ENEL353 Computer Hardware Engineering I

Digital Logic Design Project

October 18, 2010

Forrest McKerchar (frm25), Henry Jenkins (hvj10), Joel Koh (jmk35), Sasha Wang (xhw11), Tracy Jackson (tnj14), Wim Looman (wgl18)

Preface

In the general the group worked in pairs while working on the project. With the use of Subversion the code was shared among the group. The numbers in Table 1 below have been calculated via the use of Subversion's blame command and personal opinions. The full Subversion log can be found on gforge[1].

Section	Authors	Coding Contribution (%)	Writing Contribution (%)
CPU	Henry Jenkins	50	16.6
	Joel Koh	40	16.6
MMU	Wim Looman	98 + 10(from CPU)	16.6
	Forrest McKerchar	2	16.6
I/O	Sasha Wang	50	16.6
	Tracy Jackson	50	16.6

Table 1: Group Contributions

[1]See http://gforge.elec.canterbury.ac.nz

Abstract

This project aims to use VHDL to develop a digital logic design of a micro-controller. The design consists of three main components: CPU, MMU and I/O. The project was done by a team of six people - the team was divided evenly into three sub-groups, with each sub-group working mainly on one of the three components. The project's design process consisted of several steps: becoming familiar with the design problem, this included reading background information and references; choosing design decisions according to the specifications, writing the VHDL code, and testing. This report contains a detailed description of this process, in addition to technically describing the design and operation of the CPU, MMU and I/O subsystems, and also the buses that are responsible for communication between the components.

Table of Contents

Introduction

Modern computer systems consist of hardware system and software system. The hardware system consists of microprocessor system and peripheral components. This project focuses on developing the microprocessor system of the computer hardware system which includes a central processing unit (CPU), memory management unit (MMU), input/output (I/O) unit and the bus.

In this project, a CPU, MMU and I/O were designed using VHDL to simulate a computer hardware system. Each unit is connected to number of buses, allowing inter-module communication. Each unit is the final result is intended for implementation on a field-programmable grid array (FPGA). An overview of the structure of the design is shown in the Figure 1.

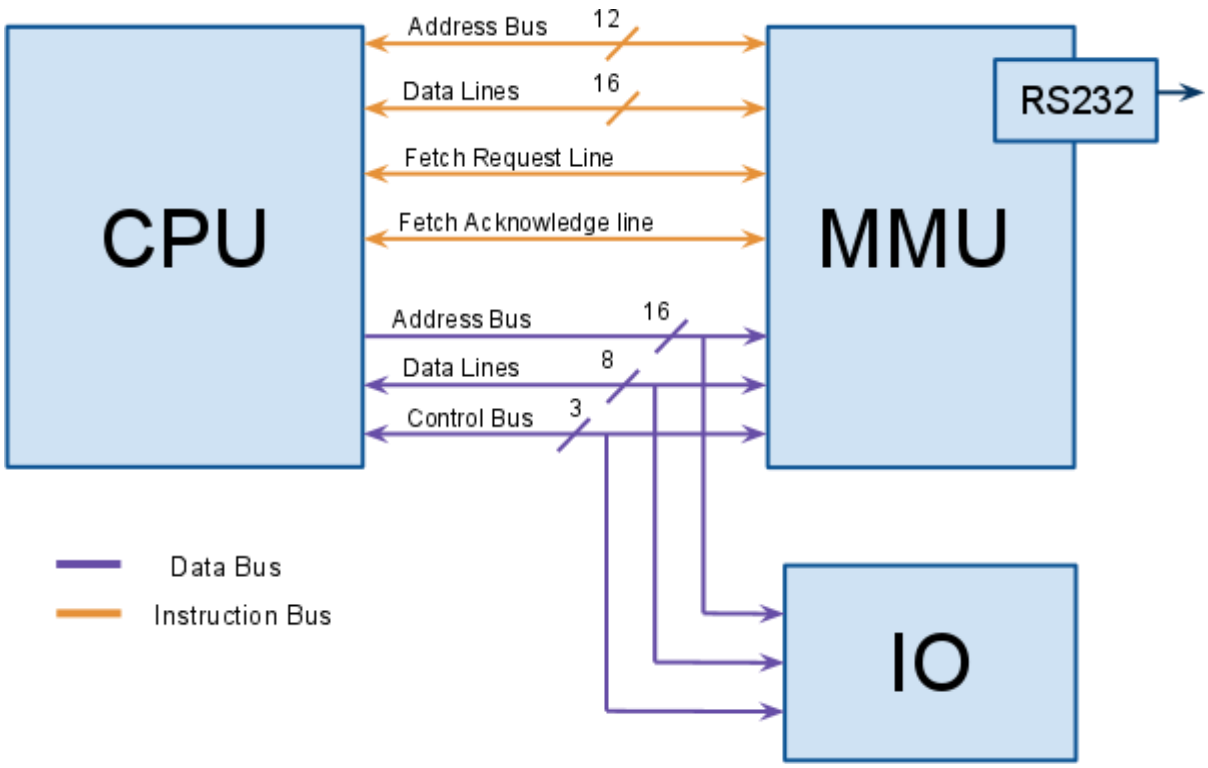


Figure 1: Overall system datapath

Architectural design

Buses

Instruction Bus

The instruction bus bi-directionally connects the CPU and the MMU carrying a 12 bit address bus, 16 bit data bus, one bit fetch request and a one bit fetch acknowledge. The fetch request is pulled low to request data and is pulled high to acknowledge the data, while the fetch acknowledge is pulled low when the data is loaded and high whenever the fetch request line goes low.

Data Bus

The data bus is connected from the CPU to both the MMU and the I/O, it carries a 16 bit address bus, 8 data lines and a 3 bit control bus. Both the data lines and the control bus are bi-directional, while the address bus is uni-directional from the CPU to either the MMU or the I/O, which is distinguished by the least significant bit.

The address bus indicates the destination of the data that the CPU is sending out. An address which has its least significant bit as a zero, indicates the data is going to the I/O, else it is a one indicating the data is going to the MMU. Therefore the I/O and MMU components both have half of the total 216 (16 bit) addresses available.

The 3-bit control bus carries three separate lines; the read/!write (R/!W) line, the bus request line and the bus acknowledge line. When idle, the bus request line is pulled high and the bus acknowledge line is set to high impedance. Before using the data lines, the CPU pulls the bus request low and sets the R/!W line to the corresponding mode, zero for write and one for read. The bus acknowledgement line is pulled low when the MMU or the I/O has completed reading from or writing to the data lines. After which he CPU clears the bus request line and the MMU or I/O return the bus acknowledgement line to high impedance again. All the control bus changes for interaction between the CPU and MMU or I/O for reading can be seen in Table 2, and for writing can be seen in Table 3.

	R/!W	**Request**	**Acknowledge**
IDLE CPU clears R/!W and sets address then sets request	**X**	**1**	**1**
CHECK ADDRESS If correct address, continue, else return to **IDLE**	**1**	**0**	**1**

WRITE DATA MMU/IO writes the data then set acknowledge	**1**	**0**	**1**
WAIT CLEAR CPU clears request MMU/IO waits for request to clear then clears acknowledge	**1**	**0**	**0**

Table 2: Changes on control bus when CPU is reading, X implies 'do not care'

	R/!W	**Request**	**Acknowledge**
IDLE CPU clears R/!W and sets address and data then sets request	**X**	**1**	**1**
CHECK ADDRESS If correct address, continue, else return to **IDLE**	**0**	**0**	**1**
WRITE DATA MMU/IO reads the data then set acknowledge	**0**	**0**	**1**
WAIT CLEAR CPU clears request MMU/IO waits for request to clear then clears acknowledge	**0**	**0**	**0**

Table 3: Changes on control buss when CPU is writing, X implies 'do not care'

Central Processing Unit (CPU)

The CPU fetches an instruction from memory, decodes it, setting all the control lines, then executes the instruction. To make both the coding and the design of the CPU the whole design was broken down into several entities. As shown in Fig. 2, the main data path is from the GPR, through the ALU then back to the AL via the common data bus. Depending on the different instruction the ALU is configured to perform a different operation. The Control unit takes care of setting all the control lines. These control lines are determined by the current instruction and the values in the status register. Because the CPU is constructed out of several smaller entities, the file cpu.vhd is there to make sure each component is instantiated in the hardware. The cpu.vhd file has no logic in it, just port maps to connect all the signals.

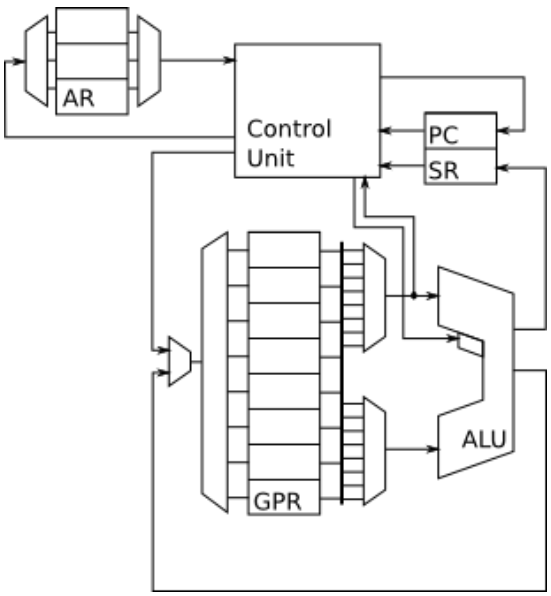
Control Unit

The control unit controls all the data flow within the CPU by setting the control lines shown in Fig. 3. The control unit runs in a fetch, decode and execute cycle. In the fetch stage, the control unit obtains an instruction from memory in the MMU, pointed to by the program counter, in the form of an opcode. The program counter is then incremented to get ready for the next cycle.

The decode stage is when most of the data processing occurs. Depending on the opcode, the control unit will control the multiplexers and demultiplexers to select the different registers required to be written to and read from. The control unit will process the necessary data from appropriate registers or from memory. Arithmetic and logical functions, however, are passed on to the ALU to calculate, after the control lines have been set. Instructions that require registers or memory to be written to will only be marked for writing in the decode stage.

In the execute stage, registers marked for writing will be enabled, and data to be written to memory will be transferred. The registers are only enabled for writing for one clock cycle to prevent loops in the data path from constantly overriding the registers.

The control unit also supports a reset signal that resets the program counter and status register when low, corresponding to power off.



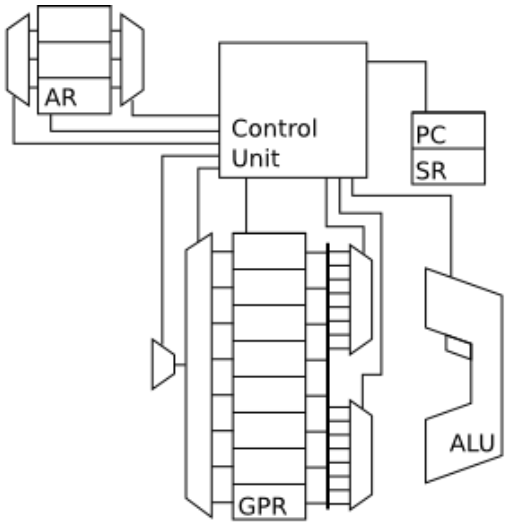


Figure 2: Data flow within CPU

Figure 3: CPU with control lines only

Arithmetic Logic Unit (ALU)

The ALU is a completely asynchronous logic entity that takes two 8 bit vectors, a four bit control line and the carry bit. The four control lines set which function is to be applied to the two 8 bit vectors. The ALU makes use of an 8 bit full adder to calculate all the required outputs. This full adder is composed of an 8 bit ripple adder. Only the ALU functions write to the status register so the input to the status register is driven directly by the ALU.

Registers

The three sets of registers, the general purpose registers, address registers and program counter/status register are all built using conditional statements that only updates the contents on the clocks rising edge. The address register also has the ability to update only 8 bits at a time as required by the specifications. All the registers are synchronous with the exception of the reset. The reset is active low so when the power is turning on the registers are reset. The reset is also able to be used to reset the program counter back to the zero address. This would allow an external reset pin or button to hard reset the whole microprocessor to be reset without having to reprogram the FPGA.

Testing

Testing of the CPU components was carried out using test benches. For each of the components a test bench file was created. These files were designed and built to test as many of the extreme cases as possible. In most of the cases, the test benches would define the inputs, wait for the clock to cycle, set some other control lines then expect an output from the module. This method worked well and brought out many bugs in our code.

Using test benches also made checking the logic easy. For most of the entities one member of the team would program the actual entity how they thought it should work, then the other team member would write the test bench. This if there was a misunderstanding of the logic, we could check it after the test bench had failed.

Memory Management Unit (MMU)

The MMU is responsible for managing all the CPU's requests for access to the instruction and data memory. The actual memory in this design is stored on a personal computer, connected to the FPGA board via an RS232 serial link. This configuration requires the MMU to interface to the RS232 link, in addition to the internal buses. Low-level interfacing to the serial link is achieved via the use of a supplied minimal UART core, originally obtained from OpenCores.org.

Design

The MMU essentially consists of bus mux/demuxers, which are controlled by a control unit. The control unit is formed from a number of finite state machines

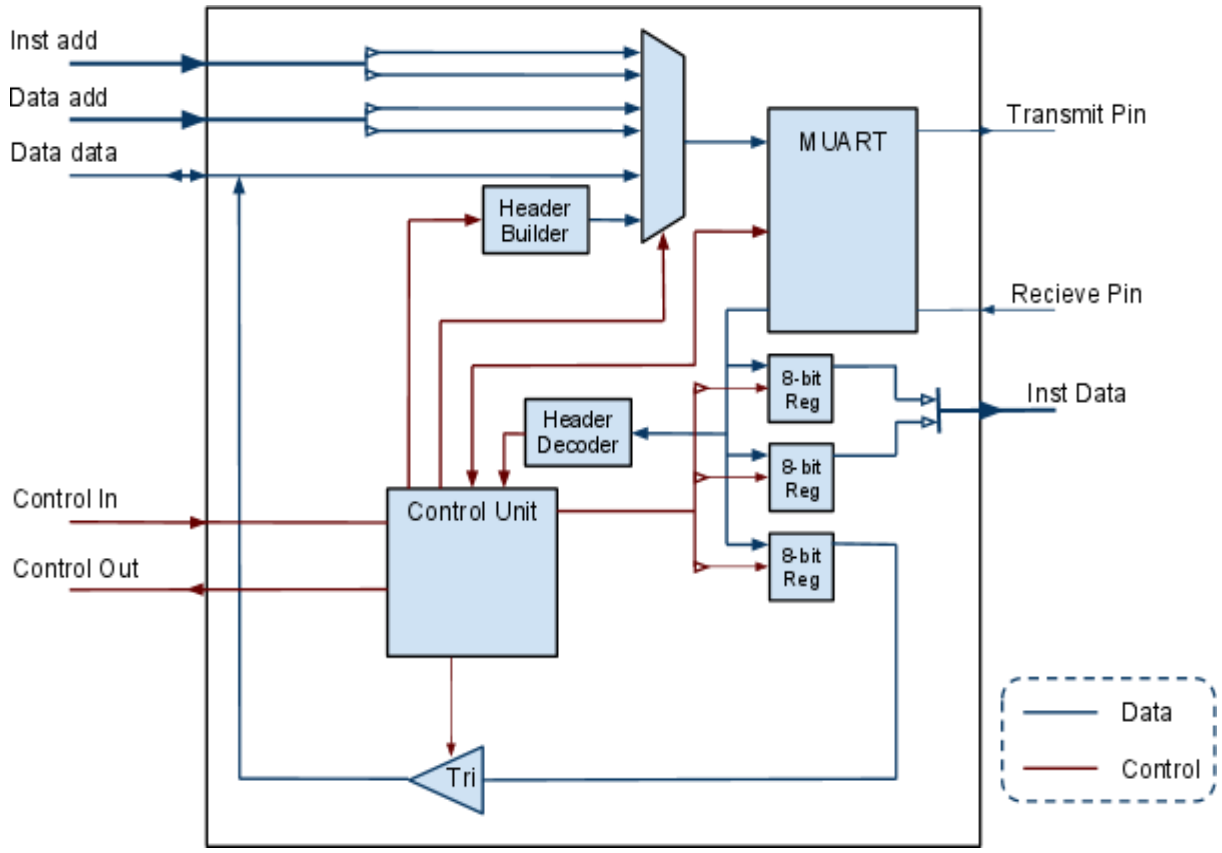


Figure 4: Data and Control paths of the MMU.

Control Unit

The control unit was split into two separate parts, the instruction control unit and the data control unit. This was mainly done to reduce the control unit file size and logic, but did mean there was some duplication of code. Each of these control units consists of 4 separate Finite State Machines (FSM), arranged in a simple hierarchy:

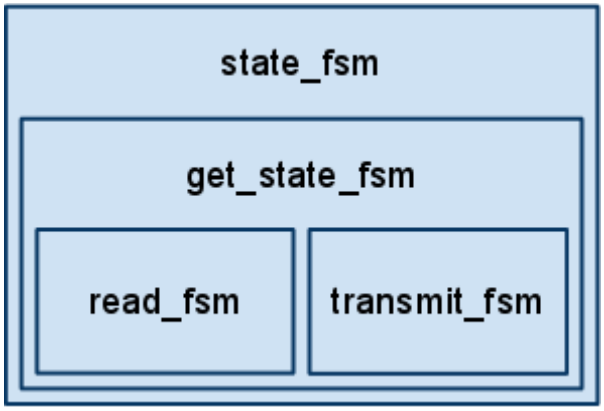


Figure 5: MMU Control Unit state machines

The lower level FSMs are activated during specific states of the higher level FSMs - the higher levels wait for the lower levels to reach their finished state, then move on to their own next state.

The highest level FSM (see Figures 10 and 11), state, is required to determine when the bus is activated, and to start the get_state FSM. Once the get_state FSM returns, then the state FSM will set the correct ack signal to low, to inform the CPU that the data is available. After this, it waits till the CPU withdraws its data request, and then sets the ack signal back to its idle state.

The get_state FSM (see Figures 12 and 13) simply encodes the order of sending and fetching data bytes. For the instruction control unit this is a simple linear state machine, while for the data control unit it has one choice, based off whether it is receiving or transmitting the data.

In the individual states of the get_state FSM, the read and transmit FSMs (see Figures 14 and 15) are activated, so that they receive or transmit the specific data currently in need of processing.

Testing

There were two test benches developed to test the MMU. One tested purely the instruction bus, whilst the other tested the data bus in conjunction with the IO unit.

- ### Instruction bus

The test bench developed to test the instruction bus (found in mmu/mmu_tb.vhd) creates and links the MMU to an extra MUART, which simulates the computer providing the memory. The address bus and request line are driven by the test bench and the data being transferred between the MMU and the MUART is then analysed and compared to what ideally should be being transferred. One full memory request, and part of a subsequent one, can be seen below: the test bench sets the inst_add bus, then pulls inst_req low. This initiates the memory transfer and causes first the header, then the low byte of the address and finally the high byte of the address to be written to the MUART, this can be seen below in

Figure 6. The fake computer then responds with a header, address bytes and data bytes, and the output of the MMU is compared to the expected data.

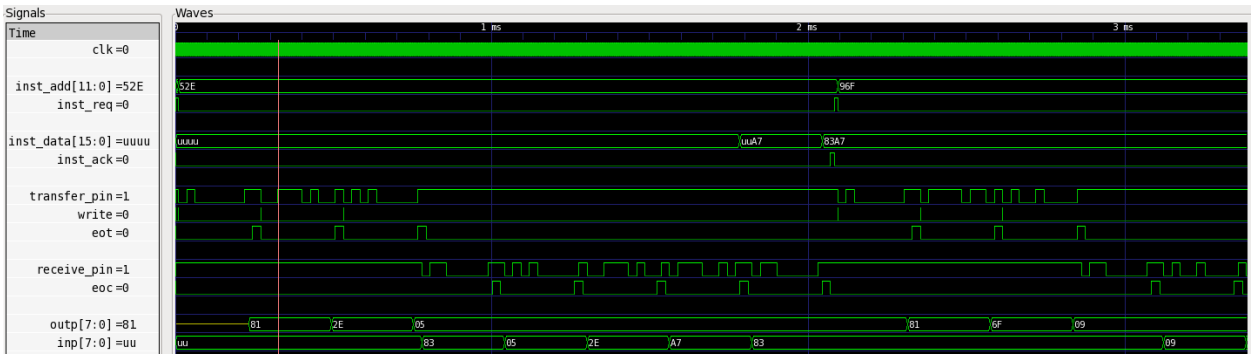


Figure 6: Instruction bus testing waveforms

- **Data bus**

The data bus test bench was almost identical. Its only difference was that it instantiated a copy of the IO unit as well, and had updated code to handle communication with the IO. When this was run with the IO unit deactivated it worked, however there were bus contention issues with the data bus when both the MMU and IO units were active.

Input and Output I/O

The I/O component is responsible for displaying the CPU's output and takes user requests for the CPU. The design of the I/O component uses 2 push buttons and 8 LEDs and consists of 5 entities: I/O, led, switch, switch register and switch debounce. The connection of these entities can be seen in Figure 7, where the I/O component is only connected to the CPU not the MMU.

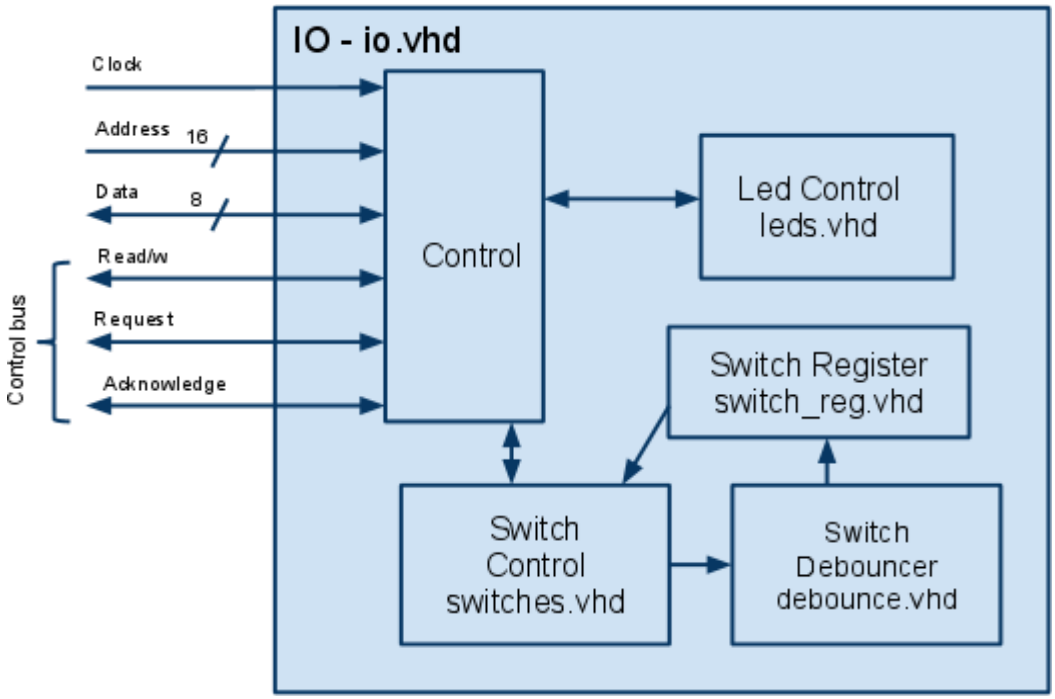


Figure 7: An overview of the I/O component's structure

Switch Control

When an input push button is pressed, a logic high is generated on the associated FPGA pin. This signal first goes through the debouncer and then is passed to the CPU on request.

Debouncer

A push button debouncer is implemented in the switch control design. When a push button is pressed, the signal does not change from one voltage to another cleanly[1], Figure 8. The signal varies up and down before settling to its final voltage. The problem can result in the system receiving multiple button presses when only a single button press is given.

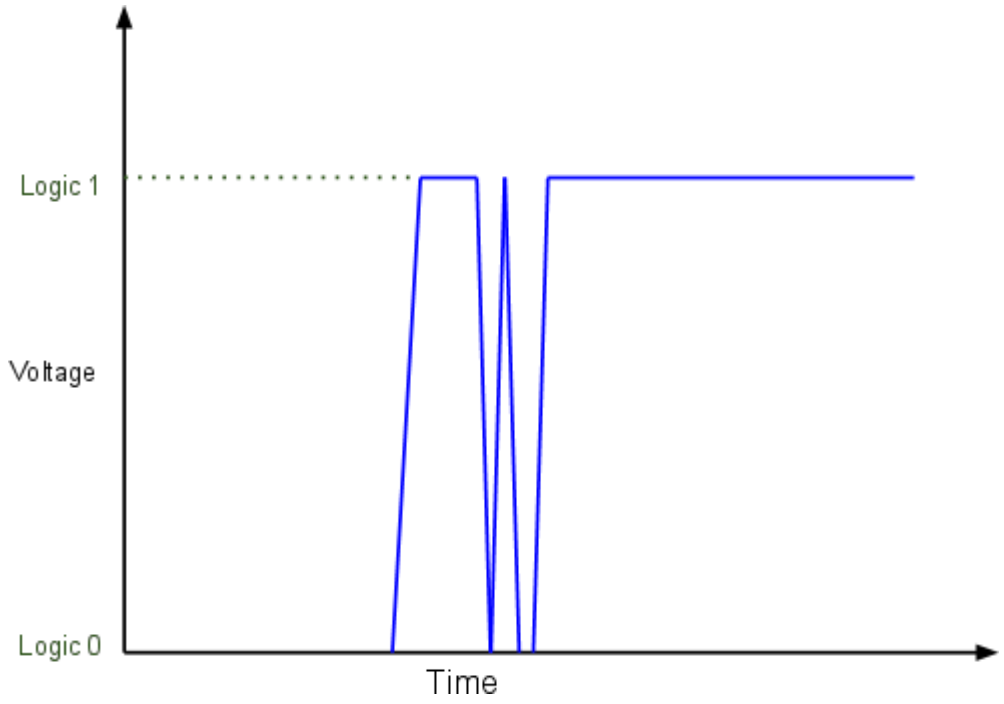


Figure 8 - Button depress "bouncing" waveform

The switch debouncer that is implemented in this design distinguishes a single button press from multiple button presses. It takes the button input and only once it has been continuously high for eight clock cycles does it output that the button is pressed.

Switch Register

A switch register is a register that stores a button press before the CPU collects this information. It is placed immediately after the switch debouncer. The "enable" port is set to low once the output of the register is high which means the register will not accept any more input when a button press is stored in the switch register. The "enable" port of the switch register is set back to high when the acknowledge bit is set to low which means the button press information is loaded on the data bus to the CPU and the switch register is ready to take new input.

Switch registers are needed because not all button presses are sent to the CPU immediately, it needs to be stored until the CPU asks for it. The current design of the switch register can only store one button press.

When the CPU asks the information about button presses (i.e. the R/!W is 1, the data request is 0 and the address is one of the buttons), the switch register will load the information onto the data bus. Then the acknowledge bit of the control bus will be set to low by the I/O component.

Figure 9 shows a simple illustration of the current implementation for one switch. The address bus and R/!W, request signal is omitted in the diagram.

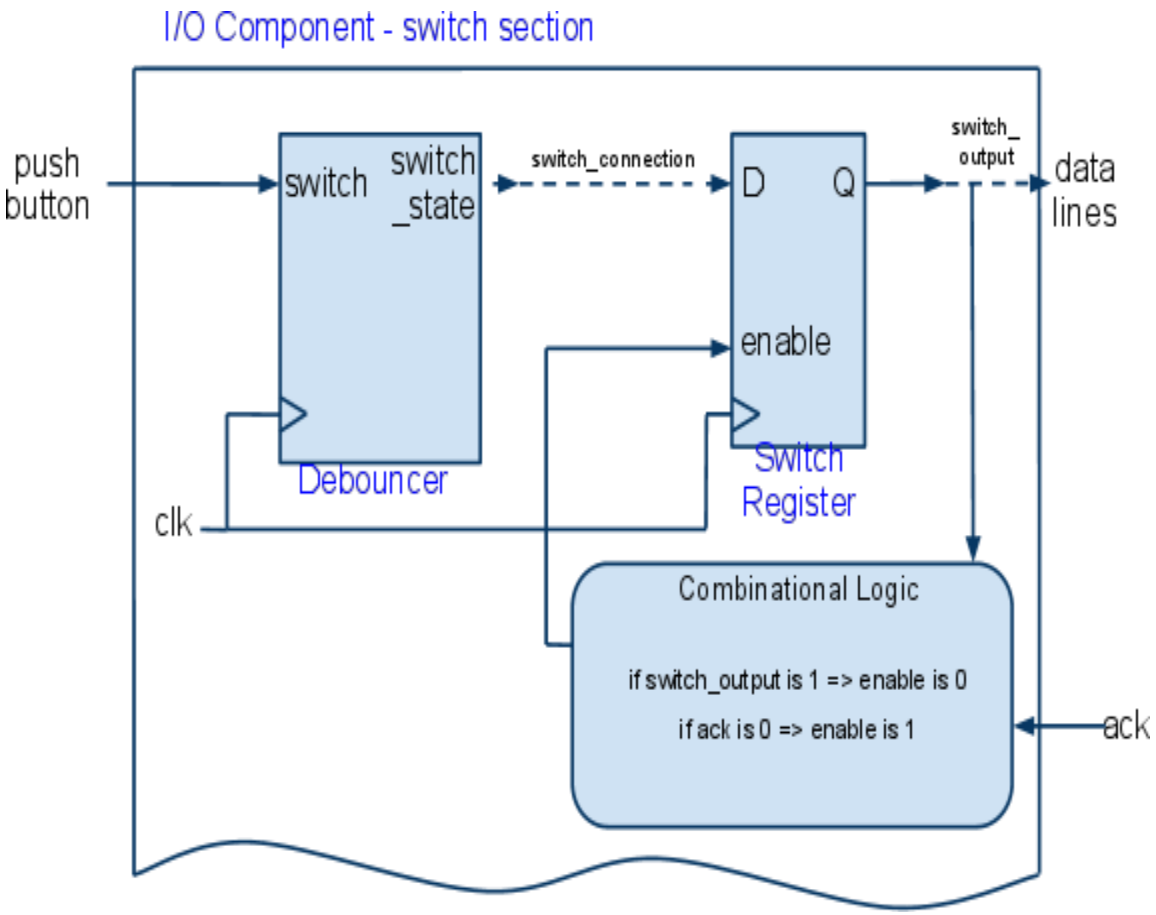


Figure 9: Switch Control

Led Control

All 8 output LEDs are accessed through the same address, in this case each bit of the 8 bit data line represents the on/off state of one LED, e.g. if the data on the data line is '11000011' then the first two and last two leds would be turned on and the other leds would be stayed off. When the CPU sets the address bus to that of the LEDs, the data line gets stored in an 8 bit register and outputted to the LEDs, the acknowledge bit of the control bus is pulled low until the CPU has reset the request bit of the control bus back to high.

Testing

Testing was carried out by implementing the code in Xilinx as a new project. In the test project a button is debounced and sets the data bus to turn the LEDs on.

Tools

Developed

Assembler

A two-pass assembler was developed in Ruby. The first pass scans the file for labels and records their absolute position within the file. The second pass then scans each line and breaks it up into opcode and operands, depending on the instruction in the line different functions are called to convert the operands into their bit pattern representation. The opcode and operands determined are then combined and added to the current output.

This was tested using a simple input file and comparing the output by hand to the expected values.

Memory Provider

A memory provider was also developed in Ruby utilising the ruby-serialport gem[2] for access to the computers serial port. This takes in a program file generated from the assembler and optionally an initial data file, a possible extension to the assembler would be to have it generate this data file from constants in the assembly file. These input files are then stored in arrays and
accessed according to the MMUs instruction over the serial line.

This was not tested for two reasons, a lack of time and no access to installing the ruby-serialport gem on the computers at Uni. While the MMU was fully working in the end there was not enough time to develop a simple test that for example could connect to the memory provider and just display the data in specific addresses. Also while the computers at Uni had Ruby installed, they lacked the Ruby Gem management program and attempts at compiling it failed so the gem required for the low level hardware access could not be installed.

Utilised

While developing the VHDL code several tools were used. Xilinx ISE was used to check the syntax and schematics, Subversion for code management and then GHDL was used to run the test benches.
As we did not have access to Modelsim, the best method we came up with to run test benches was to use GHDL. GHDL is an open source simulator. This allowed the test benches to assert the outputs from entities and write to stdout if something unexpected happens. GHDL was also able to generate signal waveforms that could be opened in GTKWave and verified.

[2] http://ruby-serialport.rubyforge.org/

Discussion

Currently the design units have not been successfully integrated into a single system due to time constraints.

I/O

Future Implementation
The switch register can easily be modified to process concurrent button presses. The control section also needs to be implemented to interface with and handle the data bus, this would allow for easier expansion of the system in terms of I/O devices.

This would, for example, allow a particular button to be used for ending the input. The CPU would then be able to perform different tasks for 1 button press, 2 button presses or different button combinations.

CPU

Because of the modular design of the CPU, implementing pipe lining would be simple. As the code is modular, all the original code could be reused, allowing the pipeline entities to interface between the existing CPU entities. The only original code that would need to be changed would be the cpu.vhd file. The port maps would also need to be rearranged to include any new code.

Because of time constraints, the CPU as a whole was not able to be simulated and tested. Tests such as those written for the ALU and registers were purely localised, setting an input and expecting an output. The control unit however, was a much larger entity that required data to flow to and from the ALU, various registers as well as memory from the MMU. This made it a lot harder to write test benches for its testing and simulation. However, this could have been implemented had more time been provided.

MMU

Queueing was not implemented in the MMU. In order to provide queueing functionality, cache registers and extra control logic would need to be developed. The bus would need to have extra functionality also, allowing the MMU to request data to be sent to the CPU, after it has been read successfully from the serial link.

Communications between group members

The group initially met to discuss the project. During this meeting, the respective design modules were assigned to groups of two group members. During the course of the project, the sub-groups met together to develop their individual modules, while all-inclusive top-level group meetings still occasionally took place. Towards the end of the design stage, meetings involving all group members increased in frequency, as the modules were integrated into the top-level design.

During the report preparation, the whole team actively collaborated together using the Google Documents online application. This meant each team member could actively take part in discussing what everyone

else was working on. This vastly improved the productivity of the team over conventional methods - the report was written more accurately and efficiently.

Problems with the design

When the group is trying to combine all the part together, it is realised that the I/O component should connect to the MMU rather than directly to the CPU. However the functionality of the I/O is the same except all the buses come out of the I/O should directed to the MMU.

What we learnt

We learnt that unforeseen issues can often arise, so adequate time needs to be allocated to allow for testing and fixing problems and bugs.

We have learnt how to code circuits using VHDL and how to implement these as a digital circuit on a FPGA through Xilinx. As well as how small errors in the VHDL code can implement a completely different digital circuit, for example latches.

Latches were also an issue during stages of the development. We learnt to change latches into flip-flops by synchronising them with the clock signal, by placing the code inside process blocks , which are sensitive to the clock signal.

We also learnt how to use test benches to verify the circuits we were implementing followed the specifications. This was just using GHDL but the same test benches with slight tweaks should be valid in ModelSim or ISim, the Xilinx ISE simulator.

Conclusion

In conclusion, the majority of each separate unit has been completed, but not successfully integrated due to time constraints. The code for the whole design has been well thought out, with the design developed to work. With the right tools and more time, the solution could be completed. Overall, the group worked well as a whole and as pairs. Separating the project into three sections meant that the pairs could get together regularly to team code, especially while trying to learn the syntax of VHDL. The end results of the project were that all group members now have a greater understanding of hardware design techniques, and how to implement them in the VHDL language.

References

[1] A. Greensted, The Lab Book Pages, June 2010, [Online], Available: http://www.labbookpages.co.uk/electronics/debounce.html
[2] Brown, Stephen D. , Vranesic, Zvonko G; Fundamentals of digital logic with VHDL design; 3rd ed; McGraw-Hill, 2009.
[3] D. A. Patterson, J. L. Hennessy, "The Processor: Datapath and Control," in Computer Organization and Design, 3th ed.Elsevier, 2005, ch. 5.
[4] P. P. Chu, FPGA prototyping by VHDL examples: Xilinx Spartan-3 version.Wiley-Interscience, 2008.
[5] P.P. Chu, RTL hardware design using VHDL coding for efficiency, portability and scalability, Wiley-Interscience, 2006.
[6] T. Gingold, GHDL guide, 2007, [Online], Available: http://ghdl.free.fr/ghdl/

Appendices

Appendix A: Figures

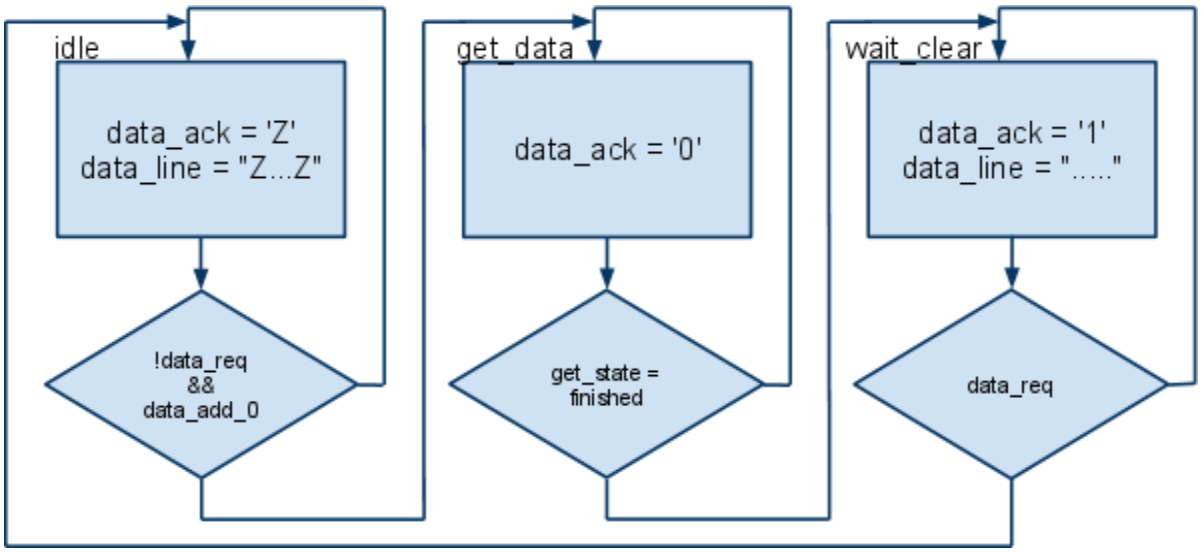


Figure 10: Main data control unit state machine.

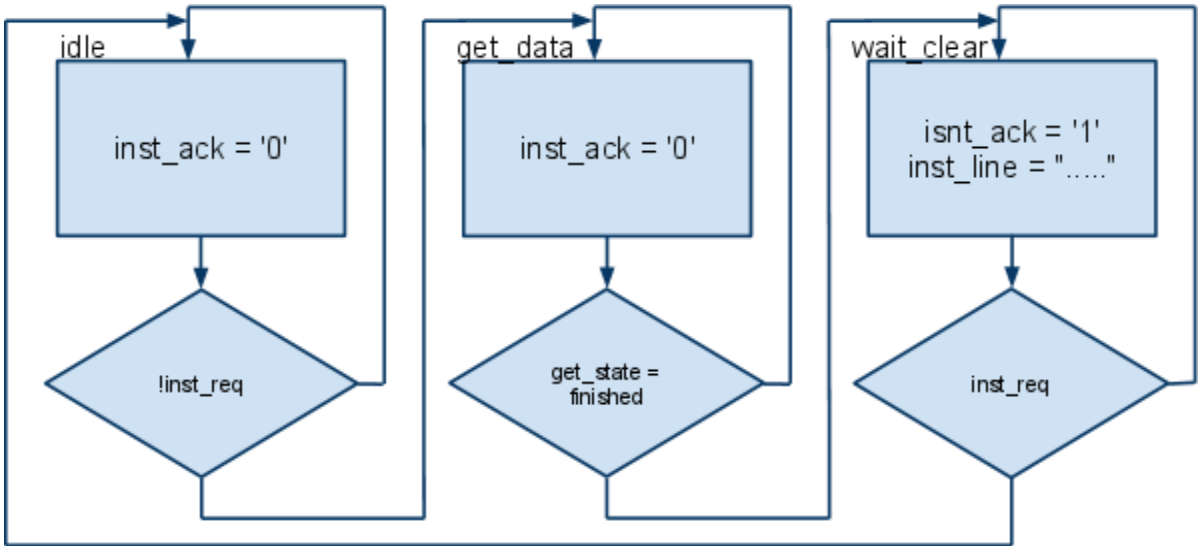


Figure 11: Main instruction control unit state machine.

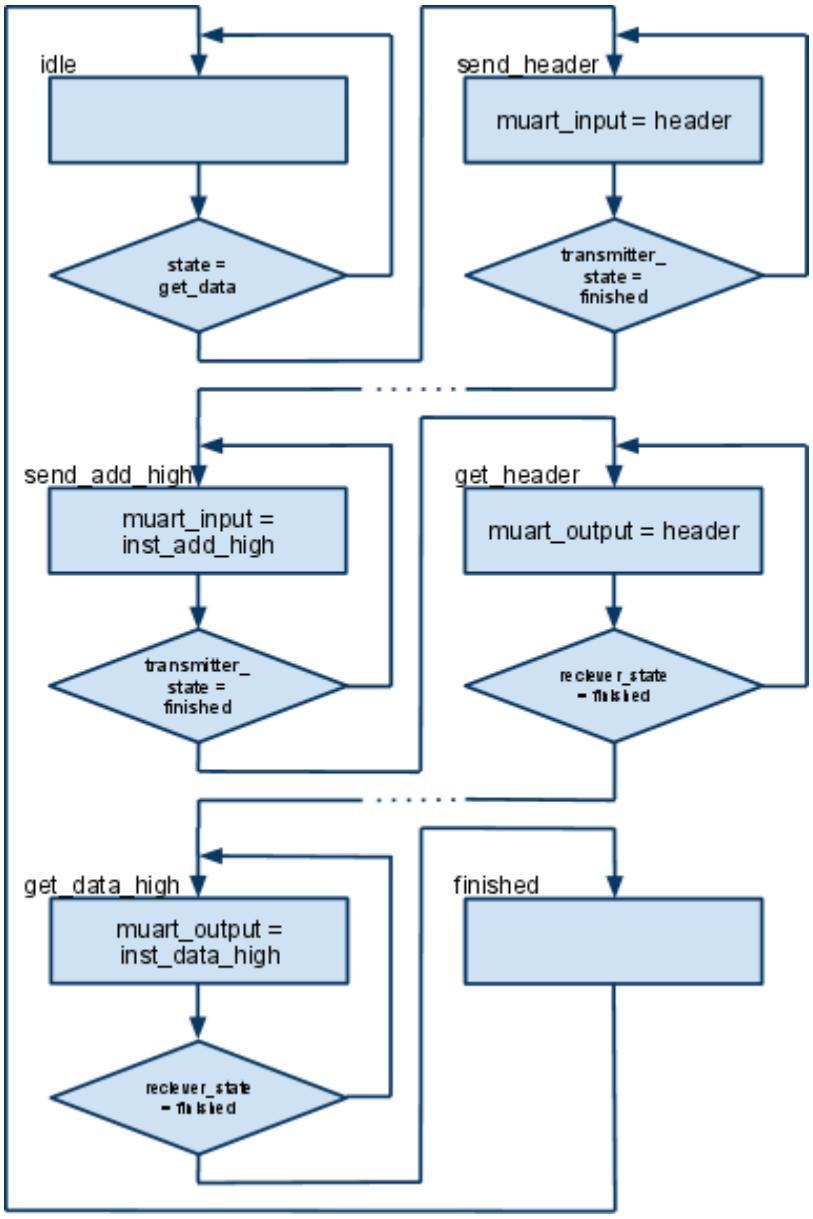


Figure 12: Instruction control unit transferring states.

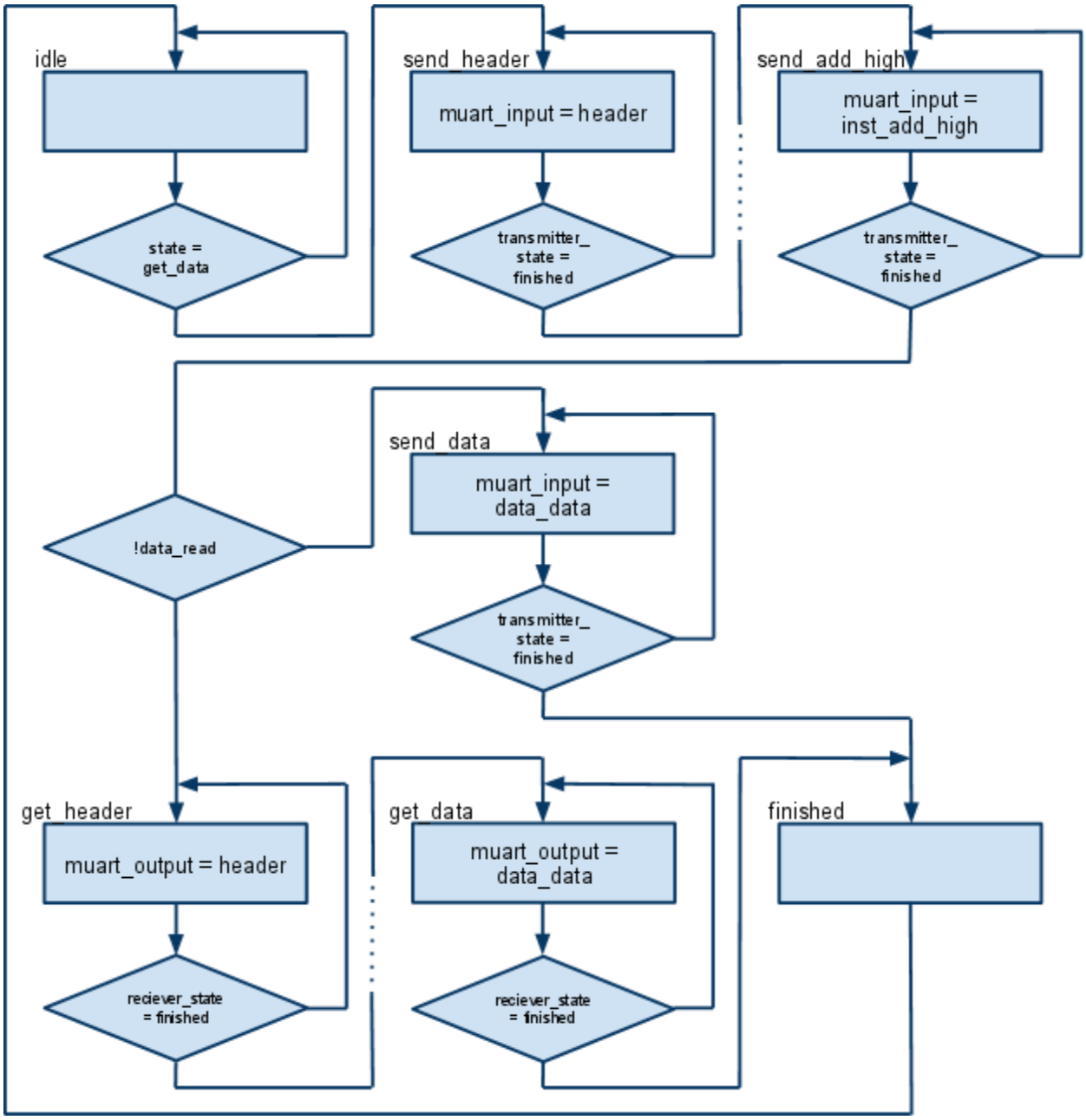


Figure 13: Data control unit transferring states.

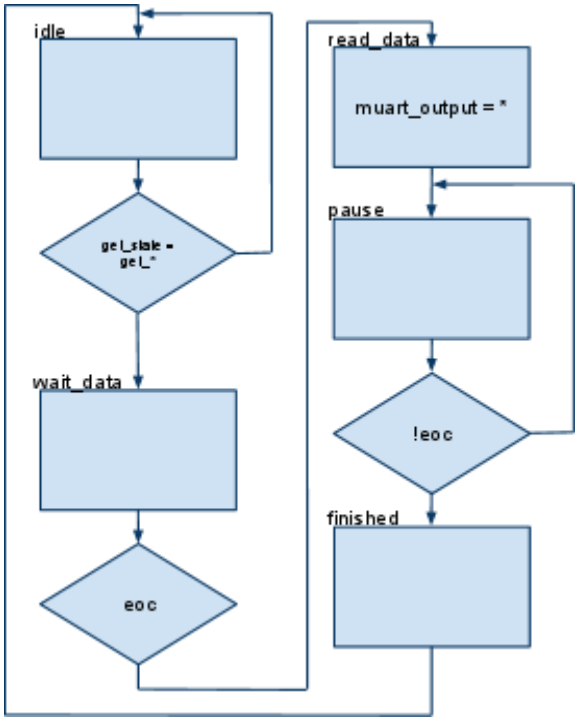


Figure 14: Receive state machine.

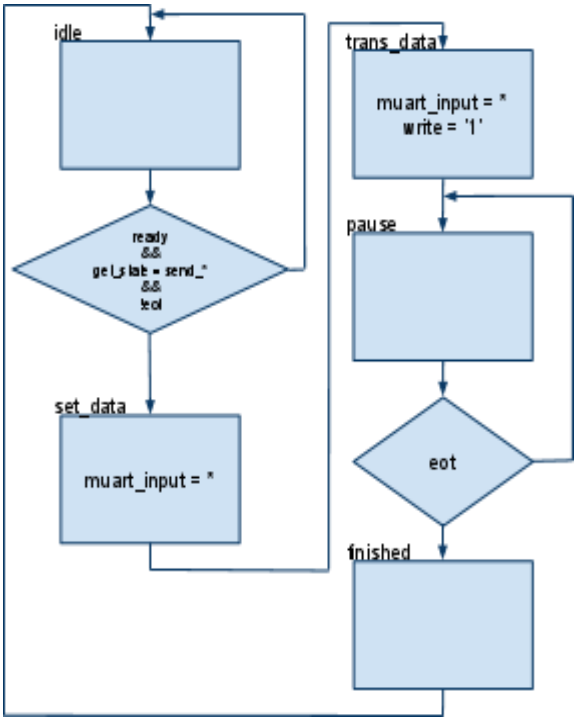


Figure 15: Transmit state machine.

Appendix B: Code listing