Chapter 8: Development of Machine Instruction set Simulator

8.1 Introduction

After having designed the ASIP, it was now required to convert the ‘C’ code of Klatt’s synthesizer into the machine language code of the ASIP. We decided to do this conversion manually in two steps. In the first step we manually translated the Klatt’s ‘C’ code into the mnemonic level code for the ASIP. In the second step we manually translated the mnemonic level code into the machine language code of the ASIP. The code so generated needed to be verified for its correctness vis-à-vis the Klatt’s ‘C’ code. While the machine language code could be simulated directly on the synthesizable RT-level VHDL model of the ASIP, however it was decided not to do so for the following reason: if the results of the ‘C’ code and the machine language code differed, it would be difficult to distinguish whether the differences resulted from an error/errors in manually generating the machine language code from the ‘C’ code or that there was an error (or errors) in the synthesizable RT-level VHDL code itself (that is in the processor hardware design itself).

It was therefore felt necessary that an instruction set simulator (that implements the functions of machine language instructions in software) be independently developed. This would also serve as a reference for comparison when the RT-level VHDL code of the processor and subsequently its post-synthesis gate-level version also would be functionally tested for design validation. Accordingly, we developed an instruction set simulator (e.g. machine language instruction set simulator) for the ASIP.

The instruction simulator is a ‘C’ code which calls a function whose job is to examine the bit pattern contained in the instruction register (a string variable) and returns an integer value (corresponding to the bit pattern in the Opcode field of the instruction). Based on this returned integer value (corresponding to the Opcode bit pattern) a switch statement selects the corresponding case branch that implements the function of the instruction. By function of the instruction we imply both its computational and housekeeping tasks (e.g. updating of the program counter and fetching and loading of the next instruction in the instruction register) so that a chained execution of instructions in the program code can be achieved.
This instruction set simulator was extensively used to both validate the RT-level synthesizable VHDL code, as well as to debug the manually generated machine language code of the application.

Since the complete instruction set simulator is a large code, the code of a reduced version of the instruction set simulator implementing a tiny subset of only two instructions from the full instruction set is reproduced below as an illustration.

The machine language C code is:

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <strings.h>

main ( )
{
    char *IR;
    char opcode[7]; //make size bigger by 1
    static float fextn_word;
    static int src_int, dest_int, opcode_int, data, data1, data2;
    static float fdata, fdata1, fdata2;
    char PC[11];
    static int PC_int;
    int Mint[10];
    float Mfp[10];
    int i;

    for ( i = 1; i < 2; i++ ) {
        Mint[i] = 2;
        Mfp[i] = 2.0;
    }

    IR = "00000000000000010000000010000000"; //ADDII
    Execute (IR);
    IR = "00101100000000010000000000000000"; //MULTFC
    fextn_word = 0.5; //
    Execute (IR);
}
```
The corresponding decoder and the description of the functions used in the instruction set simulator are given below:

```c
void execute (char IR [])
{
    find_opcode (IR);
    switch (opcode_int)
    {
        case 0 :        // ADDII
            find_src (IR);
            find_dest (IR);
            data 1 = MINT [src_int];
            data 2 = MINT [dest_int];
            data = data 1 + data 2;
            MINT [dest_int] = data;
            break;
            ---
        case 11 :       // MULTFC
            find_dest (IR);
            fdata2 = fextn_word;
            fdata1 = MFP [dest_int];
            fdata = fdata1 * fdata2;
            MFP [dest_int] = fdata;
            break;
            --
        ---
    }

    void find_opcode (char IR []) // Find_opcode Function
```
{  
    int i;  
    for (i=0; i<6; i++)  
        opcode[i] = IR[i];  
    opcode_int = decode (opcode);  
    return;  
}

int decode (char val[]) // Decode Function
{
    int str_int_val = 0;  
    int i, M;  
    int k = 1;  
    M = strlen (val);  
    for (i=0; i<M; i++)  
    {
        if (val[M-i-1] == '1')  
            str_int_val += k;  
        k = k*2;  
    }
    return (str_int_val);  
}

void find_src (char IR[]) // Find_src Function
{
    int i;  
    for (i=16; i<26; i++)  
        src_ptr[i-16] = IR[i];  
    src_int = decode (src_ptr);  
    return;  
}
void find_dest (char IR[ ]) // Find_dest Function:
{
    int i;
    for (i=6; i<16; i++)
        dest_ptr[i-6] = IR[i];
    dest_int=decode (dest_ptr);
    return;
}

Here function find_opcode () finds the opcode of the instruction and function decode returns its integer value. The functions find_src () and find_dest () find the integer value of the source pointer address (parameter 1) and the destination pointer (as well as the second source) (parameter 2) respectively and return the two integer parameters, namely: src_int and dest_int. Here MINT is the integer memory, MFP is the floating-point memory and MPRO is the program memory. The operator ‘+’ represent the functional unit for integer addition and ‘*’ represent the functional unit for multiplication.