2.1 Basic Architectural Features

A programmable DSP device should provide instructions similar to a conventional microprocessor. The instruction set of a typical DSP device should include the following,

a. Arithmetic operations such as ADD, SUBTRACT, MULTIPLY etc
b. Logical operations such as AND, OR, NOT, XOR etc
c. Multiply and Accumulate (MAC) operation
d. Signal scaling operation

In addition to the above provisions, the architecture should also include,

a. On chip registers to store immediate results
b. On chip memories to store signal samples (RAM)
c. On chip memories to store filter coefficients (ROM)

1. Investigate the basic features that should be provided in the DSP architecture to be used to implement the following Nth order FIR filter

\[ y(n) = \sum \sum h(i) x(n-i) \quad n=0,1,2\ldots \]

In order to implement the above operation in a DSP, the architecture requires the following features

i. A RAM to store the signal samples x (n)
ii. A ROM to store the filter coefficients h (n)
iii. An MAC unit to perform Multiply and Accumulate operation
iv. An accumulator to store the result immediately
v. A signal pointer to point the signal sample in the memory
vi. A coefficient pointer to point the filter coefficient in the memory
vii. A counter to keep track of the count
viii. A shifter to shift the input samples appropriately
2.2 DSP Computational Building Blocks

Each computational block of the DSP should be optimized for functionality and speed and in the meanwhile the design should be sufficiently general so that it can be easily integrated with other blocks to implement overall DSP systems.

2.2.1 Multipliers

The advent of single chip multipliers paved the way for implementing DSP functions on a VLSI chip. Parallel multipliers replaced the traditional shift and add multipliers nowadays. Parallel multipliers take a single processor cycle to fetch and execute the instruction and to store the result. They are also called as Array multipliers.

The key features to be considered for a multiplier are:

a. Accuracy
b. Dynamic range
c. Speed

The number of bits used to represent the operands decide the accuracy and the dynamic range of the multiplier. Whereas speed is decided by the architecture employed. If the multipliers are implemented using hardware, the speed of execution will be very high but the circuit complexity will also increase considerably. Thus there should be a tradeoff between the speed of execution and the circuit complexity. Hence the choice of the architecture normally depends on the application.

2.2.2 Parallel Multipliers

Consider the multiplication of two unsigned numbers A and B. Let A be represented using m bits as \((A_{m-1} A_{m-2} \ldots A_1 A_0)\) and B be represented using n bits as \((B_{n-1} B_{n-2} \ldots B_1 B_0)\). Then the product of these two numbers is given by,

\[
\begin{array}{cccc}
A_3 & A_2 & A_1 & A_0 \\
B_3 & B_2 & B_1 & B_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
\end{array}
\]

\[
\begin{array}{cccc}
P_7 & P_6 & P_5 & P_4 \\
P_3 & P_2 & P_1 & P_0 \\
\end{array}
\]
This operation can be implemented paralleling using Braun multiplier whose hardware structure is as shown in the figure 2.1.

Fig 2.1 Braun Multiplier for a 4X4 Multiplication

2.2.3 Multipliers for Signed Numbers

In the Braun multiplier the sign of the numbers are not considered into account. In order to implement a multiplier for signed numbers, additional hardware is required to modify the Braun multiplier. The modified multiplier is called as Baugh-Wooley multiplier.

Consider two signed numbers A and B,
2.2.4 Speed

Conventional Shift and Add technique of multiplication requires $n$ cycles to perform the multiplication of two $n$ bit numbers. Whereas in parallel multipliers the time required will be the longest path delay in the combinational circuit used. As DSP applications generally require very high speed, it is desirable to have multipliers operating at the highest possible speed by having parallel implementation.

2.2.5 Bus Widths

Consider the multiplication of two $n$ bit numbers $X$ and $Y$. The product $Z$ can be atmost $2n$ bits long. In order to perform the whole operation in a single execution cycle, we require two buses of width $n$ bits each to fetch the operands $X$ and $Y$ and a bus of width $2n$ bits to store the result $Z$ to the memory. Although this performs the operation faster, it is not an efficient way of implementation as it is expensive.

Many alternatives for the above method have been proposed. One such method is to use the program bus itself to fetch one of the operands after fetching the instruction, thus requiring only one bus to fetch the operands. And the result $Z$ can be stored back to the memory using the same operand bus. But the problem with this is the result $Z$ is $2n$ bits long whereas the operand bus is just $n$ bits long. We have two alternatives to solve this problem,

a. Use the $n$ bits operand bus and save $Z$ at two successive memory locations. Although it stores the exact value of $Z$ in the memory, it takes two cycles to store the result.

b. Discard the lower $n$ bits of the result $Z$ and store only the higher order $n$ bits into the memory. It is not applicable for the applications where accurate result is required.

\[
A = A_{m-1}2^{m-1} + \sum_{i=0}^{m-2} A_i 2^i
\]

\[
B = B_{n-1}2^{n-1} + \sum_{j=0}^{n-2} B_j 2^j
\]

Product $P = P_{m+n-1} \ldots P_1 P_0$

\[
P = A_{m-1}B_{n-1}2^{m+n-2} + \sum_{i=0}^{m-2} \sum_{j=0}^{n-2} A_i B_j 2^{i+j} + \sum_{i=0}^{m-2} A_i B_{n-1}2^{n-1+i} + \sum_{j=0}^{n-2} A_{m-1} B_j 2^{m-1+j}
\]
Another alternative can be used for the applications where speed is not a major concern. In which latches are used for inputs and outputs thus requiring a single bus to fetch the operands and to store the result (Fig 2.2).

![Fig 2.2 A Multiplier with Input and Output Latches](image)

2.2.6 Shifters

Shifters are used to either scale down or scale up operands or the results. The following scenarios give the necessity of a shifter

a. While performing the addition of N numbers each of n bits long, the sum can grow up to n+\log_2 N bits long. If the accumulator is of n bits long, then an overflow error will occur. This can be overcome by using a shifter to scale down the operand by an amount of \log_2 N.

b. Similarly while calculating the product of two n bit numbers, the product can grow up to 2n bits long. Generally the lower n bits get neglected and the sign bit is shifted to save the sign of the product.

c. Finally in case of addition of two floating-point numbers, one of the operands has to be shifted appropriately to make the exponents of two numbers equal.

From the above cases it is clear that, a shifter is required in the architecture of a DSP.

2. It is required to find the sum of 64, 16 bit numbers. How many bits should the accumulator have so that the sum can be computed without the occurrence of overflow error or loss of accuracy?

The sum of 64, 16 bit numbers can grow up to \((16+ \log_2 64)=22\) bits long. Hence the accumulator should be 22 bits long in order to avoid overflow error from occurring.
3. In the previous problem, it is decided to have an accumulator with only 16 bits but shift the numbers before the addition to prevent overflow, by how many bits should each number be shifted?

As the length of the accumulator is fixed, the operands have to be shifted by an amount of $\log_2 64 = 6$ bits prior to addition operation, in order to avoid the condition of overflow.

4. If all the numbers in the previous problem are fixed point integers, what is the actual sum of the numbers?

The actual sum can be obtained by shifting the result by 6 bits towards left side after the sum being computed. Therefore

Actual Sum = Accumulator content $\times 2^6$

2.2.7 Barrel Shifters

In conventional microprocessors, normal shift registers are used for shift operation. As it requires one clock cycle for each shift, it is not desirable for DSP applications, which generally involves more shifts. In other words, for DSP applications as speed is the crucial issue, several shifts are to be accomplished in a single execution cycle. This can be accomplished using a barrel shifter, which connects the input lines representing a word to a group of output lines with the required shifts determined by its control inputs. For an input of length $n$, $\log_2 n$ control lines are required. And an additional control line is required to indicate the direction of the shift.

The block diagram of a typical barrel shifter is as shown in figure 2.3.

Fig 2.3 A Barrel Shifter
Figure 2.4 depicts the implementation of a 4 bit shift right barrel shifter. Shift to right by 0, 1, 2 or 3 bit positions can be controlled by setting the control inputs appropriately.

5. A Barrel Shifter is to be designed with 16 inputs for left shifts from 0 to 15 bits. How many control lines are required to implement the shifter?

As the number of bits used to represent the input are 16, \( \log_2 16 = 4 \) control inputs are required.

2.3 Multiply and Accumulate Unit

Most of the DSP applications require the computation of the sum of the products of a series of successive multiplications. In order to implement such functions a special unit called a Multiply and Accumulate (MAC) unit is required.

A MAC consists of a multiplier and a special register called Accumulator. MACs are used to implement the functions of the type \( A+BC \). A typical MAC unit is as shown in the figure 2.5.
Although addition and multiplication are two different operations, they can be performed in parallel. By the time the multiplier is computing the product, accumulator can accumulate the product of the previous multiplications. Thus if N products are to be accumulated, N-1 multiplications can overlap with N-1 additions. During the very first multiplication, accumulator will be idle and during the last accumulation, multiplier will be idle. Thus N+1 clock cycles are required to compute the sum of N products.

6. If a sum of 256 products is to be computed using a pipelined MAC unit, and if the MAC execution time of the unit is 100nsec, what will be the total time required to complete the operation?

As N=256 in this case, MAC unit requires N+1=257 execution cycles. As the single MAC execution time is 100nsec, the total time required will be, 

\[(257 \times 100\text{nsec}) = 25.7\mu\text{sec}\]
2.3.1 Overflow and Underflow

While designing a MAC unit, attention has to be paid to the word sizes encountered at the input of the multiplier and the sizes of the add/subtract unit and the accumulator, as there is a possibility of overflow and underflows.

Overflow/underflow can be avoided by using any of the following methods viz

a. Using shifters at the input and the output of the MAC

b. Providing guard bits in the accumulator

c. Using saturation logic

Shifters

Shifters can be provided at the input of the MAC to normalize the data and at the output to denormalize the same.

Guard bits

As the normalization process does not yield accurate result, it is not desirable for some applications. In such cases we have another alternative by providing additional bits called guard bits in the accumulator so that there will not be any overflow error. Here the add/subtract unit also has to be modified appropriately to manage the additional bits of the accumulator.

7. Consider a MAC unit whose inputs are 16 bit numbers. If 256 products are to be summed up in this MAC, how many guard bits should be provided for the accumulator to prevent overflow condition from occurring?

As it is required to calculate the sum of 256, 16 bit numbers, the sum can be as long as \((16+ \log_2 256)=24\) bits. Hence the accumulator should be capable of handling these 22 bits. Thus the guard bits required will be \((24-16)=8\) bits.

The block diagram of the modified MAC after considering the guard or extention bits is as shown in the figure 2.6.
8. What should be the minimum width of the accumulator in a DSP device that receives 10 bit A/D samples and is required to add 64 of them without causing an overflow?

As it is required to calculate the sum of 64, 10 bit numbers, the sum can be as long as \((10 + \log_2 64) = 16\) bits. Hence the accumulator should be capable of handling these 16 bits. Thus the guard bits required will be \((16-10) = 6\) bits.
Saturation Logic

Overflow/underflow will occur if the result goes beyond the most positive number or below the least negative number the accumulator can handle. Thus the overflow/underflow error can be resolved by loading the accumulator with the most positive number which it can handle at the time of overflow and the least negative number that it can handle at the time of underflow. This method is called as saturation logic. A schematic diagram of saturation logic is as shown in figure 2.7.

In saturation logic, as soon as an overflow or underflow condition is satisfied the accumulator will be loaded with the most positive or least negative number overriding the result computed by the MAC unit.

2.4 Arithmetic and Logic Unit

A typical DSP device should be capable of handling arithmetic instructions like ADD, SUB, INC, DEC etc and logical operations like AND, OR, NOT, XOR etc. The block diagram of a typical ALU for a DSP is as shown in the figure 2.8.

It consists of status flag register, register file and multiplexers.
Status Flags

ALU includes circuitry to generate status flags after arithmetic and logic operations. These flags include sign, zero, carry and overflow.

Overflow Management

Depending on the status of overflow and sign flags, the saturation logic can be used to limit the accumulator content.

Register File

Instead of moving data in and out of the memory during the operation, for better speed, a large set of general purpose registers are provided to store the intermediate results.
2.5 Bus Architecture and Memory

Conventional microprocessors use Von Neumann architecture for memory management wherein the same memory is used to store both the program and data (Fig 2.9). Although this architecture is simple, it takes more number of processor cycles for the execution of a single instruction as the same bus is used for both data and program.

![Fig 2.9 Von Neumann Architecture](image)

In order to increase the speed of operation, separate memories were used to store program and data and a separate set of data and address buses have been given to both memories, the architecture called as Harvard Architecture. It is as shown in figure 2.10.

![Fig 2.10 Harvard Architecture](image)

Although the usage of separate memories for data and the instruction speeds up the processing, it will not completely solve the problem. As many of the DSP instructions require more than one operand, use of a single data memory leads to the fetch the operands one after the other, thus increasing the delay of processing. This problem can be
overcome by using two separate data memories for storing operands separately, thus in a single clock cycle both the operands can be fetched together (Figure 2.11).

Fig 2.11 Harvard Architecture with Dual Data Memory

Although the above architecture improves the speed of operation, it requires more hardware and interconnections, thus increasing the cost and complexity of the system. Therefore there should be a trade off between the cost and speed while selecting memory architecture for a DSP.

2.5.1 On-chip Memories

In order to have a faster execution of the DSP functions, it is desirable to have some memory located on chip. As dedicated buses are used to access the memory, on-chip memories are faster. Speed and size are the two key parameters to be considered with respect to the on-chip memories.

**Speed**

On-chip memories should match the speeds of the ALU operations in order to maintain the single cycle instruction execution of the DSP.
Size
In a given area of the DSP chip, it is desirable to implement as many DSP functions as possible. Thus the area occupied by the on-chip memory should be minimum so that there will be a scope for implementing more number of DSP functions on-chip.

2.5.2 Organization of On-chip Memories

Ideally whole memory required for the implementation of any DSP algorithm has to reside on-chip so that the whole processing can be completed in a single execution cycle. Although it looks as a better solution, it consumes more space on chip, reducing the scope for implementing any functional block on-chip, which in turn reduces the speed of execution. Hence some other alternatives have to be thought of. The following are some other ways in which the on-chip memory can be organized.

a. As many DSP algorithms require instructions to be executed repeatedly, the instruction can be stored in the external memory, once it is fetched can reside in the instruction cache.
b. The access times for memories on-chip should be sufficiently small so that it can be accessed more than once in every execution cycle.
c. On-chip memories can be configured dynamically so that they can serve different purpose at different times.

2.6 Data Addressing Capabilities

Data accessing capability of a programmable DSP device is configured by means of its addressing modes. The summary of the addressing modes used in DSP is as shown in the table below.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Operand</th>
<th>Sample Format</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Immediate Value</td>
<td>ADD #imm</td>
<td>#imm +A → A</td>
</tr>
<tr>
<td>Register</td>
<td>Register Contents</td>
<td>ADD reg</td>
<td>reg +A → A</td>
</tr>
<tr>
<td>Direct</td>
<td>Memory Address Register</td>
<td>ADD mem</td>
<td>mem+A → A</td>
</tr>
<tr>
<td>Indirect</td>
<td>Memory contents with address in the register</td>
<td>ADD *addrreg</td>
<td>*addrreg +A → A</td>
</tr>
</tbody>
</table>
2.6.1 Immediate Addressing Mode

In this addressing mode, data is included in the instruction itself.

2.6.2 Register Addressing Mode

In this mode, one of the registers will be holding the data and the register has to be specified in the instruction.

2.6.3 Direct Addressing Mode

In this addressing mode, instruction holds the memory location of the operand.

2.6.4 Indirect Addressing Mode

In this addressing mode, the operand is accessed using a pointer. A pointer is generally a register, which holds the address of the location where the operands resides.

Indirect addressing mode can be extended to inculcate automatic increment or decrement capabilities, which has lead to the following addressing modes.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Sample Format</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post Increment</td>
<td>ADD *addreg+</td>
<td>A ➝ A + *addreg addreg ➝ addreg+1</td>
</tr>
<tr>
<td>Post Decrement</td>
<td>ADD *addreg-</td>
<td>A ➝ A + *addreg addreg ➝ addreg-1</td>
</tr>
<tr>
<td>Pre Increment</td>
<td>ADD +*addreg</td>
<td>addreg ➝ addreg+1</td>
</tr>
<tr>
<td>Pre Decrement</td>
<td>ADD -*addreg</td>
<td>A ➝ A + *addreg addreg ➝ addreg-1</td>
</tr>
<tr>
<td>Post_Add_Offset</td>
<td>ADD *addreg, offsetreg+</td>
<td>A ➝ A + *addreg addreg ➝ addreg+offsetreg</td>
</tr>
<tr>
<td>Post_Sub_Offset</td>
<td>ADD *addreg, offsetreg-</td>
<td>A ➝ A + *addreg addreg ➝ addreg-offsetreg</td>
</tr>
<tr>
<td>Pre_Add_Offset</td>
<td>ADD offsetreg+,*addreg</td>
<td>addreg ➝ addreg+offsetreg</td>
</tr>
<tr>
<td>Pre_Sub_Offset</td>
<td>ADD offsetreg-,*addreg</td>
<td>A ➝ A + *addreg addreg ➝ addreg-offsetreg</td>
</tr>
</tbody>
</table>
9. What are the memory addresses of the operands in each of the following cases of indirect addressing modes? In each case, what will be the content of the \texttt{addreg} after the memory access? Assume that the initial contents of the \texttt{addreg} and the \texttt{offsetreg} are 0200h and 0010h, respectively.
   a. ADD \texttt{*addreg-}
   b. ADD \texttt{+*addreg}
   c. ADD \texttt{offsetreg+,*addreg}
   d. ADD \texttt{*addreg,offsetreg-}

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Addressing Mode</th>
<th>Operand Address</th>
<th>addrreg Content after Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD \texttt{*addreg-}</td>
<td>Post Decrement</td>
<td>0200h</td>
<td>0200-01=01FFh</td>
</tr>
<tr>
<td>ADD \texttt{+*addreg}</td>
<td>Pre Increment</td>
<td>0200+01=0201h</td>
<td>0201h</td>
</tr>
<tr>
<td>ADD \texttt{offsetreg+,*addreg}</td>
<td>Pre_Add_Offset</td>
<td>0200+0010=0210h</td>
<td>0210h</td>
</tr>
<tr>
<td>ADD \texttt{*addreg,offsetreg-}</td>
<td>Post_Sub_Offset</td>
<td>0200h</td>
<td>0200-0010=01F0h</td>
</tr>
</tbody>
</table>

10. Identify the addressing modes of the operands in each of the following instructions
   a. ADD \texttt{#1234h}
   b. ADD \texttt{1234h}
   c. ADD \texttt{*AR+}
   d. ADD \texttt{offsetreg-,*AR}

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD \texttt{#1234h}</td>
<td>Immediate Addressing Mode</td>
</tr>
<tr>
<td>ADD \texttt{1234h}</td>
<td>Direct Addressing Mode</td>
</tr>
<tr>
<td>ADD \texttt{*AR+}</td>
<td>Post Increment Indirect Addressing Mode</td>
</tr>
<tr>
<td>ADD \texttt{offsetreg-,*AR}</td>
<td>Pre-Sub_Offset Indirect Addressing Mode</td>
</tr>
</tbody>
</table>

2.7 Special Addressing Modes

For the implementation of some real time applications in DSP, normal addressing modes will not completely serve the purpose. Thus some special addressing modes are required for such applications.
2.7.1 Circular Addressing Mode

While processing the data samples coming continuously in a sequential manner, circular buffers are used. In a circular buffer the data samples are stored sequentially from the initial location till the buffer gets filled up. Once the buffer gets filled up, the next data samples will get stored once again from the initial location. This process can go forever as long as the data samples are processed in a rate faster than the incoming data rate.

Circular Addressing mode requires three registers viz
a. Pointer register to hold the current location (PNTR)
b. Start Address Register to hold the starting address of the buffer (SAR)
c. End Address Register to hold the ending address of the buffer (EAR)

There are four special cases in this addressing mode. They are
a. SAR < EAR & updated PNTR > EAR
b. SAR < EAR & updated PNTR < SAR
c. SAR > EAR & updated PNTR > SAR
d. SAR > EAR & updated PNTR < EAR

The buffer length in the first two case will be (EAR-SAR+1) whereas for the next tow cases (SAR-EAR+1)

The pointer updating algorithm for the circular addressing mode is as shown below.

; Pointer Updating Algorithm

Updated PNTR ← PNTR ± increment

IF SAR < EAR
And if Updated PNTR > EAR then
   New PNTR ← Updated PNTR – Buffer size
And if Updated PNTR < SAR then
   New PNTR ← Updated PNTR + Buffer size

IF SAR > EAR
And if Updated PNTR > SAR then
   New PNTR ← Updated PNTR – Buffer size
And if Updated PNTR < EAR then
   New PNTR ← Updated PNTR + Buffer size

Else
   New PNTR ← Updated PNTR
Four cases explained earlier are as shown in the figure 2.12.

Fig 2.12 Special Cases in Circular Addressing Mode
11. A DSP has a circular buffer with the start and the end addresses as 0200h and 020Fh respectively. What would be the new values of the address pointer of the buffer if, in the course of address computation, it gets updated to
   a. 0212h
   b. 01FCh

Buffer Length = (EAR-SAR+1) = 020F-0200+1 = 10h

   a. New Address Pointer = Updated Pointer - buffer length = 0212 - 10 = 0202h
   b. New Address Pointer = Updated Pointer + buffer length = 01FC + 10 = 020Ch

12. Repeat the previous problem for SAR = 0210h and EAR = 0201h

Buffer Length = (SAR-EAR+1) = 0210-0201+1 = 10h

   c. New Address Pointer = Updated Pointer - buffer length = 0212 - 10 = 0202h
   d. New Address Pointer = Updated Pointer + buffer length = 01FC + 10 = 020Ch

2.7.2 Bit Reversed Addressing Mode

To implement FFT algorithms we need to access the data in a bit reversed manner. Hence a special addressing mode called bit reversed addressing mode is used to calculate the index of the next data to be fetched.

It works as follows. Start with index 0. The present index can be calculated by adding half the FFT length to the previous index in a bit reversed manner, carry being propagated from MSB to LSB.

\[ \text{Current index} = \text{Previous index} + B \left( \frac{1}{2} \text{FFT Size} \right) \]

13. Compute the indices for an 8-point FFT using Bit reversed Addressing Mode

Start with index 0. Therefore the first index would be (000)

Next index can be calculated by adding half the FFT length, in this case it is (100) to the previous index.

i.e. Present Index = (000) + B (100) = (100)

Similarly the next index can be calculated as

Present Index = (100) + B (100) = (010)

The process continues till all the indices are calculated. The following table summarizes the calculation.
<table>
<thead>
<tr>
<th>Index in Binary</th>
<th>BCD value</th>
<th>Bit reversed index</th>
<th>BCD value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>100</td>
<td>4</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>011</td>
<td>3</td>
<td>110</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>111</td>
<td>7</td>
<td>111</td>
<td>7</td>
</tr>
</tbody>
</table>

2.8 Address Generation Unit

The main job of the Address Generation Unit is to generate the address of the operands required to carry out the operation. They have to work fast in order to satisfy the timing constraints.

As the address generation unit has to perform some mathematical operations in order to calculate the operand address, it is provided with a separate ALU.

Address generation typically involves one of the following operations.

a. Getting value from immediate operand, register or a memory location
b. Incrementing/ decrementing the current address
c. Adding/subtracting the offset from the current address
d. Adding/subtracting the offset from the current address and generating new address according to circular addressing mode
e. Generating new address using bit reversed addressing mode

The block diagram of a typical address generation unit is as shown in figure 2.13.
2.9 Programmability and Program Control

A programmable DSP device should provide the programming capability involving branching, looping and subroutines.

The implementation of repeat capability should be hardware based so that it can be programmed with minimal or zero overhead. A dedicated register can be used as a counter.

In a normal subroutine call, return address has to be stored in a stack thus requiring memory access for storing and retrieving the return address, which in turn reduces the speed of operation. Hence a LIFO memory can be directly interfaced with the program counter.

2.9.1 Program Control

Like microprocessors, DSP also requires a control unit to provide necessary control and timing signals for the proper execution of the instructions. In
microprocessors, the controlling is micro coded based where each instruction is divided into microinstructions stored in micro memory. As this mechanism is slower, it is not applicable for DSP applications. Hence in DSP the controlling is hardwired base where the Control unit is designed as a single, comprehensive, hardware unit. Although it is more complex it is faster.

2.9.2 Program Sequencer

It is a part of the control unit used to generate instruction addresses in sequence needed to access instructions. It calculates the address of the next instruction to be fetched. The next address can be from one of the following sources.

a. Program Counter
b. Instruction register in case of branching, looping and subroutine calls
c. Interrupt Vector table
d. Stack which holds the return address

The block diagram of a program sequencer is as shown in figure 2.14.

Fig 2.14 Program Sequencer
Program sequencer should have the following circuitry

a. PC has to be updated after every fetch

b. Counter to hold count in case of looping

c. A logic block to check conditions for conditional jump instructions

d. Condition logic-status flag