UNIVERSITY OF VICTORIA

Department of Electrical and Computer Engineering

CENG 450 – Computer Systems and Architecture

**Final Project Report**

Title: Computer Processor Design

Report Submitted on: April 1, 2014

Names: Kilian Loftis (V00687362)

Joel Geddert (V00716948)

Table of Contents

1.0 Project Description 4

2.0 Design 4

2.1 Overview 4

2.2 Functional Blocks 5

2.2.1 Program Counter 5

2.2.2 ROM 5

2.2.3 Register File 6

2.2.4 RAM 6

2.2.5 ALU 6

2.3 Pipeline & Buffers 6

2.3.1 Instruction Fetch/Instruction Decode Buffer 7

2.3.2 Instruction Decode/Execute Buffer 7

2.3.3 Execute/Writeback Buffer 8

2.4 Execution 8

2.5 Writeback 9

2.6 Branching 9

2.7 Forwarding 9

2.8 Timing 9

3.0 Discussion 10

4.0 Conclusion 10

5.0 References 10

7.0 Appendix A: Full Schematic 12

8.0 Appendix B: Source Code 13

# Project Description

For the CENG 450 lab, a design project was undertaken throughout the term. The project was to design a pipelined processor that operates on a given RISC-like instruction set. This design was then implemented using a Xilinx FPGA.

# Design

## Overview

Since the processor uses a RISC-like instruction set, all operations are simple and can be executed in a single clock cycle. Because of this, a very simple pipeline can be used, with each stage simply performing its function and then passing its result to the following stage, since there is no need for any stage to wait for its function to take several cycles. This means there is no need for a central controller to monitor and time each pipeline stage.

The pipeline stages are as follows:

1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Write Back

This is identical to a MIPS pipeline except without a memory access stage. This is because the instruction set has no instructions that need to use both the ALU and memory – thus memory access can be rolled into the execution stage; there is no need to wait for the ALU to complete its operation before accessing memory. Because this reduces the pipeline size by 1, stalls are reduced by one clock cycle. Additionally, this allows read-after-write hazards to be completely avoided, as explained in section 2.6.

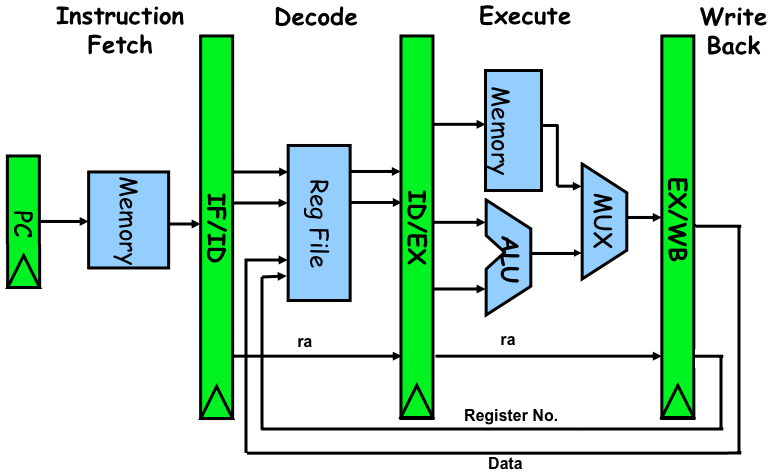


Figure : Simplified Block Diagram

## Functional Blocks

### Program Counter

The basic function of the program counter is to store the current instruction address and increase it by 1 at each rising clock edge. However, it also has a few other capabilities:

* Jump to an instruction, optionally storing the previous address (see section 2.6)
* Increase by 2 instead of 1, for 2-byte instructions (see section 2.3.1)
* Disable or re-enable the counter for branching (see section 2.6)

Figure : Program Counter

### ROM

Figure : ROM

This block stores the program instructions. It simply receives an address and outputs the value stored at that address, as well as the value stored at the following address (see section 2.3.1 for an explanation).

### Register File

The register file contains 4 registers. It receives 2 register indices to be read and outputs the data at these registers asynchronously. It has a 3rd register index to be written on falling clock edges when wr\_enable is set, used in the writeback stage (see section 2.5).

### RAM

The RAM behaves like a simplified version of the register file. It receives a single address and always outputs the data at this address (if this data is not to be used, it is ignored by the following buffer). Additionally, if wr\_enable is set, the input data is written on the falling clock edge.

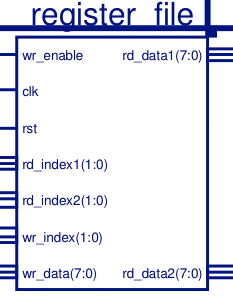
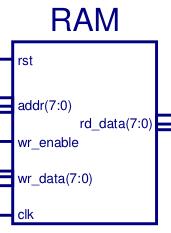
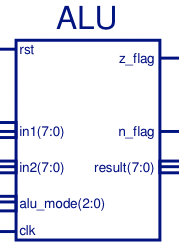
  

Figure : Register File, RAM, and ALU

### ALU

The ALU receives 2 pieces of data, as well as a 3-bit opcode. This opcode is different from the 4-bit instruction opcode (the preceding buffer translates the instruction opcode to ALU opcode). It computes the result of the operation asynchronously. For operations that only require 1 input, input 2 is ignored.

The ALU also has an additional “NOP” code to simply pass the 1st data input through the ALU, leaving it unaffected. This is explained further in section 2.4.

Additionally, the ALU sets the n and z flags. Unlike the operation result, these are set synchronously, on the falling clock edge.

## Pipeline & Buffers

The main purpose of the buffers is to keep data synchronized between stages. For example, when performing a register move operation, the register to be written back to is known immediately, whereas the data to be written needs to be read from its register first. The buffers act as a barrier, such that each piece of information is passed forward at the same time (on rising clock edges) regardless of how long the operation takes within a pipeline stage. This means that even if data is not needed in at all in a stage (but is needed later in the pipeline), it is passed through each buffer and simply not used until it reaches the relevant stage.

Since there is no central controller, the logic to control the CPU (for example, deciding when to branch) is also contained within the buffers.

### Instruction Fetch/Instruction Decode Buffer

The main purpose of this buffer is to decode the instruction. It takes the data output from ROM, analyzes the opcode, and decides which data to pass to the register file and ID/EX buffer depending on the opcode.

The only inputs to this buffer are data from ROM and the input port. This buffer actually accepts 2 data bytes from ROM; this is because L-format instructions are two data bytes long. If the buffer determines the input instruction is of the L format, it analyzes the second byte; otherwise, it is ignored.

This buffer also controls some of the functions of the program counter. For one, if the opcode indicates an L-format instruction, the buffer tells the program counter to increase by 2 instead of 1, since the next byte is part of the current instruction. Additionally, when the operation indicates returning from a subroutine, it is this buffer that tells the program counter through the “LR\_rtrn” signal. Finally, if the instruction is a normal branch, this buffer tells the program counter to pause (see section 2.6 for more information).

### Instruction Decode/Execute Buffer

The main purpose of this branch is to take the results received from the decode stage and prepare them for the execution stage. This means determining what the correct ALU inputs are, which ALU opcode to use, and whether or not to access memory (as well as which address).

This buffer also handles branching. If the received instruction is a branch, it determines whether the branch conditions are met. Conditions are based on the ‘n’ and ‘z’ flags provided by the ALU. This covered in more detail in section 2.6.

Additionally, this buffer has extra logic to handle forwarding such that read-after-write hazards will always be avoided (see section 2.7).

### Execute/Writeback Buffer

This buffer is very simple; it passes most of its inputs back to the register file to be written back. This includes whatever the result of the execution stage was, regardless of whether it is actually being written back or not. This is because the ID/EX buffer decides whether or not to write back any data, and if not, a ‘0’ is passed along on the “WB\_EN” line – so by passing this value back to the register file, the execution result will be ignored regardless.

The only other function of this buffer is to control output port: In the case of the “OUT” opcode, the ID/EX buffer will have sent a ‘1’ on the “IO\_op” line, so this buffer outputs the execution result to the output port. Otherwise, it sends a ‘Z’ signal on this bus, such that if this processor were to be hooked up to a larger system, other elements in the system could also use this output bus.

## Execution

The execution stage is connected such that all operations must go through either the ALU or memory, even if neither is using these. To accommodate for this, the ALU has an additional opcode for “NOP”, which simply passes its 1st data input through the ALU unaffected. Using this design instead of having a separate line for data to skip the ALU simplifies the design greatly without any performance cost (as a clock period needs to be long enough for an ALU operation to complete regardless, so the slight latency caused by the ALU needing to read the NOP opcode does not affect the final timing).

Since the execution stage encompasses both RAM access and the ALU, a MUX is used to select which of these two results is correct. The previous buffer stage selects which data is to be used. This way, even though both the RAM and ALU are outputting data, the ID\_EX\_Buffer selects which of the two is valid. Additionally, the buffer ensures no data will be overwritten incorrectly nor the ALU flags erroneously reset, since if the ALU is being used the RAM operation will be set to “read”, or if the memory is used the ALU operation will be “NOP”.

The value output from this MUX is also the result that is used for forwarding (see section 2.7 for more information on forwarding.)

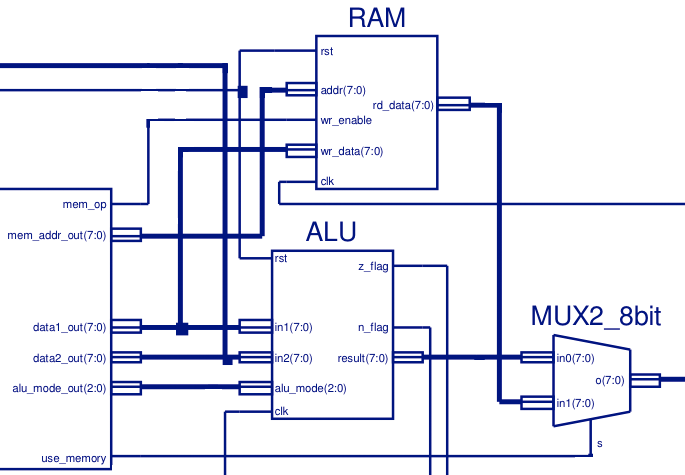


Figure : Execution Stage (signals not relevant to this stage omitted)

## Writeback

The writeback stage consists of 3 lines: data, register index, and enable. The data and index are always passed along from the decode stage, even for instructions which have nothing to do with writing back. However, in these cases the writeback enable signal will be ‘0’, so this is irrelevant because the writeback data will be ignored anyway.

## Branching

The primary logic for branching takes place in the IF\_ID\_Buffer, however, the Program Counter receives branching-related information from several different sources. The first thing to happen when the IF\_ID\_Buffer detects a branch is to stall the Program Counter by asserting the ‘DIS’ signal. Once this has happened, the branch instruction can continue on to the ID\_EX\_Buffer where it will check the conditions of the branch instruction. The ‘n’ and ‘z’ flags from the previous instruction are fed back to the buffer and will be available for to the instruction to determine branch path.

Based on the outcome of the branch instruction, the buffer can assert any of three signals: ‘br’ to signal that the Program counter needs to jump to another address, ‘lr\_goto’ to signal that this is a subroutine and the current value must be saved, and ‘pc\_en’ to let the Program Counter know that it should resuming counting. ‘pc\_en’ will always be asserted when the branch condition is resolved, and ‘br’ will always be asserted if the branch is a subroutine call.

The address of the branch location is carried through from the Register on the DATA1 line as any other data would be. The line, however, is also connected to the Program Counter, and is read whenever ‘br’ is asserted.

Finally, if the IF\_ID\_Buffer decodes an instruction to return from the subroutine, the buffer will signal the Program Counter on the ‘lr\_rtrn’ line to restore saved Counter Value. This does not cause a stall.

## Forwarding

Since there is only one stage between execution and writeback, any read-after-write dependencies are eliminated without causing a pipeline stall, by feeding the new data back to the ID\_EX\_Buffer at the same time as it is being written to the register (one stage further back). To do this, the ID\_EX\_Buffer stores the writeback register name of the previously executed instruction and compares it to either of the registers referenced in the current instruction. If one of these is the same as the previous writeback location, the buffer will use the forwarded value instead of the value taken from the register.

## Timing

It is important to note that, while almost all synchronous units are rising-edge triggered, with the exception of the logic in the IF\_ID\_Buffer governing branching. The logic routine triggers on the falling edge in order to pause the Program Counter before it can increment again. Similarly, the ALU sets the ‘n’ and ‘z’ flags on the falling edge of clock in order to have them ready for branch instructions in the ID\_EX\_Buffer. Also, any time data is written, it is done on the falling edge in order to ensure that valid data is available.

All read operations, such as from the Register, RAM and Memory, as well as all calculations are done asynchronously so that they can be ready before the next clock cycle.

# Discussion

This processor design has many strengths that allow it to quickly execute code and efficiently. Foremost is the elimination of data hazards. With fewer pipeline stages, the pipeline does not need to stall to wait for data to be written back, or even for calculations to finish. It can always be guaranteed that the previous calculation will have finished, and the most recent data can be forwarded to the instruction about to be executed. In addition, the two-byte L-format instructions need only one clock cycle to be read, instead of the typical two, due to the fact that Memory always sends information in 2-byte blocks.

Unfortunately, the process must stall for every branch instruction due to a lack of branch prediction. However, the stall is only 2 clock cycles, not the full 3 needed to traverse the entire CPU, due to the timing discussed in section 2.8. The flags from the previous instruction can be read immediately after the preceding instruction is executed and used to determine the branch outcome.

The stall caused by branch instructions could be eliminated by implementing a branch predictor, however delays will still occur if the prediction is incorrect. Even so, the simplest predictor, one that always chooses the same branch can improve performance.

Other improvements could be made by increasing the amount of stages in the pipeline. While the current arrangement eliminates pipeline hazards, the actual performance may be decreased due to the larger pipeline stages. Smaller stages can be executed quicker, leading to a higher clock speed, but care must be taken to avoid pipeline hazards.

The largest obstacles we faced stemmed not from design, but from the Xilinx software we used for the design. In addition to crashing on occasion, Xilinx, many times, failed to update schematics, requiring a restart before the changes would be implemented. Furthermore, the schematic interface was poorly implemented, meaning that data lines may connect without showing, or not connect at all. This led to significant frustration while debugging. Finally, when it came time to program the FPGA, Xilinx failed to do so, and we were forced to use a different computer and chip to have our design loaded properly.

# Conclusion

We were able to design and implement a pipelined processor using VHDL in the Xilinx programming environment, and program it onto an FPGA. The design was able to execute a RISC instruction set and, while being limited by stalls when branching and slower maximum clock speed due to fewer pipeline stages, it did effectively eliminate pipeline hazards.

# References

[1] < PUT LAB WEBSITE HERE >

*Previous:*

*[1] Morgan, Rakhmatov, Jones. CENG 355: Microprocessor-Based Laboratory Manual, University of Victoria 2009*

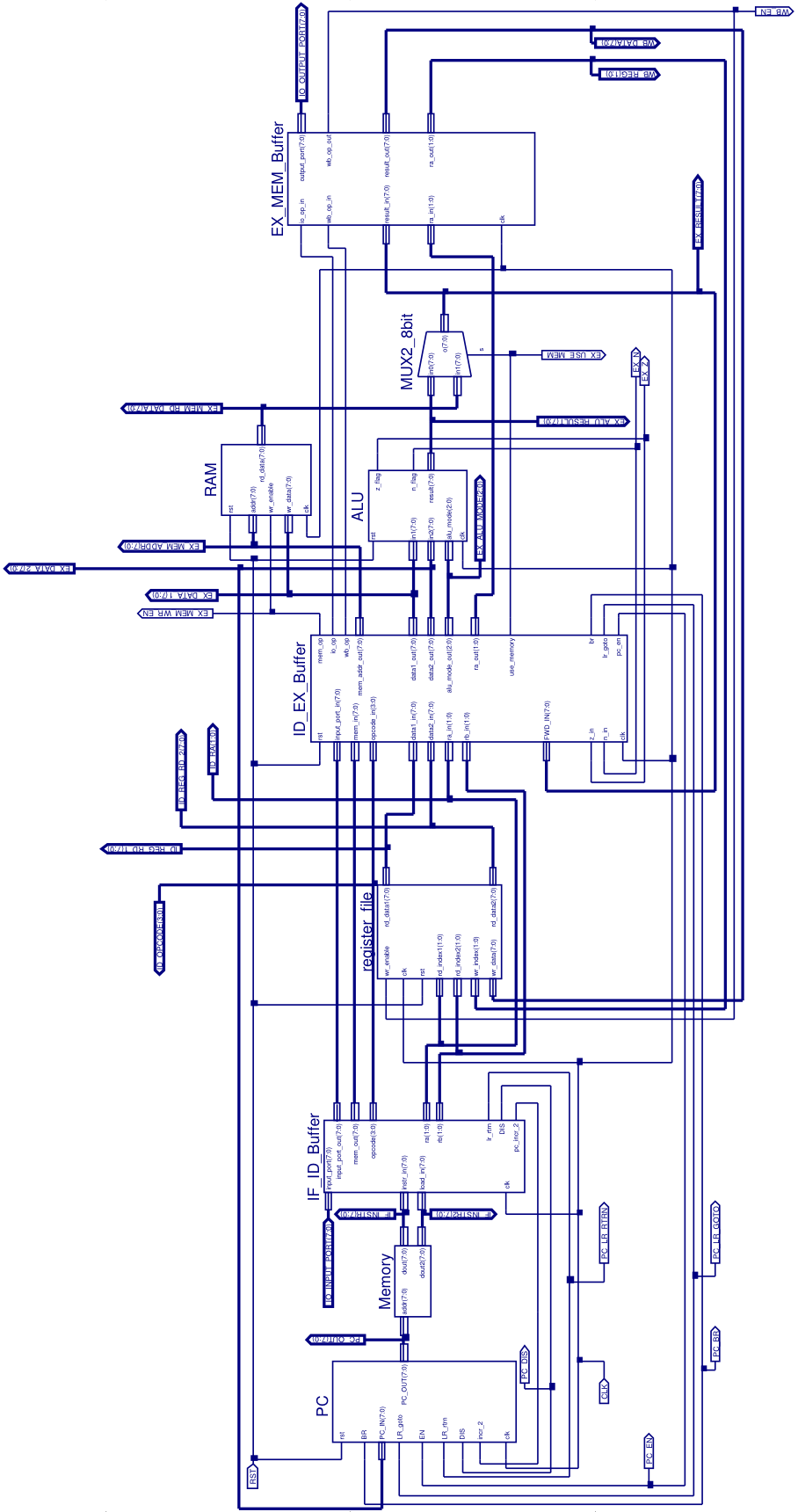
*[2] Microchip Technology Inc. MCF4921/4922 Data sheet , 2007*

*[3] MCF52235 Coldfire Integrated Microcontroller Reference Manual. MCF52235RM. Revision 5. 09/2007*

*[4] M52233DEMO Demonstration Board for Freescale MCF52233*

*[5] Teghan Godkin. Laboratory Instructor for CENG 355; Private Communication September – November 2012.*

# Appendix A: Full Schematic



# Appendix B: Source Code

Note: every file contains the following includes, listed once here instead of in each file to save space:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

## ALU

entity ALU is

Port ( in1 : in STD\_LOGIC\_VECTOR (7 downto 0);

in2 : in STD\_LOGIC\_VECTOR (7 downto 0);

alu\_mode : in STD\_LOGIC\_VECTOR (2 downto 0);

clk : in STD\_LOGIC;

rst : in STD\_LOGIC;

result : out STD\_LOGIC\_VECTOR (7 downto 0);

z\_flag : out STD\_LOGIC;

n\_flag : out STD\_LOGIC);

end ALU;

architecture Behavioral of ALU is

begin

--write operation

process(clk)

begin

if(clk='0' and clk'event) then

if(rst='1') then

n\_flag <= '0';

z\_flag <= '0';

elsif (alu\_mode="000") then

--NAND

if (((in1 NAND in2) AND "10000000") = "10000000" ) then

n\_flag <= '1';

else

n\_flag <= '0';

end if;

if ((in1 NAND in2) = 0) then

z\_flag <= '1';

else

z\_flag <= '0';

end if;

elsif (alu\_mode="001") then

--NOP

elsif (alu\_mode="100") then

--ADD

if (((in1 + in2) AND "10000000") = "10000000" ) then

n\_flag <= '1';

else

n\_flag <= '0';

end if;

if ((in1 + in2) = 0) then

z\_flag <= '1';

else

z\_flag <= '0';

end if;

elsif (alu\_mode="101") then

--SUB

--if ((in1 - in2) < 0) then

if (((in1 - in2) AND "10000000") = "10000000" ) then

n\_flag <= '1';

else

n\_flag <= '0';

end if;

if ((in1 - in2) = 0) then

z\_flag <= '1';

else

z\_flag <= '0';

end if;

elsif (alu\_mode="110") then

--SHL

z\_flag <= in1(7);

elsif (alu\_mode="111") then

--SHR

z\_flag <= in1(0);

end if;

end if;

end process;

--with alu\_mode select result <=

--in1 NAND in2 when "000",

--in1 + in2 when "100",

--in1 - in2 when "101",

--in1(6 downto 0)&'0' when "110",

--'0'&in1(7 downto 1) when "111",

--"XXXXXXXX" when others;

result <= "00000000" when (rst='1') else

in1 when (alu\_mode="001") else

in1 NAND in2 when (alu\_mode="000") else

in1 + in2 when (alu\_mode="100") else

in1 - in2 when (alu\_mode="101") else

in1(6 downto 0)&'0' when (alu\_mode="110") else

'0'&in1(7 downto 1) when (alu\_mode="111") else

"XXXXXXXX";

end Behavioral;

## ROM

(Functional parts only; Program Omitted)

package prog\_mem is

constant mem\_address\_width : integer := 8;

constant mem\_depth : integer := 2\*\*mem\_address\_width;

type storage\_array is array ((mem\_depth-1) downto 0) of std\_logic\_vector(7 downto 0);

end package;

USE work.prog\_mem.all;

ENTITY Memory IS

port (

addr: IN std\_logic\_VECTOR((mem\_address\_width - 1) downto 0);

dout: OUT std\_logic\_VECTOR(7 downto 0);

dout2: OUT std\_logic\_vector(7 downto 0));

END Memory;

ARCHITECTURE CFG OF Memory IS

constant storage : storage\_array := (

-- cut and past your program here

-- be careful to make the distinction between decimal and hex values, especially

-- Indexes in the first column,

-- below, are in decimal, whereas the data supplied is in hexadecimal

--Leave this line in to make sure all unspecified values are set to 00.

others => x"00");

begin

--respond to ROM accesses

dout <= storage(conv\_integer(addr));

dout2 <= storage(conv\_integer(addr+1));

END CFG;

## RAM

entity RAM is

Port ( addr : in STD\_LOGIC\_VECTOR (7 downto 0);

wr\_enable : in STD\_LOGIC;

wr\_data : in STD\_LOGIC\_VECTOR (7 downto 0);

rst : in STD\_LOGIC;

clk : in STD\_LOGIC;

rd\_data : out STD\_LOGIC\_VECTOR (7 downto 0));

end RAM;

architecture Behavioral of RAM is

type mem\_array is array (integer range 0 to 255) of std\_logic\_vector(7 downto 0);

signal mem\_file : mem\_array;

begin

--write operation

process(clk)

begin

if(clk='0' and clk'event) then

if(rst='1') then

for i in 0 to 255 loop

mem\_file(i)<= (others => '0');

end loop;

elsif(wr\_enable='1')then

mem\_file(conv\_integer(addr)) <= wr\_data;

end if;

end if;

end process;

--read operation

rd\_data <= mem\_file(conv\_integer(addr));

end Behavioral;

## Register File

entity register\_file is

Port ( rd\_index1 : in STD\_LOGIC\_VECTOR (1 downto 0);

rd\_index2 : in STD\_LOGIC\_VECTOR (1 downto 0);

wr\_enable : in STD\_LOGIC;

wr\_index : in STD\_LOGIC\_VECTOR (1 downto 0);

wr\_data : in STD\_LOGIC\_VECTOR (7 downto 0);

clk : in STD\_LOGIC;

rst : in STD\_LOGIC;

rd\_data1 : out STD\_LOGIC\_VECTOR (7 downto 0);

rd\_data2 : out STD\_LOGIC\_VECTOR (7 downto 0));

end register\_file;

architecture behavioural of register\_file is

type reg\_array is array (integer range 0 to 3) of std\_logic\_vector(7 downto 0);

signal reg\_file : reg\_array;

begin

--write operation

process(clk)

begin

if(clk='0' and clk'event) then

if(rst='1') then

for i in 0 to 3 loop

reg\_file(i)<= (others => '0');

end loop;

elsif(wr\_enable='1')then

case wr\_index(1 downto 0) is

when "00" => reg\_file(0) <= wr\_data;

when "01" => reg\_file(1) <= wr\_data;

when "10" => reg\_file(2) <= wr\_data;

when "11" => reg\_file(3) <= wr\_data;

when others => NULL;

end case;

end if;

end if;

end process;

--read operation

rd\_data1 <= reg\_file(0) when(rd\_index1="00") else

reg\_file(1) when(rd\_index1="01") else

reg\_file(2) when(rd\_index1="10") else reg\_file(3);

rd\_data2 <= reg\_file(0) when(rd\_index2="00") else

reg\_file(1) when(rd\_index2="01") else

reg\_file(2) when(rd\_index2="10") else reg\_file(3);

end behavioural;

## 8-bit MUX

entity MUX2\_8bit is

Port ( in0 : in STD\_LOGIC\_VECTOR (7 downto 0);

in1 : in STD\_LOGIC\_VECTOR (7 downto 0);

s : in STD\_LOGIC;

o : out STD\_LOGIC\_VECTOR (7 downto 0));

end MUX2\_8bit;

architecture Behavioral of MUX2\_8bit is

begin

o <= in1 when (s='1') else in0;

end Behavioral;

## Program Counter

entity PC is

Port ( PC\_IN : in STD\_LOGIC\_VECTOR (7 downto 0);

EN : in STD\_LOGIC;

DIS : in STD\_LOGIC;

BR : in STD\_LOGIC;

incr\_2 : in STD\_LOGIC;

LR\_goto : in STD\_LOGIC;

LR\_rtrn : in STD\_LOGIC;

clk : in STD\_LOGIC;

rst : in STD\_LOGIC;

PC\_OUT : out STD\_LOGIC\_VECTOR (7 downto 0));

end PC;

architecture Behavioral of PC is

signal enabled : STD\_LOGIC;

signal LR\_VAL : STD\_LOGIC\_VECTOR (7 downto 0);

signal PC : STD\_LOGIC\_VECTOR (7 downto 0);

begin

process(clk)

begin

if(clk='1' and clk'event) then

if (rst = '1') then

-- reset

PC <= "00000000";

LR\_VAL <= "00000000";

PC\_OUT <= "00000000";

elsif (enabled = '1') then

-- increment, branch, or return from subroutine

if (LR\_rtrn = '1') then

-- Return from subroutine

PC <= LR\_VAL;

PC\_OUT <= LR\_VAL;

elsif (BR = '1') then

-- Branch

if (LR\_goto = '1') then

-- If branch was subroutine, store subroutine return

LR\_VAL <= PC + '1';

end if;

PC <= PC\_IN;

PC\_OUT <= PC\_IN;

else

-- Increment

if (incr\_2 = '1') then

PC <= PC + "10";

PC\_OUT <= PC + "10";

else

PC <= PC + '1';

PC\_OUT <= PC + '1';

end if;

end if;

else

PC\_OUT <= "00000000";

end if;

end if;

end process;

-- EN takes priority over DIS to prevent deadlock

enabled <= '1' when(rst = '1' OR EN = '1') else

'0' when(DIS = '1') else enabled;

end Behavioral;

## Instruction Fetch/Instruction Decode Buffer

entity IF\_ID\_Buffer is

Port ( instr\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

load\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

input\_port : in STD\_LOGIC\_VECTOR (7 downto 0);

ra : out STD\_LOGIC\_VECTOR (1 downto 0);

rb : out STD\_LOGIC\_VECTOR (1 downto 0);

lr\_rtrn : out STD\_LOGIC;

opcode : out STD\_LOGIC\_VECTOR (3 downto 0);

input\_port\_out : out STD\_LOGIC\_VECTOR (7 downto 0);

mem\_out : out STD\_LOGIC\_VECTOR (7 downto 0);

DIS : out STD\_LOGIC;

pc\_incr\_2 : out STD\_LOGIC;

clk : in STD\_LOGIC);

end IF\_ID\_Buffer;

architecture Behavioral of IF\_ID\_Buffer is

begin

process(clk)

begin

if(clk='1' and clk'event) then

ra <= instr\_in(3 downto 2);

rb <= instr\_in(1 downto 0);

input\_port\_out <= input\_port;

opcode <= instr\_in(7 downto 4);

mem\_out <= load\_in;

end if;

end process;

process(clk)

begin

if(clk='0' and clk'event) then

-- If opcode (instr\_in 7:4) is 1, 2, or 3

if (instr\_in(7 downto 6) = "00" AND NOT(instr\_in(5 downto 4) = "00")) then

pc\_incr\_2 <= '1';

else

pc\_incr\_2 <= '0';

end if;

-- If branch

if (instr\_in (7 downto 4) = "1001") then

DIS <= '1';

else

DIS <= '0';

end if;

-- If return from subroutine

if (instr\_in (7 downto 4) = "1110") then

lr\_rtrn <= '1';

else

lr\_rtrn <= '0';

end if;

end if;

end process;

end Behavioral;

## Instruction Decode/Execute Buffer

entity ID\_EX\_Buffer is

Port ( opcode\_in : in STD\_LOGIC\_VECTOR (3 downto 0);

data1\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

data2\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

ra\_in : in STD\_LOGIC\_VECTOR (1 downto 0);

rb\_in : in STD\_LOGIC\_VECTOR (1 downto 0);

input\_port\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

FWD\_IN : in STD\_LOGIC\_VECTOR (7 downto 0);

mem\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

z\_in : in STD\_LOGIC;

n\_in : in STD\_LOGIC;

clk : in STD\_LOGIC;

rst : in STD\_LOGIC;

mem\_addr\_out : out STD\_LOGIC\_VECTOR (7 downto 0);

data1\_out : out STD\_LOGIC\_VECTOR (7 downto 0);

data2\_out : out STD\_LOGIC\_VECTOR (7 downto 0);

ra\_out : out STD\_LOGIC\_VECTOR (1 downto 0);

alu\_mode\_out : out STD\_LOGIC\_VECTOR (2 downto 0);

io\_op : out STD\_LOGIC;

wb\_op : out STD\_LOGIC;

br : out STD\_LOGIC;

lr\_goto : out STD\_LOGIC;

pc\_en : out STD\_LOGIC;

mem\_op : out STD\_LOGIC;

use\_memory : out STD\_LOGIC);

end ID\_EX\_Buffer;

architecture Behavioral of ID\_EX\_Buffer is

signal ra\_old : std\_logic\_vector(1 downto 0);

signal oldwb\_op : std\_logic;

begin

process(clk)

begin

if(clk='1' and clk'event) then

if rst = '1' then

pc\_en <= '1';

br <= '0';

lr\_goto <= '0';

else

-- \*\*\*\*\* data1 \*\*\*\*\*

-- Pass data1 along, unless IN operation (then use input port instead)

if(opcode\_in = "1011") then

-- Input port

data1\_out <= input\_port\_in;

elsif(opcode\_in = "0011") then

-- LOADIMM

-- mem\_in is 2nd byte of opcode (either an address or IMM value - but it's always safe to pass through)

data1\_out <= mem\_in;

elsif(opcode\_in = "1101") then

-- MOVE

-- The value we write back is whatever was input to data2

if ((rb\_in = ra\_old) AND (oldwb\_op = '1')) then

-- If this would be Read after Write, take value out of next stage instead

data1\_out <= FWD\_IN;

else

data1\_out <= data2\_in;

end if;

else

if ((ra\_in = ra\_old) AND (oldwb\_op = '1')) then

-- If this would be Read after Write, take value out of next stage instead

data1\_out <= FWD\_IN;

else

data1\_out <= data1\_in;

end if;

end if;

-- \*\*\*\*\* data2 \*\*\*\*\*

-- Pass data2 through (If not needed, ALU op will be such that ALU will ignore it)

if ((rb\_in = ra\_old) AND (oldwb\_op = '1')) then

-- If this would be Read after Write, take value out of next stage instead

data2\_out <= FWD\_IN;

else

data2\_out <= data2\_in;

end if;

-- \*\*\*\*\* I/O \*\*\*\*\*

-- If io\_op = 1, value will be written to output port when it reaches WB stage

if(opcode\_in = "1100") then

io\_op <= '1';

else

io\_op <= '0';

end if;

-- \*\*\*\*\* Ra \*\*\*\*\*

-- pass along Ra so that WB stage will have it

ra\_out <= ra\_in;

-- Writeback op

-- 1 if writeback, 0 if not

-- Writeback on all arithmetic operations, as well as IN, MOV, LOAD, LOADIMM

-- Writeback on: 1, 3, 4, 5, 6, 7, 8, 11, 13

-- Don't writeback on: 0, 2, 9, 12, 14

-- 10 depends on ra

if (opcode\_in="0001" OR opcode\_in="0011" OR opcode\_in="0100" OR

opcode\_in="0101" OR opcode\_in="0110" OR opcode\_in="0111" OR

opcode\_in="1000" OR opcode\_in="1011" OR opcode\_in="1101") then

wb\_op <= '1';

oldwb\_op <= '1';

else

wb\_op <= '0';

oldwb\_op <= '0';

end if;

ra\_old <= ra\_in;

-- \*\*\*\*\* ALU mode \*\*\*\*\*

if (opcode\_in="0100" OR opcode\_in="0101" OR opcode\_in="0110" OR opcode\_in="0111" OR opcode\_in="1000")then

-- If it's an arithmetic operation (4-8), take last 3 bits of opcode.

alu\_mode\_out <= opcode\_in(2 downto 0);

else

-- Otherwise, send ALU NOP code (just passes data1 through)

alu\_mode\_out <= "001";

end if;

-- \*\*\*\*\* Memory (except IMM, which was taken care of with data1) \*\*\*\*\*

if (opcode\_in="0010") then

-- Mem store

mem\_op <= '1';

else

-- All other opcodes

mem\_op <= '0';

end if;

if (opcode\_in="0001") then

-- Only use the value that was output from memory as "result" if reading from memory

use\_memory <= '1';

else

use\_memory <= '0';

end if;

-- Memory address

-- mem\_in is 2nd byte of opcode (either an address or IMM value - but it's always safe to pass through)

mem\_addr\_out <= mem\_in;

-- \*\*\*\*\* Branching \*\*\*\*\*

-- For branching, ra\_in is actually brx

if opcode\_in = "1001" then

if (ra\_in = "00") or (ra\_in = "01" and z\_in = '1') or (ra\_in = "10" and n\_in = '1') then

br <= '1';

lr\_goto <= '0';

pc\_en <= '1';

elsif (ra\_in = "01" and z\_in = '0') or (ra\_in = "10" and n\_in = '0') then

br <= '0';

lr\_goto <= '0';

pc\_en <= '1';

elsif (ra\_in = "11") then

br <= '1';

lr\_goto <= '1';

pc\_en <= '1';

else

br <= '0';

lr\_goto <= '0';

pc\_en <= '1';

end if;

else

br <= '0';

lr\_goto <= '0';

pc\_en <= '0';

end if;

end if;

end if;

end process;

end Behavioral;

## Execute/Writeback Buffer

entity EX\_MEM\_Buffer is

Port ( result\_in : in STD\_LOGIC\_VECTOR (7 downto 0);

wb\_op\_in : in STD\_LOGIC;

ra\_in : in STD\_LOGIC\_VECTOR (1 downto 0);

io\_op\_in : in STD\_LOGIC;

clk : in STD\_LOGIC;

output\_port : out STD\_LOGIC\_VECTOR (7 downto 0);

result\_out : out STD\_LOGIC\_VECTOR (7 downto 0);

wb\_op\_out : out STD\_LOGIC;

ra\_out : out STD\_LOGIC\_VECTOR (1 downto 0));

end EX\_MEM\_Buffer;

architecture Behavioral of EX\_MEM\_Buffer is

begin

process(clk)

begin

if(clk='1' and clk'event) then

result\_out <= result\_in;

if (io\_op\_in='1') then

output\_port <= result\_in;

else

-- Output high impedance so something else could use output bus

output\_port <= "ZZZZZZZZ";

end if;

wb\_op\_out <= wb\_op\_in;

ra\_out <= ra\_in;

end if;

end process;

end Behavioral;