Faculty of Engineering & Technology Electrical & Computer Engineering Department

ENCS3310: Advanced Digital Systems Design

Multi Cycle Processor Design Project Report

Prepared by:

Taleed Mahmoud Hamadneh 1220006

Qasim Nidal Batrawi 1220204

Instructor: Dr. Ayman Hroub

Section: 1

Date: 17 Aug 2024

Table of Contents

Brief introduction and background

This project shows the design of a very simple multi-cycle processor in Verilog. A multi-cycle processor is a processor that executes an instruction in multiple clock cycles. Moreover, in such processors, each instruction is given only the number of clock cycles required to execute it.

Register File

The register file contains 16 16 bit registers, this file has one write port (Rd) which refers to the destination register and two read ports (Rs1 & Rs2) that are considered as the sources registers.

The register file was used in many stages:

1- Fetching the operands for the execution stage (ALU operations)
2- Storing the result of the ALU
3- Getting the content of the address register in the load stage and store stage
4- Storing the data out from the memory in the store stage

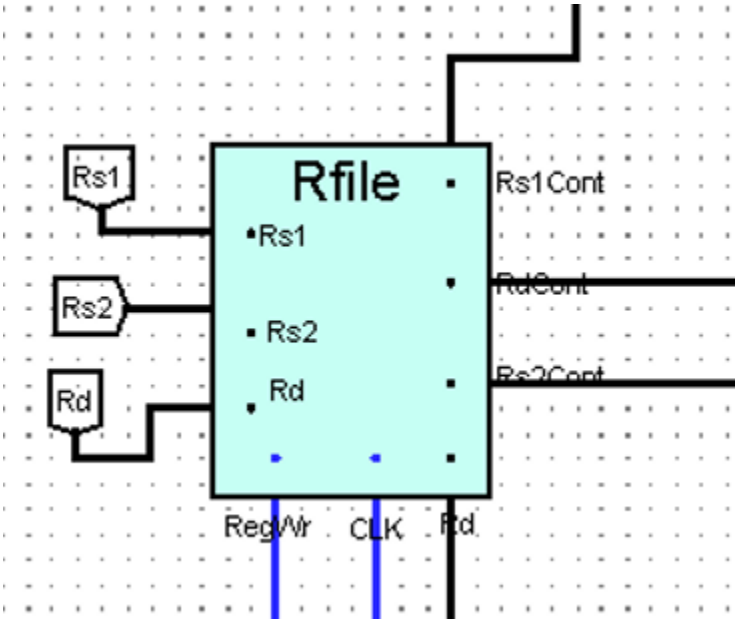

Figure 1

ALU

This processor has a 16 bit ALU to operate the following operations: ADD, SUB, OR, AND, XOR.

The table below describes the functionality of the ALU.

Instruction	Meaning	Opcode
ADD Rd, Rs1, Rs2	Reg [Rd] = Reg [Rs1] + Reg [Rs2]	0000
SUB Rd, Rs1, Rs2	Reg [Rd] = Reg [Rs1] - Reg [Rs2]	0001
AND Rd, Rs1, Rs2	Reg [Rd] = Reg [Rs1] & Reg [Rs2]	0010
OR Rd, Rs1, Rs2	Reg [Rd] = Reg [Rs1] | Reg [Rs2]	0011
XOR Rd, Rs1, Rs2	Reg [Rd] = Reg [Rs1] ^ Reg [Rs2]	0100

As shown in the figure below, the ALU has 3 inputs: two of them are fetched from the register file, and the third input, the opcode, determines the operation to be performed between the two inputs.

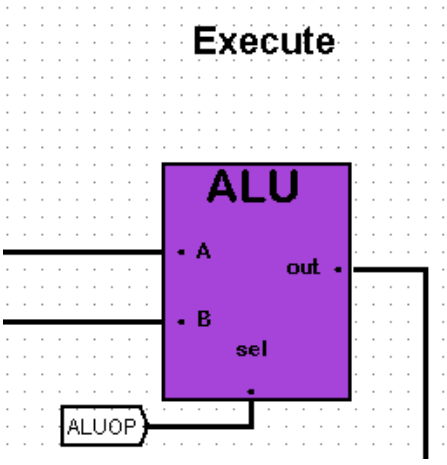

Figure 2

Control Unit

The control unit is the most important component in the design due to controlling the flow of the processor, and determining the stages of the instruction.

The control unit has one input which is the opcode of the instruction. It has 4 output control signals:

- RegWr: Register file write enable
- ALUOP: ALU operation code
- MemRd: Data memory read enable
- MemWr: Data memory write enable

If the operation is one of the ALU operations, then the RegWr will be enabled, MemRd & MemWr will be disabled, ALUOP will depend on the operation type.

If the operation is Load, then the RegWr will be enabled (to write the result back), MemRd will be enabled, ALUOP will be considered as don't care but the MemWr will be disabled.

If the operation is Store, then the RegWr will be disabled, MemRd will also be disabled, ALUOP will be considered as don't care. However the MemWr will be Enabled.

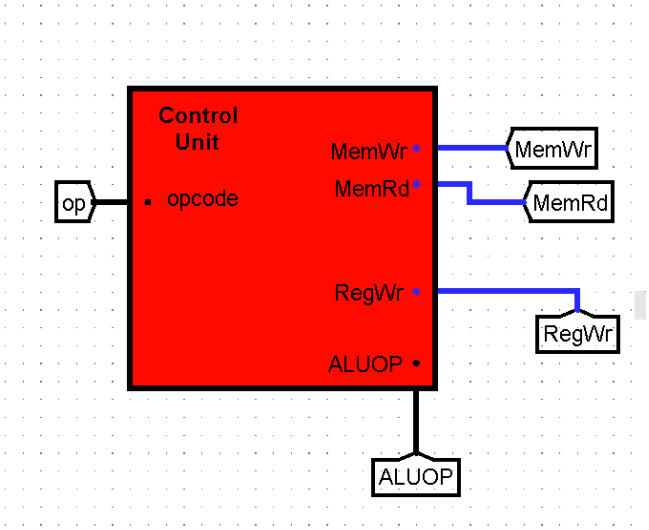

Figure 3

Memory

In our processor, there are two types of memory: Instruction Memory (read-only), Data Memory (read-write).

- The Instruction Memory is used to fetch the instruction from it, The output of the memory is the read data.
  It receives 16 bit from the PC to determine the location of the instruction and outputs on of the two instruction types, R-type, M-type, to be divided by a splitter to obtain : Opcode, Rd,Rs1,Rs2.

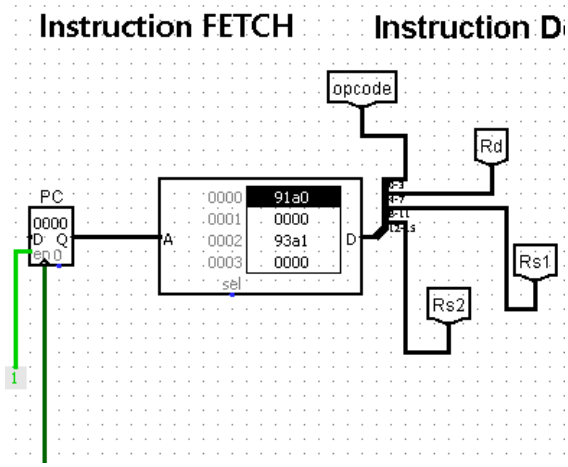

Figure 4

- The Data Memory is used to store the result of the ALU or to load data to the register file. The store and load operations are controlled by two inputs control signals of the memory and are executed on the rising edge of the third input “clk”. The output of the memory is the data stored or loaded.

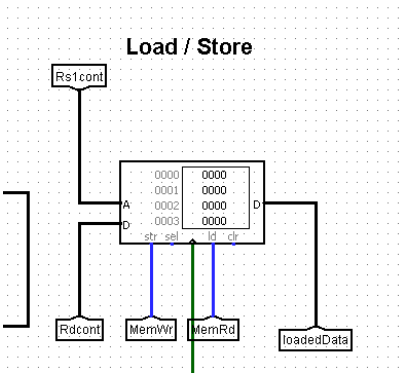

Figure 5

Design philosophy

Processor

```
module Processor(clk , IR , Loaded , Stored , Out) ;

    // define the states
    `define FETCH 3'b000
    `define DECODE 3'b001
    `define EXECUTE 3'b010
    `define RESSTORE1 3'b011
    `define RESSTORE2 3'b100
    `define RESSTORE3 3'b101

    // define alu operation
    `define OPADD 4'b0000
    `define OPSUB 4'b0001
    `define OPAND 4'b0010
    `define OPOR 4'b0011
    `define OPXOR 4'b0100
    `define OPLOAD 4'b0101
    `define OPSTORE 4'b0110

    // define for control signals
    `define ADD 6'b100000
    `define SUB 6'b100001
    `define AND 6'b100010
    `define OR 6'b100011
    `define XOR 6'b100100
    `define LOAD 6'b110000
    `define STORE 6'b001000

    input clk ; // input clock

    output reg [15:0] Out , Stored , Loaded ; // Out: the result of the whole processor. Stored; the data stored in memory. Loaded: the data loaded from memory
    output reg [15:0] IR ; // Instruction register to store the fetched instrction
```

Figure 6

The processor takes one input (clk), and outputs 4 reg outputs:

Loaded, Stored => used in the load or store operation

Out => used in the ALU operations

IR=> its only used to show the content of the IR in the simulation

** The figure above shows some defined parameters used in the design.

```
    reg RegWr , MemRd , MemWr ; // control signals
    reg [2:0] ALUOP ; // control signals

    reg [3:0] source1 , source2 ; // this is the address for the source register to read data from it
    reg [3:0] destination ; // this is the address for the dest. reg. to store data in it
    reg [3:0] opcode ; // this is the instruction opcode

    reg [15:0] source1Content , source2Content ; // to fetch the data inside the registers
    reg [15:0] destinationContent ; // this is to store the destinationn content
    reg [15:0] pc ; // program counter

    reg [2:0] state ; // to determine the stage of the instruction (fetch, decode, execute, store)

    reg [15:0] AluOut ; // the output of the alu
```

Figure 7

The figure above shows the used regs in our processor:

The control signals, the ISA format, the registers' contents, PC, the state, and the ALU output.

```
// the regiser file with its initail values
reg [15:0] registers [15:0] ;
initial begin
    registers[0]  = 16'h0010;
    registers[1]  = 16'h0020;
    registers[2]  = 16'h0030;
    registers[3]  = 16'h00AA;
    registers[4]  = 16'h1C3A;
    registers[5]  = 16'h0000;
    registers[6]  = 16'h22E0;
    registers[7]  = 16'h1C86;
    registers[8]  = 16'h22DA;
    registers[9]  = 16'h0414;
    registers[10] = 16'h1A32;
    registers[11] = 16'h0102;
    registers[12] = 16'h1CBA;
    registers[13] = 16'h0CDE;
    registers[14] = 16'h3994;
    registers[15] = 16'h1984;
end
```

Figure 8

```
// the data memory with its initial value
reg [15:0] DataMemory [15:0] ;
initial begin
    pc = 0 ;
    state = 0 ;
    DataMemory[0] = 16'h0001 ;
    DataMemory[1] = 16'h0020 ;
    DataMemory[2] = 16'h0000 ;
    DataMemory[3] = 16'h00AA ;
    DataMemory[4] = 16'h1C3A ;
    DataMemory[5] = 16'h0000 ;
    DataMemory[6] = 16'h22E0 ;
    DataMemory[7] = 16'h0000 ;
    DataMemory[8] = 16'h22DA ;
    DataMemory[9] = 16'h0000 ;
    DataMemory[10] = 16'h1A32 ;
    DataMemory[11] = 16'h0102 ;
    DataMemory[12] = 16'h000 ;
    DataMemory[13] = 16'h0CDE ;
    DataMemory[14] = 16'h0000 ;
    DataMemory[15] = 16'h1984 ;
end
```

Figure 9

```
// the instruction memory wiht its initail values
reg [15:0] InstructionMemory [15:0] ;
initial begin
    InstructionMemory[0] = 16'h0312 ;
    InstructionMemory[1] = 16'h1534 ;
    InstructionMemory[2] = 16'h5756 ;
    InstructionMemory[3] = 16'h623A ;
    InstructionMemory[4] = 16'h1C3A ;
    InstructionMemory[5] = 16'h1180 ;
    InstructionMemory[6] = 16'h22E0 ;
    InstructionMemory[7] = 16'h1C86 ;
    InstructionMemory[8] = 16'h22DA ;
    InstructionMemory[9] = 16'h0414 ;
    InstructionMemory[10] = 16'h1A32 ;
    InstructionMemory[11] = 16'h0102 ;
    InstructionMemory[12] = 16'h1CBA ;
    InstructionMemory[13] = 16'h0CDE ;
    InstructionMemory[14] = 16'h3994 ;
    InstructionMemory[15] = 16'h1984 ;
end
```

Figure 10

These are the initial contents of the register file, Data memory, Instruction memory respectively.

```
always @ (posedge clk)  begin

    case (state)

        `FETCH: begin  // fetch the instruction

            IR = InstructionMemory[pc] ;

            pc = pc + 1 ; // update the pc to point to the next instruction

            state = 3'b001 ; // move to next state (decode)

        end

        `DECODE: begin //decode and initialize control signals

            {opcode , destination , source1 , source2} = IR ;

            case (opcode)

                `OPADD: begin RegWr = 1 ; MemRd=0 ; MemWr=0 ; ALUOP=0 ; end
                `OPSUB: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=1 ;   end
                `OPAND: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=2 ; end
                `OPOR: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=3 ; end
                `OPXOR: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=4 ; end
                `OPLOAD: begin RegWr=1 ; MemRd=1 ; MemWr=0 ; ALUOP=0 ; end // 0 is any value for ALUOP
                `OPSTORE: begin RegWr=0 ; MemRd=0 ; MemWr=1 ; ALUOP=0 ; end // 0 is any value for ALUOP
                default: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=0 ; end  // default case is add

            endcase

            state = 3'b010 ; // move to next state (execute)

        end
```

Figure 11

All the stages are implemented in an always block that is sensitive at the rising edge of the clock, this figure shows the fetch and decode stages, at the fetch stage the IR will store the content of the current instruction to be executed, and the pc will be incremented. At the decode stage, the instruction will be divided into four 4 bits parts, also the control signals are generated with their appropriate values depending on the opcode.

```
`EXECUTE: begin // execute the operation depending on the control signals

    case ({RegWr , MemRd , MemWr , ALUOP})

        `ADD: begin // alu will perform + operation

                // fetch the content of the sources
                 source1Content = registers[source1] ;
                 source2Content = registers[source2] ;

                 AluOut = source1Content + source2Content ;

                 state = 3'b011 ; // move to next state (store the result in the destination)

        end


        `SUB: begin  // alu will perform - operation

                source1Content = registers[source1] ;
                source2Content = registers[source2] ;

                AluOut = source1Content - source2Content ;

                state = 3'b011 ;   // move to next state (store the result in the destination)

        end

        `AND: begin   // alu will perform & operation

            source1Content = registers[source1] ;
            source2Content = registers[source2] ;

            AluOut = source1Content & source2Content ;

            state = 3'b011 ;        // move to next state (store the result in the destination)
```

Figure 12

```
`OR: begin   // alu will perform | operation

    source1Content = registers[source1] ;
    source2Content = registers[source2] ;

    AluOut = source1Content | source2Content ;

    state = 3'b011 ; // move to next state (store the result in the destination)

end
`XOR: begin // alu will perform ^ operation

    source1Content = registers[source1] ;
    source2Content = registers[source2] ;

    AluOut = source1Content ^ source2Content ;

    state = 3'b011 ; // move to next state (store the result in the destination)

end


`LOAD: begin

    // get the content of the source1
    source1Content = registers[source1] ;

    state = 3'b100 ; // move to next state (load from memory)

end

`STORE: begin

    // get the content of the source1
    source1Content = registers[source1] ;

    state = 3'b101 ; // move to next state (store to memory)

end
```

Figure 13

```
        `STORE: begin

            // get the content of the source1
            source1Content = registers[source1] ;

            state = 3'b101 ; // move to next state (store to memory)

        end


        default: begin // default case is return to fetch

                 state = 3'b000 ;

        end


    endcase

end
```

Figure 14

At the execute stage, the control signals will determine the operation to be done using case statements.

The first five cases are: ADD, SUB, AND, OR, XOR. In these cases, the contents of the source registers are fetched and the operation were performed. Here, the next stage is “RESSTORE1”.

The last two cases are: LOAD, STORE. In both of them, the content of the source register are fetched. In LOAD, the next stage is “RESSTORE2”. In STORE, the next stage is “RESSTORE3”.

The default state of the case statement is to return to the fetch stage.

```
        `RESSTORE1: begin // after (add sub or xor and)
                registers[destination] = AluOut ; // store the result in the destination
                Out = AluOut ;
                state = 3'b000 ;  // return to fectch stage
        end
        `RESSTORE2: begin // after load
                    registers[destination] = DataMemory[source1Content] ; // load data from memory and store in the destination
                    Loaded = registers[destination] ; // load output
                    Out = DataMemory[source1Content] ;
                    state = 3'b000 ; // return to fetch
        end
        `RESSTORE3: begin // after store
            DataMemory[source1Content] = registers[destination] ; // store the destination content in the memory
            Stored = DataMemory[source1Content] ; // store output
            Out = registers[destination] ;
            state = 3'b000 ; // retrun to fech stage
        end
        default: begin // default case is return to fetch
             state = 3'b000 ;
        end

    endcase
```

Figure 15

After the ALU operation was performed, the state will be “RESSTORE1”, which stores the result of the operation in the destination register.

After the LOAD operation, the state will be “RESSTORE2”, which stores the content of the source1 register in the destination register.

After the STORE operation, the state will be “RESSTORE3”, which stores the destination register content in the memory.

TestBench

```
1  module TB;
2
3      reg clk ;
4      wire [15:0] Out ;
5      wire [15:0] IR , Stored , Loaded ;
6
7      Processor P1(clk , IR , Loaded , Stored , Out) ;
8
9      initial begin
10
11         clk = 0 ;
12
13         repeat(64) begin
14             #10ns clk = ~clk;
15         end
16
17     end
18
19
20 endmodule
```

Figure 16

The testbench has a reg clock that is the input of the module instance (P1), and 4 wires (IR, Stored, Loaded, Out) which are the output of the module instance (P1).

This testbench change the value of the clock every 10 ns to obtain the rising and falling edges of it.

Block diagram

The Processor Block

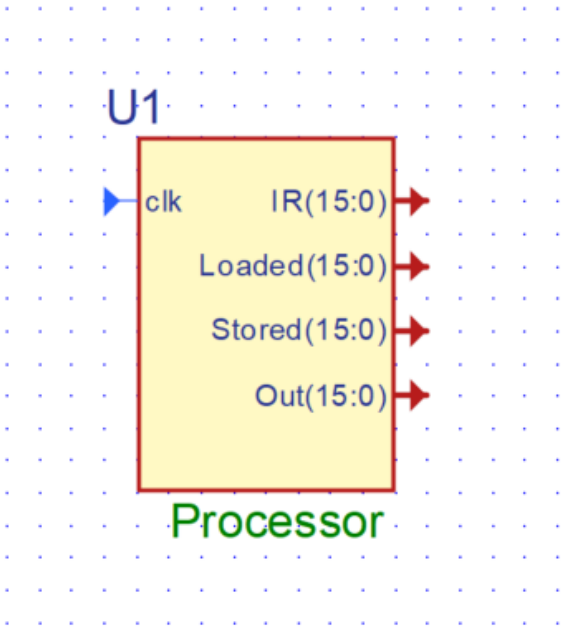

Figure 17

The interior processor design blocks

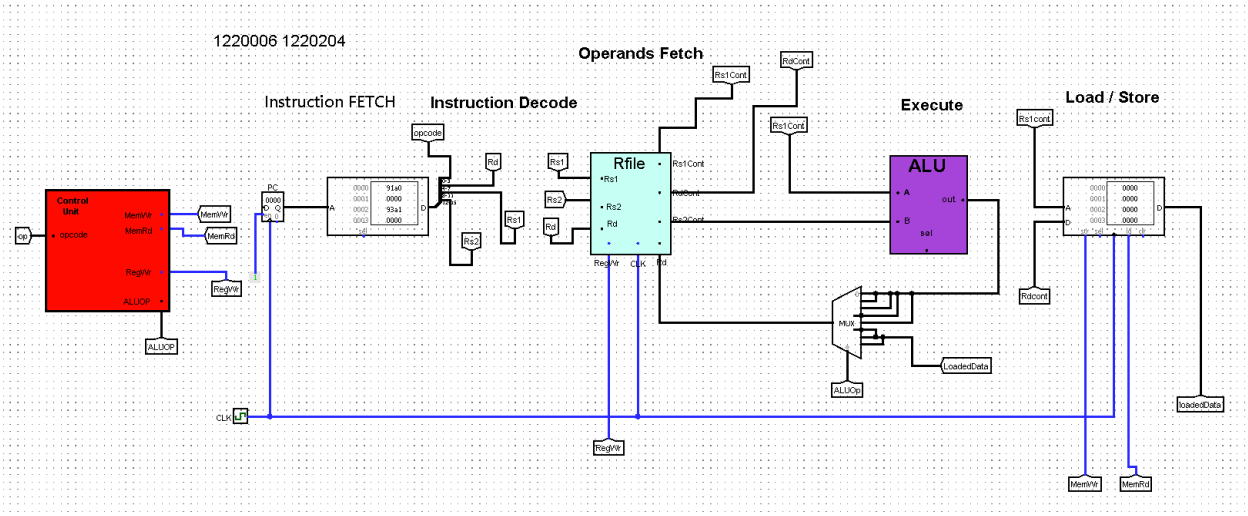

Figure 18

Test cases and Simulation results

The following program was added to the instruction memory and will be executed :

0x0312 0x1534 0x5956 0x695A

Addition

The instruction = 0x0312 => (opcode = 0, Rd=3, Rs1=1, Rs2=2)
** since the opcode=0, so the operation is Addition.
** The content of Rs1=0x0020 , Rs2=0x0030
** The expected result will be out = 0x0050 , loaded and stored are don't care (no memory access).

Figure 19

- The result was as expected.

Subtraction

The instruction = 0x1534 => (opcode = 1, Rd=5, Rs1=3, Rs2=4)
** since the opcode=1, so the operation is Subtraction.
** The content of Rs1=0x00AA , Rs2=0x1C3A
** The expected result will be out = Rs2-Rs1 = 0x1C3A – 0x00AA = 0xE416 , loaded and stored are don't care (no memory access).

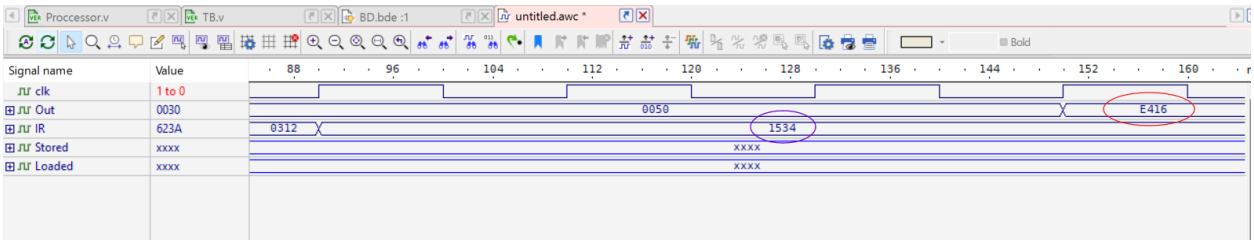

Figure 20

- The result was as expected

Memory Load

The instruction = 0x5956 => (opcode = 5, Rd=9, Rs1=5, Rs2=6)
** Rs2 is unused (M-type)
** since the opcode=5, so the operation is memory Load.
** The content of Rs1=0x0001
**The content of data memory in 0x0000 is 0x0020
** The expected result will be: loaded=0x0020.

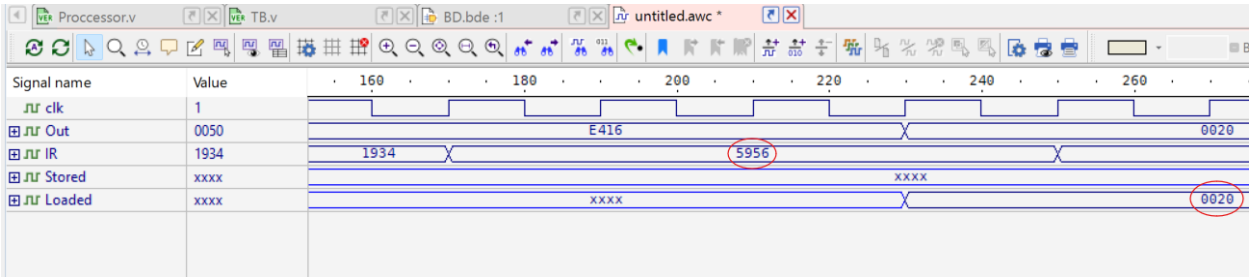

Figure 21

- The result was as expected

Memory Store

The instruction = 0x695A => (opcode = 6, Rd=9, Rs1=5, Rs2=A)
** Rs2 is unused (M-type)
** since the opcode=6, so the operation is memory Store.
** The content of Rs1=0x0001 (memory location)
**The content of Rd=0x0020 due to the past load operation (data to be stored in the memory location)
** The expected result will be: the stored=0x0020

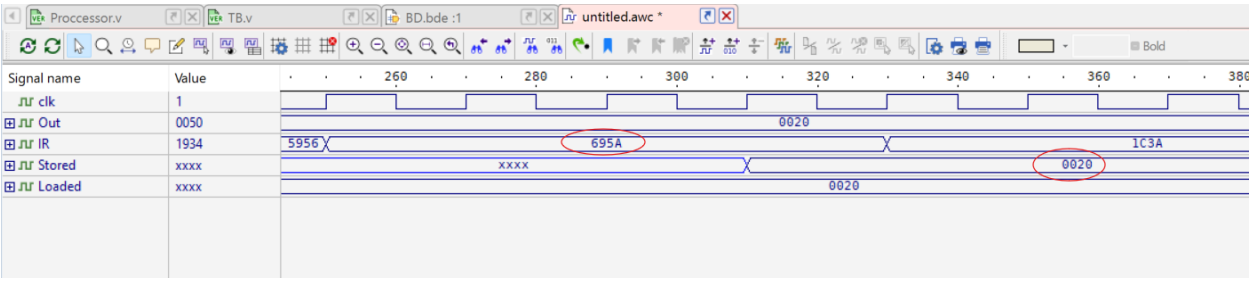

Figure 22

- The result was as expected

Conclusion and Future works

In conclusion, a multi cycle processor has better cost and performance compared to a single processor because the single cycle processor execute the instruction in one clock cycle, so the processor needs a long clock cycle to finish executing. We have used behavioral Verilog and all components of the processor works correctly, while the multi cycle can divide the execution of the instruction into 4 stages (4 clock cycles): fetch, decode, execute ALU, and store which is better for the processor since it doesn't need a long clock cycle.

The processor was designed using the behavioral modeling.

Some future updates on the processor may be obtained by:

1- Improve Memory Access by using caches. This reduces the time spent waiting for memory, speeding up the processor.

2- Increase the number of ports of the register file. This reduces the time needed to access data.

Appendix

Processor

```
module Processor(clk , IR , Loaded , Stored , Out) ;


// define the states
`define FETCH 3'b000
`define DECODE 3'b001
`define EXECUTE 3'b010
`define RESSTORE1 3'b011
`define RESSTORE2 3'b100
`define RESSTORE3 3'b101

// define alu operation
`define OPADD 4'b0000
`define OPSUB 4'b0001
`define OPAND 4'b0010
`define OPOR 4'b0011
`define OPXOR 4'b0100
`define OPLOAD 4'b0101
`define OPSTORE 4'b0110

// define for control signals
`define ADD 6'b100000
`define SUB 6'b100001
`define AND 6'b100010
`define OR 6'b100011
`define XOR 6'b100100
`define LOAD 6'b110000
```

```
`define STORE 6'b001000

input clk ; // input clock

output reg [15:0] Out , Stored , Loaded ; // Out: the result of the whole processor. Stored; the data stored in memory. Loaded: the data loaded from memory

output reg [15:0] IR ; // Instruction register to store the fetched instrction

reg RegWr , MemRd , MemWr ; // control signals

reg [2:0] ALUOP ; // control signals

reg [3:0] source1 , source2 ; // this is the address for the source register to read data from it

reg [3:0] destination ; // this is the address for the dest. reg. to store data in it

reg [3:0] opcode ; // this is the instruction opcode

reg [15:0] source1Content , source2Content ; // to fetch the data inside the registers

reg [15:0] destinationContent ; // this is to store the destinationn content

reg [15:0] pc ;    // program counter

reg [2:0] state ; // to determine the stage of the instruction (fetch, decode, execute, store)

reg [15:0] AluOut ;         // the output of the alu

// the regiser file with its initail values

reg [15:0] registers [15:0] ;

initial begin

registers[0]  = 16'h0010;

registers[1]  = 16'h0020;

registers[2]  = 16'h0030;
```

```
registers[3]  = 16'h00AA;
registers[4]  = 16'h1C3A;
registers[5]  = 16'h0001;
registers[6]  = 16'h22E0;
registers[7]  = 16'h1C86;
registers[8]  = 16'h22DA;
registers[9]  = 16'h0414;
registers[10] = 16'h1A32;
registers[11] = 16'h0102;
registers[12] = 16'h1CBA;
registers[13] = 16'h0CDE;
registers[14] = 16'h3994;
registers[15] = 16'h1984;
end

// the data memory with its initial value
reg [15:0] DataMemory [15:0] ;
initial begin
pc = 0 ;
state = 0 ;
DataMemory[0] = 16'h0001 ;
DataMemory[1] = 16'h0020 ;
DataMemory[2] = 16'h0000 ;
DataMemory[3] = 16'h00AA ;
DataMemory[4] = 16'h1C3A ;
DataMemory[5] = 16'h0000 ;
DataMemory[6] = 16'h0000 ;
DataMemory[7] = 16'h0000 ;
DataMemory[8] = 16'h22DA ;
```

```
DataMemory[9] = 16'h0000 ;
DataMemory[10] = 16'h1A32 ;
DataMemory[11] = 16'h0102 ;
DataMemory[12] = 16'h000 ;
DataMemory[13] = 16'h0CDE ;
DataMemory[14] = 16'h0000 ;
DataMemory[15] = 16'h1984 ;
end

// the instruction memory wiht its initail values
reg [15:0] InstructionMemory [15:0] ;
initial begin
InstructionMemory[0] = 16'h0312 ;
InstructionMemory[1] = 16'h1934 ;
InstructionMemory[2] = 16'h5956 ;
InstructionMemory[3] = 16'h695A ;
InstructionMemory[4] = 16'h1C3A ;
InstructionMemory[5] = 16'h1180 ;
InstructionMemory[6] = 16'h22E0 ;
InstructionMemory[7] = 16'h1C86 ;
InstructionMemory[8] = 16'h22DA ;
InstructionMemory[9] = 16'h0414 ;
InstructionMemory[10] = 16'h1A32 ;
InstructionMemory[11] = 16'h0102 ;
InstructionMemory[12] = 16'h1CBA ;
InstructionMemory[13] = 16'h0CDE ;
InstructionMemory[14] = 16'h3994 ;
InstructionMemory[15] = 16'h1984 ;
end
```

```
always @ (posedge clk)  begin

case (state)

`FETCH: begin  // fetch the instruction

IR = InstructionMemory[pc] ;

pc = pc + 1 ; // update the pc to point to the next instruction

state = 3'b001 ; // move to next state (decode)

end

`DECODE: begin //decode and initialize control signals

{opcode , destination , source1 , source2} = IR ;

case (opcode)

`OPADD: begin RegWr = 1 ; MemRd=0 ; MemWr=0 ; ALUOP=0 ; end
`OPSUB: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=1 ;      end
`OPAND: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=2 ; end
`OPOR: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=3 ; end
`OPXOR: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=4 ; end
`OPLOAD: begin RegWr=1 ; MemRd=1 ; MemWr=0 ; ALUOP=0 ; end // 0 is any value for ALUOP
`OPSTORE: begin RegWr=0 ; MemRd=0 ; MemWr=1 ; ALUOP=0 ; end // 0 is any value for ALUOP
default: begin RegWr=1 ; MemRd=0 ; MemWr=0 ; ALUOP=0 ; end  // default case is add
```

```
endcase

state = 3'b010 ; // move to next state (execute)

end

`EXECUTE: begin // execute the operation depending on the control signals

case ({RegWr , MemRd , MemWr , ALUOP})

`ADD: begin // alu will perform + operation

   // fetch the content of the sources
    source1Content = registers[source1] ;
    source2Content = registers[source2] ;

    AluOut = source1Content + source2Content ;

    state = 3'b011 ; // move to next state (store the result in the destination)

end


`SUB: begin  // alu will perform - operation

   source1Content = registers[source1] ;
   source2Content = registers[source2] ;
```

```
    AluOut = source1Content - source2Content ;

    state = 3'b011 ;   // move to next state (store the result in the destination)

end

`AND: begin   // alu will perform & operation

source1Content = registers[source1] ;
source2Content = registers[source2] ;

AluOut = source1Content & source2Content ;

state = 3'b011 ;              // move to next state (store the result in the destination)

end

`OR: begin          // alu will perform | operation

source1Content = registers[source1] ;
source2Content = registers[source2] ;

AluOut = source1Content | source2Content ;

state = 3'b011 ; // move to next state (store the result in the destination)

end

`XOR: begin // alu will perform ^ operation
```

```
source1Content = registers[source1] ;
source2Content = registers[source2] ;

AluOut = source1Content ^ source2Content ;

state = 3'b011 ; // move to next state (store the result in the destination)

end


`LOAD: begin

// get the content of the source1
source1Content = registers[source1] ;

state = 3'b100 ; // move to next state (load from memory)

end

`STORE: begin

// get the content of the source1
source1Content = registers[source1] ;

state = 3'b101 ; // move to next state (store to memory)

end
```

```
default: begin // default case is return to fetch

    state = 3'b000 ;

end



endcase



end

`RESSTORE1: begin // after (add sub or xor and)

registers[destination] = AluOut ; // store the result in the destination

Out = AluOut ;

state = 3'b000 ;  // return to fectch stage
end

`RESSTORE2: begin // after load

registers[destination] = DataMemory[source1Content] ; // load data from memory and store in the
destination

Loaded = DataMemory[source1Content] ; // load output
```

```
Out = DataMemory[source1Content] ;

state = 3'b000 ; // return to fetch
end

`RESSTORE3: begin // after store

DataMemory[source1Content] = registers[destination] ; // store the destination content in the memory

Stored = registers[destination] ; // store output

Out = registers[destination] ;

state = 3'b000 ; // retrun to fech stage

end

default: begin // default case is return to fetch

state = 3'b000 ;

end


endcase


end
endmodule
```

TestBench

```
module TB;


   reg clk ;

   wire [15:0] Out ;

   wire [15:0] IR , Stored , Loaded ;


   Processor P1(clk , IR , Loaded , Stored , Out) ;


   initial begin


      clk = 0 ;


      repeat(64) begin

         #10ns clk = ~clk;

      end


   end



endmodule
```