

# ADSP-21262 SHARC Basic Tutorial

## Contents

	page
1. <u>Introduction</u>	2
2. <u>Basic Instructions</u>	
1. <u>Syntax Form</u>	
2. <u>ALU Instructions</u>	
3. <u>Multiplier</u>	
4. <u>Shifter</u>	3
5. <u>Memory Access</u>	
6. <u>If Conditionals</u>	
7. <u>Program Flow</u>	4
1. <u>Loops</u>	
2. <u>Subroutines and Jump</u>	
8. <u>Reserving Memory</u>	5
9. <u>Circular Buffering</u>	
10. <u>Parallel Processing</u>	6
1. <u>MAC (Multiply and Accumulate)</u>	
3. <u>Other Useful Commands and Conventions</u>	
1. <u>Assembler and Preprocessor</u>	
2. <u>Numeric Formats</u>	7
3. <u>Comments</u>	
4. <u>External Files</u>	
5. <u>Loading Data From Text Files</u>	8
4. <u>Registers</u>	
5. <u>Advanced Stuff</u>	
1. <u>Second Register Sets</u>	9
2. <u>SIMD (Single Instruction Multiple Data)</u>	

# 1. Introduction

This document is meant to be a basic tutorial for syntax, registers and some other useful things when programming SHARC. If you have any problems with your work, check first this document. There will also be some references to manuals so you can easily find corresponding information there. All needed manuals should be available at course website. This document is meant as a complementary for SHARC-Instructions by Juha Merimaa.

Code examples will look like this.

```
this is a code example
```

# 2. Basic Instructions

## 2.1 Syntax Form

SHARC uses arithmetic assembly. This means that every arithmetic command has a certain look to it. For example, add command looks like this.

```
R2 = R0 + R1;
```

So here sum of R0 and R1 will be put to register R2. Memory access and other functions also follow same convention. In the following lines you can see first a multiplication and then a memory write. Notice that every command ends in semicolon.

```
F2 = F0 + F1;  
dm(I0,M1) = F2;
```

In following sub-chapters there will be more information about specific commands.

## 2.2 ALU (Arithmetic Logic Unit) Instructions

Arithmetic logic unit handles all basic arithmetic and logic operations. These include summation, logical operations, comparisons and conversions between fixed-point and floating point form. Fixed-point and floating point commands use different registers access even though the register bank is same. Next you can see pairs of instructions where the first one is fixed-point one and second is same instruction in floating point.

```
R2 = R1 - R0;  
F2 = F1 - F0;  
  
R9 = MIN(R2,R14);  
F9 = MIN(F2,F14);
```

There is different instructions sets for fixed-point ALU- and floating point ALU-operations. You can find more information in the ADSP-21160 Instruction Set Reference starting from page 7-4.

## 2.3 Multiplier

Multiply operation is done on a separate unit and thus has a own set of instructions. Basic operation works like with ALU and multiply looks like this.

```
R2 = R0 * R1;  
F2 = F0 * F1;
```

With fixed-point numbers multiplier can work differently based on chosen instructions. You can find multiplier instructions in the ADSP-21160 Instruction Set Reference starting from page 7-54.

## 2.4 Shifter

Separate shifter unit is capable of basic bit-shifting operations with fixed-point numbers. You can find shifter instructions in the ADSP-21160 Instruction Set Reference starting from page 7-65.

## 2.5 Memory Access

SHARC contains two separate memories, data memory and program memory. You can access both of them and they use different commands. There is also different versions of addressing and modification of data address registers. Here are some variations which are used. First reading from memory.

```
F0 = dm(0x000015F0);
```

This form uses absolute addressing and memory address is given directly to the instruction. In most cases you have defined a name for absolute value and you use the name instead of value. Instruction itself gets value stored in the memory corresponding to the address and stores the value to register F0.

```
F0 = dm(I0,M3);
```

This command uses indirect addressing with register I0 and post-modifies I0 with register M3 after read. So value stored in memory address contained in register I0 is stored to F0 and after that M3 will be added to I0 for a new memory address which is stored to I0.

```
F0 = dm(M3,I0);
```

This is same as above but now with pre-modify. So now we read from address I0+M3. And this modified address will not be stored to I0.

```
F0 = dm(I5,4);
```

This is the same indirect addressing with post-modify but now immediate value of 4 is used for modifying. So after read the register I5 contains value I5+4.

```
F0 = dm(-3,I5);
```

Again, this pre-modify version and value is not stored after use.

```
F0 = pm(I8,M9);
```

This command is same indirect post-modify as before but now it reads from program memory instead of data memory. Only difference to data memory read is that the command is pm and used data address registers are at different range. More information about registers can be found at [chapter 4](#). Full memory addressing summary can be found in the ADSP-21160 Instruction Set Reference starting from page 2-14.

## 2.6 If Conditionals

You can control program execution with if-commands. This can be done to multitude of commands and basic form looks like this.

```
comp(R0,R1);  
if EQ call routine;
```

So first we do a comparison with ALU and this sets ALU output registers to certain form corresponding to result. Then immediately after that we check if those registers were equal. If it was, then we call subroutine. The condition EQ could be replaced with any other correct condition. These can be found in the ADSP-21160 Instruction Set Reference page 2-18 and 2-19. Also the command after condition could be almost anything. All specifications can be found in the aforementioned documents.

## 2.7 Program Flow

### 2.7.1 Loops

Loops are important tools for signal processing and thus there is a specialized way of using them in SHARC. All loops in SHARC are “do-until” loops which can be analogous to while loops or for loops based on situation. For loop looks like this.

```
LCNTR = 10, DO my_loop UNTIL LCE;  
    R0 = dm(I0,1);  
    R3 = R0*R0;  
my_loop: pm(I8,1) = R3;
```

In this code the first row initializes loop. LCNTR is a special register dedicated for this purpose and now value of ten is put in it. Then after that the loop point is specified which is my\_loop in here. After that comes UNTIL LCE which specifies that this code loops until LCNTR is equal to zero. Thus this loop does commands inside the loop ten times. Notice that the last command executed in the loop is the one at the same row where the loop point my\_loop: is. So in this loop there is three commands.

Another version of loop is while loop and it looks like this.

```
DO my_loop2 UNTIL EQ;  
    R0 = R0-1;  
my_loop2: comp(R0,R1);
```

So this loop does not have specific counter. And the ending condition is equality. So this functions like the if-condition in 2.6. All this loop does is decrementing R0 by one until it is equal to R1.

### 2.7.2 Sub-Routines and Jump

You can also use sub-routines with SHARC. Basic sub-routine looks like this.

```
my_routine:  
    R0 = R1+R2;  
    dm(I0,M1) = R0;  
    rts;  
my_routine.end:
```

So this is a routine which takes parameters R1 and R2 and calculated sum of them which is stored to memory defined by parameters I0. After that routine is ended by returning to calling position with instruction rts. To call this sub-routine you will use following code.

```
R1 = 5;  
R2 = 10;
```

```
I0 = address;  
M1 = 1;  
call my_routine;
```

So here the parameters are set to specific values and then the routine is called. Notice that I0 uses value "address" which is a memory variable. More of this is in the next chapter.

## 2.8 Reserving Memory

Reserving memory is done with assembler directives at compiling time. Here is an example how a variable is reserved from data memory.

```
.section /dm seg_dmda;  
.var my_value[1];  
.var my_array[10] = 1,2,3,4,5,6,7,8,9,10;
```

First row states that all following code until next section contains memory reservations from data memory. Second row reserves a 32-bit memory spot for variable named my\_value. And third row reserves memory for an array of ten values and then assigns values to them. So there is no specification of data type stored in the variable. It is up to you how you handle the information. Name of the variable works as a symbolic name for a memory value like a pointer in C. So you can use the name for memory loads and writes, but be careful not to use memory out of bounds. There is no handling of errors so you must be careful. Example of loading memory with this symbolic value follows. Here is shown versions for absolute and indirect addressing. Both do the same thing,

```
F0 = dm(my_value);  
  
I0 = my_value;  
F0 = dm(I0,0);
```

If you are reserving memory from program memory then you should use following section.

```
.section /pm seg_pmda
```

And program code also has its own section.

```
.section /pm seg_pmco
```

In most cases you should put your corresponding code to the positions already in the course template file as to avoid confusion and errors. You can find more information about using .var command from Visual DSP 5.0 Assembler and Preprocessor manual starting from page 1-133.

## 2.9 Circular Buffering

Circular buffering is one basis of DSP-programming. This enables easy calculation of inner products and thus filtering. Normally circular buffering is disabled in the template so if you want to use it you will have to enable it with following command.

```
BIT SET Mode1 CBUFEN;
```

So this changes specific bit in register Mode1. After enabling the circular buffering works with data address registers. Here is an example how to set a circular buffer for the array specified in [2.8](#).

```
B0 = my_array;  
L0 = 10;
```

So B0 register contains the starting position of circular buffer and L0 contains the length of buffer. When setting B0 also register I0 is set to same value. Memory is accessed normally with I0 register but when modify with M-register or immediate value would move the value of I0 out of buffer area, it will wrap around to the other end of buffer. This works in both directions. Here is an example assuming previous has been executed.

```
R0 = dm(I0, -1);  
R1 = dm(I0, 2);
```

So now register R0 contains value 1, R1 contains value 10 and I0 points to second value in buffer.

Length contained in L-register can also be different size than reserved size of buffer. This might be useful in some cases. However it is again a bad idea to make L-register larger than the reserved memory. Also moving I-register out of bounds with assignment might be a bad idea if you do not know exactly what you are doing.

## 2.10 Parallel Processing

SHARC has highly parallel architecture. This enables using different instructions at the same time. Here is an example of parallel instructions.

```
F8 = F0 * F4, dm(I0, M1) = F8;
```

This instruction calculates a multiply and store to memory at the same time. Notice that the result of multiply is not the same value which is stored to memory even though the register is the same. Result of calculation is ready only after the execution of instruction.

There are some restrictions when using parallel computing. You can only use three parallel instructions and none of them can use the same unit. ALU, multiplier, program memory and data memory are separate units so you can use them in parallel. However there are few exceptions to this rule. Sum and difference of the same registers can be calculated in parallel. Also MAC-instruction is counted as only one instruction. Another restriction is that you cannot use any immediate values when using parallel instructions.

You can find more information about parallel computing with SHARC in the ADSP-21160 Instruction Set Reference starting from page 7-93.

### 2.10.1 MAC (Multiply and Accumulate)

Multiply and accumulate is a special command in SHARC. It is counted only as one instruction so you can use for example two memory accesses in parallel with it. Also there is a restriction with registers used when using parallel MAC. Source registers for multiply must be R0-R3 and R4-R7 and source for summation must be R8-R11 and R12-R15. Of course corresponding F-registers work as well. Here is shown a maximized parallel MAC with two memory reads.

```
F9 = F2 * F6, F15 = F9 + F15, F2 = dm(I0, M4), F6 = pm(I12, M14);
```

## 3. Other Useful Commands and Conventions

### 3.1 Assembler and Preprocessor

There are some useful commands to use outside of basic syntax. `.VAR` specified in chapter 2.8 is one of them. Another very useful command is `LENGTH` which gives the length of an array in memory. So following code would store value of 10 in R0 if we used array specified earlier.

```
R0 = LENGTH(my_array);
```

Preprocessor also at least one really useful command.

```
#define MY_CONSTANT 2008
```

This command works just like in C and defines symbolic name for a value. So previous line you could use MY\_CONSTANT in code instead of value 2008. Preprocessor then replaces every instance of that symbol in code with corresponding value before compiling.

You can also use macros like in C. These are not really needed in course projects. More information about them and anything relating to assembler and preprocessor can be found in the Visual DSP 5.0 Assembler and Preprocessor manual.

### 3.2 Numeric Formats

SHARC uses both fixed-point and floating point calculation. When programming this leads to some differences. Basically when using fixed-point you will use registers R0-R15 and with floating point use F0-F15. However in memory these use same spots so basically R0 and F0 are same but handled differently.

When using immediate values, assigning values to memory and loading from file, you should use specific format to specify used numeric format. Here is shown different possible formats.

```
R0 = 5;  
F1 = 5.0;  
R2 = 0.55r;  
I0 = 0x00001A4F;  
R3 = b00110000110110011100110100001111;
```

First row creates a integer decimal number. Second row creates floating point number, third a fractional number, fourth hexadecimal and fifth binary. However every number is stored either as fixed-point (integers and fractionals) or floating point.

### 3.3 Comments

Commenting is desirable for increasing code readability. Comments in SHARC programming are done exactly like in C. Here is an example of comments.

```
/*  
    This is a commented area  
*/  
R0 = R1+R2;          //This is one row comment
```

So you can use area comments like the first one or one row comments which end after line change.

### 3.4 External Files

You can also write your code in separate .asm files if you prefer that. It however not needed. Separate file should contain specific commands at the start and here is an example.

```
.section /pm seg_pmco;  
.global _my_routine;
```

```
. . .  
  
_my_routine:  
    (some code)  
    rts;  
_my_routine.end:
```

So you should remember to start with section declaration that following is assembly code. Then comes global and the name of routine to be used somewhere else. This enables other files to see this routine in this file. Then when you want to use this function in some other file you will have to use `.extern` command. Here is an example.

```
.section /pm seg_pmco;  
.extern _my_routine;  
  
. . .  
  
routine:  
    call _my_routine;  
    rts;  
routine.end:
```

### 3.5 Loading Data From Text Files

Visual DSP enables a handy way of loading data from text files directly to memory. This way you can for example design filters in Matlab and then export them to a text file which is then read into Visual DSP. Here is an example.

```
.section /dm seg_dmda;  
.var filterCoeffs[] = "FIR-filter.txt";
```

Like all memory reservations, this command goes to memory section like specified by the first row. Second row contains the memory reservation. Now we declare size implicitly so the buffer will reserve space for all the data from the file and none more. File format is white-space separated decimal digits or hexadecimal strings. Commonly for filters you would use one value at a row.

## 4. Registers

ADSP-21262 SHARC has a number of registers for use. There 16 universal registers used for calculations and these use symbols R0-R15 (or F0-F15 when in floating point). For addressing memory a different set of registers is used. These contain I-registers which are used for addressing memory, M-registers for modifying, B-registers for base of circular buffer and L-registers for length of circular buffer. These are divided so that symbols X0-X7 are used with data memory and X8-X15 are used with program memory (X should be replaced with I, M, B or L).

There are also some other registers and you can find full summary of registers in the ADSP-21160 Instruction Set Reference starting from page 2-10.

## 5. Advanced Stuff

Here is some more advanced techniques which SHARC is capable of but should not be needed when doing the course project.

## **5.1 Second Register Sets**

SHARC contains alternate register sets for fast context switching. This means that you can at any time switch part or whole register set to the alternate one and registers not in use will hold their values. This is possible for both universal registers and data address generator registers. This is a useful feature but not really needed in course projects. You can find information about usage in the ADSP-2126x Core Manual.

## **5.2 SIMD (Single Instruction Multiple Data)**

ADSP-21262 SHARC also contains second similar processing element. It can be used in SIMD mode to achieve great performance boost when doing same algorithm for two separate or closely related data like calculating fast fourier transform. Usage needs some careful planning and there should not be any reason to use it within course project because of added complexity.