VHDL MODELLING OF THE MAC-1 ARCHITECTURE

Giuliano de Souza Vilela Cid[1]

[1]Informatic and Applied Mathematics Department (DIMAp)
Universidade Federal do Rio Grande do Norte (UFRN). Natal/RN, Brasil.
[1]E-mail: giulianoxt@gmail.com

ABSTRACT

This paper presents a VHDL model specification for the MAC-1 architecture proposed by Andrew Tanenbaum in [1]. The model takes different approaches from the ones originally designed in respect to the control logic and the final design. It's also shown how the development and testing process ocurred using the Altera Quartus II software and a FPGA board.

1. OVERVIEW OF THE MAC ARCHITECTURE

The MAC-1 architecture, as Tanenbaum presents it, is a multicicle machine consisting in a small instruction set, a group of registers, an ALU and three internal buses. Both data and instruction buses are 16 bits wide. It's very simple design is not oriented towards efficiency, hence it's use is more expressive as a academic tool for teaching in computer architecture introductory courses.

This architecture is refined in two main blocks: operative block and control unit. These two define an interface between them, with the control unit sending control signals to the operative block, determining the current state of the processor.

1.1. Operative block

This block is responsible for all the logic, arithmetic, memory-based and register aware operations. It contains the basic blocks upon wich the main architecture is built, and where all the elementary operations upon data are done, using only a few registers and a single ALU. From the outside, the MAC architecture instruction set doesn't expose the details of this component: in fact, only two registers are visible to the outside programmer.

From the inside, the data path designed by Tanenbaum is very simple and leads to an elegant and concise stack based platform.

This block executes instructions that come from the memory under a certain bus. From there, the programmer can work with an acumulator (AC) register, where temporary data can be stored while it's needed in the operative block, and with an stack point (SP) register. With SP comes the ability to push and pop data from a certain memory region. The main design is shown in the following picture.

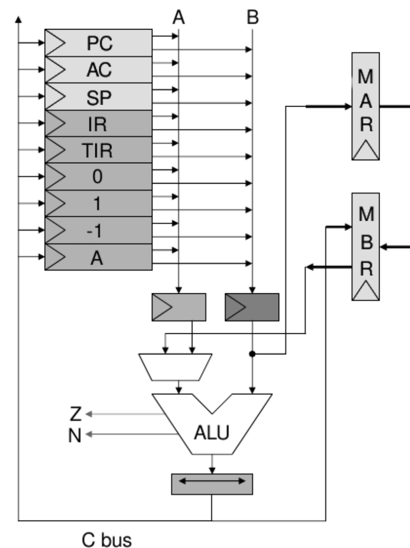

Illustration 1: Data path - operative block

Values and instructions are constantly moved between the 3 buses, the memory and the registers, according to the current state of the processor. Data is operated in the ALU, wich performs addition, subtraction, binary and / or and negation. Addresses of memory positions must be put in the MAR (*Memory Address Register)* register so that the position can be either read or written. Values that were read from memory come from the MBR (*Memory Bus Register)* register.

The AC register is the main placeholder for the architecture. Almost all instructions use it in one way or another, from the destination of the result of a calculation to being the holder of a memory address to be read or written. For simplicity, all the signals hooked up to the operative block, coming from the control unit, are missing from Illustration 1.

1.2. Control unit

The control unit is responsible for sending control signals to all the components of the architecture. It's designed with correctness and efficiency in mind, leading to a few different modelling approaches. The control unit holds the current state of the processor, and knows how to instruct the operative block about the operation to be performed in the current cycle. In [1] it's shown a

microprogramming approach to the implementation of the MAC-1 control unit.

The basic idea is that, to control the underlying architecture, called MIC-1, that supports the MAC-1 instruction set we could make a small interpreter for a micro-instruction set that controls the operative block. Then, we could store a small microprogram in a memory inside the control unit. With it's own sequential logic and well defined behaviour, this microprogram would run "forever", interpreting the MAC instruction set and, at each clock step, interpreting a single MIC-1 micro-instruction and generating a bigger set of control signals to rule how the architecture is currently executing a MAC-1 instruction.

1.3 The MAC-1 instruction set

The VHDL specification that was made supports the entire instruction set designed in [1]. For brevity, what is shown here is only a subset of the original (without the encodings for the instructions).

Mnemonic	Semantics
`lodd x;`	`ac := m[x];`
`stod x;`	`m[x] := ac;`
`addd x;`	`ac := ac + m[x];`
`subd x;`	`ac := ac – m[x];`
`jpos x;`	`if ac >= 0: pc := x;`
`jump x;`	`pc := x;`
`loco x;`	`ac := x;`
`lodl x;`	`ac := m[sp + x];`
`stol x;`	`m[sp + x] := ac;`
`pshi x;`	`sp := sp – 1;` `m[sp] := m[ac];`
`popi x;`	`m[ac] := m[sp];` `sp := sp + 1;`
`swap;`	`tmp := ac; ac := sp;` `sp := tmp;`

Table 1: MAC-1 instruction set

2. SPECIFICATION DETAILS

Although simple and efficient, the microprogramming-based implementation design proposed in [1] has some problems. Let's take for example a single micro-instruction, wich might look like this using a pseudo notation:

```
mar := ir; mbr := ac; wr;
```

This single micro-instruction states how the MIC-1 underlying architecture will behave at a single clock step. It is stored in the microprogram memory as a set of control signals (written as signal:value, where signal is a control signal identification and value is the non-zero value of that signal):

```
alu:2; mbr:1; mar:1; wd:1; b:3; a:1;
```

As one can see, it is hard to both program in this micro-language and to read a micro-program. This is due to the fact that the language isn't designed for human programming, only to machine interpreting and execution.

Also, when the microprogram is highly optimized, we start to see some odd behaviour. The design tends to get like a pipeline processor, executing steps of two or more instructions simultaneously, without it actually being one. This has serious implications when a teacher wants to present this kind of design to his students.

Taking these and others aspects into account, some specification decisions where made to simplify the VHDL code. These will be shown bellow.

2.1. Control unit specification

The control unit design chosen for this specification uses a finite state machine implementation. This leads to a common pattern used in VHDL modelling: each state is represented by a code block in the specification. Both the current state and the next state being signals of a user defined type. That way, it was possible to keep the processor design very simple and intuitive.

Each step taken to decode, execute and store results of a single instruction is represented by a single processor state. We are able to see clearly the interactions between specific states of the processor.

For example, observe the state transition diagram representing the execution of the following program:

```
0 : loco x;
1 : exit;
```

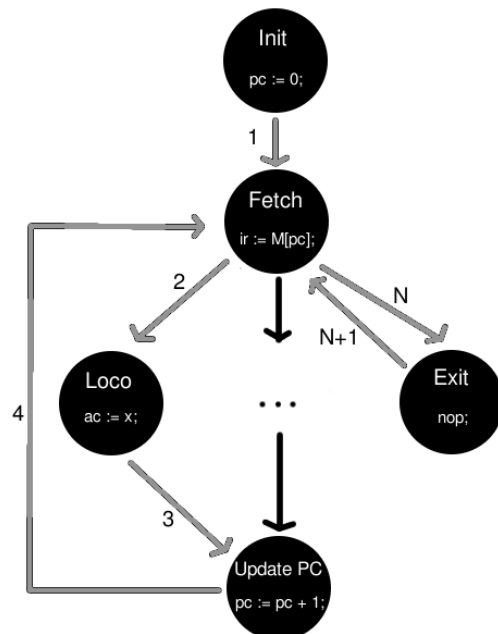

Illustration 2: State transition diagram

The diagram used in Illustration 2 shows the current state of the processor at each step of the execution of the program. Under the labels for the states are corresponding pseudo-codes for each task. Along the transition edges are numbers, showing the order of execution of each state. This picture, however, is very simplified. It doesn't show a decoding state, and assumes that every instruction can execute itself in just one state. That is not true. Almost all instructions interpreted by the processor executes in more than one state / clock cycle. There is actually a direct mapping from a multicicle based implementation, meaning that each state can be seen as a subcycle executing.

Among the advantages of this approach are: code readability and simplicity, easy debugging, etc. Also, in a microprogramming approach, the designer has to write, possibly directly in the VHDL source, the entire microprogram. It is extensive and not intuitive. The approach chosen here doesn't present this problem.

2.2. Operative and memory block specification

It is in this part of the specification that most of the high level funcionality of VHDL, as supported by the Quartus software, was used. The components described in [1] for the MAC-1 architecture are mostly of eletronic low level. By choosing VHDL as the hardware speficiation language, it was possible to write concise and clean “high level” code. This way, the Quartus compiler and optimizer can work on the code and make the best implementation decisions for the platform chosen (a Altera FPGA board).

The VHDL libraries and mega-functions provided by the Quartus software helped in this sense, since they contained useful implementations for arithmetic operations, type conversions, etc. Specifically, the lpm_ram_dq mega-function was fully used for the memory block implementation.

2.3. Code snippets

Here I present a small part of the VHDL specification, divided in several sections. They show the main concepts and structures used in the development.

2.3.1. Code 1 (Memory block)

```
component lpm_ram_dq is
  generic (
    lpm_address_control: string;
    lpm_file: string;
    lpm_indata: string;
    lpm_outdata: string;
    lpm_type: string;
    lpm_width: natural;
    lpm_widthad: natural
  );
  port (
    outclock : in std_logic;
    address: in
      std_logic_vector(5 downto 0);
    inclock: in std_logic;
    q: out
      std_logic_vector(15 downto 0);
    data: in
      std_logic_vector(15 downto 0);
    we: in std_logic
  );
end component;
```

This part shows the instanciation of the lpm_ram_dq megafunction. It provides a convenient RAM specification to plug.

2.3.2. Code 2 (Operative block)

```
AcReg: register_n
   generic map (data_size)
   port map (bar_c,clk,reg_rd(0),ac_val);

SpReg: register_n
   generic map (data_size)
   port map (bar_c,clk,reg_rd(1),sp_val);

AReg: register_n
   generic map (data_size)
   port map (bar_c,clk,reg_rd(2),a_val);
```

A very small snippet of the operative block code. Only shows the main instanciations of components (register), that compose an entity close to Illustration 1.

2.3.3. Code 3 (Control unit state machine)

```
UpdateState:
process(clk) begin
  if (clk'event and clk = '1') then
    st_now <= st_next;
  end if;
end process;

ProcessState:
process(st_now) begin
  case st_now is
    when init => -- pc := 0;
      mem_addr <= (others => 'X');
      mem_wr_data <= (others => 'X');
      mem_wr <= '0';
      op_aux <= (others => 'X');
      op_aux_en <= 'X';
      alu_op <= 'X';
      alu_out_rd <= '0';
      reg_rd <= (others => '0');
      reg_wa <= (others => 'X');
      reg_wb <= (others => 'X');
      pc_cond <= "011"; -- reset
      st_next <= fetch0;
```

The first process represents the mechanism used for state transitions (wich are synchronous and sequential). The second process shows how the states are represented. A *case* statement detects with state is the current state, and according to that it determines the behaviour of each state. On the snippet it's shown the *init*

state. It's purpose is to set the initial value for *pc* and begin the fetching of instructions on the next clock cycle. All the others signals wich are irrelevant for this state are set to 'X' (*don't care* value).

2.3.4. Code 4 (Main block)

```
entity mac_proc is
  port (
    clk_board: in std_logic;
    ac : out
      std_logic_vector(15 downto 0);
    ...
  );
end mac_proc;
...
begin
...
  OpBlock : op_block
  generic map (16)
  port map (
    clk,op_aux,op_aux_en,
    alu_out_rd,
    reg_rd,reg_wa,reg_wb,
    alu_op,alu_val,ac_val,sp_val
  );

  ControlUnit : control
  generic map (16, 12)
  port map (
    clk,
    mem_data,mem_wr_data,mem_addr,mem_wr,
    op_aux,op_aux_en,
    alu_val,ac_val,sp_val,
    alu_op,alu_out_rd,
    reg_rd,reg_wa,reg_wb
  );
end mac_proc;
```

This snippet shows the main entity. It glues together all the components, showing how the signals interact with each other.

3. SIMULATION AND FINAL TESTING

The simulation and testing process was divided in two phases: one software-based and the other hardware-based.

At first, the Quartus software simulation features where used to debug the VHDL code. The test programs are currently being stored in a *mif* (*Memory Initialisation File*) file. When the code is compiled, the memory block is initialized with the contents described in that file. After that, the code is then simulated using a *vector waverform file*, where the final results produced by the simulation can be seen.

One of the MAC-1 programs used to test the processor is shown bellow. It is written according to the *mif* notation used by Quartus.

```
0 : 0111000000000010; -- loco 2; .inc_x
1 : 0001000000111011; -- stod 59;
2 : 0111000000000000; -- loco 0;
3 : 0001000000111100; -- stod 60;.i
4 : 0001000000111101; -- stod 61;.x
5 : 0111000000000010; -- loco 2; .lim
6 : 0001000000111110; -- stod 62;
7 : 0111000000000001; -- loco 1; .inc_i
8 : 0001000000111111; -- stod 63;
9 : 0000000000111100; -- lodd 60;
10: 0011000000111110; -- subd 62;.i!=lim?
11: 0101000000010011; -- jzer 19;.goto 19
12: 0000000000111100; -- lodd 60;
13: 0010000000111111; -- addd 63;.i++
14: 0001000000111100; -- stod 60;
15: 0000000000111101; -- lodd 61;
16: 0010000000111011; -- addd 59;.x+=incx
17: 0001000000111101; -- stod 61;
18: 0110000000001001; -- jump 9; .goto 9
19: 0000000000111101; -- lodd 61;.show x
20: 1111111111111111; -- exit;
```

The program is basically a for loop, looking like this in C notation:

```
int i =0, x =0; for (;i != 2;i+=1,x+=2);
```

The code is then ran into the Quartus simulation tool, were all the results and internal signals of the architecture can be seen, as Illustration 3 shows.

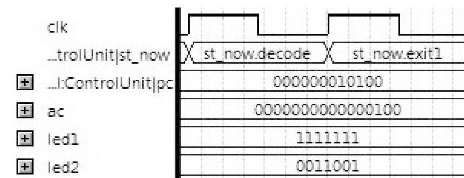

Illustration 3: Output showing ac = 4 (x value)

After that, the processor was tested in a FLEX10K family Altera FPGA board. The results were also successful, with the correct value for x (4) being shown in a 7-segment led display in a decimal notation.

4. CONCLUSION

It was described how the VHDL model for the MAC-1 architecture was made and wich development decisions were taken. Finally, the result was proved to be successful using software simulation and a FPGA board, both Altera tools. Further projects related to this work are being planned. The idea is to integrate this model in a higher architecture development environment, focused on computer organization teaching.

5. REFERENCES

[1] A. S. Tanenbaum, *Structured Computer Organization*, Prentice Hall, June 2001.

[2] D. A. Patterson, J. L. Hennessy and P. J. Ashenden, *Computer Organization and Design,* Morgan Kauffman, August 2004.

[3] D. A. Patterson, J. L. Hennessy, *Computer Organization: a Quantitative approach,* Morgan Kauffman, May 2002.