An Implementation of Picoblaze 8-bit Microcontroller in Verilog

Project Supervisor: Prof. Shashank K. Mehta

Abhinav Bhatele
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur,
Kanpur, INDIA 208 016
Email: bhatele@cse.iitk.ac.in

***Abstract*— The microcontroller which has been discussed here is a Constant(k) Coded Programmable State Machine (KCPSM) developed by Xilinx. It is more commonly referred to as Picoblaze and is a 8-bit fully embedded microcontroller. This means that it can be totally embedded into a device without any external support.**

PicoBlaze design supports up to 49 different 16-bit instructions, has 16 general-purpose 8-bit registers and 256 directly and indirectly addressable ports. I have realized the datapath of the microcontroller from its instruction-set description and partially available VHDL implementation. I have implemented the microcontroller in Verilog HDL with minor changes. This report talks about the realization of its datapath and the description of its various sub-modules in detail. There is a also description of the instruction set and various other design considerations which have been looked into during the implementation.

I. Introduction

Picoblaze[1] microcontroller is a 8-bit fully embedded machine. It has 8 16-bit registers and a program store for 256 instructions. The instruction size is 16-bits. The Arithmetic Logic Unit is 8-bit wide and has carry and zero indicator flags. It has 256 input and 256 output ports. It also has a 15-location CALL/ RETURN stack and has facility for reset and interrupt.

The PicoBlaze module can be totally embedded into the device and requires no external support. Any logic can be connected to the module meaning that any additional features can be added to provide ultimate flexibility. It is supported by a suite of development tools including the assembler which gives the corresponding Verilog HDL[2] code for a program.

November 26, 2004

A. Constant(k) Coded Machine

Let us see the reason for its nomenclature as a Constant(k) Coded Machine. It uses constant values for a variety of purposes. It is in many ways a state machine based on constants.

Constant values are specified for use in the following aspects of a program:

- Constant data value for use in an ALU operation
- Constant port address to access a specific piece of information or control logic external to the PicoBlaze module
- Constant address values for controlling the execution sequence of the program

The PicoBlaze instruction set coding is designed to allow constants to be specified within any instruction word. Hence, the use of a constant carries no additional overhead to the program size or its execution. This effectively extends the simple instruction set with a whole range of virtual instructions.

All instructions under all conditions execute over a constant time of two clock cycles. It also has constant program length of 256 instructions. All address values are specified as 8-bits contained within the instruction coding. The fixed memory size promotes a consistent level of performance from the module.

II. Functional Blocks

We now discuss the feature set of the microcontroller (Figure 01) in somewhat greater detail.

A. General-Purpose Registers

The feature set includes 16 general-purpose 8-bit registers, specified as s0 to sF (can be renamed in the assembler). All register operations are completely flexible, with no registers reserved for special tasks or given any priority over any other register. No accumulator exists as any register can be adopted for use as an accumulator.

B. Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) provides all the simple operations expected in an 8-bit processing unit. All operations are performed using an operand provided by any register. The result is returned to the same register. For operations requiring a second operand, a second register can be specified or a constant 8-bit value can be supplied. Bit-wise operators (LOAD, AND, OR, XOR) provide the ability to manipulate and test values. There is also a comprehensive Shift and Rotate group.

C. Flags Program Flow Control

The ALU operation results affect the ZERO and CARRY flags. This information determines the execution sequence of the program using conditional and non-conditional program

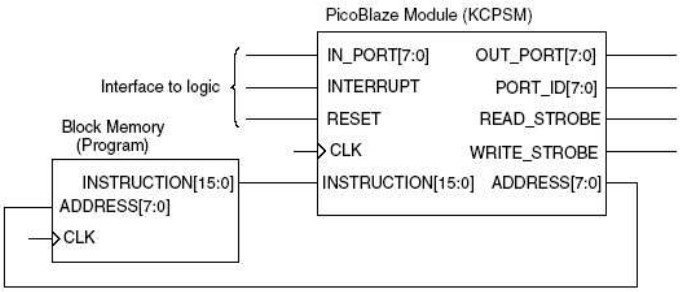

Fig. 1. Microcontroller Diagram

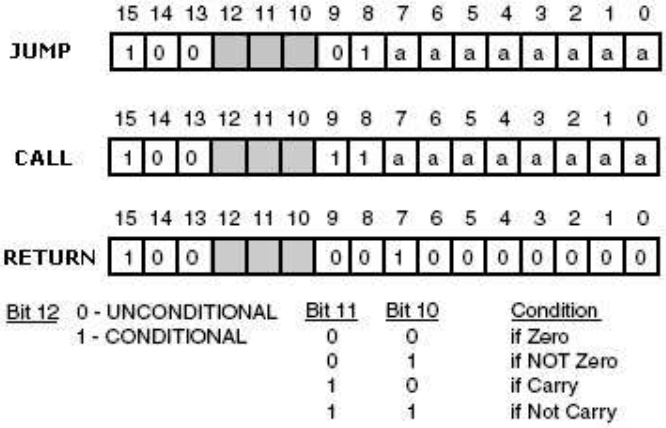

Fig. 2. Program Flow Instructions

flow control instructions. JUMP commands specify absolute addresses within the program space. CALL and RETURN commands provide subroutine facilities for commonly used sections of code.

D. Reset

The RESET input forces the processor back into the initial state. The program executes from address 00 and interrupts are disabled. The status flags and CALL/ RETURN stack are also reset. Note that the register contents are not affected.

E. Input/Output

The PicoBlaze module has 256 input ports and 256 output ports. An 8-bit address value provided on the PORT_ ID bus together with READ_ STROBE or WRITE_ STROBE signals indicates the accessed port. The port address can be either supplied in the program as an absolute value, or specified indirectly as the contents of any of the 16 registers. In the real sense the microcontroller just has one input and output port which contains the value and a port_ id which gives the address of the port where this is to be written. REst of it is a part of the user interface logic.

During an INPUT operation, the value provided at the input port is transferred into any of the 16 registers. An input operation is indicated by a READ_ STROBE output pulse. Although using this signal in the design input interface logic is not always vital, it indicates that data has been acquired by the PicoBlaze module.

During an OUTPUT operation, the contents of any of the 16 registers are transferred to the output port. A WRITE_ STROBE output pulse indicates an output operation. This strobe signal is used in the design output interface logic to ensure that only valid data is passed to external systems. Typically, WRITE_ STROBE is used as a clock enable or write enable signal.

F. Interrupt

The processor provides a single interrupt input signal. Using simple logic, multiple signals can be combined and applied to this one input signal. By default, the effect of the interrupt signal is disabled and is then under program control to be enabled and disabled as required.

An active interrupt forces the PicoBlaze macro to initiate a CALL FF (i.e., a subroutine call to the last program memory location) for the user to define a suitable course of action. Automatically, the interrupt process preserves the current ZERO and CARRY flag contents and disables any further interrupts. A special RETURNI command ensures that the end of an interrupt service routine restores the status of the flags and controls the enable of future interrupts.

III. INSTRUCTION SET

A. Program Control Group

1) JUMP: Under normal conditions, the program counter (PC) increments to point to the next instruction. The JUMP instruction (Figure 02) can be used to modify this sequence by specifying a new address.

However, the JUMP instruction can be conditional. A conditional JUMP is only performed if a test performed on either the ZERO flag or CARRY flag is valid. Each JUMP instruction must specify the 8-bit address as a two-digit hexadecimal value. The assembler supports labels to simplify this process.

2) CALL: The CALL instruction is similar in operation to the JUMP instruction. It modifies the normal program execution sequence by specifying a new address. The CALL instruction can also be conditional. In addition to supplying a new address, the CALL instruction also causes the current program counter (PC) value to be pushed onto the program counter stack.

The program counter stack supports a depth of 15 address values, enabling nested CALL sequences to a depth of 15 levels to be performed. Since the stack is also used during an interrupt operation, at least one of these levels should be reserved when interrupts are enabled. The stack is implemented as a separate cyclic buffer.

3) RETURN: The RETURN instruction is the complement to the CALL instruction. The RETURN instruction is also conditional. The new program counter (PC) value is formed internally by incrementing the last value on the program

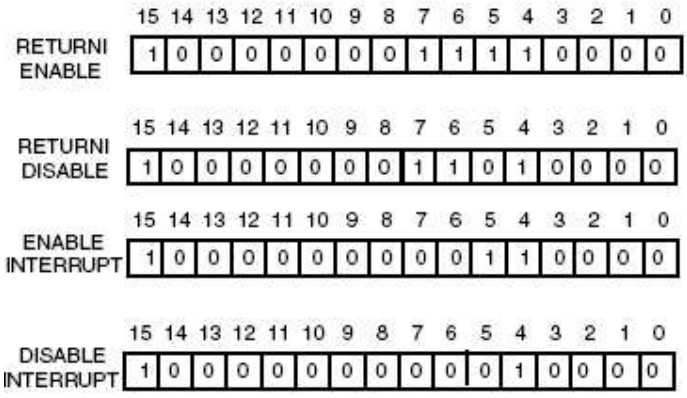

Fig. 3. Interrupt Group Instructions

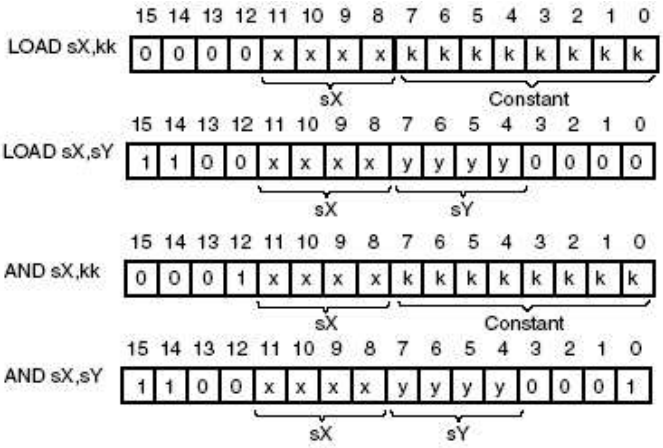

Fig. 4. LOAD and AND Instructions

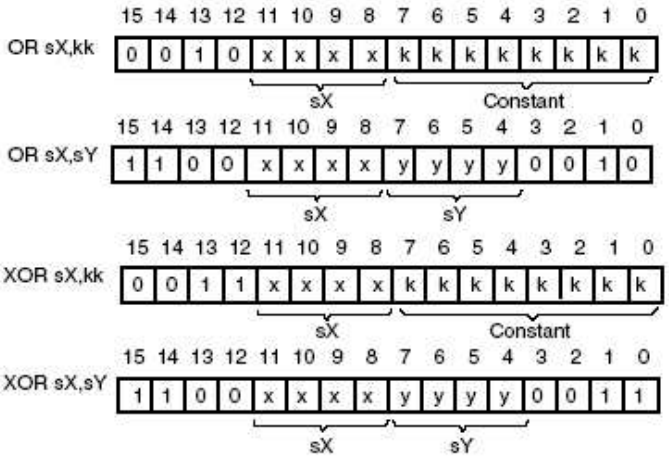

Fig. 5. OR and XOR Instructions

address stack, ensuring that the program executes the instruction following the CALL instruction which resulted in the subroutine.

B. Interrupt Group

1) RETURNI: The RETURNI instruction (Figure 03) is a special variation of the RETURN instruction. It concludes an interrupt service routine. The RETURNI is unconditional and always loads the program counter (PC) with the last address on the program counter stack. The address does not increment in this case, because the instruction at the address stored needs to be executed.

The RETURNI instruction restores the flags to the point of interrupt condition. It also determines the future ability of interrupts using ENABLE and DISABLE as an operand.

2) ENABLE INTERRUPT and DISABLE INTERRUPT: These instructions are used to set and reset the INTERRUPT ENABLE flag. Before using ENABLE INTERRUPT, a suitable interrupt routine must be associated with the interrupt address vector (FF).

C. Logical Group

1) LOAD: The LOAD instruction specifies the contents of any register. The new value is either a constant or the contents of any other register. Since the LOAD instruction does not affect the flags, it is used to reorder and assign register contents at any stage of the program execution.

Some implied virtual instructions are listed. LOAD s0,s0 - Loading any register with its own contents achieves nothing and hence is a NO OPERATION consuming two clock cycles. This may be used to form a delay in the program. LOAD sX,00 - Loading zero is the equivalent of a CLEAR register command.

Each LOAD instruction (Figure 04) must specify the first operand register as s followed by a single hexadecimal digit. The second operand must then specify a second register value in a similar way or specify an 8-bit constant using two hexadecimal digits. The assembler supports register naming and constant labels to simplify programming.

2) AND: The AND instruction performs a bit-wise logical AND operation between two operands. For example, 00001111 AND 00110011 produces the result 00000011. The first operand is any register and it is the register assigned the result of the operation. A second operand is also any register or an 8-bit constant value. Both ZERO and CARRY flags are affected by this operation.

3) OR: The OR (Figure 05) instruction performs a bit-wise logical OR operation between two operands. The first operand is any register. This register is assigned the result of this operation. A second operand is also any register or an 8-bit constant value. Flags are affected by the OR operation.

4) XOR: The XOR instruction performs a bit-wise logical XOR operation between two operands. For example, 00001111 XOR 00110011 produces the result 00111100. The first operand is any register and this register is assigned the result of the operation. A second operand is also any register

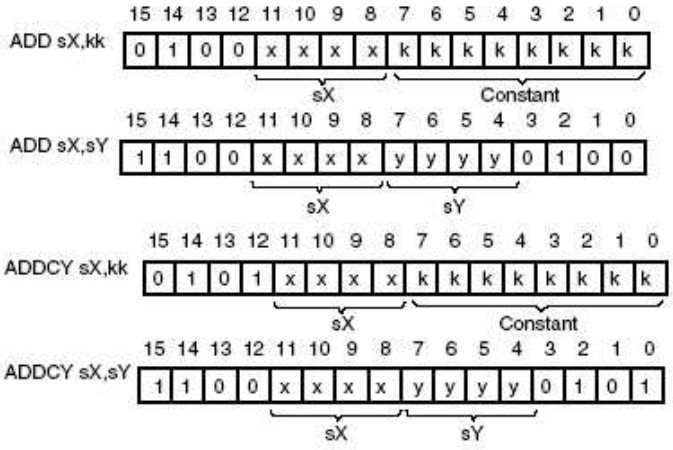

Fig. 6. ADD and ADDCY Instructions

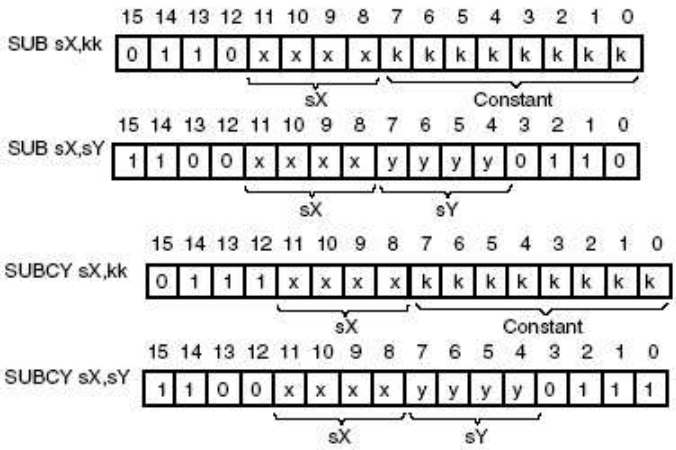

Fig. 7. SUB and SUBCY Instructions

or an 8-bit constant value.

D. Arithmetic Group

1) ADD: The ADD instruction performs an 8-bit addition of two operands without a carry. The first operand is any register and it is this register that is assigned the result of the operation. A second operand is also any register or an 8-bit constant value (Figure 06).

2) ADDCY: The ADDCY instruction performs an addition of two 8-bit operands together with the contents of the CARRY flag. The first operand is any register, and this register is assigned the result of the operation. A second operand is also any register, or an 8-bit constant value.

3) SUB: The SUB instruction performs an 8-bit subtraction of two operands without a carry. The first operand is any register and this register is assigned the result of the operation. The second operand is also any register or an 8-bit constant value (Figure 07).

4) SUBCY: The SUBCY instruction performs an 8-bit subtraction of two operands together with the contents of the CARRY flag. The first operand is any register and this register is assigned the result of the operation. The second operand is also any register, or an 8-bit constant value.

E. Shift and Rotate Group

1) SR0, SR1, SRX, SRA, RR: The shift and rotate right group all modify the contents of a single register (Figure 08). The result is also returned to the same register. The bit to be injected is decided depending by the second and third LSB bits of the instruction. All instructions in the group have an effect on the flags.

2) SL0, SL1, SLX, SLA, RL: The shift and rotate left group all modify the contents of a single register. All instructions in the group have an effect on the flags. The result is also returned to the same register. The bit to be injected is decided depending by the second and third LSB bits of the instruction.

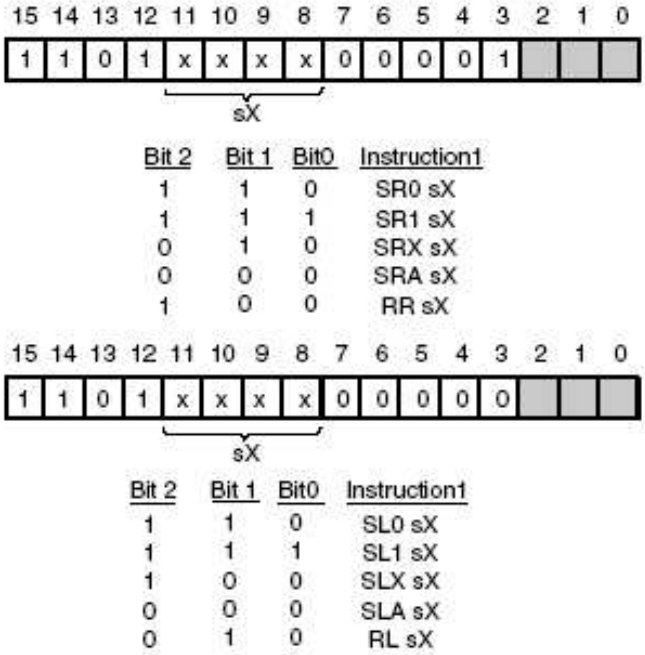

Fig. 8. Shift and Rotate Group Instructions

F. Input and Output Group

1) INPUT: The INPUT instruction (Figure 09) enables data values external to the PicoBlaze module to be transferred into any one of the internal registers The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of the any other register.

The user interface logic is required to decode the PORT_ID port address value and supply the correct data to the IN_PORT. The READ_STROBE is set during an input operation but is not vital for the interface logic to decode this strobe in most applications. However, it can be useful for determining

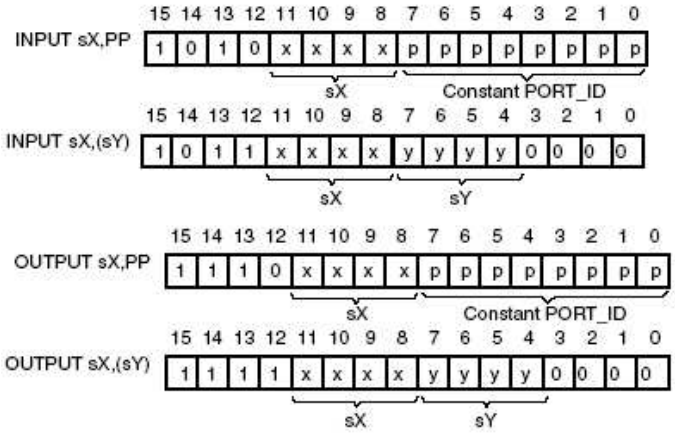

Fig. 9. Input and Output Instructions

when data has been read, such as when reading a FIFO buffer.

2) OUTPUT: The OUTPUT instruction enables the contents of any register to be transferred to logic external to the PicoBlaze module. The port address (in the range 00 to FF) is defined by a constant value, or indirectly as the contents of the any other register.

In this case also, the user interface logic is required to decode the PORT_ID port address value and capture the data provided or the OUT_PORT. The WRITE_STROBE is set during an output operation and should be used to clock enable the capture register (or write enable a RAM).

IV. VHDL Code and Datapath

An implementation of the microcontroller is available in VHDL. The code is divided into modules which do some specific functions. These modules are instantiated and used by the main module. The individual modules use pre-defined libraries like unisim for a variety of functions. The first attempt was to look at the modules and figure out their function. Let's see an example of the module which is a 3-bit bus 2 to 1 multiplexer.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library unisim;
use unisim.vcomponents.all;

entity ALU_control_mux2 is
  Port ( D1_bus: in
           std_logic_vector(2 downto 0);
         D0_bus: in
           std_logic_vector(2 downto 0);
         instruction15: in
           std_logic;
         Y_bus: out
           std_logic_vector(2 downto 0));
```

Fig. 10. ALU Control submodule

```
  end ALU_control_mux2;

architecture low_level_definition of
  ALU_control_mux2 is
begin
  bus_width_loop: for i in 0 to 2 generate
  attribute INIT : string;
  attribute INIT of mux_lut : label is "E4";
  begin
    mux_lut: LUT3
    generic map (INIT => X"E4")
    port map( I0=> instruction15,
          I1 => D0_bus(i),
          I2 => D1_bus(i),
          O => Y_bus(i) );
  end generate bus_width_loop;
end low_level_definition;
```

Though the description of the ports looks simple but the module uses a certain *LUT3* from the unisim library. Since the attempt at figuring out the work done by such LUTs and FDs failed so we changed our approach.

We first looked at the input and output ports of these modules. We then looked at the instantiations of these modules in the main module. The port mapping of signals and IO ports there finally helped us arrive at the data path.Let us look at an example of that. The module described above is instantiated in the main module as:

```
component ALU_control_mux2
  Port ( D1_bus : in
           std_logic_vector(2 downto 0);
         D0_bus : in
           std_logic_vector(2 downto 0);
         instruction15 : in
           std_logic;
         Y_bus : out
           std_logic_vector(2 downto 0));
  end component;
```

The port mapping for this module in the main module is as follows:

```
ALU_control_select: ALU_control_mux2
port map ( D1_bus => instruction(2 downto 0),
           D0_bus => instruction(14 downto 12),
           instruction15 => instruction(15),
           Y_bus => ALU_control );
```

Looking at the port mapping, we can say that the inputs

to this module are instruction bits 2 to 0, 14 to 12 and 15. The output is a 3-bit ALU_control. We thus arrive at block diagram (Figure 10) of this module. This way we found out the connections between different functional units and finally arrived at the datapath for the microcontroller.

V. Individual Modules

Once we could figure out the datapath, the next important task before us was to find out what each module does. We used the information about instruction set to find out the relation between the inputs and outputs of each module. Let us look at a few examples of how this was done.

A. Operand Select

This module takes the following inputs: instruction bits 15 to 12 and 7 to 0, sY_register and gives the second operand as output. We looked at the various instructions and found out the op-codes (most significant 4 bits) of the instructions which use a constant as a second operand and of others which uses the contents of a register as a second operand. This helps us arrive at the following values:

second operand = sY_register if op-code = 1100, 1011, 1111
second operand = instruction bits 7 to 0 if op-code = 0***, 1010, 1110
second operand = δ if op-code = 1101, 1000, 1001

B. Arithmetic Group

The function in the Arithmetic Group instructions is decided by the instruction bits 13 and 12 if instruction bit 15 is 0 and by instruction bits 1 and 0 otherwise. This group takes the two operands, a carry and ALU_control bits 1 and 0 as input. It gives the arithmetic result and carry as output. The function to be performed is decided by the two control bits. So we arrive at the following conclusion:

operation = ADD if ALU_control[1..0] = 00
= ADDCY if ALU_control[1..0] = 01
= SUB if ALU_control[1..0] = 10
= SUBCY if ALU_control[1..0] = 11

C. Logical Group

In an analogous way, the function in the Logical Group instructions is also decided by the instruction bits 13 and 12 if instruction bit 15 is 0 and by instruction bits 1 and 0 otherwise. This group takes the two operands and ALU_control bits 1 and 0 as input. It gives the arithmetic result and carry as output. The function to be performed is decided by the two control bits. So we arrive at the following conclusion:

operation = LOAD if ALU_control[1..0] = 00
= AND if ALU_control[1..0] = 01
= OR if ALU_control[1..0] = 10
= XOR if ALU_control[1..0] = 11

D. Shift and Rotate Group

Shift and Rotate Group takes the following inputs: the operand i.e. sX_register, carry and the instruction bits 3 to 0. Bit 3 decides whether to shift left or right (right if the bit is 1). Bits 2 and 1 decide the bit to be injected while shifting and bit 0 is used as a inject bit in one of the cases. For this module, we just need to decide the bit to be injected. Looking at the instruction set, we can say that:

inject-bit = carry if instruction[1..0] = 00
= sX_register[7] if instruction[1..0] = 01
= sX_register[0] if instruction[1..0] = 10
= instruction[0] if instruction[1..0] = 11

This is valid both for shifting to the left and to the right.

E. ALU Multiplexer

ALU Multiplexer is a 8-bit bus 4 to 1 multiplexer which takes in the following inputs: logical_result, arithmetic_result, shift_rotate_result, in_port, instruction bits 15 to 12 and ALU_control[2]. We look at the instruction set to find out which result to choose in case of a particular instruction. When the op-code is 1100, we also need to look at the least significant four bits of the instruction. What we find is that only instruction bit 2 is important to distinguish between logical and arithmetic instructions. So the results are:

ALU_result = in_port if op-code = 101*
ALU_result = shift_rotate_result if op-code = 1101
ALU_result = arithmetic_result if op-code = 01** or 1100 with ALU_control = 1
ALU_result = logical_result if op-code = 00** or 1100 with ALU_control = 0

F. ALU Control

ALU Control is the instantiation of ALU_control_mux2 which we have already seen. The inputs to this module are instruction bits 2 to 0, 14 to 12 and 15. The output is a 3-bit ALU_control. Since we have already seen what ALU_control does it is trivial to figure it out as a function of the input bits.

ALU_control[2] = instruction[2]
ALU_control[1..0] = instruction[13..12] if instruction[15] = 0
ALU_control[1..0] = instruction[1..0] if instruction[15] = 1

The function of other modules was also figured out in a similar way. The next and final step was to do the coding of these modules in Verilog. An example of the Verilog code for the module ALU Control is given here:

```
module alucontrol(aluc, d1, d2, in, clk);

input [2:0] d1;
input [2:0] d2;
input in;
input clk;
output [2:0] aluc;
reg [2:0] aluc;
```

```
always@(posedge clk)
begin
  if(in==1'b1)
  begin
    aluc=d1;
  end
  if(in==1'b0)
  begin
    aluc[2]=d1[2];
    aluc[1]=d2[1];
    aluc[0]=d2[0];
  end
end

endmodule
```

Here d1[2..0] stands instruction bits 2 to 0 and d2[2..0] stands instruction bits 14 to 12.

VI. Conclusion

The partial VHDL implementation and a complete description of the instruction set helped us arrive at the datapath.

We now have the codes of the various modules ready. We just need to link them up together. A module of the program ROM is used together with the processors main module. Together they define the complete microcontroller.

The latest assembler (an executable) which comes the Picoblaze distribution can be used to convert a program in the instruction set into a Verilog file. This can then be run on the processor which we have written!

Acknowledgment

The author would like to thank Prof. Shashank K. Mehta for his invaluable help and guidance in taking this project forward. He contributed a lot in deciphering the datapath through long sessions with the author. Without his support and encouragement the project could not have reached to completion.

References

[1] http://www.xilinx.com/products/design_resources/proc_central/grouping/picoblaze.htm

[2] Samir Palnitkar, *Verilog HDL, A Guide to Digital Design and Synthesis*

[3] http://www.xilinx.com/bvdocs/appnotes/xapp387.pdf