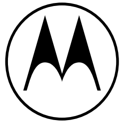


Wireless Engineering Bulletin:

INTRODUCTION TO THE DSP56300

An Approach in 8 Exercises

Version	Comments	Release date
0.1	Draft Review Copy	1st August 1996

Embedded Systems Group,
Motorola Semiconductor Products Sector,
East Kilbride, Scotland, U.K.

Motorola reserves the right to make changes without further notice to any product herein to improve reliability, function, or design. Motorola does not assume any liability arising out of the application or use of any product, circuit, or software described herein; neither does it convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such intended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola and the Motorola logo* are registered trademarks of Motorola.

*

© MOTOROLA LTD., 1996. All trademarks are recognised.

INTRODUCTION TO THE DSP56300

An Approach in 8 Exercises

1. Purpose of the Exercises

The aim of these exercises is to provide a quick and easy hands on introduction to the DSP56300 family of processors and the corresponding development tools. The exercises were designed for both engineers who are familiar with other DSPs but are new to the Motorola architecture and tools, and those who are using DSPs for the first time.

In these exercises the assembler and linker will be used to

- generate executable files from assembler code
- check the listings for correct assembly syntax
- check the memory allocation and form the memory space.

In addition the DSP simulator will be used to

- verify the correct operation of software
- debug code reliably by running or stepping through
- log and restore I/O data
- count cycles to measure the performance.

The exercise code consists of all the assembly and control files you need for the exercises as well as a set of step-by-step instructions detailing how to run the exercises. The directory structure of this code should be preserved as each directory (ex1..ex8) contains all the files for a different exercise.

The Exercise Code. Each exercise directory consists of :

ex*x*_main.asm	-	the main assembler file
ex*x_func*.asm.	-	subroutines are provided (optionally)
ex*x*.ctl	-	defines the memory mapping (linker memory control file)
ex*x*.txt	-	a textfile comprising all the instructions for the test run (can also be found in documentation)

The first five sections of this document apply to all of the exercises and it is recommended that you read these sections before starting to work on the exercises themselves. In addition to this there is a section dedicated to each individual exercise.

The Exercise Documentation. Each exercise documentation section has the following general structure:

Introduction
Technical Considerations
Implementation Description
Test Run

The introduction contains a list of the features and topics that are covered in the exercise. The technical considerations section discusses DSP issues and background information. The implementation description reports why the program structure was chosen and what you should know about the assembler implementation. Finally, the instructions to run the exercise are

provided in the test run section.

Although the test run section contains full instructions on how to work through each exercise, it is suggested that after each test run, engineers will experiment further by changing the code to using additional features of the development tools.

List of Exercises

Exercise 1: Calculate a Sum of Products. This small piece of code shall serve mainly to inspect and understand the general file format of the assembler files, get started with the assembler and linker tools, and to start working with the simulator.
In order to demonstrate the Arithmetic Logic Unit (ALU) and the general concept of operation we will use the 'mac' instruction in a hardware do-loop. This is the basic form for digital filtering.

Exercise 2: Addressing Modes. Effective addressing of data is essential for good performance of any algorithm. All possible modes of data addressing are shown here with the use of simple examples. The principles of the Address Generation Unit (AGU) are explained in the documentation.

Exercise 3: Division. A binary fractional division of two numbers is performed in this exercise. The 'div' instruction is used here to get a quotient with full 24 bit resolution, and the user can verify the result with respect to fractional number representation. Further actions in the simulation environment are shown.

Exercise 4: FIR Filtering. One of the most common applications is taken here as an example to show how the optimized MAC instruction works, combined with parallel data moves, to get a minimum of instruction cycles per sample. In this exercise the user can run an FIR filter in a subroutine. This is tested using real environment simulation, i.e. multiple data values are accessed from a source external to the DSP.

Exercise 5: Root calculation. This exercise uses another common application to clarify the fractional data format in another context. The approach taken here is based on a recursive estimation technique producing full 24 bit resolution obtained from a 48 bit long input figure. The algorithm can be tested with different sets of input data.

Exercise 6: Matrix multiplication. The multiplication of 2 matrices is modelled for the demonstration of nested loops. Optimization of loop overheads is shown with this application and the hardware stack operation may also be analysed here in the context of the hardware loops.

Exercise 7: Selected Instruction Examples. This short exercise demonstrates a number of instructions which have special features and formats. These instructions are used in a few lines of code to show the instruction format and operation.

Exercise 8: Power Analysis. The last exercise tries to demonstrate a simple signal analysis task, i.e. the calculation of a sliding power window in a signal finding the maximum power. When the maximum is found, the signal power is evaluated with the help of a threshold look up table.

The following sections contain general information that is to be applied to all the exercises. The way from a text file (assembler) to the executable is explained as well as memory organisation and some general hints to avoid the most common errors.

2. How to get an executable file from assembler code:

Having written assembler code that is stored in a source file (*.asm), the assembly tool (asm56300) is called to check the syntax of the source code, translate and process the assembler directives and macros, and to generate a file that is generally referred to as the object file.

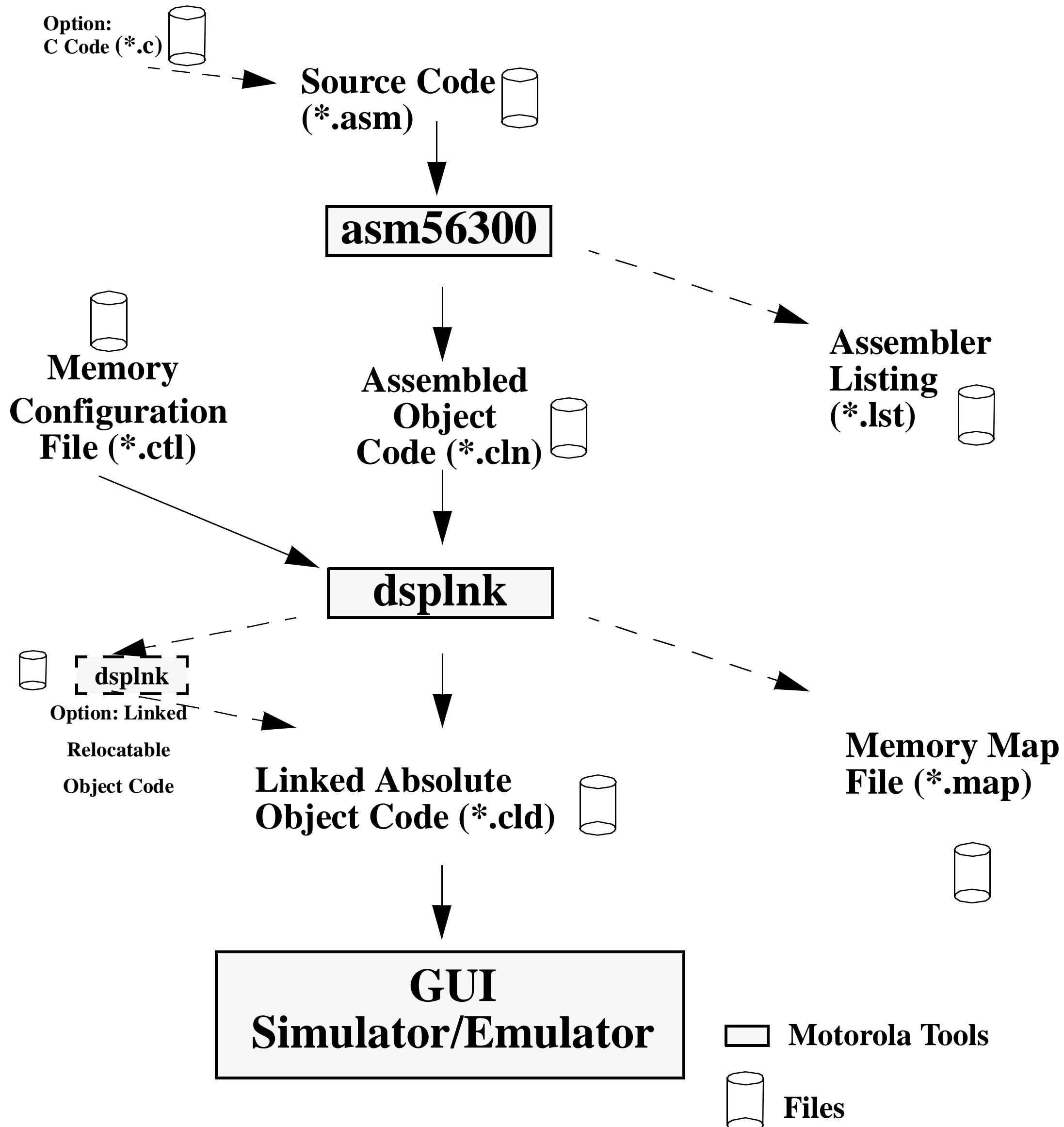


Figure 1: File Generation

This object file contains the syntax checked mnemonics, relatively addressed references and it often contains unresolved symbols, since several of these object files may be part of one loadable file. However, Figure 1 shows the example using only a single file.

Using the assembler it is possible to generate the ‘listing file’ which makes it possible to check all the actions which occurred during assembly time. The listing file is only a means to check, it is not used as input for further processing (see Figure 1).
The linker is called to link all the assembled files together. This happens in order to get a software image that is mapped onto the destination’s memory size and configuration. References to external sources (different files) are resolved at this point by replacing global names with real values or addresses. (If the linker is called with the option to produce a relocatable

output file, this file is generated now.)

The normal case is that the linker now takes the memory configuration file to convert all the symbols into absolute addresses and thus producing a file format (COFF, common object file format), that is directly loadable into the simulation/emulation platform.

3. The DSP56300 Memory Map

The memory of the DSP56300 is organised into three different areas: the p: memory where all the program code and the interrupt vectors are located, and two areas for data storage, x: and y: memory (see Figure 2).

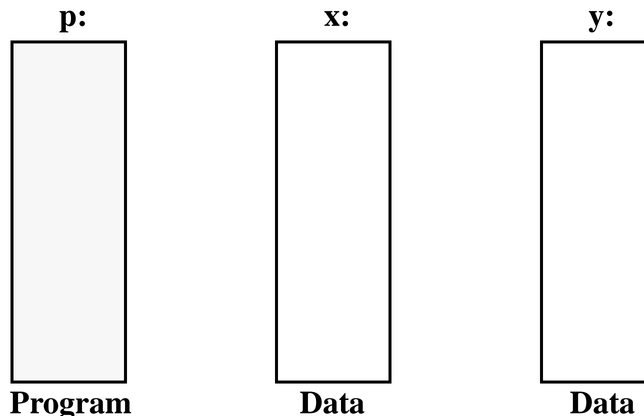


Figure 2: DSP56300 Memory Organisation

Since each of these areas are connected to a dedicated bus, the DSP is able to process most of the operations in a single clock cycle. However, this requires that the data is located correctly and sensible memory allocation is essential for fast running algorithms. For instance, if one has two streams of input data to be combined during processing these streams should obviously not be located in the same data memory area.

4. Recommended Reading

If you have never written assembler code before it is recommended that you read the following paragraphs before starting with the exercises.

DSP Development Software, Assembler Reference Manual [3], Chapter 2, describes briefly the format of the assembler files, i.e. the use of the instructions, directives and other conventions.
DSP56300 Family Manual [1], Page 1-3, shows the core architecture. Here you can get an idea of the functional blocks of the 56300 core and the general bus structure of the chip.

If you want to go even one step further, you can take a look at the schematics of the ALU (Page 3-3), AGU (Page 4-1) and read the section dealing with the data representation.

5 References

The following documents ([1], [2], [3]) should be available when going through the exercises, in addition, [4] and [5] may be useful.

[1] DSP 56300 Family Manual, DSP56300FM/AD covering the following topics
- Detailed description of the core, DMA, cache
- PLL, OnCE
- Instruction set
- Benchmarks.

[2] DSP 56301 User's Manual, DSP56301UM/AD containing
- Chip description
- Pins, memory, I/O
- Host interface
- Timer, ESSI
- SCI
- Bootstrap program
- Interrupt equates .

[3] DSP Development Software Manual covering the software topics such as
- Simulator reference
- C library functions
- GUI (graphical user interface)
- Assembler and COFF file
- Linker and librarian reference
- Instruction set
- Additional readings are mentioned in the Exx.txt files.

[4] Fractional and Integer Arithmetic using the DSP56000 Family of General-Purpose DSPs, which explains
- Data Representations
- Addition and Subtraction
- Multiplication and Division

[5] Real Time Digital Signal Processing Applications with Motorola's DSP56000 Family with the following topics
- FIR, IIR, FFT, ..

If you have further questions, please contact your local MOTOROLA distribution centre.

MOTOROLA
Semiconductor Products Sector
Embedded Systems Group
East Kilbride
Glasgow G75 OTG
Scotland, U.K.

EXERCISE 1 - CALCULATE A SUM OF PRODUCTS

Introduction

This first exercise is intended to show the first steps on the way to successful assembler code development. If you have experience in writing assembler code and running it on a development system you should go through this quickly or even skip this exercise completely.

After having done this exercise you should

- Define the location of code and data in DSP memory
- Load a program into the simulator
- Step through, debug, and set breakpoints
- Use the MAC instruction in a loop
- Perform and interpret a fractional multiplication

Technical Considerations

Performing Operations in the 56300. The general principle of data processing is shown below in Figure 3. Most of the instructions use either one or two of the registers x0, x1, y0, y1 as an input and the result is stored in one of the accumulators a or b. In addition to this, two moves may be done in parallel to most of the ALU operations where a move operation can comprise data transport between a x/y register and a location in memory in either direction.

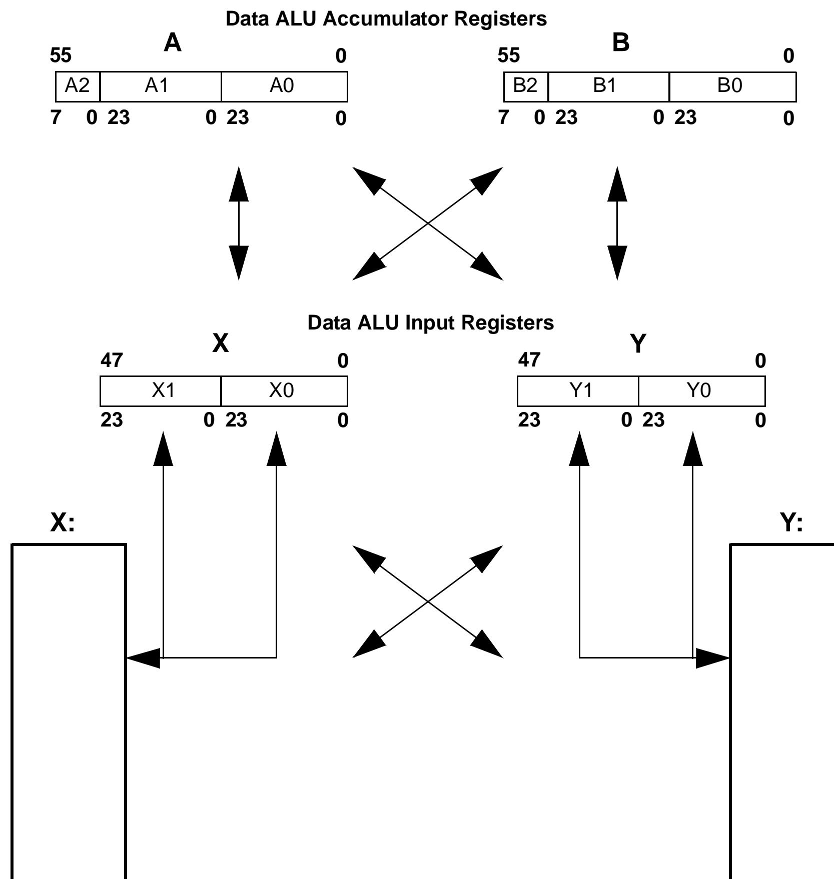


Figure 3: DSP56300 ALU Core and Data Transfer

This architecture provides the means to do most of the signal processing operations in a single cycle per input data value. The related address register updates (not shown in Figure 3) are described in exercise 2 (Addressing Modes).
Some rules due to hardware design impacts have to be followed: The two parallel moves cannot use the same bus (refer to core architecture, [1], p.1-3) because the moves are done in the same clock cycle. They should not have two of the same register types (x or y) as a destination.

Program Control Unit (PCU). The program control unit features Loop Address (LA) and Loop Count (LC) registers dedicated to supporting the hardware DO loop instruction in addition to the standard program flow-control resources, such as a Program Counter (PC), Status Register (SR), and System Stack (SS). All registers are read/write to facilitate system debugging. The diagram below illustrates the PCU programming model with the registers and the SS. For more details please refer to Section 6 in [1].

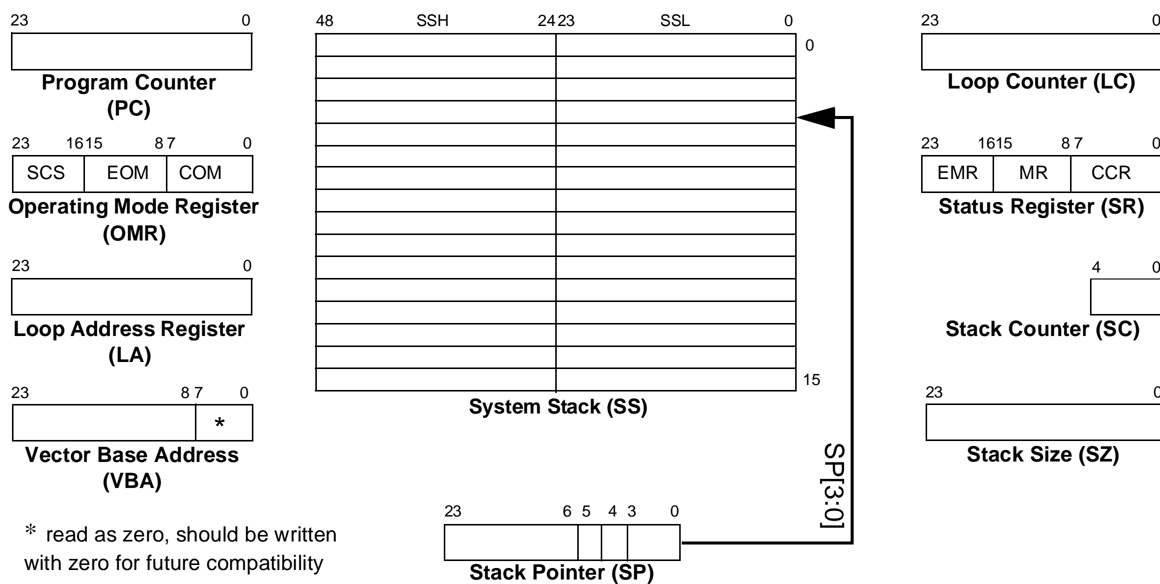


Figure 4: PCU Structure

Implementation Description

The implementation of this small program is performed in two parts that are both to be found in the same file: The data declaration and the program code section. In addition to this, a linker configuration file is provided and a text file where the running instructions shown below can be found as well. Thus, you have three files in your directory '/tutorial/ex1': ex1_main.asm, ex1.ctl, and ex1.txt.

In the file ex1.asm, the data declaration with the 'ds' (define storage) directive is just reserving memory without initialisation. Within the first lines of the program section the accumulators a and b are cleared. (A professional program would probably need a lot more initialisation.). The first action of the program is to initialize the address registers with the start address of the data buffers. The buffers in x: and y: are then initialised with increasing and decreasing (a and b) numbers, respectively. This happens in a hardware do-loop being executed SINGLE_LEN times ($40). Thus, we get two memory areas of length $40 words with known contents. The memory locations start at $100, this is done to show locating arrays in memory (it could be any address!).

When the memory initialisation is finished, the registers are initialised again to the start of the buffers in order to perform the mac-loop now. In the loop the overall sum of the $40 products is accumulated in a.

Test Run

1. Open the file ex1_main.asm and take a look at the code and the comments, respectively.

2. In the code, the memory initialisation is performed twice. This is done to demonstrate a major issue for an optimized single cycle DSP:

Programming with respect to the instruction pipeline.

Now, open a shell (dos/unix) and invoke the assembler, make sure that you are in the correct directory (tutorial/ex1).
'asm56300 -l -bex1_main.cln ex1_main.asm'

The assembler will issue two warnings: 'Pipeline stall reading register written in previous instruction' means that you are trying to access data that is still being processed in the ALU:
a is written in execute stage of pipeline (last stage):

```
    inc a              ; increment a
                       ;a is read in an earlier stage: CONFLICT
    move a0,x:(r0)+ ; write a0 to memory
```

This fact will be detected by the DSP hardware during runtime and a no-op will be automatically inserted to avoid an access to the register that is not updated yet (thus containing wrong data). The assembler checks this in advance and reminds you to write your code accordingly. If you do not mind a wait cycle, you may ignore these warnings. But the second initialisation procedure shows an example how to avoid the so called pipeline conflicts:
read the result a of the last loop cycle first

```
move a0,x:(r0)+        ; write a0 then calculate the new one.
inc a                  ; increment a
```

In almost all of the applications it is possible to organise the code such that no pipeline latency slows the code execution down.

NOTE: FURTHER READING The pipeline operation is described starting at page 7-1 in the Family Manual [1]. Please read paragraph 7.1.1 carefully to understand how the pipeline works. You may also refer to Appendix B in [1] for detailed information on certain instructions and to page 3-3 for the architecture of the data ALU. And finally you should read pages 3-20 through 3-22 to get an overview of the impacts on programming.

3. Now call the linker:

'dsplnk -mex1.map -rex1.ctl -bex1.cld ex1_main.cln'

This means, that the loadable file 'ex1.cld' is filled with the relocatable code in ex1_main.cln and all the addressing is made absolute using the memory configuration file 'ex1.ctl'. You can open this file to see where your code and data will be located in DSP memory during runtime. To verify this you can check the map file 'ex1.map' where all the symbols and sections are listed with their addresses.

4. Call the simulator now, look for the directory

/tutorial/ex1

and load 'ex1.cld'.

(MENU: File,Load, Memory COFF)

NOTE Loading a file brings the simulator into a state where you can start from. No explicit reset is required on top of this. The entry point (First instruction after reset) is defined in 'ex1.ctl'.

5. Open an assembly window (MENU: Windows,Assembly). Open two memory windows: (MENU: Windows, Memory then select x mem space, then do same with y space). Open a command window (MENU: Windows, Command) and a register window (MENU: Windows, Register). And finally open a watch window (MENU: Windows, Watch, then enter 'cyc' for cycles).

6. Do a few steps (step button), and check in the register window, if r0 and r1 are initialised correctly.

7. Now set a breakpoint to the symbol 'init_2': either (MENU: Execute,Breakpoints,Set and enter 'end_init' to the field 'start address') or just double click the address of the program line in the assembly window. Run the program by clicking on the green light. Check the written memory ranges in x: $100..$13f and y: $100..$13f. Both of them should contain in/decreasing numbers now. Data is correct in memory now and the cycle counter should show $191 = 401. This is:

6 cycles for pipeline init
6 cycles for the first four instructions
5 cycles for the 'do'
6 x 64 cycles for the loop

8. Now set a breakpoint to the symbol 'end_init', reset the cycle counter (enter: 'change cyc 0') and run the program again. Between break points the DSP did almost the same job this time and the cycle counter shows that you have saved 134 cycles now in comparison to the other solution (cyc should be $10b = 267 now).

9. Let the program now run until the end by setting another breakpoint to 'end_mac' and running the simulator again. Please check the accumulator a (a0, lower 24 bits of a) now for the result (register window), it should contain $fd6540.

10. Well done. If you were wondering why ($000001 * $ffffff) has the result $fffffe -> (1*(-1))=-2 (this was the result of the second loop run), please refer to the explanations of the fractional format of number representations in exercise 3.

EXERCISE 2 - ADDRESSING MODES

Introduction

This exercise describes the function of the Address Generation Unit (AGU) and its main objective is to illustrate:

- The DSP56300 Special Addressing Modes
- The Address Register Indirect operation
- The function of the Modifier Registers
- Modulo and Reverse Carry Addressing techniques

Technical Considerations

AGU Architecture. The AGU provides all the addressing modes required by DSP algorithms, such as modulo addressing for circular buffer generation and bit-reverse arithmetic for FFT butterflies. The AGU is divided into two halves, each of which has an address arithmetic logic unit (ALU) and three sets of registers. There are a total of eight independent pointer registers (R), eight offset registers (N) and eight modifier registers (M). All addresses are 24-bits supporting 16-Mwords in each memory space. Two address arithmetic units permit computation and generation of two 24-bit data addresses for dual operand access in every clock cycle.

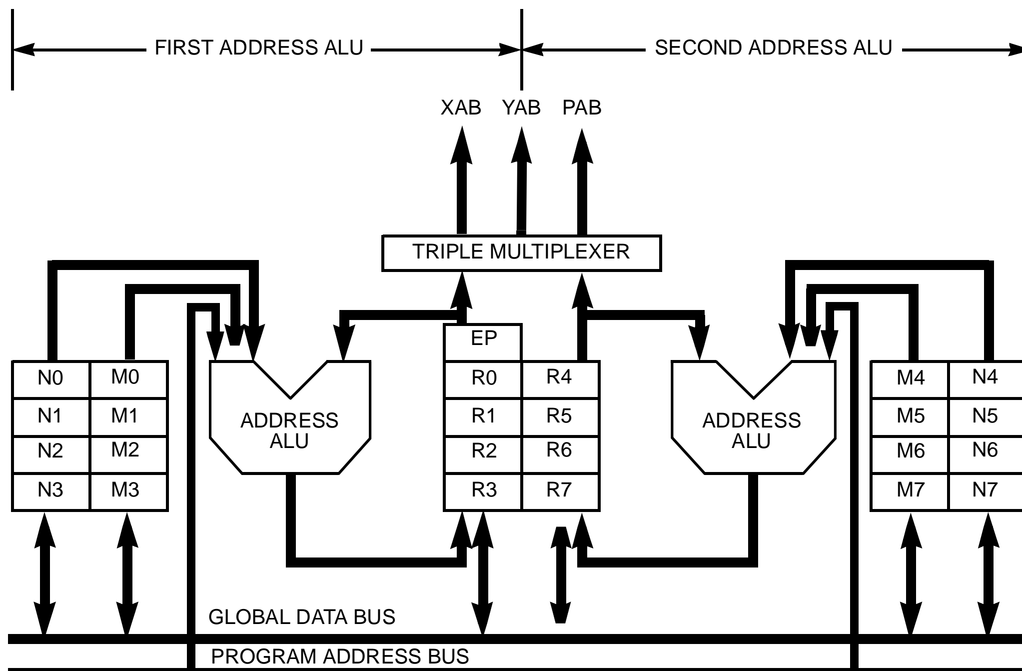


Figure 5: AGU Architecture

For more details please refer to [1], section 4.

The AGU Programming Model. The programmer's view of the AGU is three sets of eight registers, which can be used as temporary data registers and indirect memory pointers. Automatic updating is available when using address register indirect addressing.

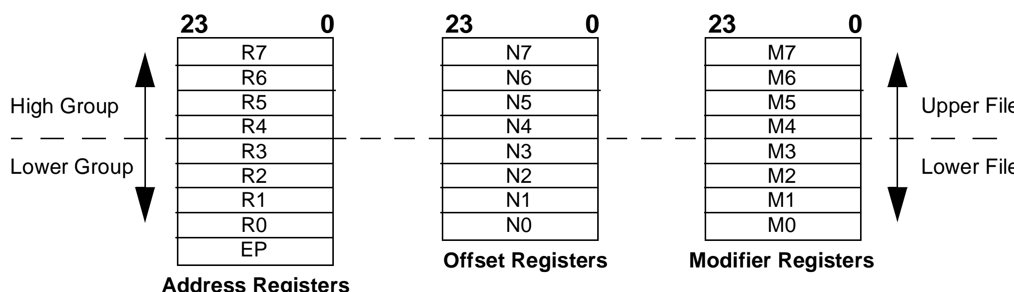


Figure 6: Address Registers

Each address register Rn has an associated offset register Nn and an associated modifier register Mn. The Rn registers are used as address pointers to locate data operands in memory, and can be programmed for linear addressing, modulo addressing (regular or multiple wrap-around), and bit-reverse addressing. The Nn registers are used to provide an offset value for offset updating of the address registers. The Mn registers select the type of address arithmetic to be performed when an address register is updated. The EP register (when enabled) is used to point to the stack extension in data memory and is referenced implicitly by instructions such as DO, JSR, RTI, etc. or directly by the MOVEC instruction. For a more detailed description on how to use the stack extension mode of operation, please refer to Section 6.3.5 in [1].

Address Modifier Types. As previously mentioned, the address modifier (Mn) defines the type of address arithmetic to be performed, and allows the user to create various data structures in memory, such as FIFOs, delay lines, circular buffers, stacks, and bit-reversed FFT buffers etc. The table below shows how the contents of the modifier register (Mn) select the type of address arithmetic to be performed. For more details please refer to [1], Section 4.

Modifier Mn	Address Calculation Arithmetic
XX0000	Reverse-Carry (Bit-Reverse)
XX0001	Modulo 2
XX0002	Modulo 3
:	:
XX7FFE	Modulo 32767 (2^{15}-1)
XX7FFF	Modulo 32768 (2^{15})
XX8001	Multiple Wrap-Around Modulo 2
XX8003	Multiple Wrap-Around Modulo 4
XX8007	Multiple Wrap-Around Modulo 8
:	:
XX9FFF	Multiple Wrap-Around Modulo 2^{13}
XXBFFF	Multiple Wrap-Around Modulo 2^{14}
XXFFFF	Linear (Modulo 2^{24})

Notes: **XX means don't care**

All other combinations are reserved

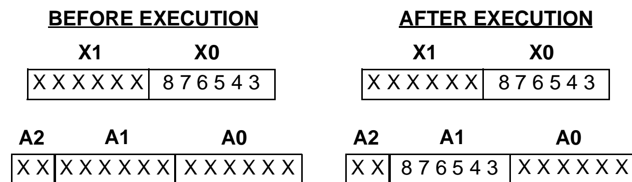
The addressing modes specify whether the operands are in registers and/or memory locations, and provide the specific address of the operands. The DSP56300 core provides four different addressing modes: register direct, address register indirect, special and PC relative.

The following sections attempt to illustrate the various addressing modes, it is recommended that the ex7_main.asm is made available as you read this document as the code will illustrate the examples discussed.

Register Direct Mode. This mode specifies that the operand is in one or more of the 10 data ALU registers (A2,A1,A0,B2,B1,B0,X1,X0,Y1,Y0), 24 address registers (R0-R7,N0-N7,M0-M7) or 7 control registers (OMR,SR,PC,VBA,LA,LC,SP).

Example: Move the contents of the 24-bit X0 data input register to the 24-bit A1 accumulator register.

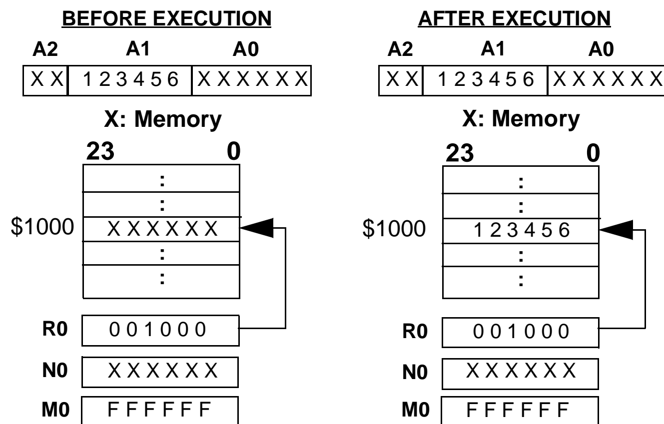
Figure 7: move x0,a1



Address Register Indirect Modes. These addressing modes specify an address register (Rn) to point to an operand stored in memory. They can also specify an address calculation to be performed either pre or post instruction execution. Each address register Rn is associated with an offset register Nn and a modifier register Mn. The Nn register contains an offset value which can be added to Rn to update its contents. The Mn register specifies the type of address arithmetic to be performed when Rn is updated. Mn is set to $FFFFFF upon reset to specify linear address arithmetic.

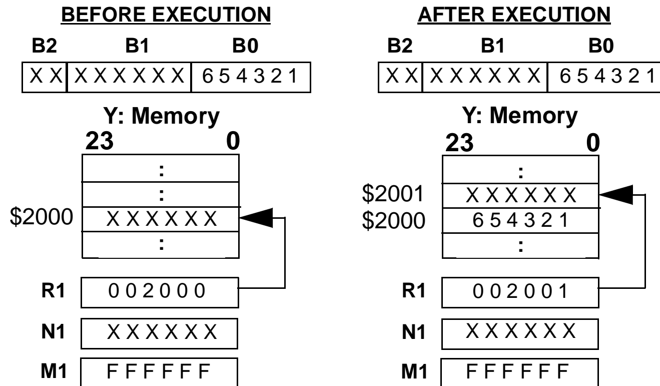
No Update (Rn) - Example: Transfer the contents of accumulator register A1 to the X-Memory location pointed to by address register R0.

Figure 8: move a1,x:(r0)



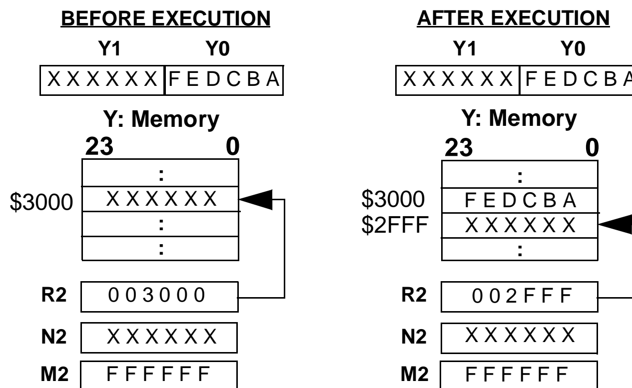
Postincrement by one (Rn)+ - Example*:* Transfer the contents of accumulator register B0 to the Y-Memory location pointed to by address register R1. Once the transfer is complete, R1 is incremented by one.

Figure 9: move b0,y:(r1)+



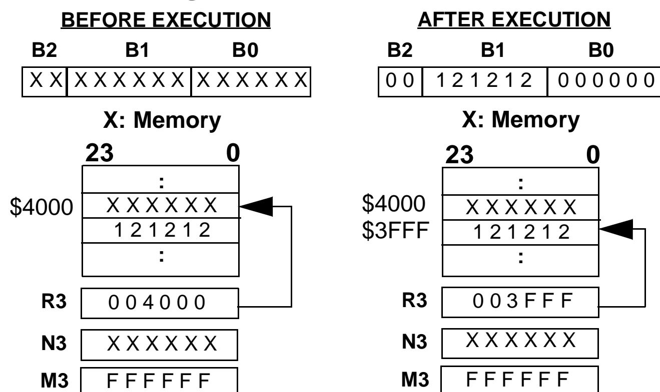
Postdecrement by one (Rn)- - Example. Transfer the contents of data register Y0 to the Y-Memory location pointed to by address register R2. Once the transfer is complete, R2 is decremented by one.

Figure 10: move y0,y:(r2)-



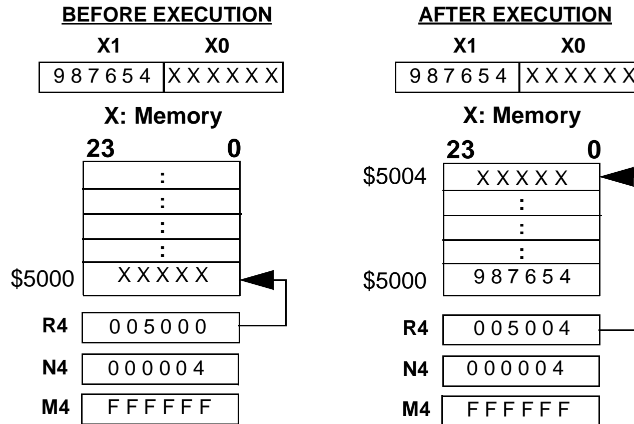
Predecrement by one -(Rn) - Example. Predecrement address register R3 by one and transfer the X-Memory location pointed to by the decremented address register R3 to accumulator register B

Figure 11: move x:-(r3),b



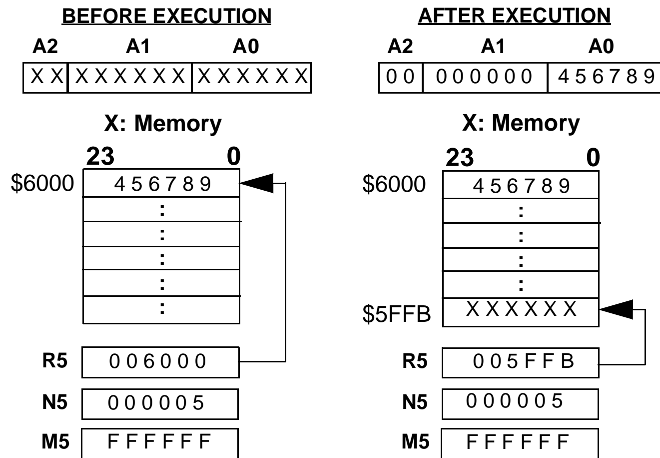
Postincrement by Offset (Rn)+Nn - Example . Transfer the contents of data input register X1 to the X-Memory location pointed to by address register R4. Once the transfer is complete, R4 is updated by adding the offset contained in offset register N4 to the contents of R4.

Figure 12: move x1,x:(r4)+n4



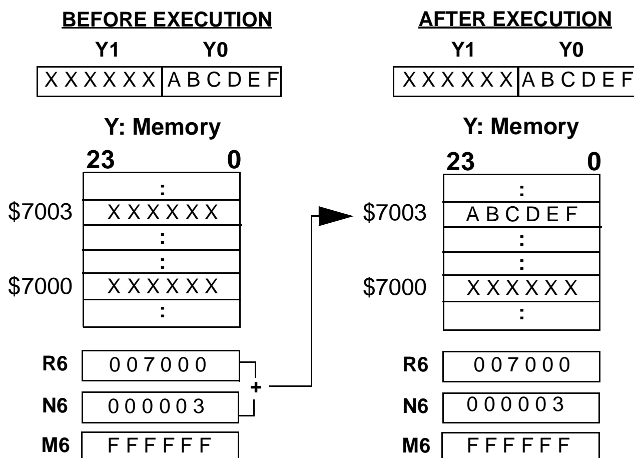
Postdecrement by Offset (Rn)-Nn - Example. Transfer the contents of the X-Memory location pointed to by address register R5 to accumulator A0. Once the transfer is complete, R5 is updated by subtracting the offset contained in offset register N5 from the contents of R5.

Figure 13: move x:(r5)-n5,a0



Indexed by Offset (Rn+Nn) - Example. Transfer the contents of data input register Y0 to the X-Memory location pointed to by the summation of the contents of the address register R6 and the offset register N6. Note that address register R6 is not updated.

Figure 14: move y0,x:(r6+n6)



Modifier Register Usage. The eight 24-bit modifier registers (M0-M7) specify the type of address arithmetic to be performed for addressing mode calculations, or can be used for general-purpose storage. The address ALU supports linear, modulo and reverse-carry arithmetic for all address register indirect modes.

Modulo Addressing (Mn = Modulus-1). For modulo arithmetic, the contents of Mn specifies the modulus i.e. circular buffer/table size. The following diagram illustrates the procedure taken for modulo register set-up, for a buffer/table size of 20 elements and an increment offset (Nn) of 3.

Modulo Register Setup:

1. MODULUS = M
Modifier register Mn = M-1

2. Beginning of Table
Lower bound = $2^J \geq M$

3. End of Table
Upper bound = $2^J + M - 1$

4. Starting point within Table
Lower bound ≤ Address register Rn ≤ Upper bound

5. Desired Increment (if any)
Offset register Nn = Increment ≤ M

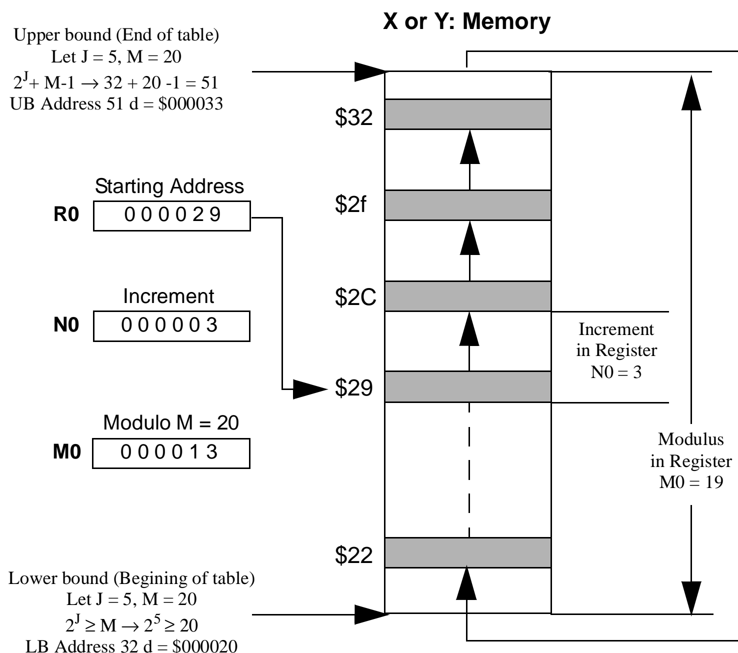


Figure 15: Modulo Adressing

Reverse-Carry Addressing (Mn = $000000). Reverse carry is selected by setting the modifier register to zero. The address modification is performed in hardware by propagating the carry in the reverse direction i.e. from the MSB to the LSB. Reverse carry is equivalent to bit reversing the contents of Rn (i.e. redefining the MSB as the LSB, the next MSB as bit 1, etc.) and the offset value, Nn, adding normally, and then bit reversing the result. If the + Nn addressing mode is used with this address modifier and Nn contains the value $2^{(k-1)}$ (a power of two), this addressing modifier is equivalent to bit reversing the k LSBs of Rn, incrementing Rn by 1, and bit reversing the k LSBs of Rn again.

To illustrate the address reordering technique, consider each element of the input sequence labelled ‘orgdatar’ in the file ‘ex2_main.asm’ and its associated binary base address $40=1000000. For an 8-point (eight-element) input data sequence (i.e. orgdatar) the three least significant bits (LSB) of the associated binary addresses are 000,001,002,...,111 respectively. To reorder the addresses of the data input sequence, the m LSBs ($2^{(k-1)} = 3$) of the address of each sequence element must be ‘Bit-Reversed’ as shown below:

Bit Reverse Register Setup:
LeT Modifier Register Mn = $000000
Let offset register Nn = 2^{k-1}
Let the number of points (Data elements/Table size) = 8
Let beginning of table (lower bound) = $2^k \geq N$, where N = Table size
Let the end of table (upper bound) = $2^k - 1$
Let starting point within table = Lower bound ≤ Address register Rn ≤ Upper bound

Original Address Input Sequence	Original Data Input Sequence	Bit-Reversed Address Modification (3 LSBs)	Resulting Input Sequence in Bit-Reverse Order
$8 -> 1000	#0.0	1000 = $8 +1	$8 -> #0.0
$9 -> 1001	#0.1	1100 = $C +1	$9 -> #0.4
$A -> 1010	#0.2	1010 = $A +1	$A -> #0.2
$B -> 1011	#0.3	1110 = $E +1	$B -> #0.6
$C -> 1100	#0.4	1001 = $9 +1	$C -> #0.1
$D -> 1101	#0.5	1101 = $D +1	$D -> #0.5
$E -> 1110	#0.6	1011 = $B +1	$E -> #0.3
$F -> 1111	#0.7	1111 = $F	$F -> #0.7

Figure 16: Bit-Reverse Addressing

This address modification is useful for addressing the twiddle factors in 2k-point FFT addressing and to unscramble 2^k-point FFT data. The range of values for Nn is 0 to + 8M i.e. Nn=2^{23}, which allows bit-reverse addressing for FFTs up to 16,777,216 points.

Special Addressing Modes. They do not use an address register in specifying an effective address. These modes specify the operand or the address of the operand in a field of the instruction or they implicitly reference the operand.

Immediate Data. The immediate data addressing mode points to a 24-bit operand located in the extension word of the instruction.

Immediate Data into a 24-Bit Accumulator - Example. Transfer the immediate value $123456 to Accumulator Register A0.

Figure 17: move #$123456,a0

BEFORE EXECUTION			AFTER EXECUTION		
A2	A1	A0	A2	A1	A0
X X	X X X X X X	X X X X X X	X X	X X X X X X	1 2 3 4 5 6

Positive Immediate Data into a 56-Bit Accumulator - Example. Transfer the immediate value $654321 to accumulator register A. Note that the accumulator register A1 is loaded and that accumulator register A2 is sign-extended from A1 i.e. A2 holds the value $00 indicating that the value stored in A1 is positive.

Figure 18: move #$654321,a

BEFORE EXECUTION			AFTER EXECUTION		
A2	A1	A0	A2	A1	A0
X X	X X X X X X	X X X X X X	0 0	6 5 4 3 2 1	0 0 0 0 0 0

Negative Immediate Data into 56-bit Accumulator - Example: Transfer the immediate value $876543 to accumulator register B. Note that the accumulator register B1 is loaded and that accumulator register B2 is sign-extended from B1 i.e. B2 holds the value $FF indicating that the value stored in B1 is negative.

Figure 19: move #$876543,b

BEFORE EXECUTION			AFTER EXECUTION		
B2	B1	B0	B2	B1	B0
X X	X X X X X X	X X X X X X	F F	8 7 6 5 4 3	0 0 0 0 0 0

Immediate Short Data. The immediate short addressing mode points to an 8-bit or a 12-bit immediate data operand located in the instruction operation word. The immediate data is interpreted as an unsigned integer. if the destination register is one of the following 24-bit registers A2, A1, A0, B2, B1, B0, R0-R7 or N0-N7. The immediate data is transferred into the least significant bits of the destination with the most significant bits zeroed. The immediate data is interpreted as a signed fraction if the destination is one of the following 24-bit registers X1, X0, Y1, Y0, or the 56-bit A and B accumulators. The immediate data is transferred into the most significant bits of the destination with the least significant bits zeroed.

Immediate Short Data into 24-Bit Register - Example: Transfer the immediate short data value $FE to accumulator register A1. Note that the immediate data $FE is interpreted as an unsigned Integer and is transferred into the least significant bits of A1.

Figure 20: move #$FE,a1

BEFORE EXECUTION			AFTER EXECUTION		
A2	A1	A0	A2	A1	A0
X X	X X X X X X	X X X X X X	X X	0 0 0 0 F E	X X X X X X

Example: Transfer the immediate short data value $FE to data input register Y1. Note that the immediate data $FE is interpreted as a signed fraction and is transferred into the most significant bits of Y1.

Figure 21: move #$FE,y1

BEFORE EXECUTION		AFTER EXECUTION	
Y1	Y0	Y1	Y0
X X X X X X	X X X X X X	F E 0 0 0 0	X X X X X X

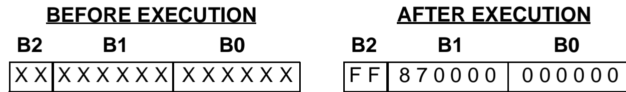
Immediate Short Data into 56-bit Accumulators - Example: Transfer the immediate short data value $34 to the accumulator register A. Note that the immediate data $34 is interpreted as a signed fraction and is transferred into the most significant bits of the accumulator register A and that accumulator register A2 is sign-extended.

Figure 22: move #$34,a

BEFORE EXECUTION			AFTER EXECUTION		
A2	A1	A0	A2	A1	A0
X X	X X X X X X	X X X X X X	0 0	3 4 0 0 0 0	0 0 0 0 0 0

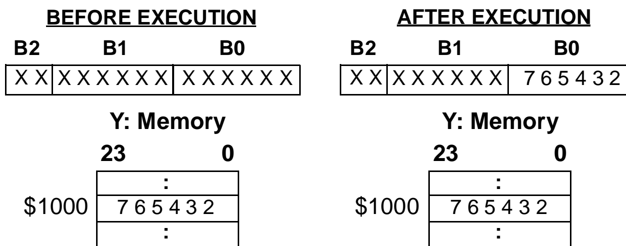
Example. Transfer the immediate short data value $87 to the accumulator register B. Note that the immediate data $87 is interpreted as a signed fraction and is transferred into the most significant bits of the accumulator register B and that accumulator register B2 is sign-extended.

Figure 23: move #$87,b



Absolute Addressing. The absolute addressing mode uses the 24-bit address operand located in the instruction extension word as a pointer to the location of the data operand.
Example: Transfer the contents of the Y-Memory location pointed to by the instruction extension word to accumulator register B0.

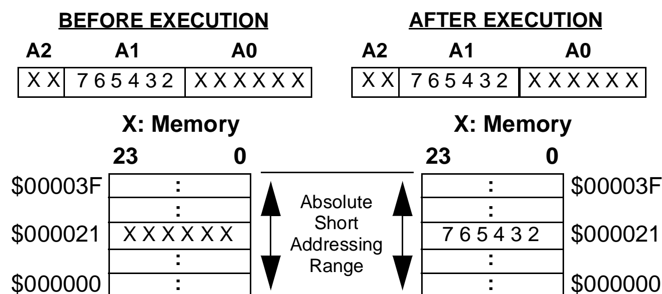
Figure 24: move y:$1000,b0



Absolute Short Addressing. This mode uses an immediate 6-bit address operand which is located in the instruction operation word and is zero-extended to form a 24-bit pointer to the data operand. This mode addresses the lowest 64 words (range 0-63 d, $0-$3F) of X,Y,L data RAM and interrupt vectors.

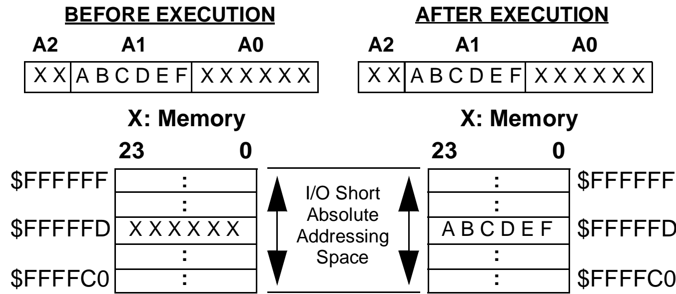
Example: Transfer the contents of the accumulator register A1 to the X-Memory location pointed to by the instruction extension word.

Figure 25: move a1,x:$21



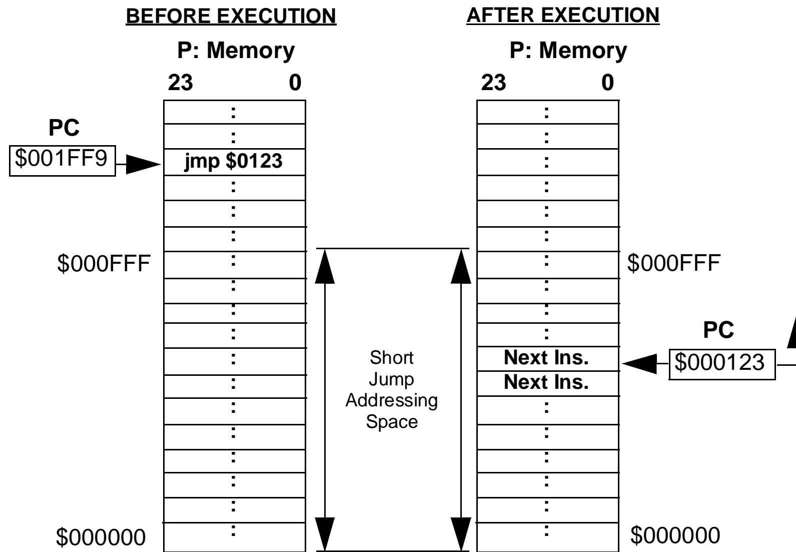
I/O Short Addressing. The I/O short addressing is similar to the absolute short addressing mode, it also uses an immediate 6-bit address operand which is located in the instruction operation word but is ones-extended to form a 24-bit pointer to the data operand rather than zero-extended. This mode addresses the highest 64 words (range 16777152-16777215 d, $FFFFC0-$FFFFFF) of X or Y memory, and is used with the bit manipulation and move peripheral data instructions.

Figure 26: movep a1,x:$FFFFFD



Short Jump Addressing. The short jump addressing mode uses a 12-bit immediate jump operand which is located in the instruction operation word and is zero-extended to form a 24-bit “jump to” operand, which is used to replace the contents of the program counter (PC). This mode addresses the lowest 4096 words (range 0-4095 d, $0-$000FFF) of program-RAM.

Figure 27: jmp <$123



Program Counter Relative Modes. In the program counter relative addressing modes, the address of the operand is obtained by adding a displacement, represented in two’s complement format, to the value of the program counter (PC). The PC points to the address of the instruction’s opcode word. The Nn and Mn registers are ignored, and the arithmetic used is always linear.

Short Displacement PC Relative. The short displacement occupies 9-bits in the instruction operation word. The displacement is first sign extended to 24 bits and then added to the PC to obtain the address of the operand.

Long Displacement PC Relative. This addressing mode requires one word of instruction extension. The address of the operand is the sum of the contents of the PC and the extension word.

Address Register PC Relative. The address of the operand is the sum of the contents of the PC and the address register Rn. The Mn and Nn registers are ignored. The contents of the Rn register are unchanged.

Parallel Data Move Descriptions. Thirty of the sixty-two DSP56300 core instructions allow an optional parallel data bus movement over the X and/or Y data bus. This allows a data ALU operation to be executed in parallel with up to two data bus moves during one instruction/clock cycle. Ten types of parallel moves are permitted, including register to register moves, register to memory moves, and memory to register moves. For example, the parallel XY memory data move must specify two independent effective addresses (e.g. (opcode/operand (<eax> and <eay>) --> add Y, A A,x:(r1)+n1 y1,y:(r5)+)) where one of the effective addresses must use the lower bank of address registers (R0–R3) while the other effective address must use the upper bank of address registers (R4–R7). However, not all addressing modes are allowed for each type of memory reference. The following paragraphs provide some examples of parallel move operations.

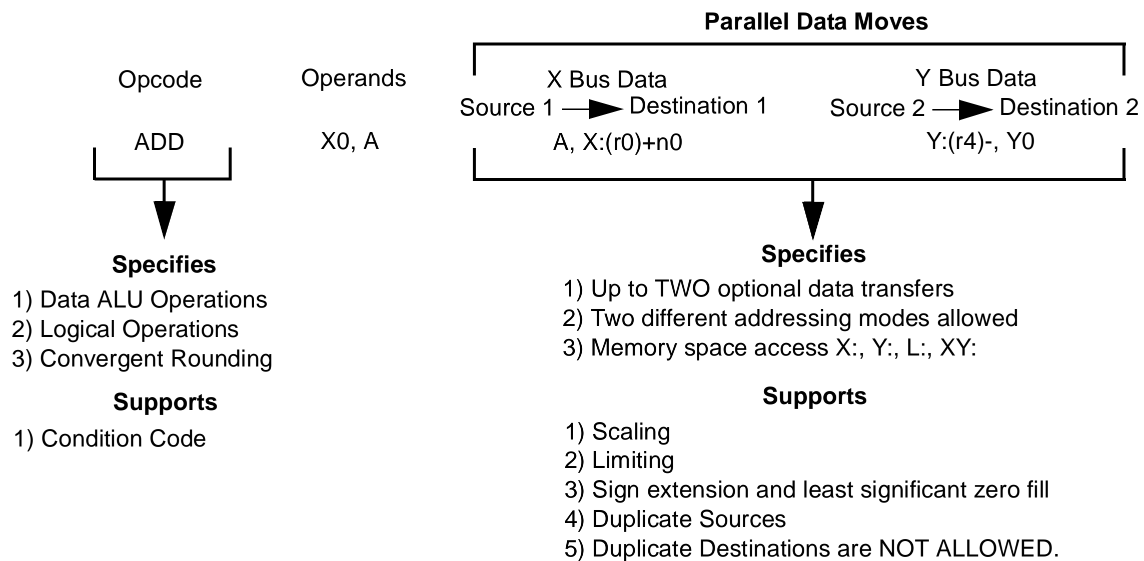


Figure 28: Parallel Data Move Instruction Syntax

Parallel Data Move Examples

	Opcode/Operand	Source 1 → Destination 1	Source 2 → Destination 2
Immediate Short Data Move	ADD A, B	#$81, A1	
Register to Register Move	ADD X0, A	A1, Y0	
Address Register Update	ADD Y1, A	(r0)+n0	
X or Y Memory Move	ADD X1, B	B, X:$1000	
X or Y Mem. and Reg. Move	ADD Y0, A	A, X1	Y:(r2)-, Y0
L Memory Move	ADD X, A	A10, L:$3000	
XY Memory Move	ADD Y, B	B, X:(r3)+n3	Y:(r7)+n7, Y1

Multiply and Accumulate Instructions

MAC ± S1, S2, D [Parallel Move]	D ± (S1 * S2) --------> D	MAC X0, Y0, A	X:(r0)+, X0	Y:(r4)-, Y0
MACR ± ±S1, S2, D [Parallel Move]	D ± (S1 * S2) + r ----> D	MACR -Y0, Y0, B	X1, X:(r1)+	B, Y:(r5)-

NOTE: each of these parallel move examples is executed in one single clock/instruction cycle which means each example is executed in <u>15.2nsec @ 66MHz</u>, or <u>12.5nec @ 80MHz</u>

Figure 29: Examples of Parallel Data Moves

Test Run

1. Open the file ex2_main.asm in a text editor and take a look at the code and the comments, respectively. The program contains many short examples of the various addressing modes that exist within the 56300 family such as: Register Direct/Indirect, Immediate Data/Short, Absolute Addressing, Short Addressing, Modulo and Bit-Reverse etc.

2. Assemble the main file by opening a dos shell/unix command window and from within the correct directory (/ex2) typing:

```
asm56300 -l -b ex2_main.asm
```

This will create two new files: ex2_main.cln, which is the file to be passed to the linker and ex2_main.lst, which is a list file generated by the assembler.

NOTE: If this does not work correctly, ensure that the default path of the machine was correctly set up during the installation procedure. During assembly, four warnings will be generated due to pipeline stalls. For detailed information on this, please refer to exercise one and the DSP56300 family manual [1].

3. View the file ex2.ctl in a text editor window. This is the file which the linker will reference to decide where to place sections of memory. In this example, data is located at address x:$100 and y:$100 and the program code is stored at p:$100.

4. Call the linker to link these files together into an absolute object file which the simulator can load. Do this by typing:

```
dsplnk -mex2.map -rex2.ctl -bex2.cld ex2_main.cln
```

This means that ex2_main.cln will be linked and located using the instructions in the ex2.ctl linker control file. The output will be a machine loadable file called ex2.cld, and a map file (ex2.map) showing the location of sections in memory.

5. Start the simulator. This action is dependant on your development environment - please refer to [3] for instructions. If the simulator is already running, RESET the device.

6. If they are not already open, open a:

Session window (MENU: Windows, Session)
Command window (MENU: Windows, Command)
Assembly window (MENU: Windows, Assembly)

The session window will show the state of the device following each step. The command window can be used to input commands directly or will show the commands executed using the menus. The assembly window will show the code in program memory, and will indicate the next instruction to be executed.

7. Open two memory windows, one for X memory and one for Y memory, each starting at address $100. The input and output matrices appear in these windows.

8. Load the program ex2.cld into the simulator
   (MENU: File, Load, Memory COFF)
   Note that the program is now loaded and displayed in the assembly window.

9. As the program demonstrates various addressing modes of the DSP56300, it is best to split-up the function of each short routine by inserting breakpoints as follows: Set breakpoints at the following labels/addresses by calling the prepared macro
   ‘runex2.cmd', MENU File, Macro:

   1st, LABEL: ex2_start ADDRESS: $100
   2nd, LABEL: indirect_start ADDRESS: $103
   3rd, LABEL: postinc_start ADDRESS: $108
   4th, LABEL: predec_start ADDRESS: $10D
   5th, LABEL: incoffset_start ADDRESS: $115
   6th, LABEL: indexoff_start ADDRESS: $11B
   7th, LABEL: modulo_start ADDRESS: $121
   8th, LABEL: bitrev_start ADDRESS: $128
   9th, LABEL: immed_start ADDRESS: $139
   10th, LABEL: posimmed_start ADDRESS: $13B
   11th, LABEL: negimmed_start ADDRESS: $13D
   12th, LABEL: intshort_start ADDRESS: $13F
   13th, LABEL: fractshort_start ADDRESS: $140
   14th, LABEL: negfractshort_start ADDRESS: $141
   15th, LABEL: absolute_start ADDRESS: $142
   16th, LABEL: abshort_start ADDRESS: $148
   17th, LABEL: parallel_start ADDRESS: $14B

10. Instruct the simulator to go. It will break once it has completed the operations between each breakpoint (highlighted in blue). Do not be surprised if it takes a second or so to break! This is a cycle exact simulator and therefore requires a lot of processing power.

11. The following notes point out what actions you should be looking out for during each address mode i.e. each conditional breakpoint:

   1st Breakpoint: Register Direct Example:
   Moves the contents of the 24-bit X0 data input reg. to the 24-bit A1 accumulator

   2nd Breakpoint: Address Register Indirect Example: No Update (Rn)
   Moves the contents of the 24-bit A1 accumulator reg. to X-Memory location pointed to by address reg. r0.

3rd Breakpoint: Postincrement (Rn)+
Moves the contents of accumulator reg. B0 to the Y-Memory location pointed to by r1. Once the transfer is complete, r1 is incremented by one.

4th Breakpoint: Predecrement -(Rn)
Predecrement address reg. r3 by one and transfer the X-Memory location pointed to by the decremented address reg. r3 to the accumulator reg. B.

5th Breakpoint: Postincrement by offset (Rn)+Nn
Moves the contents of data reg. X1 to the X-Memory location pointed to by address reg. r4. Once the transfer is complete, r4 is updated by adding the offset contained in n4 to the contents of r4.

6th Breakpoint: Indexed by offset (Rn+Nn)
Moves the contents of data reg. Y0 to the X-Memory location pointed to by the summation of the contents of the address reg. r6 and the offset reg. n6. Note that r6 is not updated.

7th Breakpoint: Modulo Addressing
The Modifier Registers Mn are used for Modulo Arithmetic and specifies the modulus i.e. circular-buffer/table-size. The example code starting at label "modulo_start" shows the procedure taken for modulo register set-up. R0 points to the data stored in X-Memory defined by: const dc 0,1,2,3,4,5,6,7,8,9. There are 10 items stored, so we therefore set the circular-buffer to 10-1 = 9 and is stored in M0 to define Modulo addressing, we have also set up an offset increment of 3, which is stored in N0. The example now executes a do loop 10 times which moves the contents of the X-Memory location pointed to by address reg. r0 to the data reg. X0. Once the transfer is complete, r0 is updated by adding the offset contained in n0 to the contents of r0, the process is then repeated. The steps taken are:

r0 = $50 X0 = 0 then r0 is updated by n0, r0+n0 = 53
r0 = $53 X0 = 3 then r0 is updated by n0, r0+n0 = 56
r0 = $56 X0 = 6 then r0 is updated by n0, r0+n0 = 59
r0 = $59 X0 = 9 then r0 is updated by n0, r0+n0 = 52
r0 = $52 X0 = 2 then r0 is updated by n0, r0+n0 = 55
r0 = $55 X0 = 5 then r0 is updated by n0, r0+n0 = 58
r0 = $58 X0 = 8 then r0 is updated by n0, r0+n0 = 51
r0 = $51 X0 = 1 then r0 is updated by n0, r0+n0 = 54
r0 = $54 X0 = 4 then r0 is updated by n0, r0+n0 = 57
r0 = $51 X0 = 7 then r0 is updated by n0, r0+n0 = 50

8th Breakpoint: Bit-Reverse Addressing
Reverse carry is selected by setting the modifier register Mn to zero. The address modification is performed in hardware by propagating the carry in the reverse direction i.e. from the MSB to the LSB. The example code starting at label "bitrev_start" shows the procedure taken for bit reverse register set-up. R0 points to the original input data stored in X-Memory defined by: orgdatar dc 0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8
R4 points to the original input data stored in X-Memory defined by:
orgdatai dc 0.15,0.25,0.35,0.45,0.55,0.65,0.75,0.85
The modifier registers m0 and m4 are set to zero for bit-reverse addressing, the offset registers n0 and n4 are set to the number of data elements divided by two (i.e. 4). R1 and r5 are set to point to the Modified Bit-Reversed Address New I/P data sequence (i.e. $200), and the m1, m5 modifier registers are set to $ffffff for linear addressing.

The example now executes a do loop 8 times which bit-reverses the original addresses in X- and Y-Memory, and stores the re-ordered data into the new addresses, as shown below:

```
X-MEMORY CONTENTS:-
Org. address I/P data seq. ---- Modified bit-reversed address new I/P data seq.
$100 -------> $0ccccd          $200 ----------------> $0ccccd
$101 -------> $19999a          $201 ----------------> $400000
$102 -------> $266666          $202 ----------------> $266666
$103 -------> $333333          $203 ----------------> $59999a
$104 -------> $400000          $204 ----------------> $19999a
$105 -------> $4ccccd          $205 ----------------> $4ccccd
$106 -------> $59999a          $206 ----------------> $333333
$107 -------> $666666          $207 ----------------> $666666

Y-MEMORY CONTENTS:-
Org. address I/P data seq. ---- Modified bit-reversed address new I/P data seq.
$100 -------> $133333          $200 ----------------> $133333
$101 -------> $200000          $201 ----------------> $466666
$102 -------> $2ccccd          $202 ----------------> $2ccccd
$103 -------> $39999a          $203 ----------------> $600000
$104 -------> $466666          $204 ----------------> $200000
$105 -------> $533333          $205 ----------------> $533333
$106 -------> $600000          $206 ----------------> $39999a
$107 -------> $6ccccd          $207 ----------------> $6ccccd
```

9th Breakpoint: Immediate Data
Moves the immediate 24-bit data value to the 24-bit accumulator A0

10th Breakpoint: Positive Immediate Data
Moves the immediate 24-bit data value to the 56-bit accumulator A. Note that the accumulator reg. A1 is loaded and that the accumulator reg. A2 is sign-extended from A1 i.e. A2 holds the value $00 indicating that the value stored in A1 is positive.

11th Breakpoint: Negative Immediate Data
Moves the immediate 24-bit data value to the 56-bit accumulator B. Note that the accumulator reg. B1 is loaded and that the accumulator reg. B2 is sign-extended from A1 i.e. B2 holds the value $FF indicating that the value stored in A1 is negative.

12th Breakpoint: Immediate Short Data into 24-bit Accumulator Registers
Move the immediate short data value to accumulator reg. A1. Note that the immediate data is interpreted as an unsigned integer and is transferred into the least significant bits of A1.

13th Breakpoint: Immediate Short Data into 24-bit Accumulator Registers
Move the immediate short data value to data input reg. Y1. Note that the immediate data is interpreted as an signed fraction and is transferred into the most significant bits of Y1.

14th Breakpoint: Immediate Short Data into 56-bit Data Registers
Move the immediate short data value to accumulator reg. B. Note that the immediate data is interpreted as an signed fraction and is transferred into the most significant bits of the accumulator B and that B2 is sign-extended.

15th Breakpoint: Absolute Addressing
Moves the contents of the Y-Memory location pointed to by the instruction extension word to accumulator reg. B0.

16th Breakpoint: Short Addressing
Moves the contents of the accumulator reg. A1 to the X-Memory location short address.

17th Breakpoint: Parallel Data Moves
This example shows you how the 56300 allows a data ALU operation (add y,b) to be executed in parallel with up to two data bus moves (b,x:(r3)+n3 y:(r7)+n7,y1) during one instruction cycle.

12. You should now be more familiar with the various addressing modes of the 56300.

13. Congratulations......you have completed Exercise 2.

EXERCISE 3- DIVISION ON THE DSP56300

Introduction

The DSP56300 core is a 24-bit, fixed point, two's complement signed, fractional DSP. Due to the fractional nature of the DSP, in some cases the arithmetic requires some additional thought i.e. if you assume the values are integers then the results of a multiplication may appear incorrect. This section gives some examples of the fractional arithmetic and what effect this has on the coding. The particular example shown is division.

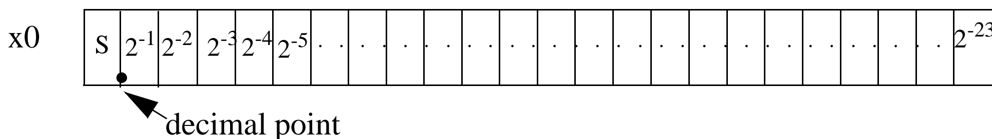
In this exercise you should

- Become more familiar with the fractional arithmetic of the DSP56300
- Learn how to implement a signed fractional division
- Become familiar with stepping, setting breakpoints, and evaluating expressions using the DSP56300 simulator

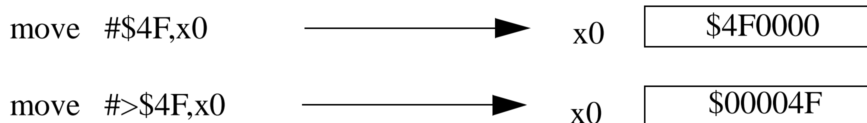
Details of working with fractional and integer arithmetic can be found in [4]. Although this document was written for the DSP56000 most of it applies equally to the DSP56300.

Technical Considerations

Data Representation. The DSP56300 uses a fractional data representation for all data ALU (Arithmetic Logic Unit) operations. This means that in a normal 24-bit ALU register, the most negative value which can be represented is -1 ($800000) and the most positive number which can be represented is $1-2^{-23}$($7FFFFF). This means that the user must be aware of how the DSP manipulates the data. The diagram below shows how the DSP sees a typical ALU register:



This fractional data representation has an effect on almost everything the DSP does in the ALU. For example, if your assembly code contains an instruction 'move #$4F,x0' then the value stored in x0 is not $00004F as you might expect, but $4F0000. The fractional nature of the data representation causes everything to be automatically aligned to the left. If on any occasion you want to have the data value right aligned then this can be achieved using the force long operator, e.g.



Multiplication can also have unexpected results if the fractional arithmetic is not taken into account. For example, if x0 contains $040000 and y0 contains $020000, then the result calculated by the instruction 'mpy x0,y0,a' is a1 = $001000. However, consider the fractional representation and it makes sense:

\$040000 is equivalent to 2^{-5} which is equivalent to 0.03125

\$020000 is equivalent to 2^{-6} which is equivalent to 0.015625

\$001000 is equivalent to 2^{-11} which is equivalent to 0.0004882812

$2^{-5} \times 2^{-6} = 2^{-11}$

Division on the DSP56300. Division is another arithmetic operation on the DSP56300 which may appear complex and produces unusual results on first sight. Demonstrating the process of division is the main aim of this exercise.

The DSP56300 has a DIV instruction. In the explanation of this instruction in [1] states that this instruction performs a 'divide iteration'. This means that the DIV instruction must be repeated a number of times depending on the accuracy of result required.

Each DIV operation calculates one quotient bit using a non-restoring division algorithm. For details of the algorithm please refer to [1]. Due to the non-restoring nature of the algorithm the remainder which exists following the DIV instruction is not the true remainder. The last div instruction must effectively be reversed to give the true remainder. The remainder is a 48-bit value with 24-bits of accuracy. It is essential that it is interpreted this way.

Division of two signed fractions. Each form of division (i.e. signed fractions, unsigned fractions, signed integers, double precision etc.) requires slightly different programming, and these are explained in [4].

In this example we shall implement the division of two signed fractions. The subroutine which executes this division (SIG24DIV) is shown below. This code implements a 4 quadrant divide (i.e. a signed dividend and a signed divisor.) Within the main divide part of the routine both variables are positive; the signs of the original inputs are saved at the beginning of the routine and restored at the end. This means that, if your algorithm ensures that both operands are always positive, the routine can be greatly simplified.

```
;Before execution of subroutine, dividend is in accumulator a, divisor is in
register x0.

;After execution of subroutine quotient is in x1, remainder is in b1

SIG24DIV

       abs  a    a,b       ;make dividend positive, copy a1 to b1
       eor  x0,b b,y0      ;save rem. sign in y0, quo sign in N
       and  #$FE,ccr       ;clear carry bit C (quotient sign bit)
       rep  #$18           ;form a 24-bit quotient
       div  x0,a           ;form quotient in a0, remainder in a1
       tfr  a,b            ;save remainder and quotient in b
       jpl  saveq          ;if quotient is positive, go to saveq
       neg  b              ;complement quotient if N bit is set
saveq  tfr  x0,b b0,x1     ;saveq. in x1, get signed divisor
       abs  b              ;get absolute value of signed divisor
       add  a,b            ;restore remainder in b1
       jclr #23,y0,done    ;go to done if remainder is positive
       move #$0,b0         ;prevent unwanted carry
       neg  b              ;complement remainder
done
       rts
```

There are four signed division examples in ex3_main.asm, showing the different cases which are accounted for in the code, i.e. positive dividend, positive divisor; positive dividend, negative divisor; negative dividend positive divisor; and negative dividend, negative divisor. However, the results have to be interpreted correctly. Two examples are shown below:

Positive dividend, positive divisor.

$$\frac{\$00065443563445}{\$123456} = \frac{0.04944650373}{0.1422222} = 0.347670778$$

Before execution:

x0	$123456	a	$00: 065443: 563445

After execution of SIG24DIV:

x1	$2c807a	b	$00: 16524d: 2c807a

This is to be interpreted as the quotient = $2c807a, and the remainder = $00000016524d

Verifying this result using standard arithmetic:
  $123456 is equivalent to 0.1422222, +$00065443563445 is equivalent to 0.04944650373
  $2c807a is equivalent to 0.3476708, $00000016524d is equivalent to 1.04 x 10 $^{-8}$
This result is correct to 24-bit accuracy

Negative dividend, positive divisor.

$$\frac{\$FFFF8734749837}{\$123456} = \frac{-0.003686373776}{0.1422222} = -0.025919814$$

Before execution:

x0	$123456	a	$FF: FF8734: 749837

After execution of SIG24DIV:

x1	$FCAEA9	b	$FF: F696AB: 000000

Verifying these results using standard fractional arithmetic:
  $123456 is equivalent to 0.1422222, $FFFF8734749837 is equivalent to -0.003686373776
  $FCAEA9 is equivalent to -0.0259198, $FFFFFFF696AB is equivalent to -4.3825 x 10 $^{-9}$
This result is correct to 24-bit accuracy.

Test Run

1. The first thing we want to do is demonstrate the basic effects of the fractional arithmetic. Rather than do this using a pre-written program, we will use the in-line assembly function of the simulator. First, call the simulator. (The means for doing this will depend on your development environment - refer to [3] for instructions)

2. If they are not already open, open a session window, an assembly window, and a command window.
MENU: Windows, Assembly / Windows, Session / Window, Command

3. Change the device program counter such that it is pointing to internal memory. This can be done in a number of ways, and each user will have preferences as to which way they would like to use the simulator.
a. At command line type 'change pc $100'
or
b. MENU: Modify, Change registers, find pc and enter $100 at 'value' or
c. MENU: Window, Registers, find pc and edit value to $100

4. The Assembly window should now contain instructions starting at p:$100, with the address p:$100 highlighted. (The instructions should all be nops!). To edit the instructions at address p:$100, click on the nop and delete.Type in the new instruction type: 'move #$45,x0' press return to move to next location.

5. Using the method above, enter the small program

```
p:$100 move #$4f,x0
p:$101 move #>$4f,y0
p:$103 move #$040000,x0
p:$105 move #$020000,y0
p:$107 mpy x0,y0,a
```

NOTES: In some circumstances the in-line assembler will change the format of the instruction. Those instructions which move long words are two instruction words long. This feature of the simulator is especially useful for 'patching' large pieces of code, so that to change one instruction you do not have to assemble, link, and re-load the entire program.

6. Step through the small program. Either press the step button on the main window or enter the word 'step' at the command window. After each step, examine the session window to see the effect of the code.

NOTE: Pressing return in the command window repeats the last instruction

7. To verify the results use the evaluate command. There are a number of ways to do this. Try three different ways to evaluate the previous multiplication.

a. At command line type 'evaluate f $040000' the fractional equivalent will appear in the session window.
(To save typing in 'evaluate' type 'e <space bar>')

b. MENU: Display, evaluate, type $020000 as expression and press Fractional button for radix.

c. At command line type 'evaluate f a' to evaluate the result in accumulator a
Division on the DSP56300

8. To assemble this file open a dos shell/ unix command window and ensuring that you are in the example3 directory type:

```
asm56300 -b -l ex3_main.asm
```

This will create two new files: ex3_main.cln, which is the file to be passed to the linker and ex3_main.lst, which is a list file generated by the assembler. This assembly will generate a number of pipeline stall warnings. For detailed information on this, please refer to exercise one and the family manual [1].

NOTE: If this does not work correctly ensure that the default path of the machine was correctly set up during the installation procedure.

9. Now open the file 'ex3_main.asm' in a text editor. This file contains the entry point for the division example, and also contains a subroutine which implements the division routine.

10. Open file ex3.ctl in a text editor window. This is the file which the linker will reference to decide where to place sections of memory.

11. Call the linker to link these files together into the so called absolute object file which the simulator can load. Do this by typing:

```
dsplnk -mex3.map -bex3.cld -rex3.ctl ex3_main.cln
```

This means that ex3_main.cln will be linked using the instructions contained in ex3.ctl. The output will be a machine loadable file called ex3.cld, and a map file showing the location of sections in memory in ex3.map

12. Load the program ex3.cld into the simulator
(MENU: File, Load, Memory COFF)

13. Step through the code until you have reached the first rts (return from subroutine) instruction. Evaluate the results of the division using the evaluate command

14. For the second execution of the division, we do not want to step through every instruction. This time step until you reach the first execution of the div instruction then, at the command window, type 'step 24'. This instruction will execute 24 steps before returning. Step through until the next 'rts' and evaluate the results.

15. For the third execution of the division routine we are not interested in anything but the result. Find the rts instruction and double click on its address. This will insert a breakpoint. (The address of that instruction should be highlighted). Now press the go button on the top menu. The simulator will now execute until it reaches the breakpoint. Once it breaks, verify the results of the division using the evaluate command.

16. Congratulations..............you have completed exercise 3!

EXERCISE 4 - FIR FILTER IMPLEMENTATION

Introduction

This example details the implementation of a digital FIR (Finite Impulse Response) filter. In this exercise you should learn to:

- Use parallel moves to code efficiently
- Implement hardware DO loops, and use the REP instruction
- Set up and use modulo buffers
- Use the simulator to take input from an external source
- Use subroutines, and link a number of .asm files together

Technical Considerations

FIR Filters. It is not the aim of this session to introduce DSP theory, however it is important to understand the basics of the algorithm you are about to implement. Therefore, this section contains a short description of the mathematics of FIR filters.

The operation of an FIR filter is represented by the equation:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

y[n] = output value
x[n-k] = input delayed by k sampling periods
b_k = tap values of filter

This can be expanded to:

$$y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] + \dots\dots\dots + b_M x[n-M]$$

Graphically the general FIR filter can be shown as:

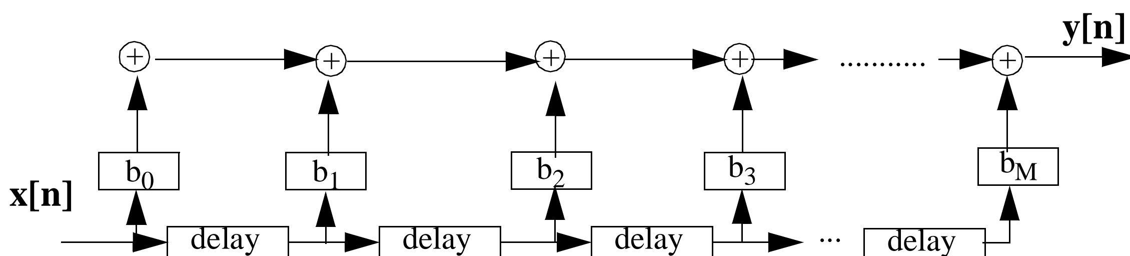


Figure 30: General Representation of an FIR Filter

Implementation Description

The following paragraphs describe a different feature of the FIR filter implementation. Each one is described to provide a basic understanding. For more details please refer to [1]. It is suggested that the files ex4_main.asm and ex4_fir.asm are available as you read this document as the code will illustrate the examples discussed.

Defining memory spaces and constants. In this example memory spaces and constants are defined in a number of ways. Basic memory space is defined using the directive ds (define storage). This directive reserves the required number of data words and labels them. The memory locations are not initialised to any value. You will notice that although the org statements are used to define where the data memory should be placed in terms of X and Y, the address at which they should be stored is not controlled here. This makes the section of code relocatable. The exact placing of the block in memory is controlled by the linker.

In certain circumstances there are limitations on where certain pieces of data memory have to be stored. For example, circular buffer must reside on modulo boundaries. This must be indicated to the assembler and linker and there are a number of ways to do this. If the circular buffer was to be defined with no initial values, then the directive dsm (define modulo storage) could be used. However, in this case our circular buffer must contain our tap values, which are defined using the dc (define constant) directive. To indicate that this set of constants must be placed in a memory location suitable for a modulo 'NUM_OF_TAPS' buffer, we use the buffer command.

Hardware DO loops and the REP instruction. A number of methods of looping are demonstrated in this example. The first example is the do forever loop, which does exactly what it suggests and continues forever. A do forever loop can be stopped by a ENDDO instruction, or a breakpoint in the code. The second example is probably the most common: do #n. In this case every time the loop is entered it is to be executed a constant number of times. This constant is therefore part of the instruction. The third example (in the fir_filter) subroutine is the do xx loop. In this case, the number of times the loop will be executed is not known at assembly time, and could indeed be a variable. In this case the loop count variable is passed in a register.

When the loop contains only one instruction then the rep instruction should be used. This has even less overhead than the hardware do loop. The rep instruction can be used with an immediate value, or have the variable in a register, as in this case.

NOTE: The dor instruction is used here instead of the do instruction. This simply means do relative and means that the loop address is stored as a relative value rather than an absolute value.

Modulo buffers. This example uses modulo (circular) buffers to store the taps. Circular buffers are created in the DSP using the modifier registers (mx). These modifier registers change the way in which the pointer registers (rx) see the data memory. Full details of the operation of the modifier registers can be found in [1].

The modifier registers are reset to $FFFF, which means that the pointer registers see data memory as one continuous block of X, and one continuous block of Y. Changing the value in each modifier register will change the way in which each address register views the memory.

Here we will illustrate the use of the modifier register with an example: Let us assume that our filter has 20 taps. We require the address pointer to automatically wrap round when it reaches the end of this list, such that it will point once again to the first value on the list. To do this we will change the value in m4 from its default value of $FFFF to $0013 (decimal 19), this causes the address pointer r4 to view the memory in the device as shown below, i.e. split into a

number of separate memory blocks, each 20 words long, and each circular in nature.

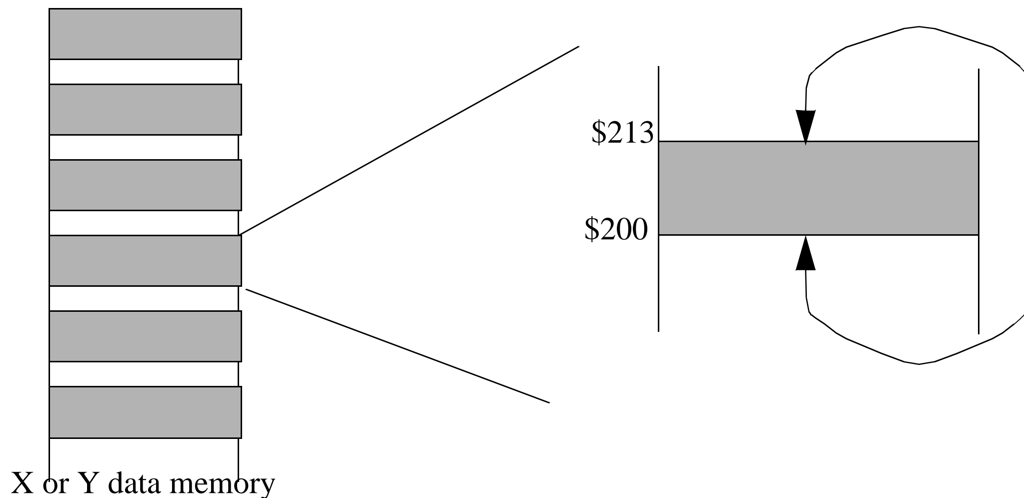


Figure 31: Modulo 20 Buffer

When r4 is pointing within one of these blocks and the pointer is updated to what would normally be out with this block, the pointer r4 automatically wraps round.
e.g.

EXAMPLE 1	EXAMPLE 2	EXAMPLE 3
before execution: r4 = $203,m4=$13	before execution: r4 = $213,m4=$13	before execution: r4 = $210, n4 = 6,m4=$13
instruction: move y:(r4)+,y0	instruction: move y:(r4)+,y0	instruction: move y:(r4)+n4,y0
after execution: r4 = $204	after execution: r4 = $200	after execution: r4 = $203

NOTE: The use of circular buffers means that the data memory must be placed according to certain rules. For details of these rules refer to [1].

Setting up input and output files. In order to test the piece of code you have developed you have to test it using some input values. For some pieces of code this can be achieved by simply manually inserting values into the relevant registers and memory locations before running the program and checking the result. This is the method used in some of the other exercises.

In some cases however, it is necessary to test a piece of code with many values, e.g using test vectors when conforming to a standard, and it would be impractical to insert all the test values manually. The DSP56300 debugging environment provides a method of connecting files to a certain memory location, pin, or port, in order that this can act as a peripheral and read values into the debugger from an external source.

In this case we will connect the external source to memory locations in high Y data memory. Using high Y memory allows us to use the movep instruction to move data directly from the high Y memory location into another memory location.

NOTE: This description of the use of input and output files is only valid when using the simulator. When using the ADS system the method is different, for details refer to the ADS users manual.

The input file should be of ASCII format. Chapter 3 of [3] gives details of the different file formats. In this example we will use the file **ip_data.io** on the disk which is in hexadecimal ASCII format.

A description of how to connect the file to the memory location is given in the section below.

Test Run

1. Open the file ex4_main.asm in a text editor. This file contains the entry point for the FIR filter program. Within this program there is a call to subroutine fir_filter.

2. Open the file ex4_fir.asm in a text editor. This file contains the subroutine which implements the digital filter.

3. Assemble the main file by opening a dos shell/unix command window and from within the correct directory typing:

    asm56300 -b -l ex4_main.asm

This will create two new files: ex4_main.cln, which is the file to be passed to the linker and ex4_main.lst, which is a list file generated by the assembler

NOTE: If this does not work correctly ensure that the default path of the machine was correctly set up during the installation procedure.

4. Assemble the filter subroutine file by typing:

    asm56300 -b -l ex4_fir.asm

This will generate one pipeline stall warning. For detailed information on this, please refer to exercise 1 one and the family manual [1].

5. Open up the file ex4.ctl in a text editor window. This is the file which the linker will reference to decide where to place sections of memory.

6. Call the linker to link these files together into the so called absolute object file which the simulator can load. Do this by typing:

    dsplnk -mex4.map -bex4.cld -rex4.ctl ex4_main.cln ex4_fir.cln

This means that ex4_main.cln and ex4_fir.cln will be linked together using the instructions contained in ex4.ctl. The output will be a machine loadable file called ex4.cld, and a map file showing the location of sections in memory in ex4.map.

7. Call the simulator (The means for doing this will depend on your development environment - refer to [3] for instructions) If the simulator is already running, RESET the device.

8. Load the program ex4.cld into the simulator (MENU: File, Load, Memory COFF)

9. If they are not already open, open a Session window, a Command window, and an Assembly window. The session window will show the state of the device following each step. The command window can be used to input command directly or will show the commands executed using the menus. The assembly window will show the code in program memory, and will indicate the next instruction to be executed.

10. Connect address y:$FFFFEE to the input file ip_data.io (MENU : File, Input, Open) The input number is 1, the input is from a file, the file is connected to memory (memory space Y, address $FFFFEE), the radix is hexadecimal, and the filename is ip_data.io.

11. Connect address y:$FFFFEF to the output file op_data.io (MENU: File, Output, Open) The output number is 1, the input is from memory to file, the radix is hexadecimal, and the file is op_data.io (if the tools report that the file exists, instruct it to overwrite.)

12. Set up a conditional breakpoint in the file. We want this test to be able to deal with any length of input file. The tools can automatically detect the end of file, and we shall use this here. The first time the device tries to read a value which isn't there, it will break. (MENU: Execute, Breakpoints, Set) The breakpoint number is 1, the type is expression, the action is halt, the expression is eof

13. Instruct the simulator to go. It will break once it has completed filtering the entire input file. Do not be surprised if it takes a few minutes to break! This is a cycle exact simulator and therefore requires a lot of processing power. When it has finished processing, close the output file (MENU: File, Output, Close).

14 To ensure that the function has operated correctly, compare op_data.io and op_ref.io....they should be identical.

15. Congratulations......you have completed Exercise 4

NOTE: When using more complicated test environments it becomes too time consuming to type all the commands in manually every time. It is much more efficient to use command files. A command file (run ex4.cmd) has been created from this application. This can be run using MENU: File, Macro. HOWEVER, this will only work if: a) the simulator is called while in the exercise3 directory or b) you edit the command file to include the directory of your files.

EXERCISE 5 - CALCULATE SQUARE ROOT

Introduction

In this Exercise you should learn how to:

- Calculate a square root on a fixed point DSP
- Test an assembler Function
- Interpret the fractional number format

There are a number of approaches possible for a root calculation on a fixed point DSP. All these techniques have a certain resolution and a corresponding complexity. The algorithm that is shown below has a good resolution with reasonable effort.

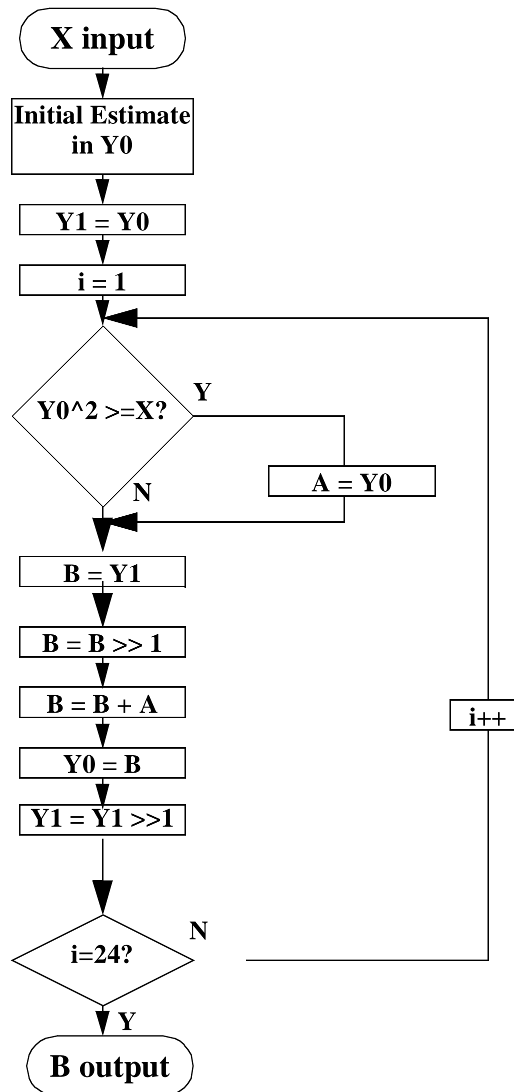


Figure 32: Flow Graph of the Root Algorithm

Technical Considerations

Root Algorithm. The root algorithm can be seen as a control loop with 24 iterations to be run until the maximum resolution is achieved. The algorithm is depicted in Figure 32. The initial

sign checking (see function file ex5_root.asm) and the rounding are not included in the figure. The root function works with simple approximation and takes advantage of the multiplier unit. It determines the square value of the actual estimation and compares this one to the input figure that is stored in the x register.If the square value is smaller, the estimation is increased by the actual bit position that is subject to the estimation. The algorithm starts with the MSB and goes down to the LSB. The square root function is shown in Figure 33.

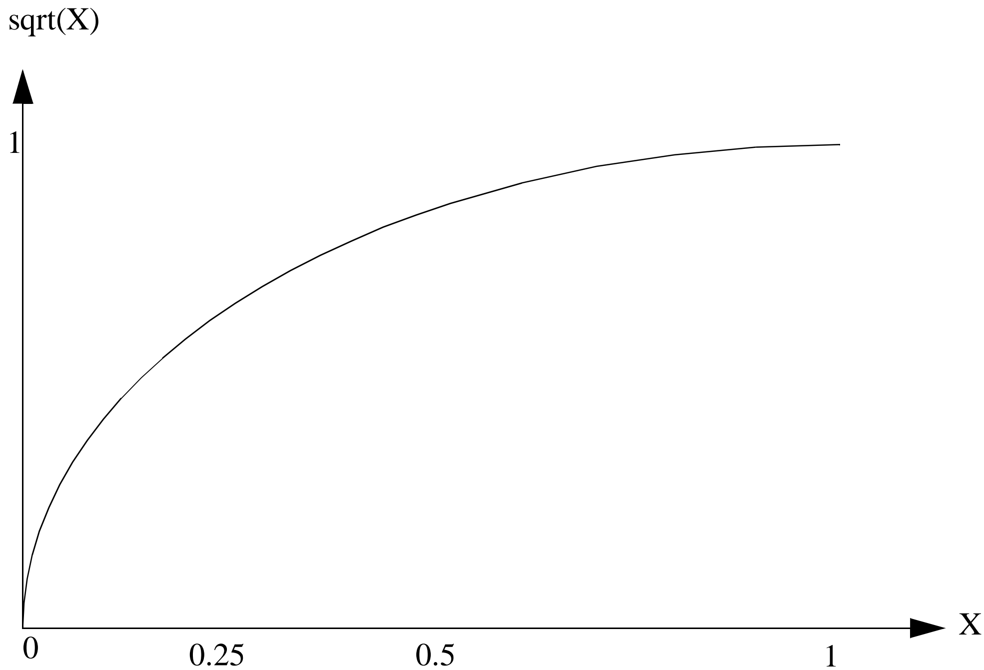


Figure 33: Square Root Function (X=0..1)

Memory Allocation of Long Words. If a part of the memory space is used for long words (48 bit format), the memory allocation should be controlled such that there is no (or a minimum) gap between the l section and the x: and y: sections before or after it.

Since the x: and y: memory is concatenated for l: data, the allocation can be most efficiently controlled by moving all the long sections at the very beginning of the data memory, this is done in ex5.ctl, it is shown in Figure 34 as well.

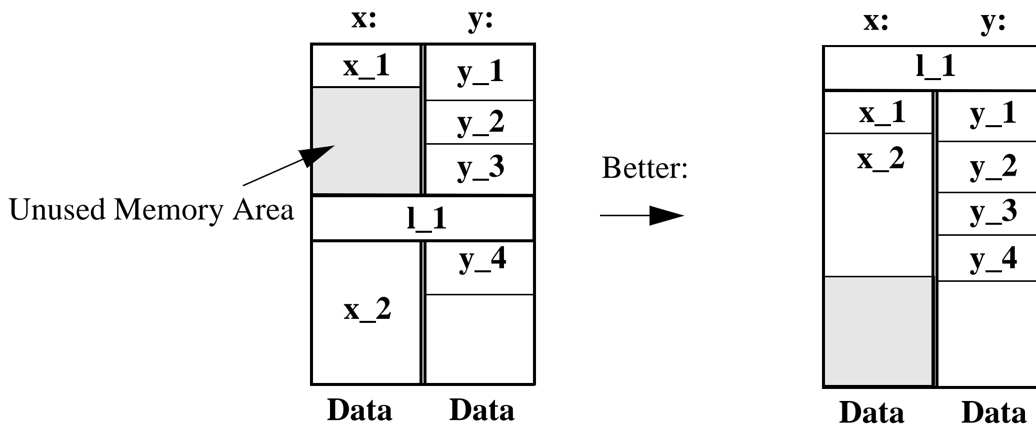


Figure 34: Long Memory Allocation

Implementation Description

The program is written as simple as possible to emphasise the main functionality. Thus, the main file consists only of the test data (5 numbers) to be checked and a main program of a few lines. Within this main loop, the function is called five times with the test data listed in the table below.

Due to the two rounding modes that can be adjusted in the mode registers, we get in this case two sets of possible results: either the left column which was processed in the convergent rounding mode, or the right column obtained by processing the same set of data in 2's complement rounding mode. Please refer to [1], p.3-8, p.6-15 for details on rounding.

In addition to that, test data was chosen such that the most extreme results are obtained from the function and these input figures should be tested here for both rounding modes. The error cases are covered as well, i.e.

a) the saturation mode, it was set to the arithmetic option so that an area of overflow near the maximum can be avoided (otherwise the rounding produces $800000 in certain cases).

b) negative inputs are ignored, the output is set to zero.

Input (48 bit)	Result I (Convergent)	Result II (2's Complement)
1	0	1
$7ffffc 000000	$7fffff	$7fffff
$7ffffc 000000	$7ffffe	$7ffffe
$200000 000000	$400000	$400001
$ffffff ffffff	0	0

Test Run

1. Open the file ex5_main.asm and take a look at the code and the comments, respectively. Do the same with the function file ex5_root.asm.

2. Run the command file to assemble and link the code: Call ’do-ex5.bat’. This is easier than typing the complete command line all the time. To see what is happening, you may take a look at it as well. The two pipeline stall warnings are completely uncritical at this point, please refer to Ex1 for explanations.

3. Call the simulator now, look for the directory /tutorial/ex5 and load ’ex5.cld’. (MENU:File, Load, Memory COFF)

4. Open an assembly window (MENU: Windows, Assembly) and one to observe the results (MENU: Windows, Memory, then select ‘L’ memory space). If you want to check the results on-line, you can either take the registers refer to documentation during the root calculation or add watches, which may be more convenient to look at (MENU: Windows, Watch, add symbol).

5. If you want to observe the function call you may step through until the function is called the first time. If not just set a breakpoint at the label 'check' (MENU:"Execute, Breakpoints, Set") and run the function for the 5 inputs provided.

6. You can either calculate the examples before running them or check them while stepping through.

7. If you get the same results, well done. For more information on the data format and the algorithms, please refer to the documentation.

EXERCISE 6- MATRIX MULTIPLICATION

Introduction

This exercise follows the implementation of a matrix multiplication and highlights a number of points:

- Use parallel moves to code efficiently.
- Implementation of nested hardware DO loops and use the REP instruction.
- Set up and use modulo buffers.

Technical Considerations

Matrix Multiplication. Matrices are commonly used to store multidimensional data in arrays in signal processing, control systems (to store coefficients of differential equations) and image processing applications (graphical data is stored in two dimensional arrays). These data matrices are manipulated by multiplying them with other matrices of coefficients. An example of a matrix multiplication is shown below:

$$\underset{A}{\begin{bmatrix} 0.10 & 0.11 & 0.12 & 0.13 \\ 0.20 & 0.21 & 0.22 & 0.23 \end{bmatrix}} \times \underset{B}{\begin{bmatrix} 0.32 & 0.52 & 0.72 \\ 0.34 & 0.54 & 0.74 \\ 0.36 & 0.56 & 0.74 \\ 0.38 & 0.58 & 0.78 \end{bmatrix}} = \underset{C}{\begin{bmatrix} 0.162 & 0.254 & 0.346 \\ 0.302 & 0.474 & 0.646 \end{bmatrix}}$$

2 x 4   4 x 3   2 x 3

The fractional data shown above is used throughout this exercise (the use of fraction avoids the necessity to add conversion routines for integers or real numbers). The dimensions of the A, B and C matrices are defined as N x M, M x P and N x P respectively where N=2, M=4 and P=3

To implement a two dimensional matrix multiplication, three nested loops are required with a total of twenty-four multiply accumulates for this example. The code used in this example is generic and contains definitions for the N, M and P dimensions. If any of these dimensions were one, then this generic code can be simplified.

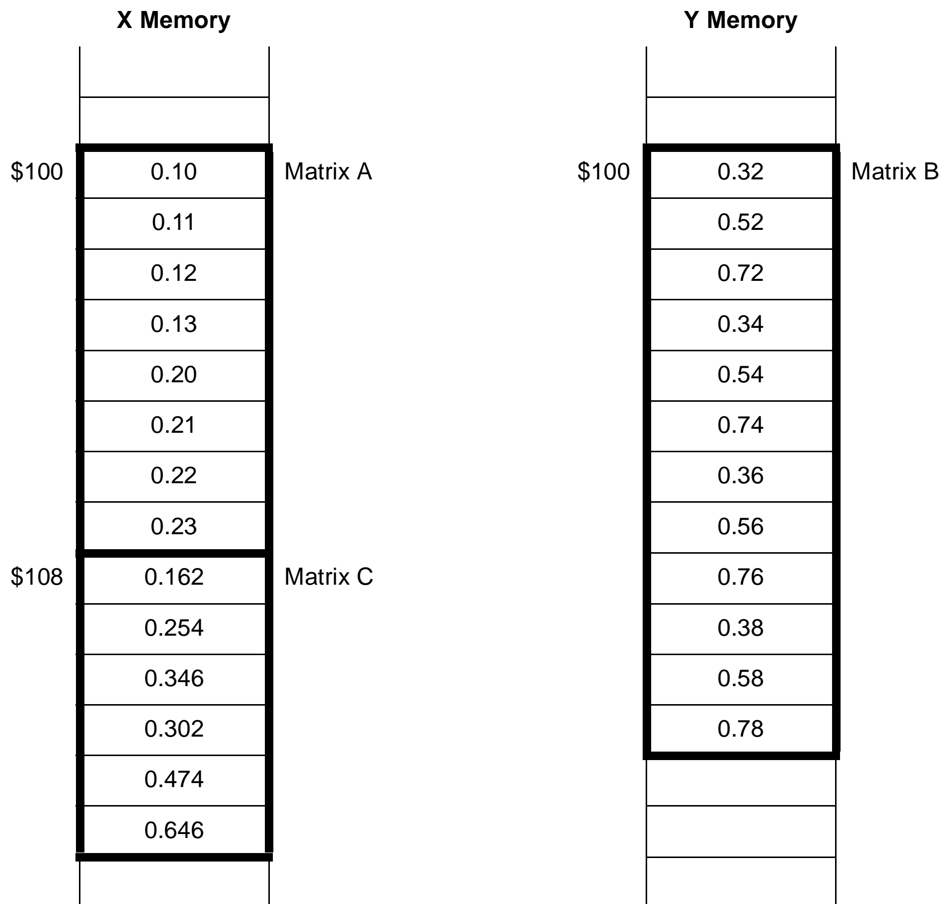
Implementation Description

Each of the following paragraphs describe a different feature of the matrix multiplication implementation. Each one is described to allow for basic understanding. For more details please refer to [1].

Defining memory spaces and constants. In this example memory spaces and constants are defined using the dc (define constant) directive. The dc directive allocates one word of space in memory for each data entry and fills these words with the A and B matrix data. The space for the result C matrix is cleared.

The data in each matrix is stored row by row. This convention is used for all three matrices to make it easier to read and to make the routine generic enough to take matrices resulting from other calculations without transposition.

The data storage arrangement and locations are shown below:



Hardware DO loop and the REP instruction. Two methods of looping are demonstrated in this example - the do #n loop and the rep instruction. These loops are nested.

The dor (do relative) instruction is used here instead of the do instruction to ensure that the loop addresses are stored as a relative rather than absolute values. The two do #n loop are executed a constant number of times and this constant is made part of the instruction by the assembler and linker. There are sixteen locations in the stack so up to sixteen do loops could be nested before stack manipulation is needed, provided that no other use is made of the stack.

When a loop contains only one instruction, the rep instruction should be used because this has less overhead than the hardware do loop. The rep instruction can be used with an immediate value, as in this case, or have the variable in a register.

Both the do and rep instructions use the loop counter (LC) register to control the loop operation. The loop counter is a down counter and the loop exits when the LC reaches zero. There is no iteration of the loop when the LC is zero so it cannot easily be used as an offset into the memory arrays storing the matrices.

Also in matrix multiplication, two indexing pointers are needed because a row from the A matrix is multiply/accumulated with a column from the B matrix. This is not conveniently arranged using the loop counters since only the LC from the inner loop is readily available (the outer LC is on the stack).

Modulo Buffers. As previously mentioned, the data in each matrix is stored row by row. When accessing consecutive data entries in the A and C matrices, the address pointer only needs to be post-incremented by one after each access. During the multiplication process, data in the B matrix must be accessed column by column, so the address pointer must be post-incremented by three after each access.

To implement these incremental addresses, modulo buffers are used for the A and B matrices. After each operation of the inner loop (the one with the rep instruction), the pointer into the A matrix needs to be pointing to the start of the same row and pointer to the B matrix needs to point to the next column. If modulo buffers were not used, then additional registers would be needed to hold pointers to the beginning of each row/column in A/B matrices and these would need to be loaded into the incrementing data pointers at the start of the appropriate loops.

The modulo (circular) buffers are created in the DSP using the modifier registers (mx). These modifier registers change the way in which the pointer registers (r0~7) use the data memory. Full details of the operation of the modifier registers can be found in exercises 2 and 4.

The A matrix pointer (register r0) is implemented as a modulo four counter (register m0=3) so that it automatically wraps round at the end of each operation of the inner loop. At the end of loop2, register r0 is incremented by four so that the next row of the A matrix is accessed.

The B matrix pointer (register r4) is implemented as a modulo twelve counter (register m4=11) so that it automatically wraps round at the end of each operation of the inner loop. At the end of each inner loop operation, register r4 is incremented by three so that the next column of the B matrix is accessed in loop2.

The C matrix is implemented as a simple buffer without modulo addressing using register r1 as the address pointer (register m1=$ffff by default after reset).

NOTE: The use of circular buffers means that the data memory must be placed according to certain rules. For details of these rules refer to [1]. When defining the data and storage locations for the A and B matrices, the buffer directive is used to ensure correct alignment of the data in memory.

Test Run

1. Open the file ex6_main.asm in a text editor. This file contains the entry point for the matrix multiplication program.

2. Assemble the main file by opening a dos shell/unix command window and from within the correct directory typing:

asm56300 -l -b ex6_main.asm

This will create two new files: ex6_main.cln, which is the file to be passed to the linker and ex5_main.lst, which is a list file generated by the assembler.

NOTE: If this does not work correctly, ensure that the default path of the machine was correctly set up during the installation procedure. During assembly, two warnings will be generated due to pipeline stalls.For detailed information on this, please refer to exercise 1 and [1].

3. View the file ex6.ctl in a text editor window. This is the file which the linker will reference to decide where to place sections of memory. In this example, data is located at address x:$100 and y:$100 and the program code is stored at p:$100.

4. Call the linker to link these files together into something which the simulator can load. Do this by typing:

dsplnk -mex6.map -rex6.ctl -bex6.cld ex6_main.cln

This means that ex6_main.cln will be linked and located using the instructions in the ex6.ctl linker control file. The output will be a machine loadable file called ex6.cld, and a map file (ex6.map) showing the location of sections in memory.

5. Change the working directory to the one containing the files for exercise 6 and start the simulator. The means for doing this will depend on your development environment - refer to [3] for instructions. If the simulator is already running, RESET the device.

6. If they are not already open, open a Session window, a Command window and an Assembly window. The session window will show the state of the device following each step. The command window can be used to input command directly or will show the commands executed using the menus. The assembly window will show the code in program memory, and will indicate the next instruction to be executed.

7. Open two memory windows, one for X memory and one for Y memory, each starting at address $100. The input and output matrices appear in these windows.

8. The matrix multiply examples use fractional data so change the radix of the X and Y memory windows and the a0 and a1 registers using the following:

   MENU: Modify, Radix, Set Display - Select Fractional button, X memory from $100 to $140.
   MENU: Modify, Radix, Set Display - Select Fractional button, Y memory from $100 to $120.
   MENU: Modify, Radix, Set Display - Select Fractional button, a0 and a1 registers.

9. Load the program ex6.cld into the simulator (MENU: File, Load, Memory COFF and select the ex6.cld file). The code will appear in the Assembly window, the input matrices in X and Y memory starting at address $100 and the result area in X memory is cleared.

10. Set a breakpoint at address p:$114, the first instruction after the calculation loops. You can do this either by double clicking on the address in the Assembly window or using the menu (MENU: Execute, Breakpoints, Set and select execute memory $114). Once the breakpoint is set, the address in the Assembly window turns blue.

11. First time around we will check that the code operates as expected and see how many cycles are needed to run. Check that the PC is set to $100 (the address $100 will be red in the Assembly window) and then press the GO button. After a few seconds, execution is complete and the results can be seen in the X memory window at addresses x:$108 to x:$10d. Results are displayed in fractional form. Verify that these are correct by comparing them to the ones given in the source code file and in the lab instructions. Check the cycle counter in the Session window. The execution time is 136 clock cycles.

12. Now repeat step 9 and reload the code and data. Notice that the breakpoint at address p:$114 remains set - this is the way the simulator operates. The PC and memory spaces are reset to allow operation again.

13. Step the program many times until address p:$113 is reached. At this point, the inner loop has been executed 12 times and the first row of the result matrix has been calculated. While stepping through the loops, you will notice that the data stored in the a0 and a1 registers and the memory is displayed in fractional form while the same data read into the x0 and y0 registers is displayed in hex form. Once address p:$113 is reached, press GO to complete the calculation without stopping again.

14. Congratulations......you have completed Exercise 6.

EXERCISE 7 - SELECTED INSTRUCTION EXAMPLES

Introduction

In this exercise you can see how the following instructions are used:

- BRKcc
- DEBUGcc
- IFcc
- CMP
- ENDDO.

Technical Considerations

The instructions covered in this exercise are the most frequently questioned by the 56300 users. Therefore, examples of code are shown here to clarify the use of them. The examples do not have a certain sense or function, they are just standalone demonstration code samples. The instructions themselves are all described in [1].

There are a few special topics you should be aware of when looking at these instructions:

Sequence Restrictions. The BRKcc instruction is one of the instructions that are affected by the so called sequence restrictions. This means that certain sequences of instructions are not to be used, because they would cause undefined states in the DSP. The BRKcc instruction for example needs one instruction delay for the condition to be tested (see Code). In addition to this, it should never be one of the last three instructions of a do loop. For the detailed list of sequence restrictions, please take a look at page B-21 in [1].

DEBUGcc. This instruction acts like a breakpoint in a running program. It stops the core and waits for commands on the emulator interface.

IFcc. The IFcc Instruction has a special format that is easier to understand in an example. Take a look at the code of the second example and you will immediately see, that the IFcc has not the format of a full, single instruction, it is rather treated as an option to an instruction like the parallel moves.

CMP. The cmp instruction is just shown to make the flag conditions transparent.

ENDDO. Finally the ENDDO instruction is applied to show how easy and effective exceptions may be handled in a hardware loop.

Implementation Description

The implementation of the examples should be understood with the help of the comments in the code.

Test Run

1. Open the file ex7_main.asm and take a look at the code and the comments, respectively.

2. Run the command file to assemble and link the code: Call 'do_ex7.bat'.

3. Call the simulator now, look for the directory /tutorial/ex7 and load 'ex7.cld'. (MENU:File, Load, Memory COFF)

4. Open an assembly window (MENU:Windows, Assembly) and 2 register windows to observe the actions.

5. Now, if you want, you can set a breakpoint to the first line of the example you are interested in and then step through the instructions.

6. If you are just reading this exercise to get general information, you can step through all the examples and observe the events on the core.

7. Congratulations......you have completed Exercise 7.

EXERCISE 8- POWER ANALYSIS

Introduction

In this Exercise you should learn how to:

- Generate simple test signals in memory
- Allocate long words in memory
- Use the max instruction
- Calculate a sliding window of a complex signal
- Do a threshold classification

Technical Considerations

Assembler Directives. Using assembler directives, options and macros (the complete set is listed in [3]) can save a lot of work during signal processing code development.

Therefore, you should be aware of what the tools are capable to do and how to make the maximum use of them. The assembler provides the following large number of useful features:

a) within Expressions, you can use:
- trigonometrical functions
- format conversions
- counting
- checksums
- list/nolist
- locate code and data

b) with directives, you can build
- conditional assembly
- control structures (loops, for, if, ..)
- sections, regions, buffers
- logical links between them
- macros

c) using the options, you can
- count cycles
- control comments
- control and generate messages
- control the assembler operation (addressing, formats, ..)
- control the output file format

An example of a possible application is shown in the beginning of the file ex8_main.asm: A complex signal of 300 samples is generated within a few lines. The directive DUPF duplicates an assembly line with optional parameters as often as you specify it. The generated signal can be seen in Figure 35.

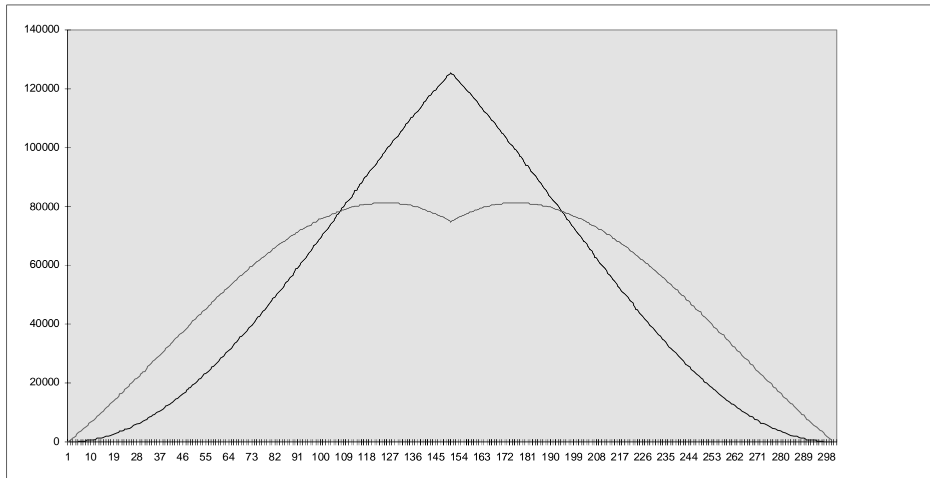


Figure 35: Generated Signal

Signal Power Calculation. The power of a complex sampled signal in general can be calculated as follows:

$$p_s = \sum_{\mu} I(\mu)^2 + Q(\mu)^2$$

A power calculation practically always is generated from a certain window of the signal that is most of the cases sampled continuously. Therefore, with a window size of M and a signal length of N, the following equation for the unnormalised power of a windowed signal can be given:

$$p_s(i) = \left. \sum_{\mu = 0}^{\mu = M-1} I(\mu + i)^2 + Q(\mu + i)^2 \right|_{i = 0 \ldots N-M}$$

In the code example, N=300 and M=3 is chosen to demonstrate the power calculation.

The calculated power is then classified logarithmically. The steps are here 1..10^14, where power class 1 means the level is between 1 (10^0) and 10^1.

Implementation Description

The assembler file ex8_main.asm contains just the initialisation of data (signal and look up table) and a simple main program with address initialisation and function call. The parameters for the function are two pointers to the first samples of the real (I) and imaginary (Q) input signal parts, and the number of samples in the input signal to be taken for the power calculation.

Since it is normally not a variable, the length of the window is set to three implicitly by the implementation of the function, i.e. if the size should be changed, the function has to be changed. The main loop in the function takes 6 cycles for execution for each window including

the maximum function and the related address storage.

The power classification is done with a loop of constant execution time, i.e. the loop is performed for all the power classes, no matter if the maximum found was power class 1 or 13. The number for the power class is derived from the address of the look up table in long memory directly. For the description of the memory use for long and short words, refer to “EXERCISE 5 - CALCULATE SQUARE ROOT” on page 41, please.

Test Run

1. Open the file ex8_main.asm and take a look at the code and the comments, respectively. Do the same with the function file ex7_func.asm.

2. Run the command file to assemble and link the code: Call 'do_ex8.bat'.

3. Call the simulator now, look for the directory /tutorial/ex8 and load 'ex8.cld'. (MENU:File, Load, Memory COFF)

4. Open an assembly window (MENU:Windows, Assembly). You may open one or more memory windows to take a look at the generated input data (MENU:Wondows,Memory, Select either x (real data, $100..),y (imag data, $100..), or l (look up table, $0..) space).

5. Now, if you want, you can insert a breakpoint in the first do-loop to check, how the maxima are detected, updated and how the corresponding address is stored.

6. At the end of this loop, the maximum level stored is subject to a power level classification. Check the output now: x0 = $d -> power class 13 (10E12..10E13) b = $1db0 7f465 -> signal power of maximum sample window r1 = $195 -> start address of the maximum window

7. If the numbers are correct, well done.

Further Support

During the exercises you should have seen the basic principles of signal processing and their applications to different problems using some of the key features of the Motorola DSP56300 core.

Hopefully these exercises would have helped you progress with Motorola's DSP architecture. If you are continuing to design with Motorola DSPs and would like more detailed infomation on any of the topic's listed below, please contact your local Motorola distributor.

1. **Documentation.**
   All the documentation referenced on page 5 is available via your local Motorola distributor.

2. **Software Development and Evaluation.**

   Software:
   You can run your DSP code clock exact on the GUI Simulator. This means that the simulator will behave exactly like the hardware that is currently available. The simulator package contains peripheral simulations for all currently available devices using the DSP56300 core. Therefore you can simulate almost every external communication with file I/O connected to the ports.

   Hardware:
   DSP development hardware maybe purchased i.e. an ADS (Application Development System) or an EVM (Evaluation module - only available for some DSP56300 derivatives) sytems.

3. **Training.**
   Training courses can be provided on the DSP56300 family which includes a more detailed introduction to:
   - the architecture,
   - the features,
   - the hardware,
   - the peripherals,
   - the tools.

4. **Software Support.**
   Motorola, and a number of third party companies can provide applications software for the DSP56000/DSP56300 families, please request a list from your distributor.

5. **Other Topics.** Please contact the Motorola Helpline:
   dsphelp@dsp.sps.mot.com
   Or view the Motorola DSP pages on the World Wide Web :
   http://www2.motorola-dsp.com/dsp/