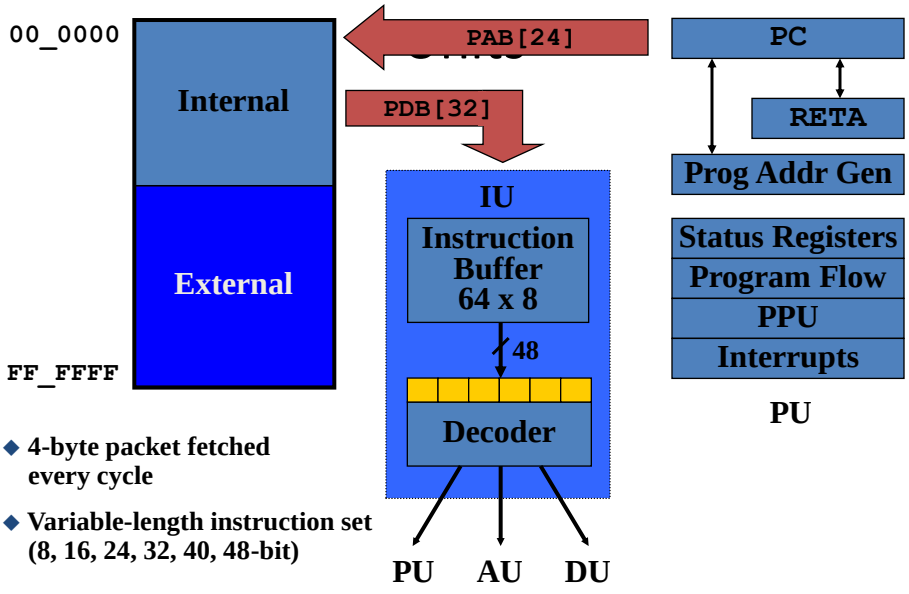
C55x: `MAC *AR2+, *CDP+, AC0` :: `MAC *AR3+, *CDP+, AC1`

C55x Architecture



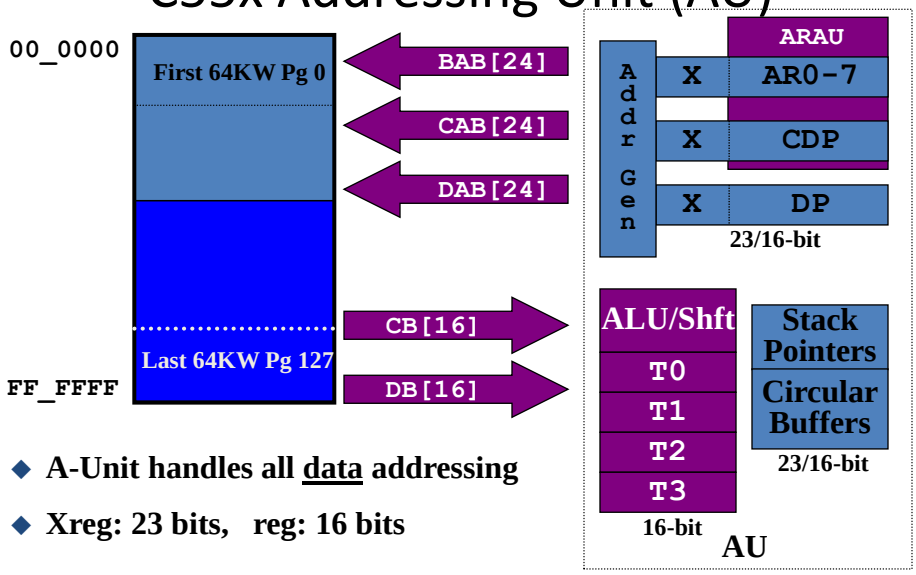
`MAC *AR2+, *CDP+, AC0 :: MAC *AR3+, *CDP+, AC1`

C55x Program and Instruction



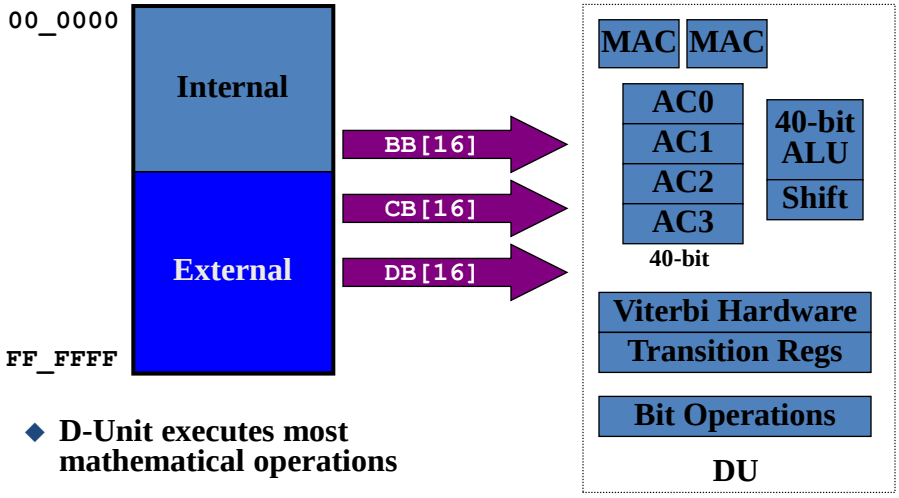
- 4-byte packet fetched every cycle
- Variable-length instruction set (8, 16, 24, 32, 40, 48-bit)

C55x Addressing Unit (AU)



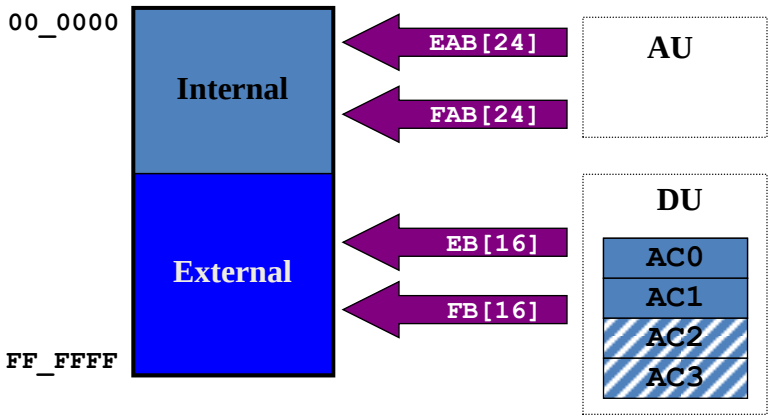
- A-Unit handles all <u>data</u> addressing
- Xreg: 23 bits, reg: 16 bits

C55x Data Computation Unit (DU)



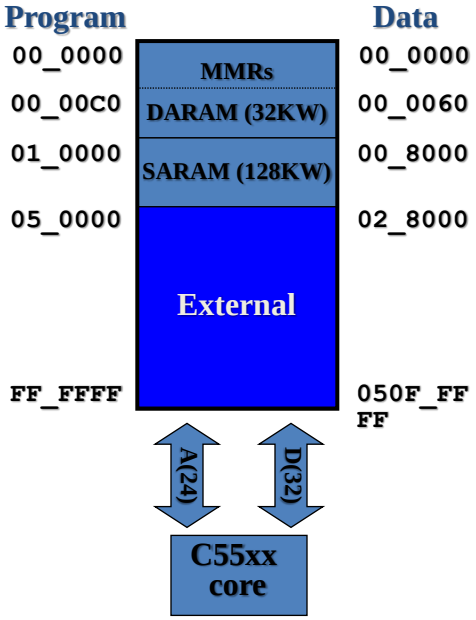
- D-Unit executes most mathematical operations
- Now, what happens to the result?...

C55x Writes (E and F buses)

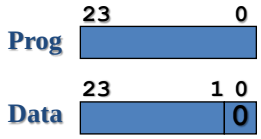


- 32-bit write in one cycle

C5515 Unified Memory Map



- **Program and data share the same map**
- **2 ways to view the map:**



1. Program - (Bytes)

- 16M x 8-bit, linear 24-bit addresses

- Used by fetch/decode logic

2. Data (Words)

- 8M x 16-bit, segmented into 64K pages, 23-bit address

- Most code written by a user will access data

Memory Access

- 16M bytes of memory are addressable as program space or data space
- When the CPU uses program space to read program code from memory, it uses 24-bit addresses to reference bytes.
- When program accesses data space, it uses 23-bit addresses to reference 16-bit words.
- In both cases, the address buses carry 24-bit values, but during a data-space access, the least significant bit on the address bus is forced to 0.

Data Memory

❖**Data space is divided into 128 main data pages (0 through 127) of 64K addresses each.**

❖**An instruction that references a main data page concatenates a 7-bit main data page value with a 16-bit offset.**

❖**On data page 0, the first 96 addresses (00 0000h-00 005Fh) are reserved for the memory-mapped registers (MMRs).**

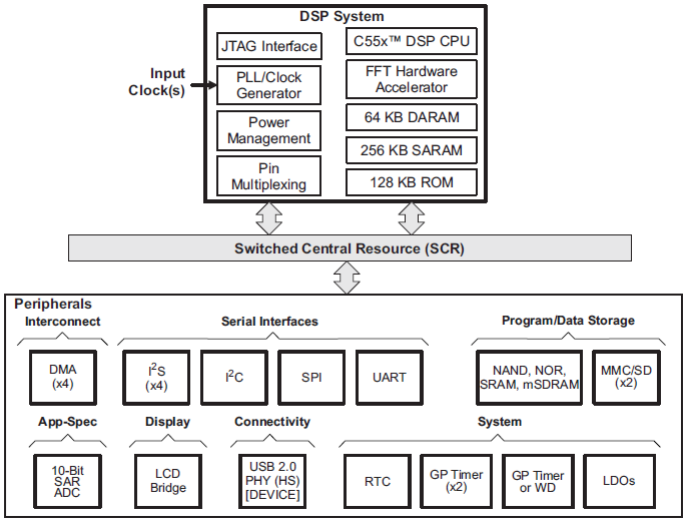
I/O Memory

- I/O space is separate from data/program space and is available only for accessing registers of the peripherals on the DSP. The word addresses in I/O space are 16 bits wide, enabling access to 64K locations

- The CPU uses the data-read address bus DAB for reads and data-write address bus EAB for writes. When the CPU reads from or writes to I/O space, the 16-bit address is concatenated with leading 0s.

Example, suppose an instruction reads a word at the 16-bit address 0102h. DAB carries the 24-bit value 00 0102h.

Functional diagram C5515



- see SPRU317K.pdf for more information

CPU Registers

C55x

C55x CPU Registers

- The study of CPU registers gives a very good understanding on the processor architecture.
- The following table summarizes a set of registers.

C55x CPU 1 of 3

Abbreviation	Name	Size
AC0–AC3	Accumulators 0 through 3	40 bits
AR0–AR7	Auxiliary registers 0 to 7	16 bits
BK03, BK47, BKC	Circular buffer size registers	16 bits
BRC0, BRC1	Block-repeat counters 0 & 1	16 bits
BRS1 BRC1	Save register	16 bits
BSA01, BSA23,BSA45, BSA67, BSA	Circular buffer start address registers	16 bits
CDP	Coefficient data pointer (low part of XCDP)	16 bits
CDPH	High part of XCDP	7 bits

C55x CPU Registers 2 of 3

CFCT	Control-flow context register	8 bits
CSR	Computed single-repeat register	16 bits
DBIER0, DBIER1	Debug interrupt enable registers 0 and 1	16 bits
DP	Data page register (low part of XDP)	16 bits
DPH	High part of XDP	7 bits
IER0, IER1	Interrupt enable registers 0& 1	16 bits
IFR0, IFR1	Interrupt flag registers 0 and 1	16 bits
IVPD, IVPH	Interrupt vector pointers	16 bits
PC	Program counter	24 bits
PDP8	Peripheral data page register	9 bits
REA0, REA1	Block-repeat end address registers 0 and 1	24 bits

C55x CPU Registers 3 of 3

RETA	Return address register	24 bits
RPTC	Single-repeat counter	16 bits
RSA0, RSA1	Block-repeat start address registers 0 and 1	24 bits
SP	Data stack pointer	16 bits
SPH	High part of XSP and XSSP	7 bits
SSP	System stack pointer	16 bits
ST0_55–ST3_55	Status registers 0 through 3	16 bits
T0–T3	Temporary registers 0 to 3	16 bits
TRN0, TRN1	Transition registers 0 and 1	16 bits
XAR0–XAR7	Extended auxiliary registers 0 through 7	23 bits
XCDP	Extended coefficient data pointer	23 bits
XDP	Extended data page register	23 bits
XSP	Extended data stack pointer	23 bits
XSSP	Extended system stack pointer	23 bits

Accumulators (AC0–AC3)

- **The C55 contains four 40-bit accumulators:**
  AC0, AC1, AC2, and AC3 (The primary function of these registers is to assist in data computation in the D unit: ALU, MACs and the shifter.
- The four accumulators are equivalent:
  any instruction that uses an accumulator can be programmed to use any one of the four.
- Each accumulator is partitioned into:
  a low word (ACxL), a high word (ACxH), and eight guard bits (ACxG).
- Each of portion can be accessed individually:
  by using addressing modes that access the memory-mapped registers.

Temporary Registers (T0–T3)

Four 16-bit general-purpose temporary registers: T0–T3 can be used for:

- Hold one of the memory multiplicands for multiply, multiply-and-accumulate, and multiply-and-subtract instructions
- Hold the shift count used in addition, subtraction, and load instructions performed in the D unit
- Keep track of more pointer values by swapping the contents of the auxiliary registers (AR0–AR7) and the temporary registers (using a swap instruction)
- Hold the transition metric of a Viterbi butterfly for dual 16-bit operations performed in the D-unit ALU

Registers Used to Address Data Space and I/O Space

Auxiliary Registers (XAR0–XAR7 / AR0–AR7)

- The CPU includes eight extended auxiliary registers XAR0–XAR7
- Each high part (ARnH) is used to specify the 7-bit main data page for accesses to data space.
- Each low part (ARn) can be used as:
  - A 16-bit offset to the 7-bit main data page (to form a 23-bit address)
  - A bit address (in instructions that access individual bits or bit pairs)
  - A general-purpose register or counter

ARn and XARn Access

- **ARn Auxiliary register n and XARn Extended auxiliary register n are accessible via dedicated instructions .**

  ARn is mapped to memory

  XARn is not mapped to memory.

- ARnH high part of extended auxiliary register n is Not individually accessible.

  To access ARnH, you must access XARn.

- XAR0–XAR7 or AR0–AR7 are used in the AR indirect addressing mode and the dual AR indirect addressing mode.
- Basic arithmetical, logical and shift operations can be performed on AR0–AR7 in the A-unit arithmetic logic unit (ALU).

Coefficient Data Pointer (XCDP / CDP)

- **CDP is a coefficient data pointer, and CDPH an associated extension register, concatenate the two form the extended CDP that is called XCDP**
- CDPH is used to specify the 7-bit main data page for accesses to data space.
- The low 16 bits part (CDP) can be used as:
  - A 16-bit offset to the 7-bit main data page (to form a 23-bit address)
  - A bit address (in instructions that access individual bits or bit pairs)
  - A general-purpose register or counter

XCDP and CDP Accesses

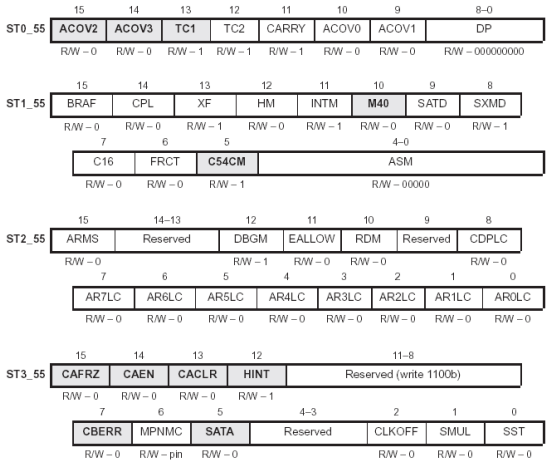
- **XCDP Extended coefficient data pointer is accessible via dedicated instructions only.**

  XCDP is not a register mapped to memory.
- CDP Coefficient data pointer is accessible via dedicated instructions and as a memory-mapped register
- CDPH High part of extended coefficient data pointer is accessible via dedicated instructions and as a memory-mapped register

Status Registers (ST0_55–ST3_55)

- The four 16-bit registers (ST0_55, ST1_55, ST2_55 and ST3_55) contain control bits and flag bits
- Control bits affect the operation of the C55x DSP
- Flag bits reflect the current status of the DSP or indicate the results of operations.
- ST0_55, ST1_55, and ST3_55 are each accessible at two addresses
  - At one address, all the TMS320C55x bits are available.
  - At the other address (the protected address), some of the bits cannot be modified.

Contents of Status Registers



CPU registers

- For more information concerning the C55xx registers refer to TI swpu073e.pdf (see toledo)

DSP C5000

Addressing Modes

Objectives

- Introduce linker command file
- Present the main addressing modes and allocation of sections
- Present the main addressing modes of the C55 family
- Explain how to use these addressing modes
- Do exercises to practice using the different addressing modes

How Do We Build a Project ?

Processing goal : z=x+y

code

get x
add y
store z
loop

constants

x = 2
y = 7

variables

z

```
        .text
Start
    rptblocal loop-1
    mov   *AR0+,AC0
     add   *AR1+,AC0
loop



        .data
x       .int    2
y       .int    7

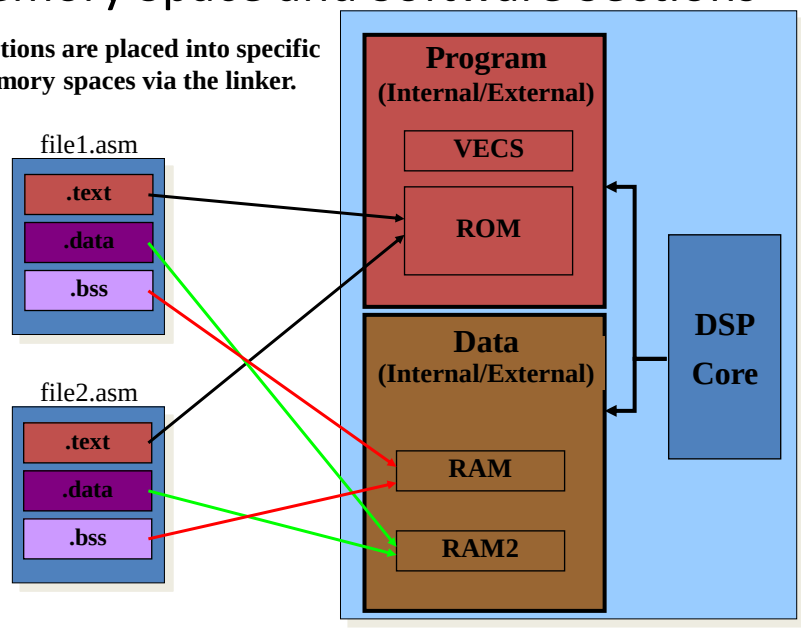
        .bss    z,1
```

Linker Command File

```
MEMORY
{
  RAM   (RWIX) : o = 0x000100,    l= 0x01feff /* Data Memory    */
  RAM2  (RWIX) : o = 0x040100,  l =  0x040000 /* Program Memory */
  ROM   (RIX)  : o = 0x020100,  l =  0x020000 /* Program Memory */
  VECS  (RIX)  : o = 0x0ffff0,  l = 0x000100  /* Reset Vector   */
}
SECTIONS
{
   vectors  > VECS    /* Interrupt vector table */
  .text     > ROM     /* CODE                   */
  .switch   > RAM     /* SWITCH TABLE INFO      */
  .const    > RAM     /* CONSTANT DATA          */
  .cinit    > RAM     /* INITIALIZATION TABLES  */
  .data     > RAM2    /* INITIALIZED DATA       */
  .bss      > RAM     /* GLOBAL & STATIC VARS   */
  .stack    > RAM     /* PRIMARY SYSTEM STACK   */
}
```

Memory Space and Software Sections

- **Sections are placed into specific memory spaces via the linker.**

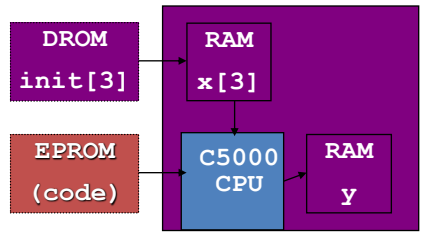


Allocating Sections

Example

Algorithm

y = x1 + x0 + x2



Procedure

- **Allocate sections (code, constants, vars)**
- **Setup addressing modes**
- **Add the values (x1 + x0 + x2)**
- **Store the result (y)**

How do we allocate the proper sections?

Writing relocatable code

- The programmer should not have to give the exact addresses:
  - where to read the code in program memory,
  - where to read the data in data memory.
- The assembler allows to use symbolic addresses.
- The assembler and the linker work with COFF files:
  - COFF = Common Object File Format.
  - In COFF files, specialized sections are used for code, variables or constants.
  - The programmer specifies in a command file for the linker where the different sections should be allocated in the memory of the system.

Definition of Sections

- Different sections for code, vars, constants.
- The sections can be initialized or not.
  - An initialized section is filled with code or constant values.
  - An uninitialized section reserves memory space for a variable.
- The sections can have default names or names given by the programmer.

Definition and names of Sections

- The programmer uses special directives to identify the sections.

	Initialized sections			Unitialized sections, reserve space for data
	code	Variables	Code or constants	
Named sections, name given by user			.sect	.usect
Unnamed sections, default name	.text	.data		.bss

Example of sections

```
    .sect ”init"
tbl .int 1,2,3
```

Initialized named section: Initialization of constants. Definition of address tbl

```
x   .usect "vars",3
y   .usect "result",1
```

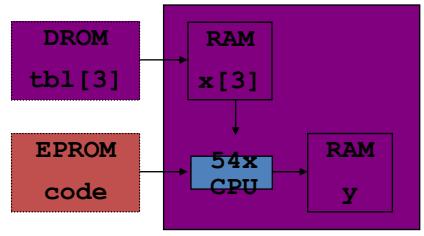
Uninitialized named section: x[3], y[1], Definition of address x and y.

```
    .sect “code”
```

Initialized named section: code

How are these sections placed into the memory areas shown?

System Diagram



Reference: Spru280i.pdf

C55x Addressing Modes

Format of Data and Instructions, Internal Busses for the C55x Family

- Unified program-data memory map: byte-aligned for program and word-aligned for data.
- Has a variable length instruction set (8-16-24-32-40-48 bits).
  - Program address bus: 24 bits, 16 Mbytes
  - 4 instructions bytes are fetched at a time
  - 6 bytes are decoded at a time
- Internal data busses: 3 data read, 2 data write
  - Data addresses: 8 Mwords of 16 bits segmented into 64K pages, 23-bit address. A 24-bit address is automatically generated by the hardware by adding a LSB = 0.

Addressing Modes: What are the Problems?

- Specify operands per instruction:
  - A single instruction can access several operands at a time thanks to the many internal data busses,
  - But how do we specify many addresses using a small number of bits?
- Repeated processing on an array of data:
  - Many DSP operations are repeated on an array of data stored at contiguous addresses in data memory.
  - There are cases where it is useful to be able to modify the addresses as part of the instruction (increment or decrement).

Main Addressing Modes of C5000 Family

- Immediate addressing
- Absolute addressing
- Direct addressing
- Indirect addressing by register
  - Support for circular indirect addressing
    - Definition
- Access to Memory Mapped Registers MMRs

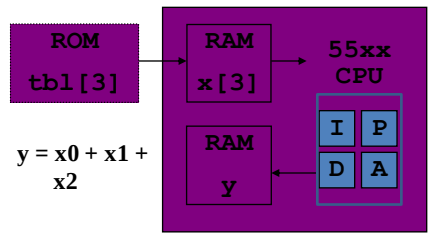
C55x Addressing Modes

Algorithm

$$y = x0 + x1 + x2$$

This algorithm will again be used as an example for the different addressing modes.

System Diagram



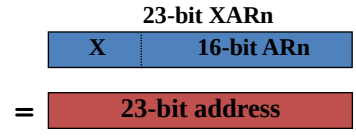
Loading Constants in Registers

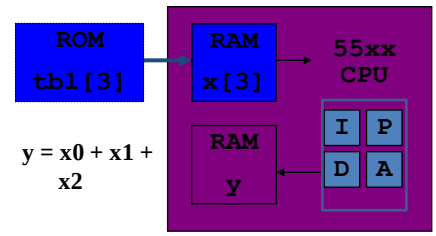
- Used for initialization of registers.
  - Used to be called immediate addressing
- Addressing registers:
  - 16-bits long: ARi, DP, CDP (Coefficient Data Pointer)
  - 23-bits long: XARi, XDP, XCDP
  - The 7 MSB of Xreg specify the 64K page.
- The ARAU (Auxiliary Register Arithmetic Unit) is 16 bits wide: update of ARi and CDP are done modulo 64K.
- Initialization example: `AMOV #adr,XAR3`

Example

```
x       .usect "vars",4
y       .usect "vars",1
        .sect  "init"
tbl     .int   1,2,3,4
        .sect  "code"
```

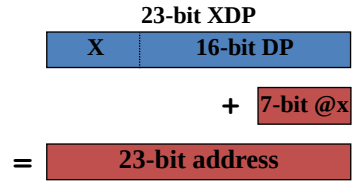
```
indir:  AMOV   #x,XAR0
        AMOV   #tbl,XAR6
```





Direct Addressing Mode @

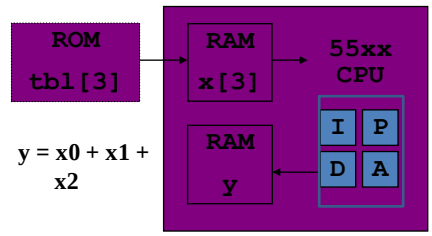
- Gives the instruction a positive 7bit offset from DP (non-aligned).
  - In the case where the bit CPL=0 in ST1.
  - Calculation in the ARAU modulo 64K



Example

```
x       .usect "vars",4
y       .usect "vars",1
        .sect  "init"
tbl     .int   1,2,3,4
        .sect  "code"
```

```
ADD:   MOV    @(x+0),AC0
       ADD    @(x+1),AC0
       ADD    @(x+2),AC0
```



How is XDP initialized?

Example

```
x       .usect "vars",4
y       .usect "vars",1
        .sect  "init"
tbl     .int   1,2,3,4
        .sect  "code"
```

A in AMOV means in AD-phase.

The XDP has to be reloaded every time we cross a 64K page.

```
dir:    AMOV   #x,XDP

ADD:    MOV    @(x+0-x),AC0
        ADD    @(x+1-x),AC0
        ADD    @(x+2-x),AC0
```

- **Constant value contained in instruction opcode**
- **`(-x)` used in instruction to tell the assembler HOW to create the 7-bit offset from non-aligned XDP**

Directive .dp for Direct Addressing

- Instead of using (-x) to help the assembler calculate the proper 7-bit offset, we can use the directive .dp to set the base address for the assembler calculation of the 7-bit offset.
  - .dp base_address

```
        .dp x
dir:    AMOV   #x,XDP

ADD:    MOV    @(x+0),AC0
        ADD    @(x+1),AC0
        ADD    @(x+2),AC0
```

The @addr in the instruction is interpreted as a 23-bit address.

The .dp provides a compile-time base address.

The assembler determines the 7-bit offset by: (@addr-.dp_value)&7F

Indirect Addressing Mode *ARi

- AR indirect
- Dual AR indirect
- CDP indirect
- Coefficient indirect

Indirect Addressing: AR indirect

- Uses one of the auxiliary registers (AR0-AR7) to point to data.
- Address generation depends on whether one is accessing data space (memory or MMR), register bits, or I/O space.

Indirect Addressing: AR indirect

– Accessing Data Space (memory or registers):

23-bit address = ARnH: ARn

where ARnH is the high part of XARn = ARnH: ARn and supplies the 7 most significant bits, and ARn supplies the 16 least significant bits

- Example: MOV *AR0, T2

23-bit address = AR0H: AR0 = XAR0

The CPU reads the value at address XAR0 and loads it into T2.

Indirect Addressing: AR indirect

- Accessing a register bit:
  - The selected 16-bit ARn register contains a bit number
  - Example for AR2 = 30:

BSET *AR2, AC2

The CPU sets bit 30 of AC2.

Indirect Addressing: AR indirect

- Accessing I/O space:
  - The selected 16-bit ARn register contains the complete 16-bit I/O address.
  - Example for AR2 = 0080h:

    MOV port(*AR2), T2

    The CPU reads the value at I/O address 0080h and loads it in T2.

Indirect Addressing: Dual AR indirect

- Used to make two data-memory accesses through AR0-AR7.
- As in AR indirect, extended registers XARn used to generate each 23-bit address.
- Example 1: Two data-memory accesses

  ADD *AR0, *AR1, AC0
- Example 2: Two instructions in parallel

  MOV *AR1, T2

  || AND *AR2, T1, AC0

Indirect Addressing: CDP indirect

- Uses the CDP (Coefficient Data Pointer) to point to data.
- The generation of 23-bit address depends on the access type: data space, register bit, or I/O space.

Indirect Addressing: CDP indirect

– Data Space Access (memory or registers):

23-bit address = CDPH: CDP

where CDPH is the high part of the extended coefficient data pointer (XCDP = CDPH:CDP) and supplies the 7 MSB of 23-bit address, and CDP supplied the 16 LSB of 23-bit address.

- Example 1: MOV *CDP, T1
  23-bit address = CDPH:CDP = XCDP
- Example 2: MOV *CDP+, T2
  23-bit address = CDPH:CDP = XCDP

Indirect Addressing: CDP indirect

- Register Bits:
  - CDP is used to access a register bit
  - CDP contains the bit number
  - Example: BSET *CDP, AC0

Indirect Addressing: CDP indirect

- I/O Space:
  - The CDP contains the complete 16-bit I/O address.
  - Example: MOV port(*CDP), T2

Indirect Addressing: Coefficient indirect

- Same address generation as CDP indirect.
- This mode is mainly used with instructions performing operations on three memory operands per cycle.
  - Two of the operands are accessed using the dual AR indirect addressing mode
  - Third operand is accessed using the coefficient indirect mode

Indirect Addressing: Coefficient indirect

- Example:

MPY *AR1, *CDP, AC0

:: MPY *AR2, *CDP, AC1

The values pointed to by AR1, AR2 and CDP are accessed in one cycle.

Indirect Addressing: pointer operations

- Both pointer modification and address generation are linear or circular according to the pointer (register) configuration in status register ST2_55.
- All additions to and subtractions from the pointers are done modulo 64K. One cannot address data across main data pages without changing the value of ARnH of XARn.

Indirect Addressing Options for Pointer ARi Modifications

`*ARn`	No Modify
`*ARn(T0/1)`	No Modify w/offset
`*ARn(#k16)`	No Modify w/offset
`*ARn +/-`	Post Modify (+/-)
`*(ARn +/- T0/1)`	Post Modify (+/- by T0/1)
`*+/- ARn`	(+/-) Pre Modify
`*+ ARn(#k16)`	(+ #k16) Pre Modify
`*(ARn +/- T0B)`	Bit reversed using T0
`*CDP`	No Modify
`*CDP(#k16)`	No Modify w/offset
`*CDP +/-`	Post Modify (+/-)
`*+CDP(#k16)`	(+ #k16) Pre Modify

- Assumes $ST2_55_{ARMS}=0$ and $ST1_55_{C54CM}=0$.
- The reset condition is C54CM=1.

Modifying TAs Registers

- TAx registers = T0-3, AR0-7.
- Special instructions:
  - AADD, ASUB, AMOV
  - Can be used to modify TAs registers during the address (AD) phase of the pipeline, while instructions without A operates during the execution (X) phase.
  - They only work on the TAx registers.

Examples:

```
AADD #const,AR1
ASUB AR1,T0
AMOV #k23,XAR2
```

Example

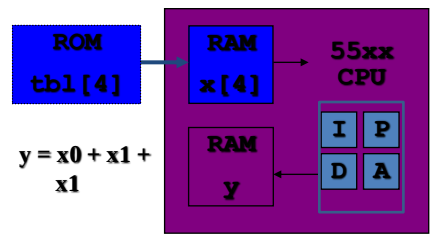
```
x           .usect "vars",4
y           .usect "vars",1
            .sect  "init"
tbl         .int   1,2,3,4
            .sect  "code"

            .dp x
indir:      AMOV   #x,XAR0
            AMOV   #tbl,XAR6
```

```
COPY:       MOV    *AR6+,*AR0+
            MOV    *AR6+,*AR0+
            MOV    *AR6 ,*AR0
```

```
dir:        AMOV   #x,XDP

ADD:        MOV    @(x+0),AC0
            ADD    @(x+1),AC0
            ADD    @(x+2),AC0
```

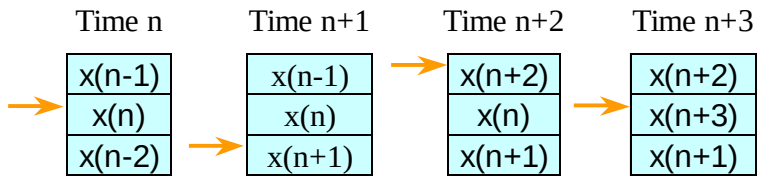


Circular buffer and circular addressing

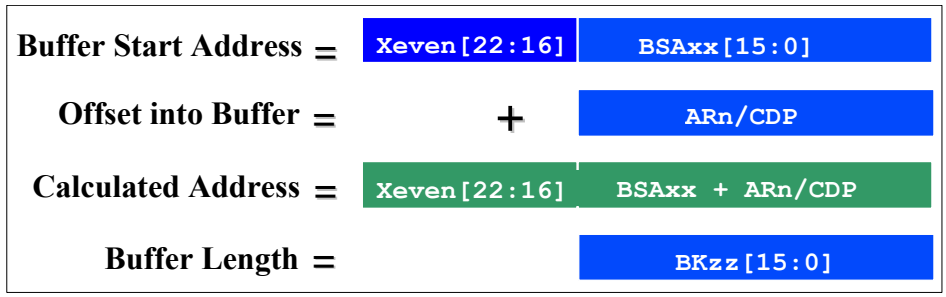
- A circular buffer of length N is a block of contiguous memory words addressed by a pointer using a modulo N addressing mode.
  - The 2 extreme words of the memory block are considered as contiguous.
- Characteristics of a circular buffer:
  - Instead of moving the N data in memory, just modify the pointers.
  - When a new data x(n) arrives, the pointer is incremented and the new data is written in place of the oldest one.

Trace of Memory and Pointer in a Circular Buffer of Length 3

- Very often used for FIR filters.



Circular Buffer Addressing Mode



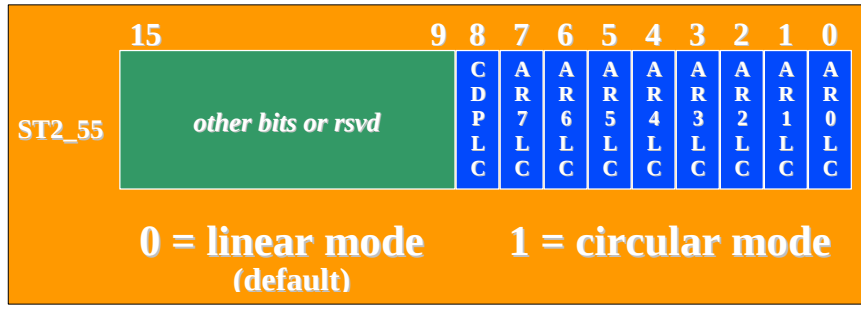
Circular Buffer Addressing Mode

Offset	Xeven	Buffer Start Address	Block size Register
AR0 AR1	XAR0[22:16]	BSA01	BK03
AR2 AR3	XAR2[22:16]	BSA23	
AR4 AR5	XAR4[22:16]	BSA45	BK47
AR6 AR7	XAR6[22:16]	BSA67	
CPD	XCDP[22:16]	BSAC	BKC

The even XARn (i.e. 0,2,4,6) determines the 64K Page

Selecting Circular or Linear Addressing Mode

- Use the LSB of Status word ST2_55



- **Set or reset status bits:**

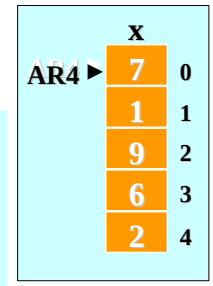
```
BSET AR5LC    ;AR5 in circular mode
BCLR AR3LC    ;AR3 in linear mode
```

Circular Buffer Exercise

Use AR4 as a circular pointer to x{5}:

```
    .sect "data"
x   .int 7,1,9,6,2          ;init data
    .sect "code"
    AMOV #x,XAR4            ;init XAR
    MOV  #x,BSA45           ;init start addr
    MOV  #5,BK47            ;init length
    MOV  #0,AR4             ;init AR4 to top
    BSET AR4LC              ;set AR4 to circ

    MOV #3,T0              ;index
    MOV *(AR4+T0),AC0      ;AC0 =_7__, AR4 =_3__
    MOV *+AR4(#4h),AC1     ;AC1 =_9__, AR4 =_2__
    MOV *AR4(T0),AC2       ;AC2 =_7__, AR4 =_2__
```



Results are cumulative

C55x circular addressing modes

- 3 BK registers in C55X, allows for several simultaneous circular buffers with different size.
- In C55x, the mode in set in status register ST2_55 for each register (linear or circular). No memory alignment constraint.

Absolute Addressing *(#)

- *(#) = 23 bit address
- Fast: no initialization,
- But long instruction because it contains the 23 bit address.
- If the address is in the 64K work page, it is possible to specify a 16-bit only address:
  - abs16*(#label)

Example

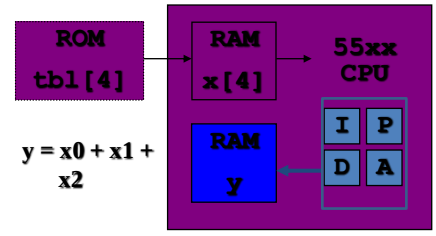
```
X       .usect "vars",4
Y       .usect "vars",1
        .sect  "init"
tbl.int   1,2,3,4
        .sect  "code"

        .dp x
indir: AMOV  #x,XAR0
       AMOV  #tbl,XAR6

COPY:  MOV   *AR6+,*AR0+
       MOV   *AR6+,*AR0+
       MOV   *AR6 ,*AR0
dir:   AMOV  #x,XDP

ADD:   MOV   @(x+0),AC0
       ADD   @(x+1),AC0
       ADD   @(x+2),AC0
```

```
STORE: MOV   AC0,*(#y)
```



MMR Addressing Using mmap()

- MMRs are located between 0 and 5F.
- Scratch memory is located between 60 and 7F.
- mmap() forces bits 22:7 to zero.
  - Useful to access MMR and scratch memory without initialization of addressing registers.
- Useful only for direct addressing.

```
; write  #1234h to ST0_55
AMOV   #0,XDP
MOV    #1234h, @(ST0_55 - 0)
```

```
; write  #1234h to ST0_55
MOV    #1234h, mmap(@ST0_55)
```

Access Peripheral Registers

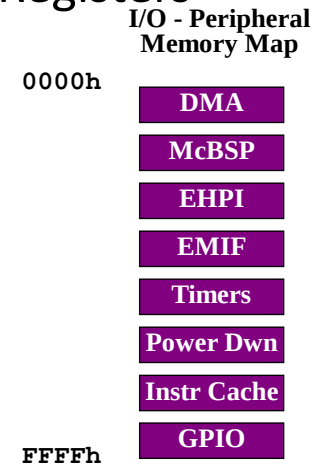
- The I/O space is internal.
- The PDP (Peripheral Data Pointer) register is used to access ports using direct addressing.
  - It is a 9bit register. Its value is concatenated with the 7 bits in the instruction to obtain a full 16-bit peripheral address.
- The port() modifier selects the peripheral map

Access Peripheral Registers

```
abs:    MOV  port(#addr),T1

dir:    MOV  #addr,PDP
        MOV  T1,port(@addr)

indir:  AMOV #addr,AR4
        MOV  port(*AR4),T1
```



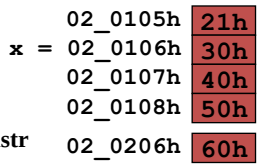
Modifying Status Bits

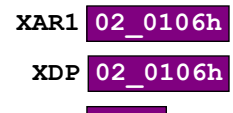
```
BSET/BCLR  bit_name
```

```
BSET CPL        ;set CPL
```

Addressing Exercise

- The initial state of each instruction is shown here...
- Below, write down the state after each instr





.dp x

	AR1	AC0	T1	02_0106h	$ST1_{M40}$
MOV @(x+1),AC0 MOV @(x+80h),AC0					
MOV T0,*AR1+ MOV *(#x),AC0					
MOV #4,@(x+128) MOV *(AR1+T0),T1					
BSET M40 MOV @(x+2),AC0					
MOV *AR1(T0),AC0 MOV *AR1(#100h),T1					
MOV @(x+129),AR1 MOV *+AR1(#-1),AC0					

Addressing Exercise – Solution

- The initial state of each instruction is shown here...
- Below, write down the state after each instr

x =

02_0105h	21h
02_0106h	30h
02_0107h	40h
02_0108h	50h
02_0206h	60h

XAR1	02_0106h
XDP	02_0106h
T0	2

.dp x

	AR1	AC0	T1	02_0106h	$ST1_{M40}$
MOV @(x+1),AC0		40h			
MOV @(x+80h),AC0		30h			
MOV T0,*AR1+	107h			2	
MOV *(#x),AC0		30h			
MOV #4,@(x+128)				4	
MOV *(AR1+T0),T1	108h		30h		
BSET M40					1
MOV @(x+2),AC0		50h			
MOV *AR1(T0),AC0	106h	50h			
MOV *AR1(#100h),T1	106h		60h		
MOV @(x+129),AR1	40h				
MOV *+AR1(#-1),AC0	105h	21h			

DSP C5000

Numerical issues

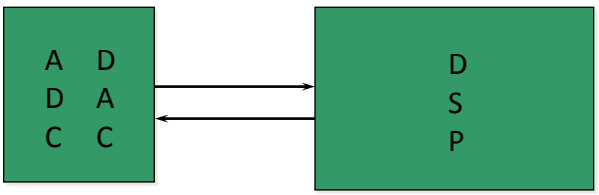
Learning Objectives

- Data formats
- Fixed point: integer and fractional numbers
- Use methods for handling multiplicative and accumulative overflow
- Floating point
- Block floating point
- Comparison of formats

Data Formats and Numerical Issues

- Common data sizes: 8, 16, 24, 32 bits
- Fixed or floating point
- For a given technology:
  - Fixed point is faster and less expensive
  - But fixed point programming is more difficult
- Processors of the ‘C5000 family are fixed point processors.
  - But they can also execute floating point operations through software

Interface ADC - DSP - DAC



Possible Conversions:

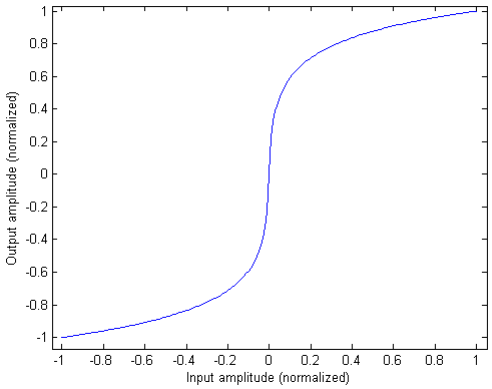
fixed point ⟷ floating point

A or mu law ⟷ linear law

(Compression-Expansion)

intermezzo μ-law

- compresses large amplitudes in a manner loosely corresponding to human loudness perception



Binary Representation of Signed Integers
used in ADC-DAC or DSP
in Fixed Point Format

- 2's Complement (digital processors)
- 1's Complement
- Sign, magnitude
- Offset Binary

Fixed Point Arithmetic

2's Complement Representation

Example of Size 3 bits for Integers, Decimal and Binary Representations

Positive integers	Positive integers	Signed integers	Signed integers	Signed integers	Signed integers
decimal	Binary	Decimal	Offset binary	Decimal	Sign + magnitude
7	1 1 1	3	1 1 1	3	0 1 1
6	1 1 0	2	1 1 0	2	0 1 0
5	1 0 1	1	1 0 1	1	0 0 1
4	1 0 0	0	1 0 0	0	0 0 0
3	0 1 1	-1	0 1 1	0	1 0 0
2	0 1 0	-2	0 1 0	-1	1 0 1
1	0 0 1	-3	0 0 1	-2	1 1 0
0	0 0 0	-4	0 0 0	-3	1 1 1
	Weights 2^2 2^1 2^0				

Example of Size 3 bits for Integers, Decimal and Binary Representations

Signed integers Decimal	Signed integers 1's complement	Signed integers 2's complement
3	0 1 1	0 1 1
2	0 1 0	0 1 0
1	0 0 1	0 0 1
0	0 0 0 or 1 1 1	0 0 0
-1	1 1 0	1 1 1
-2	1 0 1	1 1 0
-3	1 0 0	1 0 1
-4		1 0 0

Representation of Signed Integers in 2's Complement Format

$$(x)_{10} \rightarrow b_{N-1} \cdots b_k \cdots b_0$$

$$(x)_{10} \geq 0 \Rightarrow x = \sum_{k=0}^{N-1} b_k 2^{+k}$$

$$(x)_{10} < 0 \Rightarrow y = -2^N + (b_{N-1} \ldots b_k \ldots b_0)_{10}$$

$$(x)_{10} = -2^{N-1} b_{N-1} + \sum_{k=0}^{N-2} b_k 2^{+k}$$

Non-Integer Numbers Using Fixed Point

- Format Qk : k fractional bits associated with negative power of 2.
- The binary representation of a number x in format Qk is the 2's complement representation of the integer y:

$$y = 2^k x$$

$$\overbrace{b_{N-1-k} \cdots b_1 b_0}^{\text{Integer Part}} , \overbrace{b_{-1} \cdots b_{-k}}^{\text{Fractional Part}}$$

$$x = -b_{N-1-k} 2^{N-1-k} + b_{N-2-k} 2^{N-2-k} + \cdots b_0 + b_{-1} 2^{-1} + \cdots b_{-k} 2^{-k}$$

Example of 2's Complement Binary Representations

- Represent x = 1.75 using N=6 bits in format Q3
  - Answer 001.110 = 1 +1/2 +1/4
- Represent x = -1.75 using N=6 bits in format Q3
  - Answer 110.0 10 = - 4 +2+1/4
- Represent x = 1. 875 using N=6 bits in format Q3
  - Answer 001.111 = 1 + 1/2 + 1/4 + 1/8

Some Properties of 2's Complement Representation

Max number=$2^{N-1}-1$

Min number=-2^{N-1}

Circular Representation: (SATD)

$(2^{N-1}-1)+1=2^{N-1} \Leftrightarrow -2^{N-1}$

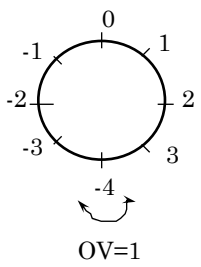
Sign bit Extension: (SXMD)

SATD = SATuration mode of the D unit on C55 DSPs

SXMD = Sign eXtension Mode of the D unit on C55 DSPs

Addition/Subtraction (1)

- Simple hardware operator when using 2's Complement: to add 2 signed N-bit integers with a result of size N bits. Whatever the sign of numbers, it is sufficient to add the 2's complement values.



Carry

111	010	110	110
+ 111	+ 001	+ 011	+ 001
1 110	0 011	1 001	0 111

Overflow (intermediate)

Addition/subtraction (2)

Addition and subtraction are not a problem when the data are the same format. Check for saturation

Example format [4, 2]:

```
  01.10  (1.50, format [4, 2])
+ 01.01  (1.25, format [4, 2])
 ------
 010.11  (2.75, format [5, 2])
```

If we had kept a [4, 2] for the result would be catastrophic: we get 10.11, that is to say -1.25

Addition/subtraction (3)

When the formats differ, the decimal point needs to be aligned.

Example : 5 + (– 0.875) = 4.125

```
  0101.      (5, format [4, 0])
+    1.001   (-0.875, format [4, 3])
  --------
  ????????
```

You can not add different formats. It extends with zeros to the right and with the sign bit zeros on the left.
If we want to avoid saturation, we still provides a bit of reserve left. It repeats the same operation adapted:

```
  0 0101.000  (5, format [8, 3])
+ 1111 1.001  (-0.875, format [8, 3])
 ---------
  00100.001  (4.125, format [8, 3])
```

Sign eXtension Mode SXMD

- With 2's complement, when 16-bit data are loaded into a 32-bit accumulator, the sign bit is also extended.
- This sign extension may be annoying: e.g. Calculation of 16-bit addresses.
- The user can choose whether or not to use sign bit extension mode.

  SXMD = Sign eXtension Mode bit for the D unit in the status word ST1_55 in C55 DSPs

Sign Bit Extension

- Example data size 6 bits, Accumulator size 12 bits

Data	1	0	1	0	0	1

Loading of ACx with sign extension

1	1	1	1	1	1	1	0	1	0	0	1

Loading of ACx without sign extension

0	0	0	0	0	0	1	0	1	0	0	1

Addition/subtraction (4)

Or the addition (or subtraction) in the fixed point: z = x + y
If you do not want "losing" to the left or right of the point, then we must apply the following rule for the size of z.

```
NbBitsFracZ = max(NbBitsFracX, NbBitsFracY)
NbBitsLeftZ = max(NbBitsLeftX, NbBitsLeftY) + 1
NbBitsTotZ  = NbBitsLeftZ + NbBitsFracZ
```

If we take the previous example:

```
NbBitsFracZ = max(0, 3) = 3
NbBitsLeftZ = max(4, 1) + 1 = 5
NbBitsTotZ  = 3 + 5 = 8
formatZ     = [8, 3]
```

In practice, we can not always “keep everything" because the results become too large in terms of bus.

Addition Overflow

- When adding 2 numbers of size N bits, the result may need N+1 bits.
- Example for integers of N=3 bits:
  - 3+3 = 6 cannot be represented using 3 bits (using 2’s complement), but can be expressed using 4 bits.
  - In format Q2 of N=3 bits, 0.75 + 0.5 =1.25 cannot be represented using 3 bits, needs 4 bits.
- When adding M numbers of N bits, the result potentially needs N+ log2(M) bits.

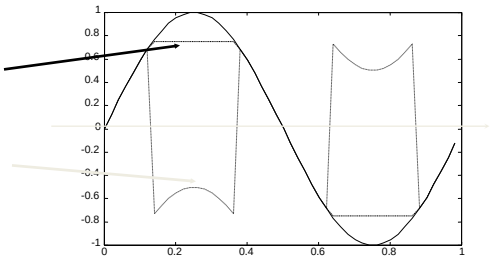
Using Saturation

- Overflows in 2's complement create unexpected sign changes and peaks that are difficult to filter.
- Saturation arithmetic detects the overflow and replaces the result with a saturation value.

Example, max value = 0.75

Saturation at 0.75

2's complement overflow



Setting saturation modes with SATD

- The user can choose whether or not to use saturation mode by setting the corresponding mode bits.
  - SATD = SATuration mode bit for the D unit in the status word ST1_55 in C55 DSPs.
    - If SATD = 1 and M40 =0,
      - positive results are saturated to 00 7FFF FFFF
      - Negative results are saturated to FF 8000 0000
    - If SATD=1 and M40 =1
      - positive results are saturated to 7F FFFF FFFF
      - Negative results are saturated to 80 0000 0000

7F FFFF FFFF: 0111 1111 1111111111111111 1111111111111111
80 00000000: 1000 0000 0000000000000000 0000000000000000

Saturation mode for the A unit in C55 DSPs

- SATA = SATuration mode bit for the A-unit ALU in the status word ST3_55 in C55 DSPs.
  - If SATA=1, if a calculation in the A-unit results in an overflow:
    - positive results are saturated to 7FFF
    - Negative results are saturated to 8000.

Effect of 2's Complement Overflow

- As 2's complement is a circular representation, if the result holds on N bits, the intermediate overflows do not alter the final result
- This is not the case for saturation
- Example of N = 3 bits:
  - Calculate x = 3+2-4, the theoretical result is 1
  - With 2's complement overflow:
    - Calculate first y=(3+2)= 011+010 =101 =-3 overflow
    - Then (y-4)=101+100=1 001 = 1 and carry =1 correct result
  - With saturation:
    - Calculate first y=(3+2)=3 saturation
    - Then (y-4) = 011+100=111=-1 wrong result
- If a system has a unity gain, saturation should not be used.

Multiplication (1)

As for multiplication that was learned in elementary school, we remove the point operands and return the result

Example : 5·(–0.875) = 5·(–875)/1000 = –4375/1000 = –4.375

```
   1001        (1.001 ⇒ -0.875, format [4, 3])
 x 0101       (0101. ⇒ 5, format [4, 0])
 ----
 1111001
 0000000
 1100100
 0000000
 --------
 1011101       (1011.101 ⇒ -4.375, format [7, 3])
```

Multiplication (2)

We see that multiplication is an operation that requires a significant amount of resources. Complexity is directly related to the number of bits of the operands.

With x in [4, 0] and y [4, 3], the format z [7, 3] works for all combinations of x and y ... except one!

```
    1000      (1.000 ⇒ -1, format [4, 3])
  x 1000      (1000. ⇒ -8, format [4, 0])
--------
01000000      (01000.000 ⇒ 8, format [8, 3])
```

When the operands x and y each have their most extreme negative value (10 .. 0), then it takes a bit more! This is an extreme case, it must nevertheless be taken into account if it can happen ...). → SMUL bit

Multiplication (3)

Is a multiplication z = x + y fixed point. Two things:

- Let x and y are both unknown and can take each value xmin and ymin. In this case, we must consider the worst case:

```
NbBitsTotZ  = NbBitsTotX + NbBitsTotY
NbBitsFracZ = NbBitsFracX + NbBitsFracY
```

  Previous example: z = 5·(–875) : format_z = [4+4, 0+3] = [8, 3]

- Either one of x or y values is known and different from its minimum value. And then x and y are unknown, but we know that one of these two values can not take its minimum value. In this case, we gain a bit since the worst case can not occur:

```
NbBitsTotZ  = NbBitsTotX + NbBitsTotY - 1
NbBitsFracZ = NbBitsFracX + NbBitsFracY
```

  Previous example: z = 5·(–875) : format_z = [4+4-1, 0+3] = [7, 3]

FRCT bit

- Consider the case of data < 1 using N=16 bits, Q15 format.
- Their products are < 1 and can be expressed using 32 bits format Q30 with 2 sign bits
- It is possible with the C5000 DSP to automatically eliminate one sign bit by a left shift of 1 bit, thus obtaining a Q31 result.
  - If bit FRCT in ST1 is set to 1, products are automatically shifted left by 1 bit.

SMUL bit

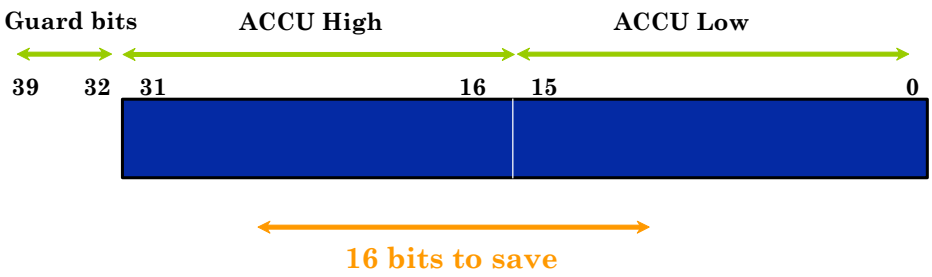
- The exception –1 x –1 can be treated using the SMUL status bit that saturate the result of the multiplication before accumulation. –1 is equal to 8000 in hexadecimal on 16 bits.
  - If SMUL=1, SATD, FRCT =1
    - The product of (-1)8000 x (-1)8000 is saturated to the positive number 7FFF FFFF after the multiplication and before accumulation in MAC or MAS instructions.
    - Consistent with ETSI-GSM specifications.

Accumulator and size of the result

- The final result of a calculation usually uses more than 16 bits (size of memory words).
- ACCUs use 32, 40, 56 ... Bits
- If we want to save the result in a single memory word, the question is:
- Which pack of N bits must be saved from accumulator?
  - Possibility of overflow and underflow
  - Overflow during accumulation or during saving.

ACCUMULATOR

- Possibility of overflow and underflow
- Scaling when adding many products

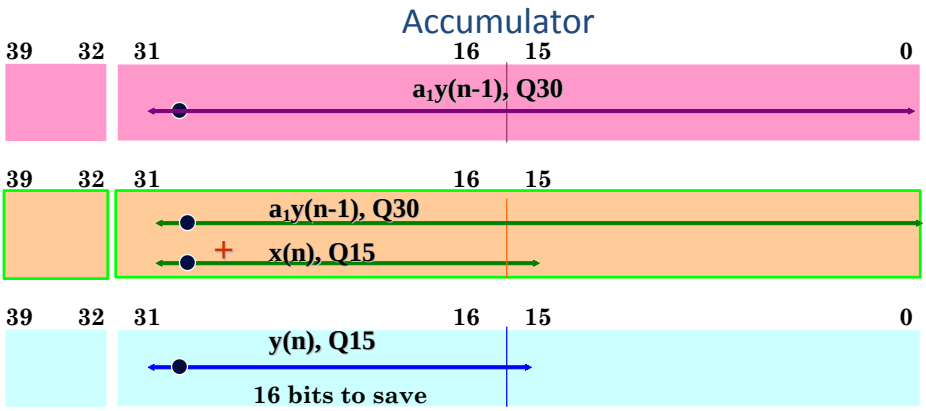


Saturation on store mode, SST bit

- SST = mode bit in ST3_55 status word.
- If SST is set, the CPU saturates a shifted or unshifted accumulator value before storing it.
- The saturation value depends on the value of the sign extension mode bit.
- ACCU remains unchanged.

Ex. of Fixed Point Processing $y(n)=x(n)+a_1y(n-1)$

- Data size N=16, product size 32 bits, accumulator size 40 bits
- The coefficient a_1 is smaller than 1: format Q15.



Solutions to Overflow

- Overflow multiplication can be prevented by using pure fractional numbers (< 1)
- Saturation of the result
- Scaling of the inputs and use of fractional arithmetic
  - But loss of precision
- Use double precision or double word
  - But decreases speed of calculation
- Use DSP with larger accumulators.
  - 8 guard bits in the C5000 accumulators.
- Design system with unity gain.
- Use floating point

Fixed Point Programming

- Perpetual compromise between dynamic range and precision constraints
- Keep enough bits to represent the integer part of the result
- Keep enough bits in the fractional part to satisfy the precision.
- Rounding results.

Remark: entering Non-Integer Values using the Software Development Tools

- The tools do not support fractions
  - To store 0.707 in Q15 use:
    - .word 32768*707/1000
  - To store 3.252 in Q13 use:
    - .word 8192*3252/1000
- Generally, to convert a real number x using 2's complement representation with size N bits and format Qk:
  - Calculate the integer y=round($(x)_2\ 2^k$)
  - The 2's comp. representation of y is the 2's comp. representation of x in format Qk.

Rounding

$ 1.53
$ 0.50
$ 2.03
$ 2.

How do you round this amount to the nearest $?
- Add $0.50
- Partial result
- Truncate result (to nearest $)

- Instruction ROUND, rounds the content of the accumulator ($ST2_55_{RDM} = 0$)
  - Direct: `ROUND AC0`
  - Store: `MOV uns(rnd(HI(saturate(AC0)))),*AR1`
    - `rnd()` and `ROUND` perform the following operation: (add 1 to bit 15) and (truncate)
      `(ACx+0x8000) & 0xFFFF0000`

Quantization noise

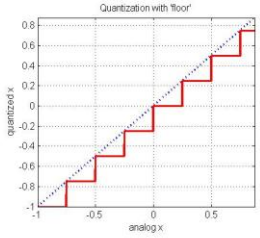
When quantifying an analog value, it creates an error similar to the noise: it is called quantization noise.
An analog value x from -1 to 1 with 3-bit quantization, so 8 levels. x_q is the quantized value. The quantization noise is defined as:

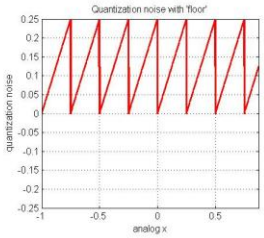
$$n_q = x - x_q$$

The next figure gives an example.

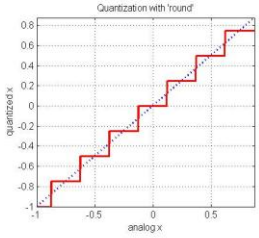
Quantization noise (1)

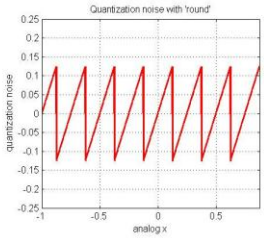
FLOOR





ROUND



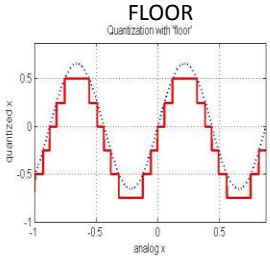


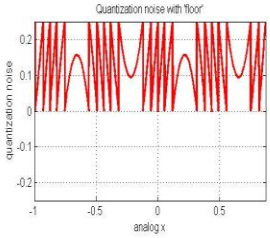
Example : signal ramp.

We note that the mode floor 'generates a quantization noise that has a DC component.

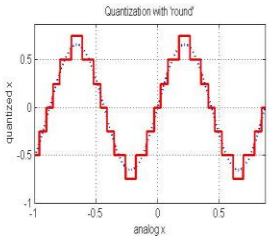
Quantization noise (2)

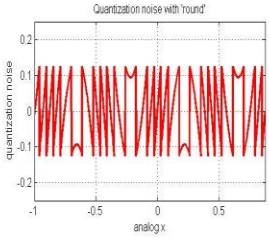
FLOOR





ROUND





Exemple : sinus.

Ici encore, on constate que le mode ‘floor’ génère un bruit de quantification qui possède une composante continue.

Quantization noise (3)

Let Δ be the quantization step, that is to say the interval between two quantized values. Is still X_{tot} signal dynamics and n the number of bits. We have: $\Delta = \frac{X_{tot}}{2^n}$

- Using 'floor'
  - DC: $N_{DC} = E\left[n_q\right] = \frac{1}{N} \cdot \sum_{i=1}^{N} n_q(i) = \frac{\Delta}{2}$
  - Power: $P_{nq} = E\left[{n_q}^2\right] = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(n_q(i)\right)^2 = \frac{\Delta^2}{3}$
- Using 'round'
  - DC: $N_{DC} = E\left[n_q\right] = 0$
  - Power: $P_{nq} = E\left[{n_q}^2\right] = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(n_q(i)\right)^2 = \frac{\Delta^2}{12}$

Signal to noise ratio (1)

Signal Power of x : $P_x = E\left[x^2\right] = \frac{1}{N} \cdot \sum_{i=1}^{N} \left(x(i)\right)^2$

Signal Over Noise Ratio (SNR) :

$$SNR = 10 \cdot \log\left(\frac{P_x}{P_{nq}}\right)$$

Considerations of mode 'round' :

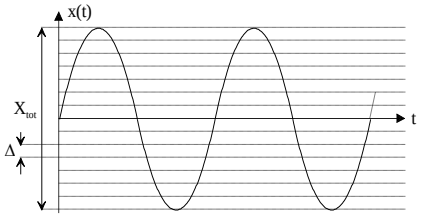
$$SNR = 10 \cdot \log\left(12 \cdot \frac{P_x}{\Delta^2}\right) = \overbrace{10 \cdot \log(12)}^{10.8} + 10 \cdot \log(P_x) - 20 \cdot \log(\Delta)$$

$$SNR = 10.8 + 10 \cdot \log(P_x) - 20 \cdot \log\left(\frac{X_{tot}}{2^n}\right) = \overbrace{10.8 + 10 \cdot \log(P_x) - 20 \cdot \log\left(X_{tot}\right)}^{K} + n \cdot \overbrace{20 \cdot \log(2)}^{6.02}$$

$SNR = K + 6.02 \cdot n$ (K dépend du signal)

Signal to noise ratio (2)

There is the very important relationship: when one removes a bit of resolution, **it lowers the SNR with 6 dB**.



It is shown that when the signal x is a sinus that covers the whole dynamic range? (of $-x_{max}/2\ x_{max}/2$, see drawing), P_x is the power x_{max}^2 / 8 and K = 1.77. We establish the following table

n	**8 bits**	**10 bits**	**12 bits**	**16 bits**
SNR	**50 dB**	**62 dB**	**74 dB**	**98 dB**

Spectral representation of the quantization noise(1)

The quantization noise is a signal power over. Tool spectral representation of such signals is the power spectral density (English: Power Spectrum Density or PSD).

A discrete system is fully defined in the range

with f_s : sampling frequency

$$-\frac{f_s}{2} \le f < \frac{f_s}{2}$$

- **Mode 'round'**

  The quantization noise is white, so it is evenly distributed in -fs / 2 to fs / 2.

- **Mode 'floor'**

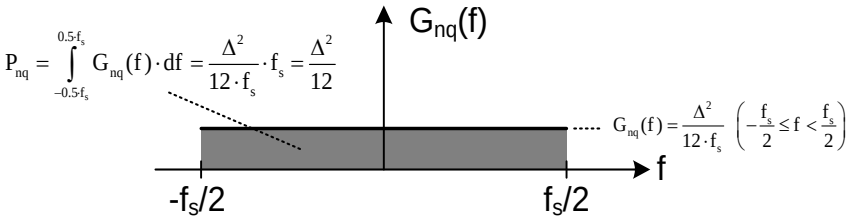
  The quantization noise is the same way 'round', except there is a ray of Dirac f = 0, representing the power of the DC component.

Spectral representation of the quantization noise, mode round

‘round’

PSD : $$G_{nq}(f) = \frac{\Delta^2}{12 \cdot f_s} \quad \left(-\frac{f_s}{2} \le f < \frac{f_s}{2} \right)$$

Noise power : $$P_{nq} = \int_{-0.5 \cdot f_s}^{0.5 \cdot f_s} G_{nq}(f) \cdot df = \frac{\Delta^2}{12 \cdot f_s} \cdot f_s = \frac{\Delta^2}{12}$$

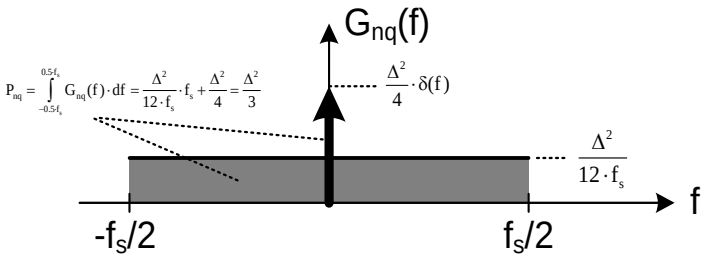


Spectral representation of the quantization noise, mode floor

‘floor’

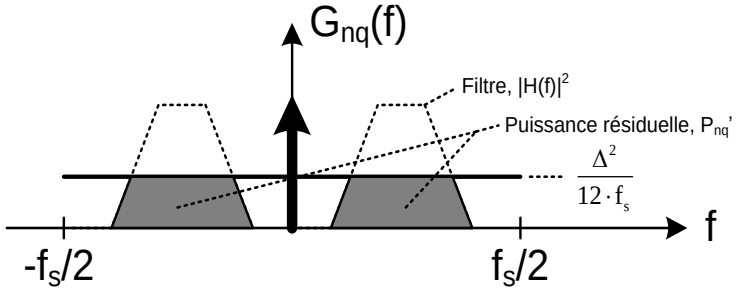
PSD : $$G_{nq}(f) = \frac{\Delta^2}{12 \cdot f_s} + \frac{\Delta^2}{4} \cdot \delta(f) \quad \left(-\frac{f_s}{2} \le f < \frac{f_s}{2} \right)$$

Noise Power : $$P_{nq} = \int_{-0.5 \cdot f_s}^{0.5 \cdot f_s} G_{nq}(f) \cdot df = \frac{\Delta^2}{12 \cdot f_s} \cdot f_s + \frac{\Delta^2}{4} = \frac{\Delta^2}{3}$$



Filtered quantization noise (1)

If quantization noise is filtered (transfer function H (f)), it is no longer white. We have the following situation:



Residual power:

$$P_{nq}' = \int_{-0.5 \cdot f_s}^{0.5 \cdot f_s} G_{nq}(f) \cdot \left|\underline{H}(f)\right|^2 \cdot df$$

Filtered quantization noise (2)

In a digital filter, the quantizing noise sources are located within the filter itself. To determine their impact on output, it is imperative to know the transfer function H (f) from the source to the output.

If the noise is white (no DC component), the previous equation can be simplified. Let h (n) the impulse response of the digital filter. It shows the relationship following practice: foo

$$\frac{1}{f_s} \cdot \int_{-0.5 \cdot f_s}^{0.5 \cdot f_s} \left|\underline{H}(f)\right|^2 \cdot df = \sum_{n=0}^{\infty} h^2(n)$$

As consequence:

$$\frac{1}{f_s} \cdot \int_{-0.5 \cdot f_s}^{0.5 \cdot f_s} \left|\underline{H}(f)\right|^2 \cdot df = \sum_{n=0}^{\infty} h^2(n)$$

Other Useful Stuff...

Absolute Value	`ABS AC0,AC1`
2's Complement	`NEG AC0,AC1`
1's Complement	`NOT AC0,AC1`
1-bit division	`SUBC Smem,ACx`
Normalization	`MANT; EXP`

Floating Point Arithmetic

Floating Point Representation

- Number x -> Mantissa M and Exponent E

$$x = M2^E$$

- If M is of size m bits and E is of size e bits, then x is of size N = m + e bits
- Range of positive numbers for $0.5 \leq |M| < 1$ and 2's comp. representation of M and E:

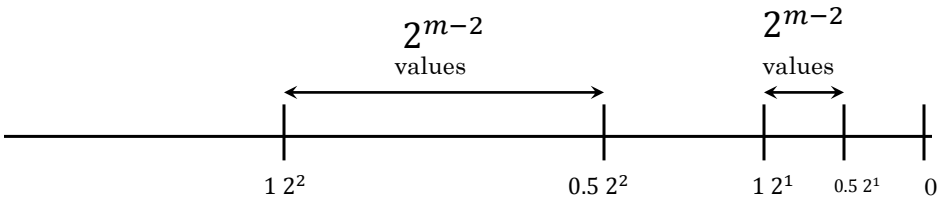
$$[\frac{1}{2}2^{-2^{e-1}}, (1-2^{1-m})2^{2^{e-1}-1}]$$

Normalization of the mantissa

- The decomposition of a real value x into the product of a mantissa and an exponent term is not unique:
  - $x=M_1 2^{E1}=M_2 2^{E2}$ …
  - Example: $12.8=0.8\ 2^4$ and also $12.8=1.6\ 2^3$
- M must be normalized to make the decomposition unique.
- The normalization is a constraint applied to M
  - for example: $0.5 \leq |M| < 1$
  - The ratio of the limits of the interval must be smaller than 2 to have the same exponent.

Floating Point Representation

- Non-linear scale:
- The precision decreases geometrically while the data size increases.



For a given number of bits:
the number of bits of the mantissa determines the precision
the number of bits of the exponent determines the dynamic range.

Floating Point Overflow or Underflow

- Very unlikely to occur

$$Overflow \quad |x| \geq 2^{2^{e-1}-1}$$

$$Underflow \quad |x| < \frac{1}{2} 2^{-(2^{e-1})}$$

Floating Point Addition Operator

$$A + B = M_a 2^{E_a} + M_b 2^{E_b}$$
$$= (M_a + M_b 2^{E_b - E_a}) 2^{E_a}$$

- It is necessary to denormalize the smallest number (B)
  - Its mantissa is multiplied by 2^{Eb-Ea} before being added to Ma.
- Loss of precision due to the rounding of the mantissa

Floating Point Multiplication Operator

$$A \times B = M_a M_b 2^{E_a + E_b}$$

- It is necessary to normalize $M_a M_b$
- 1 extra bit would be necessary to prevent overflow of $E_a + E_b$.
- 2m-1 bits are necessary to represent $M_a M_b$
- If M is truncated to m bits, the absolute error increases rapidly.

Examples of Floating Point DSP

- Some DSP devices of the C6000 family:
  - C67xx support both single and double precision format.
- The C5000 DSP are fixed point DSP but can be programmed in floating point if necessary.

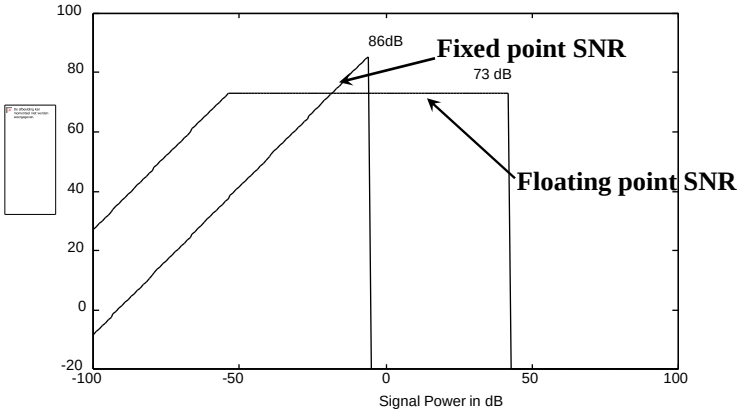
Example of Floating Point Representation

- Represent x=1.75 in Floating Point
  - Use N=8, Mantissa size m=5 bits, exponent size e=3 bits, M and E in 2's complement
  - Mantissa normalized to $0.5 \leq |M| < 1$
- Solution:
  - E=1 in binary representation: 001
  - M=0.875 in binary representation 0.1110

Comparison of Fixed and Floating Point Formats

- Fixed point: linear scale
  - Absolute error more or less constant
  - SNR decreases when the input decreases
- Floating point: non-linear scale with a geometrical progression
  - Relative error more or less constant
  - SNR more or less constant over the full data range

Comparison of Fixed Point and Floating Point SNR



Example for N=16 bits: m=12 e=4

Comparison of Fixed Point and Floating Point

- For N bits, with Floating Point format there is a compromise between dynamic range (E), and precision (M).
- Example for N=32 bits:

Fixed Point 32 bits		Floating Point: m=24 b=8	
Dynamic Range	$> 10^9$	Dynamic Range	$> 10^{77}$
Precision max	$> 9 digits$	Precision	$> 7 digits$

Dynamic range is defined as the ratio of the largest positive value on the smallest non zero positive value

Fixed Point vs. Floating Point

- Fixed Point:
  - Simple operators of addition and multiplication
  - But it is necessary to monitor overflow and underflow in order to keep precision and dynamic range at their best.
- Floating Point:
  - Greater dynamic range and simpler programming
  - More complex operators, so the performances in terms of speed or power consumption are not so good as those of fixed point DSP.

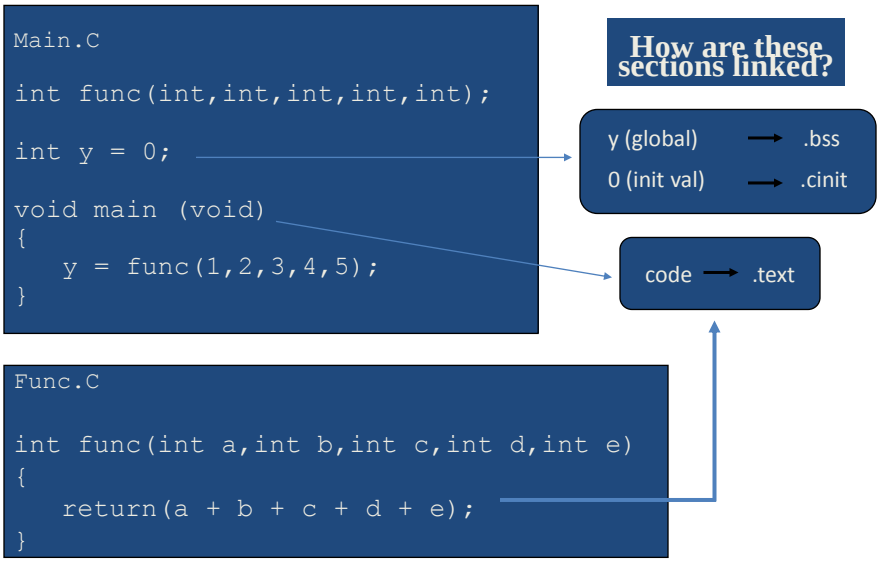
DSP C5000

C and Assembly Interface

Objectives

- Understand the C Environment
- Describe how to Mix C and Assembly
- A note on writing C code

C Run-time Environment



C Linker Command File (part 1)

```
-stack    0x2000     /* Primary stack size   */
-sysstack 0x1000     /* Secondary stack size */
-heap     0x2000     /* Heap area size       */
-c                   /* Use C linking conventions: auto-init vars at runtime */
-u _Reset            /* Force load of reset interrupt handler                */

MEMORY
{
 PAGE 0:  /* ---- Unified Program/Data Address Space ---- */

  MMR    (RWIX): origin = 0x000000, length = 0x0000c0  /* MMRs */
  DARAM0 (RWIX): origin = 0x0000c0, length = 0x00ff40  /*  64KB - MMRs
*/
  SARAM0 (RWIX): origin = 0x010000, length = 0x010000  /*  64KB */
  SARAM1 (RWIX): origin = 0x020000, length = 0x020000  /* 128KB */
  SARAM2 (RWIX): origin = 0x040000, length = 0x00FE00  /*  64KB */
  VECS   (RWIX): origin = 0x04FE00, length = 0x000200  /*  512B */
  PDROM   (RIX): origin = 0xff8000, length = 0x008000  /*  32KB */

 PAGE 2:  /* -------- 64K-word I/O Address Space -------- */

  IOPORT (RWI) : origin = 0x000000, length = 0x020000
}
```

C Linker Command File (part 2)

```
SECTIONS
{
  .text    >> SARAM1|SARAM2|SARAM0  /* Code                  */

  /* Both stacks must be on same physical memory page          */
  .stack   >  DARAM0              /* Primary system stack     */
  .sysstack >  DARAM0             /* Secondary system stack   */

  .data    >> DARAM0|SARAM0|SARAM1  /* Initialized vars       */
  .bss     >> DARAM0|SARAM0|SARAM1  /* Global & static vars   */
  .const   >> DARAM0|SARAM0|SARAM1  /* Constant data          */
  .sysmem  >  DARAM0|SARAM0|SARAM1  /* Dynamic memory (malloc) */
  .switch  >  SARAM2              /* Switch statement tables  */
  .cinit   >  SARAM2              /* Auto-initialization tables */
  .pinit   >  SARAM2              /* Initialization fn tables */
  .cio     >  SARAM2              /* C I/O buffers            */
  .args    >  SARAM2              /* Arguments to main()      */

   vectors >  VECS               /* Interrupt vectors        */

  .ioport  >  IOPORT PAGE 2       /* Global & static ioport vars */
}
```

Run-time Environment (STx_55)

ST0_55	Name	Compiler Expects	Compiler Modifies
ACOV[0–3]	Overflow detection		Yes
C	Carry		Yes
TC[1–2]	Test control		Yes
DP[7–15]	Local data page reg		No

ST1_55	Name	Compiler Expects	Compiler Modifies
BRAF	Block-repeat active		No
CPL	Compiler mode	1	No
INTM	Interrupt mode		No
M40	32/40-bit (D unit)	0	No
SATD	Saturation (D unit)	0	Yes
SXMD	Sign-exten. (D unit)	1	No
C16	Dual 16-bit math		No
FRCT	Fractional mode	0	Yes
C54CM	Compatibility mode	0	No
ASM	Accumulator shift		Yes

Run-time Environment (STx_55)

ST2_55	Name	Compiler Expects	Compiler Modifies
ARMS	AR mode	1	No
RDM	Rounding mode	0	No
CDPLC	CDP linear/circular	0	No
AR[0–7]LC	AR[0–7] lin/circular	0	No

ST3_55	Name	Compiler Expects	Compiler Modifies
MPNMC	MPNMC mode		No
SATA	Saturation (A unit)	0	Yes
SMUL	Sat on multiply	0	Yes
SST	Sat on store		No

- **If the user modifies a "presumed value", this value must be restored by the function**

Initializing the C Environment ...

Boot.c in rts55.lib

`_c_int00:`

- Initialize global and static variables
- Initialize C environment variables to expected values
- Setup stacks (SP & SSP)
- Call _main

On reset, how do you tell the CPU to begin execution at _c_int00?

```
;cvectors.asm
        .ref _c_int00
        .sect "vectors"
rsv:    .ivec  _c_int00
```

All assembler symbols accessed in C require a leading underscore

Objectives

- Understand the C Environment
- Describe how to Mix C and Assembly
- A note on writing C code

Creating a C-Callable Assembly Routine

```
Main.C

int func(int,int,int,int,int);

int y = 0;

void main (void)
{
   y = func(1,2,3,4,5);
}
```

Main.C:
- prototypes called function
- calls function

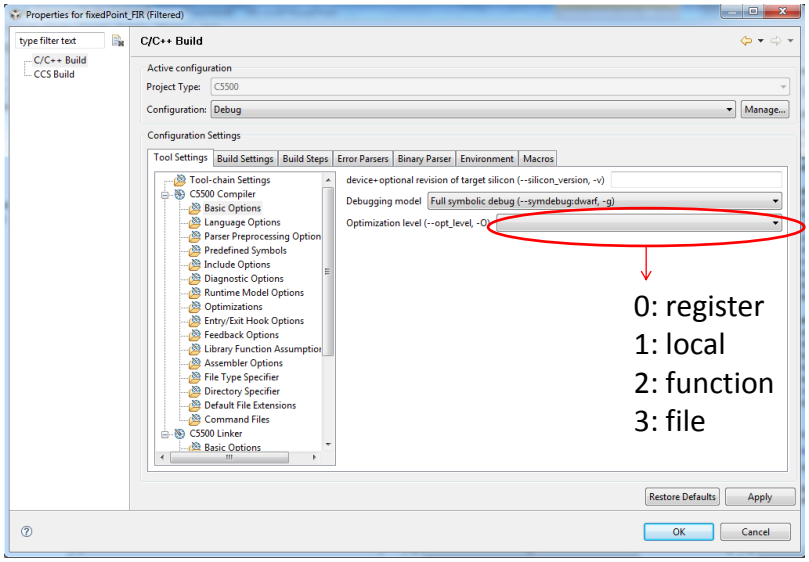
How are the parameters passed to func( ) ?

```
Func.C

int func(int a,int b,int c,int d,int e)
{
   return(a + b + c + d + e);
}
```

With -o3 enabled, func is deleted and main simply does: `MOV #15,*(#_y)`

Compiler options



Parameter Passing Conventions

```
y = func(1,2,3,4,5);
```

- **The compiler will scan the parameters (from left to right) and place them into the following registers (from left to right):**

16-bit Integers	Pointers	Longs
T0-1, AR0-4	XAR0-4	AC0-2

- **The registers are filled in the order shown:**
  - **`T0` gets the first 16-bit int, `T1` gets the 2nd, etc.**
  - **`XAR0` gets the first pointer if available, etc.**
  - **32-bit values are passed in `AC0-2`**
- **All parameters that don't find a home in a register, are placed on the stack (SP).**
- **So, which registers will contain the parameters listed above?**

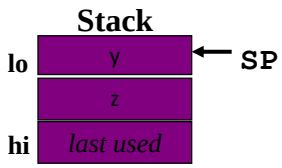
Exercise/Recommendations

```
var = func(0x98765432,1,2,3,4,5,a,x,y,z);
```

a, x, y, and z are pointers

Where are these parameters placed by the compiler?

T0=	1
T1=	2
XAR0=	3
XAR1=	4
XAR2=	5
XAR3=	a
XAR4=	x
AC0=	0x98765432



(parameters are placed on the stack as shown)

Recommend:

- **Pass pointers first**
- **Pass most-used parameters first**

Calling Convention, Accessing the Stack

So, who's responsible for saving registers? The parent or child?

Parent - SOC (save on call)	
T0-1	XAR0-4 AC0-3

Child - SOE (save on entry)	
T2-3	XAR5-7

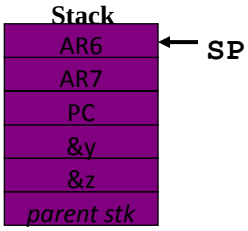
If the child routine modifies the SOE registers, it must preserve their contents

Return values from a function are placed here:

Return Values	
16/32-bit integer: T0/AC0,	Data/Func pointer: XAR0/AC0

Accessing parameters on the stack:

```
PSH   AR6,AR7
MOV   *SP(#3),AR6
MOV   *SP(#4),AR7
```



Remember Func.C ?

```
Main.C

int func(int,int,int,int,int);

int y = 0;

void main (void)
{
   y = func(1,2,3,4,5);
}
```

Now let's write Func.C in assembly...

```
Func.C

int func(int a,int b,int c,int d,int e)
{
   return(a + b + c + d + e);
}
```

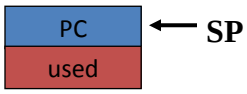
Writing Func.ASM

Entry
- declare func as global
- define entry point (label)
- save SOE registers

Algorithm
- execute the algorithm
  return(a + b + c + d + e);
- place result in return reg

Exit
- restore SOE registers
- return to calling routine

```
   .def _func
_func:
   ;push SOE registers
   ;adjust STx_55 values
    ADD  T1,T0        ;a + b
    ADD  AR0,T0       ;+ c
    ADD  AR1,T0       ;+ d
    ADD  AR2,T0       ;+ e
   ;restore STx_55 values
   ;pop SOE registers
    RET
```



Other C Stuff...

```
asm("IDLE");

#include <intrindefs.h>
y = _sadd(x1, x2);

#pragma Data_Section(y,"Var");
int y = 0;

volatile unsigned int *ctrl;
while (*ctrl != 0xFF);

interrupt void my_ISR (void)
```

In-Line Assembly
- can disrupt C

Intrinsics
- ASM instructions in C
- see C Compiler guide

Data/Code Sections

Volatile Keyword
- compiler may remove code w/o volatile kywd

Interrupt Keyword
- context save/restore

Linker Command Options
- stack, sysstack and heap sizes can be changed via CCS

Objectives

- Understand the C Environment
- Describe how to Mix C and Assembly
- A note on writing C code

Introduction

- The C55x C compiler translates a C source code into an assembly source code
- C55x assembler converts the assembly code into a binary object file
- The C55x linker combines it with other object files and library files → executable file.
- Knowing how the C compiler generates the assembly code can help us to write correct and efficient C programs.

Addition and multiplication (1)

- One common mistake of working with short variables is not using data type cast to force them into 32-bit data.
- For example, the correct C statement of multiplying two 16-bit numbers is:

```
long mult_32bit;
short data1_16bit, data2_16bit;
mult_32bit = (long)data1_16bit * data2_16bit;
```

Addition and multiplication (2)

Incorrect statements:

```
mult_32bit = data1_16bit * data2_16bit;
mult_32bit = (long)(data1_16bit * data2_16bit);
add_32bit = data1_16bit + data2_16bit;
sub_32bit = (long)(data1_16bit - data2_16bit);
```

Addition and multiplication (3)

- The C55x compiler generates the assembly code that performs a 16-bit data ‘a’ multiplied by another 16-bit data ‘b’.

```
short a=0x4000, b=0x6000;
long c;
void main(){
// Wrong: only the lower 16-bit result in AC0 is saved to c
c = a * b;
000100 A511018F     mov *abs16(#018fh),T1
000104 D31105018E   mpym *abs16(#018eh),T1,AC0
000109 A010 98      mov mmap(AC0L),AC0
00010C EB11080190   mov AC0,dbl(*abs16(#0190h))
```

Addition and multiplication (4)

```
// Wrong: only the lower 16-bit result in AC0 is saved to c
c = (long)(a * b);
000111 D31105018E   mpym *abs16(#018eh),T1,AC0
000116 A010 98      mov mmap(AC0L),AC0
000119 EB11080190   mov AC0,dbl(*abs16(#0190h))

// Correct: 32-bit result in AC0 is saved to c
c = (long)a * b;
00011E D31105018E   mpym *abs16(#018eh),T1,AC0
000123 EB11080190   mov AC0,dbl(*abs16(#0190h))
```

Defining loops (1)

- C compiler generates different assembly code according to the data type of the loop counter i.
- Loop can be efficiently implemented using the C55x block-repeat instructions rpt and rptb.
- BRC0 and BRC1 are 16-bit registers, limits loopcounter to 16 bits.
- If a max. of 65535 ($2^{16} - 1$) is ok, define counter either as short or unsigned short.

Defining loops (2)

- more efficient code when the loop counter is defined as a 16-bit short variable.

```
short a[300];
void main(){
long i; short *p=a, j;
// Inefficient for-loop using long loop counter i
for (i=0; i<100; i++){
*p++ = 0;
000269                L1:
000269 E66300         mov #0,*AR3+
00026C 76006418       mov #100,AC1
000270 4010           add #1,AC0
000272 120410         cmp AC0 < AC1, TC1
000275 0464F1         bcc L1,TC1
}
```

Defining loops (3)

```
// More efficient for-loop using short loop counter j
for (j=0; j<100; j++)
{
*p++ = 0;
000278 4C63           rpt #99
00027A E66300         mov #0,*AR3+
}
}
```

Built-in library functions (1)

- Avoid compiler built-in supporting library functions to improve the real-time efficiency.
- Less function calls (set up and pass arguments, collect return values, SOC)
- An example of modulo statement in C is listed below:

Built-in library functions (2)

```
void main(){
short a=30;
000102 E6001E mov #30,*SP(#00h)
// Modulo operation calls library function
a = a % 7;
000105 A400 3D75    mov *SP(#00h),T0 || mov #7,T1
000109 6C000297     call __remi
00010D C400         mov T0,*SP(#00h)
```

Built-in library functions (3)

```
// Inefficient modulo operation of a power of 2 number
a = a % 8;
00010F 2240 3D7A          mov T0,AC0 || mov #7,AR2
000113 36AA 11453E        not AR2,AR2 || sfts AC0,#-2,AC1
000118 76E00091           bfxtr #57344,AC1,AR1
00011C 2409               add AC0,AR1
00011E 289A               and AR1,AR2
000120 26A0               sub AR2,AC0
000122 C000               mov AC0,*SP(#00h)
// Efficient modulo operation of a power of 2 number
a = a & (8-1);
000124 F4000007           and #7,*SP(#00h)
}
```

“Getting Started”

Texas Instruments University Programme
Teaching Materials

Objectives

- To build your first project using the TMS320C5505 USB Stick.
- Playback music using the TMS320C5505 USB stick.
- Modify the code.

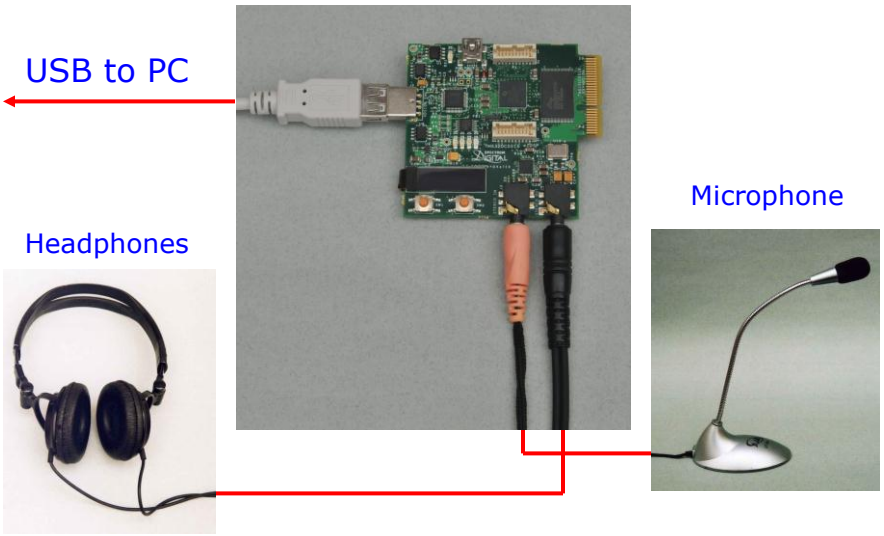
Important

- Follow the steps carefully!
- If the wrong memory models are used, the project will not build.

Compatibility

- All of the applications provided on this teaching CD will also run on the TMS320C5505 and TMS320C5515 USB Stick without modification.
- To run the applications on the older TMS320VC5505 USB stick, it is necessary to make a change to the file `csl_general.h`

TMS320C5515 USB Stick Setup



Workspace

- Code Composer Studio creates a “workspace”, that is a folder where projects are stored
- The default directory is:

  Desktop -> My Documents -> Workspace, change to be somewhere on the I drive
- Make a MyFirstProject folder in this workspace

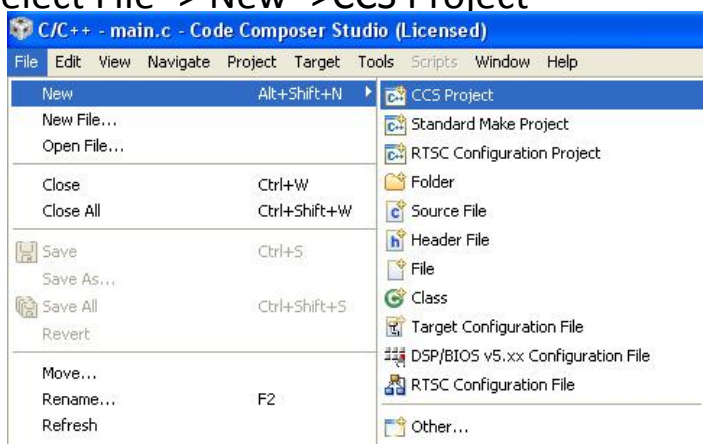
Start Code Composer Studio

- Start Code Composer Studio



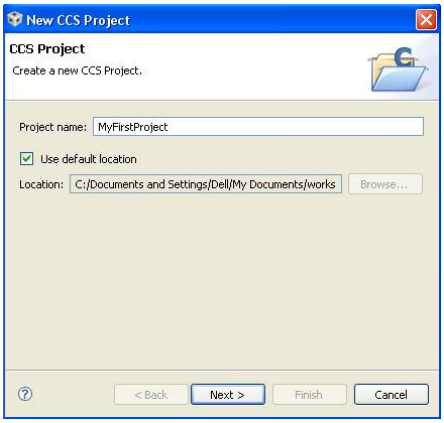
Creating a New Project

- Select File -> New ->CCS Project



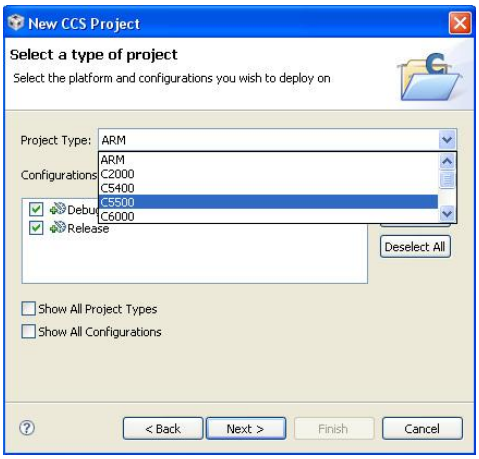
Naming the New Project

- For Project Name use MyFirstProject. Click Next >



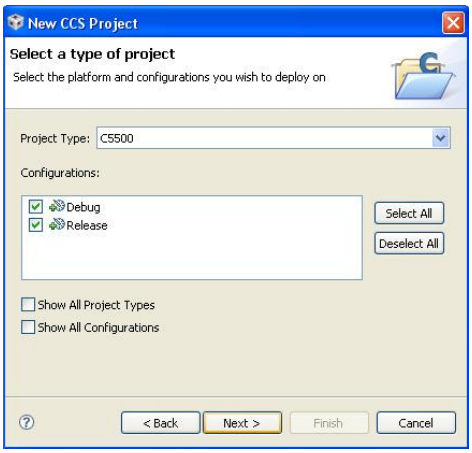
Project Types Available

- Default is ARM. Select Project Type C5500 from list.



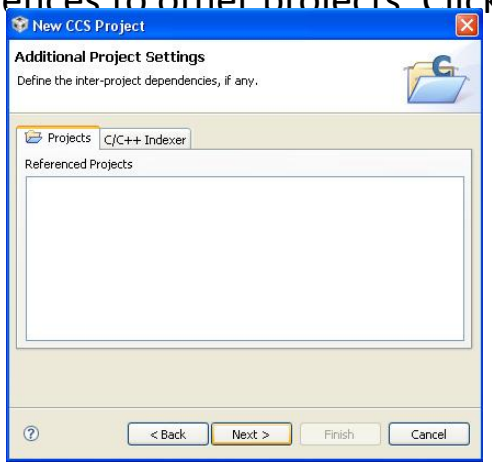
Project Type Selected

- Click on Next >



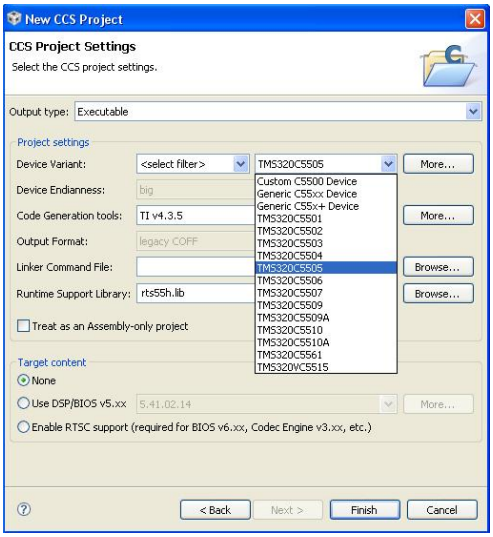
Additional Project Settings

- No references to other projects. Click Next >



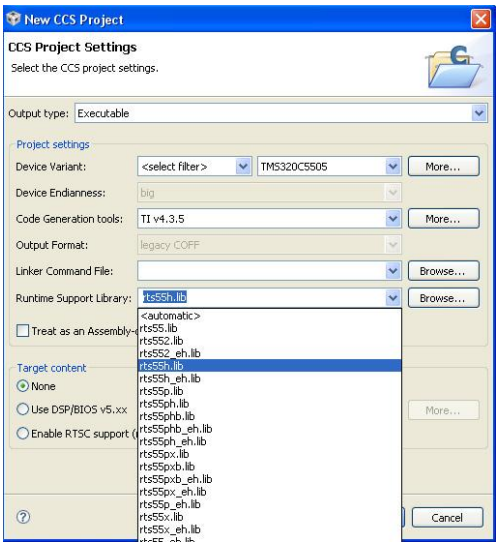
Select TMS320C5515

- From the pull-down menu, select TMS320C5515
- Note: This and others windows may look slightly different, depending on the release version of CCS 4.



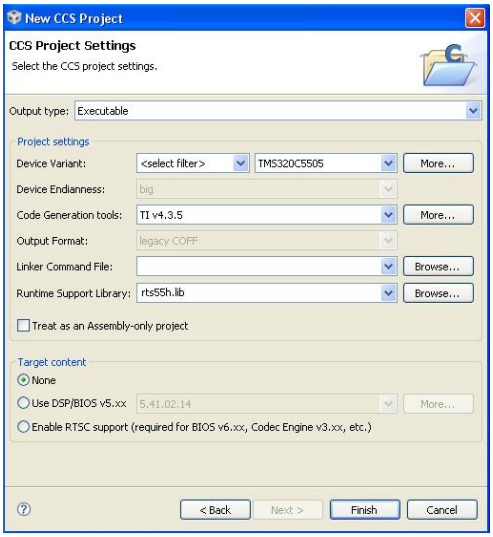
Select Runtime Support Library

- The Runtime Support Library required is rts55h.lib, which is automatically selected.



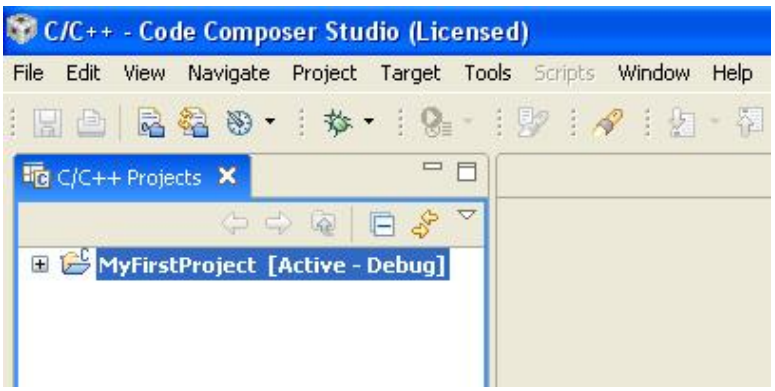
Select Linker Command File

- Do not put anything in Linker Command File.
- Click on Finish.



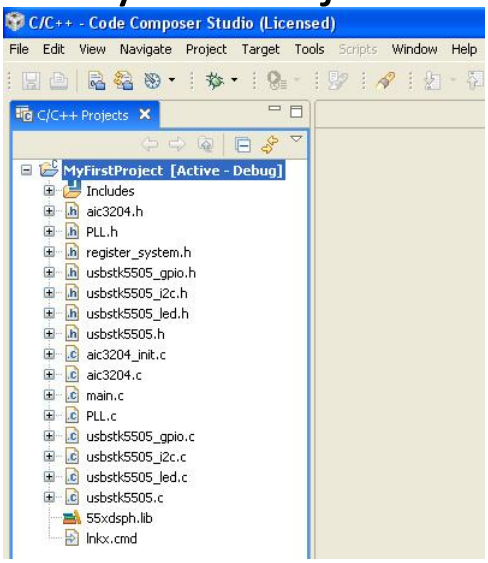
Active Project

- The the C/C++ Projects window, MyFirstProject is the Active Project for Debug.



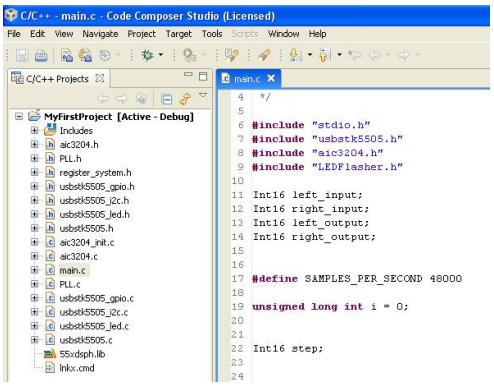
Show Files in MyFirstProject

- Click on + to expand the list of files.



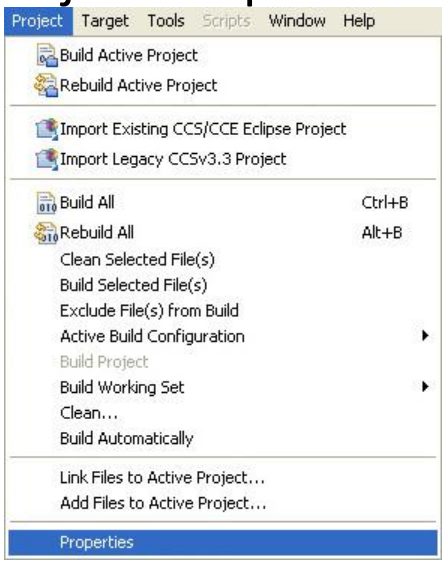
View `main.c`

- Double left click on `main.c` to open the `main.c` file.
- Right click on main1.c and select “exclude from build”.

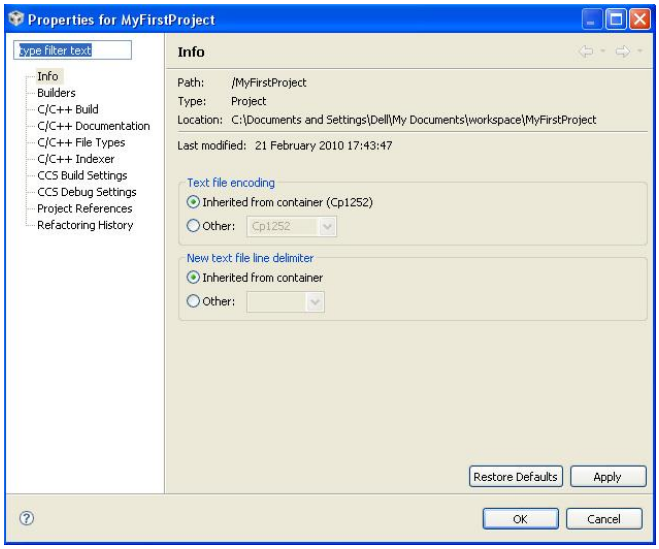


Select Project Properties

- Select Project -> Properties

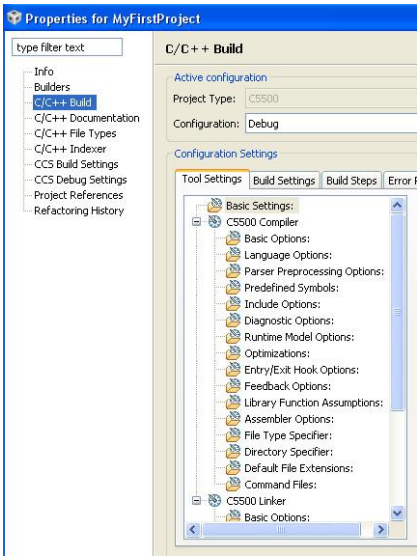


Properties for MyFirstProject



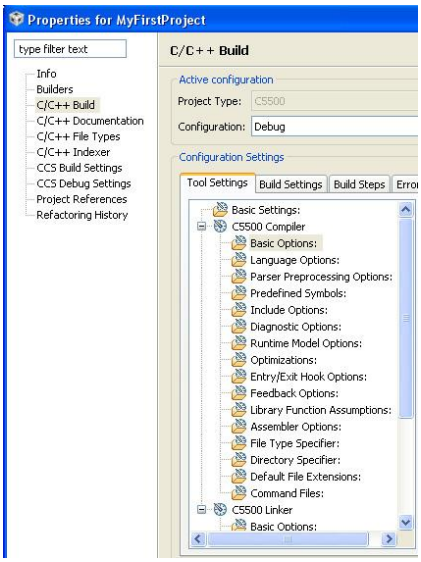
C/C++ Build – Basic Settings

- Double Click on C/C++ Build



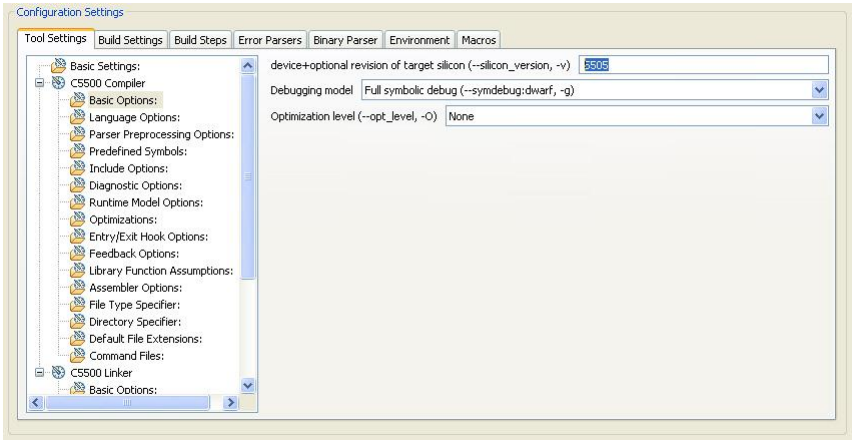
Basic Options

- Click on Basic Options.



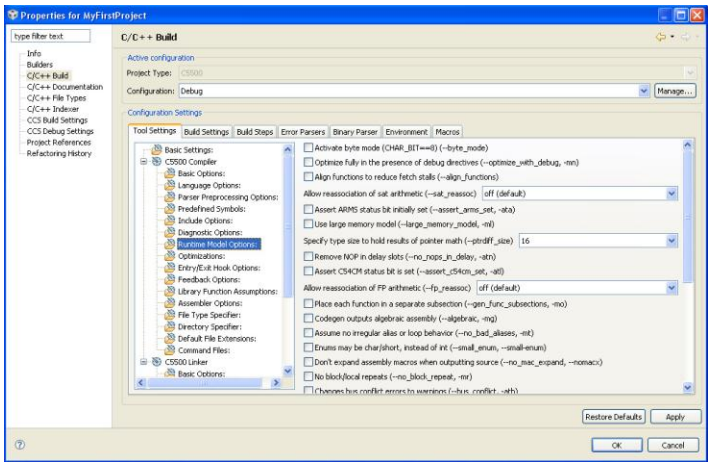
Setup Silicon Version

- Type 5515 in for (--silicon_version, -v)



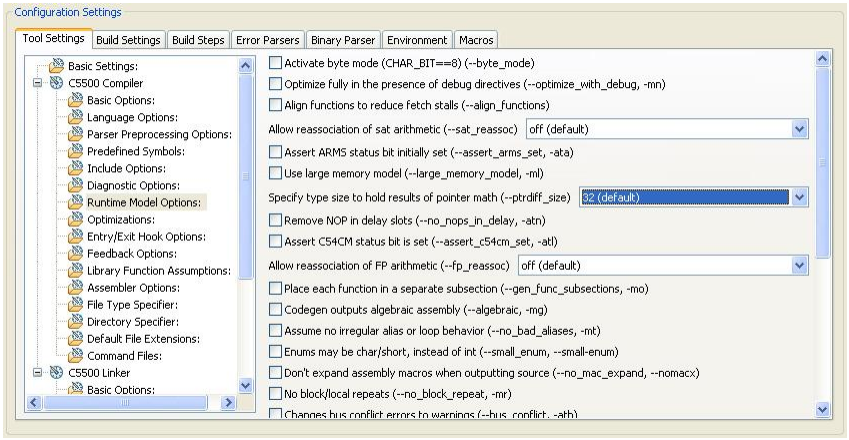
Runtime Model Options

- Double click on Runtime Model Options



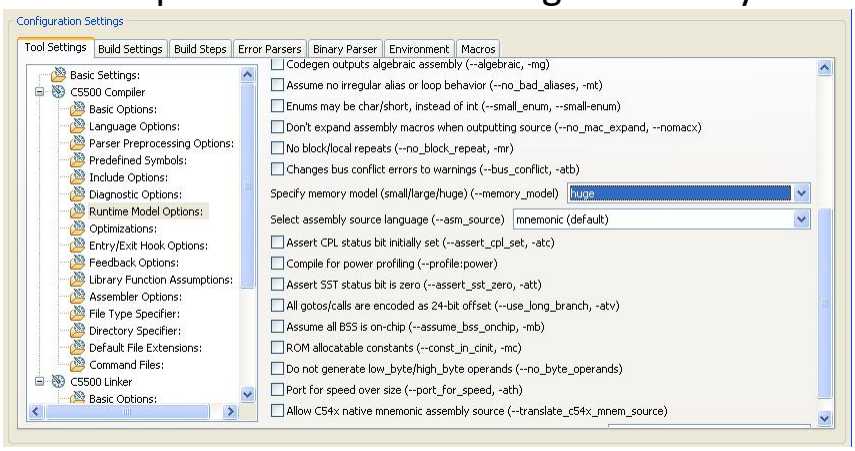
Setting the Pointer Size

- Set (--ptrdiff_size) to 32 (default).



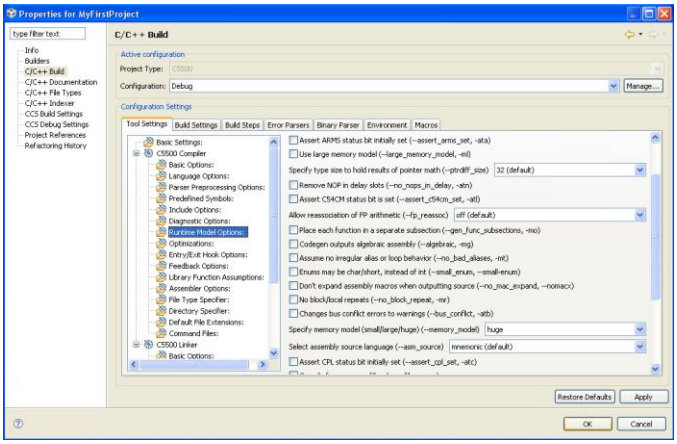
Selecting the Memory Model

- It is important to use the “huge” memory



Runtime Model Options Selected

- Click on OK.

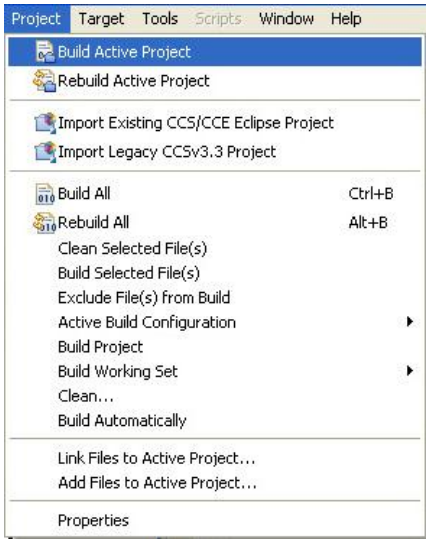


Building the Project

- Select:

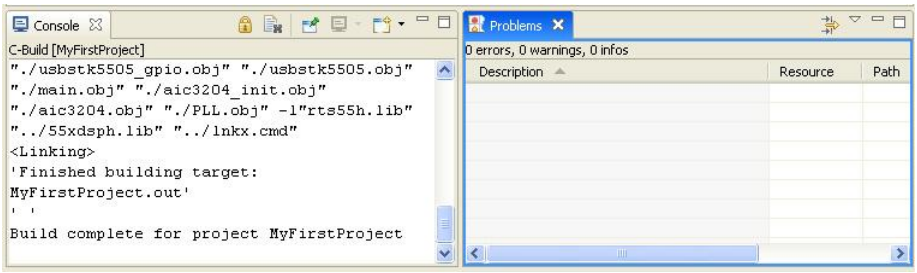
Project ->

Build Active Project.



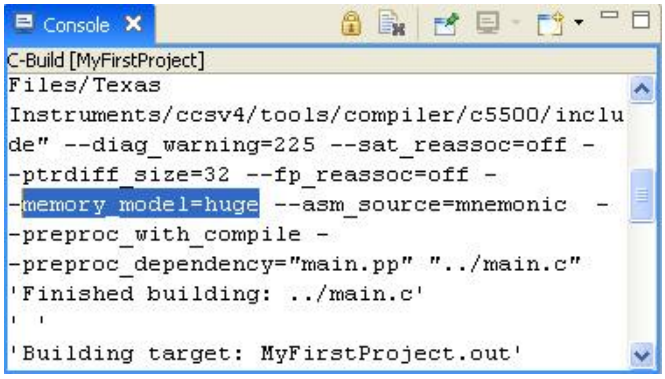
Build Complete

- Console windows shows build is complete
- Problems window should contain no errors or warnings.



Confirming Memory Model

- Scroll up the Console Window to check that memory_model=huge.



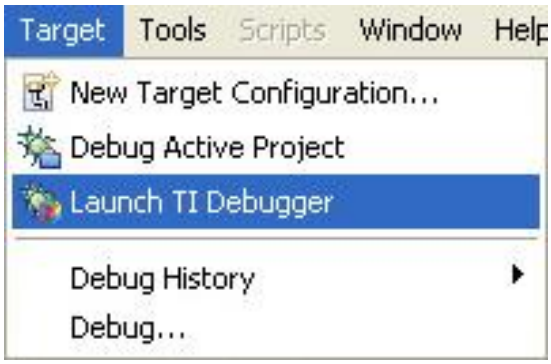
Problems with the Build?

- If you are having specific problems with the build, it is most likely that you have missed out a step in the process
- It is important to have:
  - Silicon version 5515
  - huge memory model
  - rts55h.lib

Debugging the Project

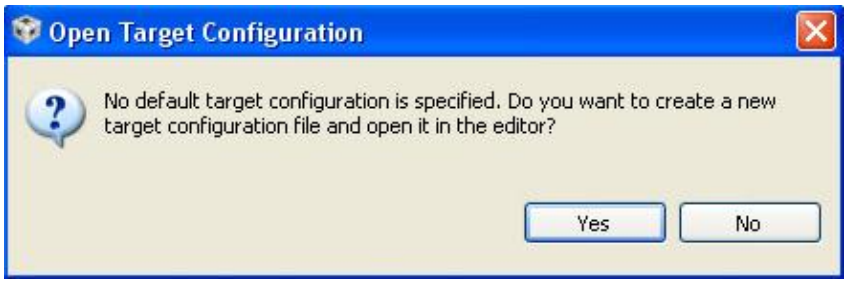
Launch TI Debugger

- Select Target -> Launch TI Debugger



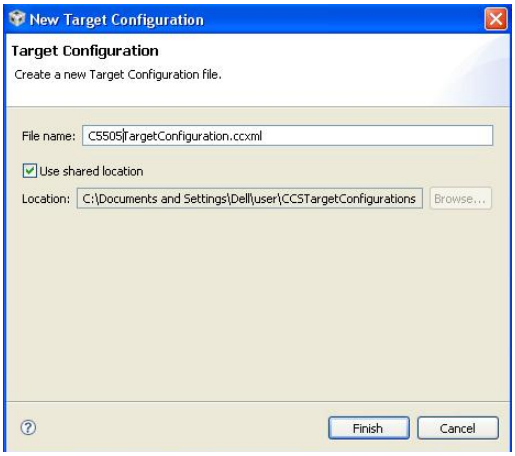
First Debug Session Only

- The first time you debug, you will be asked to Open Target Configuration. Click Yes.



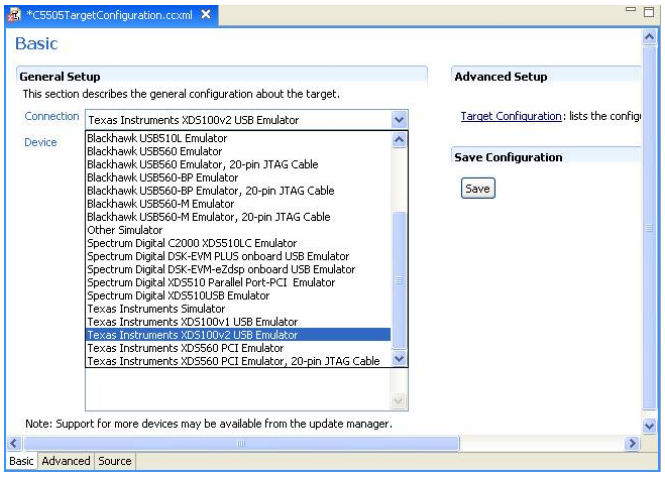
Naming Target Configuration

- You may wish to change the Target Configuration Name.



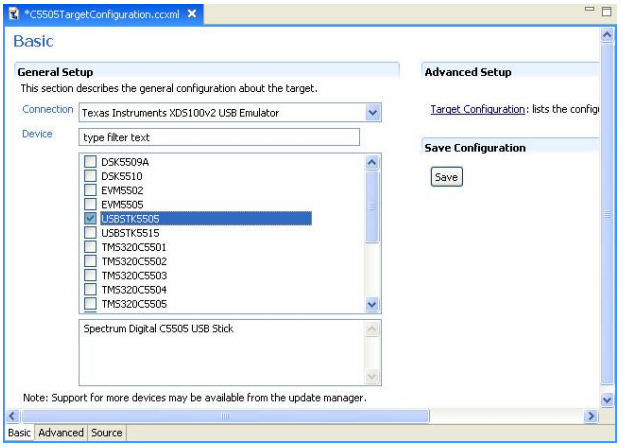
Setup XDS100 USB Emulator

- Select Texas Instruments XDS100v2 USB Emulator.



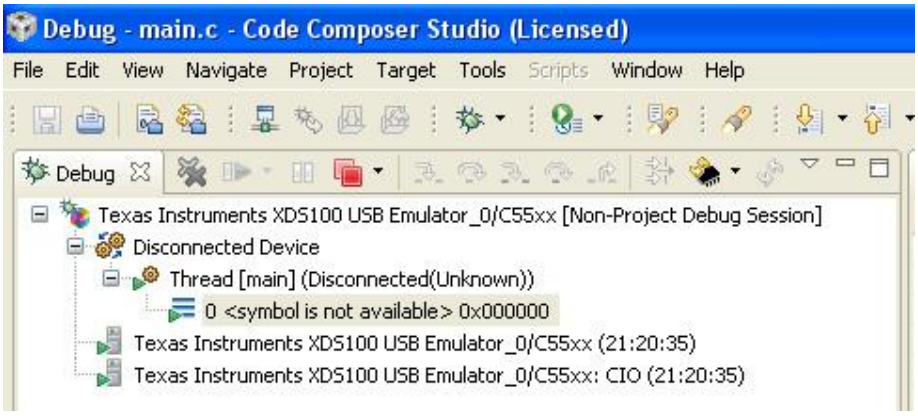
Configure C5515 USB Stick

- Select USBSTK5515. Click on Save.



XDS100 Now Connected

- The XDS100 USB Emulator is now connected.

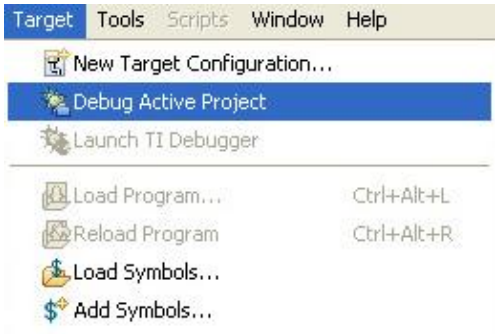


XDS100 Emulator Configured

- The Console window shows that the Phase Lock Loop has been set up for 100MHz operation
- The XDS100 USB Emulator has been successfully set up.

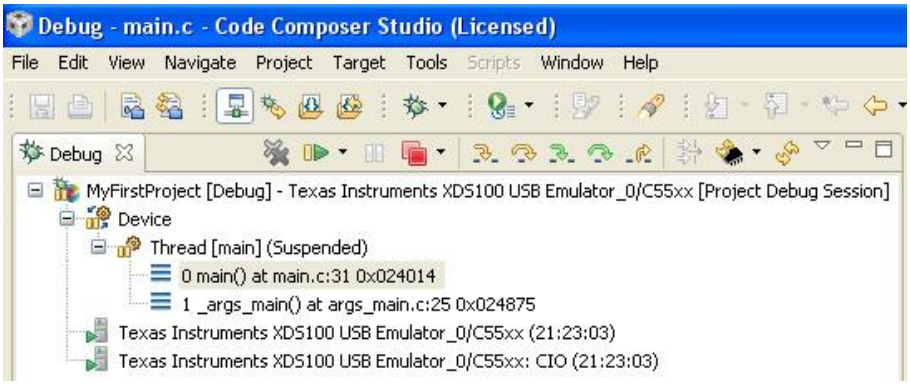
Start Debug of MyFirstProject

- Select Target -> Debug Active Project.



Debugging MyFirstProject

- XDS100 USB Emulator is debugging MyFirstProject.

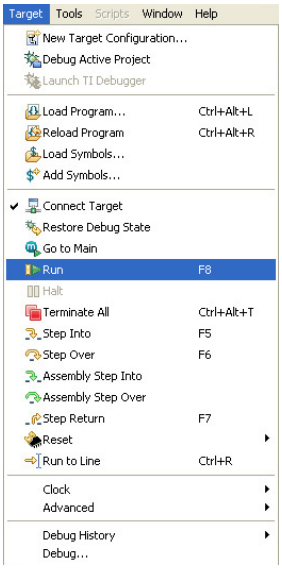


Go to `main()`

- Arrow shows that the program is ready to run from the beginning of `main()`.

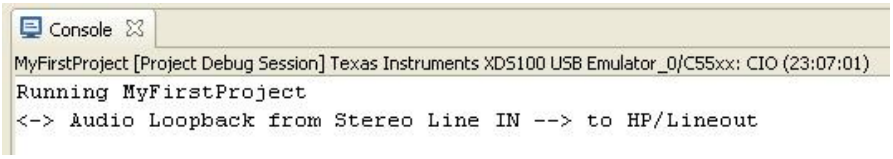
Run Program

- Select Target -> Run.



Program Running

- The following text amongs others is shown in the Console Window:



Halt Program Execution

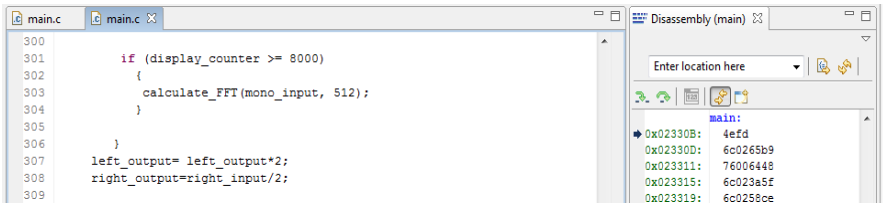
- To stop the program, select Target -> Halt.



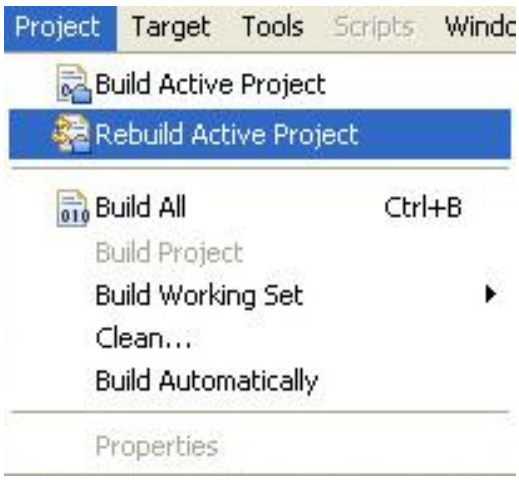
Modifying C Code

After Modification

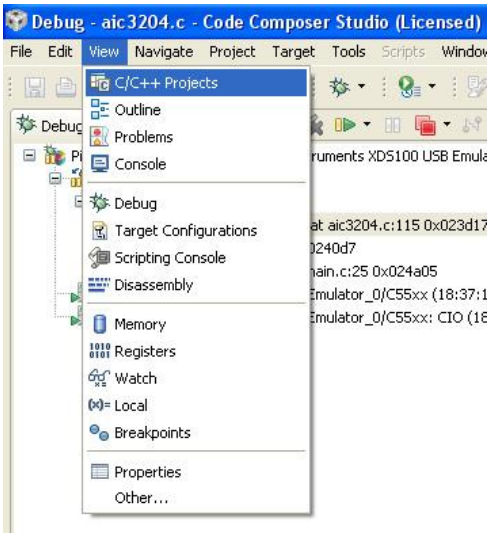
- `main.c` after modification. Change gain of channels.



Rebuild Active Project

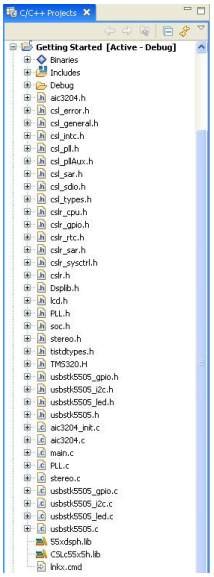


Viewing Files in Debug



Viewing Source Files

- Highlight any file and click on it to edit.



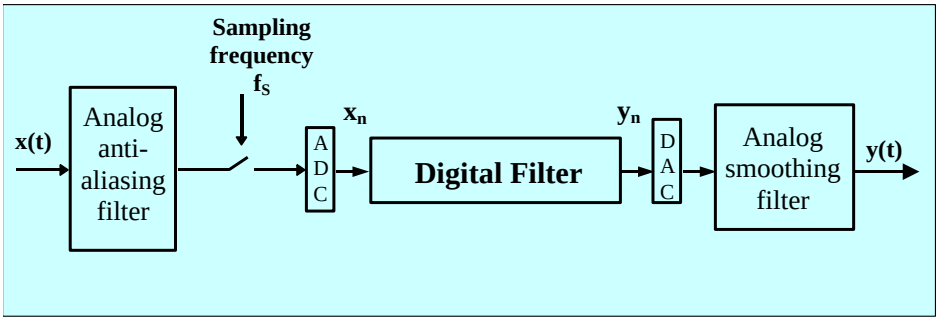
DSP C5000

Finite Impulse Response (FIR) Filter Implementation

Outline of FIR Filters

- FIR Filters with Matlab
- Implementation of FIR Filters

Digital Filters





Rational z Transfer Function

$$H(z) = \frac{N(z)}{D(z)} = \frac{\sum_{i=0}^{Q} b_i z^{-i}}{1 + \sum_{k=1}^{P} a_k z^{-k}}$$

- Linear equation with constant coefficients.

$$y_n = \sum_{i=0}^{Q} b_i x_{n-i} - \sum_{k=1}^{P} a_k y_{n-k}$$

FIR and IIR

- FIR: output y_n is a linear combination of a finite number of input samples.

$$y_n = \sum_{i=0}^{Q} b_i x_{n-i}$$

- IIR: output y_n is a linear combination of a finite number of input and of output samples. Recursive form.

$$y_n = \sum_{i=0}^{Q} b_i x_{n-i} - \sum_{k=1}^{P} a_k y_{n-k}$$

Filter design

- Design of FIR filters using MATLAB (cfr. First semester)
- Design of FIR filters using FDATool

FIR design with Matlab

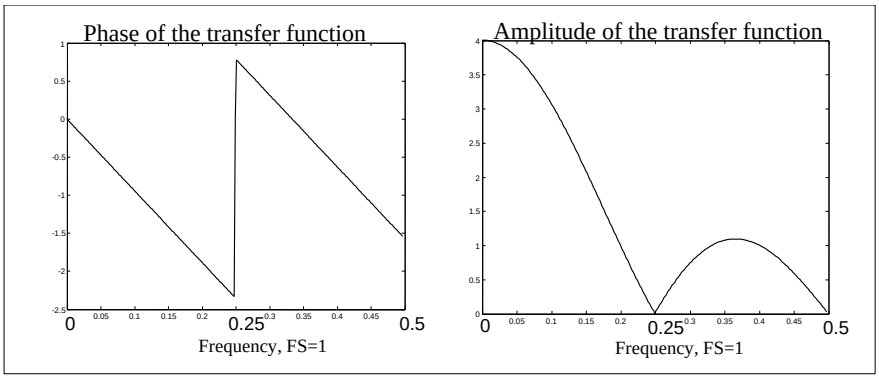
- Example for a FIR filter:

$$N(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}$$
$$b = [b_0 \quad b_1 \quad b_2 \quad b_2] = [1 \quad 1 \quad 1 \quad 1].$$

- Enter the filter coefficients vector b:
  - b=[1 1 1 1]; a=1;
- Calculate transfer function Hf, its amplitude and phase on 256 samples, with fs=1:
  - [Hf,f]=freqz(b,a,256,1);
  - HfA=abs(Hf);
  - Hfphi=angle(Hf);

Some Useful Matlab Functions

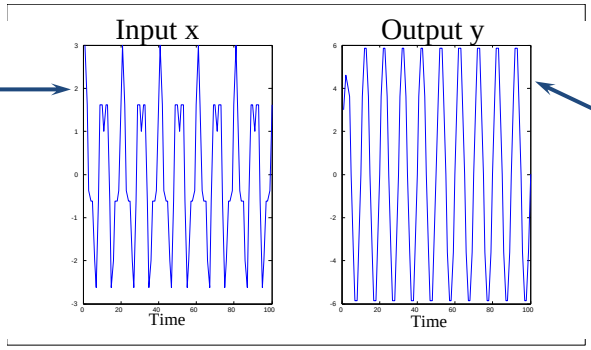
- Plot impulse response: stem(b)
- Plot amplitude and phase of transfer function: plot(f,HfA) and plot(f,Hfphi)



Some Useful Matlab Functions

- Generate a test signal = sum of cosines:
  - x=cos(2*pi*[0:99]*0.25)+2*cos(2*pi*[0:99]*0.1);
- Apply the filter to x. Output is y:
  - y=filter(b,a,x);
- Plot the results: plot(x); plot(y)

x is the sum of 2 frequencies : 0.25 and 0.1.



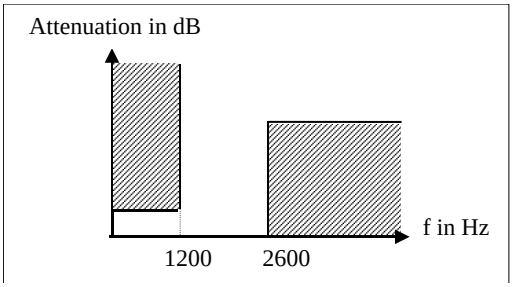
The filter cancels the frequency 0.25. y has only the freq. 0.1.

Calculation of a FIR using Matlab

- For given attenuation and frequency response characteristics, the transfer function can be calculated using different methods:
  - Mean square error, miniMax (Chebychev)
  - Empirical window method
- Corresponding Matlab functions
  - firls and remez.
  - fir and fir1.

Example using Matlab

- Design a low pass filter:
  - Sampling frequency = 9600 Hz
  - Maximum attenuation (passband) = 0.1 dB
  - Minimum attenuation (stopband) = 50 dB
  - Limit frequencies of passband and stopband = 1200 Hz and 2600 Hz.

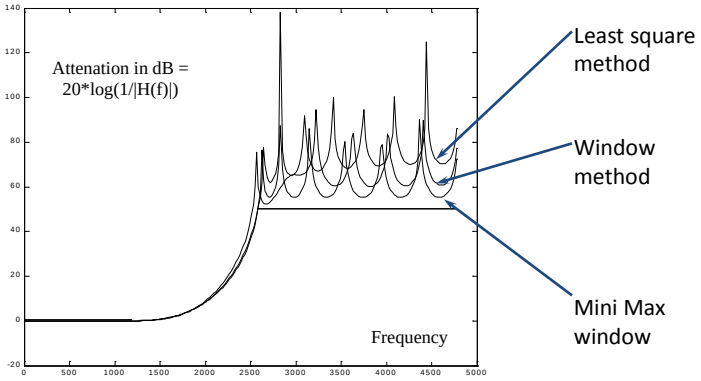


Example using Matlab

- Vector of limited frequencies (normalized)
  - F=[0 1200 2600 4800]/4800;
- Vector of required amplitudes:
  - A=[1 1 0 0];
- Least square calculation of filter:
  - Bls=firls(23,F,A);
- Mini Max calculation of filter:
  - Bre=remez(21,F,A);
- Window method (Hamming):
  - Bwin=fir1(25,(1200+2600)/9600);

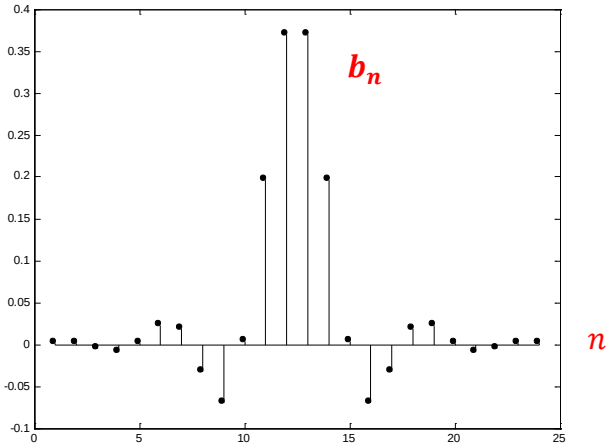
Results of Matlab Example

- The minimum orders to satisfy the constraints are 23 for LS, 21 for minimax and 25 for the window method.



Results of Matlab Example

- Impulse Response



FIR Filters with Constant Group Delay or Linear Phase

- For many applications, it is desirable to use a filter with a constant group delay (independent of the frequency).
  - The phase will be linear or affine.

    - 2 possible cases:
      - symmetrical or asymmetrical FIR.
      - Constant group delay = T_S (N-1)/2
      - Symmetrical: *h(n)=h(N-1-n)*
      - Asymmetrical: *h(n)=-h(N-1-n)*

FIR filters with Constant Group Delay or Linear Phase

- Symmetric case: linear phase

$$\Phi(f) = kf$$

- Asymmetrical case:

$$\Phi(f) = kf \pm \frac{\pi}{2}$$

Design digital filter using FDATool

FDATool (>>fdatool) has the following functions:

- designing filters by setting filter specifications;
- analyzing designed filters;
- converting filters to different structures; and
- quantizing and analyzing quantized filters.

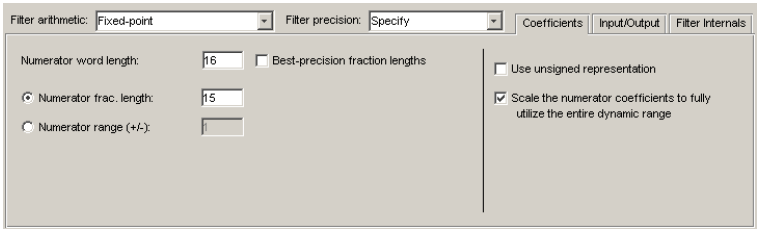
FDATool example

Design a bandpass FIR filter for a 16-bit fixed-point DSP processor following specifications:

- Sampling frequency = 8000 Hz.
- Lower stopband cutoff frequency (*Fstop1) = 1200 Hz.*
- Lower passband cutoff frequency (*Fpass1) = 1400 Hz.*
- Upper passband cutoff frequency (*Fpass2) = 1600 Hz.*
- Upper stopband cutoff frequency (*Fstop2) = 1800 Hz.*
- Passband ripple = 1 dB.
- Stopband (both lower and upper) attenuation = 60 dB.

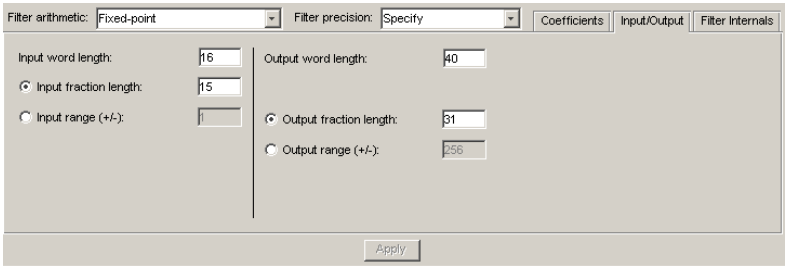
FDATool example

- Once the filter has been verified, choose Set Quantization Parameters 
  - Coefficients tab defines the coefficient quantization



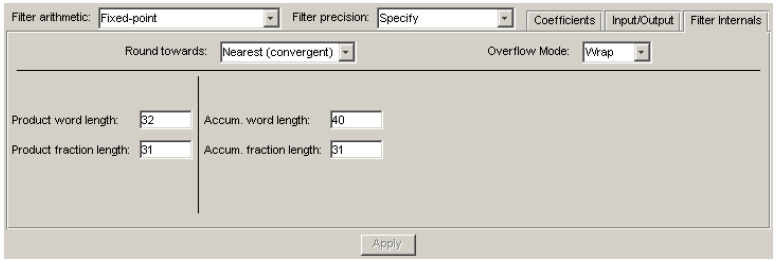
FDATool example

- Input/Output tab quantizes the input and output signals for the filter.



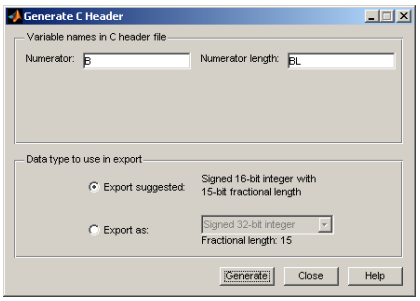
FDATool example

– Filter Internals tab sets a variety of options for the arithmetic.



Export data

- To create a C header file containing filter coefficients, select Generate C header from the Targets menu.



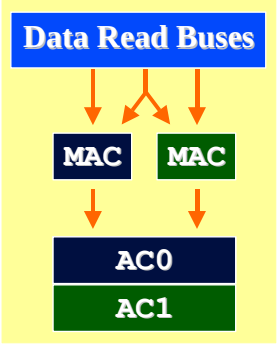
- Or, if TMS320C5000 CCS installed choose option Composer Studio (tm) IDE from Targets menu

Implementation of FIR Filters on C55x

- Implementation of block filters

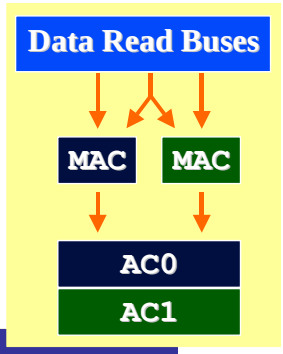
Implementation of FIR Filters using C55x

- 2 MAC units accessed using 3 data buses D, B, C make it possible to:
  - Calculate 2 output samples y at a time using same set of coefficients and different data x.
  - Calculate 2 output samples y at a time using same input data x but 2 set of coefficients.



Using the 2 MAC Units

- Use of block filtering in order to calculate 2 output samples at a time.



$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + b_3 x_{n-3}$

$y_{n+1} = b_0 x_{n+1} + b_1 x_n + b_2 x_{n-1} + b_3 x_{n-2}$

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

$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + b_3 x_{n-3}$

C54x MAC *AR2+, *AR3+, A

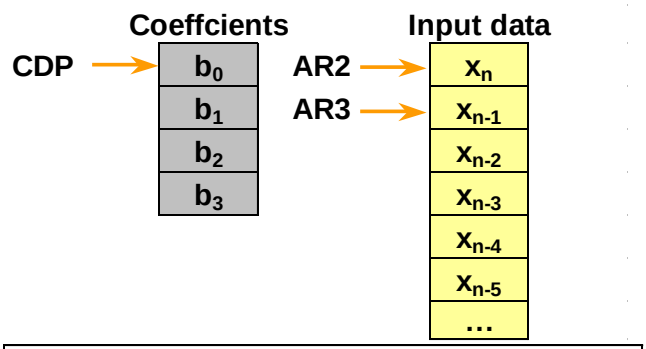
Block Filter

- Calculate a block of M output samples:
  - Avoids interrupts sample by sample
  - Allows calculation of 2 samples at a time

$$y_{n-m} = \sum_{i=0}^{N-1} b_i x_{n-m-i} \quad \forall m \in [0, M-1].$$

- M+N-1 inputs necessary to calculate M output samples.
  - Because of N-1 initial conditions.

Block Filter, example N=4, M=3



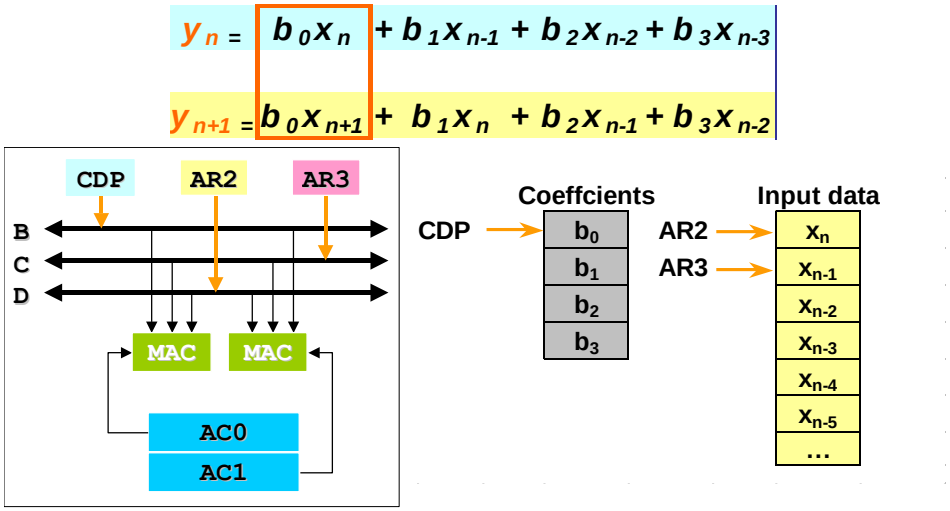
$$y_n = b_0x_n+b_1x_{n-1}+b_2x_{n-2}+b_3x_{n-3}$$

$$y_{n-1} = b_0x_{n-1}+b_1x_{n-2}+b_2x_{n-3}+b_3x_{n-4}$$

$$y_{n-2} = b_0x_{n-2}+b_1x_{n-3}+b_2x_{n-4}+b_3x_{n-5}$$

Block Filter Example

- Double loop:
  - On coefficients and on m
- Coefficients accessed by CDP:
  - CDP (Cmem) modifications limited to: *CDP, *CDP+, *CDP-, *(CDP+T0).
  - CDP uses B bus *only* for dual-MAC. Because B bus is internal only, coefficients must also be internal.
- Place data operands carefully to avoid memory conflicts (SA/DARAM). Signal buffer and coefficient buffer in separate memory blocks.



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

Initialization of Pointers

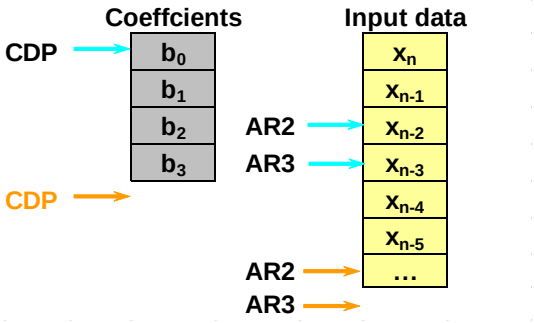
- Use AMOV to do transfers during the "AD" pipeline phase.
- Init AR2 to point to the 1st value of input data : (x)
- Init AR3 to point to the 2nd value of input data (x+1)
- Init CDP to point to coefficient array (a)

```
AMOV #x,XAR2
AMOV #(x+1),XAR3
AMOV #a0,XCDP
```

Inner Loop on Coefficients

```
RPT  #3
    MAC *AR2+,*CDP+,AC0
:: MAC *AR3+,*CDP+,AC1
```

Pointers at the end of the repeat instruction:

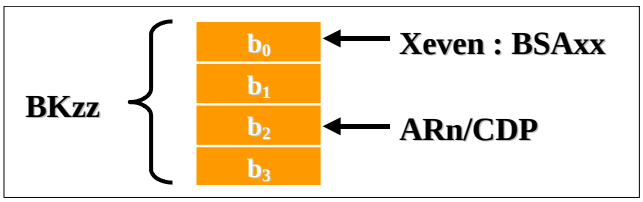


Reinitialization of pointers for next output sample:

```
ASUB #2,AR2
ASUB #2,AR3
MOV  #a0,CDP
```

Circular Addressing Mode for Coefficients

- Initialize size of the circular buffer: BK
- Set up Buffer Start Address: BSA and Xeven
- Set up ARi or CDP
- No memory alignment constraint



Circular Buffer Addressing Mode



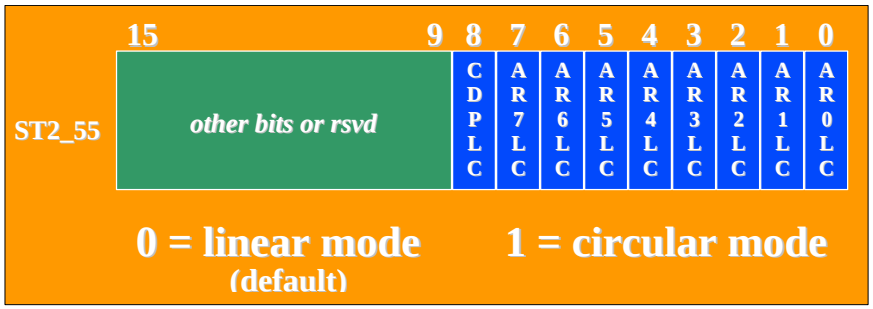
Circular Buffer Addressing Mode

Offset	Xeven	Buffer Start Address	Block size Register
AR0 AR1	XAR0[22:16]	BSA01	BK03
AR2 AR3	XAR2[22:16]	BSA23	
AR4 AR5	XAR4[22:16]	BSA45	BK47
AR6 AR7	XAR6[22:16]	BSA67	
CPD	XCDP[22:16]	BSAC	BKC

The even XARn (i.e. 0,2,4,6) determines the 64K Page

Selecting Circular or Linear Addressing Mode

- Use the LSB of Status word ST2_55



- **Set or reset status bits:**

```
BSET AR5LC    ;AR5 in circular mode
BCLR AR3LC    ;AR3 in linear mode
```

Circular Buffer for Coefficients

- Table of coefficients b0 … b3:
  - Circular buffer addressed by CDP.
  - Initialize XCDP: 7 MSB
  - Initialize CDP to 0: offset in the buffer
  - Set up CDP in circular addressing mode

```
s1: AMOV   #x,XAR2
    AMOV   #a0,XCDP
    AMOV   #(x+1),XAR3
    MOV    #a0,BSAC
    MOV    #0,CDP
    MOV    #4,BKC
    BSET   CDPLC
```

Store Results, 32-bit Moves

- Assuming fractional mode, 2 results are in high parts of AC0 and AC1
- AC0 and AC1 can be saved separately:

```
MOV HI(AC0), *AR4+
MOV HI(AC1), *AR4+
```

  - AC0, AC1 can be saved at the same time:

```
MOV pair(hi(AC0)),dbl(*AR4+)
```

    - Pairs: (AC0,AC1), (AC2,AC3)
    - ARi incremented by 2

Store Results, 32-bit Moves

```
MOV pair(hi(AC0)),dbl(*AR4+)
```

- Previous command only possible in case output (y) is even aligned
  - In linker command file:

```
output: {} < RAM0 align 4
```

  - In C-file:

```
#pragma DATA_ALIGN(y,2);
```

Block Filter Inner Loop

```
s1:     AMOV   #x,XAR2
        AMOV   #a0,XCDP
        AMOV   #(x+1),XAR3
        AMOV   #y,XAR4
        MOV    #a0,BSAC
        MOV    #0,CDP
        MOV    #4,BKC
        BSET   CDPLC

        MOV    #0,AC0
        MOV    #0,AC1
        RPT    #3
        MAC    *AR2+,*CDP+,AC0
        ::MAC    *AR3+,*CDP+,AC1
        ASUB   #2,AR2
        ASUB   #2,AR3
e1:     MOV    pair(hi(AC0)),dbl(*AR4+)
```

Outer Loop Using RPTB or RPTBlocal

- Use RPTB Repeat Block instruction
- We must specify:
  - Start address of the block: next instruction
  - End address: label specifies last instruction
  - The number of repetitions counter:
    - BRC0: loop counter initialized with count-1
    - Min count = 2
- RPTBlocal: executes from the IBU
  - 56 bytes maximum (if > 56 Bytes use RPTB)
  - Reduces power consumption

Outer Loop on m: Calculate M yn-m

```
s1:     AMOV   #x,XAR2
        AMOV   #a0,XCDP
        AMOV   #(x+1),XAR3
        AMOV   #y,XAR4
        MOV    #a0,BSAC
        MOV    #0,CDP
        MOV    #4,BKC
        BSET   CDPLC
        MOV    #((samps-taps)/2),BRC0
        RPTBLOCAL e1
        MOV    #0,AC0
        MOV    #0,AC1
        RPT    #3
        MAC    *AR2+,*CDP+,AC0
     :: MAC    *AR3+,*CDP+,AC1
        ASUB   #2,AR2
        ASUB   #2,AR3
e1:     MOV    pair(hi(AC0)),dbl(*AR4+)
```

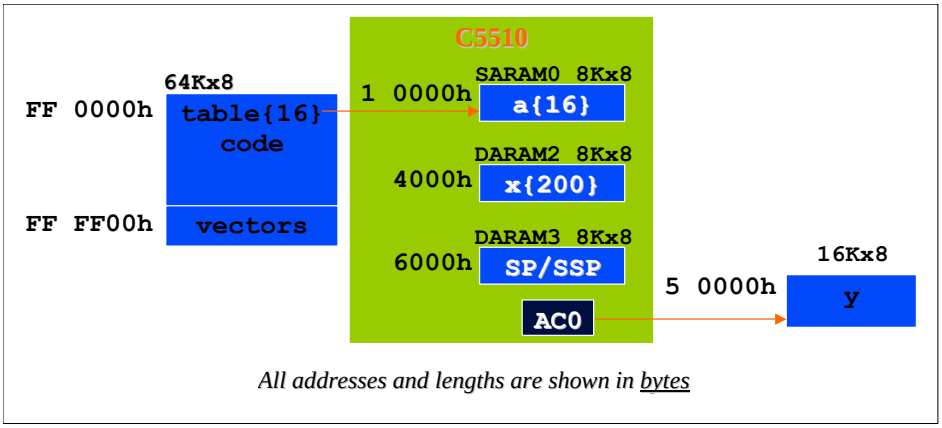
More Nested loops ?

- Nesting RPTB or RPTBlocal:
  - 2 levels supported using BRC0 (outer) and BRC1/BRS1 (inner)
  - No saving of registers required for nested block repeat.

```
        MOV  #outer_cnt,BRC0   ;load outer loop count
        MOV  #inner_cnt,BRC1   ;load BRC1, auto-load BRS1
        RPTBLOCAL outer        ;use BRC0
        . . .
        RPTBLOCAL inner        ;BRC1: decrements, BRS1-no change
        . . .
inner:  last_inner
        . . .
outer:  last outer
```

EXAMPLE on Block Filter

- Implement a block FIR with 16 coefficients and input block size = 200.
- Implement subroutine



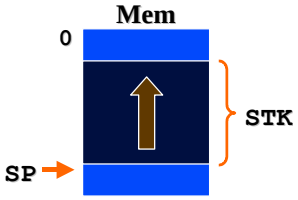
Using the Stack and Subroutines

- Subroutines require call and ret.
- During a call the return address is stored on the Stack (SP).
- Let us call fir the subroutine:
  - call fir

Initialize the Stack

- Declare an unitialized section (.usect) of appropriate length to reserve space.
- Initialize stack pointer to point to the top of stack +1.
- Recommendation: place the stack in internal memory and align on a 4-byte boundary:
  - ALIGN= specifies bytes

```
Size   .set   100h
Stack  .usect "STK",size
       AMOV   #(Stack+Size),XSP
```



The System Stack SSP

- When a call occurs PC[15:0] is pushed on the stack
- The upper 8 bits XSP[23:16] are pushed on the system stack accessed by SSP System Stack Pointer.
- CFCT is used to store the active loop context.
- XSP and XSSP share the same upper 7 bits.
- Place SP and SSP with care to avoid dual-access delays.

Data Types

- Byte: 8 bits
- Word: 16 bits
- Long: 32 bits
  - Long access assumes address points to MSW
    - LSW read from same address with LSB *toggled*.
    - Ptr=100h, MSW=100h, LSW = 101h
    - Ptr=101h, MSW=101h, LSW = 100h
- To ensure proper alignment:
  - Constants (int, long) are automatically aligned on type boundaries
  - Variables:
    - 16 bit: no problem
    - 32 bits use: use the even-align flag:
      - .usect "vars",Nwords,,1

Example code: Declarations

```
            .sect "indata"
x0          .copy  in7.dat
            .def start
            .cpl_off
            .arms_off
            .c54cm_off

stklen      .set   100
a0          .usect "coeffs",16,1,1
y0          .usect "results",200,1,1
BOS         .usect "STK", stklen,1,1
BOSS        .usect "SSTK",stklen,1,1

            .sect "init"

table       .int 7FCh, 7FDh, 7FEh, 7FFh
            .int 800h, 801h, 802h, 803h
            .int 803h, 802h, 801h, 800h
            .int 7FFh, 7FEh, 7FDh, 7FCh
```

Example code : Code

```
          .sect "code"
          .DP  a0

start:    AMOV #BOS+stklen,XSP  ;set up Stack +
          MOV  #BOSS+stklen,SSP ;System Stack Ptrs

          CALL copy             ;copy coeffs

          BSET FRCT        ;turn on mult. shift
          BSET M40         ;turn on 40 bit math
          BSET SXMD        ;turn on sign exten.

          CALL fir         ;perform fir
          nop
here:     B    here        ;stop
```

Example code : Subroutine copy

```
copy:     AMOV #table,XAR2     ;load pointers
          AMOV #a0,XAR3
          RPT  #7
          MOV  dbl(*AR2+),dbl(*AR3+)
;move from table to a
          RET
```

Example code : Subroutine fir

```
fir:  MOV  #92,BRC0    ;block repeat count
      AMOV #x0,XAR2    ;initialize pointers
           AMOV #x0+1,XAR3  ;for data,
           AMOV #y0,XAR4    ;results
           AMOV #a0,XCDP    ;and coefficients
           MOV  #a0,BSAC    ;buffer start address
           MOV  #16,BKC     ;buffer size
           MOV  #0, CDP     ;index
           BSET CDPLC ;turn on circ addr CDP

           RPTBlocal end
            MPYM *AR2+,*CDP ,AC0 ;AC0 1st product
            MPYM *AR3+,*CDP+,AC1 ;AC1 gets 2nd prod
            RPT  #14
             MAC  *AR2+,*CDP+,AC0      ;form results
             :: MAC  *AR3+,*CDP+,AC1
            MOV pair(hi(AC0)),dbl(*AR4+)     ;store AC0/AC1
            ASUB #14,AR2   ;wrap data pointers
end         ASUB #14,AR3    ;next calculation
           RET
```

DSP C5000

Infinite Impulse Response (IIR)
Filter Implementation

IIR Filters

- Rational Z transfer function

$$H(z) = \frac{N(z)}{D(z)} = \frac{\sum_{i=0}^{Q} b_i z^{-i}}{1 + \sum_{k=1}^{P} a_k z^{-k}}$$

- Linear difference equation

$$y_n = \sum_{i=0}^{Q} b_i x_{n-i} - \sum_{k=1}^{P} a_k y_{n-k}$$

IIR Filters – Poles and Zeros

- Roots of the numerator

$$\sum_{i=0}^{Q} b_i z^{-i} = b_0 \prod_{i=0}^{Q} \left(1 - r_i z^{-1}\right)$$

r_i are the roots of the z polynomial with b_i coefficients. $H(z)$ is null when z is equal to one of the values. They are called the zeroes of the filter and often noted by z_i.

- Roots of the denominator

$$1 + \sum_{i=1}^{P} a_i z^{-i} = \prod_{i=0}^{P} \left(1 - r_i z^{-1}\right)$$

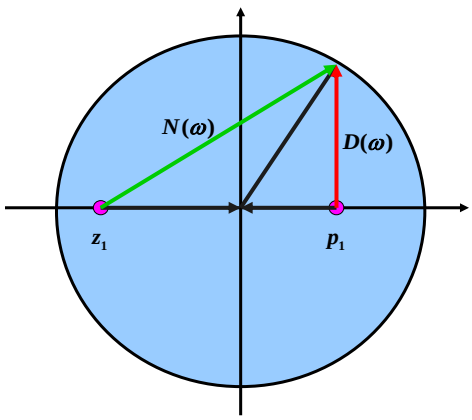
r_i are the roots of the z polynomial with a_i coefficients. $H(z)$ tends to infinity when z is close to one of these values. They are called the poles of the filters and often noted by p_i.

Z Transfer Function

- Define frequency behaviour of the filter
  - Consider a first order z rational filter:

$$H(z)\Big|_{z=e^{j\omega T_e}} = \frac{1 + b_1 z^{-1}}{1 + a_1 z^{-1}}\Bigg|_{z=e^{j\omega T_e}} = \frac{z - z_1}{z - p_1}\Bigg|_{z=e^{j\omega T_e}} = \frac{|N(\omega)|e^{j\phi_N}}{|D(\omega)|e^{j\phi_D}} = \frac{|N(\omega)|}{|D(\omega)|} e^{j(\phi_N - \phi_D)}$$

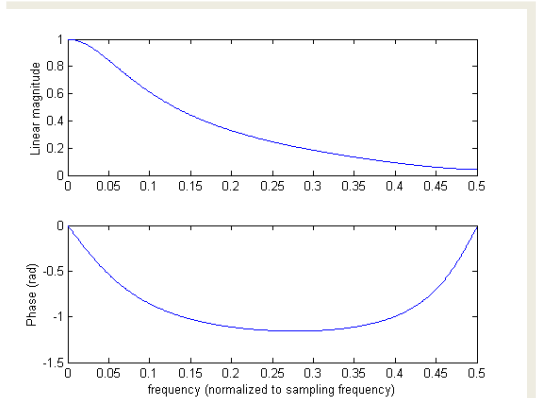
magnitude phase



$H(z)$ can be evaluated for each value ω_n from 0 to 1 with 1 corresponding to sampling frequency

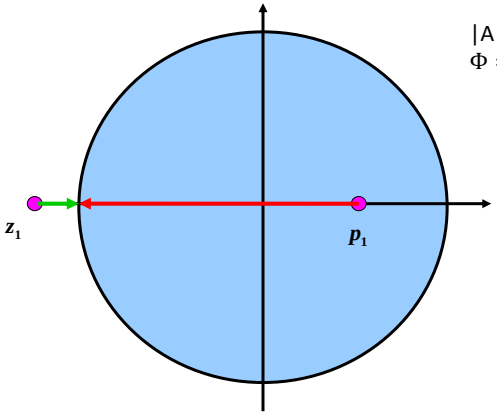
Z Transfer Function

- We obtain the transfer function by evaluation of the z transform on the unit circle



- We can see that it is a minimum phase filter (the phase comes back at 0 at $F_e/2$) because the zero of the filter is inside the unit circle.

Unit circle for maximum phase filter

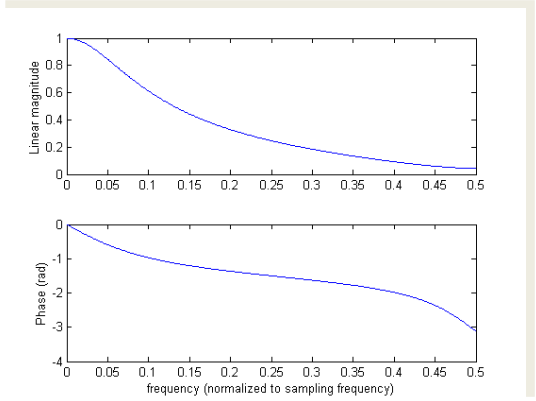


|A| → small

$\Phi = \Phi_N - \Phi_D = 0 - \Pi$

Z Transfer Function

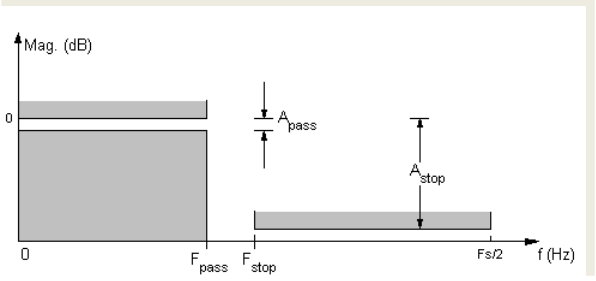
- If we change the zero z_1 to $1/z_1$ we get the same magnitude transfer function (up to a scale factor) …



- But a maximum phase filter (the phase goes to $-\pi$ at $F_e/2$) because now, the zero lies outside the unit circle.

IIR Filter Synthesis

– Starting from frequency specifications (here low pass filter):



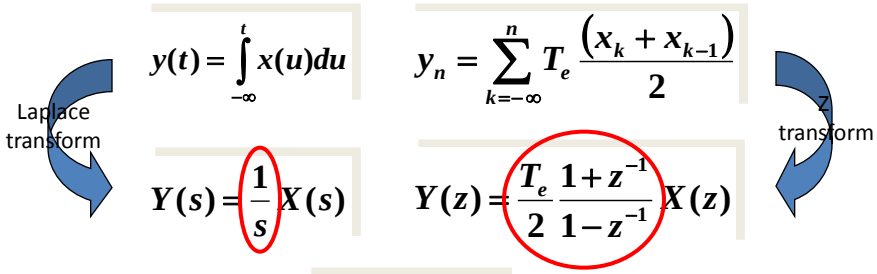
- F_{pass} : passband end frequency,
- F_{stop} : stopband start frequency,
- A_{pass} : maximum passband ripple,
- A_{stop} : minimum stopband attenuation.

IIR Filters Synthesis

- Analog prototype with analog to digital transformation (bilinear transform) :
  - Digital to analog (cut-off) frequency specification transformation using prewarping
  - Analog filter prototype
  - Analog transfer function to digital transfer function transformation using bilinear transform.
- Direct digital method : Yule Walker
  - Try to find the recursive filter of order *N* which is as close as possible to the frequency specifications using the least square optimization method.

IIR Filters Synthesis

- Bilinear transform :
  - One to one map of analog frequencies to digital frequencies.
  - Based on the approximation of the continuous integral operator by the trapezoïdal method.



Equating integral operators, we get the bilinear transform.

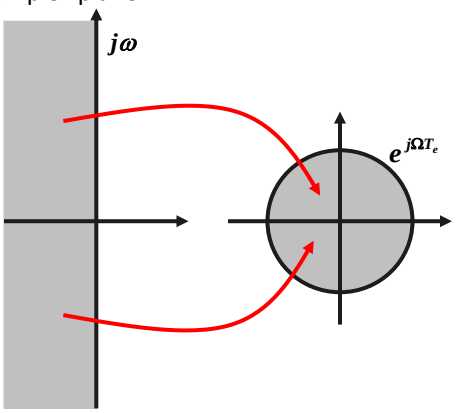
$$s = \frac{2}{T_e} \frac{1-z^{-1}}{1+z^{-1}}$$

IIR Filters Synthesis

- Bilinear transform
  - Maps the stability region of the Laplace plane inside the unit circle of the complex plane

$$s\Big|_{s=j\omega} = \frac{2}{T_e}\frac{1-z^{-1}}{1+z^{-1}}\Bigg|_{z=e^{j\Omega T_e}}$$

$$\omega = \frac{2}{T_e}\tan\left(\frac{\Omega T_e}{2}\right)$$

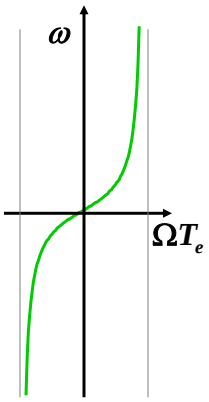


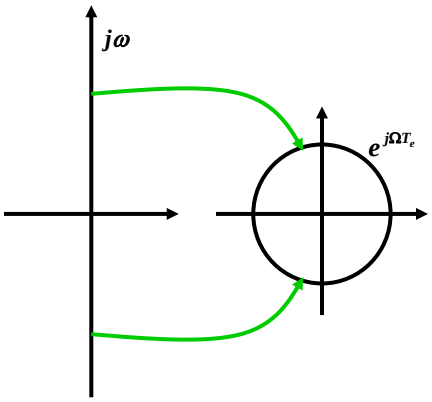
IIR Filters Synthesis

- Bilinear transform :

$$\omega = \frac{2}{T_e}\tan\left(\frac{\Omega T_e}{2}\right)$$

The one to one mapping achieved by this transform prevents aliasing.





IIR Filters Synthesis

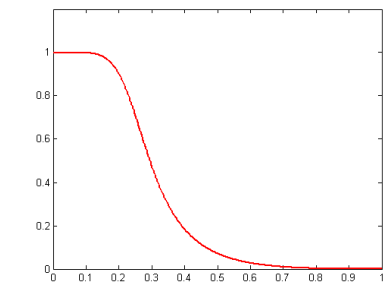
- Characteristics frequencies (F_p, F_a) of the target specifications have to be warped.
- This warped specifications is used to compute an analog prototype using approximation functions :
  - » Butterworth
  - » Chebyshev I
  - » Chebyshev II
  - » Elliptic
- Then the analog prototype is transformed into a digital filter that matches target frequency specification thanks to Bilinear Transform (BT) (this cancels the warping introduce at the first step).

IIR Characteristics

– Butterworth filters :

$$|H(f)|^2 = \frac{1}{1+\left(\frac{f}{f_p}\right)^{2N}}$$

- Defined by its order N and its cut-off frequency f_p.
- Monotonic magnitude transfer function at both pass and attenuation band with slow role-off.



Matlab commands:
- *buttord* : estimate the needed order
- *butter* : compute the digital filter from analog prototype using warping and BT, given the order and cut-off frequency.

IIR Characteristics

– Butterworth filters :

```
% Butterworth filter
% sampling frequency : 16 kHz
% Fp : 1800 Hz
% Fa : 4000 Hz
% Rp : 0.01 dB
% Ra : 50 dB

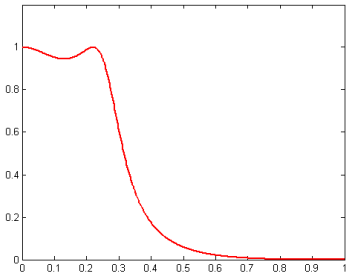
[N, Wn] =
buttord(1800/8000,4000/8000,0.01,50);
[B,A] = butter(N,Wn);
H=freqz(B,A);
plot(linspace(0,1,512),abs(H))
```

IIR Characteristics

– Chebyshev I filters :

$$|H(f)|^2 = \frac{1}{1+\varepsilon^2 T_N{}^2\left(\frac{f}{f_p}\right)}$$

• Defined by its order *N*, its passband corner frequency f_p and its passband ripple ε.
• Ripple in passband and monotonic in stopband. Given the same number of poles sharper cutt-off than Butterworth.



Matlab commands:
• *cheb1ord* : estimate the needed order
• *cheby1* : compute the digital filter from analog prototype using warping and BT, Given the order and passband ripple and Corner frequency.

$T_N(\ )$ is a Chebyshev polynomial of order *N*

IIR Characteristics

– Chebyshev I filters :

```
% Chebyshev I filter
% sampling frequency : 16 kHz
% Fp : 1800 Hz
% Fa : 4000 Hz
% Rp : 0.01 dB
% Ra : 50 dB

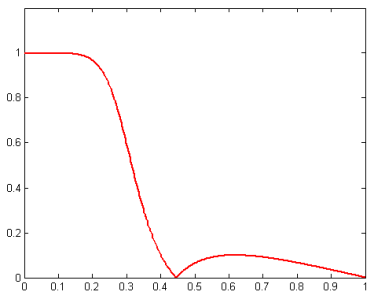
[N, Wn] =
cheb1ord(1800/8000,4000/8000,0.01,50);
[B,A] = cheby1(N,0.01,Wn);
H=freqz(B,A);
plot(linspace(0,1,512),abs(H))
```

IIR Characteristics

– Chebyshev II filters :

$$|H(f)|^2 = 1 - \frac{1}{1+\varepsilon^2 T_N{}^2\left(\frac{f_s}{f}\right)}$$

- Defined by its order *N*, its stopband edge frequency f_s and its stopband attenuation ε.
- Monotonic in passband and ripple in stopband.



Matlab commands:
- *cheb2ord* : estimate the needed order
- *cheby2* : compute the digital filter from analog prototype using warping and BT, Given the order and stopband attenuation and edge frequency.

$T_N(\,)$ is a Chebyshev polynomial of order *N*

IIR Characteristics

– Chebyshev II filters :

```
% Chebyshev II filter
% sampling frequency : 16 kHz
% Fp : 1800 Hz
% Fa : 4000 Hz
% Rp : 0.01 dB
% Ra : 50 dB

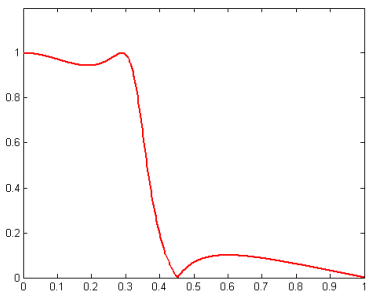
[N, Wn] =
cheb2ord(1800/8000,4000/8000,0.01,50);
[B,A] = cheby2(N,50,Wn);
H=freqz(B,A);
plot(linspace(0,1,512),abs(H))
```

IIR Characteristics

– Elliptic filters :

$$|H(f)|^2 = \frac{1}{1+\varepsilon^2 R_N{}^2\left(\frac{f}{\sqrt{f_p f_s}}\right)}$$

- Defined by its order *N*, its passband and stopband edge frequencies, f_p and f_s, its passband ripple and its stopband attenuation ε.
- Ripple in passband and in stopband sharpest roll-off.



Matlab commands:
- *ellipord* : estimate the needed order
- *ellip* : compute the digital filter from analog prototype using warping and BT, given the order, passband ripple, stopband attenuation and center frequency.

$R_N(\,)$ is a Chebyshev rationnal polynomial of order *N*

IIR Characteristics

– Elliptic filters :

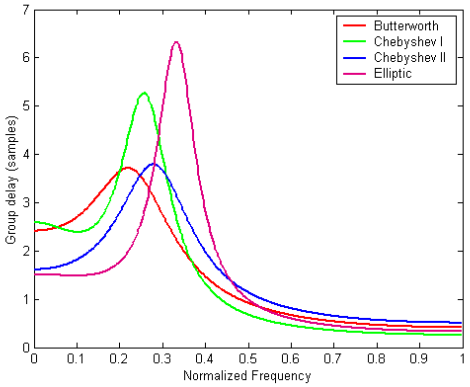
```
% Elliptic filter
% sampling frequency : 16 kHz
% Fp : 1800 Hz
% Fa : 4000 Hz
% Rp : 0.01 dB
% Ra : 50 dB

[N, Wn] =
ellipord(1800/8000,4000/8000,0.01,50);
[B,A] = ellip(N,0.01,50,Wn);
H=freqz(B,A);
plot(linspace(0,1,512),abs(H))
```

IIR Characteristics

- Group delay
  - Characterize the phase distorsion (waveform distorsion) introduced by the filter.

$$\tau(\omega) = -\frac{\partial \phi(\omega)}{\partial \omega}$$

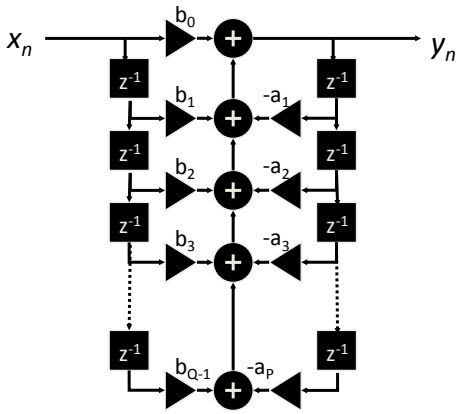


IIR structure

- Derived from difference equation

Direct form I

$$y_n = \sum_{i=0}^{Q-1} b_i x_{n-i} - \sum_{k=1}^{P} a_k y_{n-k}$$

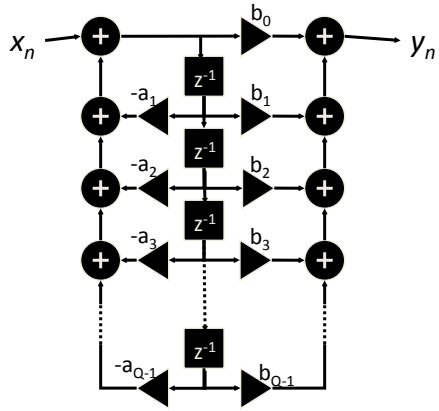


$$\boldsymbol{H(z) = B(z)\frac{1}{A(z)}}$$

- Non canonical form
- Q+P coefficients, Q+P+1 memory locations to store $\{x_n(n-l), l = 0,1, \dots, Q-1\}$ and $\{y(n-p), p = 1, \dots, P\}$

IIR structure

Direct form II

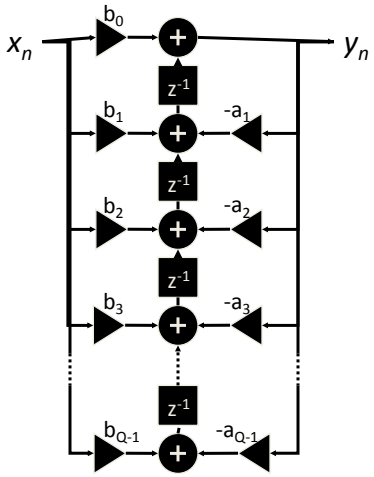


$$H(z) = \frac{Y(z)}{W(z)}\frac{W(z)}{X(z)}$$

- Multiplication is commutative, hence factors can be changed position
- Canonical form since minimum numbers of memory location are necessary (Q-1 memory locations are required)

IIR structure

Transposed direct form II



$$H(z) = \frac{1}{A(z)} B(z)$$

- Canonical form

IIR – Coefficients quantization

- Finite precision of DSP involves coefficients quantization:

Let consider the denominator of the transfer function with $\bar{a}_k = a_k + \Delta a_k$ the k^{th} quantized coefficients and Δa_k the quantification error. Quantized denominator is then:

$$\bar{A}(z) = 1 + \sum_{k=1}^{Q-1} \bar{a}_k z^{-k} = \prod_{l=1}^{Q-1} \left(1 - \bar{p}_l z^{-1}\right)$$

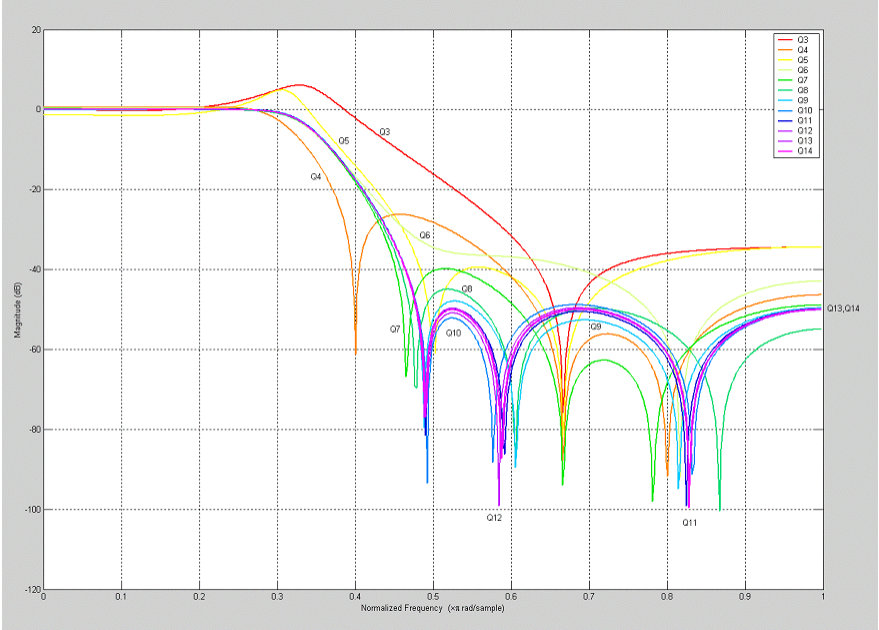
The resulting quantified poles will disrupt the transfer function.

The higher order the polynomial is, the greater will be pertubation on its roots due to quantization.

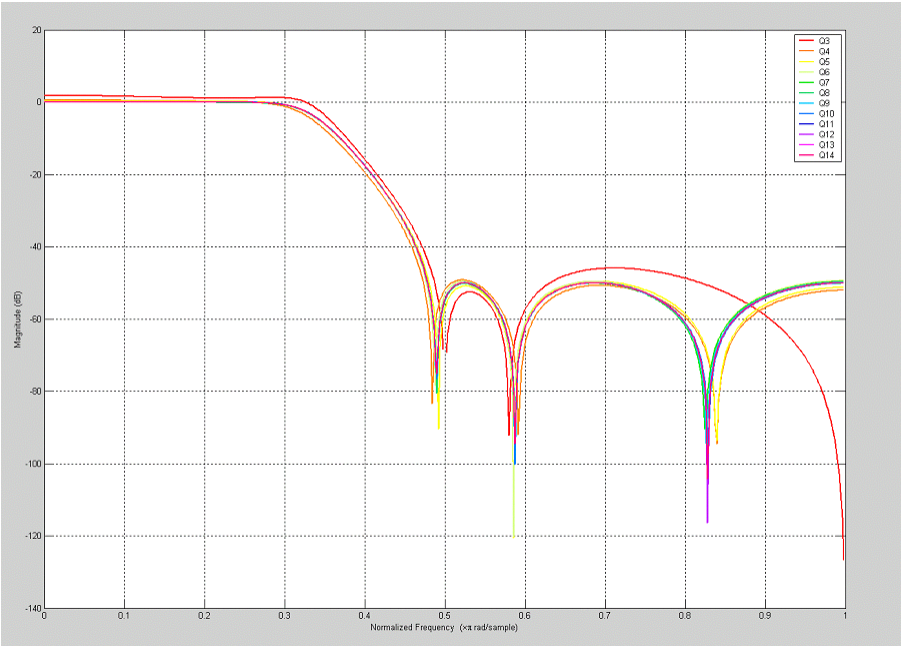
Following slides illustrate this fact:

- Next slide shows the transfer function of a 6th order direct form filter for different quantification.
- Following one shows the transfer function obtained for the same filter and same quantification, but with a cascade structure of second order section, this last structure is much less sensitive to quantization than the previous one.

Direct structure



Cascade structure of second order section

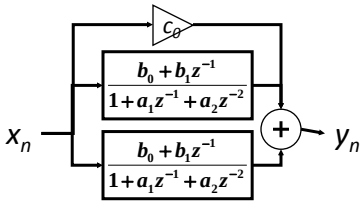


IIR structure

- This sensitivity to coefficients quantization leads to second order cascade or parallel form.
- Second order section is chosen to get the least order together with complex conjugated roots.

4th order example:

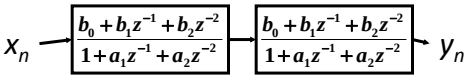
Parallel form



Partial fraction expansion

$$\frac{B(z)}{A(z)} = c_0 + \sum_{i=0}^{N-1} \frac{b_{i0}+b_{i1}z^{-1}}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}$$

Cascade form



Spectral factorisation

$$\frac{B(z)}{A(z)} = b_0 \prod_{i=0}^{N-1} \frac{1+b_{i1}z^{-1}+b_{i2}z^{-2}}{1+a_{i1}z^{-1}+a_{i2}z^{-2}}$$

IIR – Cascade structure

– Cascade structure involves addressing two problems :

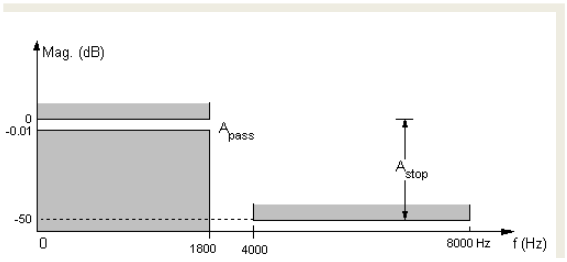
- Pairing: which zeros with which poles to form a second order rational transfer function.

  The goal will be minimize the overshoot caused by the poles.
- Ordering: which second order section will be ahead and which one will be the last.

  To answer to this question we will have consider quantification noise and the way to minimize it.

IIR – case study (1)

– Consider the following specification:



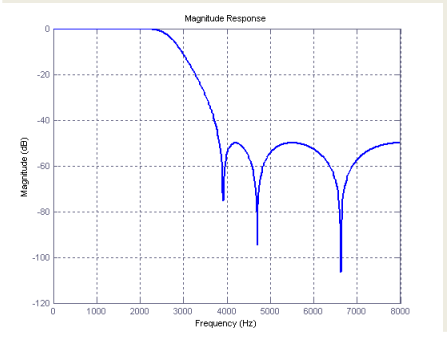
– Using inverse Chebyshev approximation, we get a 6th order filter (matlab commands)

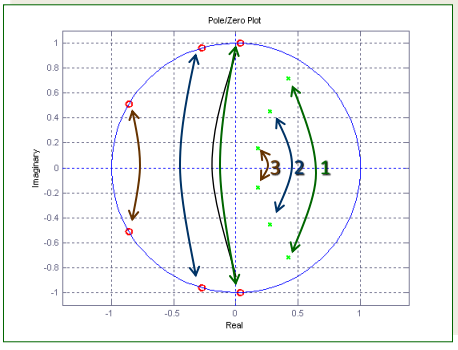
```
[N,Wn]=CHEB2ORD(1800/8000,4000/8000,0.01,50);
[B,A]=CHEBY2(N,50,Wn)
```

IIR – case study (2)

Actual transfer function is obtained with:
`freqz(B,A)`

Plot of poles and zeros with:
`zplane(A,B)`





Pairing: complex conjugate poles closest to the unit circle (responsible for the greatest overshoot) are paired with complex conjugate zeros closest in frequency (angle on unit circle). Then the process iterate with the next complex conjugate closest to the unit circle.
This done with the routine on the next slide

IIR – case study (3)

```
% roots extraction
P=roots(A)
Z=roots(B)

% create 2nd order polynomials from complex
conjugated roots
A1=poly(P(1:2))
B1=poly(Z(3:4))
A2=poly(P(3:4))
B2=poly(Z(5:6))
A3=poly(P(5:6))
B3=poly(Z(1:2))

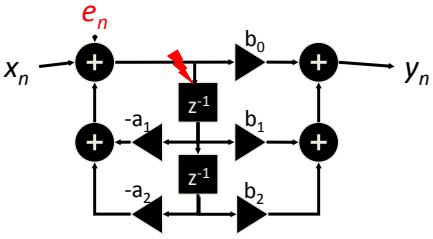
% This could be done automaticaly with
matlab function
% [sos,g]=tf2sos(B,A);
```

IIR – data quantization

For DSP, quantification noise appear when we truncate the accumulator to store its high part. (e_n equivalent noise source)

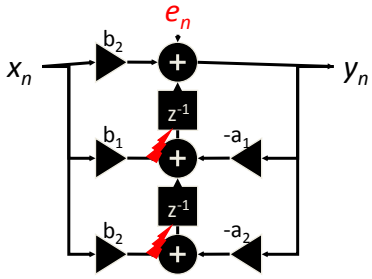
Direct form II

one noise source



Transposed direct form II

two noise sources



Output noise power is reduced by the *ENB* of the complete filter

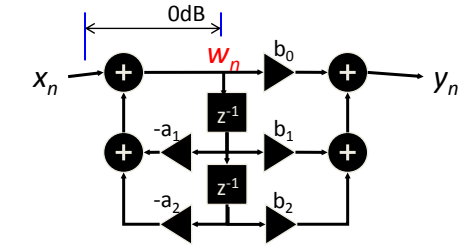
$$\sigma^2_{e_out} = \sigma^2_e \frac{1}{F_e} \int_{-F_e/2}^{F_e/2} \frac{B(f)}{A(f)} df$$

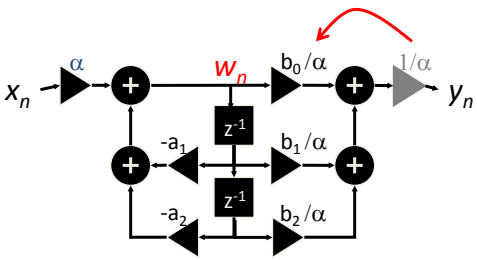
Output noise power is only reduced by the *ENB* of the denominator

$$\sigma^2_{e_out} = \sigma^2_e \frac{1}{F_e} \int_{-F_e/2}^{F_e/2} \frac{1}{A(f)} df$$

IIR – scaling factor

- Direct form II





To prevent overflow when storing at the node w_n, we need a scale factor to have a 0 dB gain from input to this node.
This scale factor, α, is commonly computed depending on the nature of signal that will be process:
Narrow band signal, in this case we use L_∞ norm and we get:

$$\alpha_1 = \frac{1}{\|1/A(f)\|_\infty}$$

Wide band signal, we use L_2 norm and we get:

$$\alpha_2 = \frac{1}{\|1/A(f)\|_2}$$

Futhermore we have:

$$\alpha_2 \geq \alpha_1$$

IIR – ordering

- Depend on:
  - Criteria for scale factor computation, L_∞ or L_2 norm.
  - Which norm of the quantization noise we want to minimize L_∞ (max value) or L_2 norm (power).

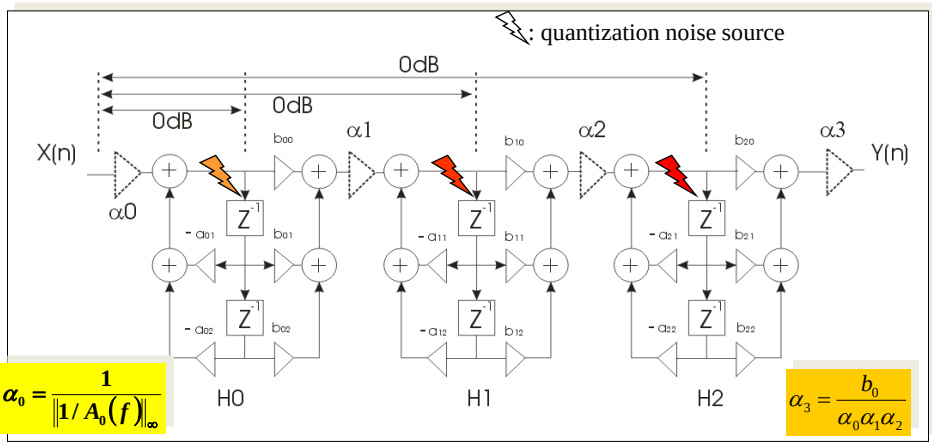
– Following rules could apply:

• L_∞ for scale factor and L_2 for noise ➡ ascending order of overshoot. (minimizes the probability of overflow)
• L_2 for scale factor and L_∞ for noise ➡ descending order of overshoot. (minimizes the peak quantization noise)

If the same norm is used ➡ no prefered order.

Scale factor & ordering

– 6th filter example (look at this routine)



$$\alpha_0 = \frac{1}{\|1/A_0(f)\|_\infty} \qquad \alpha_3 = \frac{b_0}{\alpha_0\alpha_1\alpha_2}$$

$$\alpha_1 = \frac{1}{\alpha_0\|B_0(f)/(A_0(f)A_1(f))\|_\infty} \qquad \alpha_2 = \frac{1}{\alpha_0\alpha_1\|B_0(f)B_1(f)/(A_0(f)A_1(f)A_2(f))\|_\infty}$$

Scale factor & ordering

– 6th filter example

```
sf=1/max(abs(freqz(1,A1)))
sf2=1/max(sf*abs(freqz(B1,A1)).*abs(freqz(1,A2)))
sf3=1/max(sf*sf2*abs(freqz(B1,A1)).*abs(freqz(B2,A2)).*abs(freqz(1,A3)))
sf4=B(1)/(sf*sf2*sf3);
% or (that would be the same)
% sf4=1/max(sf*sf2*sf3*abs(freqz(B1,A1)).*abs(freqz(B2,A2)).*abs(freqz(B3,A3)))
B1=B1*sf2
B2=B2*sf3
B3=B3*sf4
```

IIR – coefficients coding

- We have to choose the right *Qn* coding for the coefficients.
  - For second order section, *A(z)* or *B(z)* can be written, if we let their roots to be $r_p e^{\pm j\theta}$ as

$$H(z) = \frac{A}{(1 - r_p e^{j\theta} z^{-1})(1 - r_p e^{-j\theta} z^{-1})} = \frac{\mathrm{A}}{1 - 2r_p \cos(\theta) z^{-1} + r_p^2 z^{-2}}$$

  - Denominator: for stability we need $|r_p| < 1$, so coefficients belong to [-2,2].
  - Numerator: same reasoning goes here.
- The right coding is then *Q14* for 16 bits word.

IIR – noise power

- Input scale factor attenuates only the input signal not the quantization noise
- The lower the scale factor is the worst the SNR will be.

IIR - computation

- Evaluation of an order 2 direct form II is as follows:

```
ACC=x(n) × α0
ACC=ACC - a1 × w(n-1)
ACC=ACC - a2 × w(n-2)
ACC<<2
w(n)=ACCH
ACC=w(n) × b0
ACC=ACC + b2 × w(n-2)
ACC=ACC + b1 × w(n-1)
ACC<<2
y(n)=ACCH
```

```
Q29=Q15 × Q14
Q29=Q29 - (Q15 × Q14)
Q29=Q29 - (Q15 × Q14)
Q31=Q29 × 2^2
Q15
Q29=Q15 × Q14
Q29=Q29 + (Q15 × Q14)
Q29=Q29 + (Q15 × Q14)
Q31=Q29 × 2^2
Q15
```

```
w(n-2)=w(n-1)
w(n-1)=w(n)
```

Adaptive filter implementation

DSP C5000

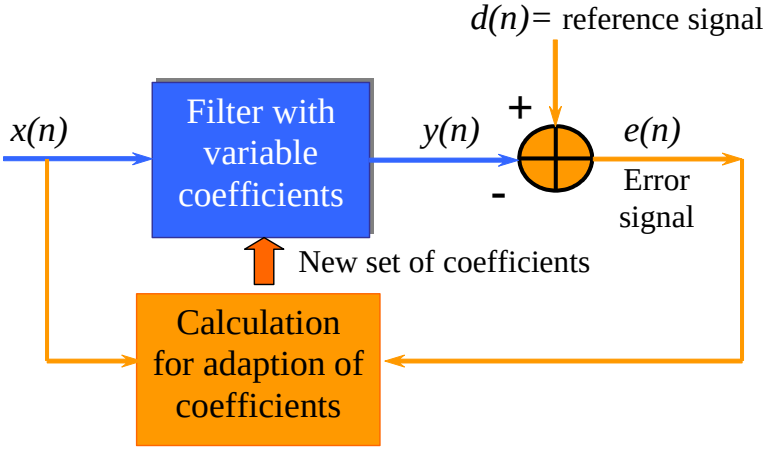
Outline

- Brief review of adaptive filters and LMS implementation
- Implementation of adaptive FIRs on C55xx

Generalities of Adaptive Filters

- The coefficients of Adaptive filters are variable with time in order to optimize a given criterion.
- The most commonly used algorithm to adapt the coefficients is the LMS (Least Mean Square) algorithm.
- Most adaptive filters are implemented as FIR filters, because they are inherently stable.
- They are widely used in digital communications:
  - Echo cancellation, equalizers ...

Diagram and Notations of Adaptive Filters



Mean Square Error Solution

$y_n = \sum_{i=0}^{N-1} w_i x_{n-i}$ (FIR filter)

$$e_n = d_n - y_n$$

Objective function: $\min_w J = \min_w E(e_n^2)$

$$\frac{\partial J}{\partial w_i} = -2E(e_n x_{n-i})$$

$$= -2E(d_n x_{n-i}) + 2\sum_{k=0}^{N-1} w_k E(x_{n-k} x_{n-i})$$

Optimum: $\boldsymbol{R_x b = d_{xr}}$

Gradient Algorithm

- Gradient of criterion J vs b:

$$\frac{\partial J}{\partial b_i} = -2E(e_n x_{n-i})$$

Gradient algorithm: iterative solution

$$\forall i \in [0, N-1], w_i(n+1) = w_i(n) + \delta E(e_n x_{n-i})$$

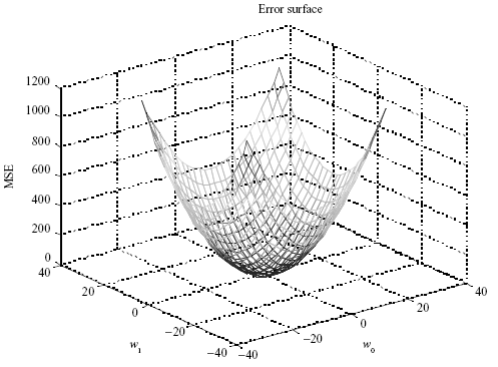
Converges if:

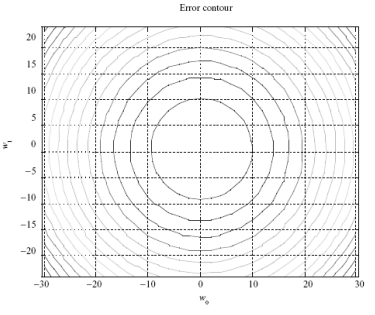
$$|\delta| < \frac{2}{\lambda_{\max}}.$$

Adaptation step

Max Eigenvalue of R_x

Gradient Algorithm: visualize objective function





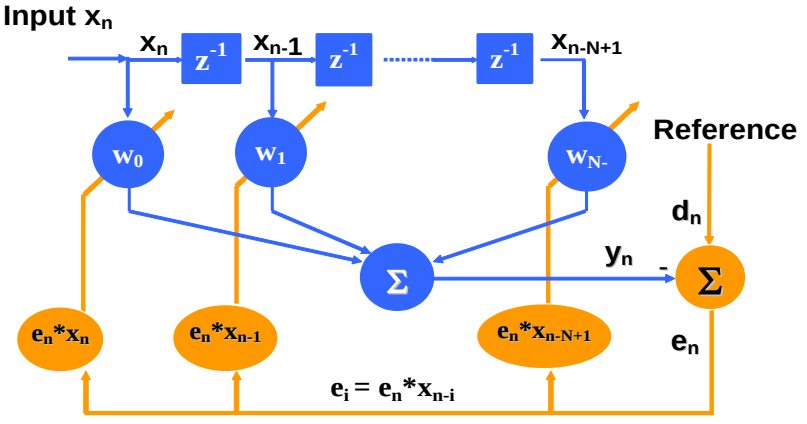
Stochastic Gradient Algorithm: LMS

- The mean values E(e(n)x(n-i)) are not known.
  - In the stochastic gradient algorithm, they are replaced by e(n)x(n-i).
  - The algorithm converges if the adaptation step is small enough.
  - Algorithm named: LMS (Least Mean Square) or Widrow algorithm:

$$\forall i \in [0, N-1] w_i(n+1) = w_i(n) + \delta e_n x_{n-i}$$

LMS Algorithm

- For each sample, the LMS algorithm:
  - Filters the input using w_i
  - Updates the w_i coefficients.



LMS Algorithm at Each Sample Time

- FIR Filtering equation:

$$y_n = \sum_{i=0}^{N-1} w_i x_{n-i}$$

Coefficient updating equation:

$$\forall i \in [0, N-1] w_i(n+1) = w_i(n) + \delta e_n x_{n-i}$$

With: $e_n = d_{n-1} - y_{n-1}$

Fixed Point Implementation

- When LMS adaptive filter is implemented on a fixed point DSP:
  - The precision of calculation is important:
    - If rnd(e_i*x_{n-i}) is smaller than the used precision, no adaptation is performed.
- The precision of convergence depends on:
  - Adaptation step δ: the largest δ, the fastest convergence but the worst precision.
  - The number of coefficients N: the residual error is proportional to N.
    - When N is too large, it may be worth considering using block adaptive filtering.

LMS steps

- Each time iteration (only once):
  - Calculates error $e_n = d_n - y_n$
  - Scale error by adaptation step δ: $e'_n = \delta e_n$.
- Each time iteration, for each coefficient:
  - Multiply error with signal: $e_i = e'_n x_{n-i}$
  - Multiply $x_{n-i} w_i$ and accumulate
  - Calculate new coefficients: $neww_i = w_i + e_i$
  - Update coefficients: $w_i = neww_i$.

Implementating LMS algorithm on C55x

- LMS specific instruction to realize:
  - Filter and coefficient updating.
  - ACy = ACy + (w_i*x_{n-i}); ACx = rnd(e_i+w_i)
  - Rounding is important because δ may be very small.

Example of LMS adaptive filter on C55X

```
      …
      .asg     AR3, Coefs
      .asg     AR4, Data
```

Pre-calculate δ*e(n) … .
AR3 pts to coeff table: w[n]
AR4 pts to data table: x[n]

```
      MOV      @DeltaE,T3
   ||MOV       #(N-2),BRC0
      MPYM     *Data,T3,AC0
   ||MOV       #0,AC1
      MOV      port(@X_new),*Data
      LMS      *Coefs,*Data,AC0,AC1
```

T3 holds error step amount
… while loading BRCO
AC0=error*oldestsample: x(n-N)
…while clearing AC1 (run FIR)
Overwrite x(n-N) with x(n)
Start FIR, update oldest coef…

```
   ||RPTBLOCAL  e1
      MOV      HI(AC0),*Coefs+
   ||MPYM      *Data+,T3,AC0
e1:   LMS      *Coefs,*Data,AC0,AC1
      MOV      HI(AC0),*Coefs
   ||MOV       HI(AC1),*Result+
```

… and start repeat block
Store update coefficient ...
...while calc. next update term
Calc FIR, update coefficient
Store final coefficient...
…while storing FIR output

DSP C5000

C55x
Interrupt Management

Objectives

- **Analyze the Interrupt Timeline for the C55x the order of events related to interrupts**
- **Describe other interrupt issues (reset, relocating vectors, priority, etc)**

Interrupt Timeline

INIT

1. Select interrupt sources
2. Create interrupt vector table
3. Enable individual interrupts
4. Enable Global Interrupts

CPU

5. Valid signal/Flag bit set
6. Int enabled? ⟶ branch to ISR

ISR

7. Context save/ISR/context restore
8. Return to main program

1. C55 Interrupt Sources

NAME	SOFTWARE (TRAP) EQUIVALENT	RELATIVE LOCATION (HEX BYTES) (1)	PRIORITY	FUNCTION
RESET	SINT0	0x0	0	Reset (hardware and software)
NMI(2)	SINT1	0x8	1	Non-maskable interrupt
INT0	SINT2	0x10	3	External user interrupt #0
INT1	SINT3	0x18	5	External user interrupt #1
TINT	SINT4	0x20	6	Timer aggregated interrupt
PROG0	SINT5	0x28	7	Programmable transmit interrupt 0 (I2S0 transmit or MMC/SD0 interrupt)
UART	SINT6	0x30	9	UART interrupt
PROG1	SINT7	0x38	10	Programmable receive interrupt 1 (I2S0 receive or MMC/SD0 SDIO interrupt)
DMA	SINT8	0x40	11	DMA aggregated interrupt
PROG2	SINT9	0x48	13	Programmable transmit interrupt 1 (I2S1 transmit or MMC/SD1 interrupt)
-	SINT10	0x50	14	Software interrupt
PROG3	SINT11	0x58	15	Programmable receive interrupt 3 (I2S1 Receive or MMC/SD1 SDIO interrupt)
LCD	SINT12	0x60	17	LCD interrupt
SAR	SINT13	0x68	18	10-bit SAR A/D conversion or pin interrupt
XMT2	SINT14	0x70	21	I2S2 transmit interrupt
RCV2	SINT15	0x78	22	I2S2 receive interrupt
XMT3	SINT16	0x80	4	I2S3 transmit interrupt
RCV3	SINT17	0x88	8	I2S3 receive interrupt

see SPRUFX5A -> DSP C5515 system guide (p. 53)

2. Creating VECTORS.ASM

```
;vectors.asm

        .sect "vectors"
rsv:    .ivec start,USE_RETA

nmi:    .ivec NMI_ISR
        INSTR

int0:   .ivec INT0_ISR
```

- Vector "packet" is 8 bytes long:
  - `0:` stk_mode or don't care
  - `1-3:` 24-bit address of ISR
  - `4-7:` 1-4byte executed instruction
- Reset vector can specify stack mode:
  - `C54X_STK:` 32-bit stk (default)
  - `USE_RETA:` 16-bit stk w/RETA
  - `NO_RETA:` 16-bit stk w/o RETA

- `.ivec` directive does the following:
  - fills remaining bytes with NOPs if not used
  - aligns each vector to an 8-byte boundary
  - places the target address in bytes 1-3

3. Enable Individual Interrupts (IER0/1)

- IER0 / IFR0 bitpositions

15	14	13	12	11	10	9	8
RCV2	XMT2	SAR	LCD	PROG3	Reserved	PROG2	DMA
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
PROG1	UART	PROG0	TINT	INT1	INT0	Reserved	
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

- IER1/IFR1 bitpositions

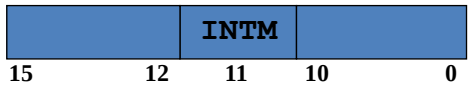
15				11	10	9	8
Reserved					RTOS	DLOG	BERR
R-0					R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
I2C	EMIF	GPIO	USB	SPI	RTC	RCV3	XMT3
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

```
enable:     OR  #k16,mmap(@IER0)
disable:    AND #k16,mmap(@IER0)
```

SPRUFX5A -> DSP C5515 system guide (p. 54)

4. Enable Global Interrupts (INTM)

ST1_55




```
enable:    BCLR  INTM
disable:   BSET  INTM
```

- **Does not affect bits in IER0/1**
- **INTM=1 (disabled) at reset**

Creating a macro:

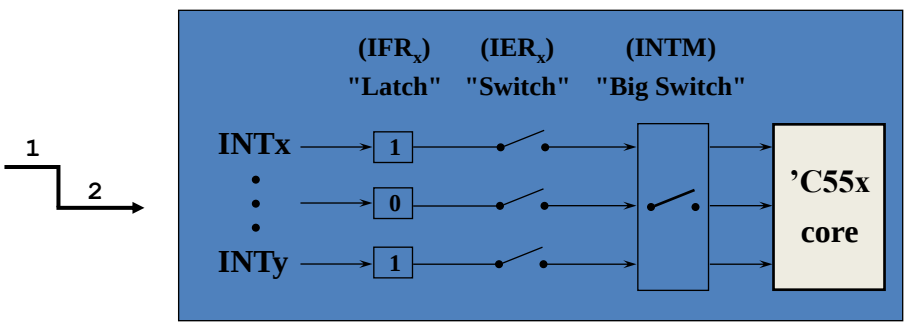
```
INTS_OFF   .macro
           BSET INTM
           .endm
```

Using a macro:

```
           . . .
           INTS_OFF
           . . .
```

Recognizing Interrupts

What events/conditions are required to recognize an interrupt?



- **IFRx$_{bit}$ must be set to a "1"**
- **IERx$_{bit}$ must be enabled ("1")**
- **INTM must be enabled ("0")**

5. Interrupt Flag Register (IFR0/1)

- Same configuration as IER0/1 registers
- Write a 1 to clear a bit

```
clear:  MOV  #mask,mmap(@IFR0)
```

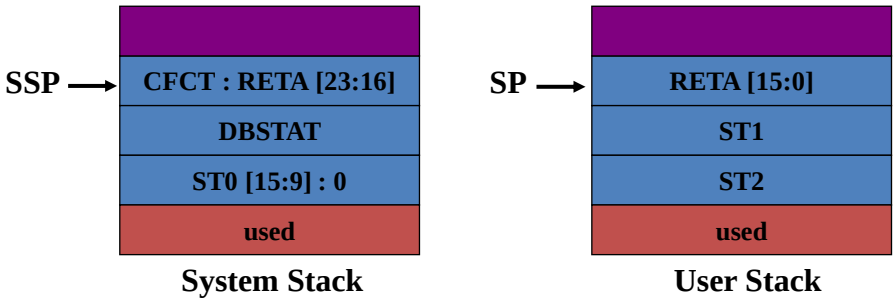
Writing a "0" to any IFR bit does nothing

7. Post Interrupt Hardware Sequence

CPU Action	Description
PC → Stacks#	Copy return address (PC) to stacks
Registers → Stacks	see next slide...
1 → INTM	Disable global interrupts
ISR address → PC	Load PC with ISR address
0 → IFR0/1 (n)	Clear corresponding interrupt flag bit

- # depends on stack configuration (32-bit or 16-bit stack)

7. View of Stacks Before ISR



- **DBSTAT: used for emulation purposes**
- **RETA: return address**

Only these items are automatically saved by the CPU

SWPU073e –>CPU reference (p4-8)

Context Save & Restore Instructions

Save	Restore
`PSH src`	`POP dst`
`PSH src1,src2`	`POP dst1,dst2`
`PSH src,Smem`	`POP dst,Smem`
`PSH ACx`	`POP ACx`
`PSH Smem`	`POP Smem`
`PSH dbl(Lmem)`	`POP dbl(Lmem)`

- **User is responsible for context save/restore**
- **Values must be popped in the reverse order that they are pushed**
- **STx is not considered “src” or “dst”. Use PSH mmap(@STx).**

8. Return From ISR

RETI

`RETI` instruction does the following:

- **Pops all CPU-pushed values from the stacks**
- **Returns program control to the interrupted task**
- **INTM is restored to it's previous state because ST1_55 is saved/restored automatically**

What Happens When Reset Occurs ?

- **All current operations are aborted, pipeline is flushed, registers are initialized, reset vector is fetched**
- **List of key registers and their initialized values:**

Interrupts

IVPH/D	FFFFh
IFRn	0h
INTM	1

Address Registers

XARn	0h
XDP	0h
XSP	0h

Status Bits

C54CM	1	M40	0
FRCT	0	CPL	0
SXMD	1	ARMS	0
SATD	0		

- **Recommendation: do not rely on initialized values - always initialize the necessary registers to the desired value during the reset ISR**

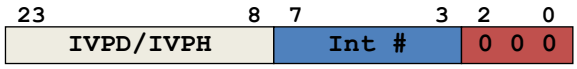
More on reset values in SWPU073e –>CPU reference

Relocating the Vector Table

- **Can be relocated to any 256-byte program page**
- **Vector table is split into two types of vectors: DSP, host**

DSP (IVPD)	Host (IVPH)
IV0…1V15 IV24…IV31	IV16…1V23

- **Allows host vectors to be placed in the DSP's shared memory area to support host access and ISRs**
- **Relocating the 24-bit vector address using the IVPD/IVPH Registers:**



- **On reset, IVPD/IVPH = FFFFh**

Interrupt "Priority"

There are three types of interrupt "priority"...

Hardware Priority

- **Selects *which* interrupt is serviced only when multiple interrupts occur at the same time or if INTM is set to 0 when more than one IFRx bit is set**
- **Based upon fixed priority as shown in the datasheet (e.g. Reset is highest, NMI is next, etc)**

Software Priority

- **User can select which interrupts are interruptible by "higher priority" interrupts by modifying the IERx register. For example, when servicing INT0 (higher hardware priority than INT3), user can modify IER to allow INT3 to interrupt INT0.**

BIOS SWI Priority

- **Within BIOS, hardware interrupts have a fixed priority (same as "hardware priority" above. However, SWIs (software interrupts) are managed easily within the BIOS Configuration Manager**

Review

1. **What are the interrupt sources?**

   Reset, NMI, Timers, Serial Ports,

   Host (HPI), DMA, External, Software

2. **How do you poll for interrupts?**

   Test the appropriate bit in IFRx, then goto ISR if TCx set

3. **What must you set up to respond to an interrupt?**

   INTM, IERx, SP/SSP and a vector

4. **What conditions affect interrupt latency?**

   Higher priority interrupts, IER_{bit}=0, processor is in hold mode, INTM = 1, memory speed, ...

ASM example (1)

```
            .sect "indata"
x0          .copy  in7.dat
            .def   start,fir
            .cpl_off
            .arms_off
            .c54cm_off

stklen      .set    100
a0          .usect "coeffs",16,1,1
y0          .usect "results",200,1,1
BOS         .usect "STK", stklen,1,1
BOSS        .usect "SSTK",stklen,1,1

            .sect "init"
table       .int 7FCh, 7FDh, 7FEh, 7FFh
            .int 800h, 801h, 802h, 803h
            .int 803h, 802h, 801h, 800h
            .int 7FFh, 7FEh, 7FDh, 7FCh

            .sect "code"

start:      AMOV #BOS+stklen, XSP     ;set up Stack and
            MOV  #BOSS+stklen,SSP     ;System Stack Pointers

            MOV  #8,mmap(@IER0)       ;Turn on INT2 and clear ints
            MOV  #0xFFFF,mmap(@IFR0)
            MOV  #0xFFFF,mmap(@IFR1)

            CALL copy                 ;copy coeffs
            CALL firsetup             ;setup for fir math
            BCLR INTM                 ;Enable Global Interrupts
main:       ADD  #1,AC0               ;main loop
            ADD  #1,AC1
            B    main
```

ASM example (2)

```
copy:    AMOV #table,XAR2                ;load pointers
         AMOV #a0,XAR3
         RPT  #7
         MOV  dbl(*AR2+),dbl(*AR3+)      ;move from table to a
         RET

firsetup:
         BSET FRCT                       ;turn on multiplier shift
         BSET M40                        ;turn on 40 bit math
         BSET SXMD                       ;turn on sign extension
         BCLR C54CM                      ;go to C55 native mode

         AMOV #x0,XAR2                   ;initialize pointers
         AMOV #x0+1,XAR3                 ;for data,
         AMOV #y0,XAR4                   ;results
         AMOV #a0,XCDP                   ;and coeffiecients
         MOV  #13,T0                     ;set up TO for ptr wrap

         MOV  #a0,BSAC                   ;buffer start address
         MOV  #16,BKC                    ;buffer size
         MOV  #0, CDP                    ;zero lower bits of XCDP

         BSET CDPLC                      ;turn on circ addressing

         RET
```

ASM example (3)

```
fir:     PSH  AC0                        ;save AC0 and 1
         PSH  AC1

         MPY *AR2+,*CDP+,AC0             ;AC0 gets 1st product
     ::  MPY *AR3+,*CDP+,AC1             ;AC1 gets 2nd product

     ||  RPT  #13                        ;RPT in parallel with MPYs
          MAC  *AR2+,*CDP+,AC0           ;form 14 results
     ::   MAC  *AR3+,*CDP+,AC1

         MAC  *(AR2-T0),*CDP+,AC0
     ::  MAC  *(AR3-T0),*CDP+,AC1        ;form last result and wrap pointers

         MOV  pair(hi(AC0)),dbl(*AR4+)   ;store AC0 & AC1 results

         POP  AC1                        ;restore AC0 and 1
         POP  AC0

         RETI
```

Vector table

```
        .def RSV
        .ref start
        .ref fir

        .sect "vectors"

RSV:    .ivec  start,USE_RETA

NMI:    .ivec  NMI

INT0:   .ivec  INT0

INT2:   .ivec  fir               ;fir ISR

TINT0:  .ivec  TINT0

RINT0:  .ivec  RINT0

RINT1:  .ivec  RINT1

; etc etc etc
```

Simulate interrupts

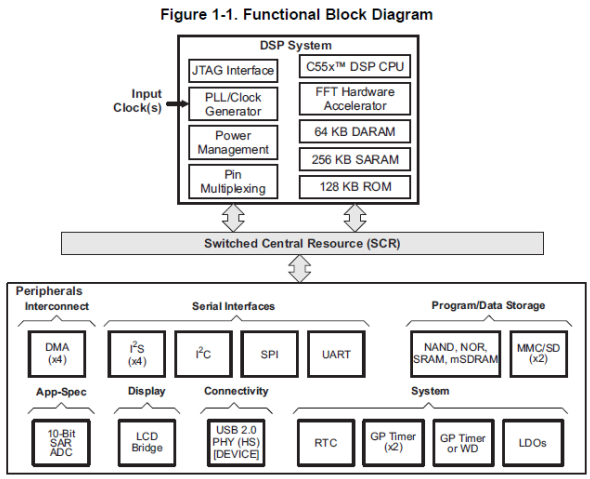
```
; to simulate interrupts, the following line must be
; used by CCS to know WHEN to fire the interrupt.
; RESET, NMI and SINT2-23 (INT0- to DSPINT) are supported.

            100 (+60) rpt EOS

; first interrupt occurs at the 100th cycle of
; the simulation. The 2nd fires at 160, the third at
; 220, etc. This pattern is repeated until the end
; of simulation (EOS)
```

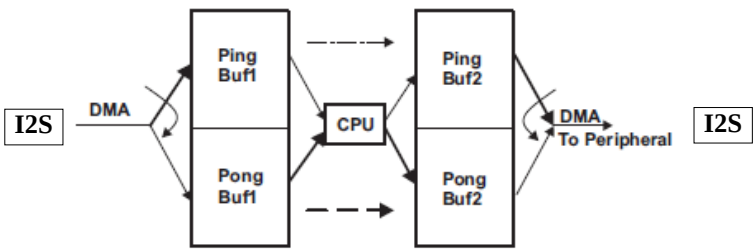
Audio example

- C5515 architecture



Ping-Pong buffering using DMA and I2S

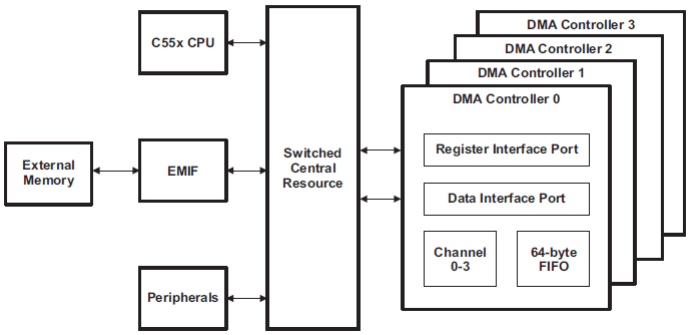
- DMA: Direct Memory Access
- I2S: Inter-IC Sound (I2S) Bus



For more info see SPRUFT2 -> DMA controller guide
-> SPRUFX4 I2S guide

4 DMA controller available

Figure 1. Conceptual Block Diagram of the DMA Controller



- **Register interface port**. The CPU uses this port to access the DMA controller registers.
- **Data interface port**. The DMA controller accesses internal dual-access RAM (DARAM), internal single-access RAM (SARAM), external memory, and on-chip peripherals through its data interface port.
- **64-byte FIFO**. As data is read from the source address, it is placed in the DMA controller FIFO. The four DMA channels must share the DMA controller FIFO; the FIFO can only be accessed by a single channel at a time.

DMA system registers

CPU Word Address	Acronym	Register Description
1C30h	DMAIFR	DMA Interrupt Flag Register
1C31h	DMAIER	DMA Interrupt Enable Register
1C1Ah	DMA0CESR1	DMA0 Channel Event Source Register 1
1C1Bh	DMA0CESR2	DMA0 Channel Event Source Register 2
1C1Ch	DMA1CESR1	DMA1 Channel Event Source Register 1
1C1Dh	DMA1CESR2	DMA1 Channel Event Source Register 2
1C36h	DMA2CESR1	DMA2 Channel Event Source Register 1
1C37h	DMA2CESR2	DMA2 Channel Event Source Register 2
1C38h	DMA3CESR1	DMA3 Channel Event Source Register 1
1C39h	DMA3CESR2	DMA3 Channel Event Source Register 2

(1) Refer to the *TMS320VC5505 DSP System Guide* (SPRUFP0) for more information on these registers.

- Not part of the DMA controllers, are part of the DSP system but do affect DMA process

DMA_IER / DMA_IFR

15	14	13	12	11	10	9	8
DMA3CH3IF	DMA3CH2IF	DMA3CH1IF	DMA3CH0IF	DMA2CH3IF	DMA2CH2IF	DMA2CH1IF	DMA2CH0IF
RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
7	6	5	4	3	2	1	0
DMA1CH3IF	DMA1CH2IF	DMA1CH1IF	DMA1CH0IF	DMA0CH3IF	DMA0CH2IF	DMA0CH1IF	DMA0CH0IF
RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

Figure 1-42. DMA Interrupt Enable Register (DMAIER) [1C31h]

15	14	13	12	11	10	9	8
DMA3CH3IE	DMA3CH2IE	DMA3CH1IE	DMA3CH0IE	DMA2CH3IE	DMA2CH2IE	DMA2CH1IE	DMA2CH0IE
RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
7	6	5	4	3	2	1	0
DMA1CH3IE	DMA1CH2IE	DMA1CH1IE	DMA1CH0IE	DMA0CH3IE	DMA0CH2IE	DMA0CH1IE	DMA0CH0IE
RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

Table 1-84. DMA Interrupt Flag Register (DMAIFR) Field Descriptions

Bit	Field	Value	Description
15-0	DMA*n*CH*m*IF		Channel interrupt status bits.
		0	DMA controller *n*, channel *m* has not completed its block transfer.
		1	DMA controller *n*, channel *m* block transfer complete.

Table 1-85. DMA Interrupt Enable Register (DMAIER) Field Descriptions

Bit	Field	Value	Description
15-0	DMA*n*CH*m*IE		Channel interrupt enable bits.
		0	DMA controller *n*, channel *m* interrupt is disabled.
		1	DMA controller *n*, channel *m* interrupt is enabled.

SPRUFP0C (p. 91)

DMA synchronization events

DMA Start *Byte* Address	CPU Start *Word* Address (I/O Space)	CPU Start *Word* Address (Data Space)	DSP Memory Map	DMA Controller 0 Memory Map	DMA Controller 1 Memory Map	DMA Controller 2 Memory Map	DMA Controller 3 Memory Map
0000 2800h	00 2800h	-	I2S0	I2S0	Reserved	Reserved	Reserved
0000 3A00h	00 3A00h	-	MMC/SD0	MMC/SD0			
0000 3B00h	00 3B00h	-	MMC/SD1	MMCSD1			
0001 0000h*	-	00 0000h (1)	DARAM	DARAM	DARAM	DARAM	DARAM
0009 0000h	-	00 8000h	SARAM	SARAM	SARAM	SARAM	SARAM
0000 1B00h	00 1B00h	-	UART	Reserved	UART	Reserved	Reserved
0000 2A00h	00 2A00h	-	I2S2		I2S2		
0000 1A00h	00 1A00h	-	I2C		Reserved	I2C	
0000 2B00h	00 2B00h	-	I2S3			I2S3	
0000 2900h	00 2900h	-	I2S1			Reserved	I2S1
0000 7000h	00 7000h	-	10-bit SAR			10-bit SAR	Reserved
0100 0000h	-	02 8000h	SDRAM			Reserved	SDRAM
0200 0000h	-	40 0000h	EMIF CS2				EMIF CS2
0300 0000h	-	60 0000h	EMIF CS3				EMIF CS3
0400 0000h	-	70 0000h	EMIF CS4				EMIF CS4
0500 0000h	-	78 0000h	EMIF CS5				EMIF CS5

(1) Word addresses 00 0000h-00 005Fh (which correspond to byte addresses 00 0000h-00 00BFh) are reserved for the memory-mapped registers (MMRs) of the DSP CPU.

SPRUFP0C (table 1.82 p. 89)

DMAnCESR1

Figure 1-43. DMA*n* Channel Event Source Register 1 (DMA*n*CESR1) [1C1Ah, 1C1Ch, 1C36h, and 1C38h]

15 12	11 8	7 4	3 0
Reserved	CH1EVT	Reserved	CH0EVT
R-0	RW-0	R-0	RW-0

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure 1-44. DMA*n* Channel Event Source Register 2 (DMA*n*CESR2) [1C1Bh, 1C1Dh, 1C37h, and 1C39h]

15 12	11 8	7 4	3 0
Reserved	CH3EVT	Reserved	CH2EVT
R-0	RW-0	R-0	RW-0

SPRUFP0C (p. 92)

DMA0 registers

CPU Word Address	Acronym	Register Description	Section
0C00h	DMACH0SSAL	Channel 0 Source Start Address (Lower Part) Register	Section 4.1
0C01h	DMACH0SSAU	Channel 0 Source Start Address (Upper Part) Register	Section 4.1
0C02h	DMACH0DSAL	Channel 0 Destination Start Address (Lower Part) Register	Section 4.2
0C03h	DMACH0DSAU	Channel 0 Destination Start Address (Upper Part) Register	Section 4.2
0C04h	DMACH0TCR1	Channel 0 Transfer Control Register 1	Section 4.3
0C05h	DMACH0TCR2	Channel 0 Transfer Control Register 2	Section 4.3
0C20h	DMACH1SSAL	Channel 1 Source Start Address (Lower Part) Register	Section 4.1
0C21h	DMACH1SSAU	Channel 1 Source Start Address (Upper Part) Register	Section 4.1
0C22h	DMACH1DSAL	Channel 1 Source Start Address (Lower Part) Register	Section 4.2
0C23h	DMACH1DSAU	Channel 1 Destination Start Address (Upper Part) Register	Section 4.2
0C24h	DMACH1TCR1	Channel 1 Transfer Control Register 1	Section 4.3
0C25h	DMACH1TCR2	Channel 1 Transfer Control Register 2	Section 4.3
0C40h	DMACH2SSAL	Channel 2 Source Start Address (Lower Part) Register	Section 4.1
0C41h	DMACH2SSAU	Channel 2 Source Start Address (Upper Part) Register	Section 4.1
0C42h	DMACH2DSAL	Channel 2 Destination Start Address (Lower Part) Register	Section 4.2
0C43h	DMACH2DSAU	Channel 2 Destination Start Address (Upper Part) Register	Section 4.2
0C44h	DMACH2TCR1	Channel 2 Transfer Control Register 1	Section 4.3
0C45h	DMACH2TCR2	Channel 2 Transfer Control Register 2	Section 4.3
0C60h	DMACH3SSAL	Channel 3 Source Start Address (Lower Part) Register	Section 4.1
0C61h	DMACH3SSAU	Channel 3 Source Start Address (Upper Part) Register	Section 4.1
0C62h	DMACH3DSAL	Channel 3 Destination Start Address (Lower Part) Register	Section 4.2

SPRUFT2 -> DMA controller guide (p. 24)

Two-part DMA transfer



Table 2. Registers Used to Define the Start Addresses for a DMA Transfer

Register	Load with...
DMACH*m*SSAL	Source start address (least-significant part)
DMACH*m*SSAU	Source start address (most-significant part)
DMACH*m*DSAL	Destination start address (least-significant part)
DMACH*m*DSAU	Destination start address (most-significant part)

Transfer Control Registers

- DMACHmTCR1 and DMACHmTCR2 specify:
  - the size of the transfer,
  - enable event synchronization,
  - auto-initialization,
  - select the source and destination addressing mode,
  - specify the burst mode of the channel,
  - monitor status of the DMA channel through these registers.

Transfer Control Registers

Figure 18. Transfer Control Register 1 (DMACH*m*TCR1)

15	0
LENGTH	
RW-0	

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Figure 19. Transfer Control Register 2 (DMACH*m*TCR2)

15	14	13	12	11	10	9	8
EN	STATUS	INTEN	AUTORLD	RSV		DSTAMODE	
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0		R/W-0	

7	6	5	3	2	1	0
SRCAMODE		BURSTMODE		SYNCMODE	LAST_XFER	PING_PONG_EN
R/W-0		R/W-0		R/W-0	R/W-0	R-0

SPRUFT2 -> DMA controller guide (p. 29)

Remark: DMA byte address

- Change word address to byte address and add offset for DMA
- Example

```
RxL1_DMA_address = (Uint32)RcvL1;
RxL1_DMA_address = (RxL1_DMA_address<<1) + 0x10000;
       // 0x10000 DARAM offset see DMA memory map
```

DMA Start *Byte* Address	CPU Start *Word* Address (I/O Space)	CPU Start *Word* Address (Data Space)	DSP Memory Map	DMA Controller 0 Memory Map	DMA Controller 1 Memory Map	DMA Controller 2 Memory Map	DMA Controller 3 Memory Map
0000 2800h	00 2800h	-	I2S0	I2S0	Reserved	Reserved	Reserved
0000 3A00h	00 3A00h	-	MMC/SD0	MMC/SD0			
0000 3B00h	00 3B00h	-	MMC/SD1	MMCSD1			
0001 0000h*	-	00 0000h (1)	DARAM	DARAM	DARAM	DARAM	DARAM
0009 0000h	-	00 8000h	SARAM	SARAM	SARAM	SARAM	SARAM
0000 1B00h	00 1B00h	-	UART		UART	Reserved	

SPRUFT2 -> DMA controller guide (p. 13)

I2S0 register mapping

Table 1-13. I2S0 Register Mapping Summary

CPU Word Address	Acronym	Description	
2800h	I2SSCTRL	I2S Serializer Control Register	Section 1.3.1
2804h	I2SSRATE	I2S Sample Rate Generator Register	Section 1.3.2
2808h	I2STXLT0	I2S Transmit Left Data 0 Register	Section 1.3.3
2809h	I2STXLT1	I2S Transmit Left Data 1 Register	Section 1.3.4
280Ch	I2STXRT0	I2S Transmit Right Data 0 Register	Section 1.3.5
280Dh	I2STXRT1	I2S Transmit Right Data 1 Register	Section 1.3.6
2810h	I2SINTFL	I2S Interrupt Flag Register	Section 1.3.7
2814h	I2SINTMASK	I2S Interrupt Mask Register	Section 1.3.8
2828h	I2SRXLT0	I2S Receive Left Data 0 Register	Section 1.3.9
2829h	I2SRXLT1	I2S Receive Left Data 1 Register	Section 1.3.10
282Ch	I2SRXRT0	I2S Receive Right Data 0 Register	Section 1.3.11
282Dh	I2SRXRT1	I2S Receive Right Data 1 Register	Section 1.3.12

SPRUFX4 -> I2S guide (p.28)

Convolutional encoder/ Viterbi decoder

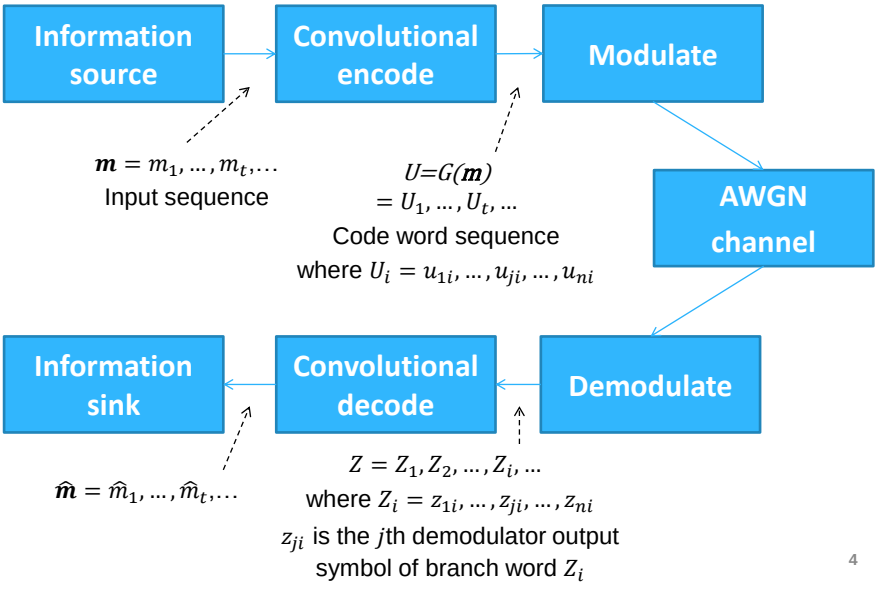
Overview

- Architecture
- Convolutional encoder
  - Graphical representations
    - State diagram
    - Tree diagram
    - Trellis diagram

Overzicht

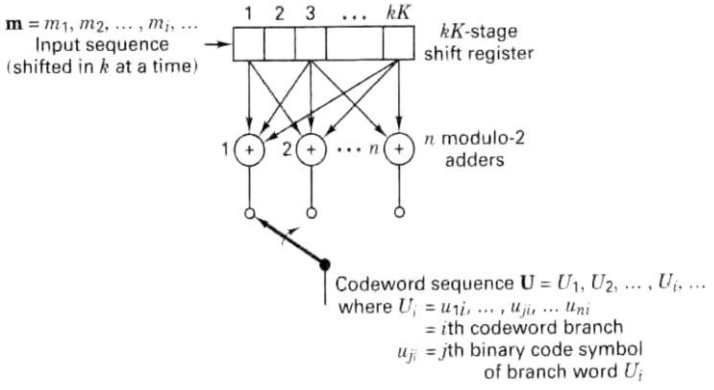
- Viterbi decoder
  - Maximum likelihood
  - Binary symmetrical channel
  - Viterbi algorithm
  - Application
- C55xx implementation

Architecture

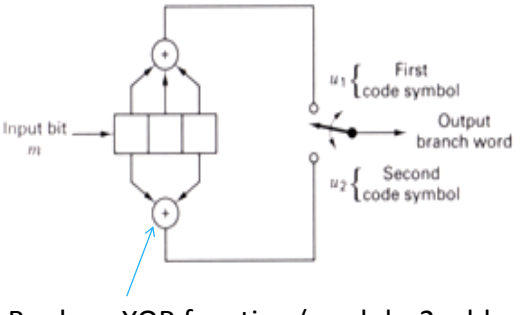


Convolutional encoder

- Algemeen schema encoder



Convolutional encoder



K= length shiftregister (constraint length)

n = # output bits

code rate: $\frac{K}{n} = \frac{1}{2}$

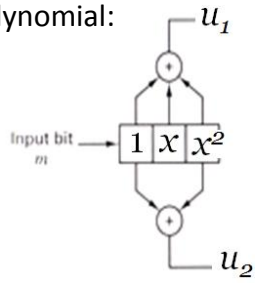
Boolean XOR function (modulo-2 adders)

Convolutional encoder

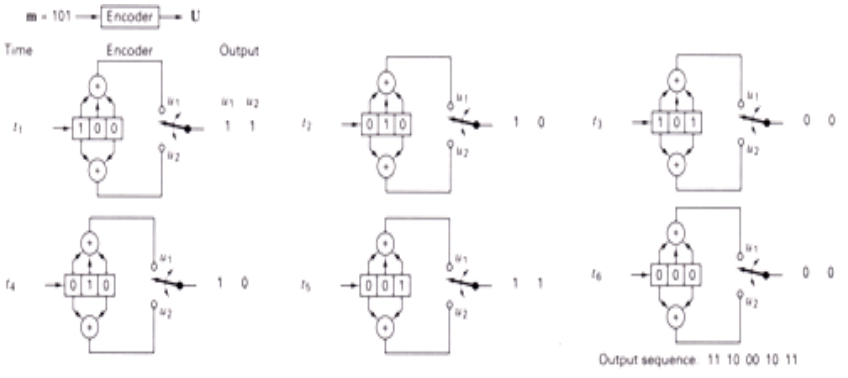
- Encoding function can be described by a polynomial:

  $u_1 = 1 \oplus x \oplus x^2$

  $u_2 = 1 \oplus x^2$



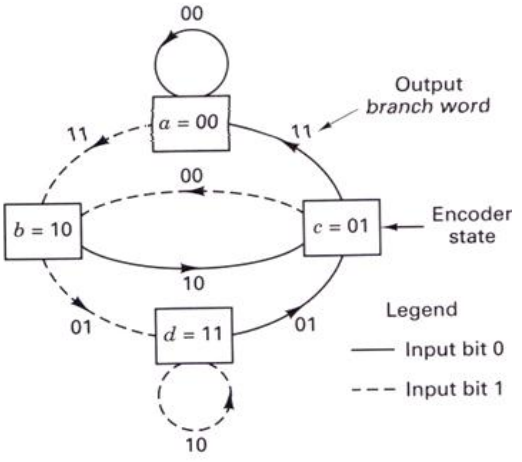
Convolutional encoder



- Timestamp 1 (t_1) output is '11' => u_1 = '1' ⊕ '0' ⊕ '0' = '1'

  u_2 = '1' ⊕ '0' = '1'

Representation: State diagram



Representation: State diagram

- Output depends on previous bits

Input bit m_j	Register contents	State at time t_j	State at time t_{j+1}	Branch word at time t_j u_1	u_2
—	0 0 0	0 0	0 0	—	
1	1 0 0	0 0	1 0	1	1
1	1 1 0	1 0	1 1	0	1
0	0 1 1	1 1	0 1	0	1
1	1 0 1	0 1	1 0	0	1
1	1 1 0	1 0	1 1	0	1
0	0 1 1	1 1	0 1	0	1
0	0 0 1	0 1	0 0	1	1

state t_j state t_{j+1}

Output sequence: $\mathbf{U} = 1\,1 \quad 0\,1 \quad 0\,1 \quad 0\,0 \quad 0\,1 \quad 0\,1 \quad 1\,1$

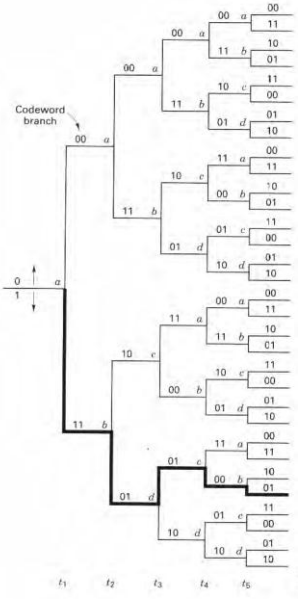
Input bit m_j	Register contents	State at time t_j	State at time t_{j+1}	Branch word at time t_j u_1	u_2
—	1 1 ×	1 ×	1 1	—	
1	1 1 1	1 1	1 1	1	0
1	1 1 1	1 1	1 1	1	0
0	0 1 1	1 1	0 1	0	1
1	1 0 1	0 1	1 0	0	0
1	1 1 0	1 0	1 1	0	1
0	0 1 1	1 1	0 1	0	1
0	0 0 1	0 1	0 0	1	1

state t_j state t_{j+1}

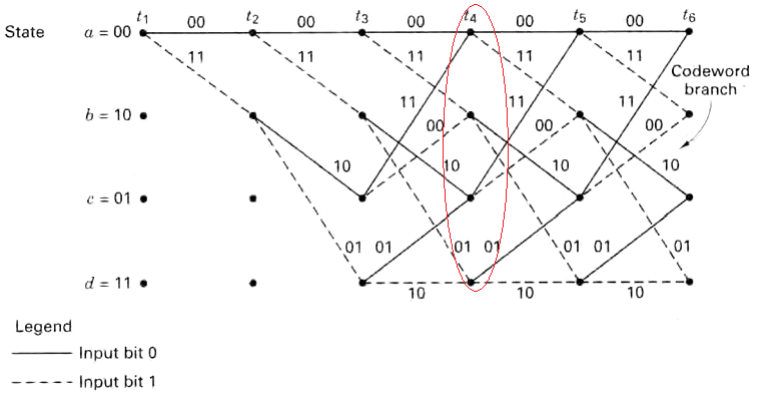
Output sequence: $\mathbf{U} = 1\,0 \quad 1\,0 \quad 0\,1 \quad 0\,0 \quad 0\,1 \quad 0\,1 \quad 1\,1$

Representation: Tree diagram

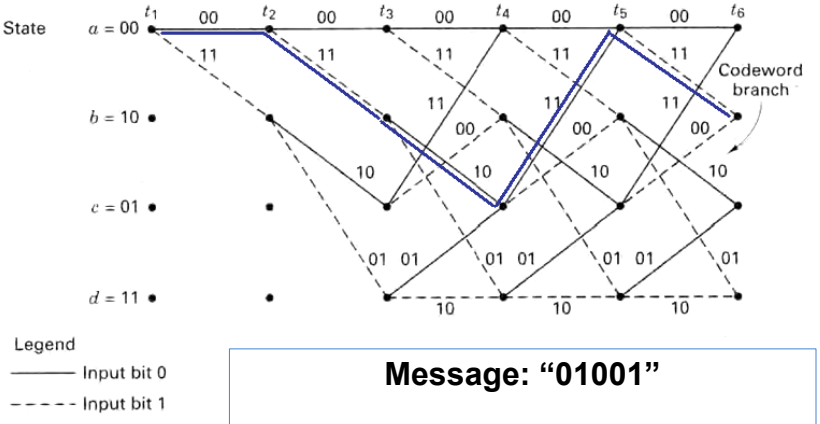
- Starting situation: “000”



Representation: Trellis



Voorstellingswijzen: Trellis



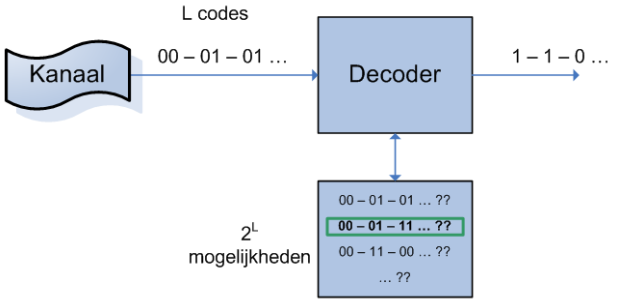
Message: “01001”

Gecodeerd bericht:”00 11 10 11 11”

Viterbi decoding

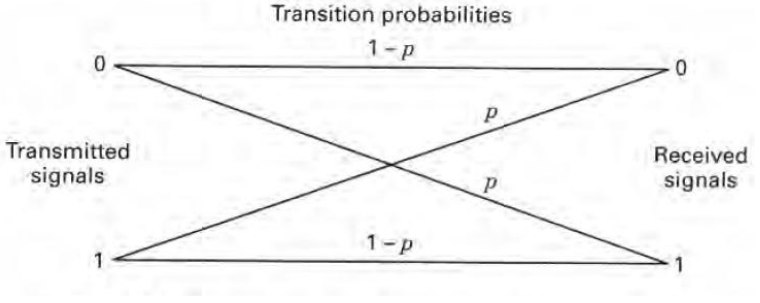
Maximum likelihood decoding

- Sequences of input messages have equal probability
- Decoder chooses the sequence with the highest probability to be correct



Binary symmetric channel

- Symmetric transition probabilities
  - P(0|1) = P(1|0) = p
  - P(0|0) = P(1|1) = (1-p)



Binary symmetric channel

- No memory effect
- Hamming distance is a robust measure for assessing the similarity between code symbols

Reveived code	Expected code	Hamming distance
00	00	0
00	01	1
00	10	1
00	11	2

Viterbi algorithm

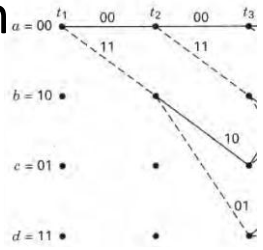
- Principal of maximum likelihood
- Complexity is independent of L
- Based on trellis-diagram
- Assume binary symmetric channel:
  - Hamming distance as a metric
  - Hard Viterbi decoding

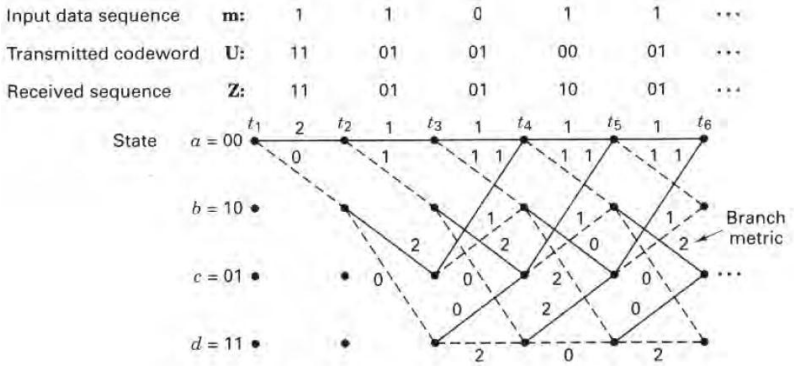
Viterbi Algorithm

```
for each frame:
{ Initialize metrics
   for each symbol:
   { //Metric Update or Add-Compare-Select (ACS)
   For each delay state:
      {Calculate local distance of input to each possible
      path
      Accumulate total distance for each path
      Select and save minimum distance
      Save indication of path taken}
   }//Traceback
   for each bit in a frame (or for minimum # bits):
      {Calculate position in transition data of the current
      state
      Read selected bit corresponding to state
      Update state value with new bit}
reverse output bit ordering}
```

Viterbi algorithm

- trellis-diagram of the decoder

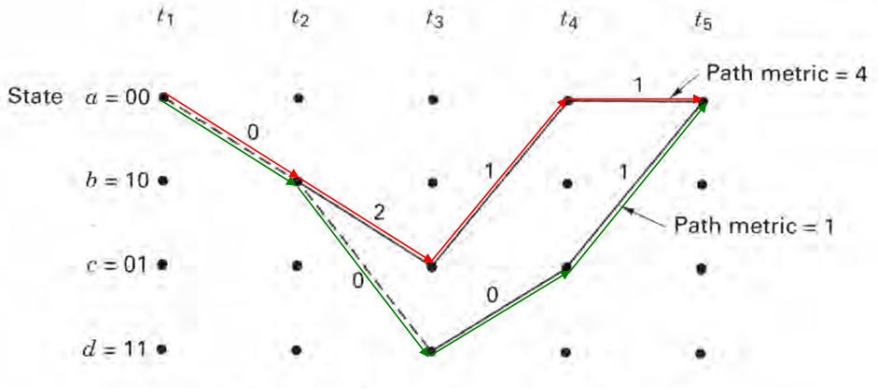




Viterbi algorithm

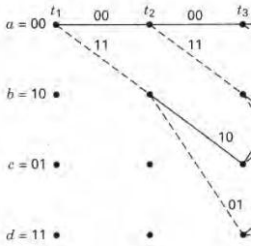
- If two sequence paths in the trellis diagram arrive in a single state, then one of the two paths can be eliminated
- At a given time t_i the a cumulative Hamming distance can be computed
- Path with the lowest cumulative Hamming distance is the most probably path.

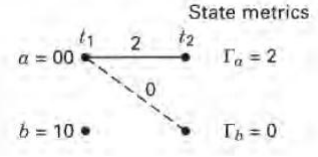
Viterbi algorithm



Viterbi algorithm: in detail

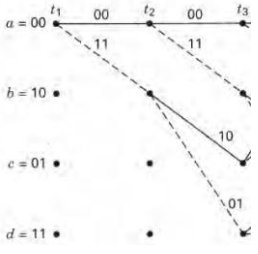
- Begin state ’00’
- At t_1 ‘11’ is received

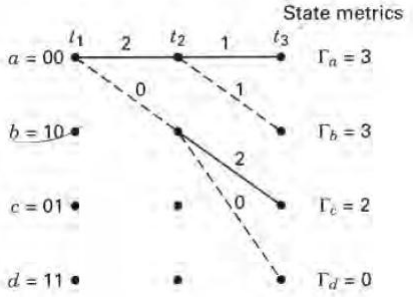




Viterbi algorithm: in detail

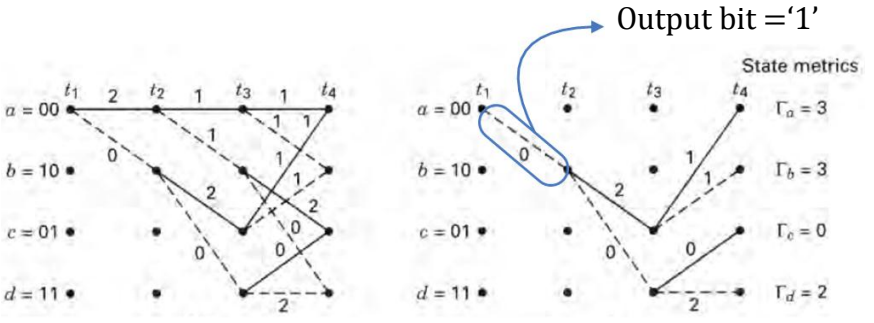
- At t_2 ‘01’ is received





Viterbi algorithm: in detail

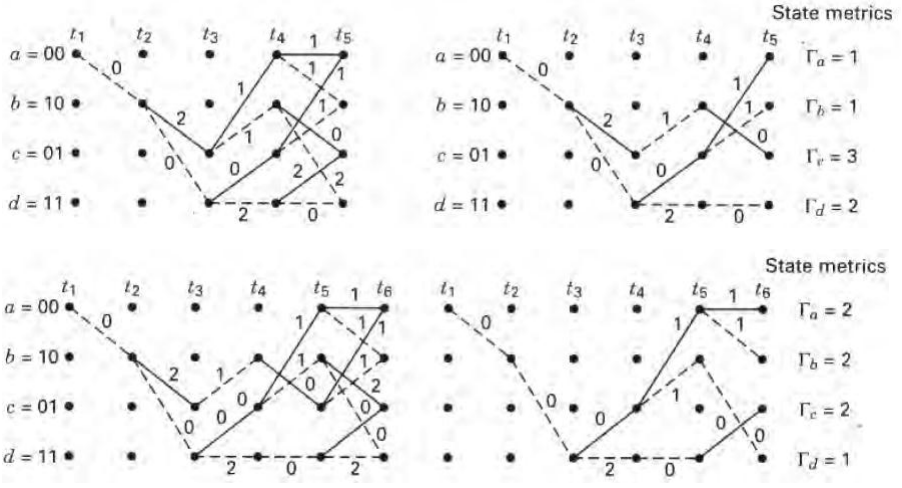
- At t_3 '01' is received
- Paths can be eliminated



Viterbi algorithm: in detail

- Procedure is repeated
- At t_4 '10' is received
- At t_5 '01' is received

Viterbi algorithm: in detail



Viterbi algorithm: in detail

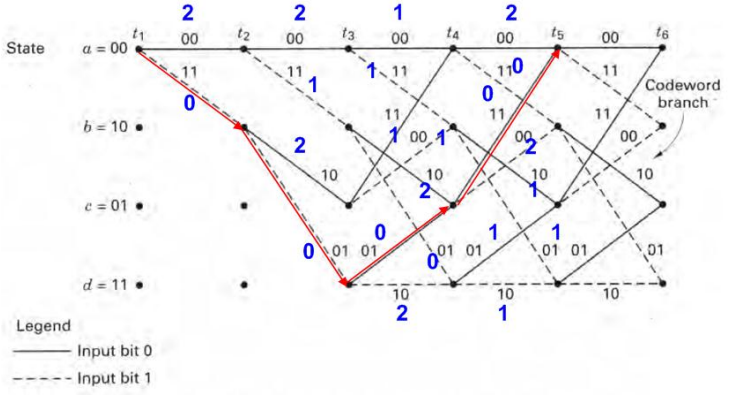
- After elimination number of paths is equal to the number of states
- Sequence determines the number of output bits
- Delay can be up to $5K$

Viterbi algorithm: Matlab

- trel = poly2trellis(3,[7 5]);
  msg = [1 1 0 0]';
  code = convenc(msg, trel);
  % code(4) = 0;
  % code(6) = 0;
  tblen = 2;
  decoded = vitdec(code, trel, tblen, 'trunc', 'hard');

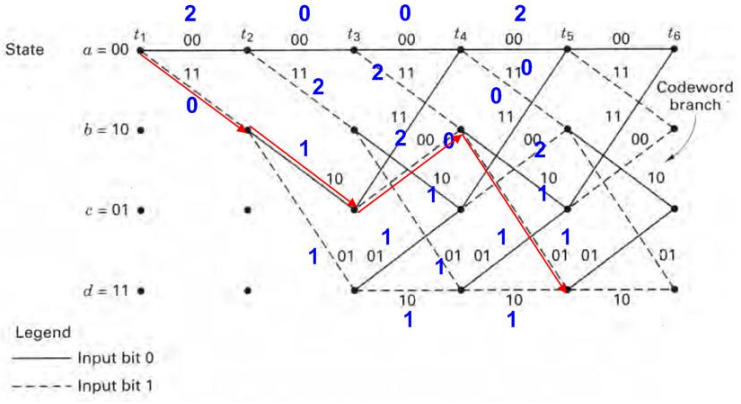
Viterbi algorithm: Matlab

- Code = 1 1 0 1 0 1 1 1 Output= 1 1 0 0 -> OK



Viterbi algorithm: Matlab

- Code = 1 1 0 0 0 0 1 1 Uitgang = 1 0 1 1



Viterbi algorithm: Matlab

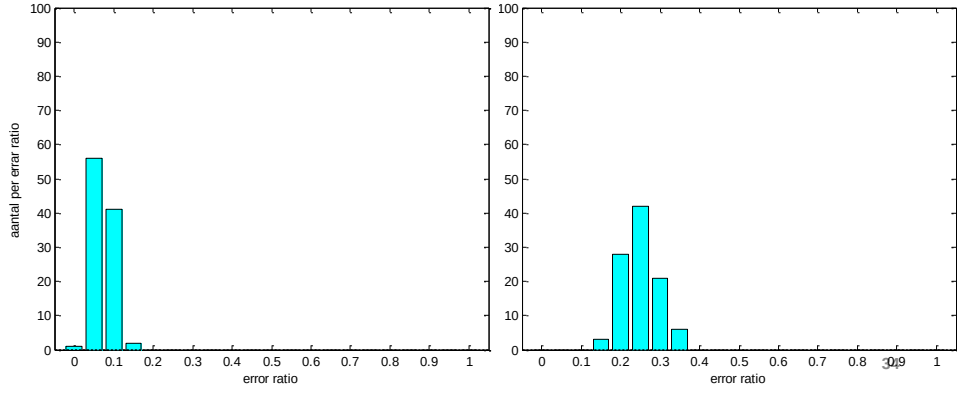
- Foutieve besluit verhelpen door:
  - Nog verder bits decoderen
  - Traceback lengte vergroten
- msg = [1 1 0 0 0 0]';
  tblen = 4;

Demo

- Begin situation
  - Original message: “0 0 1 0 1 0 0 1 0 0 0”
  - Length original message: 11-bit
  - Traceback length: 11
  - 100 messages for 1 error ratio
  - 100 simulations to have variance

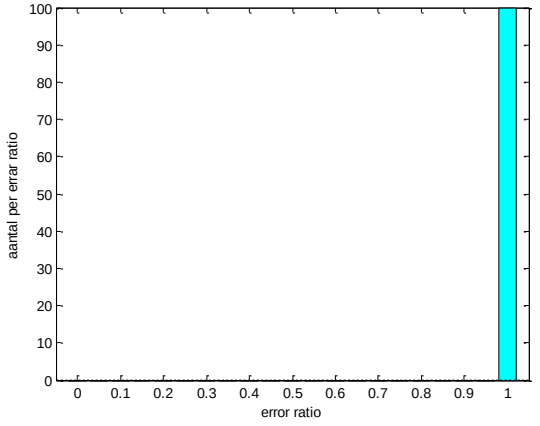
Demo

- Simulation 1: 1-bit and 2-bit error



Demo

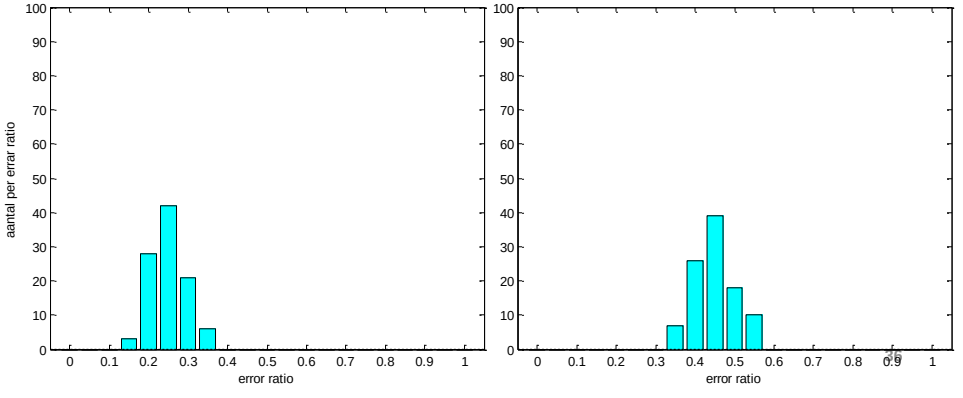
- Simulation 2: 5-bit errors



Demo

- Simulation 3: Traceback length

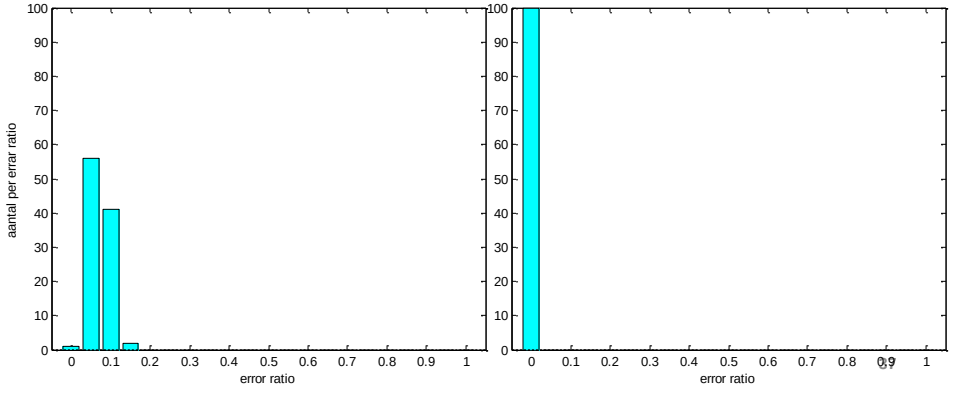
Traceback length = 11 2-bit error Traceback length = 3



Demo

- Simulation 4: Errors not in last two bits
  Traceback length = 11 1-bit error

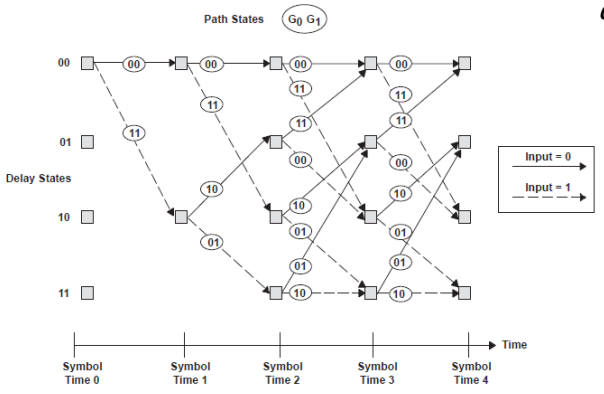
Random errors Errors not in last two bits



C55xx decoder implementation

- Assume $K = 3, \frac{1}{2}$ code rate, 2^{K-1} states (same encoder as previously seen)

received symbols denoted by SD_0SD_1



Decoder (Viterbi) Algorithm

```
for each frame:
{ Initialize metrics
   for each symbol:
   { //Metric Update or Add-Compare-Select (ACS)
   For each delay state:
      {Calculate local distance of input to each possible
      path
      Accumulate total distance for each path
      Select and save minimum distance
      Save indication of path taken}
   }//Traceback
   for each bit in a frame (or for minimum # bits):
      {Calculate position in transition data of the current
      state
      Read selected bit corresponding to state
      Update state value with new bit}
reverse output bit ordering}
```

Distance function

- Euclidean distance as an alternative for Hamming distance

$$\mathrm{d}_j = \left(SD_0 - G_0(j)\right)^2 + \left(SD_1 - G_1(j)\right)^2$$

$$d_j = (SD_0^2 - 2SD_0G_0(j) + G_0(j)^2) + (SD_1^2 - 2SD_1G_1(j) + G_1(j)^2)$$

- Use of antipodal values 1→-1, 0→1 for $G_i(j)$
- $SD_0^2 + SD_1^2$ and $G_0(j)^2 + G_1(\mathrm{j})^2$ are the same for each path so eliminate them, also remove 2 ‘s
- Maximize local distance $d_j' = SD_0G_0(j) + SD_1G_1(j)$

Distance function

- $d_j = SD_0G_0(j) + SD_1G_1\ (j)$
- Recall that antipodal values 1→-1, 0→1 are used for $G_i(j)$

$G_0(j)G_1(j)$	d_j
00	$A = SD_0 + SD_1$
01	$B = SD_0 - SD_1$
10	$-B = SD_1 - SD_0$
11	$-A = -SD_0 - SD_1$

Metric update

Two buffers are used are swapped at the end

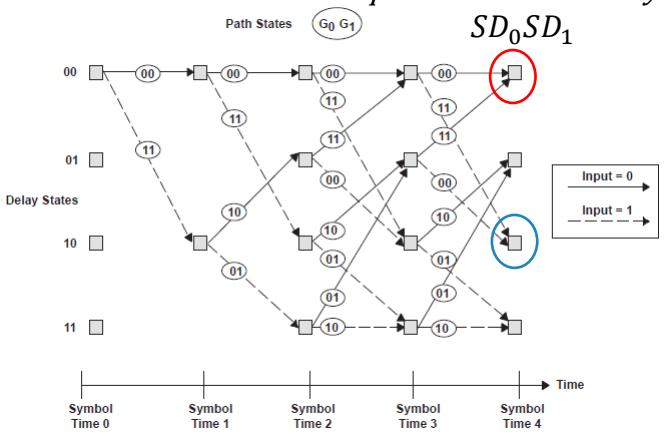
```
; AR5: pointer to the old metrics table
; AR4: pointer to the new metrics table
; T2 = SD(2*j) + SD(2*j+1)
;Compute New_metric (i)&(i+2)
;hi(AC0) = *AR5 + T2, ;AC0=Old_Met(2*j) +T2
;lo(AC0) = *AR5+ - T2 ;AC0=Old_met(2*j)-T2
ADDSUB T2,*AR5+,AC0
;hi(AC1) = *AR5 - T2, ;AC1=Old_Met(2*j+1) -T2
;lo(AC1) = *AR5+ + T2 ;AC1=Old_met(2*j+1)+T2
SUBADD T2,*AR5+,AC1
MAXDIFF AC0, AC1, AC2, AC1 ;Compare AC0, AC1
||*AR4(#2) = lo(AC2), ;Store New_metric(i-1)&(i-1+2)
*AR4+ = hi(AC2)
```

Also update TRN0 and TRN1

Metric update

Compare with received symbol

SD_0SD_1



TRN

- For a single trellis column

TRN0=$b_1 b_0 xx\ xxxx\ xxxx\ xxxx$

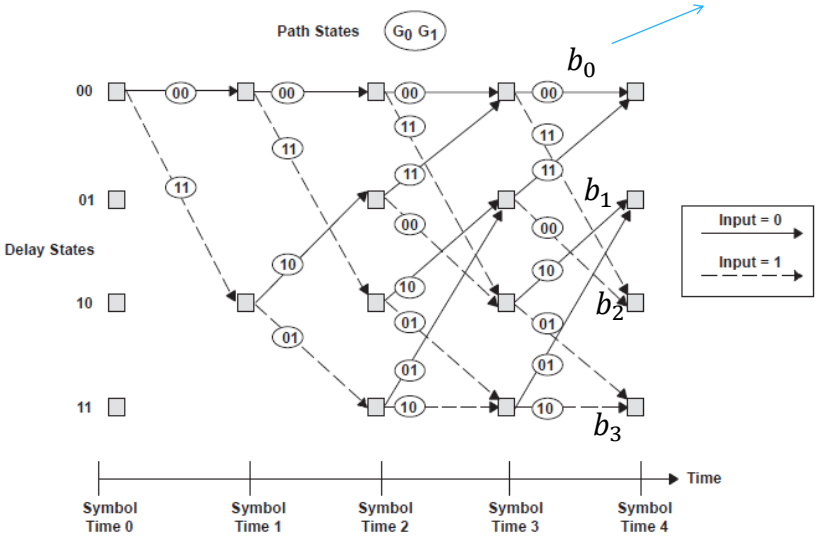
TRN1=$b_3 b_2 xx\ xxxx\ xxxx\ xxxx$

TRN=$b_3 b_2 b_1 b_0$

AR2

TRN

If bit=0 then upperarrow else lowerarrow was best

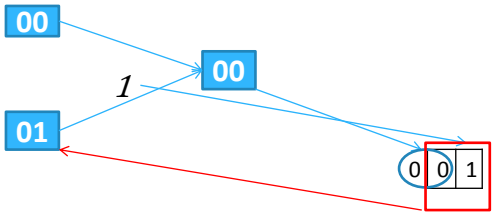


Traceback function

```
;   AC0 accumulator = STATE value
;   AR2 = pointer to STATE_TRANS array

;single iterate of loop
;TC2 = bit (*AR2-, AC0) ;Test bit in state variable.
BTST AC0, *AR2-, TC2

ROL CARRY, AC0, TC2, AC0
```



References

- Hendrix Henry, *Application report SPRA776A*, April 2009