ENEL353 Computer Hardware Engineering I

Digital Logic Design Project

October 18, 2010

Group 1:
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.

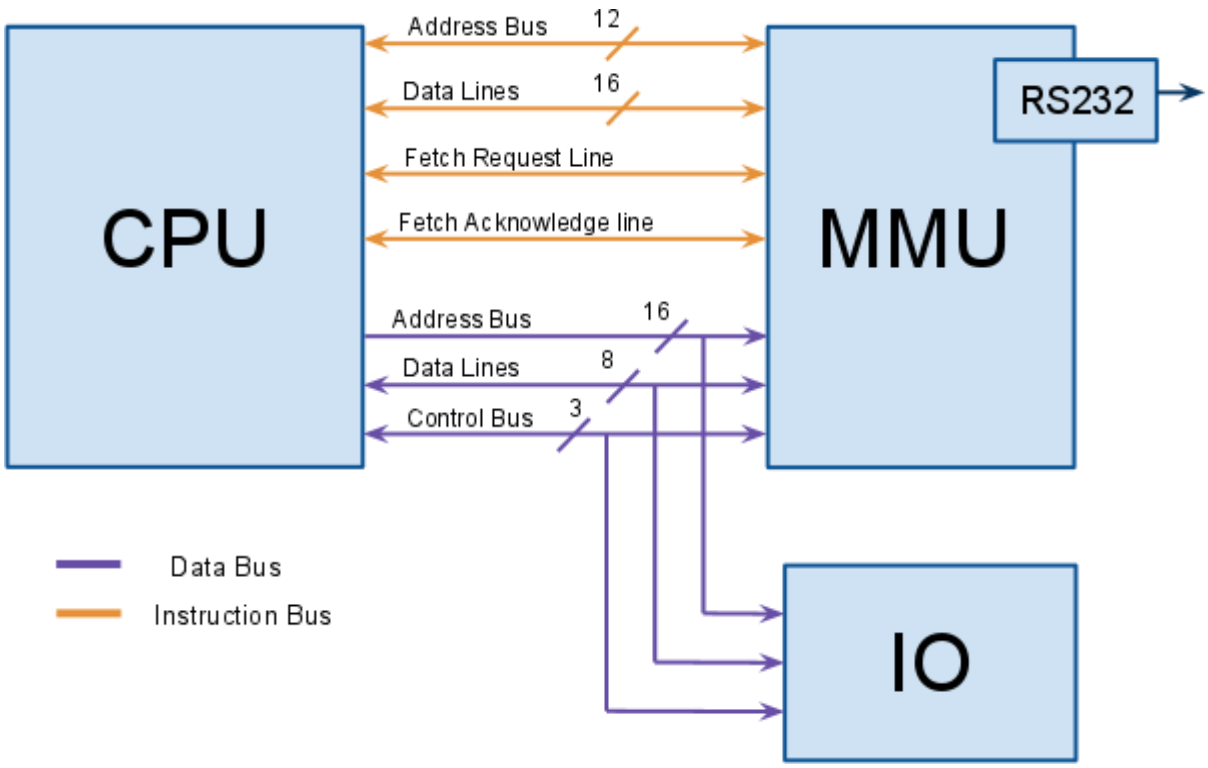

Figure 1: Overall system datapath

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	**1**	**0**	**1**

MMU/IO writes the data then set acknowledge			
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.

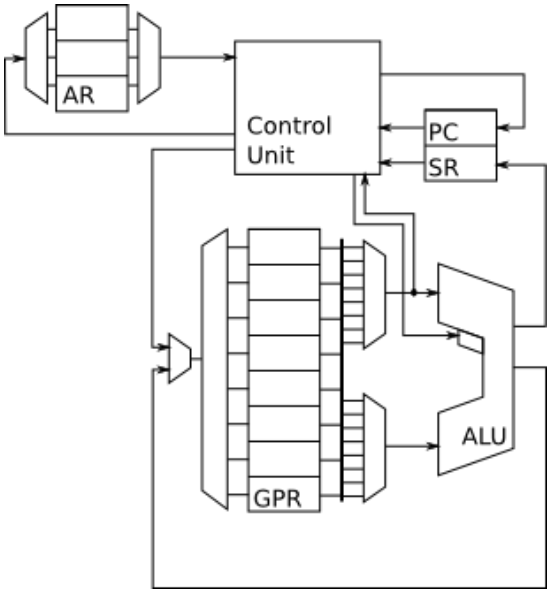

Figure 2: Data flow within CPU

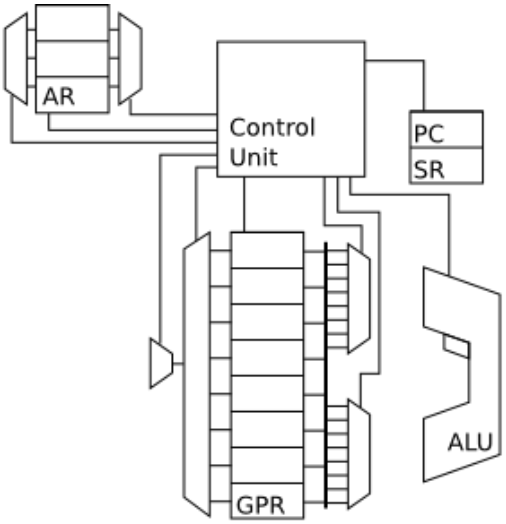

Figure 3: CPU with control lines only

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

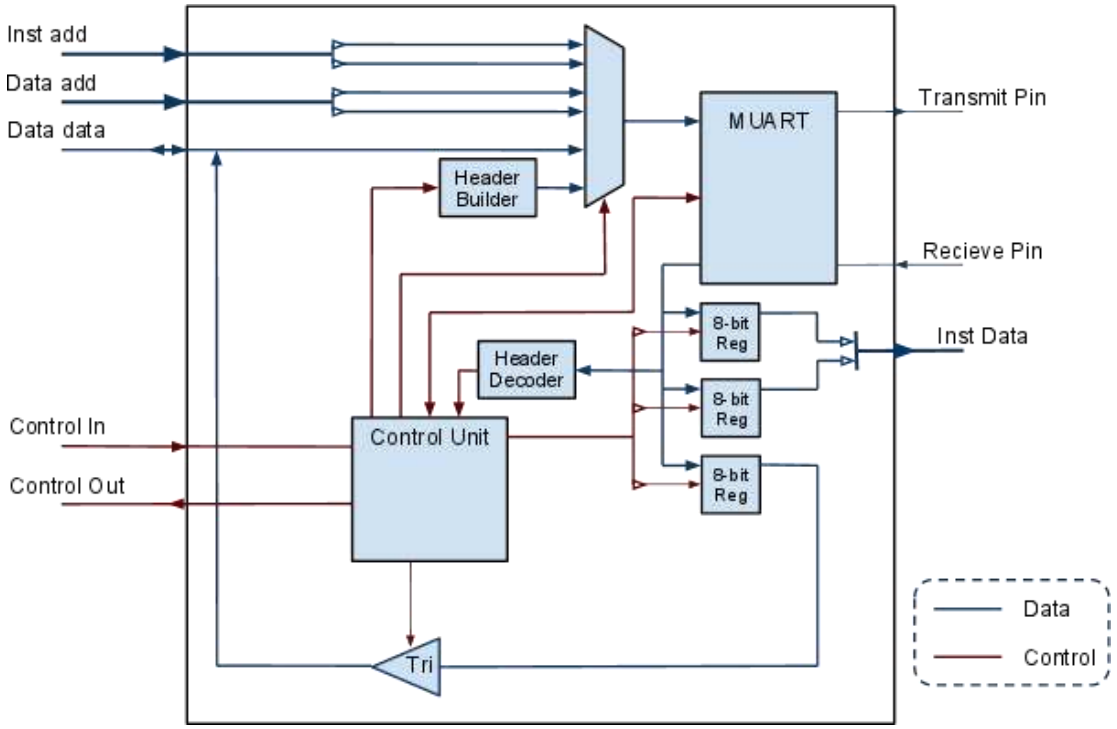

Figure 4: Data and Control paths of the MMU.

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:

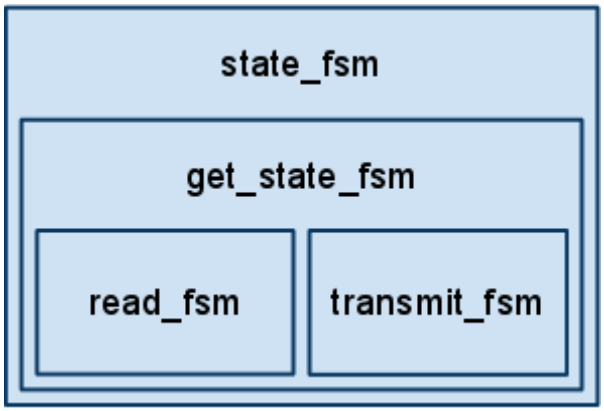

Figure 5: MMU Control Unit state machines

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.

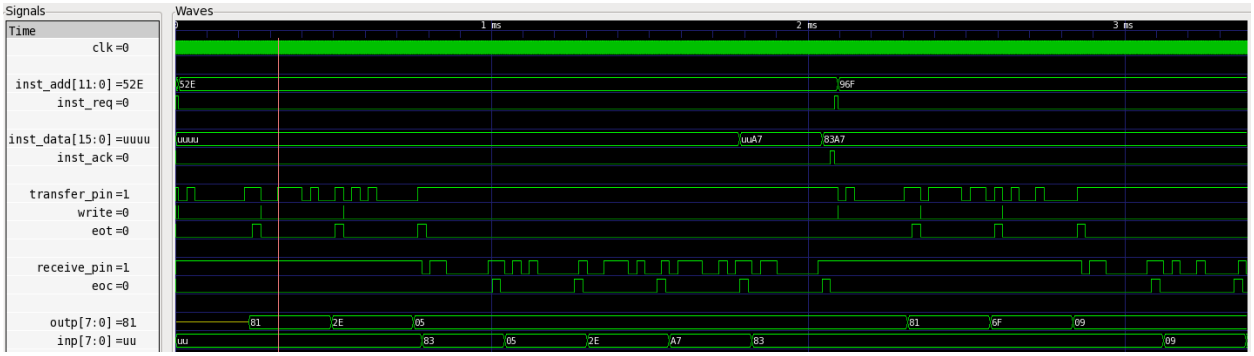

Figure 6: Instruction bus testing waveforms

- **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.

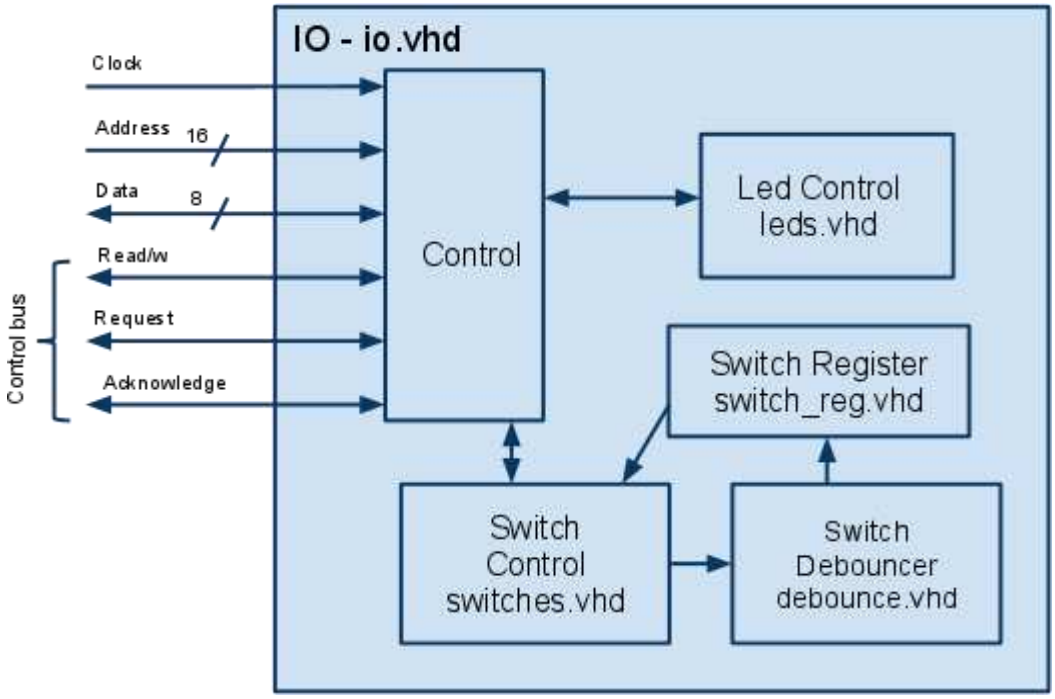

Figure 7: An overview of the I/O component's structure

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.

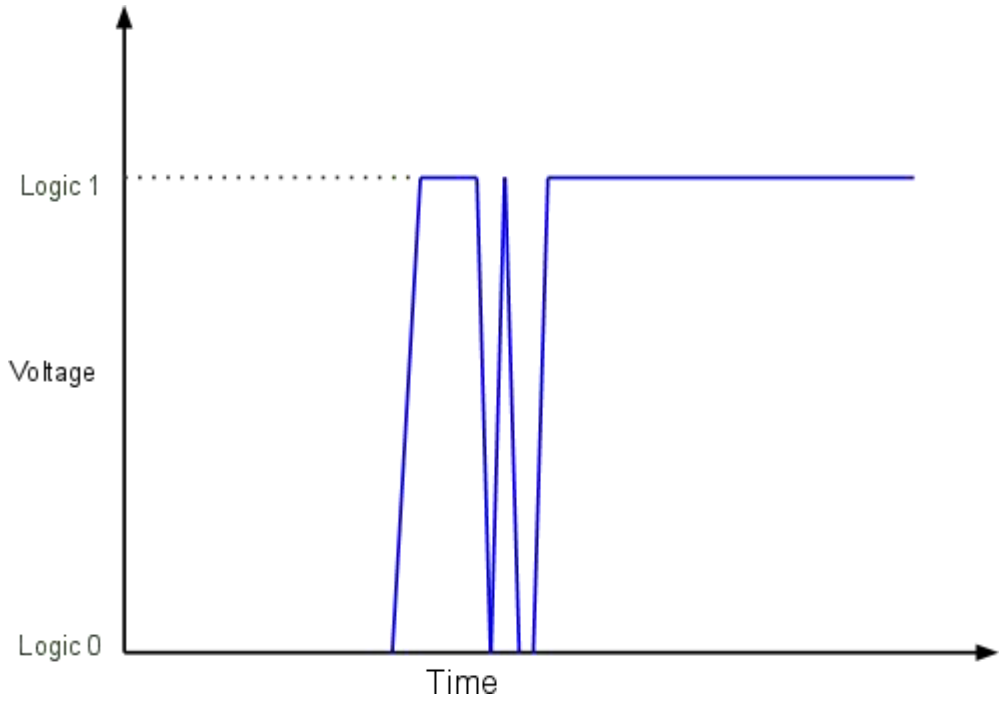

Figure 8 - Button depress "bouncing" waveform

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.

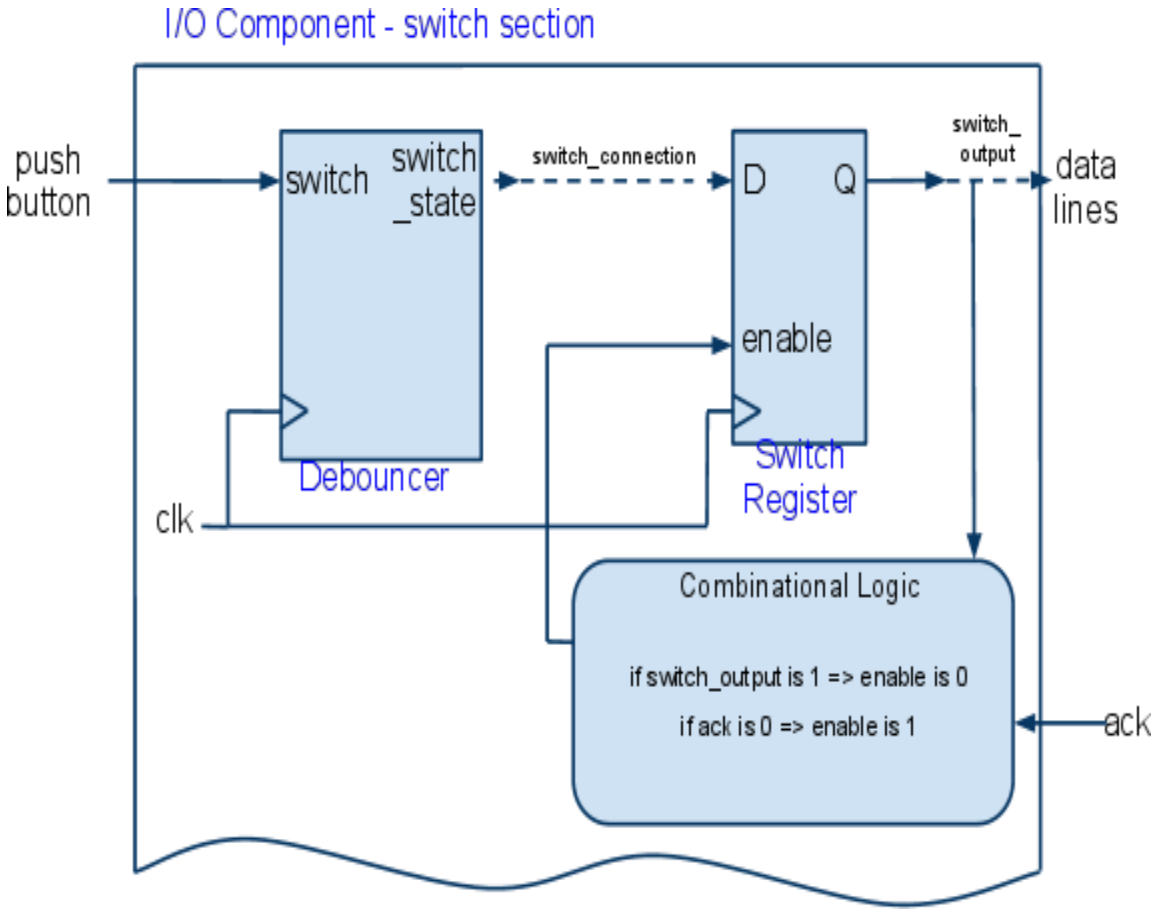

Figure 9: Switch Control

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

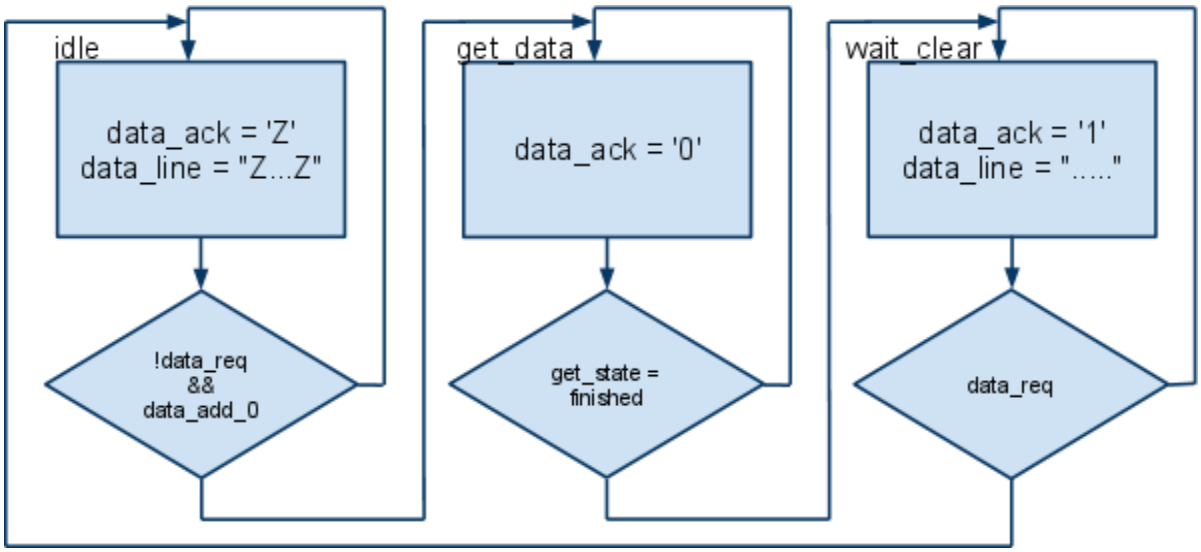

Figure 10: Main data control unit state machine.

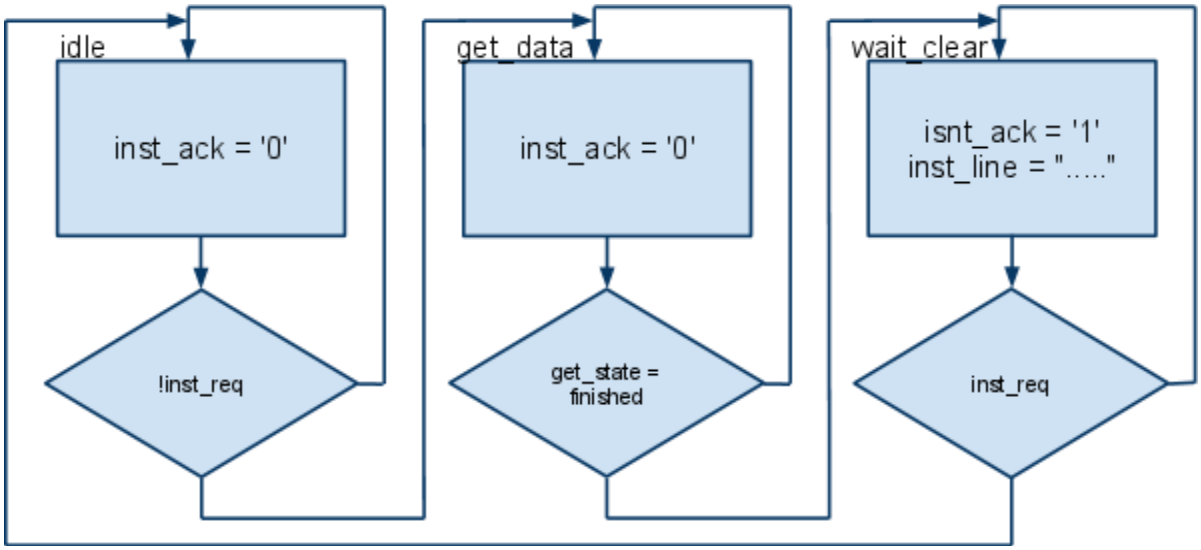

Figure 11: Main instruction control unit state machine.

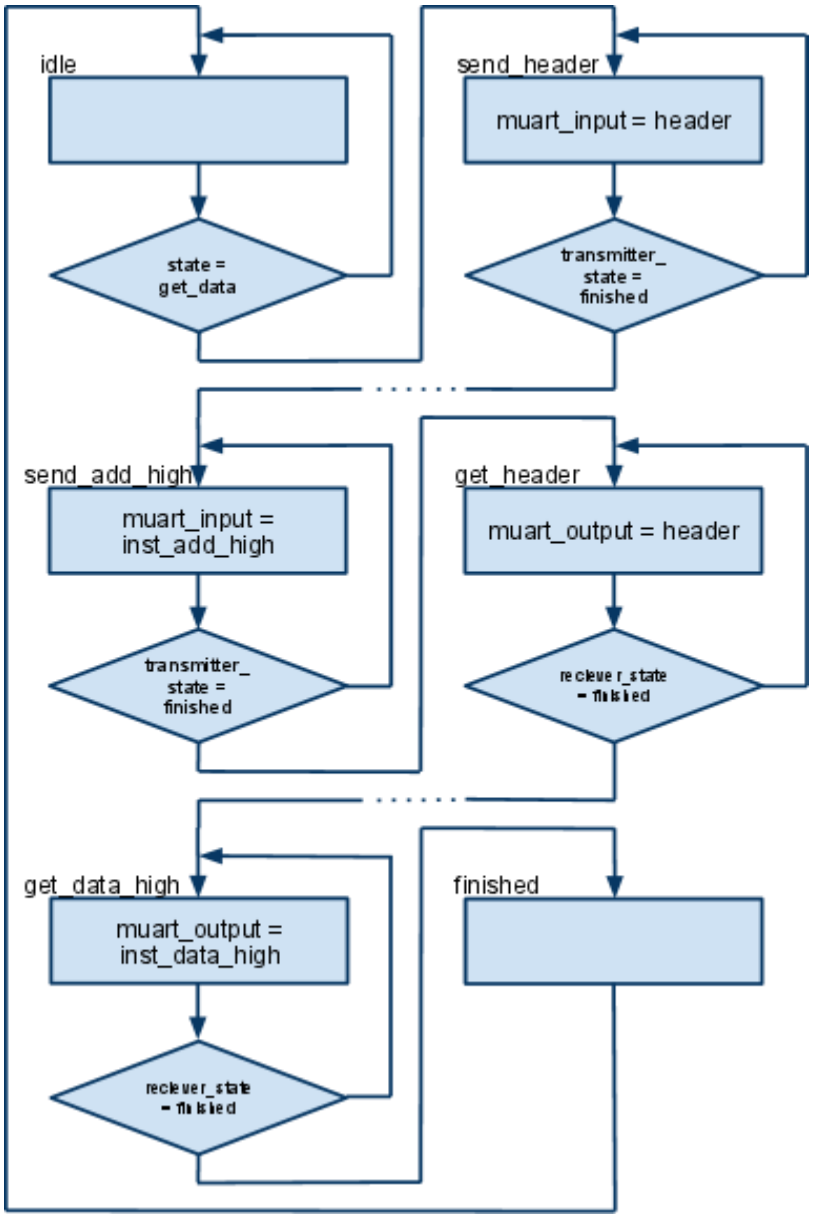

Figure 12: Instruction control unit transferring states.

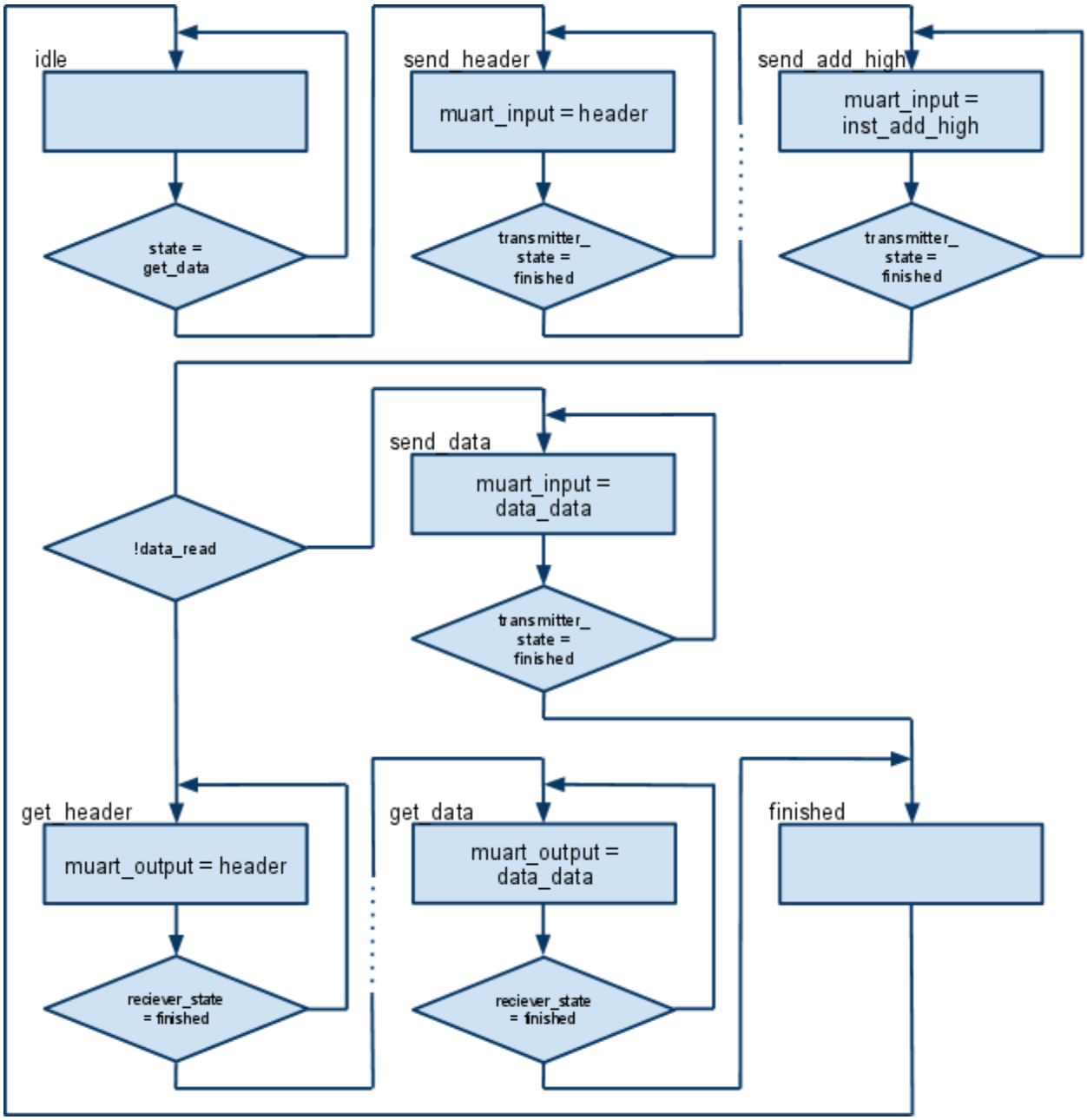

Figure 13: Data control unit transferring states.

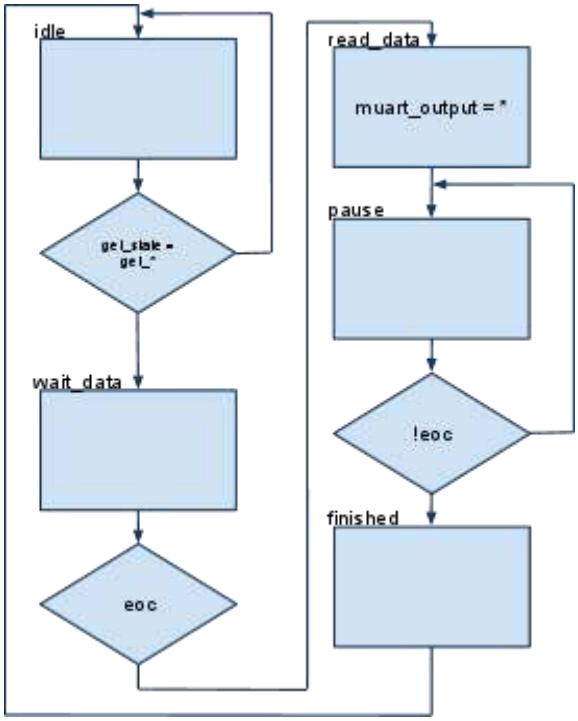

Figure 14: Receive state machine.

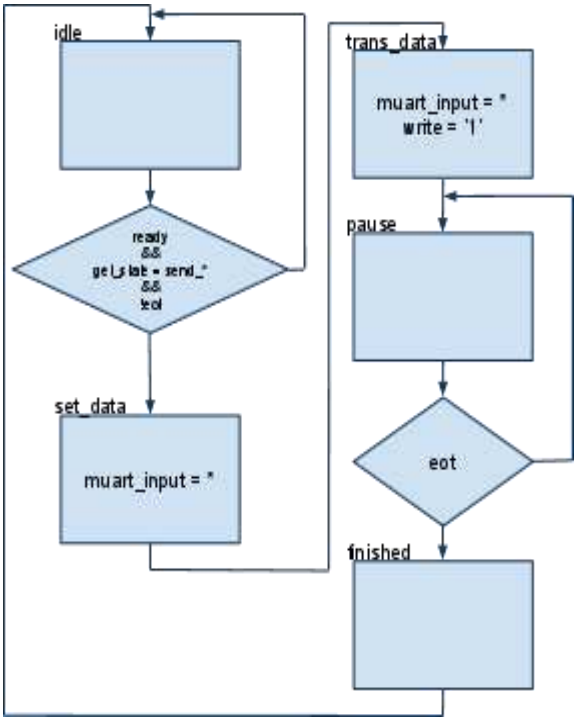

Figure 15: Transmit state machine.

Appendix B: Code listing

Listings

Main

Listing 1: main.vhd

```
1  -- Authors:
   --       Wim Looman, Forrest McKerchar, Henry Jenkins, Joel Koh, Sasha Wang, Tracy
       Jackson
3
   library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;

7  library work;

9  entity main is
     port (
11          clk    : in  std_logic;
            reset  : in  std_logic;
13          tx     : out std_logic;
            rx     : in  std_logic;
15          sw1    : in  std_logic;
            sw2    : in  std_logic
17   );
   end main;
19
   architecture main_arch of main is
21   component cpu IS
       PORT(
23       -- instruction bus
         inst_add  : out std_logic_vector(11 downto 0); -- Address lines.
25       inst_data : in  std_logic_vector(15 downto 0); -- Data lines.
         inst_req  : out std_logic;                     -- Pulled low to request bus
             usage.
27       inst_ack  : in  std_logic;                     -- Pulled high to inform of
             request completion.
         -- data bus
29       data_add  : out    std_logic_vector(15 downto 0); -- Address lines.
         data_line : inout std_logic_vector(7 downto 0);  -- Data lines.
31       data_read : out    std_logic;                     -- High for a read request,
             low for a write request.
         data_req  : out    std_logic;                     -- Pulled low to request bus
             usage.
33       data_ack  : inout std_logic;                     -- Pulled high to inform of
             request completion.
         -- extras
35       clk       : in  std_logic;
         reset     : in  std_logic
37     );
     end component;
39   component mmu_main is
       port (
41       -- instruction bus
         inst_add  : in  std_logic_vector(11 downto 0); -- Address lines.
43       inst_data : out std_logic_vector(15 downto 0); -- Data lines.
         inst_req  : in  std_logic;                     -- Pulled low to request bus
             usage.
45       inst_ack  : out std_logic;                     -- Pulled high to inform of
             request completion.
```

```
        -- data bus
47      data_add  : in    std_logic_vector(15 downto 0); -- Address lines.
        data_line : inout std_logic_vector(7 downto 0);  -- Data lines.
49      data_read : in    std_logic;                     -- High for a read request,
            low for a write request.
        data_req  : in    std_logic;                     -- Pulled low to request bus
            usage.
51      data_ack  : inout std_logic;                     -- Pulled high to inform of
            request completion.
        -- extras
53      clk          : in  std_logic;
        receive_pin  : in  std_logic;
55      transfer_pin : out std_logic
    );
57  END component;
    component IO is
59       PORT(
             -- data bus --
61           data_add    : IN      std_logic_vector(15 DOWNTO 0); -- address lines --
             data_data   : INOUT   std_logic_vector(7 DOWNTO 0);  -- data lines --
63           data_read   : INOUT   std_logic;                     -- pulled high for
                 read, low for write --
             data_req    : INOUT   std_logic;                     -- pulled low to
                 request bus usage --
65           data_ack    : INOUT   std_logic;                     -- pulled high to
                 inform request completion --
             -- io --
67           clk         : IN      std_logic;
             sw1         : IN      std_logic;
69           sw2         : IN      std_logic);
             --leds      : OUT std_logic_vector(7 DOWNTO 0);
71  END component;
    -- instruction bus
73  signal inst_add  : std_logic_vector(11 downto 0); -- Address lines.
    signal inst_data : std_logic_vector(15 downto 0); -- Data lines.
75  signal inst_req  : std_logic;                     -- Pulled low to request bus
        usage.
    signal inst_ack  : std_logic;                     -- Pulled high to inform of
        request completion.
77  -- data bus
    signal data_add  : std_logic_vector(15 downto 0); -- Address lines.
79  signal data_line : std_logic_vector(7 downto 0);  -- Data lines.
    signal data_read : std_logic;                     -- High for a read request, low
        for a write request.
81  signal data_req  : std_logic;                     -- Pulled low to request bus
        usage.
    signal data_ack  : std_logic;                     -- Pulled high to inform of
        request completion.
83
    begin
85    c : cpu port map(
        -- instruction bus
87      inst_add  => inst_add, -- Instruction address
        inst_data => inst_data,-- Instruction data
89      inst_req  => inst_req, -- Request
        inst_ack  => inst_ack, -- Instruction obtained
91      -- data bus
        data_add  => data_add, -- Data address
93      data_line => data_line,-- Data
        data_read => data_read,-- 1 for read, 0 for write
95      data_req  => data_req, -- Request
        data_ack  => data_ack, -- Data written to/ read from
97      -- extras
        clk       => clk,
99      reset     => reset
        );
101   m : mmu_main port map(
        -- instruction bus
103     inst_add     => inst_add,  -- Address lines.
        inst_data    => inst_data, -- Data lines.
105     inst_req     => inst_req,  -- Pulled low to request bus usage.
        inst_ack     => inst_ack,  -- Pulled high to inform of request completion.
107     -- data bus
```

```
        data_add     => data_add,  -- Address lines.
        data_line    => data_line, -- Data lines.
        data_read    => data_read, -- High for a read request, low for a write request.
        data_req     => data_req,  -- Pulled low to request bus usage.
        data_ack     => data_ack,  -- Pulled high to inform of request completion.
        -- extras
        clk          => clk,
        receive_pin  => rx,
        transfer_pin => tx
    );
    i : io  port map(
            clk          => clk,
            data_add     => data_add,
            data_data    => data_line,
            data_read    => data_read,
            data_req     => data_req,
            data_ack     => data_ack,
            -- io --
            sw1          => sw1,
            sw2          => sw2
        );
end architecture main_arch;
```

IO

Listing 2: IO/debounce.vhd

```
----------------------------------------------------------------------------------
-- Module Name:    debounce
-- Description: Entity to debounce a mechanical switch/button
-- Authors: Tracy Jackson
--          Sasha Wang
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED."+";

library work;

ENTITY debounce IS
       PORT(clk : IN STD_LOGIC;
       switch : IN STD_LOGIC;
       switch_state : OUT STD_LOGIC);
END debounce;

ARCHITECTURE debounced_switch OF debounce IS
    SIGNAL count : STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
     -- Debounce the switch using a counter
     PROCESS(clk, switch)
     BEGIN
          IF switch = '0' THEN
             count <= "000";
          ELSIF rising_edge(clk) THEN
                IF count /= "111" THEN
                   count <= count + 1;
                END IF;
          END IF;
          IF count = "111" AND switch = '1' THEN
             switch_state <= '1';
          ELSE
              switch_state <= '0';
          END IF;
     END PROCESS;
END debounced_switch;
```

Listing 3: IO/IO.vhd

```
----------------------------------------------------------------------------------
-- Module Name:    IO
-- Description: Entity to hangle IO
-- Authors: Tracy Jackson
--          Sasha Wang
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
--use IEEE.STD_LOGIC_UNSIGNED.ALL;

library work;
use work.debounce;
use work.switch_reg;
use work.led_io;


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

```
24
   entity IO is
26         PORT(
                   -- data bus --
28                 data_add         : IN            std_logic_vector(15 DOWNTO 0);
                       -- address lines --
                   data_data    : INOUT         std_logic_vector(7 DOWNTO 0);    -- data
                       lines --
30                 data_read    : INOUT         std_logic;
                       -- pulled high for read, low for write --
                   data_req         : INOUT     std_logic;
                       -- pulled low to request bus usage --
32                 data_ack         : INOUT     std_logic;
                       -- pulled high to inform request completion --
                   -- io --
34                     clk          : IN            std_logic;
                       sw1          : IN            std_logic;
36                     sw2          : IN            std_logic);
               --leds       : OUT std_logic_vector(7 DOWNTO 0);
38 end IO;

40 architecture io of IO is


42
   COMPONENT led_io
44     PORT(
               data_add         : IN            std_logic_vector(15 DOWNTO 0);       --
                   address lines --
46             data_data    : INOUT         std_logic_vector(7 DOWNTO 0);    -- data lines
                    --
               data_read    : INOUT         std_logic;                                --
                   pulled high for read, low for write --
48             data_req         : INOUT     std_logic;                                --
                   pulled low to request bus usage --
               data_ack         : INOUT     std_logic;                                --
                   pulled high to inform request completion --
50
               clock        : IN            std_logic
52             );
   END COMPONENT;
54
   COMPONENT switch_io IS
56     PORT ( data_add          : IN            std_logic_vector(15 DOWNTO 0);
              data_data         : INOUT         std_logic_vector(7 DOWNTO 0);
58            data_read         : INOUT         std_logic;
              data_req          : INOUT         std_logic;
60            data_ack          : INOUT         std_logic;
              clk               : IN            std_logic;
62            sw1               : IN            std_logic;
              sw2               : IN            std_logic
64            );
   END COMPONENT;
66


68
   BEGIN
70
   led: led_io PORT MAP(data_add, data_data, data_read, data_req, data_ack, clk);
72 switch: switch_io PORT MAP(data_add, data_data, data_read, data_req, data_ack,
       clk,sw1,sw2);

74 ----------------------------------------------------------------------------------------------

76 END io;
```

Listing 4: IO/leds.vhd

```
   ----------------------------------------------------------------------------
2  -- Module Name:    led_io
   -- Description: Entity control output LEDs
4  -- Authors: Tracy Jackson
   --          Sasha Wang
```

```
6  --
   ----------------------------------------------------------------------------
8  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;

12 library work;

14 ENTITY led_io IS
         PORT(
16                 data_add        : IN            std_logic_vector(15 DOWNTO 0);
                       -- address lines --
                   data_data    : INOUT         std_logic_vector(7 DOWNTO 0);    -- data
                       lines --
18                 data_read    : INOUT         std_logic;
                       -- pulled high for read, low for write --
                   data_req        : INOUT     std_logic;
                       -- pulled low to request bus usage --
20                 data_ack        : INOUT     std_logic;
                       -- pulled high to inform request completion --
                   --
22                 clock       : IN            std_logic;
                   );
24 END led_io;

26 ARCHITECTURE led_arch OF led_io IS
         Signal led_enable       : std_logic;
28       Signal led_state        : std_logic_vector(7 DOWNTO 0);
   BEGIN
30
      -- Determine if it is the LEDs being accessed
32    PROCESS(clock, data_req, data_add, data_read)
      BEGIN
34              IF data_req = '0' AND data_add = "0000000000001110" AND data_read = '0'
                    THEN
                        led_enable <= '1';
36              ELSE
                        led_enable <= '0';
38              END IF;
      END PROCESS;
40
        -- process of data from the CPU and output to LEDS
42     PROCESS(clock, led_enable)
       BEGIN
44              IF rising_edge(clock) THEN
                    IF led_enable = '1' THEN
46                          led_state <= data_data;
                            data_ack <= '0';
48                  END IF;
                END IF;
50     END PROCESS;


52

54 END led_arch;
```

Listing 5: IO/switch_register.vhd

```
1  ----------------------------------------------------------------------------
   -- Module Name:    switch_reg
3  -- Description: Entity to store switch state (can be extended to more than one)
   -- Authors: Tracy Jackson
5  --          Sasha Wang
   --
7  ----------------------------------------------------------------------------
   library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
   use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;

13 library work;
```

```
ENTITY switch_reg IS
PORT( D                 : IN STD_LOGIC;
      clk,enable    : IN STD_LOGIC;
      Q                 : OUT STD_LOGIC);
END switch_reg;

ARCHITECTURE reg_arch OF switch_reg IS
BEGIN
     PROCESS(D, enable, clk)
      BEGIN
            IF rising_edge(clk) THEN --Need else there???
                    IF enable = '1' THEN
                        Q <= D;
                    END IF;
            END IF;
     END PROCESS;
END reg_arch;
```

Listing 6: IO/switches.vhd

```
--------------------------------------------------------------------------------
-- Module Name:    switch_io
-- Description: Entity to control input from switches
-- Authors: Tracy Jackson
--          Sasha Wang
--
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

library work;
use work.debounce;
use work.switch_reg;

entity switch_io is
       PORT(
                -- data bus --
                data_add        : IN            std_logic_vector(15 DOWNTO 0);
                    -- address lines --
                data_data   : INOUT         std_logic_vector(7 DOWNTO 0);   -- data
                    lines --
                data_read   : INOUT         std_logic;
                    -- pulled high for read, low for write --
                data_req        : INOUT     std_logic;
                    -- pulled low to request bus usage --
                data_ack        : INOUT     std_logic;
                    -- pulled high to inform request completion --
                -- io --
            clk         : IN            std_logic;
            sw1         : IN            std_logic;
            sw2         : IN            std_logic);
end switch_io;

architecture Behavioral of switch_io is


signal enable1                  : std_logic;
signal switch1_connection   : std_logic;
signal switch1_output       : std_logic;

signal enable2                  : std_logic;
signal switch2_connection   : std_logic;
signal switch2_output       : std_logic;


COMPONENT debounce
         PORT(clk, switch : IN STD_LOGIC;
                switch_state: OUT STD_LOGIC);
END COMPONENT;

COMPONENT switch_reg
```

```
        PORT( D            : IN STD_LOGIC;
              clk, enable  : IN STD_LOGIC;
              Q               : OUT STD_LOGIC);
END COMPONENT;


BEGIN

sw1_debouncer: debounce PORT MAP(clk, sw1, switch1_connection);
sw1_status: switch_reg PORT MAP(switch1_connection,clk, enable1, switch1_output);

sw2_debouncer: debounce PORT MAP(clk, sw2,switch2_connection);
sw2_status: switch_reg PORT MAP(switch2_connection,clk, enable2, switch2_output);

PROCESS(clk,switch1_output,switch2_output, data_ack)
BEGIN
IF rising_edge(clk) THEN
    IF switch1_output = '1' AND data_ack = 'Z' THEN --when the switch_reg has stored
        1, disable switch_reg from getting any more info
        enable1 <= '0';

    --ELSIF data_ack = '0' AND data_add = "0000000000001110" THEN -- when the data is
         sent to the CPU, enable the switch_reg again
    --  enable1 <= '1';
    ELSE
        enable1 <= '1';
    END IF;

    IF switch2_output = '1' AND data_ack = 'Z' THEN --when the switch_reg has stored
        1, disable switch_reg from getting any more info
        enable2 <= '0';

    --ELSIF data_ack = '0' AND data_add = "0000000000001100" THEN -- when the data is
         sent to the CPU, enable the switch_reg again
    --  enable2 <= '1';
    ELSE
        enable2 <= '1';
    END IF;


END IF;
END PROCESS;

PROCESS(clk, data_add, data_read)
BEGIN
    IF rising_edge(clk) THEN
        IF data_req = '0' AND data_read = '1' THEN
            IF data_add = "0000000000001110" THEN -- switch1 address
                IF switch1_output = '1' THEN
                    data_data <= "00000001";
                ELSE
                    data_data <= "00000000";
                END IF;
                data_ack <= '0';
            END IF;
            IF data_add = "0000000000001100" THEN -- switch2 address
                IF switch2_output = '1' THEN
                    data_data <= "00000001";
                ELSE
                    data_data <= "00000000";
                END IF;
                data_ack <= '0';
            END IF;
        ELSIF data_req = '1' AND data_ack = '0' THEN
            data_ack <= 'Z';
        END IF;
    END IF;
 END PROCESS;
```

```
117
    ------------------------------------------------------------------------------------------------
119
    END Behavioral;
```

CPU

Listing 7: processor/alu.vhd

```
-- Authors:
--       Henry Jenkins, Joel Koh

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_arith.all;
--use ieee.std_logic_unsigned.all;


library work;
use work.fulladder8;
--use work.cpu.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

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

entity alu is
    Port (f   : in   STD_LOGIC_VECTOR (3 downto 0);  -- Function (opcode)
          rx  : in   STD_LOGIC_VECTOR (7 downto 0);  -- Input x (Rx)
          ry  : in   STD_LOGIC_VECTOR (7 downto 0);  -- Input y (Ry)
          ro  : out  STD_LOGIC_VECTOR (7 downto 0);  -- Output Normaly (Ry)
          Cin : in   STD_LOGIC;                      -- Carry in
          sr  : out  STD_LOGIC_VECTOR (15 downto 0)); -- Status register out Z(0),
              C(1), N(2)
end alu;


architecture alu_arch of alu is
    component fulladder8 IS
    Port (A    : in   STD_LOGIC_VECTOR( 7 downto 0);
          B    : in   STD_LOGIC_VECTOR( 7 downto 0);
          Cin  : in   STD_LOGIC;
          Sum  : out  STD_LOGIC_VECTOR( 7 downto 0);
          Cout : out  STD_LOGIC
          );
    end component;
    signal   A         : std_logic_vector(7 downto 0);
    signal   B         : std_logic_vector(7 downto 0);
    signal   AdderCin  : std_logic;
    signal   Sum       : std_logic_vector(7 downto 0);
    signal   AdderCout : std_logic;
    signal   Z,C,N     : std_logic; -- Make the code easier to read
    signal   output    : std_logic_vector(7 downto 0); -- used to allow reading of ro
BEGIN
    Adder: fulladder8 port map(A, B, AdderCin, Sum, AdderCout);
    process(f, rx, ry, Cin, Sum, AdderCout)
      --signal Z,C,N  : std_logic; -- Make the code easier to read
    BEGIN
      -- use case statement to achieve
      -- different operations of ALU

       AdderCin <= '0';
       A <= (others => '0');
       B <= (others => '0');
       output <= (others => '0');
       C <= '0';
       N <= '0';
       IF f = "0001" THEN -- Do AND operation
          output <= ry and rx;
       ELSIF f = "0011" THEN -- Do OR  operation
         output <= ry or rx;
```

```
68        ELSIF f = "0101" THEN
            output <= not rx;
70        ELSIF f = "0111" THEN -- Do XOR operation
            output <= ry xor rx;
72        ELSIF f = "1001" THEN -- Do ADD operation
            AdderCin <= '0';
74          A <= ry;
            B <= rx;
76          output <= Sum;
          ELSIF f = "1011" THEN -- Do ADC operation
78          AdderCin <= Cin;
            A <= ry;
80          B <= rx;
            output <= Sum;
82        ELSIF f = "1101" THEN -- Do SUB operation
            AdderCin <= '1';
84          A <= ry;
            B <= (not rx);
86          output <= Sum;
          ELSIF f = "1111" THEN -- Do SBB operation
88          AdderCin <= (not Cin);
            A <= ry;
90          B <= (not rx);
            output <= Sum;
92        ELSIF f = "0100" THEN -- Do NEG operation ( two's complement )
            AdderCin <= '1';
94          A <= (others => '0');
            B <= (not rx);
96          output <= Sum;
            C <= AdderCout;
98          N <= output(7);
          ELSIF f = "0110" THEN -- Do CMP operation
100         AdderCin <= '1';
            A <= rx;
102         B <= (not ry);
            output <= Sum;
104         C <= AdderCout;
            N <= output(7);
106       ELSE
            AdderCin <= '0';
108         A <= (others => '0');
            B <= (others => '0');
110         output <= (others => '0');
            C <= '0';
112         N <= '0';
          END IF;
114 --    if (output = "00000000") then -- Set the Zero in status register
    --      sr(0) <= '1';
116 --    ELSE
    --      sr(0) <= '0';
118 --    end if;

120       C <= AdderCout; -- Carry is always 0
          N <= output(7); -- This might need to be changed to '0'
122       ro <= output;
      end process;
124     Z <= not (output(0) AND output(1) AND output(2) AND output(3) AND output(4)
                  AND output(5) AND output(6) AND output(7));
126     sr(0) <= Z; --Z(0)
        sr(1) <= C; --C(1)
128     sr(2) <= N; --N(2)
        sr(15 downto 3) <= (others => '0');
130
    end alu_arch;
```

Listing 8: processor/ar.vhd

```
1 -- Authors:
  --       Henry Jenkins, Joel Koh
3
  library IEEE;
5 use IEEE.STD_LOGIC_1164.ALL;
```

```
7  library work;
   use work.reg16;
9
   entity ar is
11   Port (clk         : in   STD_LOGIC;
          enable      : in   STD_LOGIC;
13        Sel8Bit     : in   STD_LOGIC;
          SelHighByte : in   STD_LOGIC;
15        ByteInput   : in   STD_LOGIC_VECTOR (7 downto 0);
          SelRi       : in   STD_LOGIC_VECTOR (1 downto 0);   -- Select the address
              register
17        SelRo       : in   STD_LOGIC_VECTOR (1 downto 0);   -- Select the address
              register
          Ri          : in   STD_LOGIC_VECTOR (15 downto 0);  -- The input
19        Ro          : out  STD_LOGIC_VECTOR (15 downto 0)); -- The output
   end ar;
21
   architecture Behavioral of ar is
23   component reg16 IS
       port(I      : in  std_logic_vector(15 downto 0);
25          clock  : in  std_logic;
            enable : in  std_logic;
27          reset  : in  std_logic;
            Q      : out std_logic_vector(15 downto 0)
29          );
     end component;
31
     signal  R0E   : std_logic;  -- Enable signals
33   signal  R1E   : std_logic;
     signal  R2E   : std_logic;
35   signal  input : std_logic_VECTOR (15 downto 0);
     signal  Q0    : std_logic_VECTOR (15 downto 0);
37   signal  Q1    : std_logic_VECTOR (15 downto 0);
     signal  Q2    : std_logic_VECTOR (15 downto 0);
39 BEGIN
       reg_0 : reg16 port map(input, clk, R0E, '0', Q0);
41     reg_1 : reg16 port map(input, clk, R1E, '0', Q1);
       reg_2 : reg16 port map(input, clk, R2E, '0', Q2);
43
     SetInput: process(clk, enable, SelRi, Ri)
45   BEGIN
       R0E <= '0';
47     R1E <= '0';
       R2E <= '0';
49     IF enable = '1' THEN
         case SelRi IS
51         WHEN "00" =>
             R0E <= '1';
53         WHEN "01" =>
             R1E <= '1';
55         WHEN "10" =>
             R2E <= '1';
57         WHEN others =>
             NULL; -- None of them are enabled
59       END CASE;
       END IF;
61   end process;

63   -- Select if 1 or 2 Bytes is to be written and if
     SetNumBytes: process(clk, Ri, SelRi, ByteInput, Sel8Bit, SelHighByte, Q0, Q1, Q2)
65   BEGIN
       IF Sel8Bit = '0' THEN
67       input <= Ri;
       ELSE
69       if SelHighByte = '1' THEN
           input(15 downto 8) <= ByteInput;
71         case SelRi IS
             WHEN "00" =>
73             input(7 downto 0) <= Q0(7 downto 0);
             WHEN "01" =>
75             input(7 downto 0) <= Q1(7 downto 0);
             WHEN others =>
```

```
77              input(7 downto 0) <= Q2(7 downto 0);
          END CASE;
79        else
          input(7 downto 0) <= ByteInput;
81        case SelRi IS
            WHEN "00" =>
83            input(15 downto 8) <= Q0(15 downto 8);
            WHEN "01" =>
85            input(15 downto 8) <= Q1(15 downto 8);
            WHEN others =>
87            input(15 downto 8) <= Q2(15 downto 8);
          END CASE;
89      END IF;
      END IF;
91  end process;

93  -- Set the output Ro
    WITH SelRo SELECT
95  Ro <= Q0 WHEN "00",
          Q1 WHEN "01",
97        Q2 WHEN others;
end Behavioral;
```

Listing 9: processor/cpu.vhd

```
1  -- Authors:
   --       Henry Jenkins, Joel Koh
3
   library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;
   --use IEEE.STD_LOGIC_ARITH.ALL;
7  --use IEEE.STD_LOGIC_UNSIGNED.ALL;

9  library work;
   use work.alu;
11 use work.cu;
   use work.ar;
13 use work.gpr;
   use work.sr;
15 use work.pc;

17
   ---- Uncomment the following library declaration if instantiating
19 ---- any Xilinx primitives in this code.
   --library UNISIM;
21 --use UNISIM.VComponents.all;

23 entity cpu is

25   PORT(
       -- instruction bus
27     inst_add  : out std_logic_vector(11 downto 0); -- Address lines.
       inst_data : in  std_logic_vector(15 downto 0); -- Data lines.
29     inst_req  : out std_logic;                     -- Pulled low to request bus
           usage.
       inst_ack  : in  std_logic;                     -- Pulled high to inform of
           request completion.
31     -- data bus
       data_add  : out    std_logic_vector(15 downto 0); -- Address lines.
33     data_line : inout std_logic_vector(7 downto 0);  -- Data lines.
       data_read : out    std_logic;                     -- High for a read request,
           low for a write request.
35     data_req  : out    std_logic;                     -- Pulled low to request bus
           usage.
       data_ack  : inout std_logic;                    -- Pulled high to inform of
           request completion.
37     -- extras
       clk       : in  std_logic;
39     reset     : in  std_logic
     );
41
   end cpu;
```

```
43
architecture cpu_arch of cpu is
45   component alu IS
       Port (f    : in    STD_LOGIC_VECTOR (3 downto 0);  -- Function (opcode)
47           rx   : in    STD_LOGIC_VECTOR (7 downto 0);  -- Input x (Rx)
             ry   : in    STD_LOGIC_VECTOR (7 downto 0);  -- Input y (Ry)
49           ro   : out   STD_LOGIC_VECTOR (7 downto 0);  -- Output Normaly (Ry)
             Cin  : in    STD_LOGIC;                      -- Carry in
51           sr   : out   STD_LOGIC_VECTOR (15 downto 0)); -- Status register out Z(0),
                 C(1), N(2)
     END component;
53   component ar is
       Port (clk         : in    STD_LOGIC;
55           enable      : in    STD_LOGIC;
             Sel8Bit     : in    STD_LOGIC;
57           SelHighByte : in    STD_LOGIC;
             ByteInput   : in    STD_LOGIC_VECTOR (7 downto 0);
59           SelRi       : in    STD_LOGIC_VECTOR (1 downto 0);   -- Select the address
                 register
             SelRo       : in    STD_LOGIC_VECTOR (1 downto 0);   -- Select the address
                 register
61           Ri          : in    STD_LOGIC_VECTOR (15 downto 0);  -- The input
             Ro          : out   STD_LOGIC_VECTOR (15 downto 0)); -- The output
63   END component;
     component cu IS
65   Port (reset       : in STD_LOGIC;                       -- '0' for reset
           clock       : in STD_LOGIC;                       -- clock
67
           alu_f       : out STD_LOGIC_VECTOR (3 downto 0);  -- Function
69         alu_Cin     : out STD_LOGIC;                      -- Carry in to ALU

71         -- General Purpose Registers
           gpr_InSel   : out STD_LOGIC;                      -- select the input path (0
                - cu, 1 - ALU)
73         gpr_en      : out STD_LOGIC;                      -- enable write to GPR
           gpr_SelRx   : out STD_LOGIC_VECTOR (2 downto 0);  -- select GPR output x
75         gpr_SelRy   : out STD_LOGIC_VECTOR (2 downto 0);  -- select GPR output y
           gpr_SelRi   : out STD_LOGIC_VECTOR (2 downto 0);  -- select GPR input
77         gpr_Ri      : out STD_LOGIC_VECTOR (7 downto 0);  -- input to GPR
           gpr_Rx      : in STD_LOGIC_VECTOR (7 downto 0);   -- output Rx from GPR
79         --gpr_Ry      : in STD_LOGIC_VECTOR (7 downto 0);   -- output Ry from GPR ,
               not used

81         -- Status Register
           sr_en       : out STD_LOGIC;                      -- enable write to SR
83         sr_reset    : out STD_LOGIC;                      -- reset SR
           sr_Ro       : in STD_LOGIC_VECTOR (15 downto 0);  -- output from SR
85         -- control unit doesnt write to SR, the ALU does

87         -- Program Counter
           pc_en       : out STD_LOGIC;                      -- enable write to PC
89         pc_reset    : out STD_LOGIC;                      -- reset PC
           pc_Ri       : out STD_LOGIC_VECTOR (15 downto 0); -- input to PC
91         pc_Ro       : in STD_LOGIC_VECTOR (15 downto 0);  -- output from PC

93         -- Address Registers
           ar_en       : out STD_LOGIC;                      -- enable write to AR
95         ar_SelRi    : out STD_LOGIC_VECTOR (1 downto 0);  -- select AR in
           ar_SelRo    : out STD_LOGIC_VECTOR (1 downto 0);  -- select AR out
97         ar_Ri       : out STD_LOGIC_VECTOR (15 downto 0); -- input to AR
           ar_Ro       : in STD_LOGIC_VECTOR (15 downto 0);  -- output from AR
99         ar_sel8Bit  : out STD_LOGIC;                      -- only write half the AR
           ar_selHByte : out STD_LOGIC;                      -- high or low half of the
               AR to write
101        ar_ByteIn   : out STD_LOGIC_VECTOR (7 downto 0);  -- 8 bit input to write
               half of AR

103        -- Instruction memory
           inst_add    : out STD_LOGIC_VECTOR (11 downto 0); -- Instruction address
105        inst_data   : in STD_LOGIC_VECTOR (15 downto 0);  -- Instruction data
           inst_req    : out STD_LOGIC;                      -- Request
107        inst_ack    : in STD_LOGIC;                       -- Instruction obtained
```

```
109         data_add    : out STD_LOGIC_VECTOR (15 downto 0); -- Data address
            data_data   : inout STD_LOGIC_VECTOR (7 downto 0);-- Data
111         data_read   : out STD_LOGIC;                      -- 1 for read, 0 for write
            data_req    : out STD_LOGIC;                      -- Request
113         data_ack    : in STD_LOGIC                        -- Data written to/ read
                from

115         );
    END component;
117 component gpr is
      Port (clk      : in   STD_LOGIC;
119           enable   : in   STD_LOGIC;
              SelRx    : in   STD_LOGIC_VECTOR (2 downto 0);  -- The Rx output selection
                  value
121           SelRy    : in   STD_LOGIC_VECTOR (2 downto 0);  -- The Ry output selection
                  value
              SelRi    : in   STD_LOGIC_VECTOR (2 downto 0);  -- The Ri input selection
                  value
123           SelIn    : in   STD_LOGIC;  -- Select where the input should be from the CU
                   or CDB
              RiCU     : in   STD_LOGIC_VECTOR (7 downto 0);  -- Input from the Control
                  Unit
125           RiCDB    : in   STD_LOGIC_VECTOR (7 downto 0);  -- Input from the Common
                  Data Bus
              Rx       : out  STD_LOGIC_VECTOR (7 downto 0);  -- The Rx output
127           Ry       : out  STD_LOGIC_VECTOR (7 downto 0)); -- The Ry output
    END component;
129 component sr is
      Port (clk      : in  STD_LOGIC;
131           enable   : in  STD_LOGIC;
              reset    : in  STD_LOGIC;
133           Ri       : in  STD_LOGIC_VECTOR (15 downto 0);  -- The input to the SR
              Ro       : out STD_LOGIC_VECTOR (15 downto 0)); -- The output from SR
135 END component;
    component pc is
137   Port (clk      : in  STD_LOGIC;
              enable   : in  STD_LOGIC;
139           reset    : in  STD_LOGIC;
              Ri       : in  STD_LOGIC_VECTOR (15 downto 0);  -- The input to the SR
141           Ro       : out STD_LOGIC_VECTOR (15 downto 0)); -- The output from SR
    END component;
143 signal alu_Cin    : std_logic;
    signal alu_f      : std_logic_vector(3 downto 0);
145 signal alu_rx     : std_logic_vector(7 downto 0);
    signal alu_ry     : std_logic_vector(7 downto 0);
147
    signal sr_reset   : std_logic;
149 signal sr_enable  : std_logic;
    signal sr_Ro      : std_logic_vector(15 downto 0);
151 signal sr_input   : std_logic_vector(15 downto 0);

153 signal ar_enable   : STD_LOGIC;                       -- enable write to AR
    signal ar_SelRi    : STD_LOGIC_VECTOR (1 downto 0);  -- select AR in
155 signal ar_SelRo    : STD_LOGIC_VECTOR (1 downto 0);  -- select AR out
    signal ar_Ri       : STD_LOGIC_VECTOR (15 downto 0); -- input to AR
157 signal ar_Ro       : STD_LOGIC_VECTOR (15 downto 0);  -- output from AR
    signal ar_sel8Bit  : STD_LOGIC;                      -- only write half the AR
159 signal ar_selHByte : STD_LOGIC;                      -- high or low half of the AR
       to write
    signal ar_ByteIn   : STD_LOGIC_VECTOR (7 downto 0);  -- 8 bit input to write half
       of AR
161
    signal pc_reset   : std_logic;
163 signal pc_enable  : std_logic;
    signal pc_Ri      : std_logic_vector(15 downto 0);
165 signal pc_Ro      : std_logic_vector(15 downto 0);

167 signal gpr_InSel  : std_logic;
    signal gpr_enable : std_logic;
169 signal gpr_SelRx  : std_logic_vector(2 downto 0);
    signal gpr_SelRy  : std_logic_vector(2 downto 0);
171 signal gpr_SelRi  : std_logic_vector(2 downto 0);
    signal gpr_RiCU   : std_logic_vector(7 downto 0);
```

```
173   signal gpr_RiCDB  : std_logic_vector(7 downto 0);
  begin
175
    a: alu port map(
177             f    => alu_f,
                rx   => alu_rx,
179             ry   => alu_ry,
                ro   => gpr_RiCDB,
181             Cin  => alu_Cin,
                sr   => sr_input
183           );
    c: cu port map(
185
                reset     => reset,-- '0' for reset
187             clock     => clk,-- clock

189             alu_f     => alu_f,-- Function
                alu_Cin   => alu_Cin, -- Carry into the ALU
191
                -- General Purpose Registers
193             gpr_InSel => gpr_InSel,-- select the input path (0 - cu, 1 - ALU)
                gpr_en    => gpr_enable,-- enable write to GPR
195             gpr_SelRx => gpr_SelRx,-- select GPR output x
                gpr_SelRy => gpr_SelRy,-- select GPR output y
197             gpr_SelRi => gpr_SelRi,-- select GPR input
                gpr_Ri    => gpr_RiCU,-- input to GPR
199             gpr_Rx    => alu_rx,-- Rx from GPR
                --gpr_Ry    => alu_ry,-- Ry from GPR
201
                -- Status Register
203             sr_en     => sr_enable,-- enable write to SR
                sr_reset  => sr_reset,-- reset SR
205             sr_Ro     => sr_Ro,-- output from SR
                -- control unit doesnt write to SR, the ALU does
207
                -- Program Counter
209             pc_en     => pc_enable,-- enable write to PC
                pc_reset  => pc_reset,-- reset PC
211             pc_Ri     => pc_Ri,-- input to PC
                pc_Ro     => pc_Ro,-- output from PC
213
                -- Address Registers
215             ar_en       => ar_enable,   -- enable write to AR
                ar_SelRi    => ar_SelRi,    -- select AR in
217             ar_SelRo    => ar_SelRo,    -- select AR out
                ar_sel8Bit  => ar_sel8Bit,
219             ar_selHByte => ar_selHByte,
                ar_ByteIn   => ar_ByteIn,
221             ar_Ri       => ar_Ri,       -- input to AR
                ar_Ro       => ar_Ro,       -- output from AR
223
                -- Instruction memory
225             inst_add  => inst_add ,-- Instruction address
                inst_data => inst_data,-- Instruction data
227             inst_req  => inst_req ,-- Request
                inst_ack  => inst_ack ,-- Instruction obtained
229
                data_add  => data_add ,-- Data address
231             data_data => data_line,-- Data
                data_read => data_read,-- 1 for read, 0 for write
233             data_req  => data_req ,-- Request
                data_ack  => data_ack  -- Data written to/ read from
235           );
    address : ar port map(
237             clk         => clk,
                enable      => ar_enable,
239             Sel8Bit     => ar_Sel8Bit,
                SelHighByte => ar_selHByte,
241             ByteInput   => ar_ByteIn,
                SelRi       => ar_SelRi,
243             SelRo       => ar_SelRo,
                Ri          => ar_Ri,
245             Ro          => ar_Ro
```

```
            );
g : gpr port map(
              clk    => clk,
              enable => gpr_enable,
              SelRx  => gpr_SelRx,
              SelRy  => gpr_SelRy,
              SelRi  => gpr_SelRi,
              SelIn  => gpr_InSel,
              RiCU   => gpr_RiCU,
              RiCDB  => gpr_RiCDB,
              Rx     => alu_rx,
              Ry     => alu_ry
            );
s : sr port map(
              clk    => clk,
              enable => sr_enable,
              reset  => sr_reset,
              Ri     => sr_input,
              Ro     => sr_Ro
              );
programcounter: pc port map(
              clk     => clk,
              enable  => pc_enable,
              reset   => pc_reset,
              Ri      => pc_Ri,
              Ro      => pc_Ro
            );
end cpu_arch;
```

Listing 10: processor/cu.vhd

```
-- Authors:
--       Henry Jenkins, Joel Koh

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_arith.all;
--use ieee.std_logic_unsigned.all;


library work;
--use work.fulladder;
--use work.cpu.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

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

entity cu is
  Port (reset       : in STD_LOGIC;                       -- '0' for reset
        clock       : in STD_LOGIC;                       -- clock

        alu_f       : out STD_LOGIC_VECTOR (3 downto 0);  -- Function
        alu_Cin     : out STD_LOGIC;                      -- Carry in to ALU

        -- General Purpose Registers
        gpr_InSel   : out STD_LOGIC;                      -- select the input path (0
            - cu, 1 - ALU)
        gpr_en      : out STD_LOGIC;                      -- enable write to GPR
        gpr_SelRx   : out STD_LOGIC_VECTOR (2 downto 0);  -- select GPR output x
        gpr_SelRy   : out STD_LOGIC_VECTOR (2 downto 0);  -- select GPR output y
        gpr_SelRi   : out STD_LOGIC_VECTOR (2 downto 0);  -- select GPR input
        gpr_Ri      : out STD_LOGIC_VECTOR (7 downto 0);  -- input to GPR
        gpr_Rx      : in STD_LOGIC_VECTOR (7 downto 0);   -- output Rx from GPR
        --gpr_Ry      : in STD_LOGIC_VECTOR (7 downto 0);   -- output Ry from GPR ,
            not used
```

```
41          -- Status Register
            sr_en       : out STD_LOGIC;                        -- enable write to SR
43          sr_reset    : out STD_LOGIC;                        -- reset SR
            sr_Ro       : in STD_LOGIC_VECTOR (15 downto 0);    -- output from SR
45          -- control unit doesnt write to SR, the ALU does

47          -- Program Counter
            pc_en       : out STD_LOGIC;                        -- enable write to PC
49          pc_reset    : out STD_LOGIC;                        -- reset PC
            pc_Ri       : out STD_LOGIC_VECTOR (15 downto 0); -- input to PC
51          pc_Ro       : in STD_LOGIC_VECTOR (15 downto 0);    -- output from PC

53          -- Address Registers
            ar_en       : out STD_LOGIC;                        -- enable write to AR
55          ar_SelRi    : out STD_LOGIC_VECTOR (1 downto 0);    -- select AR in
            ar_SelRo    : out STD_LOGIC_VECTOR (1 downto 0);    -- select AR out
57          ar_Ri       : out STD_LOGIC_VECTOR (15 downto 0); -- input to AR
            ar_Ro       : in STD_LOGIC_VECTOR (15 downto 0);    -- output from AR
59          ar_sel8Bit  : out STD_LOGIC;                        -- only write half the AR
            ar_selHByte : out STD_LOGIC;                        -- high or low half of the
                AR to write
61          ar_ByteIn   : out STD_LOGIC_VECTOR (7 downto 0);    -- 8 bit input to write
                half of AR

63          -- Instruction memory
            inst_add    : out STD_LOGIC_VECTOR (11 downto 0); -- Instruction address
65          inst_data   : in STD_LOGIC_VECTOR (15 downto 0);    -- Instruction data
            inst_req    : out STD_LOGIC;                        -- Request
67          inst_ack    : in STD_LOGIC;                         -- Instruction obtained

69          data_add    : out STD_LOGIC_VECTOR (15 downto 0); -- Data address
            data_data   : inout STD_LOGIC_VECTOR (7 downto 0);-- Data
71          data_read   : out STD_LOGIC;                        -- 1 for read, 0 for write
            data_req    : out STD_LOGIC;                        -- Request
73          data_ack    : in STD_LOGIC                          -- Data written to/ read
                from

75          );
    end cu;
77

79  architecture Behavioral of cu is
      component fulladder16 IS
81    Port (A    : in   STD_LOGIC_VECTOR(15 downto 0);
            B    : in   STD_LOGIC_VECTOR(15 downto 0);
83          Cin  : in   STD_LOGIC;
            Sum  : out  STD_LOGIC_VECTOR(15 downto 0);
85          Cout : out  STD_LOGIC
            );
87    end component;

89    type states is (reset_state, fetch, decode, execute);
      signal state      : states := reset_state;
91    signal next_state : states := reset_state;

93    signal opcode     : std_logic_vector(15 downto 0);  -- unprocessed instruction

95    -- Decoded data
      signal rx : std_logic_vector(2 downto 0);
97    signal ry : std_logic_vector(2 downto 0);
      signal ay : std_logic_vector(1 downto 0);
99
      -- Indicates what needs to be executed
101   signal write_gpr    : std_logic;
      signal write_sr     : std_logic;
103   signal write_pc     : std_logic;
      signal write_ar     : std_logic;
105   signal write_memory : std_logic;

107   -- full adders
      signal   A16         : std_logic_vector(15 downto 0);
109   signal   B16         : std_logic_vector(15 downto 0);
```

```
    signal   AdderCin16  : std_logic;
111 signal   Sum16       : std_logic_vector(15 downto 0);
    signal   AdderCout16 : std_logic;
113
    signal v    : STD_LOGIC_VECTOR(7 downto 0);  -- 8-bit immediate
115

117 BEGIN
    Adder16: fulladder16 port map(A16, B16, AdderCin16, Sum16, AdderCout16);
119
    -- Process instruction
121 -- Assumes all instructions are valid
    process(clock, state, opcode, gpr_Rx, sr_Ro, pc_Ro, ar_Ro, inst_data, inst_ack,
        data_data, data_ack,
123         rx, ry, ay, v, write_gpr, write_sr, write_pc, write_ar, write_memory)
    BEGIN
125   if rising_edge(clock) then
        case state is
127       when reset_state =>
            sr_reset <= '0';
129         pc_reset <= '0';
            next_state <= fetch;
131
          when fetch =>
133         sr_reset <= '1';
            pc_reset <= '1';
135
            gpr_en <= '0';
137         sr_en <= '0';
            pc_en <= '0';
139         ar_en <= '0';

141         write_gpr <= '0';
            write_sr <= '0';
143         write_pc <= '0';
            write_ar <= '0';
145         write_memory <= '0';

147         inst_add <= pc_Ro(11 downto 0);
            if inst_ack = '0' then
149           inst_req <= '1';
            else
151           opcode <= inst_data;
              inst_req <= '0';
153
              -- increment program counter
155           AdderCin16 <= '1';
              A16 <= PC_Ro;
157           B16 <= "0000000000000000";
              pc_Ri <= Sum16;
159           pc_en <= '1';

161           next_state <= decode;
            end if;
163       when decode =>
            pc_en <= '0';
165
            -- ALU
167         if opcode(15) = '0' and opcode(10) = '0' and not opcode(14 downto 11) =
                "0010" then

169           ry <= opcode(7 downto 5);
              rx <= opcode(2 downto 0);
171
              gpr_SelRy <= ry;
173           gpr_SelRx <= rx;
              gpr_SelRi <= ry;
175           gpr_InSel <= '1';
              alu_f <= opcode(14 downto 11);
177           alu_Cin <= sr_Ro(1);  -- Carry

179           if not opcode(14 downto 11) = "0110" then -- CMP doesnt write to gpr, all
                  others do
```

```
            write_gpr <= '1';
          end if;

          write_sr <= '1';
          next_state <= execute;

        -- Branching
        elsif opcode(11 downto 10) = "11" then

          v <= "00000000";  -- initialise v

          if opcode(15) = '1' then
            case opcode(14 downto 12) is
              when "000" => -- BEQ
                if sr_Ro(0) = '1' then -- Z=1
                  v <= opcode(9 downto 2);
                end if;
              when "001" => -- BNE
                if sr_Ro(0) = '0' then -- Z=0
                  v <= opcode(9 downto 2);
                end if;
              when "010" => -- BLT
                if sr_Ro(0) = '0' and sr_Ro(2) = '1' then -- Z=0 and N=1
                  v <= opcode(9 downto 2);
                end if;
              when "011" => -- BGT
                if sr_Ro(0) = '0' and sr_Ro(2) = '0' then -- Z=0 and N=0
                  v <= opcode(9 downto 2);
                end if;
              when "100" => -- BC
                if sr_Ro(1) = '1' then -- C=1
                  v <= opcode(9 downto 2);
                end if;
              when "101" => -- BNC
                if sr_Ro(1) = '0' then -- C=0
                  v <= opcode(9 downto 2);
                end if;
              when "110" => -- RJMP
                v <= opcode(9 downto 2);

              when others =>
                v <= "00000000";
            end case;

            -- PC <- PC + v
            AdderCin16 <= '0';
            A16 <= PC_Ro;
            B16 <= "00000000" & v;
            pc_Ri <= Sum16;

          elsif opcode(15 downto 12) = "0111" then -- JMP
            ay <= opcode(6 downto 5);

            -- PC <- ay
            ar_SelRo <= ay;
            pc_Ri <= ar_Ro;
          else
            -- should not reach here
            pc_Ri <= pc_Ro; -- no change
          end if;

          write_pc <= '1';
          next_state <= execute;


        -- Addressing
        else
          gpr_Insel <= '0';

          case opcode(12 downto 10) is

            when "001" => -- Load
              if opcode(15) = '1' then  -- immediate
```

```
253                   rx <= '0' & opcode(1 downto 0);
                      v <= opcode(9 downto 2);
255
                      -- rx <- v
257                   gpr_SelRi <= rx;
                      gpr_Ri <= v;
259                   write_gpr <= '1';
                      next_state <= execute;
261                 else                        -- direct
                      rx <= opcode(2 downto 0);
263                   ay <= opcode(6 downto 5);

265                   -- rx <- [ay]
                      gpr_SelRi <= rx;
267                   ar_selRo <= ay;
                      data_add <= ar_Ro;
269                   data_read <= '1';
                      if data_ack = '0' then  -- request data
271                     data_req <= '1';
                      else                    -- data obtained
273                     gpr_Ri <= data_data;
                        data_req <= '0';
275                     write_gpr <= '1';

277                     case opcode(14 downto 13) is
                          when "01" =>            -- auto increment
279                         AdderCin16 <= '1';
                            A16 <= ar_Ro;
281                         B16 <= "0000000000000000";
                            ar_selRi <= ay;
283                         ar_sel8bit <= '0';
                            ar_Ri <= Sum16;
285                         write_ar <= '1';
                          when "10" =>            -- auto decrement
287                         AdderCin16 <= '0';
                            A16 <= ar_Ro;
289                         B16 <= "1111111111111111";
                            ar_selRi <= ay;
291                         ar_sel8bit <= '0';
                            ar_Ri <= Sum16;
293                         write_ar <= '1';
                          when others =>
295                         -- do nothing
                        end case;
297                     next_state <= execute;
                      end if;
299                 end if;

301               when "101" => -- Store
                    if opcode(15) = '1' then  -- immediate
303                   ay <= opcode(1 downto 0);
                      v <= opcode(9 downto 2);
305
                      -- [ay] <- v
307                   ar_selRo <= ay;
                      data_add <= ar_Ro;
309                   data_read <= '0';
                      data_data <= v;
311                   write_memory <= '1';
                      next_state <= execute;
313                 else                        -- direct
                      rx <= opcode(2 downto 0);
315                   ay <= opcode(6 downto 5);

317                   -- [ay] <- rx
                      gpr_selRx <= rx;
319                   ar_selRo <= ay;
                      data_add <= ar_Ro;
321                   data_read <= '0';
                      data_data <= gpr_Rx;
323                   write_memory <= '1';

325                   case opcode(14 downto 13) is
```

```
                    when "01" =>            -- auto increment
327                   AdderCin16 <= '1';
                      A16 <= ar_Ro;
329                   B16 <= "0000000000000000";
                      ar_selRi <= ay;
331                   ar_sel8bit <= '0';
                      ar_Ri <= Sum16;
333                   write_ar <= '1';
                    when "10" =>            -- auto decrement
335                   AdderCin16 <= '0';
                      A16 <= ar_Ro;
337                   B16 <= "1111111111111111";
                      ar_selRi <= ay;
339                   ar_sel8bit <= '0';
                      ar_Ri <= Sum16;
341                   write_ar <= '1';
                    when others =>
343                   -- do nothing
                  end case;
345               next_state <= execute;
                end if;
347
              when "100" => -- Move
349             if opcode(9) = '1' then
                  rx <= opcode(2 downto 0);
351               ay <= opcode(6 downto 5);

353               -- ayn <- rx
                  gpr_selRx <= rx;
355               ar_selRi <= ay;
                  ar_sel8bit <= '1';
357               ar_ByteIn <= gpr_Rx;

359               if opcode(8) = '1' then     -- high
                    ar_selHByte <= '1';
361               else                        -- low
                    ar_selHByte <= '0';
363               end if;

365               write_ar <= '1';
                  next_state <= execute;
367
                elsif opcode(4) = '1' then
369               rx <= opcode(7 downto 5);
                  ay <= opcode(1 downto 0);
371
                  -- rx <- ayn
373               gpr_selRi <= rx;
                  ar_selRo <= ay;
375
                  if opcode(3) = '1' then     -- high
377                 gpr_Ri <= ar_Ro(15 downto 8);
                  else                        -- low
379                 gpr_Ri <= ar_Ro(7 downto 0);
                  end if;
381
                  write_gpr <= '1';
383               next_state <= execute;

385             else
                  rx <= opcode(2 downto 0);
387               ry <= opcode(7 downto 5);

389               -- ry <- rx
                  gpr_SelRx <= rx;
391               gpr_SelRi <= ry;
                  gpr_Ri <= gpr_Rx;
393
                  write_gpr <= '1';
395               next_state <= execute;

397             end if;
```

```
399
                when others =>
401               -- should not reach here

403           end case;

405         end if;

407       when execute =>
            if write_memory = '1' then
409           if data_ack = '0' then  -- request write
                data_req <= '1';
411           else                    -- data written
                data_req <= '0';
413       gpr_en <= write_gpr;
          sr_en <= write_sr;
415       pc_en <= write_pc;
          ar_en <= write_ar;
417             next_state <= fetch;
            end if;
419       else
        gpr_en <= write_gpr;
421     sr_en <= write_sr;
        pc_en <= write_pc;
423     ar_en <= write_ar;
            next_state <= fetch;
425       end if;

427       when others =>
            -- shouldnt reach here
429         next_state <= reset_state;
      end case;
431   end if;
  end process;
433
  process(clock, reset, next_state)
435 BEGIN
    if reset = '0' then
437   state <= reset_state;
    elsif rising_edge(clock) then
439   state <= next_state;
    end if;
441 end process;

443 end Behavioral;
```

Listing 11: processor/fulladder.vhd

```
  -- Authors:
2 --       Henry Jenkins, Joel Koh

4 library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
6
  entity fulladder is
8   Port (Ax    : in   STD_LOGIC;
          Bx    : in   STD_LOGIC;
10        Ci    : in   STD_LOGIC;
          Sx    : out  STD_LOGIC;
12        Co    : out  STD_LOGIC
          );
14 end fulladder;

16
  architecture arch_fulladder of fulladder is
18 BEGIN
    process(Ax, Bx, Ci)
20  BEGIN
      Sx <= (Ax XOR Bx) XOR Ci;
22    Co <= (Ax and Bx) or (Ax and Ci) OR (Bx AND Ci);
    end process;
24 end arch_fulladder;
```

```
26 --------------------------------------------------------------------------------

28 library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
30
   entity fulladder8 is
32   Port (A    : in   STD_LOGIC_VECTOR( 7 downto 0);
           B    : in   STD_LOGIC_VECTOR( 7 downto 0);
34         Cin  : in   STD_LOGIC;
           Sum  : out  STD_LOGIC_VECTOR( 7 downto 0);
36         Cout : out  STD_LOGIC
           );
38 end fulladder8;

40 architecture arch_fulladder8 of fulladder8 is
     component fulladder IS
42   Port (Ax   : in   STD_LOGIC;
           Bx   : in   STD_LOGIC;
44         Ci   : in   STD_LOGIC;
           Sx   : out  STD_LOGIC;
46         Co   : out  STD_LOGIC
           );
48   end component;
     signal Carry   : std_logic_vector(8 downto 0);
50 BEGIN
     Carry(0) <= Cin;
52
       FA0: fulladder PORT MAP(A(0), B(0), Carry(0), Sum(0), Carry(1));
54     FA1: fulladder PORT MAP(A(1), B(1), Carry(1), Sum(1), Carry(2));
       FA2: fulladder PORT MAP(A(2), B(2), Carry(2), Sum(2), Carry(3));
56     FA3: fulladder PORT MAP(A(3), B(3), Carry(3), Sum(3), Carry(4));
       FA4: fulladder PORT MAP(A(4), B(4), Carry(4), Sum(4), Carry(5));
58     FA5: fulladder PORT MAP(A(5), B(5), Carry(5), Sum(5), Carry(6));
       FA6: fulladder PORT MAP(A(6), B(6), Carry(6), Sum(6), Carry(7));
60     FA7: fulladder PORT MAP(A(7), B(7), Carry(7), Sum(7), Carry(8));

62   Cout <= Carry(8);
   end arch_fulladder8;
64
   --------------------------------------------------------------------------------
66
   library IEEE;
68 use IEEE.STD_LOGIC_1164.ALL;

70 entity fulladder16 is
     Port (A    : in   STD_LOGIC_VECTOR( 15 downto 0);
72         B    : in   STD_LOGIC_VECTOR( 15 downto 0);
           Cin  : in   STD_LOGIC;
74         Sum  : out  STD_LOGIC_VECTOR( 15 downto 0);
           Cout : out  STD_LOGIC
76         );
   end fulladder16;
78
   architecture arch_fulladder16 of fulladder16 is
80   component fulladder IS
     Port (Ax   : in   STD_LOGIC;
82         Bx   : in   STD_LOGIC;
           Ci   : in   STD_LOGIC;
84         Sx   : out  STD_LOGIC;
           Co   : out  STD_LOGIC
86         );
     end component;
88   signal Carry   : std_logic_vector(16 downto 0);
   BEGIN
90   Carry(0) <= Cin;

92     FA0:  fulladder PORT MAP(A(0), B(0), Carry(0), Sum(0), Carry(1));
       FA1:  fulladder PORT MAP(A(1), B(1), Carry(1), Sum(1), Carry(2));
94     FA2:  fulladder PORT MAP(A(2), B(2), Carry(2), Sum(2), Carry(3));
       FA3:  fulladder PORT MAP(A(3), B(3), Carry(3), Sum(3), Carry(4));
96     FA4:  fulladder PORT MAP(A(4), B(4), Carry(4), Sum(4), Carry(5));
       FA5:  fulladder PORT MAP(A(5), B(5), Carry(5), Sum(5), Carry(6));
```

```
98      FA6:  fulladder PORT MAP(A(6), B(6), Carry(6), Sum(6), Carry(7));
        FA7:  fulladder PORT MAP(A(7), B(7), Carry(7), Sum(7), Carry(8));
100     FA8:  fulladder PORT MAP(A(8), B(8), Carry(8), Sum(8), Carry(9));
        FA9:  fulladder PORT MAP(A(9), B(9), Carry(9), Sum(9), Carry(10));
102     FA10: fulladder PORT MAP(A(10), B(10), Carry(10), Sum(10), Carry(11));
        FA11: fulladder PORT MAP(A(11), B(11), Carry(11), Sum(11), Carry(12));
104     FA12: fulladder PORT MAP(A(12), B(12), Carry(12), Sum(12), Carry(13));
        FA13: fulladder PORT MAP(A(13), B(13), Carry(13), Sum(13), Carry(14));
106     FA14: fulladder PORT MAP(A(14), B(14), Carry(14), Sum(14), Carry(15));
        FA15: fulladder PORT MAP(A(15), B(15), Carry(15), Sum(15), Carry(16));
108
    Cout <= Carry(16);
110 end arch_fulladder16;
```

Listing 12: processor/gpr.vhd

```
    -- Authors:
2   --       Henry Jenkins, Joel Koh

4   library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
6
    library work;
8   use work.reg8;

10  entity gpr is
      Port (clk       : in    STD_LOGIC;
12          enable    : in    STD_LOGIC;
            SelRx     : in    STD_LOGIC_VECTOR (2 downto 0);  -- The Rx output selection
                value
14          SelRy     : in    STD_LOGIC_VECTOR (2 downto 0);  -- The Ry output selection
                value
            SelRi     : in    STD_LOGIC_VECTOR (2 downto 0);  -- The Ri input selection
                value
16          SelIn     : in    STD_LOGIC;  -- Select where the input should be from the CU
                or CDB
            RiCU      : in    STD_LOGIC_VECTOR (7 downto 0);  -- Input from the Control
                Unit
18          RiCDB     : in    STD_LOGIC_VECTOR (7 downto 0);  -- Input from the Common Data
                 Bus
            Rx        : out   STD_LOGIC_VECTOR (7 downto 0);  -- The Rx output
20          Ry        : out   STD_LOGIC_VECTOR (7 downto 0)); -- The Ry output
    end gpr;
22

24  architecture gpr_arch of gpr is
      component reg8 IS
26      port(I      : in  std_logic_vector(7 downto 0);
             clock  : in  std_logic;
28           enable : in  std_logic;
             reset  : in  std_logic;
30           Q      : out std_logic_vector(7 downto 0)
            );
32    end component;

34    signal  reset: std_logic := '0';
      signal  input: std_logic_VECTOR (7 downto 0);
36    signal  R0E  : std_logic;  -- Enable signals
      signal  R1E  : std_logic;
38    signal  R2E  : std_logic;
      signal  R3E  : std_logic;
40    signal  R4E  : std_logic;
      signal  R5E  : std_logic;
42    signal  R6E  : std_logic;
      signal  R7E  : std_logic;
44    signal  Q0   : std_logic_VECTOR (7 downto 0);
      signal  Q1   : std_logic_VECTOR (7 downto 0);
46    signal  Q2   : std_logic_VECTOR (7 downto 0);
      signal  Q3   : std_logic_VECTOR (7 downto 0);
48    signal  Q4   : std_logic_VECTOR (7 downto 0);
      signal  Q5   : std_logic_VECTOR (7 downto 0);
50    signal  Q6   : std_logic_VECTOR (7 downto 0);
```

```
    signal  Q7   : std_logic_VECTOR (7 downto 0);
BEGIN
    reg_0 : reg8 port map(input, clk, R0E, reset, Q0);
    reg_1 : reg8 port map(input, clk, R1E, reset, Q1);
    reg_2 : reg8 port map(input, clk, R2E, reset, Q2);
    reg_3 : reg8 port map(input, clk, R3E, reset, Q3);
    reg_4 : reg8 port map(input, clk, R4E, reset, Q4);
    reg_5 : reg8 port map(input, clk, R5E, reset, Q5);
    reg_6 : reg8 port map(input, clk, R6E, reset, Q6);
    reg_7 : reg8 port map(input, clk, R7E, reset, Q7);

  -- Select where the input should come from
    SelectInput: process(SelIn, RiCDB, RiCU)
    BEGIN
      IF SelIn = '1' THEN
          input <= RiCDB;
      ELSE
        input <= RiCU;
      END IF;
    END process;

  -- Set Ri the input
  SetInput: process(clk, enable, SelRi)
  BEGIN
    R0E <= '0';
    R1E <= '0';
    R2E <= '0';
    R3E <= '0';
    R4E <= '0';
    R5E <= '0';
    R6E <= '0';
    R7E <= '0';
   IF enable = '1' THEN
      case SelRi IS
        WHEN "000" =>
          R0E <= '1';
        WHEN "001" =>
          R1E <= '1';
        WHEN "010" =>
          R2E <= '1';
        WHEN "011" =>
          R3E <= '1';
        WHEN "100" =>
          R4E <= '1';
        WHEN "101" =>
          R5E <= '1';
        WHEN "110" =>
          R6E <= '1';
        WHEN "111" =>
          R7E <= '1';
        WHEN others =>
          NULL; -- None of them are enabled
      end case;
   END IF;
  end process;

  -- Set the Rx output
  WITH SelRx SELECT
  Rx <= Q0 WHEN "000",
        Q1 WHEN "001",
        Q2 WHEN "010",
        Q3 WHEN "011",
        Q4 WHEN "100",
        Q5 WHEN "101",
        Q6 WHEN "110",
        Q7 WHEN others;

-- Set the Ry output
  WITH SelRy SELECT
  Ry <= Q0 WHEN "000",
        Q1 WHEN "001",
        Q2 WHEN "010",
        Q3 WHEN "011",
```

```
124         Q4 WHEN "100",
            Q5 WHEN "101",
126         Q6 WHEN "110",
            Q7 WHEN others;
128
    end gpr_arch;
```

Listing 13: processor/reg.vhd

```
    -- Authors:
 2  --       Henry Jenkins, Joel Koh

 4  library ieee;
    use ieee.std_logic_1164.all;
 6
    entity reg8 is
 8  port(I      : in  std_logic_vector(7 downto 0);
         clock  : in  std_logic;
10       enable : in  std_logic;
         reset  : in  STD_LOGIC;
12       Q      : out std_logic_vector(7 downto 0)
        );
14  end reg8;

16  architecture behv of reg8 is
    begin
18
      process(I, clock, enable, reset)
20    begin
        IF reset = '1' THEN
22        Q <= (others => '0');
        ELSIF rising_edge(clock) then
24        if enable = '1' then
            Q <= I;
26        end if;
        end if;
28
      end process;
30
    end behv;
32
    --------------------------------------------------------------------------------
34
    library ieee;
36  use ieee.std_logic_1164.all;

38  entity reg16 is
    port(I      : in  std_logic_vector(15 downto 0);
40       clock  : in  std_logic;
         enable : in  std_logic;
42       reset  : in  STD_LOGIC;
         Q      : out std_logic_vector(15 downto 0)
44      );
    end reg16;
46
    architecture behv of reg16 is
48  begin

50    process(I, clock, enable, reset)
      begin
52      IF reset = '1' THEN
          Q <= (others => '0');
54      ELSIF rising_edge(clock) then
          if enable = '1' then
56          Q <= I;
          end if;
58      end if;

60    end process;

62  end behv;
```

Listing 14: processor/spr.vhd

```
-- Authors:
--       Henry Jenkins, Joel Koh

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library work;
use work.reg16;

entity sr is
  Port (clk      : in  STD_LOGIC;
        enable   : in  STD_LOGIC;
        reset    : in  STD_LOGIC;
        Ri       : in  STD_LOGIC_VECTOR (15 downto 0);  -- The input to the SR
        Ro       : out STD_LOGIC_VECTOR (15 downto 0)); -- The output from SR
end sr;

architecture sr_arch of sr is
  component reg16 IS
  port(I      : in  std_logic_vector(15 downto 0);
       clock  : in  std_logic;
       enable : in  std_logic;
       reset  : in  STD_LOGIC;
       Q      : out std_logic_vector(15 downto 0)
     );
  end component;
BEGIN
  reg_sr : reg16 port map(Ri, clk, enable, reset, Ro);
end sr_arch;

--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity pc is
  Port (clk      : in  STD_LOGIC;
        enable   : in  STD_LOGIC;
        reset    : in  STD_LOGIC;
        Ri       : in  STD_LOGIC_VECTOR (15 downto 0);  -- The input to the SR
        Ro       : out STD_LOGIC_VECTOR (15 downto 0)); -- The output from SR
end pc;


architecture pc_arch of pc is
  component reg16 IS
  port(I      : in  std_logic_vector(15 downto 0);
       clock  : in  std_logic;
       enable : in  std_logic;
       reset  : in  STD_LOGIC;
       Q      : out std_logic_vector(15 downto 0)
     );
  end component;
BEGIN
  reg_pc : reg16 port map(Ri, clk, enable, reset, Ro);
end pc_arch;
```

MMU

Listing 15: mmu/control_unit.vhd

```
1  -- Authors:
   --       Wim Looman, Forrest McKerchar
3
   library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;

7  library work;
   use work.mmu_types.all;
9
   entity mmu_control_unit is
11   port (
       eoc              : in  std_logic; -- High on muart has finished collecting data
13     eot              : in  std_logic; -- High on muart has finished transmitting
       ready            : in  std_logic; -- High if the muart is ready for new transfer
15     data_read        : in  std_logic; -- High if the cpu requests a read, else write
       data_req         : in  std_logic; -- Low to start a transfer
17     data_add_0       : in  std_logic; -- High for memory address, else IO
       inst_req         : in  std_logic; -- Low to start a transfer
19     fr               : in  std_logic; -- Input headers fetch request bit
       inst_or_data_in  : in  std_logic; -- Input headers inst or data bit
21     rw               : in  std_logic; -- Input headers read/!write bit
       write            : out std_logic; -- Pulled high to start muart writing data
23     inst_or_data_out : out std_logic; -- Ouput headers inst or data bit
       inst_ack         : out std_logic; -- Idles high, pulled low when data ready
25     data_ack         : inout std_logic; -- Idles 'Z', high when data not ready,
           pulled low when data ready
       muart_input      : out muart_input_state; -- Signal connected to muart input
27     muart_output     : out muart_output_state; -- Signal connected to muart output
       clk              : in  std_logic
29   );
   end mmu_control_unit;
31
   architecture mmu_control_unit_arch of mmu_control_unit is
33   component data_control_unit is
       port (
35       eoc          : in  std_logic; -- High on muart has finished collecting data
         eot          : in  std_logic; -- High on muart has finished transmitting
37       ready        : in  std_logic; -- High if the muart is ready for new transfer
         data_read    : in  std_logic; -- High if the cpu requests a read, else write
39       data_req     : in  std_logic; -- Low to start a transfer
         data_add_0   : in  std_logic; -- High for memory address, else IO
41       write        : out std_logic; -- Pulled high to start muart writing data.
         data_ack     : inout std_logic; -- Idles 'Z', high when data not ready, pulled
             low when data ready
43       muart_input  : out muart_input_state; -- Signal connected to muart input
         muart_output : out muart_output_state; -- Signal connected to muart output
45       clk          : in  std_logic
       );
47   end component;

49   component inst_control_unit is
       port (
51       eoc          : in  std_logic; -- High on muart has finished collecting data
         eot          : in  std_logic; -- High on muart has finished transmitting
53       ready        : in  std_logic; -- High if the muart is ready for new transfer
         inst_req     : in  std_logic; -- Low to start a transfer
55       write        : out std_logic; -- Pulled high to start muart writing data
         inst_or_data : out std_logic; -- Ouput headers inst or data bit
57       inst_ack     : out std_logic; -- Idles high, pulled low when data ready
         muart_input  : out muart_input_state; -- Signal connected to muart input
59       muart_output : out muart_output_state; -- Signal connected to muart output
         clk          : in  std_logic
61     );
   end component;
63
   signal data_write, inst_write, inst_inst_or_data_out : std_logic;
65 signal data_muart_input, inst_muart_input : muart_input_state;
   signal data_muart_output, inst_muart_output : muart_output_state;
```

```
67 begin
     data_cu : data_control_unit port map (
69     eoc,
       eot,
71     ready,
       data_read,
73     data_req,
       data_add_0,
75     data_write,
       data_ack,
77     data_muart_input,
       data_muart_output,
79     clk
     );
81   inst_cu : inst_control_unit port map (
       eoc,
83     eot,
       ready,
85     inst_req,
       inst_write,
87     inst_inst_or_data_out,
       inst_ack,
89     inst_muart_input,
       inst_muart_output,
91     clk
     );
93
     inst_or_data_out <= inst_inst_or_data_out ;
95   write            <= inst_write or data_write;
     muart_input  <= inst_muart_input  when inst_inst_or_data_out = '1' else
97                   data_muart_input;
     muart_output <= inst_muart_output when inst_inst_or_data_out = '1' else
99                   data_muart_output;
   end mmu_control_unit_arch;
```

Listing 16: mmu/data_control_unit.vhd

```
   -- Authors:
2  --       Wim Looman, Forrest McKerchar

4  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
6
   library work;
8  use work.mmu_types.all;

10 entity data_control_unit is
     port (
12       eoc          : in  std_logic; -- High on muart has finished collecting data
         eot          : in  std_logic; -- High on muart has finished transmitting
14       ready        : in  std_logic; -- High if the muart is ready for new transfer
         data_read    : in  std_logic; -- High if the cpu requests a read, else write
16       data_req     : in  std_logic; -- Low to start a transfer
         data_add_0   : in  std_logic; -- High for memory address, else IO
18       write        : out std_logic; -- Pulled high to start muart writing data.
         data_ack     : inout std_logic; -- Idles 'Z', high when data not ready, pulled
             low when data ready
20       muart_input  : out muart_input_state; -- Signal connected to muart input
         muart_output : out muart_output_state; -- Signal connected to muart output
22       clk          : in  std_logic
     );
24 end data_control_unit;

26 architecture data_control_unit_arch of data_control_unit is
     type m_state_type is (
28     idle,
       send_header, send_add_high, send_add_low, send_data,
30     get_header,  get_add_high,  get_add_low,  get_data,
       finished
32   );
     type state_type          is (idle, get_data, wait_clear);
34   type read_state_type     is (idle, wait_data, read_data, pause, finished);
```

```
    type transmit_state_type is (idle, set_data, trans_data, pause, finished);
36
    signal state,             next_state             : state_type          := idle;
38  signal get_state,         next_get_state         : m_state_type        := idle;
    signal reader_state,      next_reader_state      : read_state_type     := idle;
40  signal transmitter_state, next_transmitter_state : transmit_state_type := idle;
  begin
42  data_fsm : process(state, data_req, clk) begin
      if (rising_edge(clk)) then
44      case state is
          when idle =>
46          if (data_req = '0' and data_add_0 = '1') then
              next_state <= get_data;
48          end if;

50        when get_data =>
            if (get_state = finished) then
52            next_state <= wait_clear;
            end if;
54
          when wait_clear =>
56          if (data_req = '1') then
              next_state <= idle;
58          end if;

60        when others =>
            NULL;
62      end case;
      end if;
64  end process data_fsm;

66  get_data_fsm : process(state, clk) begin
      if rising_edge(clk) then
68      if state = get_data then
          case get_state is
70          when idle =>
              next_get_state <= send_header;
72
            when send_header =>
74            if transmitter_state = finished then
                next_get_state <= send_add_low;
76            end if;

78          when send_add_low =>
              if transmitter_state = finished then
80              next_get_state <= send_add_high;
              end if;
82
            when send_add_high =>
84            if transmitter_state = finished then
                if data_read = '1' then
86                next_get_state <= get_header;
                else
88                next_get_state <= send_data;
                end if;
90            end if;

92          when send_data =>
              if transmitter_state = finished then
94              next_get_state <= finished;
              end if;
96
            when get_header =>
98            if reader_state = finished then
                next_get_state <= get_add_low;
100           end if;

102         when get_add_low =>
              if reader_state = finished then
104             next_get_state <= get_add_high;
              end if;
106
            when get_add_high =>
```

```
108             if reader_state = finished then
                  next_get_state <= get_data;
110             end if;

112           when get_data =>
                if reader_state = finished then
114               next_get_state <= finished;
                end if;
116
              when finished =>
118             next_get_state <= idle;

120           when others =>
                NULL;
122         end case;
        end if;
124   end if;
    end process get_data_fsm;
126
    transmit_fsm : process(clk, get_state, transmitter_state, eot) begin
128   if rising_edge(clk) then
        if ((get_state = send_header) or
130         (get_state = send_add_low) or
            (get_state = send_add_high) or
132         (get_state = send_data)) then
          case transmitter_state is
134         when idle =>
            if ready = '1' and eot = '0' then
136             next_transmitter_state <= set_data;
              end if;
138
            when set_data =>
140           next_transmitter_state <= trans_data;

142         when trans_data =>
              next_transmitter_state <= pause;
144
            when pause =>
146           if eot = '1' then
                next_transmitter_state <= finished;
148           end if;

150         when finished =>
              next_transmitter_state <= idle;
152
            when others =>
154           NULL;
          end case;
156     end if;
      end if;
158 end process transmit_fsm;

160 read_fsm : process(clk, get_state, reader_state, eoc) begin
      if rising_edge(clk) then
162     if ((get_state = get_header) or
            (get_state = get_add_low) or
164         (get_state = get_add_high) or
            (get_state = get_data)) then
166       case reader_state is
            when idle =>
168           next_reader_state <= wait_data;

170         when wait_data =>
              if eoc = '1' then
172             next_reader_state <= read_data;
              end if;
174
            when read_data =>
176           next_reader_state <= pause;

178         when pause =>
              if eoc = '0' then
180             next_reader_state <= finished;
```

```
            end if;

          when finished =>
            next_reader_state <= idle;

          when others =>
            NULL;
        end case;
      end if;
    end if;
  end process read_fsm;

  switch_states : process(
    clk,
    next_state,
    next_get_state,
    next_reader_state,
    next_transmitter_state
  ) begin
    if rising_edge(clk) then
      state              <= next_state;
      get_state          <= next_get_state;
      reader_state       <= next_reader_state;
      transmitter_state  <= next_transmitter_state;
    end if;
  end process switch_states;

  -- Outputs
  with state select
    data_ack <= '1' when wait_clear,
                'Z' when idle,
                '0' when others;

  with transmitter_state select
    write <= '1' when trans_data,
             '0' when others;

  muart_input <= idle          when transmitter_state /= set_data      else
                 idle          when transmitter_state /= trans_data    else
                 header        when get_state          = send_header   else
                 data_add_high when get_state          = send_add_high else
                 data_add_low  when get_state          = send_add_low  else
                 data_data     when get_state          = send_data     else
                 idle;

  muart_output <= clear_data when state         = idle       else
                  idle       when reader_state /= read_data  else
                  header     when get_state     = get_header else
                  data_data  when get_state     = get_data   else
                  idle;
end data_control_unit_arch;
```

Listing 17: mmu/header_builder.vhd

```
-- Author:
--      Forrest McKerchar

-- builds a header to feed into the RS-232 link

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library work;

entity header_builder is
  port (
    read_write : in  std_logic; -- 1 = read, 0 = write
    inst_data  : in  std_logic; -- 1 = inst, 0 = data
    header     : out std_logic_vector(7 downto 0)
  );
end header_builder;
```

```
19 architecture header_builder_arch of header_builder is
   begin
21   header(7) <= read_write or inst_data; -- reading from or writing to
                                             -- memory? (can't write
23                                           -- instructions)
     header(6) <= '0'; -- reserved
25   header(5) <= '0'; -- reserved
     header(4) <= '0'; -- reserved
27   header(3) <= '0'; -- diagnostic
     header(2) <= '0'; -- diagnostic;
29   header(1) <= '0'; --  1 if data is being retrieved from RS-232 link
     header(0) <= inst_data; -- instruction data or data data?
31 end header_builder_arch;
```

Listing 18: mmu/header_decoder.vhd

```
   -- Author:
2  --       Forrest McKerchar

4  -- decodes a header received from the RS-232 link

6  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
8
   library work;
10
   entity header_decoder is
12   port (
       read_write    : out  std_logic; -- 1 = read, 0 = write
14     fetch_request : out std_logic;
       inst_data     : out  std_logic; -- 1 = inst, 0 = data
16     header        : in std_logic_vector(7 downto 0)
     );
18 end header_decoder;

20 architecture header_decoder_arch of header_decoder is
   begin
22   read_write <= header(7); -- reading or writing? (should be 1 in this case)
     fetch_request <= header(1); -- fetch request? (should be 1n this case)
24   inst_data <= header(0); -- instruction data or data data?
   end header_decoder_arch;
```

Listing 19: mmu/inst_control_unit.vhd

```
   -- Authors:
2  --       Wim Looman, Forrest McKerchar

4  library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
6
   library work;
8  use work.mmu_types.all;

10 entity inst_control_unit is
     port (
12      eoc           : in  std_logic; -- High on muart has finished collecting data
        eot           : in  std_logic; -- High on muart has finished transmitting
14      ready         : in  std_logic; -- High if the muart is ready for new transfer
        inst_req      : in  std_logic; -- Low to start a transfer
16      write         : out std_logic; -- Pulled high to start muart writing data
        inst_or_data  : out std_logic; -- Ouput headers inst or data bit
18      inst_ack      : out std_logic; -- Idles high, pulled low when data ready
        muart_input   : out muart_input_state; -- Signal connected to muart input
20      muart_output  : out muart_output_state; -- Signal connected to muart output
        clk           : in  std_logic
22   );
   end inst_control_unit;
24
   architecture inst_control_unit_arch of inst_control_unit is
26   type m_state_type is (
```

```
    idle,
28  send_header,   send_add_high, send_add_low,
    get_header,    get_add_high,  get_add_low,
30  get_data_high, get_data_low,  finished
  );
32  type state_type          is (idle, get_data, wait_clear);
  type read_state_type     is (idle, wait_data, read_data, pause, finished);
34  type transmit_state_type is (idle, set_data, trans_data, pause, finished);

36  signal state,              next_state             : state_type          := idle;
  signal get_state,          next_get_state         : m_state_type        := idle;
38  signal reader_state,       next_reader_state      : read_state_type     := idle;
  signal transmitter_state, next_transmitter_state : transmit_state_type := idle;
40 begin
  inst_fsm : process(state, inst_req, clk) begin
42  case state is
      when idle =>
44      if (inst_req = '0') then
          next_state <= get_data;
46      end if;

48    when get_data =>
        if (get_state = finished) then
50        next_state <= wait_clear;
        end if;
52
      when wait_clear =>
54      if (inst_req = '1') then
          next_state <= idle;
56      end if;

58    when others =>
        NULL;
60  end case;
  end process inst_fsm;
62
  get_inst_fsm : process(state, clk) begin
64  if state = get_data then
      case get_state is
66      when idle =>
          next_get_state <= send_header;
68
        when send_header =>
70        if transmitter_state = finished then
            next_get_state <= send_add_low;
72        end if;

74      when send_add_low =>
          if transmitter_state = finished then
76          next_get_state <= send_add_high;
          end if;
78
        when send_add_high =>
80        if transmitter_state = finished then
            next_get_state <= get_header;
82        end if;

84      when get_header =>
          if reader_state = finished then
86          next_get_state <= get_add_low;
          end if;
88
        when get_add_low =>
90        if reader_state = finished then
            next_get_state <= get_add_high;
92        end if;

94      when get_add_high =>
          if reader_state = finished then
96          next_get_state <= get_data_low;
          end if;
98
        when get_data_low =>
```

```
100           if reader_state = finished then
                next_get_state <= get_data_high;
102           end if;

104         when get_data_high =>
              if reader_state = finished then
106             next_get_state <= finished;
              end if;
108
            when finished =>
110           next_get_state <= idle;

112         when others =>
              NULL;
114       end case;
      end if;
116   end process get_inst_fsm;

118   transmit_fsm : process(clk, get_state, transmitter_state, eot) begin
      if ((get_state = send_header) or
120       (get_state = send_add_low) or
          (get_state = send_add_high)) then
122     case transmitter_state is
          when idle =>
124         if ready = '1' and eot = '0' then
              next_transmitter_state <= set_data;
126         end if;

128       when set_data =>
            next_transmitter_state <= trans_data;
130
          when trans_data =>
132         next_transmitter_state <= pause;

134       when pause =>
            if eot = '1' then
136           next_transmitter_state <= finished;
            end if;
138
          when finished =>
140         next_transmitter_state <= idle;

142       when others =>
            NULL;
144     end case;
      end if;
146   end process transmit_fsm;

148   read_fsm : process(clk, get_state, reader_state, eoc) begin
      if ((get_state = get_header) or
150       (get_state = get_add_low) or
          (get_state = get_add_high) or
152       (get_state = get_data_low) or
          (get_state = get_data_high)) then
154     case reader_state is
          when idle =>
156         next_reader_state <= wait_data;

158       when wait_data =>
            if eoc = '1' then
160           next_reader_state <= read_data;
            end if;
162
          when read_data =>
164         next_reader_state <= pause;

166       when pause =>
            if eoc = '0' then
168           next_reader_state <= finished;
            end if;
170
          when finished =>
172         next_reader_state <= idle;
```

```
174         when others =>
              NULL;
176       end case;
      end if;
178   end process read_fsm;

180   switch_states : process(
        clk,
182     next_state,
        next_get_state,
184     next_reader_state,
        next_transmitter_state) begin
186     if rising_edge(clk) then
          state <= next_state;
188       get_state <= next_get_state;
          reader_state <= next_reader_state;
190       transmitter_state <= next_transmitter_state;
        end if;
192   end process switch_states;

194   -- Outputs
      with state select
196     inst_ack <= '1' when wait_clear,
                  '0' when others;
198
      with state select
200     inst_or_data <= '0' when idle,
                      '1' when others;
202
      with transmitter_state select
204     write <= '1' when trans_data,
               '0' when others;
206

208   muart_input <= idle          when transmitter_state /= set_data      else
                     idle          when transmitter_state /= trans_data    else
210                  header        when get_state          = send_header   else
                     inst_add_high when get_state          = send_add_high else
212                  inst_add_low  when get_state          = send_add_low  else
                     idle;
214
      muart_output <= idle           when reader_state /= read_data     else
216                   header         when get_state     = get_header    else
                      inst_data_high when get_state     = get_data_high else
218                   inst_data_low  when get_state     = get_data_low  else
                      idle;
220 end inst_control_unit_arch;
```

Listing 20: mmu/mmu.vhd

```
  -- Authors:
2 --       Wim Looman, Forrest McKerchar

4 library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
6
  library work;
8 use work.mmu_types.all;
  use work.mmu_control_unit;
10 use work.header_builder;
  use work.header_decoder;
12 use work.reg8;

14 use work.minimal_uart_core;

16   entity mmu_main is
       port (
18       -- instruction bus
         inst_add  : in  std_logic_vector(11 downto 0); -- Address lines.
20       inst_data : out std_logic_vector(15 downto 0); -- Data lines.
```

```
    inst_req  : in  std_logic;                    -- Pulled low to request bus
        usage.
    inst_ack  : out std_logic;                    -- Pulled high to inform of
        request completion.
    -- data bus
    data_add  : in    std_logic_vector(15 downto 0); -- Address lines.
    data_line : inout std_logic_vector(7 downto 0);  -- Data lines.
    data_read : in    std_logic;                    -- High for a read request,
        low for a write request.
    data_req  : in    std_logic;                    -- Pulled low to request bus
        usage.
    data_ack  : inout std_logic;                    -- Pulled high to inform of
        request completion.
    -- extras
    clk          : in  std_logic;
    receive_pin  : in  std_logic;
    transfer_pin : out std_logic
  );
end mmu_main;

architecture mmu_arch of mmu_main is
  component mmu_control_unit is
    port (
      eoc              : in  std_logic; -- High on muart has finished collecting
          data
      eot              : in  std_logic; -- High on muart has finished transmitting
      ready            : in  std_logic; -- High if the muart is ready for new
          transfer
      data_read        : in  std_logic; -- High if the cpu requests a read, else
          write
      data_req         : in  std_logic; -- Low to start a transfer
      data_add_0       : in  std_logic; -- High for memory address, else IO
      inst_req         : in  std_logic; -- Low to start a transfer
      fr               : in  std_logic; -- Input headers fetch request bit
      inst_or_data_in  : in  std_logic; -- Input headers inst or data bit
      rw               : in  std_logic; -- Input headers read/!write bit
      write            : out std_logic; -- Pulled high to start muart writing data
      inst_or_data_out : out std_logic; -- Ouput headers inst or data bit
      inst_ack         : out std_logic; -- Idles high, pulled low when data ready
      data_ack         : inout std_logic; -- Idles 'Z', high when data not ready,
          pulled low when data ready
      muart_input      : out muart_input_state; -- Signal connected to muart input
      muart_output     : out muart_output_state; -- Signal connected to muart
          output
      clk              : in  std_logic
    );
  end component;

  component header_builder is
    port (
      read_write : in  std_logic; -- 1 = read, 0 = write
      inst_data  : in  std_logic; -- 1 = inst, 0 = data
      header     : out std_logic_vector(7 downto 0)
    );
  end component;

  component header_decoder is
    port (
      read_write    : out  std_logic; -- 1 = read, 0 = write
      fetch_request : out std_logic;
      inst_data     : out  std_logic; -- 1 = inst, 0 = data
      header        : in std_logic_vector(7 downto 0)
    );
  end component;

  component minimal_uart_core is
    port(
      clock : in    std_logic;
      eoc   : out   std_logic;
      outp  : inout std_logic_vector(7 downto 0) := "ZZZZZZZZ";
      rxd   : in    std_logic;
      txd   : out   std_logic;
      eot   : out   std_logic;
```

```
84          inp    : in     std_logic_vector(7 downto 0);
            ready  : out    std_logic;
86          wr     : in     std_logic
          );
88      end component;

90      component reg8 IS
          port(
92          I      : in  std_logic_vector(7 downto 0);
            clock  : in  std_logic;
94          enable : in  std_logic;
            reset  : in  std_logic;
96          Q      : out std_logic_vector(7 downto 0)
          );
98      end component;

100
        signal eoc           : std_logic;
102     signal eot           : std_logic;
        signal ready         : std_logic;
104     signal fr            : std_logic;
        signal inst_or_data_in : std_logic;
106     signal rw            : std_logic;

108     signal write         : std_logic;
        signal inst_or_data_out : std_logic;
110     signal muart_input   : muart_input_state;
        signal muart_output : muart_output_state;
112
        signal muart_out : std_logic_vector(7 downto 0);
114     signal muart_in  : std_logic_vector(7 downto 0);
        signal header_in   : std_logic_vector(7 downto 0);
116     signal header_out  : std_logic_vector(7 downto 0);
        signal inst_data_high_enable : std_logic;
118     signal inst_data_low_enable : std_logic;
        signal data_data_enable : std_logic;
120     signal data_line_tri : std_logic_vector(7 downto 0);

122     begin
        muart : minimal_uart_core port map (
124       clk,
          eoc,
126       muart_out,
          receive_pin,
128       transfer_pin,
          eot,
130       muart_in,
          ready,
132       write
        );
134     cu  : mmu_control_unit port map (
          eoc,
136       eot,
          ready,
138       data_read,
          data_req,
140       data_add(0),
          inst_req,
142       fr,
          inst_or_data_in,
144       rw,
          write,
146       inst_or_data_out,
          inst_ack,
148       data_ack,
          muart_input,
150       muart_output,
          clk
152     );
        hb  : header_builder port map (
154       data_read,
          inst_or_data_out,
156       header_out
```

```
    );
158 hd  : header_decoder port map (
      rw,
160   fr,
      inst_or_data_in,
162   header_in
    );
164 idh : reg8 port map (
      muart_out,
166   clk,
      inst_data_high_enable,
168   '0',
      inst_data(15 downto 8)
170 );
    idl : reg8 port map (
172   muart_out,
      clk,
174   inst_data_low_enable,
      '0',
176   inst_data(7  downto 0)
    );
178 dd  : reg8 port map (
      muart_out,
180   clk,
      data_data_enable,
182   '0',
      data_line_tri
184 );

186 with muart_input select
      muart_in <=  header_out                       when header,
188                "0000" & inst_add(11 downto 8)  when inst_add_high,
                   inst_add(7  downto 0)           when inst_add_low,
190                '0' & data_add(15 downto 9)     when data_add_high,
                   data_add(8  downto 1)           when data_add_low,
192                data_line                       when data_data,
                   (others => '0')                 when others;
194
    route_output : process(muart_output, muart_out) begin
196   header_in <= (others => '0');
      inst_data_high_enable <= '0';
198   inst_data_low_enable <= '0';
      data_data_enable <= '0';
200   case muart_output is
        when header =>
202       header_in <= muart_out;

204     when inst_data_high =>
          inst_data_high_enable <= '1';
206
        when inst_data_low =>
208       inst_data_low_enable <= '1';

210     when data_data =>
          data_data_enable <= '1';
212     when others =>
          NULL;
214   end case;
    end process;
216
    data_line <= data_line_tri when data_add(0) = '1' and data_read = '1' else
218              (others => 'Z');

220 end mmu_arch;
```

Listing 21: mmu/mmu_types.vhd

```
  -- Authors:
2 --      Wim Looman, Forrest McKerchar

4 library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
```

```
6
  package mmu_types is
8    type muart_input_state is (header, inst_add_high, inst_add_low, data_add_high,
         data_add_low, data_data, idle);
     type muart_output_state is (header, inst_data_high, inst_data_low, data_data,
         clear_data, idle);
10 end mmu_types;
```

Listing 22: mmu/muart/BRG.vhd

```
  --*************************************************************************
2 --*  Minimal UART ip core                                                  *
  --* Author: Arao Hayashida Filho        arao@medinovacao.com.br           *
4 --*                                                                        *
  --*************************************************************************
6 --*                                                                        *
  --* Copyright (C) 2009 Arao Hayashida Filho                               *
8 --*                                                                        *
  --* This source file may be used and distributed without                  *
10 --* restriction provided that this copyright statement is not            *
  --* removed from the file and that any derivative work contains           *
12 --* the original copyright notice and the associated disclaimer.         *
  --*                                                                       *
14 --* This source file is free software; you can redistribute it           *
  --* and/or modify it under the terms of the GNU Lesser General            *
16 --* Public License as published by the Free Software Foundation;         *
  --* either version 2.1 of the License, or (at your option) any            *
18 --* later version.                                                        *
  --*                                                                       *
20 --* This source is distributed in the hope that it will be               *
  --* useful, but WITHOUT ANY WARRANTY; without even the implied            *
22 --* warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR               *
  --* PURPOSE.  See the GNU Lesser General Public License for more          *
24 --* details.                                                              *
  --*                                                                       *
26 --* You should have received a copy of the GNU Lesser General            *
  --* Public License along with this source; if not, download it            *
28 --* from http://www.opencores.org/lgpl.shtml                              *
  --*                                                                       *
30 --*************************************************************************

32 library ieee;
  use ieee.std_logic_1164.all;
34 use ieee.std_logic_arith.all;
  use ieee.std_logic_unsigned."+";
36
  entity br_generator is
38   generic (divider_width: integer := 16);
     port (
40     clock      : in  std_logic;
       rx_enable  : in  std_logic;
42     clk_txd    : out std_logic;
       tx_enable  : in  std_logic;
44     clk_serial : out std_logic
     );
46 end br_generator;

48 architecture principal of br_generator is
     -- change the following constant to your desired baud rate
50   -- one hz equal to one bit per second
     signal   count_brg     : std_logic_vector(divider_width - 1 downto 0) := (others =>
         '0');
52   signal   count_brg_txd : std_logic_vector(divider_width - 1 downto 0) := (others =>
         '0');
     constant brdvd         : std_logic_vector(divider_width - 1 downto 0) := x"0516";
         -- 38400 bps @ 50MHz
54
     begin
56     txd : process (clock)
       begin
58       if (rising_edge(clock)) then
           if (count_brg_txd = brdvd) then
```

```
60            clk_txd       <= '1';
              count_brg_txd <= (others => '0');
62          elsif (tx_enable = '1') then
              clk_txd       <= '0';
64            count_brg_txd <= count_brg_txd + 1;
            else
66            clk_txd       <= '0';
              count_brg_txd <= (others => '0');
68          end if;
          end if;
70      end process txd;

72      rxd : process (clock)
        begin
74        if (rising_edge(clock)) then
            if (count_brg=brdvd) then
76            count_brg   <= (others => '0');
              clk_serial <= '1';
78          elsif (rx_enable = '1') then
              count_brg   <= count_brg+1;
80            clk_serial <= '0';
            else
82            count_brg   <= '0' & brdvd(divider_width - 1 downto 1);
              clk_serial <= '0';
84          end if;
          end if;
86      end process rxd;
    end principal;
```

Listing 23: mmu/muart/serial.vhd

```
1  --*****************************************************************************
   --*  Minimal UART ip core                                                     *
3  --* Author: Arao Hayashida Filho        arao@medinovacao.com.br               *
   --*                                                                           *
5  --*****************************************************************************
   --*                                                                           *
7  --* Copyright (C) 2009 Arao Hayashida Filho                                   *
   --*                                                                           *
9  --* This source file may be used and distributed without                      *
   --* restriction provided that this copyright statement is not                 *
11 --* removed from the file and that any derivative work contains               *
   --* the original copyright notice and the associated disclaimer.              *
13 --*                                                                           *
   --* This source file is free software; you can redistribute it                *
15 --* and/or modify it under the terms of the GNU Lesser General                *
   --* Public License as published by the Free Software Foundation;              *
17 --* either version 2.1 of the License, or (at your option) any                *
   --* later version.                                                            *
19 --*                                                                           *
   --* This source is distributed in the hope that it will be                    *
21 --* useful, but WITHout ANY WARRANTY; without even the implied                 *
   --* warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR                   *
23 --* PURPOSE.  See the GNU Lesser General Public License for more              *
   --* details.                                                                  *
25 --*                                                                           *
   --* You should have received a copy of the GNU Lesser General                 *
27 --* Public License along with this source; if not, download it                *
   --* from http://www.opencores.org/lgpl.shtml                                  *
29 --*                                                                           *
   --*****************************************************************************
31
   library IEEE;
33 use IEEE.STD_LOGIC_1164.ALL;

35 entity minimal_uart_core is
     port(
37     clock : in    std_logic;
       eoc   : out   std_logic;
39     outp  : inout std_logic_vector(7 downto 0) := "ZZZZZZZZ";
       rxd   : in    std_logic;
41     txd   : out   std_logic;
```

```
    eot   : out   std_logic;
    inp   : in    std_logic_vector(7 downto 0);
    ready : out   std_logic;
    wr    : in    std_logic
  );
end minimal_uart_core;

architecture principal of minimal_uart_core  is
  type state is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9);
  signal clk_serial       : std_logic := '0';
  signal start            : std_logic := '0';
  signal eocs, eoc1, eoc2 : std_logic := '0';
  signal rx_ck_enable     : std_logic := '0';
  signal receiving        : std_logic := '0';
  signal transmitting     : std_logic := '0';
  signal clk_txd          : std_logic := '0';
  signal txds             : std_logic := '1';
  signal eots             : std_logic := '0';
  signal inpl             : std_logic_vector(7 downto 0) := x"00";
  signal data             : std_logic_vector(7 downto 0) := x"00";
  signal atual_state, next_state, atual_state_txd, next_state_txd: state := s0;
  signal tx_enable        : std_logic := '0';
  signal tx_ck_enable     : std_logic := '0';

  component br_generator
    port (
      clock      : in  std_logic;
      rx_enable  : in  std_logic;
      clk_txd    : out std_logic;
      tx_enable  : in  std_logic;
      clk_serial : out std_logic
    );
  end component;

  begin
    ready <= not(tx_enable);
    brg : br_generator port map (clock, rx_ck_enable, clk_txd, tx_ck_enable,
        clk_serial);
    rx_ck_enable <= start or receiving;
    tx_ck_enable <= tx_enable or transmitting;

    start_detect : process(rxd, eocs)
    begin
      if (eocs = '1') then
        start <= '0';
      elsif (falling_edge(rxd)) then
        start <= '1';
      end if;
    end process start_detect;

    rxd_states : process (clk_serial)
    begin
      if (rising_edge(clk_serial)) then
        atual_state <= next_state;
      end if;
    end process rxd_states;

    rxd_state_machine : process(start, atual_state)
    begin
      if (start = '1' or receiving = '1') then
        case atual_state is
          when s0 =>
            eocs <= '0';
            if (start = '1') then
              next_state <= s1;
              receiving  <= '1';
            else
              next_state <= s0;
              receiving  <= '0';
            end if;

          when s1 =>
            receiving  <= '1';
```

```
            eocs       <= '0';
            next_state <= s2;

          when s2   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s3;

          when s3   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s4;

          when s4   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s5;

          when s5   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s6;

          when s6   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s7;

          when s7   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s8;

          when s8   =>
            receiving  <= '1';
            eocs       <= '0';
            next_state <= s9;

          when s9   =>
            receiving  <= '1';
            eocs       <= '1';
            next_state <= s0;

          when others =>
            null;

        end case;
      end if;
    end process rxd_state_machine;

    rxd_shift : process(clk_serial)
    begin
      if (rising_edge(clk_serial)) then
        if (eocs = '0') then
          data <= rxd & data(7 downto 1);
        end if;
      end if;
    end process rxd_shift;

    process (clock)
    begin
      if (rising_edge(clock)) then
        eoc <= eocs;
      end if;
    end process;

    process(atual_state)
    begin
      if (atual_state=s9) then
        outp <= data;
      end if;
    end process;
```

```
187     txd_states : process(clk_txd)
        begin
189       if (rising_edge(clk_txd)) then
            atual_state_txd <= next_state_txd;
191       end if;
        end process txd_states;
193
        txd_state_machine : process(atual_state_txd, tx_enable)
195     begin
          case atual_state_txd is
197         when s0 =>
              inpl <= inp;
199           eots <= '0';
              if (tx_enable = '1') then
201             txds           <= '0';
                transmitting   <= '1';
203             next_state_txd <= s1;
              else
205             txds           <= '1';
                transmitting   <= '0';
207             next_state_txd <= s0;
              end if;
209
            when s1 =>
211           txds           <= inpl(0);
              eots           <= '0';
213           transmitting   <= '1';
              next_state_txd <= s2;
215
            when s2 =>
217           txds           <= inpl(1);
              eots           <= '0';
219           transmitting   <= '1';
              next_state_txd <= s3;
221
            when s3 =>
223           txds           <= inpl(2);
              eots           <= '0';
225           transmitting   <= '1';
              next_state_txd <= s4;
227
            when s4 =>
229           txds           <= inpl(3);
              eots           <= '0';
231           transmitting   <= '1';
              next_state_txd <= s5;
233
            when s5 =>
235           txds           <= inpl(4);
              eots           <= '0';
237           transmitting   <= '1';
              next_state_txd <= s6;
239
            when s6 =>
241           txds           <= inpl(5);
              eots           <= '0';
243           transmitting   <= '1';
              next_state_txd <= s7;
245
            when s7 =>
247           txds           <= inpl(6);
              eots           <= '0';
249           transmitting   <= '1';
              next_state_txd <= s8;
251
            when s8 =>
253           txds           <= inpl(7);
              eots           <= '0';
255           transmitting   <= '1';
              next_state_txd <= s9;
257
            when s9 =>
259           txds           <= '1';
```

```
            eots           <= '1';
            transmitting   <= '1';
            next_state_txd <= s0;

          when others =>
            null;

      end case;
    end process txd_state_machine;

    tx_start:process (clock, wr,  eots)
    begin
      if (eots = '1') then
        tx_enable <= '0';
      elsif (falling_edge(clock)) then
        if (wr = '1') then
          tx_enable <= '1';
        end if;
      end if;
    end process tx_start;

    eot<=eots;

    process (clock)
    begin
      if (rising_edge(clock)) then
        txd <= txds;
      end if;
    end process;

end principal ;
```

Test Benchs

Listing 24: processor/alu_tb.vhd

```
-- Authors:
--      Henry Jenkins, Joel Koh

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--  A testbench has no ports.
entity alu_tb is
  end alu_tb;

architecture behav of alu_tb is
  --  Declaration of the component that will be instantiated.
  component alu
   Port (f   : in   STD_LOGIC_VECTOR (3 downto 0);  -- Function (opcode)
         rx  : in   STD_LOGIC_VECTOR (7 downto 0);  -- Input x (Rx)
         ry  : in   STD_LOGIC_VECTOR (7 downto 0);  -- Input y (Ry)
         ro  : out  STD_LOGIC_VECTOR (7 downto 0);  -- Output Normaly (Ry)
         Cin : in   STD_LOGIC;                      -- Carry in
         sr  : out  STD_LOGIC_VECTOR (15 downto 0)); -- Status register out Z(0),
             C(1), N(2)
  end component;
  --  Specifies which entity is bound with the component.
  for alu_0: alu use entity work.alu;
    signal f          : STD_LOGIC_VECTOR (3 downto 0);
    signal rx, ry, ro : STD_LOGIC_VECTOR (7 downto 0);
    signal Cin        : STD_LOGIC;
    signal sr         : STD_LOGIC_VECTOR (15 downto 0);
  begin
    --  Component instantiation.
    alu_0: alu port map (f => f, rx => rx, ry => ry, ro => ro, Cin => Cin, sr => sr);

    --  This process does the real job.
    process
    type pattern_type is record
      f          : STD_LOGIC_VECTOR (3 downto 0);
      rx, ry     : STD_LOGIC_VECTOR (7 downto 0);
      ro         : STD_LOGIC_VECTOR (7 downto 0);
      Cin        : STD_LOGIC;
      sr         : STD_LOGIC_VECTOR (15 downto 0);
    end record;
  --  The patterns to apply.
  type pattern_array is array (natural range <>) of pattern_type;
  constant patterns : pattern_array :=
  -- f       rx          ry          ro          Cin   sr
  (("0001", "00000000", "00000000", "00000000", '0', "0000000000000001"), --AND tests
       - 1ns
   ("0001", "00000001", "00000001", "00000001", '0', "0000000000000000"), --AND tests
   ("0001", "00000000", "00000001", "00000000", '0', "0000000000000001"), --AND tests
   ("0001", "10101010", "10101010", "10101010", '0', "0000000000000100"), --AND tests
   ("0001", "01010101", "01010101", "01010101", '0', "0000000000000000"), --AND tests
       - 5ns
   ("0001", "11111111", "00000000", "00000000", '0', "0000000000000001"), --AND tests
   ("0001", "11111111", "11111111", "11111111", '0', "0000000000000100"), --AND tests
   ("0001", "00000000", "01010101", "00000000", '0', "0000000000000001"), --AND tests
   ("0001", "00000000", "10101010", "00000000", '0', "0000000000000001"), --AND tests
   ("0001", "11111111", "01010101", "01010101", '0', "0000000000000000"), --AND tests
       - 10 ns
   ("0001", "11111111", "10101010", "10101010", '0', "0000000000000100"), --AND tests
   ("0001", "10000011", "10110010", "10000010", '0', "0000000000000100"), --AND tests
   ("0001", "00000011", "00110010", "00000010", '0', "0000000000000000"), --AND tests

   ("0011", "00000000", "00000000", "00000000", '0', "0000000000000001"), --OR tests
       - 14 ns
   ("0011", "00000001", "00000001", "00000001", '0', "0000000000000000"), --OR tests
   ("0011", "00000000", "00000001", "00000001", '0', "0000000000000000"), --OR tests
   ("0011", "10101010", "10101010", "10101010", '0', "0000000000000100"), --OR tests
   ("0011", "01010101", "01010101", "01010101", '0', "0000000000000000"), --OR tests
   ("0011", "11111111", "00000000", "11111111", '0', "0000000000000100"), --OR tests
```

```
64    ("0011", "11111111", "11111111", "11111111", '0', "0000000000000100"), --OR tests
          - 20 ns
      ("0011", "00000000", "01010101", "01010101", '0', "0000000000000000"), --OR tests
66    ("0011", "00000000", "10101010", "10101010", '0', "0000000000000100"), --OR tests
      ("0011", "11111111", "01010101", "11111111", '0', "0000000000000100"), --OR tests
68    ("0011", "11111111", "10101010", "11111111", '0', "0000000000000100"), --OR tests
      ("0011", "10000011", "10110010", "10110011", '0', "0000000000000100"), --OR tests
          - 25 ns
70    ("0011", "00000011", "00110010", "00110011", '0', "0000000000000000"), --OR tests

72    ("0101", "00000000", "00000000", "11111111", '0', "0000000000000100"), --NOT tests
          - ry should not matter
      ("0101", "00000001", "00000001", "11111110", '0', "0000000000000100"), --NOT tests
74    ("0101", "00000000", "00000001", "11111111", '0', "0000000000000100"), --NOT tests
      ("0101", "10101010", "10101010", "01010101", '0', "0000000000000000"), --NOT tests
          - 30 ns
76    ("0101", "01010101", "01010101", "10101010", '0', "0000000000000100"), --NOT tests
      ("0101", "11111111", "00000000", "00000000", '0', "0000000000000001"), --NOT tests
78    ("0101", "11111111", "11111111", "00000000", '0', "0000000000000001"), --NOT tests
      ("0101", "00000000", "01010101", "11111111", '0', "0000000000000100"), --NOT tests
80    ("0101", "00000000", "10101010", "11111111", '0', "0000000000000100"), --NOT tests
          - 35 ns
      ("0101", "11111111", "01010101", "00000000", '0', "0000000000000001"), --NOT tests
82    ("0101", "11111111", "10101010", "00000000", '0', "0000000000000001"), --NOT tests
      ("0101", "10000011", "10110010", "01111100", '0', "0000000000000000"), --NOT tests
84    ("0101", "00000011", "00110010", "11111100", '0', "0000000000000100"), --NOT tests
          - 39 ns

86    ("0111", "00000000", "00000000", "00000000", '0', "0000000000000001"), --XOR tests
          - 40 ns
      ("0111", "00000001", "00000001", "00000000", '0', "0000000000000001"), --XOR tests
88    ("0111", "00000000", "00000001", "00000001", '0', "0000000000000000"), --XOR tests
      ("0111", "10101010", "10101010", "00000000", '0', "0000000000000001"), --XOR tests
90    ("0111", "01010101", "01010101", "00000000", '0', "0000000000000001"), --XOR tests
      ("0111", "11111111", "00000000", "11111111", '0', "0000000000000100"), --XOR tests
          - 45 ns
92    ("0111", "11111111", "11111111", "00000000", '0', "0000000000000001"), --XOR tests
      ("0111", "00000000", "01010101", "01010101", '0', "0000000000000000"), --XOR tests
94    ("0111", "00000000", "10101010", "10101010", '0', "0000000000000100"), --XOR tests
      ("0111", "11111111", "01010101", "10101010", '0', "0000000000000100"), --XOR tests
96    ("0111", "11111111", "10101010", "01010101", '0', "0000000000000000"), --XOR tests
          - 50 ns
      ("0111", "10000011", "10110010", "00110001", '0', "0000000000000000"), --XOR tests
98    ("0111", "00000011", "00110010", "00110001", '0', "0000000000000000")  --XOR tests
    );
100 begin
    --  Check each pattern.
102   for i in patterns'range loop
      --  Set the inputs.
104     Cin <= patterns(i).Cin;
      f <= patterns(i).f;
106     rx <= patterns(i).rx;
      ry <= patterns(i).ry;
108     --  Wait for the results.
      wait for 1 ns;
110     --  Check the outputs.
      assert ro = patterns(i).ro
112     report "bad output register value" severity error;
      assert sr = patterns(i).sr
114     report "bad status register value" severity error;
      assert sr(0) = patterns(i).sr(0)
116     report " *Zero is incorrect" severity error;
      assert sr(1) = patterns(i).sr(1)
118     report " *Carry is incorrect" severity error;
      assert sr(2) = patterns(i).sr(2)
120     report " *Negitive is incorrect" severity error;
    end loop;
122   assert false report "end of test" severity note;
    --  Wait forever; this will finish the simulation.
124   wait;
  end process;
126 end behav;
```

Listing 25: processor/fulladder_tb.vhd

```
-- Authors:
--       Henry Jenkins, Joel Koh

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--  A testbench has no ports.
entity fulladder8_tb is
  end fulladder8_tb;

architecture behav of fulladder8_tb is
  --  Declaration of the component that will be instantiated.
  component fulladder8
  Port (A    : in   STD_LOGIC_VECTOR( 7 downto 0);
        B    : in   STD_LOGIC_VECTOR( 7 downto 0);
        Cin  : in   STD_LOGIC;
        Sum  : out  STD_LOGIC_VECTOR( 7 downto 0);
        Cout : out  STD_LOGIC
        );
  end component;
  --  Specifies which entity is bound with the component.
  for fulladder8_0: fulladder8 use entity work.fulladder8;
    signal A,B,Sum    : STD_LOGIC_VECTOR (7 downto 0);
    signal Cin,Cout   : STD_LOGIC;
  begin
    --  Component instantiation.
    fulladder8_0: fulladder8 port map (A => A, B => B, Cin => Cin, Sum => Sum, Cout
        => Cout);

    --  This process does the real job.
    process
    type pattern_type is record
      A    :  STD_LOGIC_VECTOR( 7 downto 0);
      B    :  STD_LOGIC_VECTOR( 7 downto 0);
      Cin  :  STD_LOGIC;
      Sum  :  STD_LOGIC_VECTOR( 7 downto 0);
      Cout :  STD_LOGIC;
    end record;
  --  The patterns to apply.
  type pattern_array is array (natural range <>) of pattern_type;
  constant patterns : pattern_array :=
  -- A           B           Cin   Sum      Cout
  (("00000000", "00000000", '0', "00000000", '0'), --AND tests - 1ns
   ("11111111", "11111111", '1', "11111111", '1'), --AND tests
   ("00000000", "00000000", '1', "00000001", '0'), --AND tests
   ("00000000", "11111111", '0', "11111111", '0'), --AND tests
   ("11111111", "00000000", '0', "11111111", '0'), --AND tests
   ("11111111", "00000000", '1', "00000000", '1'), --AND tests
   ("10101010", "01010101", '0', "11111111", '0'), --AND tests
   ("10101010", "01010101", '1', "00000000", '1'), --AND tests
   ("11111111", "11111111", '0', "11111110", '1')  --XOR tests
  );
begin
  --  Check each pattern.
  for i in patterns'range loop
    --  Set the inputs.
    A   <= patterns(i).A;
    B   <= patterns(i).B;
    Cin <= patterns(i).Cin;
    --  Wait for the results.
    wait for 1 ns;
    --  Check the outputs.
    assert Sum = patterns(i).Sum
    report "The sum check failed" severity error;
    assert Cout = patterns(i).Cout
    report "The carry out is wrong" severity error;
  end loop;
  assert false report "end of test" severity note;
  --  Wait forever; this will finish the simulation.
  wait;
end process;
```

```
71 end behav;
```

Listing 26: processor/spr_tb.vhd

```
   -- Authors:
 2 --       Henry Jenkins, Joel Koh

 4 library ieee;
   use ieee.std_logic_1164.all;
 6 --use ieee.std_logic_unsigned.all;
   --use ieee.std_logic_arith.all;
 8
   entity spr_TB is      -- entity declaration
10   end spr_TB;

12 architecture TB of spr_TB is

14   component sr
     Port (clk      : in   STD_LOGIC;
16         enable   : in   STD_LOGIC;                          -- Enable write
           reset    : in   STD_LOGIC;                          -- Reset the register
18         Ri       : in   STD_LOGIC_VECTOR (15 downto 0);  -- The input to the SPR
           Ro       : out  STD_LOGIC_VECTOR (15 downto 0)); -- The output from SPR
20   end component;

22   signal sr_enable : std_logic;
     signal sr_reset  : std_logic;
24   signal sr_Ri     : std_logic_vector(15 downto 0);
     signal sr_Ro     : std_logic_vector(15 downto 0);
26
     component pc
28   Port (clk      : in   STD_LOGIC;
           enable   : in   STD_LOGIC;                          -- Enable write
30         reset    : in   STD_LOGIC;                          -- Reset the register
           Ri       : in   STD_LOGIC_VECTOR (15 downto 0);  -- The input to the SPR
32         Ro       : out  STD_LOGIC_VECTOR (15 downto 0)); -- The output from SPR
     end component;
34
     signal pc_enable : std_logic;
36   signal pc_reset  : std_logic;
     signal pc_Ri     : std_logic_vector(15 downto 0);
38   signal pc_Ro     : std_logic_vector(15 downto 0);

40   signal T_clk  : std_logic;

42 begin

44   U_sr: sr port map (clk => T_clk, enable => sr_enable, reset => sr_reset, Ri =>
         sr_Ri, Ro => sr_Ro);
     U_pc: pc port map (clk => T_clk, enable => pc_enable, reset => pc_reset, Ri =>
         pc_Ri, Ro => pc_Ro);
46
       -- concurrent process to offer the clk signal
48   process
     begin
50     T_clk <= '0';
       wait for 5 ns;
52     T_clk <= '1';
       wait for 5 ns;
54   end process;

56   process

58     variable err_cnt: integer :=0;

60   begin

62     -- Write
       sr_enable <= '1';
64     sr_reset  <= '0';
       sr_Ri     <= "0100011001011001";
66     pc_enable <= '1';
```

```
    pc_reset  <= '0';
    pc_Ri     <= "0101011010110100";
    wait for 20 ns;

    -- Read
    assert (sr_Ro="0100011001011001") report "Read sr #1 failed" severity error;
    assert (pc_Ro="0101011010110100") report "Read pc #1 failed" severity error;

    -- Change Ri
    sr_Ri <= "1001100101110100";
    pc_Ri <= "0001010001110000";
    wait for 20 ns;
    assert (sr_Ro = "1001100101110100") report "Read sr #2 failed" severity error;
    assert (pc_Ro = "0001010001110000") report "Read pc #2 failed" severity error;

    -- Disable sr, pc still enabled
    sr_enable <= '0';
    sr_Ri <= "0101010101010101";
    pc_Ri <= "1010101010101010";
    wait for 20 ns;
    assert (sr_Ro = "1001100101110100") report "Wrote to sr while disabled" severity
        error;
    assert (pc_Ro = "1010101010101010") report "Read pc #3 failed" severity error;

    -- Enable sr
    sr_enable <= '1';
    wait for 20 ns;
    assert (sr_Ro = "0101010101010101") report "Read sr #3 failed" severity error;

    -- Disable pc, sr still enabled
    pc_enable <= '0';
    sr_Ri <= "0000000011111111";
    pc_Ri <= "1111111100000000";
    wait for 20 ns;
    assert (sr_Ro = "0000000011111111") report "Read sr #4 failed" severity error;
    assert (pc_Ro = "1010101010101010") report "Wrote to pc while disabled" severity
        error;

    -- Enable pc
    pc_enable <= '1';
    wait for 20 ns;
    assert (pc_Ro = "1111111100000000") report "Read pc #4 failed" severity error;


    assert false report "End of test" severity note;
    wait; -- wait forever to end the test

  end process;

end TB;

-------------------------------------------------------------------
configuration CFG_TB of spr_TB is
  for TB
  end for;
end CFG_TB;
```

Listing 27: processor/gpr_tb.vhd

```
-- Authors:
--       Henry Jenkins, Joel Koh

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--  A testbench has no ports.
entity gpr_tb is
  end gpr_tb;

architecture behav of gpr_tb is
  --  Declaration of the component that will be instantiated.
  component gpr
```

```
14    Port(clk                  : IN std_logic;                      -- Clock
          enable                : IN std_logic;                      -- Enable input
              (output is always enabled)
16        SelRx, SelRy, SelRi   : IN std_logic_vector(2 DOWNTO 0);   -- Selecti which
              registers to use
          Ri                    : IN std_logic_vector(7 DOWNTO 0);   -- Input
18        Rx, Ry                : OUT std_logic_vector(7 DOWNTO 0)); -- Outputs
    end component;
20  --  Specifies which entity is bound with the component.
    for gpr_0: gpr use entity work.gpr;
22  signal clk, enable          : std_logic;
    signal SelRx, SelRy, SelRi  : std_logic_vector(2 DOWNTO 0);
24  signal Ri, Rx, Ry           : std_logic_vector(7 DOWNTO 0);
  begin
26  --  Component instantiation.
    gpr_0: gpr port map (clk => clk, enable => enable, SelRx => SelRx, SelRy => SelRy,
        SelRi => SelRi, Ri => Ri, Rx => Rx, Ry => Ry);
28
    -- Does the clock signal
30  process
    begin
32    clk <= '0';
      wait for 5 ns;
34    clk <= '1';
      wait for 5 ns;
36  end process;

38  --  This process does the real job.
    process
40  begin

42    -- Write to R0
      SelRi <= "000";
44    Ri <= "00010100";
      enable <= '1';
46    wait for 20 ns;

48    -- Read R0 from Rx
      SelRx <= "000";
50    wait for 20 ns;
      assert (Rx = "00010100") report "Read from Rx failed #1" severity error;
52
      -- Read R0 from Ry
54    SelRy <= "000";
      wait for 20 ns;
56    assert (Ry = "00010100") report "Read from Ry failed #1" severity error;

58    -- Disable write
      enable <= '0';
60    wait for 20 ns;

62    -- Change Ri (should not write as it is disabled)
      Ri <= "00101010";
64    wait for 20 ns;
      assert (Rx = "00010100") report "Wrote to register while disabled #1" severity
          error;
66    assert (Ry = "00010100") report "Wrote to register while disabled #2" severity
          error;

68    -- Enable write
      enable <= '1';
70    wait for 20 ns;
      assert (Rx = "00101010") report "Read from Rx failed #2" severity error;
72    assert (Ry = "00101010") report "Read from Ry failed #2" severity error;

74    -- Write to R2
      SelRi <= "010";
76    Ri <= "01010001";
      wait for 20 ns;
78
      -- Read R2 from Rx
80    SelRx <= "010";
      wait for 20 ns;
```

```
82      assert (Rx = "01010001") report "Read from Rx failed #3" severity error;

84      -- Read R2 from Ry
        SelRy <= "010";
86      wait for 20 ns;
        assert (Ry = "01010001") report "Read from Ry failed #3" severity error;
88
        -- Read R0 from Rx again (should not have changed from previous results)
90      SelRx <= "000";
        wait for 20 ns;
92      assert (Rx = "00101010") report "Read from Rx failed #4" severity error;

94      -- Read R0 from Ry again (should not have changed from previous results)
        SelRy <= "000";
96      wait for 20 ns;
        assert (Ry = "00101010") report "Read from Ry failed #4" severity error;
98
        -- Wait for a long time
100     wait for 1 ms;
        assert (Rx = "00101010") report "Read from Rx failed #5" severity error;
102     assert (Ry = "00101010") report "Read from Ry failed #5" severity error;

104     -- Read R2 after a long time
        SelRx <= "010";
106     SelRy <= "010";
        wait for 1 ms;
108     assert (Rx = "01010001") report "Read from Rx failed #6" severity error;
        assert (Ry = "01010001") report "Read from Ry failed #6" severity error;
110

112     assert false report "End of test" severity note;
        wait; -- wait forever to end the test
114
    end process;
116 end behav;
```

Listing 28: mmu/mmu_tb.vhd

```
1  -- Author:
   --      Wim Looman
3
   library IEEE;
5  use IEEE.STD_LOGIC_1164.ALL;

7  library work;
   use work.mmu_main;
9  use work.minimal_uart_core;
   use work.txt_util.all;
11
   entity mmu_tb is
13 end mmu_tb;

15 architecture tb of mmu_tb is
     component mmu_main is
17     port (
         -- instruction bus
19       inst_add  : in  std_logic_vector(11 downto 0); -- Address lines.
         inst_data : out std_logic_vector(15 downto 0); -- Data lines.
21       inst_req  : in  std_logic;                     -- Pulled low to request bus
             usage.
         inst_ack  : out std_logic;                     -- Pulled high to inform of
             request completion.
23       -- data bus
         data_add  : in    std_logic_vector(15 downto 0); -- Address lines.
25       data_line : inout std_logic_vector(7 downto 0);  -- Data lines.
         data_read : in    std_logic;                     -- High for a read request,
             low for a write request.
27       data_req  : in    std_logic;                     -- Pulled low to request bus
             usage.
         data_ack  : inout std_logic;                     -- Pulled high to inform of
             request completion.
29       -- extras
```

```
      clk          : in  std_logic;
      receive_pin  : in  std_logic;
      transfer_pin : out std_logic
    );
  end component;

  component minimal_uart_core is
    port(
      clock : in    std_logic;
      eoc   : out   std_logic;
      outp  : inout std_logic_vector(7 downto 0) := "ZZZZZZZZ";
      rxd   : in    std_logic;
      txd   : out   std_logic;
      eot   : out   std_logic;
      inp   : in    std_logic_vector(7 downto 0);
      ready : out   std_logic;
      wr    : in    std_logic
    );
  end component;

  signal inst_add     : std_logic_vector(11 downto 0);
  signal inst_data    : std_logic_vector(15 downto 0);
  signal inst_req     : std_logic := '1';
  signal inst_ack     : std_logic;
  signal data_add     : std_logic_vector(15 downto 0);
  signal data_line    : std_logic_vector(7 downto 0);
  signal data_read    : std_logic;
  signal data_req     : std_logic;
  signal data_ack     : std_logic;
  signal clk          : std_logic;
  signal receive_pin  : std_logic;
  signal transfer_pin : std_logic;

  signal eoc, rxd, txd, eot, ready, wr: std_logic;
  signal outp, inp : std_logic_vector(7 downto 0);

  signal current_recv : std_logic_vector(7 downto 0);
  signal current_send : std_logic_vector(7 downto 0);
begin
  m : mmu_main port map (inst_add, inst_data, inst_req, inst_ack, data_add,
      data_line, data_read, data_req, data_ack, clk, receive_pin, transfer_pin);
  muart : minimal_uart_core port map (clk, eoc, outp, rxd, txd, eot, inp, ready, wr);

  rxd <= transfer_pin;
  receive_pin <= txd;

  clk_gen : process begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
  end process;

  inst_test : process
    type pattern_type is record
      inst_add      : std_logic_vector(11 downto 0);
      recv_head     : std_logic_vector( 7 downto 0);
      send_head     : std_logic_vector( 7 downto 0);
      inst_data     : std_logic_vector(15 downto 0);
    end record;
    type pattern_array is array (natural range <>) of pattern_type;
    constant patterns : pattern_array :=
--      inst_add       recv_head   send_head   inst_data
    ((x"52E", x"81", x"83", x"83A7"),
     (x"96F", x"81", x"83", x"4F5E"),
     (x"8F1", x"81", x"83", x"5937"),
     (x"65A", x"81", x"83", x"A8F2"));
  begin
    wr <= '0';
    for i in patterns'range loop
      wait for 10000 ns;
      inst_add <= patterns(i).inst_add;
      wait for 20 ns;
```

```
        inst_req <= '0';
103
        wait until eoc'event;
105     assert outp = patterns(i).recv_head
          report "Bad header expected '" & str(patterns(i).recv_head) & "' recieved '"
              & str(outp) & "'"
107       severity error;
        wait until eoc'event;
109
        assert false report "passed header" severity note;
111
        wait until eoc'event;
113     assert outp = patterns(i).inst_add(7 downto 0)
          report "Bad address low expected '" & str(patterns(i).inst_add(7 downto 0)) &
               "' recieved '" & str(outp) & "'"
115       severity error;
        wait until eoc'event;
117
        assert false report "passed address low" severity note;
119
        wait until eoc'event;
121     assert outp = "0000" & patterns(i).inst_add(11 downto 8)
          report "Bad address high expected '" & str(patterns(i).inst_add(11 downto
              8)) & "' recieved '" & str(outp) & "'"
123       severity error;
        wait until eoc'event;
125
        assert false report "passed address high" severity note;
127
        wait for 100 ns;
129     inp <= patterns(i).send_head;
        wait for 20 ns;
131     wr <= '1';
        wait for 20 ns;
133     wr <= '0';
        wait until eot'event;
135     wait until eot'event;

137
        wait for 100 ns;
139     inp <= "0000" & patterns(i).inst_add(11 downto 8);
        wait for 20 ns;
141     wr <= '1';
        wait for 20 ns;
143     wr <= '0';
        wait until eot'event;
145     wait until eot'event;

147
        wait for 100 ns;
149     inp <= patterns(i).inst_add(7 downto 0);
        wait for 20 ns;
151     wr <= '1';
        wait for 20 ns;
153     wr <= '0';
        wait until eot'event;
155     wait until eot'event;

157
        wait for 100 ns;
159     inp <= patterns(i).inst_data(7 downto 0);
        wait for 20 ns;
161     wr <= '1';
        wait for 20 ns;
163     wr <= '0';
        wait until eot'event;
165     wait until eot'event;

167
        wait for 100 ns;
169     inp <= patterns(i).inst_data(15 downto 8);
        wait for 20 ns;
171     wr <= '1';
```

```
        wait for 20 ns;
173     wr <= '0';
        wait until eot'event;
175     wait until eot'event;

177     assert inst_ack = '1'
          report "receipt not acknowledged"
179       severity error;

181     assert inst_data = patterns(i).inst_data
          report "Wrong data recieve expected '" & str(patterns(i).inst_data) & "'
              recieved '" & str(inst_data) & "'"
183       severity error;

185     assert false report "finished transmission" severity note;

187     wait for 20 ns;

189     inst_req <= '1';
      end loop;
191   wait;
    end process;
193 end tb;
```

Listing 29: data_tb.vhd

```
 1 library IEEE;
   use IEEE.STD_LOGIC_1164.ALL;
 3
   library work;
 5 use work.mmu_main;
   use work.minimal_uart_core;
 7 use work.txt_util.all;

 9 entity data_tb is
   end data_tb;
11
   architecture tb of data_tb is
13   component mmu_main is
       port (
15       -- instruction bus
         inst_add  : in  std_logic_vector(11 downto 0); -- Address lines.
17       inst_data : out std_logic_vector(15 downto 0); -- Data lines.
         inst_req  : in  std_logic;                     -- Pulled low to request bus
             usage.
19       inst_ack  : out std_logic;                     -- Pulled high to inform of
             request completion.
         -- data bus
21       data_add  : in    std_logic_vector(15 downto 0); -- Address lines.
         data_line : inout std_logic_vector(7 downto 0);  -- Data lines.
23       data_read : in    std_logic;                     -- High for a read request,
             low for a write request.
         data_req  : in    std_logic;                     -- Pulled low to request bus
             usage.
25       data_ack  : inout std_logic;                     -- Pulled high to inform of
             request completion.
         -- extras
27       clk          : in  std_logic;
         receive_pin  : in  std_logic;
29       transfer_pin : out std_logic
       );
31   end component;

33   component IO is
       PORT(
35       -- data bus --
         data_add    : IN      std_logic_vector(15 DOWNTO 0);    -- address lines --
37       data_data : INOUT     std_logic_vector(7 DOWNTO 0);   -- data lines --
         data_read : INOUT     std_logic;                -- pulled high for read, low
             for write --
39       data_req    : INOUT   std_logic;                -- pulled low to request bus
             usage --
```

```
        data_ack    : INOUT   std_logic;                -- pulled high to inform
            request completion --
41      -- io --
        clk         : IN      std_logic;
43      sw1         : IN      std_logic;
        sw2         : IN      std_logic
45    );
    end component;
47
    component minimal_uart_core is
49    port(
        clock : in    std_logic;
51      eoc   : out   std_logic;
        outp  : inout std_logic_vector(7 downto 0) := "ZZZZZZZZ";
53      rxd   : in    std_logic;
        txd   : out   std_logic;
55      eot   : out   std_logic;
        inp   : in    std_logic_vector(7 downto 0);
57      ready : out   std_logic;
        wr    : in    std_logic
59    );
    end component;
61
    signal inst_add     : std_logic_vector(11 downto 0);
63  signal inst_data    : std_logic_vector(15 downto 0);
    signal inst_req     : std_logic := '1';
65  signal inst_ack     : std_logic;
    signal data_add     : std_logic_vector(15 downto 0);
67  signal data_line    : std_logic_vector(7 downto 0);
    signal data_read    : std_logic;
69  signal data_req     : std_logic;
    signal data_ack     : std_logic;
71  signal clk          : std_logic;
    signal receive_pin  : std_logic;
73  signal transfer_pin : std_logic;
    signal sw1, sw2 : std_logic;
75
    signal eoc, rxd, txd, eot, ready, wr: std_logic;
77  signal outp, inp : std_logic_vector(7 downto 0);
  begin
79  m : mmu_main port map (inst_add, inst_data, inst_req, inst_ack, data_add,
        data_line, data_read, data_req, data_ack, clk, receive_pin, transfer_pin);
    i : IO       port map (data_add, data_line, data_read, data_req, data_ack, clk,
        sw1, sw2);
81  muart : minimal_uart_core port map (clk, eoc, outp, rxd, txd, eot, inp, ready, wr);

83  rxd <= transfer_pin;
    receive_pin <= txd;
85
    clk_gen : process begin
87    clk <= '0';
      wait for 10 ns;
89    clk <= '1';
      wait for 10 ns;
91  end process;

93  data_test : process
      type pattern_type is record
95      data_add      : std_logic_vector(15 downto 0);
        recv_head     : std_logic_vector( 7 downto 0);
97      send_head     : std_logic_vector( 7 downto 0);
        data_data     : std_logic_vector( 7 downto 0);
99      switch_data   : std_logic_vector( 1 downto 0);
        rw            : std_logic;
101   end record;
      type pattern_array is array (natural range <>) of pattern_type;
103   constant patterns : pattern_array :=
  -- data_add, recv_head, send_head, data_data, switch_data, rw
105   ((x"0581", x"80", x"00", x"A7", "00", '1' ),
       (x"0273", x"00", x"00", x"5E", "00", '0' ));
107 begin
      wr <= '0';
109   data_req <= '1';
```

```
    for i in patterns'range loop
      wait for 10000 ns;
      data_add <= patterns(i).data_add;
      data_read <= patterns(i).rw;
      wait for 20 ns;
      if patterns(i).rw = '0' then
        data_line <= patterns(1).data_data;
      else
        data_line <= (others => 'Z');
      end if;
      wait for 20 ns;
      data_req <= '0';

      if patterns(i).data_add(0) = '1' then
        wait until eoc'event;
        assert outp = patterns(i).recv_head
          report "Bad header expected '" & str(patterns(i).recv_head) & "' recieved
              '" & str(outp) & "'"
          severity error;
        wait until eoc'event;

        assert false report "passed header" severity note;

        wait until eoc'event;
        assert outp = patterns(i).data_add(8 downto 1)
          report "Bad address low expected '" & str(patterns(i).data_add(7 downto
              0)) & "' recieved '" & str(outp) & "'"
          severity error;
        wait until eoc'event;

        assert false report "passed address low" severity note;

        wait until eoc'event;
        assert outp = "0" & patterns(i).data_add(15 downto 9)
          report "Bad address high expected '" & str(patterns(i).data_add(11 downto
              8)) & "' recieved '" & str(outp) & "'"
          severity error;
        wait until eoc'event;

        assert false report "passed address high" severity note;
        if patterns(i).rw = '0' then
          wait until eoc'event;
          assert outp = patterns(i).data_data
            report "Bad data expected '" & str(patterns(i).data_data) & "' recieved
                '" & str(outp) & "'"
            severity error;
          wait until eoc'event;

          assert false report "passed data" severity note;
        else
          wait for 100 ns;
          inp <= patterns(i).send_head;
          wait for 20 ns;
          wr <= '1';
          wait for 20 ns;
          wr <= '0';
          wait until eot'event;
          wait until eot'event;


          wait for 100 ns;
          inp <= patterns(i).data_add(8 downto 1);
          wait for 20 ns;
          wr <= '1';
          wait for 20 ns;
          wr <= '0';
          wait until eot'event;
          wait until eot'event;


          wait for 100 ns;
          inp <=  "0" & patterns(i).data_add(15 downto 9);
          wait for 20 ns;
```

```
179             wr <= '1';
                wait for 20 ns;
181             wr <= '0';
                wait until eot'event;
183             wait until eot'event;

185

                wait for 100 ns;
187             inp <= patterns(i).data_data;
                wait for 20 ns;
189             wr <= '1';
                wait for 20 ns;
191             wr <= '0';
                wait until eot'event;
193             wait until eot'event;
            end if;
195         else

197         end if;

199         assert data_ack = '1'
              report "receipt not acknowledged"
201           severity error;

203         if  patterns(i).rw = '1' then
              assert data_line = patterns(i).data_data
205             report "Wrong data recieve expected '" & str(patterns(i).data_data) & "'
                    recieved '" & str(data_line) & "'"
                severity error;
207         end if;

209         assert false report "finished transmission" severity note;

211         wait for 20 ns;

213         data_req <= '1';
        end loop;
215     wait;
    end process;
217 end tb;
```

Tools

Listing 30: assembler.rb

```
#!/usr/bin/env ruby

# Author:
#       Wim Looman
# Copyright:
#       Copyright (c) 2010 Wim Looman
# License:
#       GNU General Public License (see http://www.gnu.org/licenses/gpl-3.0.txt)

def assert(error=nil)
  raise (error || "Assertion Failed!") unless yield
end

# For 8-bit twos complement
def twos_complement(num)
  return 256 + num
end


def logical_operands(chunks)
      y = chunks[1][1..1].to_i
      assert(chunks[1] + " is not a valid register") {y >= 0 && y < 8}
      x = chunks[2][1..1].to_i
      assert(chunks[2] + " is not a valid register") {x >= 0 && x < 8}
      return (y << 5) + x
end


def immediate(chunk, symbols=nil, move_from=nil)
  v = chunk.to_i
  if v == 0 && chunk != "0" && symbols != nil
    assert(chunk + " is not a valid symbol") {symbols.include?(chunk)}
    move_to = symbols[chunk]
    diff = move_to - move_from
    if diff < 1
      diff = twos_complement(diff)
    end
    return diff
  else
    assert(chunk + " is not a valid immediate") {v >= -127 && v < 128}
    if v < 0
      v = twos_complement(v)
    end
    return v
  end
end


def register(chunk, num_registers)
  x = chunk[1..1].to_i
  assert(chunk + " is not a valid register") {x >=0 && x < num_registers}
  return x
end


def auto(chunk)
  if chunk[-1..-1] == "+"
    return 0x08
  elsif chunk[-1..-1] == "-"
    return 0x10
  else
    return 0x00
  end
end


def convert(lines)
  table = first_pass(lines)
```

```
69    return second_pass(lines, table)
  end
71
73  def first_pass(lines)
      instruction = 0
75    symbols = {}
      lines.each do |line|
77      chunks = line.sub(",", " ").split
        case chunks[0]
79        when "LDI", "LD", "STI", "ST", "MV", "AND", "OR", "NOT", "XOR", "ADD", "ADC",
              "SUB", "SBB", "NEG", "CMP", "BEQ", "BNE", "BLT", "BGT", "BC", "BNC", "RJMP",
               "JMP"
            instruction += 1
81
          when "label:"
83          symbols[chunks[1]] = instruction
        end
85    end
      return symbols
87  end
89
  def second_pass(lines, symbols)
91    line_no = 0
      output = []
93    lines.each do |line|
        label = line.sub(",", " ").split[0]
95      case label
          when "LDI", "LD", "STI", "ST", "MV", "AND", "OR", "NOT", "XOR", "ADD", "ADC",
              "SUB", "SBB", "NEG", "CMP", "BEQ", "BNE", "BLT", "BGT", "BC", "BNC", "RJMP",
               "JMP"
97          line_no += 1
            output.push(convert_line(line, symbols, line_no))
99      end
      end
101   return output.flatten
  end
103
105 def convert_line(line, symbols, line_no)
      chunks = line.sub(",", " ").split#.partition(";")[0].split
107
      case chunks[0]
109     when "LDI"
          instruction = 0x21
111       x = register(chunks[1], 4)
          v = immediate(chunks[2])
113       operands = (v << 2) + x

115     when "LD"
          instruction = 0x01 + auto(chunks[2])
117       x = register(chunks[1], 8)
          y = register(chunks[2], 3)
119       operands = (y << 5) + x

121     when "STI"
          instruction = 0x25
123       y = register(chunks[1], 3)
          v = immediate(chunks[2])
125       operands = (v << 2) + y

127     when "ST"
          instruction = 0x05 + auto(chunks[1])
129       y = register(chunks[1], 3)
          x = register(chunks[2], 8)
131       operands = (y << 5) + x

133     when "MV"
          instruction = 0x04
135       if chunks[1][0] == 'r'[0] && chunks[2][0] == 'r'[0]
            y = register(chunks[1], 8)
137         x = register(chunks[2], 8)
```

```
        operands = (y << 5) + x
      elsif chunks[1][0] == 'a'[0]
        y = register(chunks[1], 3)
        x = register(chunks[2], 8)
        n = chunks[1][-1] == 'H' ? 1 : 0
        operands = (1 << 9) + (n << 8) + (y << 5) + x
      elsif chunks[2][0] == 'a'[0]
        y = register(chunks[1], 8)
        x = register(chunks[2], 3)
        n = chunks[2][-1] == 'H' ? 1 : 0
        operands = (1 << 4) + (n << 3) + (y << 5) + x
      else
        # explode
      end

    when "AND"
      instruction = 0x02
      operands = logical_operands(chunks)

    when "OR"
      instruction = 0x06
      operands = logical_operands(chunks)

    when "NOT"
      instruction = 0x0A
      operands = logical_operands(chunks)

    when "XOR"
      instruction = 0x0E
      operands = logical_operands(chunks)

    when "ADD"
      instruction = 0x12
      operands = logical_operands(chunks)

    when "ADC"
      instruction = 0x16
      operands = logical_operands(chunks)

    when "SUB"
      instruction = 0x1A
      operands = logical_operands(chunks)

    when "SBB"
      instruction = 0x1E
      operands = logical_operands(chunks)

    when "NEG"
      instruction = 0x08
      operands = logical_operands(chunks)

    when "CMP"
      instruction = 0x0C
      operands = logical_operands(chunks)

    when "BEQ"
      instruction = 0x23
      v = immediate(chunks[1], symbols, line_no)
      operands = (v << 2)

    when "BNE"
      instruction = 0x27
      v = immediate(chunks[1], symbols, line_no)
      operands = (v << 2)

    when "BLT"
      instruction = 0x2B
      v = immediate(chunks[1], symbols, line_no)
      operands = (v << 2)

    when "BGT"
      instruction = 0x2F
      v = immediate(chunks[1], symbols, line_no)
```

```
211       operands = (v << 2)

213     when "BC"
          instruction = 0x33
215       v = immediate(chunks[1], symbols, line_no)
          operands = (v << 2)
217
        when "BNC"
219       instruction = 0x37
          v = immediate(chunks[1], symbols, line_no)
221       operands = (v << 2)

223     when "RJMP"
          instruction = 0x3B
225       v = immediate(chunks[1], symbols, line_no)
          operands = (v << 2)
227
        when "JMP"
229       instruction = 0x2F
          y = register(chunks[1], 3)
231       operands = (y << 5)
      end
233   opcode = (instruction << 10) + operands
      return [(opcode >> 8), (opcode & 0xFF)]
235 end

237
    if __FILE__ == $0
239   if !(1..2).include?(ARGV.length) || !File.exist?(ARGV[0])
        p "Usage: ruby #{$0} <input_file> [<output_file>]"
241     exit
      end
243
      input = IO.readlines(ARGV[0])
245
      output = convert(input)
247   if ARGV.length == 2:
        File.open(ARGV[1], "wb") do |file|
249       output.each do |char|
            file.putc(char)
251       end
        end
253   else
        output.each do |char|
255       $stdout.putc(char)
        end
257   end
    end
```

Listing 31: memory.rb

```
    #!/usr/bin/env ruby
 2
    # Author:
 4  #       Wim Looman
    # Copyright:
 6  #       Copyright (c) 2010 Wim Looman
    # License:
 8  #       GNU General Public License (see http://www.gnu.org/licenses/gpl-3.0.txt)

10  require 'rubygems'
    require 'serialport'
12
    def serve(program, data_file, sp)
14    while 1
        header = sp.getc
16      diagnostic_mode = (header >> 2) & 0x03
        instruction = header & 0x01 == 0x01
18      address = sp.getc + (sp.getc << 8)

20      case (header >> 7) & 0x01 # read/write bit
          when 0x01               # read
```

```
22          header = 0x82
            header += 0x01 if instruction
24          sp.putc(header)
            instruction ? sp.write(program[address]) :
26                        sp.write(data_file[address])
            p "Sending data for #{instruction ? "Instruction" : "Data"} bus, address:" +
28            "#{address}, data: #{instruction ? program[address] : data_file[address]}"
          when 0x00                # write, doesn't support writing to instruction memory
30          data = sp.getc
            data_file[address] = data
32          p "Writing data, address: " + address + ", data: " + data_file[address]
        end
34    end
  end
36
  if __FILE__ == $0
38
    if ARGV.size < 3
40    STDERR.print "Usage: ruby #{$0} <device> <baud_rate> <program_file>
          [<data_file>]\n"
      exit
42    end

44    device = ARGV[0]
      baud_rate = ARGV[1].to_i
46
      program = Array.new(2**12, 0x00)
48    i = 0
      File.open(ARGV[2], 'rb') do |input|
50      input.each_byte do |byte|
          program[i] =  byte
52        i += 1
        end
54    end

56    data = Array.new(2**15, 0x00)
      File.open(ARGV[3], 'rb') do |input|
58      input.each_byte do |byte|
          data += byte
60      end
      end if ARGV.size > 3
62
      sp = SerialPort.new(device, baud_rate, 8, 1, SerialPort::NONE)
64
      serve(program, data, sp)
66  end
```