Design of an 8-bit RISC Microprocessor

Final Project Report
December 9, 1999
E157

Kavish Shah and Brent Hinderberger

Abstract:

Our project is to design an 8-bit RISC microprocessor implementing a subset of the MIPS instruction set. This project is an extension of the designing of a 4-bit RISC microprocessor that E157 had in previous years. We have added memory to our design, whereas in the past, it was not required. We are using the Xilinx Spartan XCS10 FPGA to store all of the logic necessary to make a microprocessor. Also, all memory and registers are on-board the FPGA. In addition to the XCS10, we are using the Motorola HC11EVB as input to the microprocessor.

Introduction

This project is to design and implement an 8-bit microprocessor implementing a subset of the MIPS instruction set. Most computers today are based off systems of similar design, so we decided to build our processor to have a better understanding of the microprocessor architecture. While our design does not have any direct real-world application, since it does not contain many features common to current processors, we hope this project will enlighten us as to the foundations of microprocessor design.

Our design is implemented on the Xilinx Spartan XCS10 FPGA using Verilog code to design the logic for the microprocessor. We have chosen the Verilog HDL instead of logic gate schematics because we are more familiar with it, and it allows us to represent very complex logic using a higher-level language. The Motorola HC11EVB is used as the source of input instructions for the microprocessor.

In addition to the microprocessor and EVB interface, we have designed a dual 7-segment display as output devices. This is used to check the current status of any register on-board the FPGA, which allows us to debug the operations of the circuit and verify the output of any program created to run on our microprocessor.

Schematics

Using Verilog has allowed us to design the microprocessor in a modular fashion. Our microprocessor layout is broken into several modules. The top-level module implements the transfer of information between the different registers, as well as the functional modules of the microprocessor. We then have the control module that acts as a finite state machine that outputs an appropriate control code decoded from the instruction opcode. Our microprocessor contains on-board memory along with the top-level registers used for our design. The ALU module acts as the microprocessor's arithmetic logic unit to perform the instructions that we have implemented. Finally, the 7-segment display module is used to mux the display of two 4-bit numbers onto a single 7-segment display.

It has been necessary to program our microprocessor into two separate FPGA units because of the limited amount of logic blocks on the Spartan FPGA's. Our design uses the secondary FPGA to run the 7-segment display modules and the ALU module.

Our implementation of the 8-bit RISC microprocessor uses two modes of operation. In the "download" mode, 8-bits of the EVB are sent to the FPGA to store the instruction onto on-board memory. In the "run" mode, four bits of the EVB are used to select which register information will be outputted to the 7-segment display.

Microcontroller Design

Our design uses the 8-bits of port B on the Motorola HC11EVB. To use port B of the EVB as an input device it is only necessary to write the desired input value to the port B register at address $1004 using the memory modify (mm) command in Buffalo. As previously mentioned, during the “download” mode, all 8 bits are used to transmit the program word by word into the on-board memory of the microprocessor. In the “run” mode all 8 bits of port B can be written to but only the four least significant bits are used as input data.

FPGA Design

Being an 8-bit microprocessor, each word in memory is 8 bits wide. A program loaded into the on-board memory consists of a series of commands, which are stored in memory in a command field. A command field consists of four words of data and looks as below:

Op-Code	Rs	Rt	Rd

The op-code word is the actual instruction to be executed, while the words Rs, Rt, Rd each have different functions depending on the instruction to be executed. We have implemented the following commands:

Name	Mnemonic	Op-Code	Operation
Load Word	lw	0	Loads word from memory address A+Rd into reg_block[Rt]
Store Word	sw	1	Stores the contents of reg_block[Rt] memory at the address A+Rd.
Add	add	2	Stores the result of binary addition of register A and reg_block[Rt] into reg_block[Rd].
Subtract	sub	3	Stores the result of binary subtraction of register A and reg_block[Rt] into reg_block[Rd].
And	and	4	Stores the result of the bitwise binary AND of register A and reg_block[Rt] into reg_block[Rd].
Or	or	5	Stores the result of the bitwise binary OR of register A and reg_block[Rt] into reg_block[Rd].
Set on Less Than	slt	6	Stores the value '1' into reg_block[Rd] if register A is less than reg_block[Rt]. Otherwise, it stores the value '0'
Jump	j	7	Change PC to the address Rd
Branch if Equal	beq	8	Move PC to the address PC + Rd if reg_block[Rt] equals register A

The register A contains the value that is in the on-board register with the address given in Rs. Registers Rt and Rd contain either addresses for registers used in the opcode instruction or offsets for branch instructions.

The primary FPGA also contains the register PC, which indicates the current address in on-board memory for execution. We also have a block of four 8-bit on-board general-purpose registers, known as reg_block[X] (X = 0..3).

In the top-level module, there are sequential logic blocks for assigning values for memory management units, the transfer of data from main memory to appropriate registers, assignment of inputs for the ALU, and the assignment of values to be output on the 7-segment display.

On the secondary FPGA, the top-level module sends information obtained from the primary FPGA to the ALU and 7-segment display modules. The 7-segement-display module displays a value on the left side during the positive part of clock and displays another value on the right side of the display during the negative part of the clock. The ALU module computes our implemented instructions.

The following procedure is used in running a program on the microprocessor. If the program is not currently on the on-board memory it must be downloaded by entering the program word by word in the manner outlined below. Once the program has been downloaded the microprocessor should be reset by using the reset input (not the reset button on the circuit board) which will set all the registers in the microprocessor to their start states but keep the program in memory. Next one puts the microprocessor in run mode and clock through the commands. The running of the microprocessor in run mode is also described below.

To understand the workings of the microprocessor it is best to look at the timing diagram for the microprocessor (see Microprocessor Timing Diagram). Each command in a program is implemented in one command cycle, taking a total of ten clock cycles to complete. Each command cycle is broken down into five sub-cycles, each taking two clock cycles to implement.

The timing diagram also shows four clocks that each have a period of one command sub-cycle. These clocks sequence the different actions that the microprocessor performs during a sub-cycle.

The microprocessor downloads a program when the input bit mode is low. While in download mode a word of the program is loaded into memory during every sub-cycle. On the rise of clk2 a word is loaded into the mem_wr register and the Addr register is set to the address of memory to which the word will be written. On the rise of clk3 the value in the mem_wr register is written to memory through the logiblox memory module. The words are loaded in order starting at memory address 00, and downloading stops when address 1F is reached since there are only 32 words in memory. A 6 bit index, that is incremented on the rise of clk4, is used to keep track of the address that each word should be written to.

In run mode each command takes five sub-cycles to complete. At the start of running a program the microprocessor should have all registers set to their default values of zero, otherwise the start state of the program may be indeterminable and cause the program to run improperly. During the First sub-cycle, known as Fetch1, the register IR is loaded with the op-code from the command field and PC is incremented. In Fetch2 the Rs word is loaded in the register Rd and PC is incremented. In Fetch3 the word Rt is loaded into the register Rt, the register A is loaded with the value contained in the general-purpose register with the address that is stored in register Rd (the address comes from the Rs word in the command field), and PC is incremented. In Fetch 4 register Rd is loaded with the word Rd and PC is incremented by one. In the Instruction sub-cycle the actions that take place depend on the instruction stored in IR and can be found in the table above.

The sequencing of a sub-cycle in run mode is similar to the one followed in download mode. During the rise of Clk1 a control code is determined by the control module finite state machine (see Schematic for Control Module Finite State Machine) which determines the actions that will be taken during each sub-cycle of the command cycle. On the rise of Clk2 all required on-board memory interfacing is done. To read from or write to memory the read/write control bit for the on-board memory must be set to the appropriate value (0 for read only, 1 for write access), the address of the memory must be set, and if memory is being written to the word to be written must be stored in the mem_wr register. On the rise of clk3 the logiblox memory module implements the memory interface that has been specified and places the word at the address stored in Addr register in the register connected to mem_rd. On the rise of clk4 the appropriate register is loaded with the value designated by the control code.

The microprocessor runs off the clock on the PC board built for the FPGA. While the microprocessor is able to run off of the oscillator at speeds of at least 1MHz, it is more practical, for demonstration purposes, to run it off of the manual clock connected using the jumper J3. The reason a real clock is not practical is due to the fact that we have not implemented a “stop” command, which means the program that has been downloaded will run continuously until the clock is disconnected. Since it is impossible to predict at which command, or even sub-command, the program will stop, it is impossible to tell what the end state of the registers should be. In any case, The timing analysis given above works for understanding the workings of the microprocessor.

Results

Our design has been implemented with a 32-word memory space, allowing for programs of up to eight commands to be run. We have implemented a program that uses all instructions correctly, so our design is sound. At the current time, the microprocessor has only been run through a manual clock.

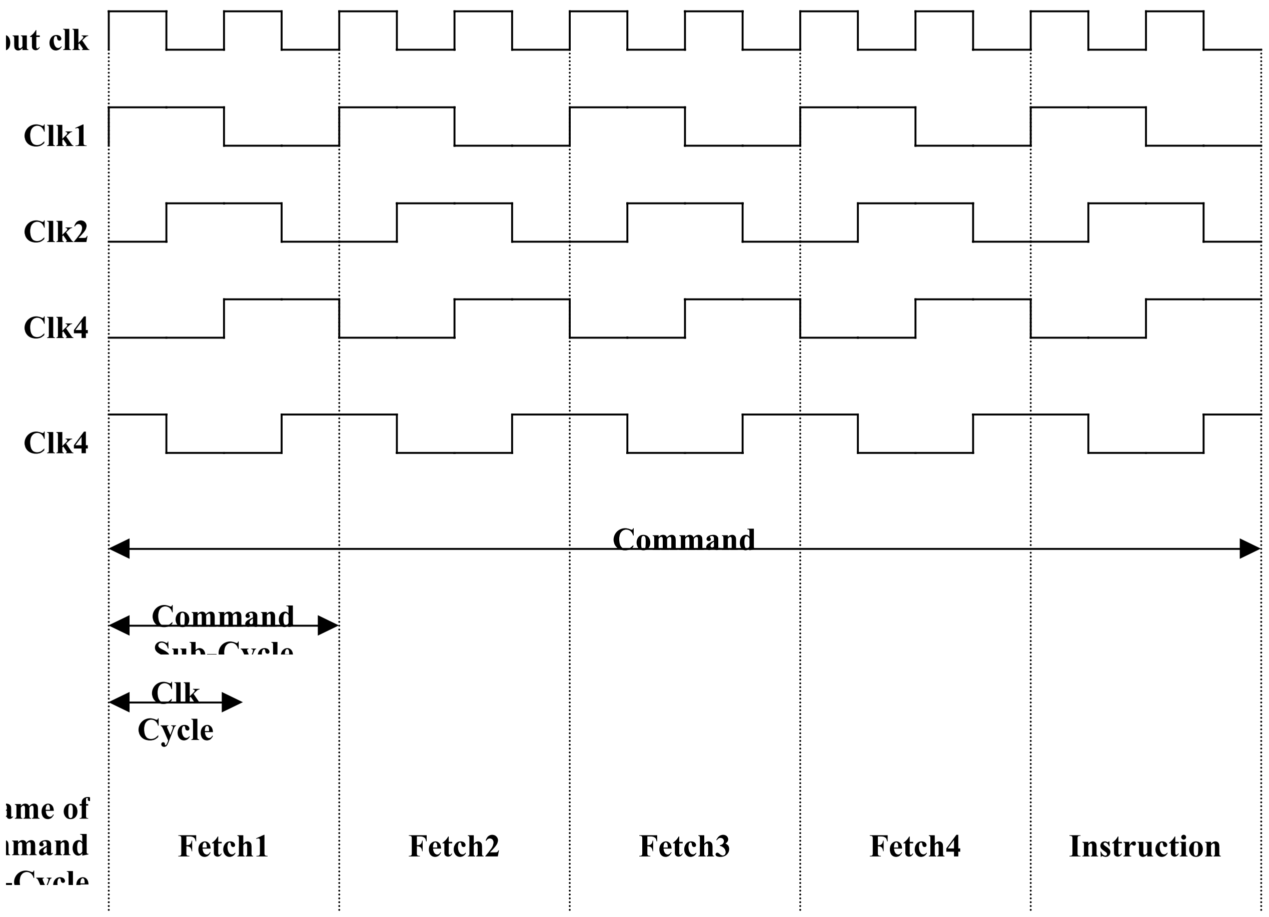
With a functional base microprocessor built it is now possible to focus on extending the current design. Some ideas we have had for extending the functionality of this design follow. One idea is to program the HC11EVB to download the program to the on-board memory in real time by setting up a master slave protocol between the microprocessor and the HC11EVB. Other commands would also be useful, including the much needed stop command which would allow one to run the processor reliably in real-time (the only other alternative to a stop command is jump command that jumps to itself, though infinite loops are never fun and would not work with programs that run sub-processes).

The main limitation on the project has been the amount of logic that can be put on one of the Spartan FPGAs. The Current design uses two FPGAs, as discussed above, which has led to a massive wiring project and used up nearly all the available input/output pins for the two FPGAs. The primary FPGA is currently running at about 94% to 100% full with a 32 word memory and only four general purpose registers so extending the design may be very difficult. Trying to extend the abilities of the microprocessor by using the secondary FPGA will be extremely difficult since there are so few pins left over communicating between the two FPGAs. To combat these problems the best approach may be to take the design and make break it down into even smaller modules, which may allow for a more efficient design and one that can be spread out to more processors with fewer pins needed.

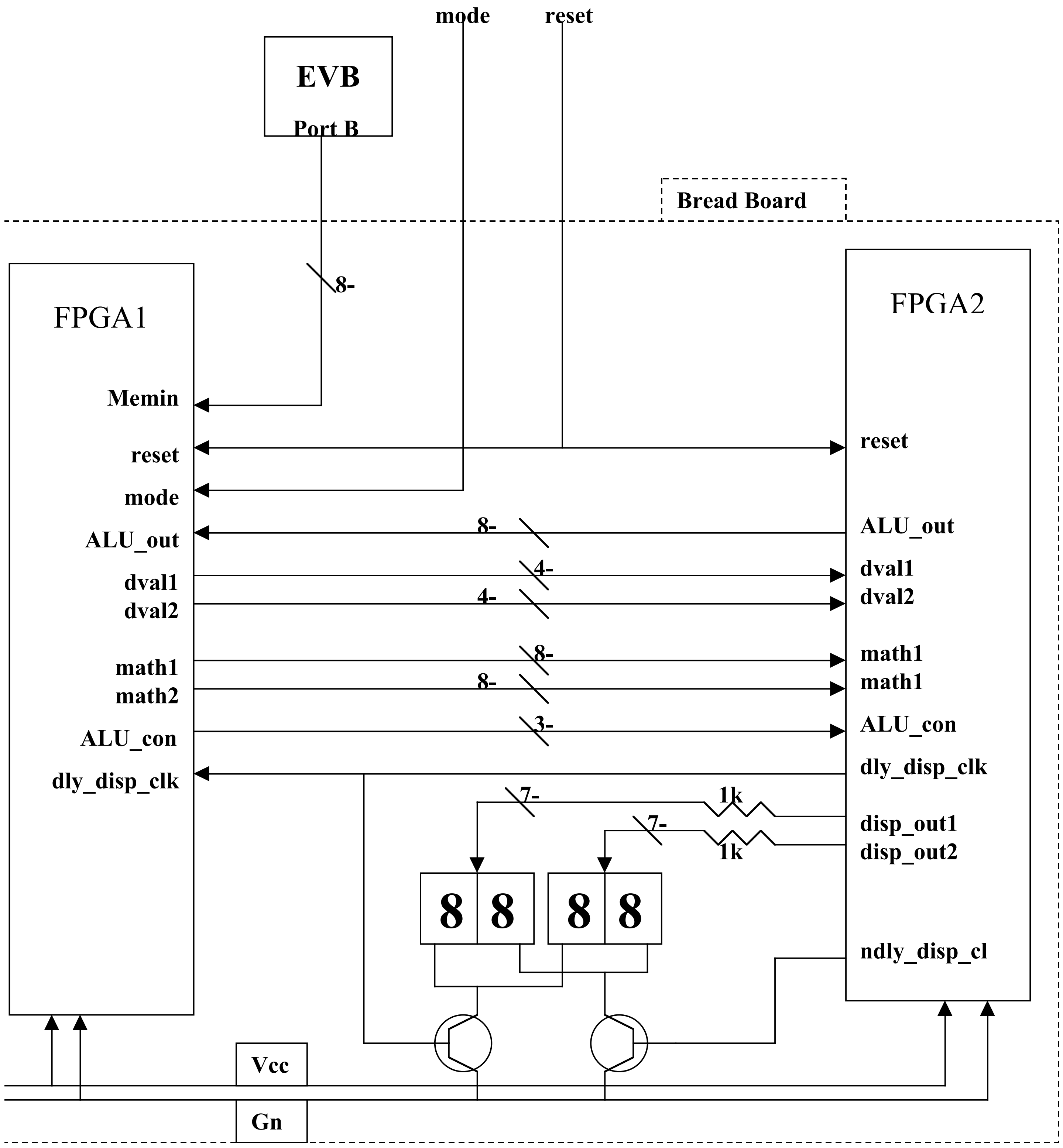
References

[1] Harvey Mudd College, Engineering Department. E157 lab manual, lab #4. 1998

Microprocessor Timing Diagram



Bread Board Schematic: FPGA-FPGA and HC11 EVB Interface


mode
reset
EVB
Port B
Bread Board
8-
FPGA1
FPGA2
Memin
reset
mode
ALU_out
dval1
dval2
math1
math2
ALU_con
dly_disp_clk
reset
ALU_out
dval1
dval2
math1
math1
ALU_con
dly_disp_clk
disp_out1
disp_out2
ndly_disp_cl
8-
4-
4-
8-
8-
3-
7-
7-
1k
1k
8 8 8 8
Vcc
Gn