# Lab 4 Tutorial

## Design of A Mini Load/Store Machine

CSE 140L, Fall 05  
Instructor: C.K. Cheng  
Computer Science and Engineering  
University of California San Diego  
November 25, 2005

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>By</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nov. 25, 2005</td>
<td>Haikun Zhu, Rui Shi</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Nov. 27, 2005</td>
<td>Haikun Zhu</td>
<td>Proofread</td>
</tr>
<tr>
<td>1.2</td>
<td>Nov. 28, 2005</td>
<td>Haikun Zhu</td>
<td>Corrected VHDL syntax error in Fig. 12. Thanks to Brian.</td>
</tr>
<tr>
<td>1.3</td>
<td>Nov. 28, 2005</td>
<td>Haikun Zhu</td>
<td>Modified Fig. 3; added missing control signal from the instruction decoder to the datapath.</td>
</tr>
</tbody>
</table>
1.0 Introduction

This tutorial tries to cover all the essential skills one will need to accomplish lab 4.

2.0 System Specification

Please refer to the webpage of lab 4 assignment (http://www.cse.ucsd.edu/classes/fa05/cse140L/Lab4/lab4.htm) for the system specification.

3.0 Design of System Architecture

We start the design by working out the system architecture and interface signals between each components. According to the specification, our mini computer system consists of the following five components:

- **Program Counter**: The program counter is the address pointer for fetching the instructions in the instruction memory.
- **Instruction Memory**: The instruction memory can hold up to 16 instructions, each of which is 7-bit wide.
- **Instruction Decoder**: The instruction decoder is the control center of the whole system. It specifies which arithmetic operation the datapath is going to execute, and takes care of the scheduling of registers.
- **Datapath**: The datapath is a pure combinational module which performs the useful computing.
- **Registers**: The registers, along with the instruction memory, provide operands for and receive the results from the datapath.

Since the capacity of the instruction memory is 16, we need a 4-bit address $addr[3:0]$ from the program counter to have complete access to the memory. Each instruction from the memory is disassembled into two parts: the higher 3 bits $instr[6:4]$ are sent to the instruction decoder to decide the operation of the next cycle. The lower four bits $instr[3:0]$, which are the data portion of the instruction, are sent to the datapath for processing or stored in the registers. The datapath can also receive a 4-bit operand from the register files; and the results, including a 4-bit number and two flags (overflow and comparison) are stored back into the registers. The control signals emanating from the instruction decoder are yet to be decided after the datapath and register files are fully studied.

From the instruction set we find there are 4 arithmetic operations: **ADD**, **SHIFT**, **Mask** and **COMP**. Each of the former three produces a 4-bit number. During each cycle, we need to make a decision that which of the three 4-bit numbers is going to be stored back into the register files. Therefore, we will need a 2-bit control signal from the instruction decoder to do this task. We then make a list of the sources and destinations of the four arithmetic operations as shown in Table 1.

<table>
<thead>
<tr>
<th>operation</th>
<th>source</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R1, R2</td>
<td>R3, overflow flag</td>
</tr>
<tr>
<td>COMP</td>
<td>R1, R2</td>
<td>comparison flag</td>
</tr>
<tr>
<td>SHIFT</td>
<td>R1, instr[3:2]</td>
<td>R3</td>
</tr>
<tr>
<td>Mask</td>
<td>R1, R2</td>
<td>R3</td>
</tr>
</tbody>
</table>

Based on the above analysis we can design the structure of the datapath component as shown in Fig. 1.
Next we make an analysis of the register usage according to the instruction set to see how the register file component should be designed. The source and destination of each register are listed in Table 2. Another important aspect about the registers is that, each register should be enabled if and only if it is needed during the next clock cycle. Therefore, we will need five enable signals for all of the registers, and the enabled signals should be supplied by the instruction decoder. The structure of the register file component is then shown in Fig. 2.

### Table 2. Register usage table.

<table>
<thead>
<tr>
<th>register name</th>
<th>source</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>memory or datapath</td>
<td>datapath</td>
</tr>
<tr>
<td>R2</td>
<td>memory</td>
<td>datapath</td>
</tr>
<tr>
<td>R3</td>
<td>datapath</td>
<td>nowhere</td>
</tr>
<tr>
<td>overflow flag register</td>
<td>datapath</td>
<td>nowhere</td>
</tr>
<tr>
<td>compare flag register</td>
<td>datapath</td>
<td>nowhere</td>
</tr>
</tbody>
</table>

### FIGURE 1. Structure of the datapath component.

![Diagram of the datapath component]

### FIGURE 2. Structure of the register file component.

![Diagram of the register file component]
Now let us put everything together. Fig. 3 shows the architecture of whole mini computer system.

FIGURE 3. System architecture of the mini load/store machine

4.0 Design of the Instruction Memory

In order to simplify the implementation, we use the ROM16x1 (Read-Only Memory) module from the Xilinx Library to realize the instruction memory. A ROM16x1 module has a 4-bit address, meaning that the depth of the storage is 16, and the output is a single bit signal. We need seven ROM16x1 instances to implement the 16x7 instruction memory. Given an address A3A2A1A0, the outputs of the seven ROM16x1 instances constitute an instruction. We use bus to represent the address and instruction signals.

FIGURE 4. ROM16x1 module from the xilinx library.

There are two ways to create a bus in Xilinx. Suppose we already created an empty schematic called instr_mem. Before you place anything on the canvas, click menu Tools->Create I/O markers. You will be prompted to type in the input/output signals, as shown in Fig. 5. However, the approach only works if the schematic is empty. Suppose we have placed 7 ROM16x1 instances on the schematic, and want to add new buses addr[3:0] and instr[6:0]. What should we do? Now here comes the trick. First draw a wire ( ) on the schematic, attach an I/O marker ( ) to it. Double click on the I/O marker to open the property window; in the name attribute row, type in addr[3:0]. You will see the bus I/O marker and the wire become thicker since it's now a bus signal. This method is shown in Fig. 6.
FIGURE 5. Create buses on an empty schematic.

FIGURE 6. Create buses on a non-empty schematic.

Now extend the bus line by dragging and stretching the little red square at the end of the wire. Click on “Add bus tap” icon ( ) to add bus taps. When selected, use ( ) to change the direction of the bus tap if necessary. Attach the taps to the bus and make proper connections to the ROM module, as shown in Fig. 7.

The last step to make the bus working is to name the nets connected to the taps properly. Click on “Add Net Name” icon ( ), in the ensuing “Add Net Name Options” windows, select “Pick up names of bus members by clicking on a bus net”, and then click on the bus wire on the canvas; the bus name will be picked up in the option window. Click on the net you want to name. You can cycle through the members of a bus signal. When it’s all done, the schematic should look like Fig. 8. Continue the process to name all the nets in the schematic including the output nets.

![FIGURE 8. Nets properly named.](image)

### 4.1 Instruction Memory Initialization

The instructions to be executed during simulation need to be stored in the ROM by initialization. Suppose we have a instruction sequence as shown in Table 3. Each instruction of the sequence is converted into its binary form. The first

<table>
<thead>
<tr>
<th>ID</th>
<th>Instruction</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>init</td>
<td>0000000</td>
</tr>
<tr>
<td>2</td>
<td>move1 0100</td>
<td>01100110</td>
</tr>
<tr>
<td>3</td>
<td>move2 0110</td>
<td>01100110</td>
</tr>
<tr>
<td>4</td>
<td>add</td>
<td>10000010</td>
</tr>
<tr>
<td>5–16</td>
<td>init</td>
<td>0000000</td>
</tr>
</tbody>
</table>

bits of all the 16 instructions are collectively stored in a single ROM16x1 module; if written in hexadecimal format, the initialization value is 0008. This initialization can be done by bringing up the property windows of the ROM16x1 module (double clicking), and typing in 0008 in the INIT row, as shown in Fig. 9. You can also make it visible on the canvas to aid future error checking.

The same procedure is carried out for the other six ROM16x1 modules (corresponding to column 2 ~ 7 of the binary encoding). The completed instruction memory component is shown in Fig. 10.

### 5.0 Design of the datapath component

See Fig. 1. Use ADD4, BRLSHFT4 and COMPM4 from the library to ease your implementation.

### 6.0 Design of the register file component

See Fig. 2. Make sure you use FDRE from the library, which is a D flip-flop with synchronous reset.
7.0 Design of the instruction decoder

The instruction decoder is the control center of the whole computer system. We choose to use VHDL program to realize it since it is much less regular than the rest of the system. To start, create a new source and select VHDL module in the new source window.

FIGURE 9. Initialization of the first ROM16x1 module.

FIGURE 10. The completed instruction memory component.

FIGURE 11. Create a new VHDL module.
In the next window, you can define the input and output ports of the instruction decoder. We can skip this step since we'll have to modify the VHDL code anyway. Fig. 12 shows an incomplete VHDL program of the instruction decoder. The I/O ports are defined between line I and II, and the syntax is rather self-explanatory. Between line III and IV we add process clause to define the behavior of the instruction decoder. Inside the parenthesis of the process keyword is the sensitivity list. In this case, we only have input signal instr in the list. Whenever there is an event (low-to-high or high-to-low transition) in instr, the process will be activated and executed. According to the value of instr certain if clause will be selected and outputs produced.

**FIGURE 12. An incomplete sample VHDL program.**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.USComponents.all;

entity instr_decoder is
    Port ( instr : in std_logic_vector(2 downto 0);
            clr : out std_logic;
            r1_en : out std_logic;
            r2_en : out std_logic);
end instr_decoder;

architecture Behavioral of instr_decoder is

begin
    process(instr)
    begin
        if (instr(2 downto 0) = "000") then
            clr <= '1';
            r1_en <= '0';
            r2_en <= '0';
        end if;
        if (instr(2 downto 0) = "001") then
            clr <= '0';
            r1_en <= '1';
            r2_en <= '0';
        end if;
        ....
    end process;
end Behavioral;
```

After the VHDL program is done, you can save it and generate a symbol from it for inference in the top design. To do so, select the VHDL program in Sources in Project windows, and double click on Create Schematic Symbol under Design Utilities in Processes for Source window. This step is shown in Fig. 13.
8.0 Additional Tips

1. Think in hardware! Work on individual components first; use Tools->Check Schematic to ensure correctness. Make each sub-component a symbol and refer them in higher level designs.

2. The ADD4 module from the library interprets the input and output as 2’s complement binary numbers, which is a signed number system. Overflow happens when two positive numbers add up to a negative number, or two negative numbers add up to a positive number. Suppose the two inputs to the adder are $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$, and the sum is $S_3S_2S_1S_0$. Use the following equation to interpret your overflow result:

$$Overflow = A_3B_3S_3 + A_3\overline{B_3}S_3$$