RTL Design of 32-bit RISC Control Processor

Haohua Li #100892262 haohuali@cmail.carleton.ca
Te Bai #7437767 tbai072@uottawa.ca
Siqi Ma #20092184 sma049@uottawa.ca
Zhongyuan Zhao #7352624 zzhao029@uottawa.ca
Peng Yang #7066970 pyang075@uottawa.ca

Abstract—this paper demonstrates most components in CPU and describe how they work together. Also, this paper offers the design of 32-bit RISC CPU by using VHDL and the result of simulation.

Keywords—RISC, Processor, VHDL, FPGA

I.Introduction

Our project is to implement a 32-bit Reduced instruction set computing (RISC) CPU core in Register Logic Level (RTL) by VHDL. Because of the principle of RISC design, we try to make less and tidy instructions so that the CPU could execute more instructions. However, we also make our core modular and expandable in order to support additional operations in future. In our design, the opcode (operating code) is 5 bits, which means we can make 32 different instructions in maximum. We also set up a instruction memory (ROM) to store different programs to be executed.

As we know, the market of embedded system including smart phones and tablets is soaring insanely. Actually, almost all of those systems use ARM as CPU. ARM is a family of instruction set architectures for computer processors based on a RISC architecture developed by British company ARM Holdings. Most versions of the architecture support a 32-bit instruction set.

Even in the market of industry micro-controller, the ARM Cortex-M which is also a group of 32-bit RISC ARM processor, takes place of the traditional 8-bit controller like 8051 by same price but more powerful capability.

Since 32-bit RISC CPU is so poplar, we try to build a simple 32-bit CPU core to utilize digital logic knowledge in practice.

II.Motivation

There are a vast number of designs of CPU. However, some of them are complex instruction set computer (CISC) like [1]; Some of them are 8-bit RISC like [2]; Even some designs like [3] are 32-bit RISC CPU but they are working on behavioural level.

For utilizing what we learn from digital logic class and demonstrating the structure of CPU clearly, different from others, we design the simple 32-bit RISC CPU by on RTL level. For people who want to learn about the function and structure of every basic components of CPU and how they work together, this paper is a good choice.

III.Problem Domain

To build a CPU, there are mainly several components including CPU core, Data path and I/O. We will talk about each component one by one.

1.CPU core

CPU is the core component of the entire computer. The information processing of computer can be divided into two procedures:

a.Enter data and program into memorizer.

b.Execute the program start at the address of the first instruction, obtain the desired result and finish running.

To control and coordinate every part of the computer and execute the instruction set of the

program in sequence, a CPU should have basic functions as follows:

Command Control: These sequential control of the program is called command control. Since a program is a set of instructions, these instructions must be executed in sequences. Therefore, To make sure the computer execute instructions in sequence is the primary task of CPU.

Operation Control: The function of a instruction in general is realized by the combination of several operate signals. Hence, CPU need to generate and manage the operate signal of each instruction fetched from memory. By send kinds of operate signals to corresponding parts to control these parts act as the instructions demand.

TimingControl:Timing control is to timing the execution time of various operations. The period of operation signals, as well as the period of the process of the execution of an instruction. Only then the systems can run smoothly.

Data Processing: The so-called data processing is to proceed the arithmetic and logical operators. Data processing is the basic task of CPU.

There are two kinds of bus architecture: Von Neumann and Harvard architecture. So far most of microprocessors adopt Harvard architecture, this kind of architecture has two separate memory with detached address bus. The CPU fetches program code from program memory first, gets the address after decoding. Then data is read from the corresponding data memory and the CPU jumps to the next step. The program memory and data memory is separate, each of them addressed and accessed independently.There are two buses are set up correspondingly: program bus and data bus, which makes the data throughput doubled. Since the data and program can have different data width, CPUs based on Harvard architecture have a higher efficiency of execution generally. The advantages of Harvard architecture is the instructions and data are stored and organized separately. During execution the next instruction can be read in advance, and fetching code and execution can be run simultaneously. This is very difficult to realize in Von Neumann architecture. In our project, we choose Harvard architecture.

2.Data path

Data path is connected with the decode part. It is used to receive the information from decode part and implement calculations in the CPU. A simple structure of data path is shown as figure 1. From the picture, it shows that three components are included in the structure of data path. The three parts are register part, memory part and arithmetic/logic unit (ALU).

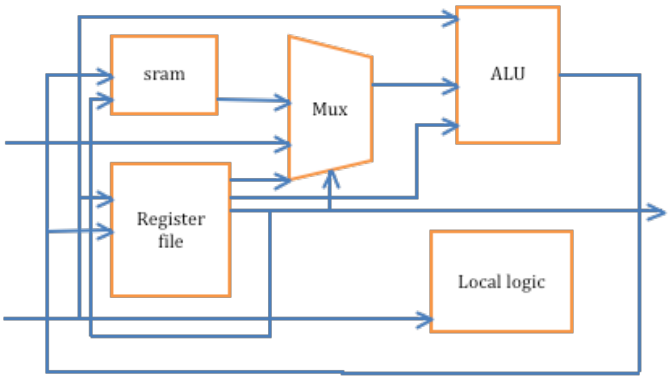


Figure 1 Simple Structure of Data Path

Among the three main components, the most important part is the ALU, which used to do the calculation of the instruments. The following part will give a detailed introduce to register part, memory part and ALU respectively.

2.1.Register file

The register file is used to store instruments received from the decode part. A clock signal is send to the register file as the trigger signal. Assume the register file is raising edge triggered, when the clock signal is on the rising edge, the data that be send from the bus will be stored in either the upper 8-bits or the lower 8-bits of the register.

However, the storage process is not always doing because the bus main send instruments sometimes. Whether there's instruments needed to be stored in the register is decided by the state controller of the CPU. If the signal is 1, it means that an instrument is send from the bus and it should be stored. If the

signal is 0, it means data I transfer from the bus and instrument will be stored.

A reset signal is send to the register to shows whether the register should be cleared or not.

Assume there are 8 buses; every instrument should be read twice. Firstly, the upper 8 bits are read. Then, the lower bits would be read.

Every instrument has two bytes, i.e. 16 bits. The upper 3 bits are used as opcode, while the lower bits are used as address. CPU itself has an address mode, which used to compute address. Figure 2 shows the computation of address.

From the discussion, it can be seen that if the opcodes went through, some capabilities must be given by the register file. Table 6.1 shows the essential capabilities provided by the register file.

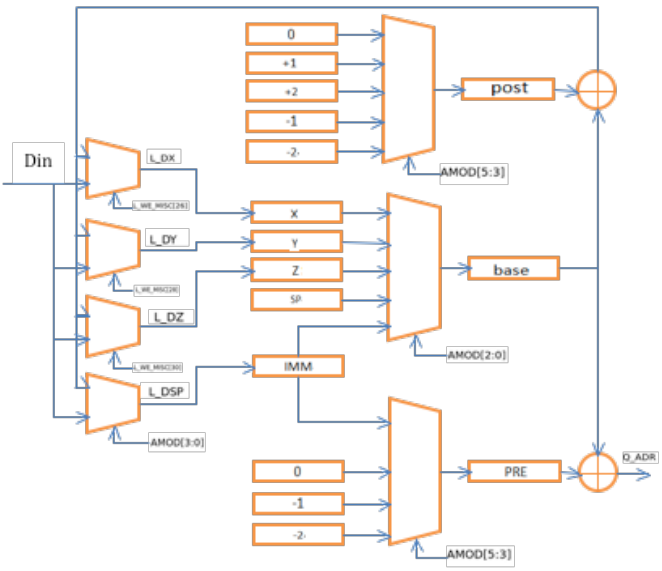


Figure 2 Computation of Address

2.2.Data memory

Apart from the register, data memory is used to store instruments temporary. The data is calculated from the ALU part. Data memory has 8 bits. However, as discussed in the last part, instruments sometimes have 16 bits. There are two ways to solve this problem, which are similar with that of the data memory.

Firstly, when one instrument reaches the memory, others should be locked in the previous stage. However, this method is quite complicated in some steps. In this case, the second way is more widely used.

The second method is to divide the memory into two parts: an even memory and an odd memory. The execution of the memory is as shown in figure 3.

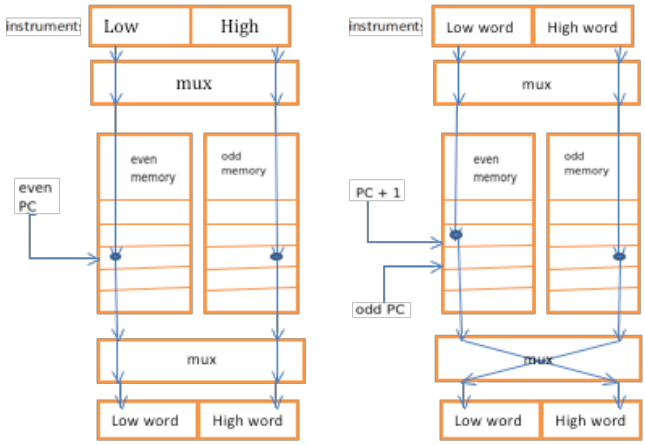


Figure 3 Divided Data Memory into Even Memory and Odd Memory

The only difference between data memory and program memory is that data memory has a mux at the input. This is because data memory is a read/ write memory, yet program memory is read only.

2.3.Arithmetic/Logic unit (ALU)

The arithmetic/logic unit is the most important part in data path. It is mainly used to execute plus, minus, multiply, divide and other logical operations in CPU. Generally, when the clock signal send to the ALU is on the rising edge, ALU will be implemented one time. This is decided by the input/ output registers belonging to the ALU and the mux which is inserted at the input of the ALU part.

There are eight basic opcodes in ALU, which is decided by the three upper bits of the register. Use the opcodes, such as ADD, AND. XOR, and JMP, a lot of other operations can be realized. Also, logic determination can be realized by these operations as well.

2.4.Other functions on Data path

In spite of the first three parts give before, some other functions is needed in the data path.

Firstly, some output is just transferred from the input values without any operations.

Secondly, change the capacity of the memory. This part is realized by using register file; RAM and I/O register which is not inside data path.

Thirdly, a special interface is used to determine the address of most instruments. However, there are two special instructions. The PC value of IJMP is determined by a pair of registers. PC values of RET and RETI is popped from the stack.

Last but not least, it is important to decide whether a branch should be taken or not.

This part mainly discusses on the data path of CPU. Register part, memory part and arithmetic/ logic unit are the three major components of data path and was introduced in detail. This part take the instruments transferred from the decode part and then calculate them in the ALU part. Register and memory are used to storage instruments and data.

Expect the three main components, some other functions are briefly introduced in the end of this part.

4.I/O Unit

I/O system is one of the most important parts of the computer system. The range of the application is enlarging with the development of computer systems. The number and type of I/O devices is becoming more and more and the way I/O device communicating with CPU is different.

The Interface can be viewed as a bridge between two systems or two components. It can be the interface circuit between two hardware components, or it can be the middleware between software. The I/ O unit usually refers to one hardware circuit and relative control unit. Different I/O devices have corresponding control unit. They communicate with the computer by the I/O interface.

The reason why setting interface between I/O devices and computer:

1. Usually, one computer has many I/O devices. All of them have the number of devices (address). The selecting of I/O devices can be realized by I/O interface.
2. There are too many types of devices, their speed is different and may have a lot of difference with the speed of CPU. So the data can be buffered by the I/O interface to make the speed matching.
3. Some of the I/O device may transmit data by serial communication while the CPU usually parallel transfer data. The data can be transferred from serial communication to parallel communication by I/O interface.
4. The electrical level of the Input and output of I/O devices may be different from the CPU's and it can be transferred by the I/O interface.
5. When the CPU start up I/O devices to work. It will send many kinds of control signal to I/O devices and the command can be transmitted by I/O interface.
6. I/O devices need to send its working state to the CPU in time such as busy, get ready, wrong, interrupt request. The I/O interface can monitor the working state of the device and save the information of state for CPU to check.

4.1.I/O bus Organization

Figure 4 shows the bus structure of computer. The I/O device connects to the system bus by I/O interface. The I/O bus includes data line, address line and control line.

a.Data line

Data line is a line that transmitting data between I/O device and the computer. The number of the line is equal to the number of digit bit of reserved word or the number of digit bit of characters. It can be duplexing or simplex. The information it carry may consist of data, complex commands, or addresses.

b.Address line

Address line is used to transmitting addresses of devices. The number of lines is depended on the number of digit bit of device address.

When a processor or DMA-enabled device needs to read or write to a memory location, it specifies that memory location on the address bus (the value to be read or written is sent on the data bus). The width of the address bus (that is, the number of

wires) determines the amount of memory a system can address.

c.Control line

The control lines are used to signal requests and acknowledgments, and to indicate what type of information is on the data lines. The control lines are used to indicate what the bus contains and to implement the bus protocol.

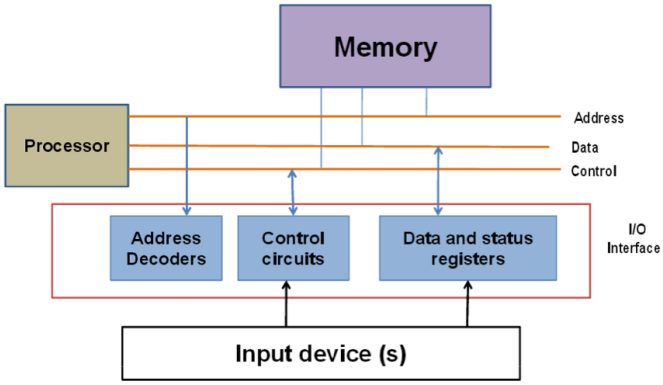


Figure 4 I/O interface for an input device

4.2.Interrupts

Overhead in a polling interface lead to the invention of interrupts to notify the processor when an I/O device requires attention from the processor. Interrupt-driven I/O employs I/O interrupts to indicate to the processor that an I/O device needs attention. When a device wants to notify the processor that it has completed some operation or needs attention, it causes the processor to be interrupted.

When I/O Device is ready, it sends the INTERRUPT signal to processor via a dedicated controller line using interrupt we are ideally eliminating WAIT period. In response to the interrupt, the processor executes the Interrupt Service Routine (ISR). All the registers, flags, program counter values are saved by the processor before running ISR. The time required to save status and restore contribute to “Interrupt Latency”.

Interrupt-acknowledge signal - I/O device interface accomplishes this by execution of an instruction in the interrupt-service routine (ISR) that accesses a status or data register in the device interface which implicitly informs the device that its interrupt request has been recognized. IRQ signal is then removed by device. ISR is a sub-routine – may belong to a different user than the one being executed and then halted. The condition code flags and the contents of any registers used by both the interrupted program and the interrupt-service routine are saved and restored. The concept of interrupts is used in operating systems and in many control applications, where processing of certain routines must be accurately timed relative to external events (e.g. real-time processing).

IV.Literature Review

For this project, we read different designs as reference. How to design your own CPU on FPGAs with VHDL [4] from UBC’s professor Dr. Juergen Sauermann is one the most important reference.

1.Data Path

The data path of this design is relative complicated. It is mainly because this CPU is going to support a large set of instructions.

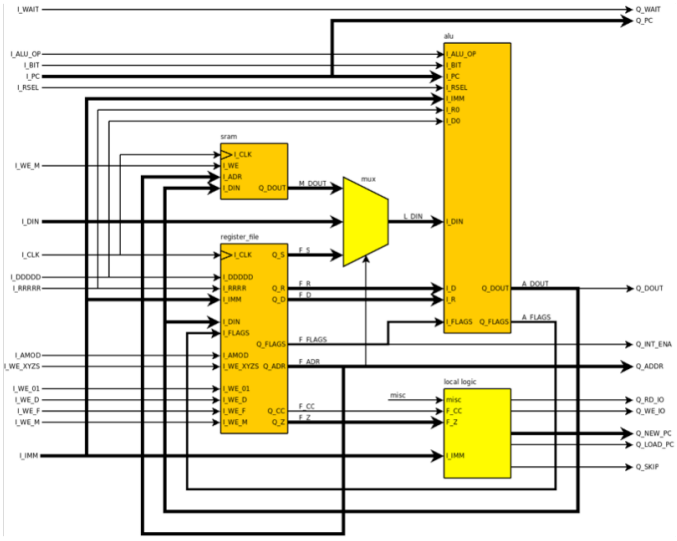


Figure 5 Data path

However, the length of data is 8 bits, which means this CPU could only process small number every circle.

Also, the length of address is 8 bits, which means the CPU can only access 256 bits ROM in maximum.

2. Fetch Instructions

This CPU is better design to fetch instructions as it uses the pipeline technology.

Now most of factories use pipe-line to produce products.Pipeline is a production mode in industry.To improve work efficiency and each production unit only focus on some segment of the assembling. This production mode can also be used in the design of CPUs to improve the performance.

The moment that the electrical level of clock changes from low to high is called the rising edge. The interval between the clock rising edges of two adjacent clock is a clock cycle. If the execution of an instruction takes one clock cycle, the CPU is a single-cycle CPU.

A multi-cycle CPU is to divide the execution of an instruction into several smaller cycles,The number of sub-cycles is decided in accordance with the complexity. The total time of sub-cycles is as same as the cycle time of a single-cycle CPU

Figure 6 is a comparison of the execution cycle of instructions between single-cycle CPU and multi-cycle CPU.

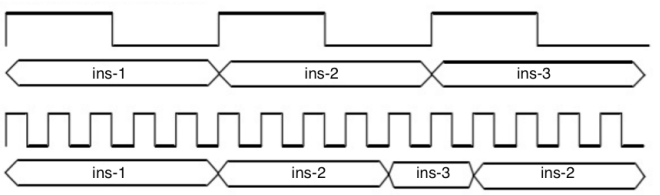


Figure 6 the execution cycle of single cycle CPUs and multi-cycle CPUs

V.Design Part

1. Used Components

There are 5 main components including ROM, RAM, ALU, PC and Decoder for 5 VHDL file respectively. There is also a top-level VHDL file to connect those 5 components as shown on Appendix A Top-level Design of CPU.

Both the length of data word and instruction word are 32-bit long. For every instruction, we use 5 MSB as operation code; the other 27 bits are used as parameter to transfer address or data.

1.1.ROM

ROM is read only memory. In this design, we use ROM as instruction memory to store programs.

The design is shown on Figure 7.

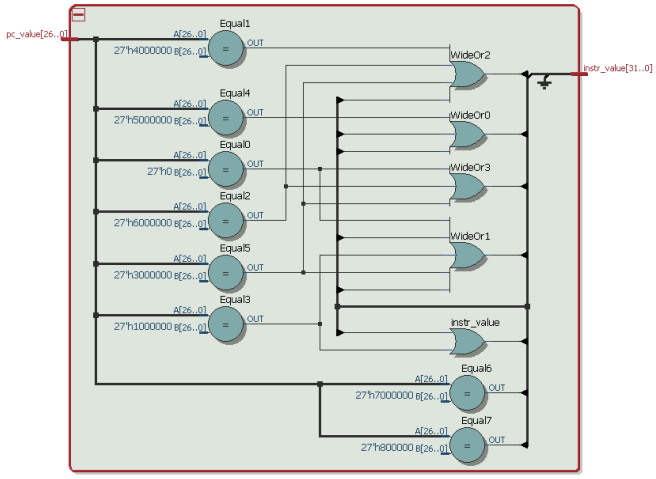


Figure 7 ROM design

As you can see, the ROM is simply to select to output different instructions based on the input PC value. The code is shown below in italic.

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity rom is
port (
pc_value : in addr ;
instr_value : out iword
) ;
end rom ;

architecture rtl of rom is
begin
with conv_integer(pc_value) select instr_value <=
"00010000000000000000000000000001" when 0, -- LOADI 1
```

"00001000000000000000000000000000" when 1, -- STORE 0

"00000000000000000000000000000000" when 2, -- LOAD 0

"00001000000000000000000000000001" when 3, -- STORE 1

"00010000000000000000000000000010" when 4, -- LOADI 2

"00100000000000000000000000000000" when 5, -- NOT

"00011000000000000000000000000001" when 6, -- ADD 1

"00111000000000000000000000000110" when 7, -- JN 6

"00110000000000000000000000001000" when 8, -- JZ 8

"00000000000000000000000000000000" when others ;

end rtl ;

The size of this ROM is 4 giga bits (2^27 * 8). The code above is our testing program in the ROM.

1.2.RAM

RAM is random access memory. In this design, we use RAM as data memory to store temporary data. The design of RAM is shown on Figure 8.

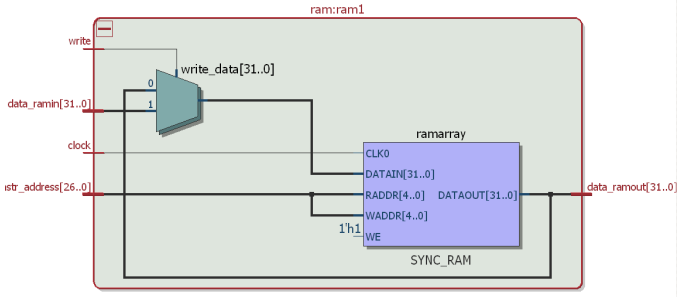


Figure 8 RAM Design

This RAM is basically an array of 32 data words and each data word is 32 bit long. Therefore, the size of RAM is 1024 bits.

When we need to read data from the RAM, RAM will find the data based on the instruction address input.

When we need to write data to the RAM, the write signal will be enabled and the data from input will send to the specified address. Every read or write action is triggered on positive edge. The code of RAM design is shown below in italic.

library ieee ;

use ieee.std_logic_1164.all ;

use ieee.std_logic_arith.all ;

use work.defines.all ;

entity ram is

port (

data_ramin : in dword ;

instr_address : in addr ;

write, clock : in std_logic ;

data_ramout : out dword

) ;

end ram ;

architecture rtl of ram is

type dataarray is array (31 downto 0) of dword ;

signal ramarray : dataarray ;

signal read_data, write_data : dword ;

begin

read_data <= ramarray (conv_integer(unsigned(instr_address))) ;

write_data <= data_ramin when write = '1' else read_data;

process(clock)

begin

if clock'event and clock = '1' then

ramarray(conv_integer(unsigned(instr_address)))

<= write_data ;

end if ;

end process ;

data_ramout <= read_data ;

end rtl ;

1.3.ALU

Arithmetic logic unit (ALU) is a digital circuit that performs integer arithmetic and logical operations. It is the core of the CPU core.

The code of ALU design are shown below in italic.

library ieee ;

```
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity alu is
port (
data_aluin : in dword ;
instr_address : in addr ;
alu_opcode : in alu_opcode ;
clock : in std_logic ;
data_aluout : out dword ;
zero, negative : out std_logic
) ;
end alu ;

architecture rtl of alu is
signal a, nexta : dword ;
begin
with alu_opcode select nexta <=
data_aluin when load,
unsigned("00000" & instr_address) when loadi,
data_aluin + a when addop,
--1 + a when addop,
unsigned(not
std_logic_vector(a)) when notop,
unsigned(std_logic_vector(data_aluin) and
std_logic_vector(a)) when andop,
a when others ;
zero <=
'1' when a = 0 else
'0' ;
negative <= a(26) ;
process(clock)
begin
if clock'event and clock = '1' then
a <= nexta ;
end if ;
end process ;
data_aluout <= a ;
end rtl ;
```

This ALU utilizes several operations including ADD, NOT, LOAD, LOADI, STORE. It also judge if the data in A register is zero or negative for operation JUMP.

The operations of LOAD, LOADI and STORE are important to a RISC CPU. In RISC simple designs, most instructions are of uniform length and similar structure, arithmetic operations are restricted to CPU registers and only separate load and store instructions access memory.

Therefore, we implement those important basic operations for RISC CPU.

1.4.Program Counter (PC)

In our design, PC is mainly to control the ROM to run different instruction flow. Normally, PC will run instruction one by one in ROM. If PC gets a JUMP signal from inputs, it will read respective instructions. The code of PC design is shown below in italic.

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity pc is
port (
instr_address : in addr ;
pc_opcode : in pc_opcode ;
reset : in std_logic ;
clock : in std_logic ;
pc_value : out addr
) ;
end pc ;

architecture rtl of pc is
signal pc, nextpc, nextaddr : addr ;
begin
with pc_opcode select nextaddr <=
pc + 1 when incr,
instr_address when jump,
pc when others ;
nextpc <=
conv_unsigned(0,addr'length) when
reset = '1' else
nextaddr ;
```

```
process(clock)
begin
if clock'event and clock = '1' then
pc <= nextpc ;
end if ;
end process ;
pc_value <= pc ;
end rtl ;
```

1.5.Decoder

Decoder reads every instruction from PC and controls the Data path. It has two duties. One is to decode the instruction operation code; the other is to decode the pc operation code. The code of decoder design is shown below in italic and the design digram is shown on Figure 9.

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity decoder is
port (
instr_value : in iword ;
zero, negative : in std_logic ;
alu_opcode : out opcode ;
pc_opcode : out pc_opcode ;
write : out std_logic
) ;
end decoder ;

architecture rtl of decoder is
signal op : opcode ;
begin
op <= instr_value(31 downto 27) ;
alu_opcode <= op ;
pc_opcode <=
jump when ( op = jz and zero = '1' ) else
jump when ( op = jn and negative = '1' ) else
incr ;
write <= '1' when op = store else '0' ;
end rtl ;
```

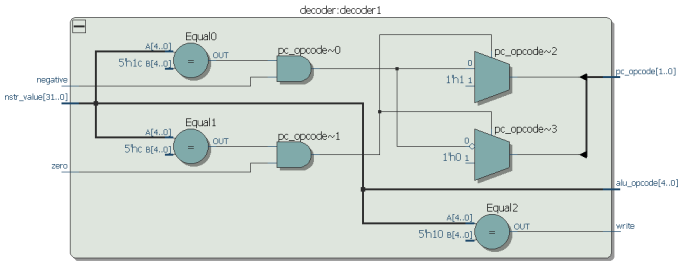


Figure 9 Decoder Design

VI.Development Tools

To simplify the process to build the tool environment and save more time on the project, we choose to use the whole set of development tools from Altera.

For IDE, we choose Quartus II Web Edition because it is free. It is powerful for analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli.

Because Quartus II hase to choose the target device for development. We choose Arria II GX FPGA as target device.

For simulation, we choose ModelSim-Altera Starter Edition which is also free to public.

Those tools are user-friendly and easy to use compared to most open source tools on Linux. However, we find it impossible to save the stimuli setting in ModerSim-Altera during after simulation. That means every time users have to set the stimuli again. It is a waste of time during testing.

VII.Challenge Problems

As we are lack of experience, it is difficult to set up the whole set of development tools at the beginning. Moreover, the process of development of HDL language is really different from software programming.

While we are doing simulation of our program, the result is different from what we expect. Then we find out it is the problem of ADD operation. We thought there is something wrong with our implementation so we check it again and again and find nothing. Finally, we found it is not the problem

of hardware design, it is the problem of our software application in the ROM.

VIII.Simulation Results

For simulation, we embed a testing program in ROM. The program is shown below in italic.

0 LOADI 1
1 STORE 0
2 LOAD 0
3 STORE 1
4 LOADI 2
5 NOT
6 ADD 1
7 JN 6
8 JZ 8

When PC points to address 0 after reset, CPU loads immediate data 1 to A register. So we can see data_alu outputs 1.

Then PC points to address 1. CPU stores the data in A register to the 0 address of RAM. So now RAM [0] stores data 1.

Then PC points to address 2. CPU loads data from RAM [0] and gets what we just store.

Then PC points to address 3, CPU copies the data from RAM [0] to RAM [1] actually.

Then PC points to address 4. CPU loads immediate data 2 to A register.

Next PC points to address 5. CPU make 0x00000002 to 0xFFFFFFFD with NOT operation.

Next PC points to address 6. CPU adds the data in RAM [1] which is 1 to data in A register and stores the result to A register. So dat in the A register will become 0xFFFFFFFE.

Next PC points to address 7. If the data in A is negative, PC will jump to address 6. So the data in A will add 1 each time until it is not a negative number any more.

Finally, if the data is not negative, PC will points to address 8. Because now the data is zero, it will keep jumping back to address 8 and stops there.

As the figure shown on Appendix B Simulation Result, the results satisfies what we expect.

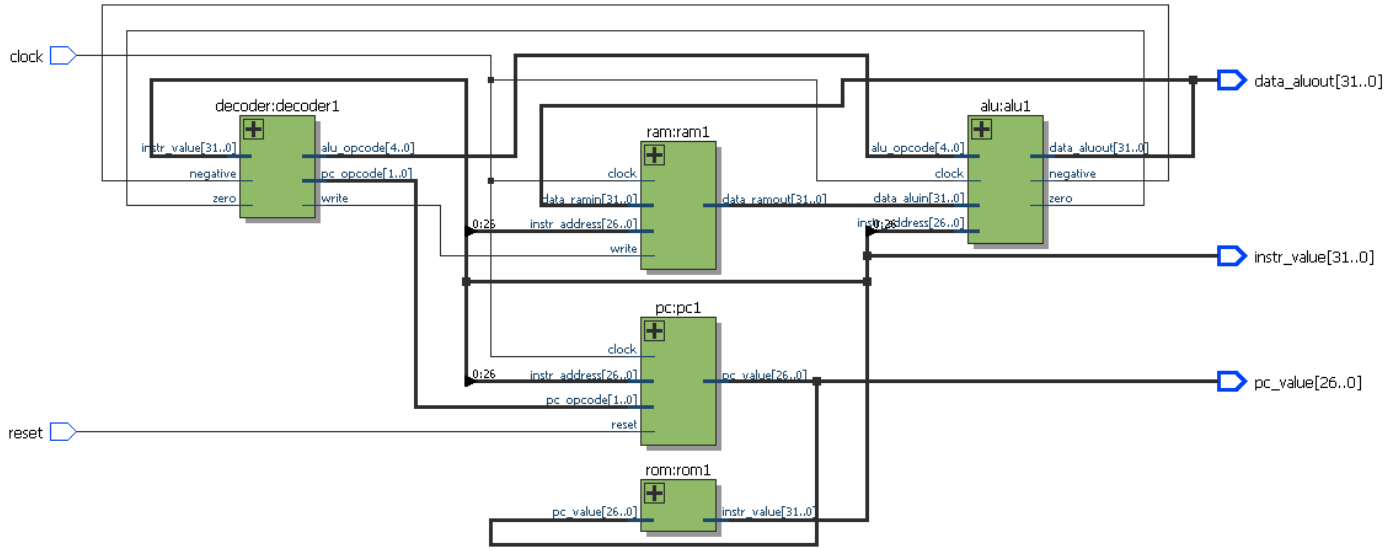
IX.Conclusion

This project successfully implements a 32-bit RISC CPU. As the CPU is extendable, we can set more instructions to enrich its function. By executing a testing application, we find the simulation results as what we suppose. In future, the pipeline operation fetch will be embedded for higher efficient execution.

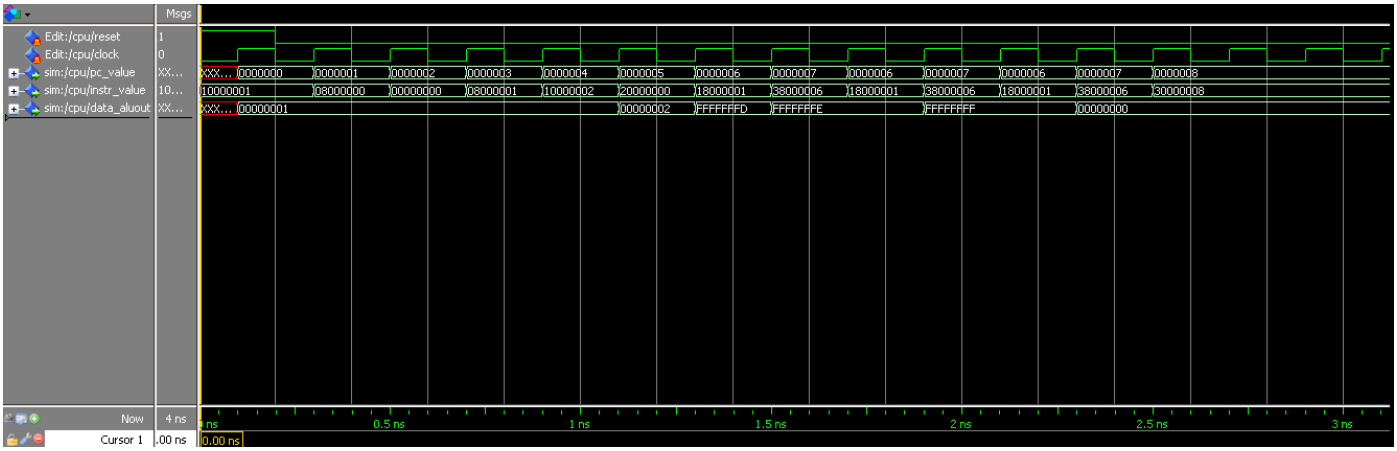
References

1. Pei-lin, W. A. N. G. "Design of an 8-bit CISC CPU based on FPGA." Coal Technology 10 (2010): 094.
2. Gal, Ryszard, et al. "FPGA implementation of 8-bit RISC microcontroller for embedded systems." Mixed Design of Integrated Circuits and Systems (MIXDES), 2011 Proceedings of the 18th International Conference. IEEE, 2011.
3. Jeong, Geun-young, and Ju-sung Park. "Design of 32-bit RISC processor and efficient verification." Science and Technology, 2003. Proceedings KORUS 2003. The 7th Korea-Russia International Symposium on. Vol. 2. IEEE, 2003.
4. Juergen Sauermann, How to design your own CPU on FPGAs with VHDL, http:// www.ece.ubc.ca/~edc/379/lectures/

Appendix A Top-level Design of CPU



Appendix B Simulation Result



Appendix C VHDL Code of Top Level

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;
use work.components.all ;
```

```
entity cpu is
port (
reset, clock : in std_logic ;
pc_value : out addr ;
instr_value : out iword ;
data_aluout : out dword
) ;
end cpu ;

architecture rtl of cpu is
signal ip : addr ;
signal instr : iword ;
signal ramout : dword ;
signal acc : dword ;
signal zero, negative : std_logic ;
signal aluop : alu_opcode ;
signal pcop : pc_opcode ;
signal write : std_logic ;
signal add : addr ;
begin
add <= unsigned(instr(26 downto 0)) ;
pc1: pc port map ( add, pcop, reset, clock, ip ) ;
rom1: rom port map ( ip, instr ) ;
ram1: ram port map ( acc, add, write, clock, ramout ) ;
alu1: alu port map ( ramout, add, aluop, clock, acc,
zero, negative) ;
decoder1:
decoder port map ( instr, zero, negative,
aluop, pcop, write ) ;
pc_value <= ip ;
instr_value <= instr ;
data_aluout <= acc ;
end rtl ;
```

Appendix D VHDL Code of Define Parameters

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
package defines is
subtype dword is unsigned (31 downto 0) ;
subtype iword is std_logic_vector (31 downto 0) ;
subtype addr is unsigned (26 downto 0) ;
```

```
subtype opcode is std_logic_vector (4 downto 0) ;
subtype alu_opcode is std_logic_vector (4 downto 0) ;
subtype pc_opcode is std_logic_vector (1 downto 0) ;
constant load  : opcode := "00000" ;
constant store : opcode := "00001" ;
constant loadi : opcode := "00010" ;
constant addop : opcode := "00011" ;
constant notop : opcode := "00100" ;
constant andop : opcode := "00101" ;
constant jz : opcode := "00110" ;
constant jn : opcode := "00111" ;
constant hold : pc_opcode := "00" ;
constant incr : pc_opcode := "01" ;
constant jump : pc_opcode := "10" ;
end defines;
```

Appendix E VHDL Code of ROM

```
-- ROM
-- Size = 2^27 * 8 = 4 giga bits

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity rom is
port (
pc_value : in addr ;
instr_value : out iword
) ;
end rom ;

architecture rtl of rom is
begin
with conv_integer(pc_value) select instr_value <=
"00010000000000000000000000000001" when 0, -- LOADI 1
"00001000000000000000000000000000" when 1, -- STORE 0
"00000000000000000000000000000000" when 2, -- LOAD  0
"00001000000000000000000000000001" when 3, -- STORE 1
"00010000000000000000000000000010" when 4, -- LOADI 2
"00100000000000000000000000000000" when 5, -- NOT
"00011000000000000000000000000001" when 6, -- ADD   1
"00111000000000000000000000000110" when 7, -- JN    6
```

```
"00110000000000000000000000001000" when 8, -- JZ    8
"00000000000000000000000000000000" when others ;

--"00010000000000000000000000000010" when 0, -- LOADI 2
--"00010000000000000000000000000001" when 1, -- LOADI 1
--"00011000000000000000000000000000" when 2, -- ADD          1
--"00011000000000000000000000000000" when 3, -- ADD          1
--"00000000000000000000000000000000" when others ;

end rtl ;
```

Appendix F VHDL Code of RAM

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity ram is
port (
data_ramin : in dword ;
instr_address : in addr ;
write, clock : in std_logic ;
data_ramout : out dword
) ;
end ram ;

architecture rtl of ram is
type dataarray is array (31 downto 0) of dword ;
signal ramarray : dataarray ;
signal read_data, write_data : dword ;
begin
read_data <= ramarray (conv_integer(unsigned(instr_address))) ;
write_data <= data_ramin when write = '1' else read_data;
process(clock)
begin
if clock'event and clock = '1' then
ramarray(conv_integer(unsigned(instr_address)))
<= write_data ;
end if ;
end process ;
data_ramout <= read_data ;
end rtl ;
```

Appendix G VHDL Code of ALU

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity alu is
port (
data_aluin : in dword ;
instr_address : in addr ;
alu_opcode : in alu_opcode ;
clock : in std_logic ;
data_aluout : out dword ;
zero, negative : out std_logic
) ;
end alu ;

architecture rtl of alu is
signal a, nexta : dword ;
begin
with alu_opcode select nexta <=
data_aluin when load,
unsigned("00000" & instr_address) when loadi,
data_aluin + a when addop,
--1 + a when addop,
unsigned(not
std_logic_vector(a)) when notop,
unsigned(std_logic_vector(data_aluin) and
std_logic_vector(a)) when andop,
a when others ;
zero <=
'1' when a = 0 else
'0' ;
negative <= a(26) ;
process(clock)
begin
if clock'event and clock = '1' then
a <= nexta ;
end if ;
end process ;
data_aluout <= a ;
end rtl ;
```

Appendix H VHDL Code of PC

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;

entity pc is
port (
instr_address : in addr ;
pc_opcode : in pc_opcode ;
reset : in std_logic ;
clock : in std_logic ;
pc_value : out addr
) ;
end pc ;

architecture rtl of pc is
signal pc, nextpc, nextaddr : addr ;
begin
with pc_opcode select nextaddr <=
pc + 1 when incr,
instr_address when jump,
pc when others ;
nextpc <=
conv_unsigned(0,addr'length) when
reset = '1' else
nextaddr ;
process(clock)
begin
if clock'event and clock = '1' then
pc <= nextpc ;
end if ;
end process ;
pc_value <= pc ;
end rtl ;
```

Appendix I VHDL Code of Decoder

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.defines.all ;
```

```
entity decoder is
port (
instr_value : in iword ;
zero, negative : in std_logic ;
alu_opcode : out opcode ;
pc_opcode : out pc_opcode ;
write : out std_logic
) ;
end decoder ;

architecture rtl of decoder is
signal op : opcode ;
begin
op <= instr_value(31 downto 27) ;
alu_opcode <= op ;
pc_opcode <=
jump when ( op = jz and zero = '1' ) else
jump when ( op = jn and negative = '1' ) else
incr ;
write <= '1' when op = store else '0' ;
end rtl ;
```