Ain Shams University,
Faculty of Engineering,
Computer and Systems Engineering Department.

Verilog Implementation of MIPS Processor

SUBMITTED BY

Group (28)	
Hossam Mohamed El-Nagy Mohamed	33737
Asim Mahmoud Abd El-Azem	33767
Abd-Elrahman Abd-Elrazek Hussein	33768
Abdullah Abd-Elrazek Hussein	33773
Essam El-Din Eid Abd-Elsalam	33775

SUBMITTED TO

Dr. Cherif Salama

Eng. Diaa El-Din Mohamed

DATE SUBMITTED

26-dec-2016

Table of Contents

1.0 Implementation Description

Single cycle MIPS processor which has the following memories:

- Instruction Memory: Its size is 256 bytes which means it can only hold 64 instructions.
- Register File: It has 32 registers, each register is 32 bit wide.
- Data Memory: Its size is 2048 bytes which means it can hold 512 word.

And supporting the following instructions:

- Arithmetic: add, addi, sub
- Load/Store: lw, sw
- Logic: sll, and, andi, nor
- Control flow: beq, jal, jr, j
- Comparison: slt

We based our implementation on references [1] and [2]. (Refer to page 26 for references)

Bonus:

- Assembler implemented by C++.
- In program #3 (page 18), instead of just displaying the final state of datapath after the simulation ends, we show the state of the datapath each cycle during the simulation.

Tools used: Aldec Active-HDL 9.3, Microsoft Visual Studio 2015

Languages: Verilog , C++

2.0 Datapath Description

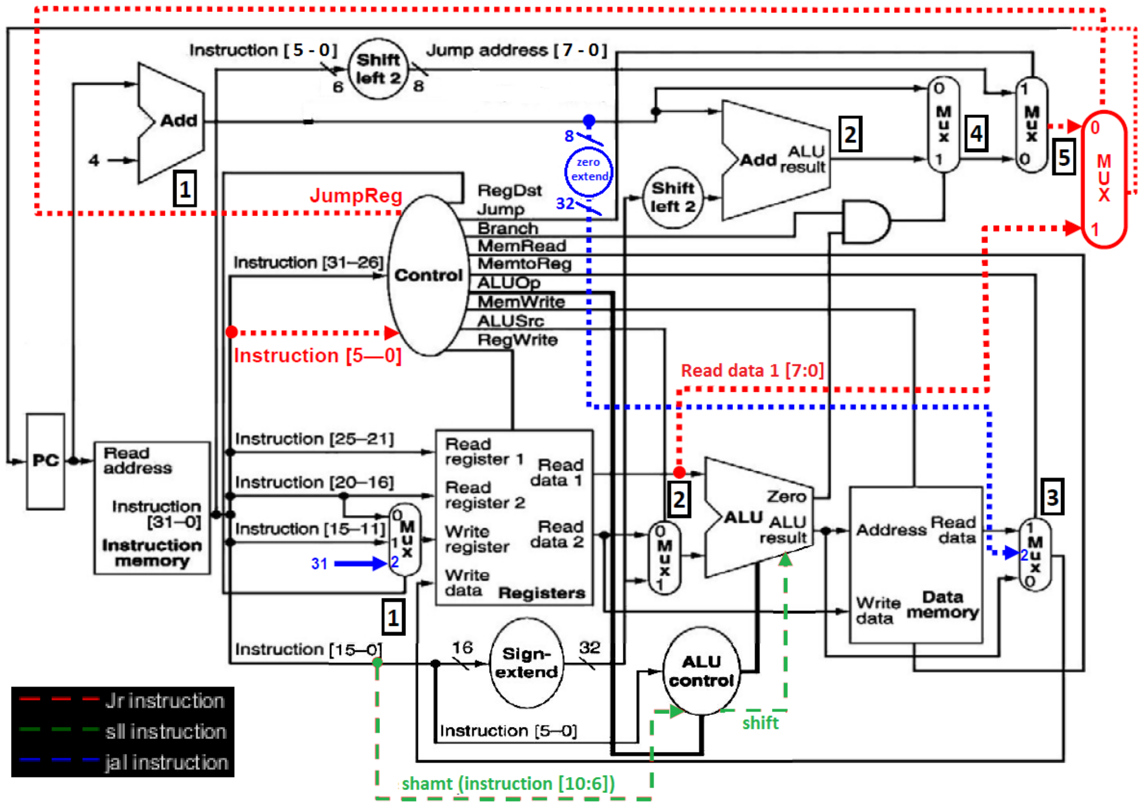


Figure 1

Figure 1 shows the datapath we implemented including the necessary extensions to support jr, jal and sll instructions.

2.1 Adding the jr instruction (red lines)

- Input the funct field (Instruction [5:0]) to the main control unit.
- Allow the new PC to come from a register (Read data 1 port).
- We only take 8 bits from the register because the instruction memory is only 256 byte in depth.
- Add a new control signal (JumpReg) to control it through a multiplexor.
- Writing to the register file will be disabled.

As jr is R-format instruction, it is easier to input the funct field (Instruction [5:0]) to the main control unit to use to generate a control signal JumpReg. The other option is to generate this signal from the ALU control unit. But, it is preferable to go through the first option because JumpReg has nothing to do with the ALU or the ALU control.

2.2 Adding the jal instruction (blue lines)

- Because the instruction memory is only 256 byte in depth we only take 6 bits of the immediate field (instruction[5:0]) and shift left it by 2 to make it 8 bits. Those 8 bits is the jump address.
- Expand the multiplexor controlled by RegDst to include the value 31 as a new second input.
- Expand the multiplexor controlled by MemtoReg to have PC+4 as new second input.
- This requires changing the control lines of these two multiplexors from a single bit to two bits.
- The Jump control signal needs to be set to 1 to operate as the j instruction.
- RegWrite should be set to 1 so the register file will be enabled to write.
- Zero extend the PC + 4 which is the 8-bit address of the next instruction to make it 32 bits so we can write it in a register.

2.3 Adding the sll instruction (green lines)

- Feed the instruction 5-bit shamt field (instruction[10:6]) to the ALU control in order to use it to determine the shift amount.
- Feed the 5-bit shift output from the ALU control to the ALU in order to use it as the shift amount to apply to its second input.
- For the sll instruction, the ALU control output “shift” is assigned the value of the instruction “shamt” field.
- The sll instruction control settings are similar to that of any other ALU (RFormat) instruction but rs field (instruction[25:21]) is always 00000.

2.4 Clock cycle of MIPS processor

- The period of the clock generated inside the MIPS_processor module is 50ns which is greater than delay of the critical path (26ns for lw instruction).
- In a single cycle datapath everything must complete within one clock cycle, before the next positive clock edge.
- Several things happen on the next positive clock edge:
    - The register file is updated for arithmetic or lw instructions.
    - Data memory is updated for sw instructions.
    - The PC is updated to point to the next address.
- That’s why for any program, $no.\,clock\ cycles\ needed = no.\,instruction + 1$

3.0 How Work Was Split Among Team Members

We first split the modules as follows:

- Abd-Elrahman Abd-Elrazek & Abdullah Abd-Elrazek: Implementation of the Instruction Memory, Register File and Date Memory.
- Essam El-Din Eid: Implementation of the main Control Unit.
- Asim Mahmoud: Implementation of ALU & ALU Control Unit.
- Hossam Mohamed: Implementation of PC Counter, sign extend, shift left and all Mux's modules.

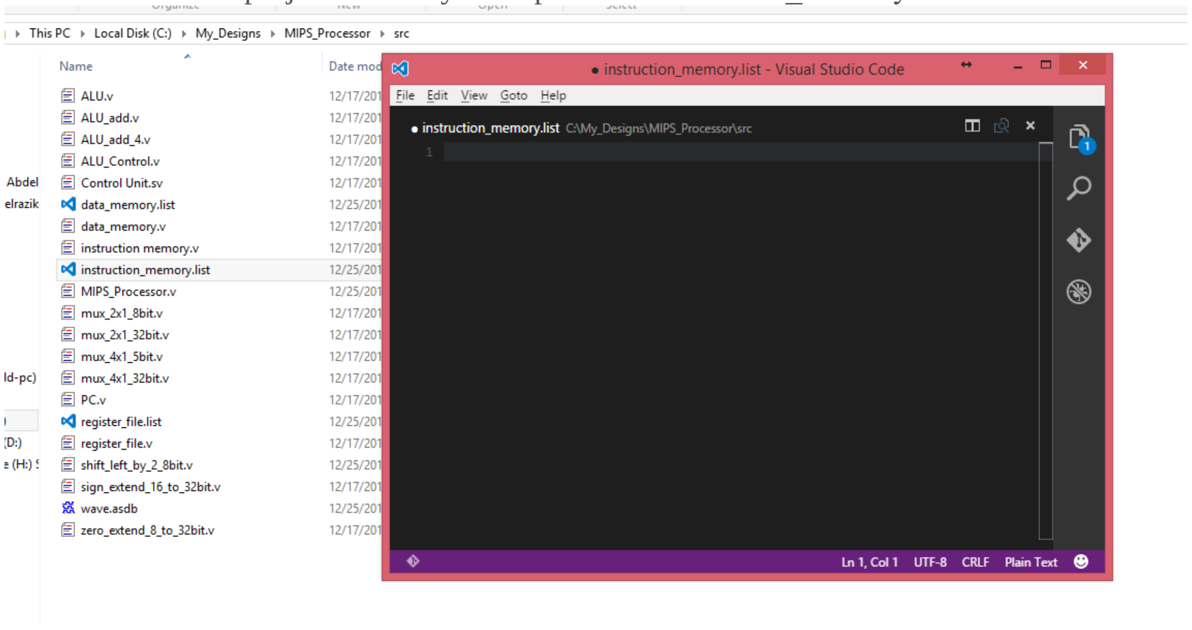
Then we split the team into two groups:

- Abd-Elrahman Abd-Elrazek, Abdullah Abd-Elrazek & Asim Mahmoud: Implementation of the top level structural module and testing.
- Hossam Mohamed & Essam El-Din Eid: Implementation of assembler.

4.0 User Guide with Snapshots

4.1 MIPS Guide

1. Go to the project directory and open "src\instruction_memory.list".

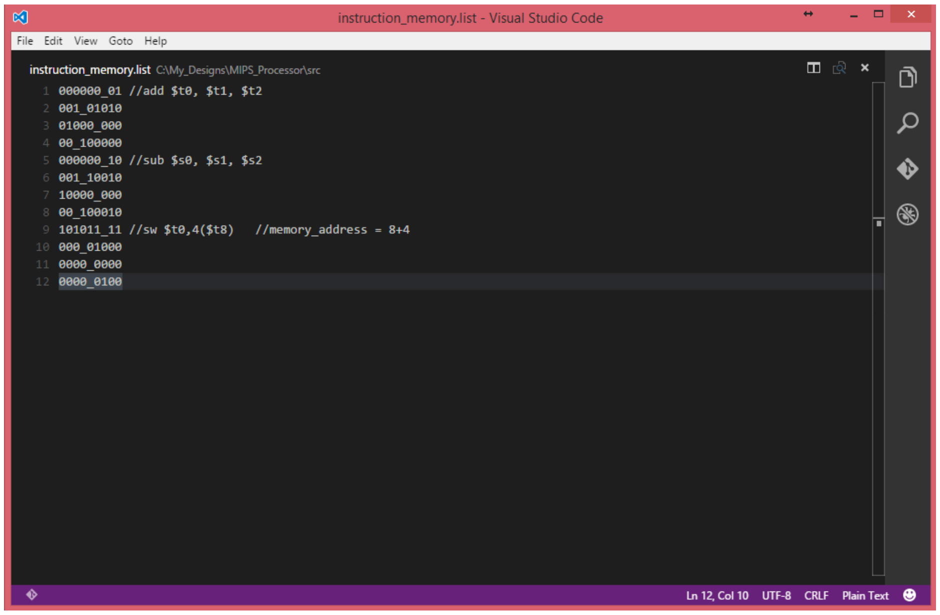

This PC ▸ Local Disk (C:) ▸ My_Designs ▸ MIPS_Processor ▸ src

Name	Date mod
ALU.v	12/17/201
ALU_add.v	12/17/201
ALU_add_4.v	12/17/201
ALU_Control.v	12/17/201
Control Unit.sv	12/17/201
data_memory.list	12/25/201
data_memory.v	12/17/201
instruction memory.v	12/17/201
instruction_memory.list	12/25/201
MIPS_Processor.v	12/25/201
mux_2x1_8bit.v	12/17/201
mux_2x1_32bit.v	12/17/201
mux_4x1_5bit.v	12/17/201
mux_4x1_32bit.v	12/17/201
PC.v	12/17/201
register_file.list	12/25/201
register_file.v	12/17/201
shift_left_by_2_8bit.v	12/25/201
sign_extend_16_to_32bit.v	12/17/201
wave.asdb	12/25/201
zero_extend_8_to_32bit.v	12/17/201

● instruction_memory.list - Visual Studio Code

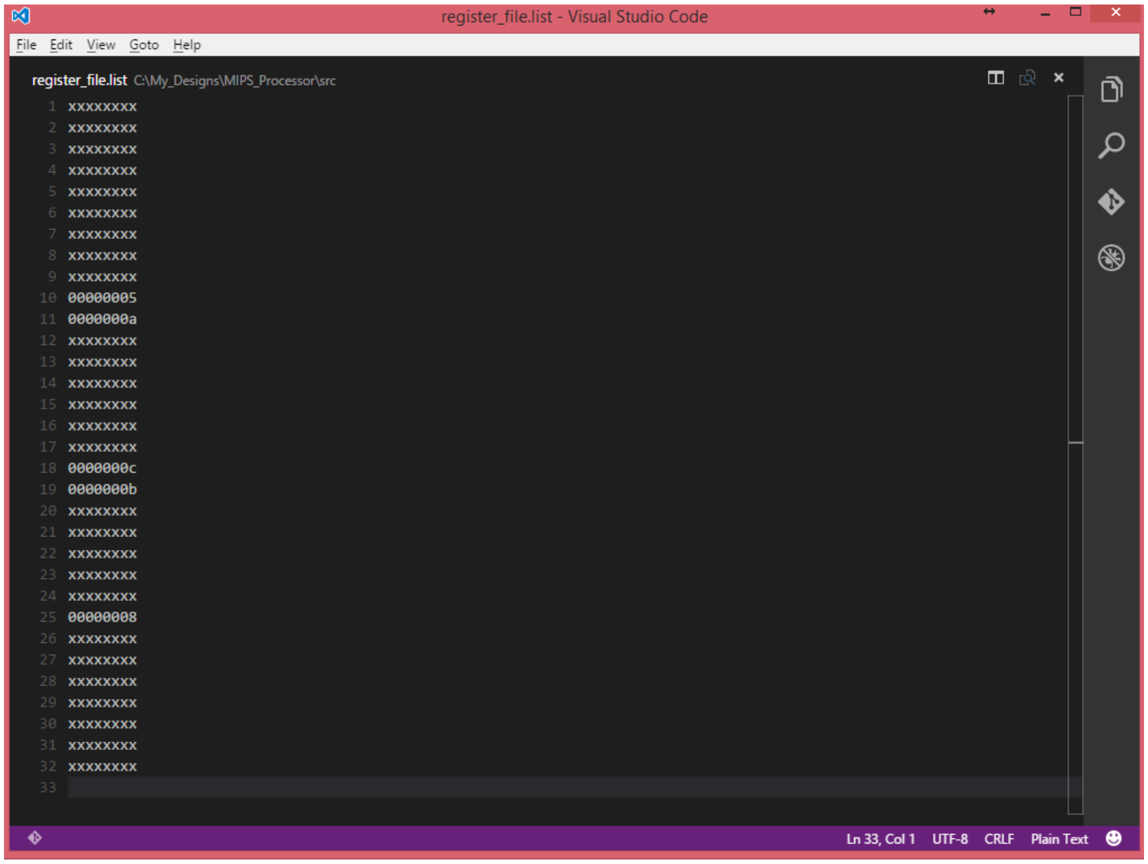
● instruction_memory.list C:\My_Designs\MIPS_Processor\src

2. Write the instructions to be executed by the mips processor in binary with each line being 8 bits (each instruction will take 4 lines).Optionally you can use the assembler to generate the memory for you (refer to page 9 for Assembler Guide).


instruction_memory.list - Visual Studio Code
File Edit View Goto Help
instruction_memory.list C:\My_Designs\MIPS_Processor\src
Ln 12, Col 10 UTF-8 CRLF Plain Text

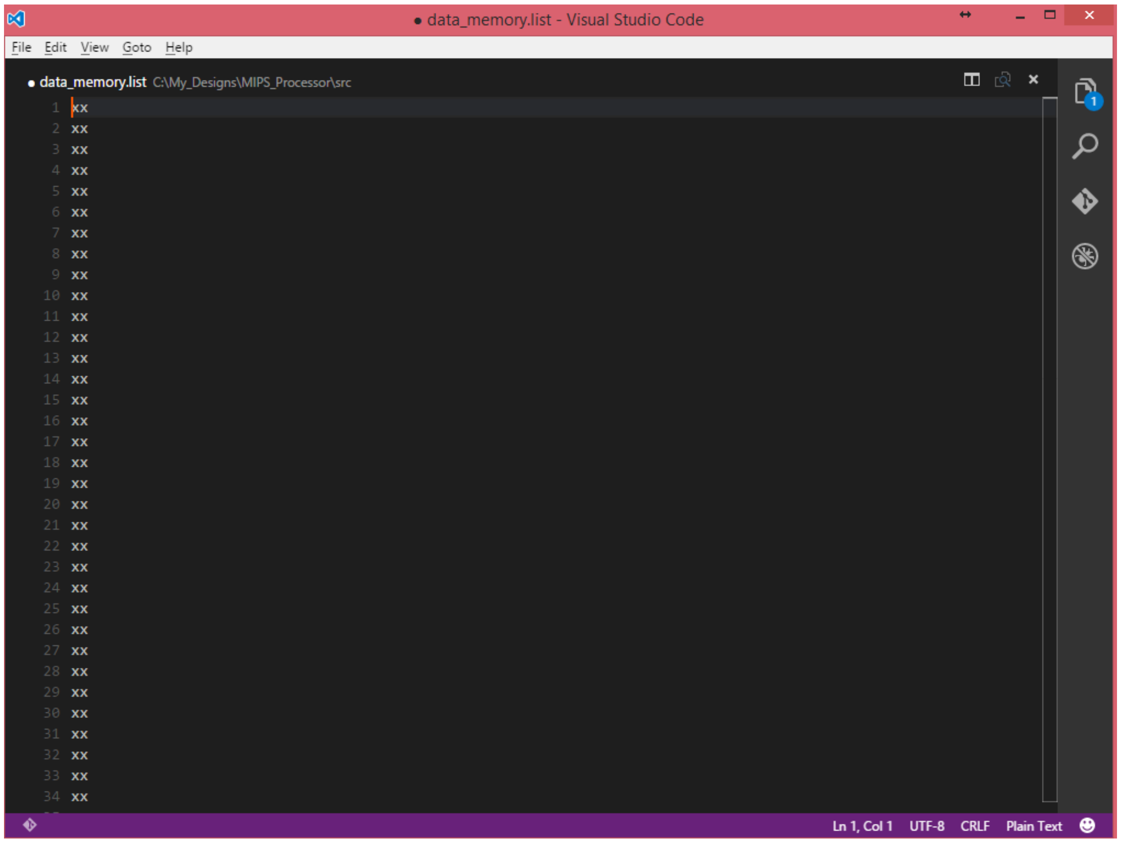
```
000000_01 //add $t0, $t1, $t2
001_01010
01000_000
00_100000
000000_10 //sub $s0, $s1, $s2
001_10010
10000_000
00_100010
101011_11 //sw $t0,4($t8)   //memory_address = 8+4
000_01000
0000_0000
0000_0100
```

3. Open “src\register_file.list” and write the associated data of the program.


register_file.list - Visual Studio Code
File Edit View Goto Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 33, Col 1 UTF-8 CRLF Plain Text

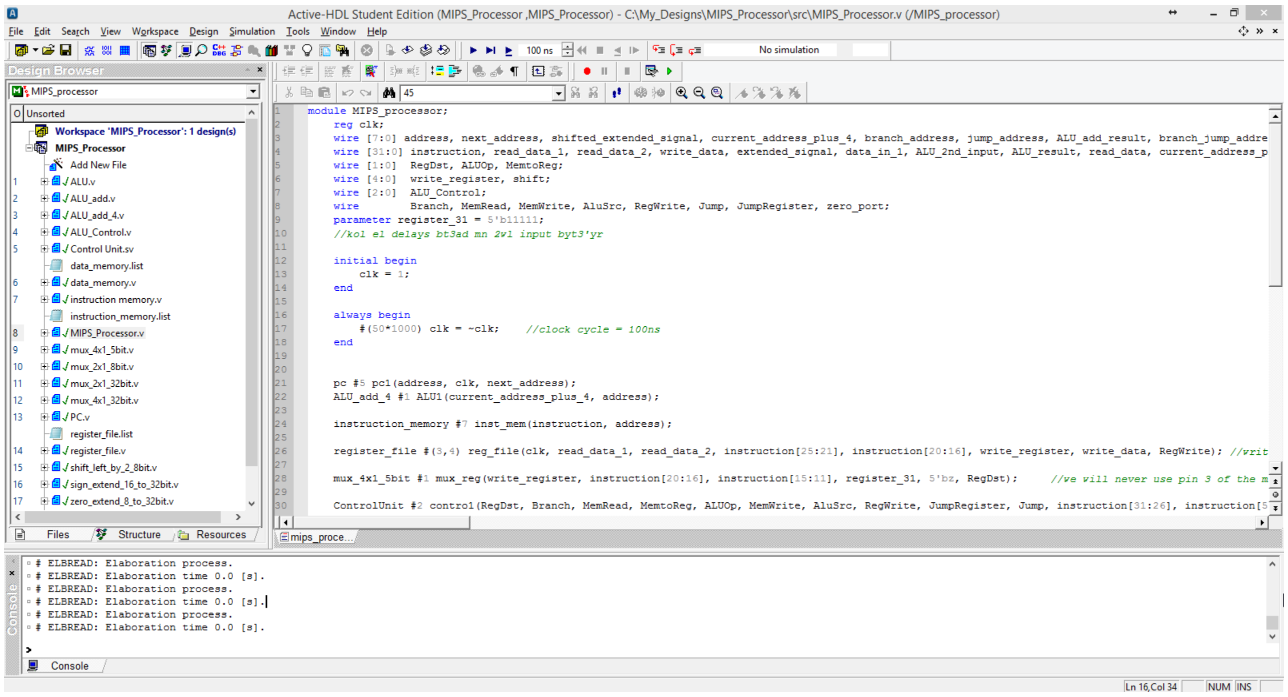
```
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
00000005
0000000a
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000c
0000000b
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
00000008
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx

```

4. Open “src\data_memory.list” and write the associated data of the program.


● data_memory.list - Visual Studio Code
File Edit View Goto Help
● data_memory.list C:\My_Designs\MIPS_Processor\src
Ln 1, Col 1 UTF-8 CRLF Plain Text

```
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
```

5. Open the project in your Verilog simulator.



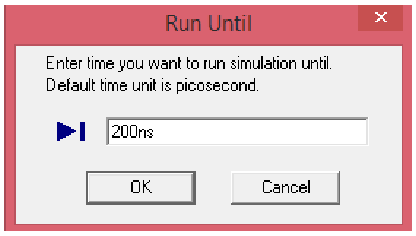
```
module MIPS_processor;
    reg clk;
    wire [7:0] address, next_address, shifted_extended_signal, current_address_plus_4, branch_address, jump_address, ALU_add_result, branch_jump_addre
    wire [31:0] instruction, read_data_1, read_data_2, write_data, extended_signal, data_in_1, ALU_2nd_input, ALU_result, read_data, current_address_p
    wire [1:0]  RegDst, ALUOp, MemtoReg;
    wire [4:0]  write_register, shift;
    wire [2:0]  ALU_Control;
    wire        Branch, MemRead, MemWrite, AluSrc, RegWrite, Jump, JumpRegister, zero_port;
    parameter register_31 = 5'b11111;
    //kol el delays bt3ad mn 2vl input byt3'yr

    initial begin
        clk = 1;
    end

    always begin
        #(50*1000) clk = ~clk;    //clock cycle = 100ns
    end


    pc #5 pc1(address, clk, next_address);
    ALU_add_4 #1 ALU1(current_address_plus_4, address);

    instruction_memory #7 inst_mem(instruction, address);

    register_file #(3,4) reg_file(clk, read_data_1, read_data_2, instruction[25:21], instruction[20:16], write_register, write_data, RegWrite); //writ

    mux_4x1_5bit #1 mux_reg(write_register, instruction[20:16], instruction[15:11], register_31, 5'bz, RegDst);     //we will never use pin 3 of the m

    ControlUnit #2 control(RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, AluSrc, RegWrite, JumpRegister, Jump, instruction[31:26], instruction[5
```

```
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.0 [s].
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.0 [s].
# ELBREAD: Elaboration process.
# ELBREAD: Elaboration time 0.0 [s].
>
```

6. Using simulation feature in your Verilog simulator, simulate MIPS_processor module for a duration according to the following formula

$$duration(ns) = number\ of\ instructions * 50 + 50$$

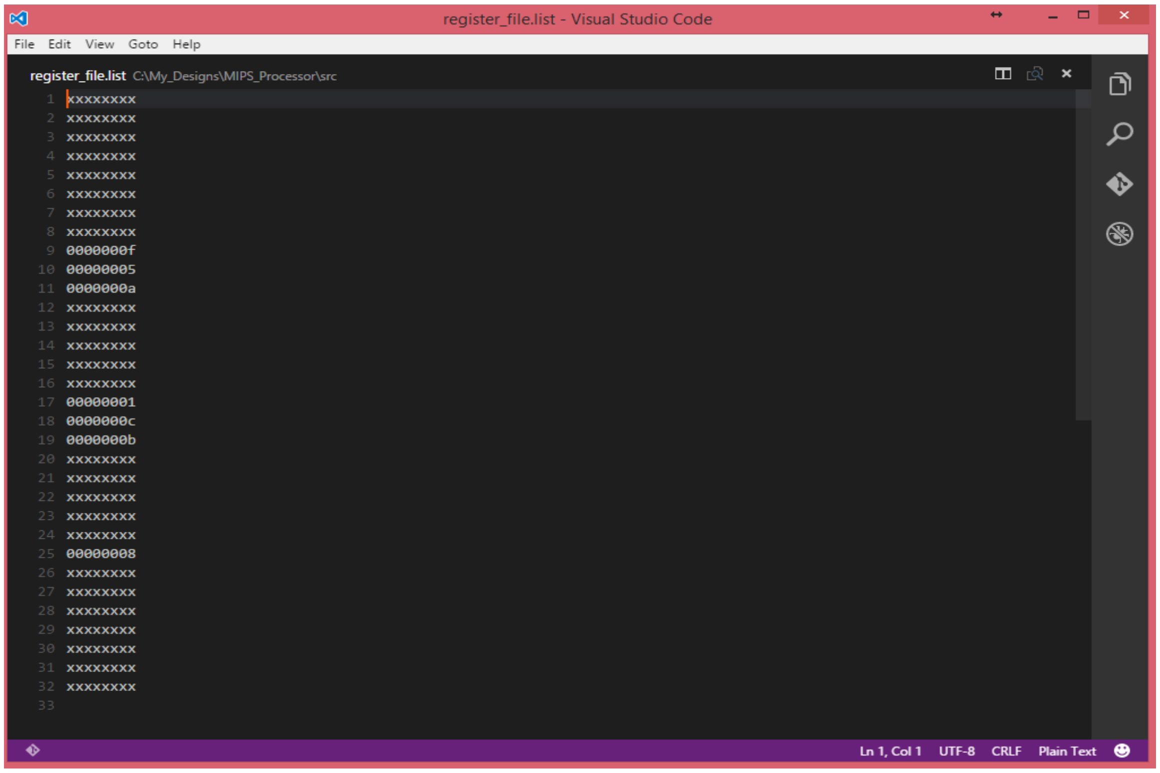
For example: here we have three instructions so we should run the simulation for 200ns.
In ActiveHDL from simulation menu choose "run until" and enter "200ns" then press ok.



Enter time you want to run simulation until.
Default time unit is picosecond.

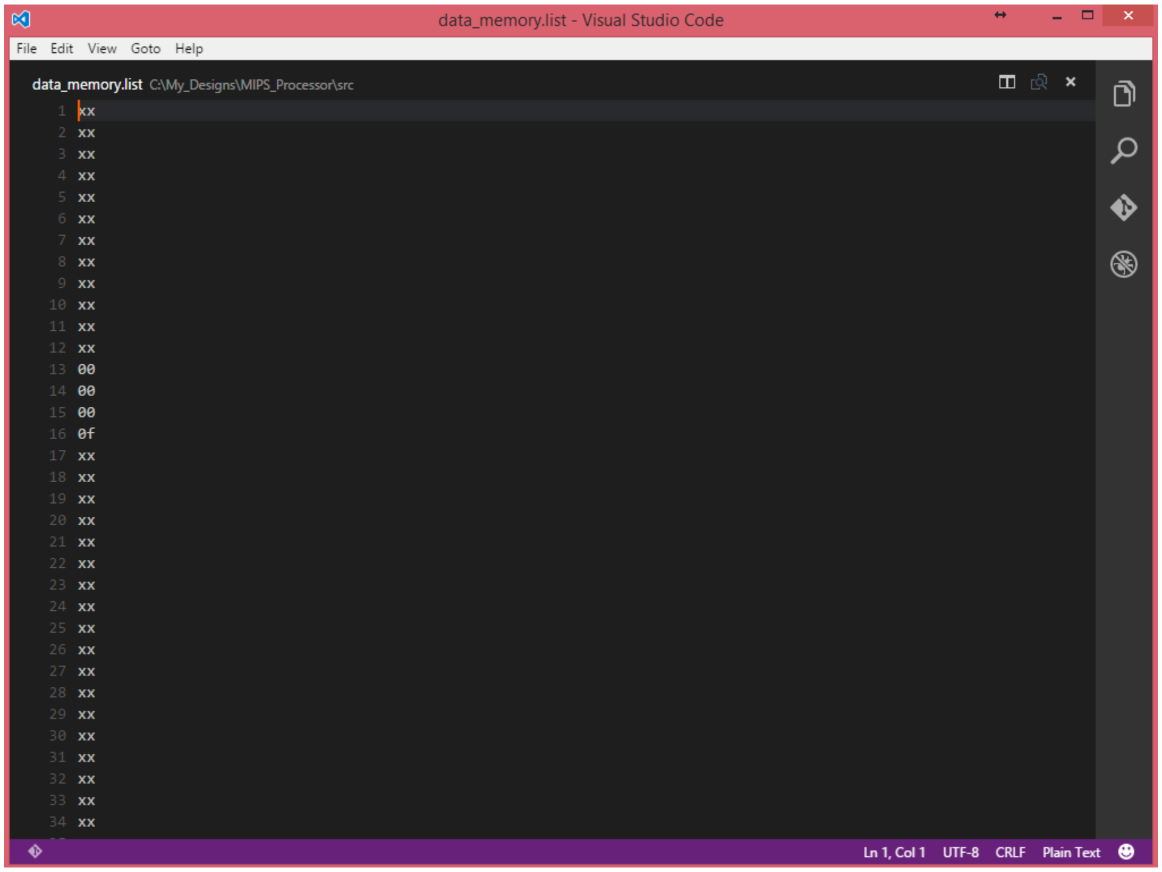
200ns

7. You can view the output in the "register_file.list".
In this example we find that:
   - It added $t1(00000005), $t2(0000000a) and stored the result (0000000f) in $t0.
   - It subtracted $s2(0000000b) from $s1(0000000c) and stored the result in $s0(00000001).


register_file.list - Visual Studio Code
File
Edit
View
Goto
Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 1, Col 1
UTF-8
CRLF
Plain Text

```
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000f
00000005
0000000a
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
00000001
0000000c
0000000b
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
00000008
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx

```

8. You can view the output in the “data_memory.list”.
In this example we find that it stored $t0 (0000000f) in the data memory in address $t8 (00000008) + 4 = 12


data_memory.list - Visual Studio Code
File Edit View Goto Help
data_memory.list C:\My_Designs\MIPS_Processor\src
Ln 1, Col 1 UTF-8 CRLF Plain Text

```
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
00
00
00
0f
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
xx
```

NOTE THAT:

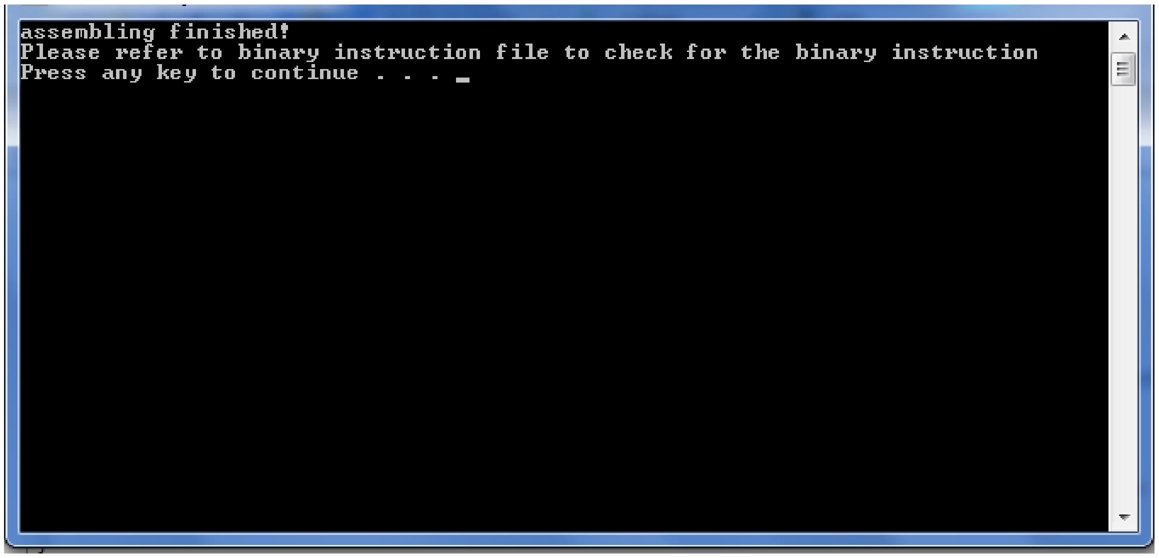
- All the simulations in this report are done using Active-HDL and the time unit in Active-HDL is 1 ps.
- If you want to use other simulation programs which have different time unit, you have to modify the simulation duration of the program.
  For example ModelSim’s time unit is 1 ns so the new simulation duration becomes:

$$duration(ns) = (number\ of\ instructions * 50 + 50) * 1000$$

4.2 Assembler Guide

To use the Assembler without producing any errors or failures in conversion from Assembly Instruction to Binary Instructions please follow these steps.

1. open the “assembly.txt” file and type your assembly code as follows:
    - Instruction is to be written as “operation” then space then the registers to be affected with, separated by commas, if spaces are typed inside between the register name and comma the assembler will automatically delete these spaces.
      For example, an instruction (`nor $s3, $s1 , $s0`) will be converted inside the assembler into (`nor $s3,$s1,$s0`) and the assembler will parse the string and register names correctly.
    - Labels are to take a separate line and ended with a colon (“:”) for example (“Label:”).
    - Don’t use “.data” or “.text” because all the supported instructions doesn’t use “.data” section. So write your program directly without actually typing ”.text”.
2. execute the executable file “Assembler 2” and wait for it to finish, you should see on the console “assembling finished please refer to binary instruction file to check for the binary instruction conversion “ , it will look like this:



```
assembling finished!
Please refer to binary instruction file to check for the binary instruction
Press any key to continue . . . _
```

3. You can now find the instruction(s) in binary inside the file instruction_memory.list.

NOTE THAT:

- both these files ("assembly.txt" and "instruction_memory.list") **MUST** exist beside the executable file for it to read from and write to both of them successfully.
- You also don't need to open the instruction_memory.list file every time you want to write a new program and remove the binary inside of it, the assembler will do that for you and clear all the old binary instruction code once the executable is started.
- Supported instructions: `add, addi, lw, sw, sll, and, andi, nor, beq, jal, jr, slt, sub.`

5.0 Tested Programs

5.1 Program #1

- **Program description:**

This program will store the numbers from 0 to 10 in the data memory beginning from address 32.

- **Assembly code:**

```
addi $t2, $zero, 1
add $t0, $zero, $zero
addi $t3, $zero, 11
addi $s1,$zero,32
loop:
sw $t0,0($s1)
addi $s1, $s1, 4
addi $t0, $t0, 1
slt $t1, $t0, $t3
beq $t1, $t2, loop
```

- **Binary code:**

```
0010_0000_0000_1010_0000_0000_0000_0001 //addi $t2, $zero, 1
0000_0000_0000_0000_0100_0000_0010_0000 //add $t0, $zero, $zero
0010_0000_0000_1011_0000_0000_0000_1011 //addi $t3, $zero, 11
0010_0000_0001_0001_0000_0000_0010_0000 //addi $s1,$zero,32
1010_1110_0010_1000_0000_0000_0000_0000 //loop:sw $t0,0($s1)
0010_0010_0011_0001_0000_0000_0000_0100 //addi $s1, $s1, 4
0010_0001_0000_1000_0000_0000_0000_0001 //addi $t0, $t0, 1
0000_0001_0000_1011_0100_1000_0010_1010 //slt $t1, $t0, $t3
0001_0001_0010_1010_1111_1111_1111_1011 //beq $t1, $t2, loop
```

- **Number of clock cycles needed to simulate the program to completion:**

As the loop has 5 instructions and the number of iterations is 11

$$Number\ of\ instructions = 4 + 5 * 11 = 59$$

$$Number\ of\ clock\ cycles = 59 + 1 = 60$$

$$\therefore simulation\ duration = 60 * 50 = 3000ns$$

- **Expected output:**

  - We use $t0 as counter for the loop so it starts from 0 and reaches 11 when the loop exits.
  - We use $s1 to store the data memory address which we will write in and increment it by 4 each iteration so we expect it to be the last address we write in + 4 = 76.
  - $t2 and $t3 will still the same throughout the program execution.
  - $t1 will be set to zero before the loop exits.

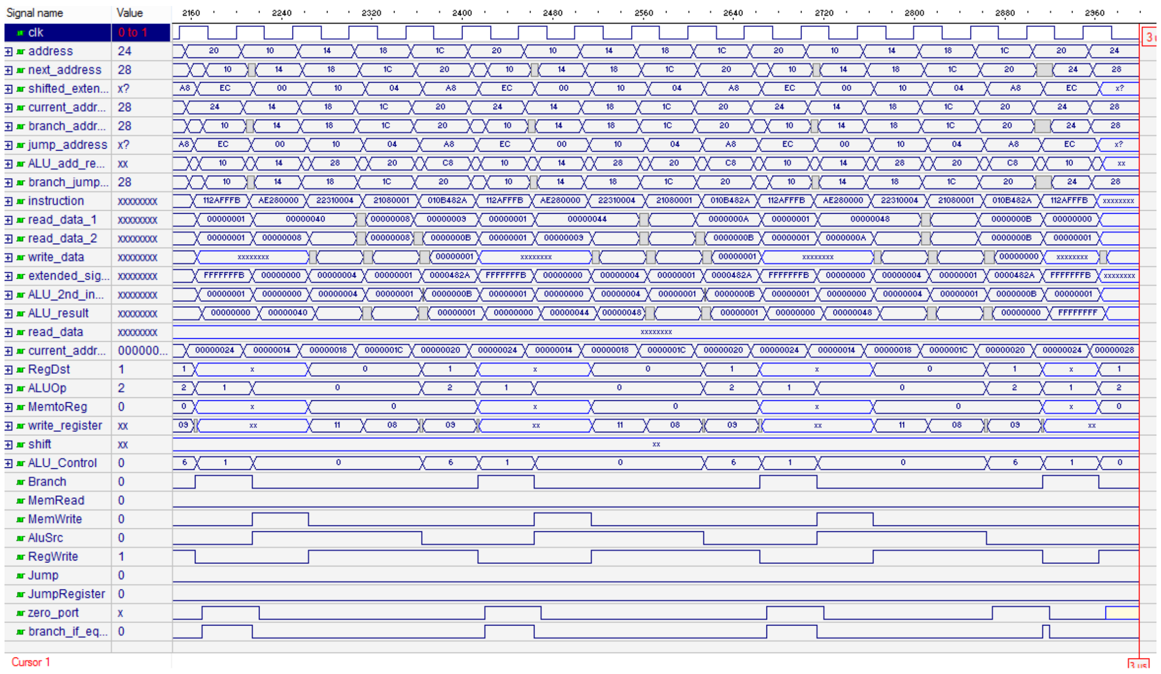
In "register_file.list" we expect:

1. $\$t2 = 1$
2. $\$t3 = (11)_{10} = (B)_{16}$
3. $\$t0 = (11)_{10} = (B)_{16}$
4. $\$s1 = (76)_{10} = (4C)_{16}$
5. $\$t1 = 0$

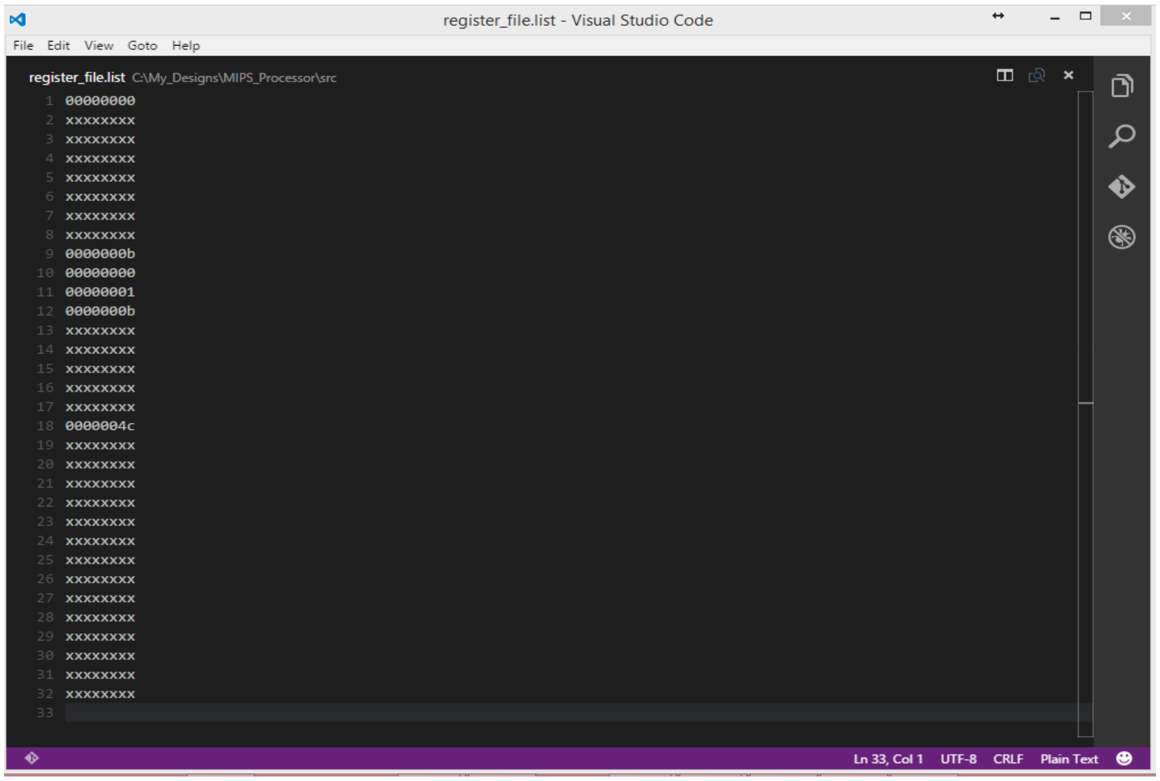
In "data_memory.list" we expect: The numbers from 0 to 10 to be stored in the memory starting from address 32 to 72.

- **Actual Outputs:**

- Waveform


Signal name
Value
clk
address
next_address
shifted_exten...
current_addr...
branch_addr...
jump_address
ALU_add_re...
branch_jump...
instruction
read_data_1
read_data_2
write_data
extended_sig...
ALU_2nd_in...
ALU_result
read_data
current_addr...
RegDst
ALUOp
MemtoReg
write_register
shift
ALU_Control
Branch
MemRead
MemWrite
AluSrc
RegWrite
Jump
JumpRegister
zero_port
branch_if_eq...
Cursor 1

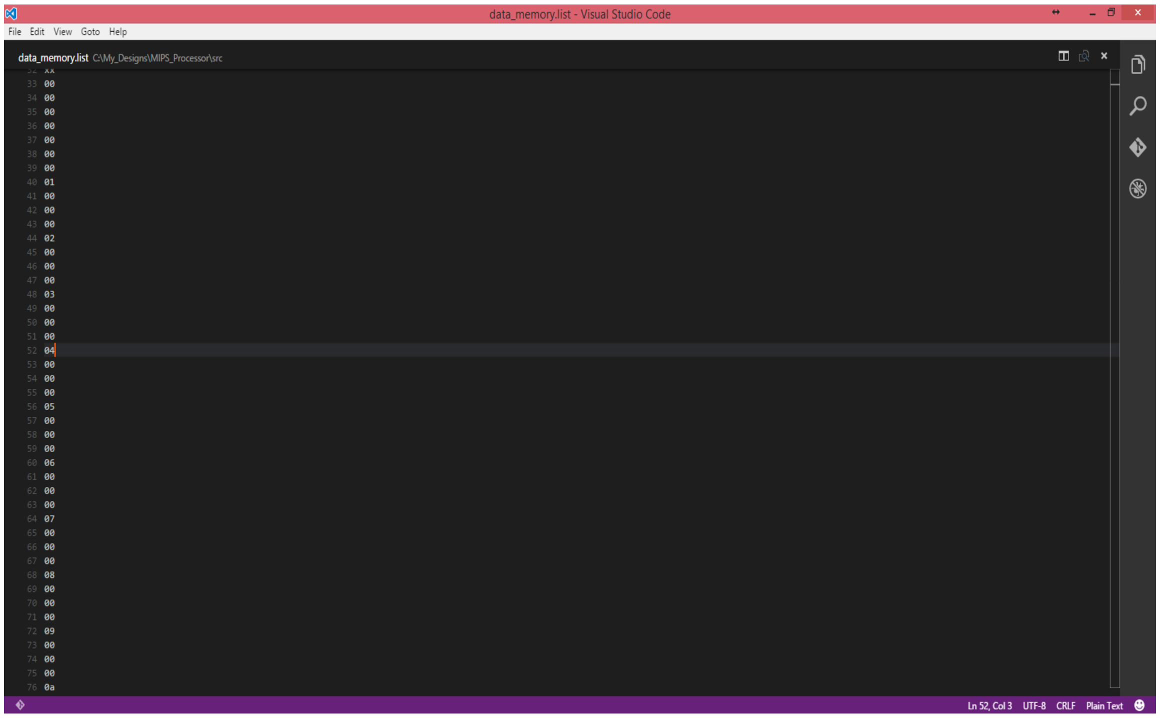
- register_file.list



register_file.list - Visual Studio Code
File Edit View Goto Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 33, Col 1 UTF-8 CRLF Plain Text

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000b
00000000
00000001
0000000b
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000004c
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx

```

- data_memory.lis



data_memory.list - Visual Studio Code
File Edit View Goto Help
data_memory.list C:\My_Designs\MIPS_Processor\src
Ln 52, Col 3 UTF-8 CRLF Plain Text

```
xx
00
00
00
00
00
00
00
01
00
00
00
02
00
00
00
03
00
00
00
04
00
00
00
05
00
00
00
06
00
00
00
07
00
00
00
08
00
00
00
09
00
00
00
0a
```

5.2 Program #2

- **Program description:**

This program will store the first 10 numbers of the Fibonacci series beginning from address 100.

- **Assembly code:**

```
addi $s0,$zero,0
addi $s1,$zero,1
sw $s0,100($zero)
sw $s1,104($zero)
addi $t0, $zero,2
addi $t1,$zero,108
addi $t2,$zero,1
addi $t4,$zero,10
loop:
jal getNextFeb
add $s0,$s1,$zero
add $s1,$s2,$zero
addi $t0,$t0,1
slt $t3,$t0,$t4
beq $t3,$t2,loop
j exit
getNextFeb:
add $s2,$s0,$s1
sw $s2,0($t1)
addi $t1,$t1,4
jr $ra
exit:
```

- **Binary code:**

```
0010_0000_0001_0000_0000_0000_0000_0000 //addi $s0,$zero,0
0010_0000_0001_0001_0000_0000_0000_0001 //addi $s1,$zero,1
1010_1100_0001_0000_0000_0000_0000_0000 //sw $s0,100($zero)
1010_1100_0001_0001_0000_0000_0000_0100 //sw $s1,104($zero)
0010_0000_0000_1000_0000_0000_0000_0010 //addi $t0, $zero,2
0010_0000_0000_1001_0000_0000_0000_1000 //addi $t1,$zero,108
0010_0000_0000_1010_0000_0000_0000_0001 //addi $t2,$zero,1
0010_0000_0000_1100_0000_0000_0000_1010 //addi $t4,$zero,10
0000_1100_0000_0000_0000_0000_0000_1111 //loop: jal getNextFeb
0000_0010_0010_0000_1000_0000_0010_0000 //add $s0,$s1,$zero
0000_0010_0100_0000_1000_1000_0010_0000 //add $s1,$s2,$zero
0010_0001_0000_1000_0000_0000_0000_0001 //addi $t0,$t0,1
0000_0001_0000_1100_0101_1000_0010_1010 //slt $t3,$t0,$t4
0001_0001_0110_1010_1111_1111_1111_1010 //beq $t3,$t2,loop
0000_1000_0000_0000_0000_0000_0001_0011 //j exit
0000_0010_0001_0001_1001_0000_0010_0000 //getNextFeb: add $s2,$s0,$s1
1010_1101_0011_0010_0000_0000_0000_0000 //sw $s2,0($t1)
0010_0001_0010_1001_0000_0000_0000_0100 //addi $t1,$t1,4
0000_0011_1110_0000_0000_0000_0000_1000 //jr $ra
```

- **Number of clock cycles needed to simulate the program to completion:**

As the loop has 10 instructions including the ones inside `getNextFeb` and the number of iterations is 8

$$Number\ of\ instructions = 9 + 8 * 10 = 89$$

$$Number\ of\ clock\ cycles = 89 + 1 = 90$$

$$simulation\ duration = 90 * 50 = 4500ns$$

- **Expected output:**

  - We use $t0 as counter for the loop but it starts from 2 and reaches 10 when the loop exits.
  - We use $t1 to store the data memory address which we will write in and increment it by 4 each iteration so we expect it to be the last address we write in + 4 = 140.
  - $t2 and $t4 will still the same throughout the program execution.
  - $t3 will be set to zero before the loop exits.

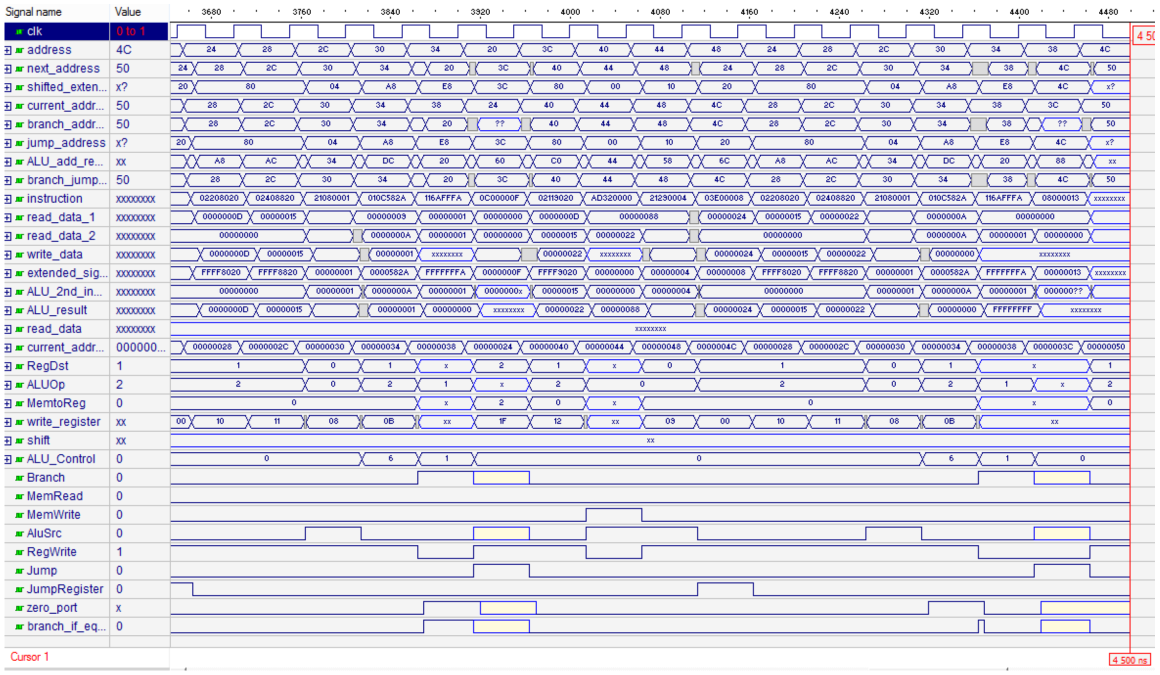
- $s0 will be the 9th Fibonacci number (21).
- $s1 and $s2 will be the 10th Fibonacci number (34).
- $ra will store the return address which is the address of the instruction following the jal instruction.

In “register_file.list” we expect:

1. $\$t0 = (10)_{10} = (A)_{16}$
2. $\$t1 = (140)_{10} = (8C)_{16}$
3. $\$t2 = 1$
4. $\$t4 = (10)_{10} = (A)_{16}$
5. $\$t3 = 0$
6. $\$s0 = (21)_{10} = (15)_{16}$
7. $\$s1 = \$s0 = (34)_{10} = (22)_{16}$

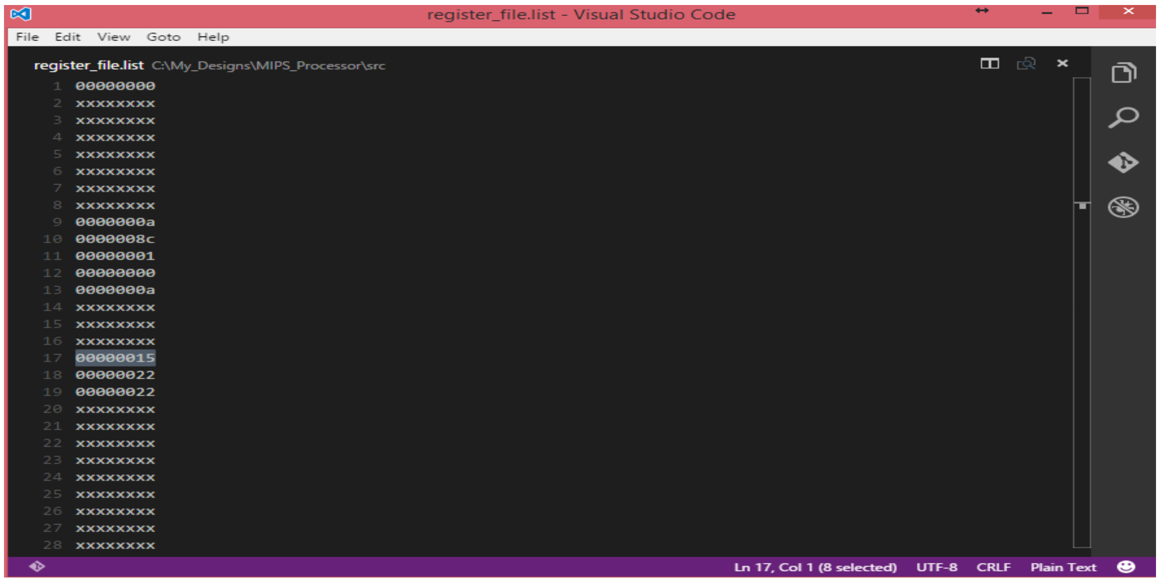
In “data_memory.list” we expect: The numbers $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)$ to be stored in the memory starting from address 100 to 136.

- **Actual Outputs:**
  - Waveform:



Signal name	Value
clk	0 to 1
address	4C
next_address	50
shifted_exten...	x?
current_addr...	50
branch_addr...	50
jump_address	x?
ALU_add_re...	xx
branch_jump...	50
instruction	xxxxxxxx
read_data_1	xxxxxxxx
read_data_2	xxxxxxxx
write_data	xxxxxxxx
extended_sig...	xxxxxxxx
ALU_2nd_in...	xxxxxxxx
ALU_result	xxxxxxxx
read_data	xxxxxxxx
current_addr...	000000...
RegDst	1
ALUOp	2
MemtoReg	0
write_register	xx
shift	xx
ALU_Control	0
Branch	0
MemRead	0
MemWrite	0
AluSrc	0
RegWrite	1
Jump	0
JumpRegister	0
zero_port	x
branch_if_eq...	0

- Register_file.list

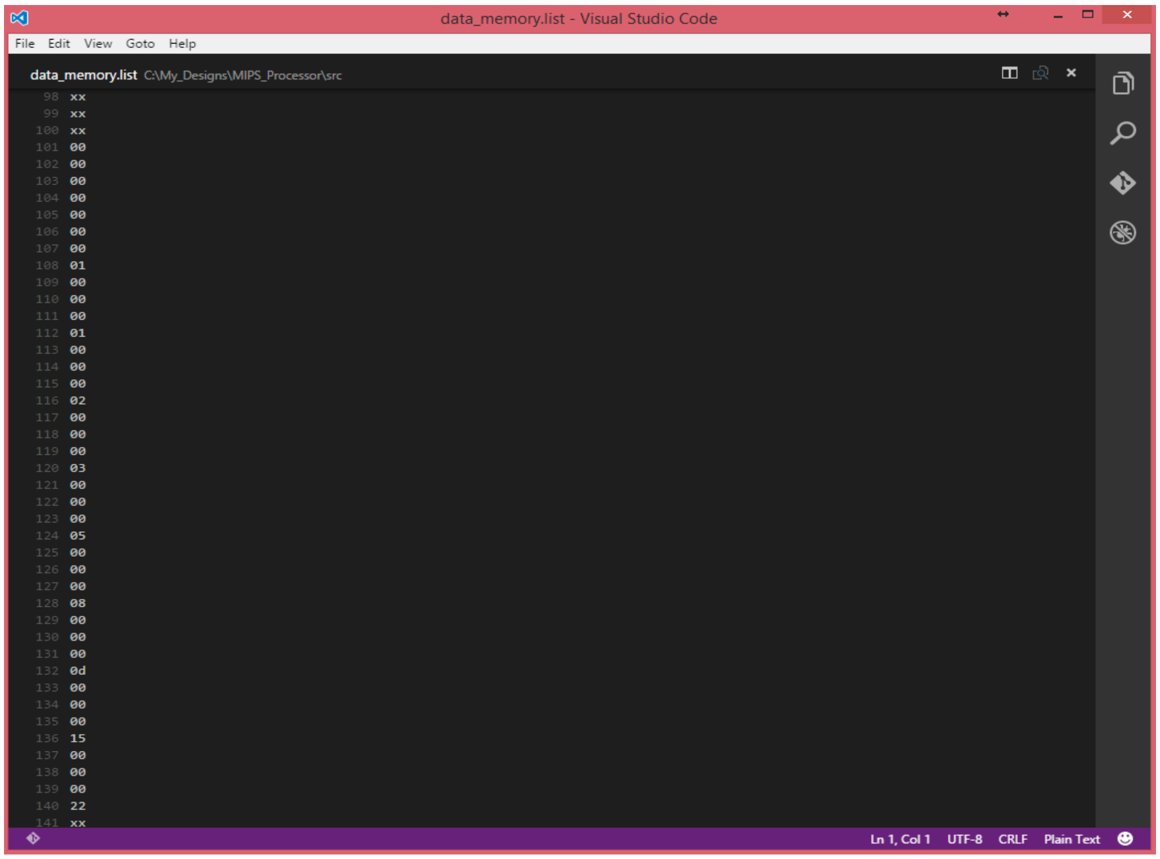

register_file.list - Visual Studio Code

register_file.list C:\My_Designs\MIPS_Processor\src

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000a
0000008c
00000001
00000000
0000000a
xxxxxxxx
xxxxxxxx
xxxxxxxx
00000015
00000022
00000022
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Ln 17, Col 1 (8 selected) UTF-8 CRLF Plain Text

- data_memory.list


data_memory.list - Visual Studio Code

data_memory.list C:\My_Designs\MIPS_Processor\src

```
xx
xx
xx
00
00
00
00
00
00
00
01
00
00
00
01
00
00
00
02
00
00
00
03
00
00
00
05
00
00
00
08
00
00
00
0d
00
00
00
15
00
00
00
22
xx
```

Ln 1, Col 1 UTF-8 CRLF Plain Text

5.3 Program #3

- **Program description:**

This program just do some arithmetic and logical operations in addition to the load word instruction.

We manually entered the number $(1A)_{16}$ in “data_memory.list” at address 64 and in “register_file.list” we added:

- $\$t0 = (0A)_{16}$
- $\$t1 = (05)_{16}$
- $\$s1 = (DB)_{16}$

- **Assembly code:**

```
lw $s0 , 64($zero)
sub $t2, $t1, $t0
and $s2, $s1, $s0
nor $s3, $s1, $s0
andi $s4, $s1, 15
sll $s5, $s1, 4
```

- **Binary code:**

```
1000_1100_0001_0000_0000_0000_0100_0000 //lw $s0 , 64($zero)
0000_0001_0010_1000_0101_0000_0010_0010 //sub $t2, $t1, $t0
0000_0010_0011_0000_1001_0000_0010_0100 //and $s2, $s1, $s0
0000_0010_0011_0000_1001_1000_0010_0111 //nor $s3, $s1, $s0
0011_0010_0011_0100_0000_0000_0000_1111 //andi $s4, $s1, 15
0000_0000_0001_0001_1010_1001_0000_0000 //sll $s5, $s1, 4
```

- **Number of clock cycles needed to simulate the program to completion:**

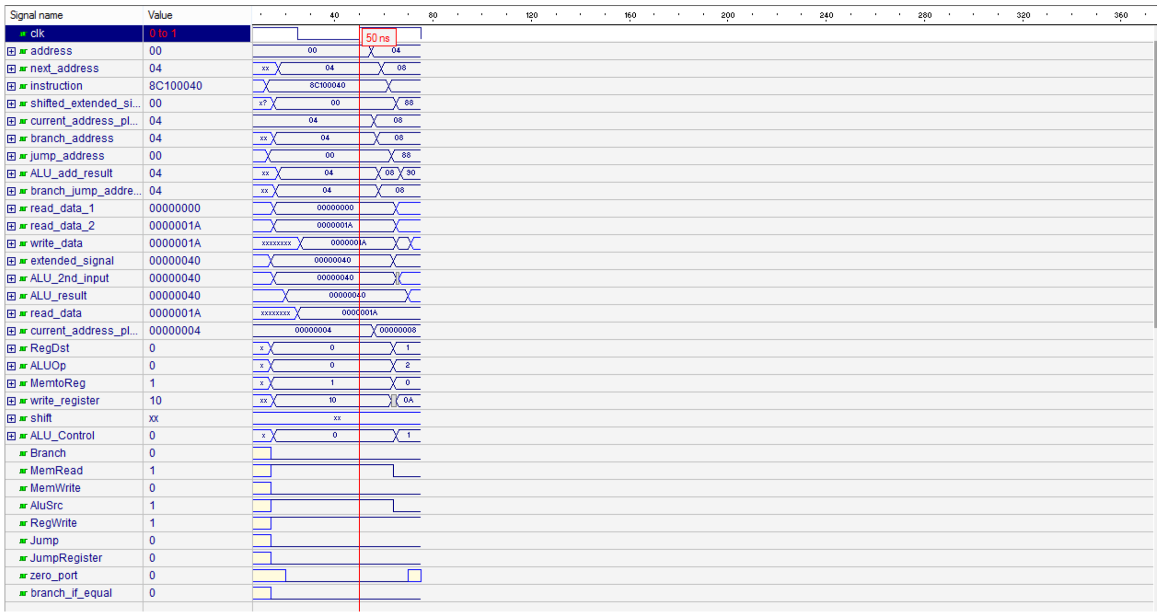
$$Number\ of\ instructions = 6$$

$$Number\ of\ clock\ cycles = 6 + 1 = 7$$

$$simulation\ duration = 7 * 50 = 350ns$$

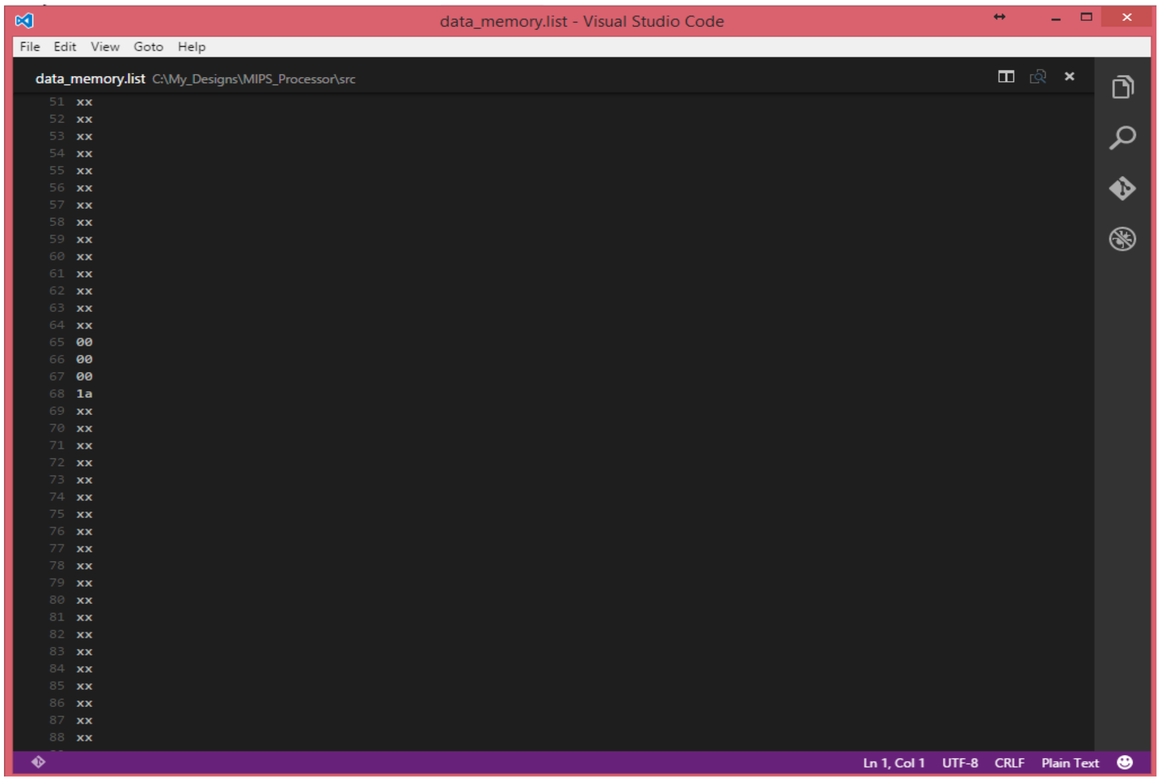
5.3.1 State of Datapath After Each clock Cycle

➢ <u>Instruction #1:</u> [lw $s0 , 64($zero)]



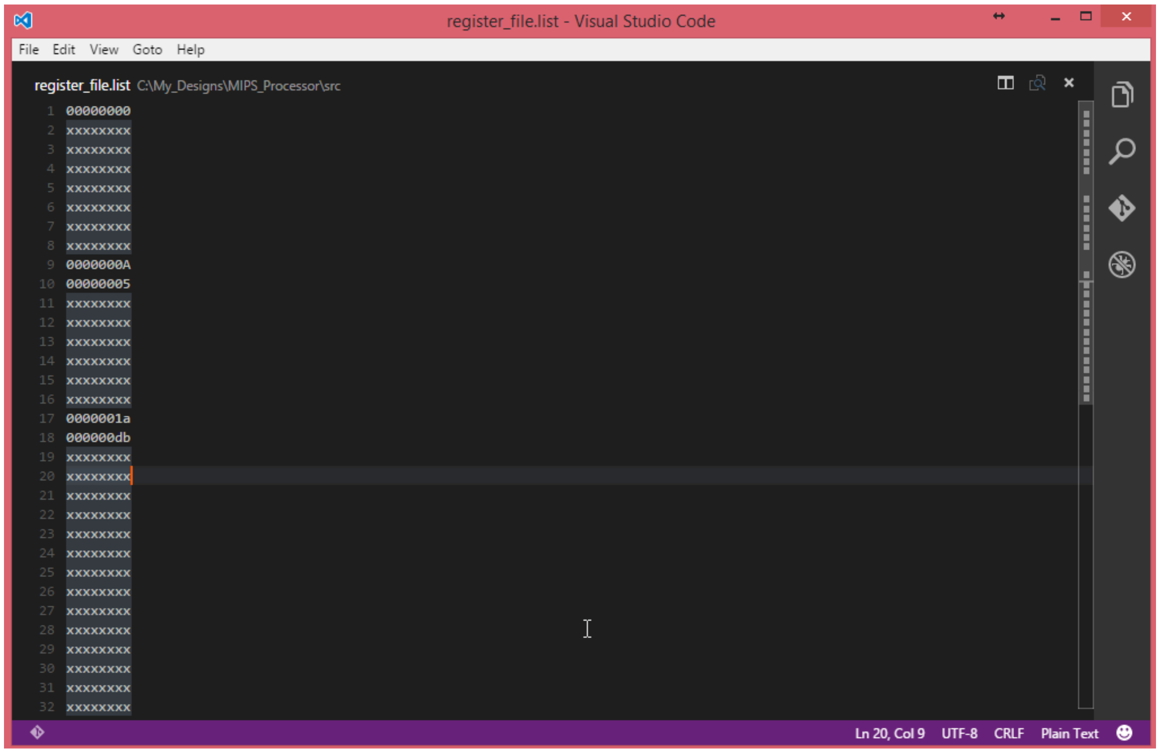
Signal name	Value
clk	0 to 1
address	00
next_address	04
instruction	8C100040
shifted_extended_si...	00
current_address_pl...	04
branch_address	04
jump_address	00
ALU_add_result	04
branch_jump_addre...	04
read_data_1	00000000
read_data_2	0000001A
write_data	0000001A
extended_signal	00000040
ALU_2nd_input	00000040
ALU_result	00000040
read_data	0000001A
current_address_pl...	00000004
RegDst	0
ALUOp	0
MemtoReg	1
write_register	10
shift	xx
ALU_Control	0
Branch	0
MemRead	1
MemWrite	0
AluSrc	1
RegWrite	1
Jump	0
JumpRegister	0
zero_port	0
branch_if_equal	0

Figure 2 instruction#1 waveform


data_memory.list - Visual Studio Code
File Edit View Goto Help
data_memory.list C:\My_Designs\MIPS_Processor\src
Ln 1, Col 1 UTF-8 CRLF Plain Text

```
51 xx
52 xx
53 xx
54 xx
55 xx
56 xx
57 xx
58 xx
59 xx
60 xx
61 xx
62 xx
63 xx
64 xx
65 00
66 00
67 00
68 1a
69 xx
70 xx
71 xx
72 xx
73 xx
74 xx
75 xx
76 xx
77 xx
78 xx
79 xx
80 xx
81 xx
82 xx
83 xx
84 xx
85 xx
86 xx
87 xx
88 xx
```

Figure 3 instruction#1 data memory

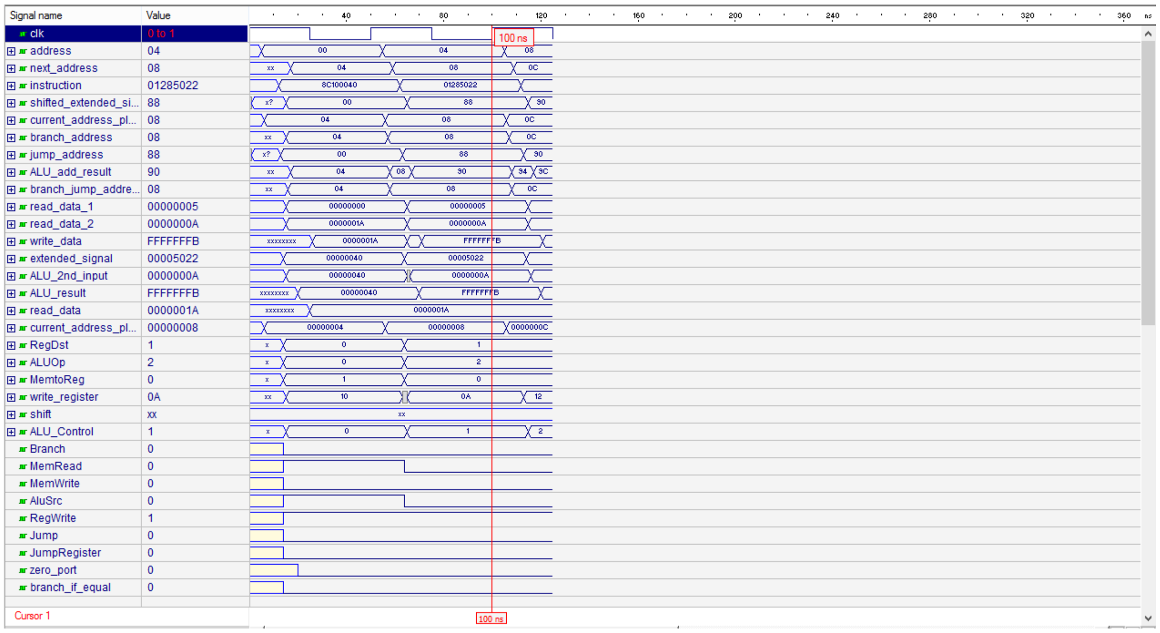

register_file.list - Visual Studio Code
File Edit View Goto Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 20, Col 9 UTF-8 CRLF Plain Text

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000A
00000005
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000001a
000000db
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Figure 4 instruction#1 register file

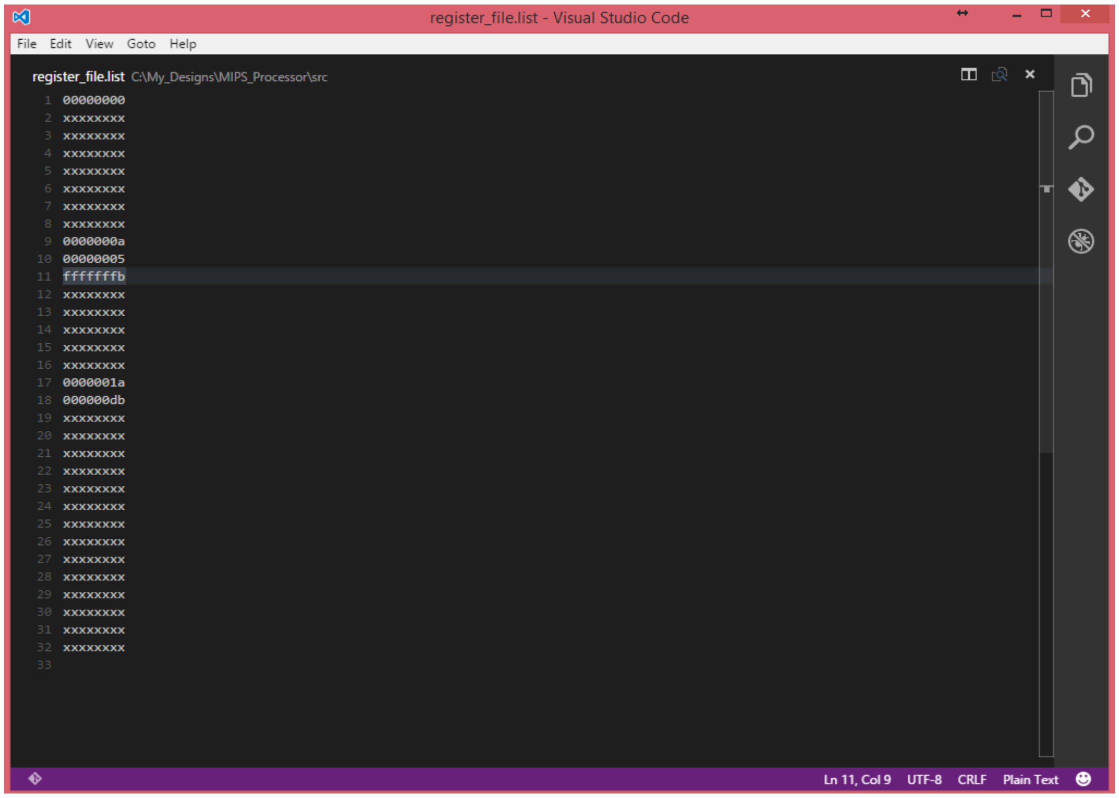
- The content of address 64 inside the data memory is $(1A)_{16}$ as shown in Figure 3, as expected after executing this instruction register \$s0 has the value $(1A)_{16}$ as shown in Figure 4
- Figure 2 shows wave forms of all the internal signal of the datapath after the instruction has finished executing.

➢ <u>Instruction #2:</u> [sub $t2, $t1, $t0]



Signal name	Value
clk	0 to 1
address	04
next_address	08
instruction	01285022
shifted_extended_si...	88
current_address_pl...	08
branch_address	08
jump_address	88
ALU_add_result	90
branch_jump_addre...	08
read_data_1	00000005
read_data_2	0000000A
write_data	FFFFFFFB
extended_signal	00005022
ALU_2nd_input	0000000A
ALU_result	FFFFFFFB
read_data	0000001A
current_address_pl...	00000008
RegDst	1
ALUOp	2
MemtoReg	0
write_register	0A
shift	xx
ALU_Control	1
Branch	0
MemRead	0
MemWrite	0
AluSrc	0
RegWrite	1
Jump	0
JumpRegister	0
zero_port	0
branch_if_equal	0

Figure 5 instruction #2 waveform



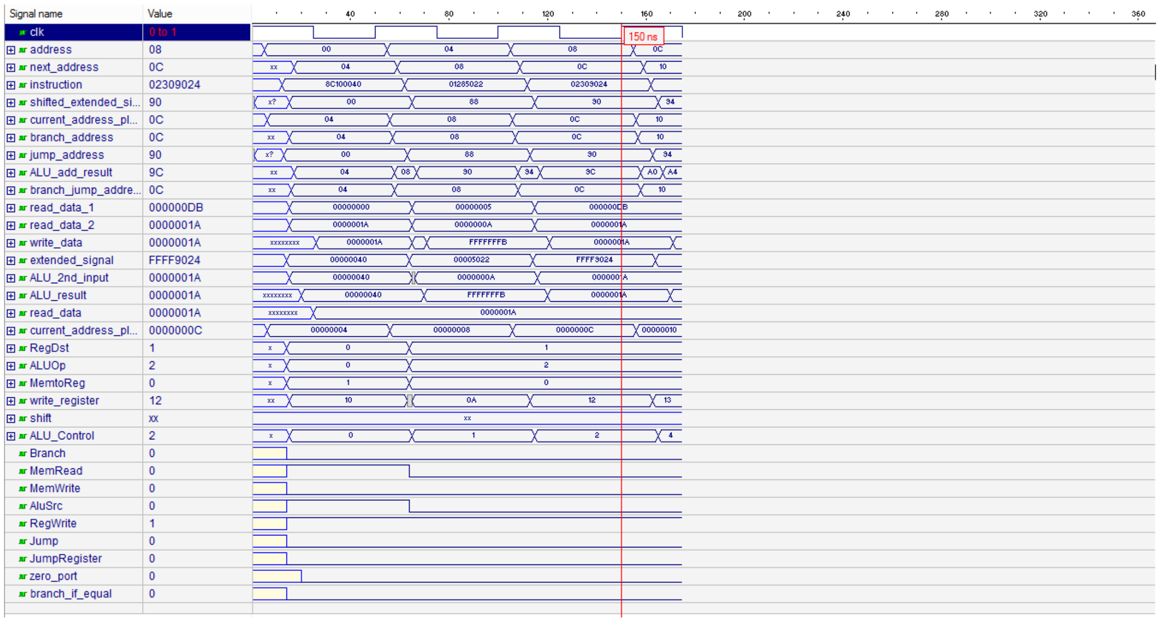
register_file.list - Visual Studio Code
File Edit View Goto Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 11, Col 9 UTF-8 CRLF Plain Text

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000a
00000005
fffffffb
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000001a
000000db
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx

```

Figure 6 instruction #2 register file

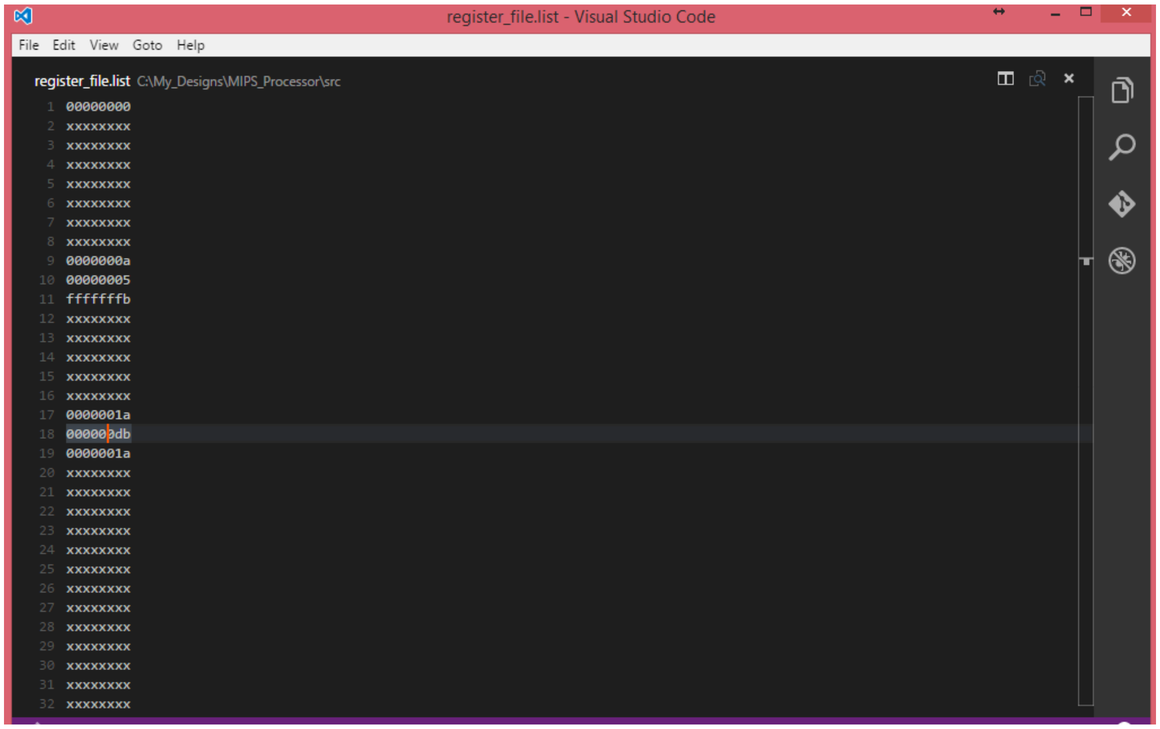
- As expected after executing this instruction register \$t2 has the value $(FFFFFFFb)_{16}$ which is equal to $(05)_{16} - (0A)_{16}$ as shown in Figure 6.
- Figure 5 shows wave forms of all the internal signal of the datapath after the instruction has finished executing.

➢ <u>Instruction #3:</u> [and \$s2, \$s1, \$s0]



Signal name	Value
clk	0 to 1
address	08
next_address	0C
instruction	02309024
shifted_extended_si...	90
current_address_pl...	0C
branch_address	0C
jump_address	90
ALU_add_result	9C
branch_jump_addre...	0C
read_data_1	000000DB
read_data_2	0000001A
write_data	0000001A
extended_signal	FFFF9024
ALU_2nd_input	0000001A
ALU_result	0000001A
read_data	0000001A
current_address_pl...	0000000C
RegDst	1
ALUOp	2
MemtoReg	0
write_register	12
shift	xx
ALU_Control	2
Branch	0
MemRead	0
MemWrite	0
AluSrc	0
RegWrite	1
Jump	0
JumpRegister	0
zero_port	0
branch_if_equal	0

Figure 7 instruction #3 waveform



register_file.list - Visual Studio Code

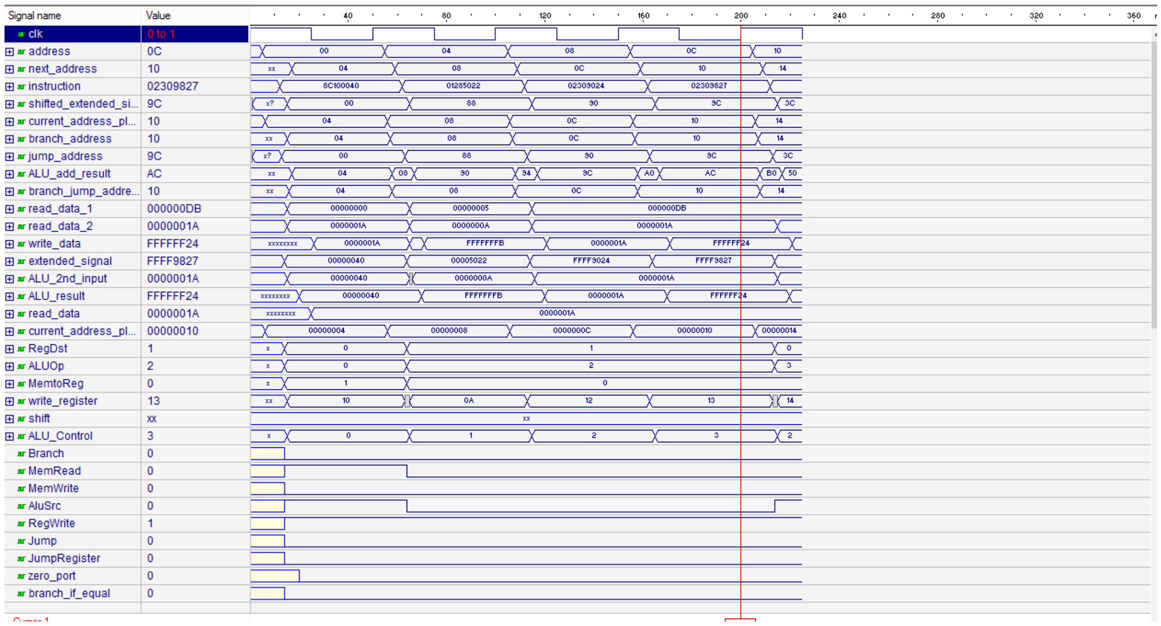
register_file.list C:\My_Designs\MIPS_Processor\src

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000a
00000005
fffffffb
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000001a
000000db
0000001a
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Figure 8 instruction #3 register file

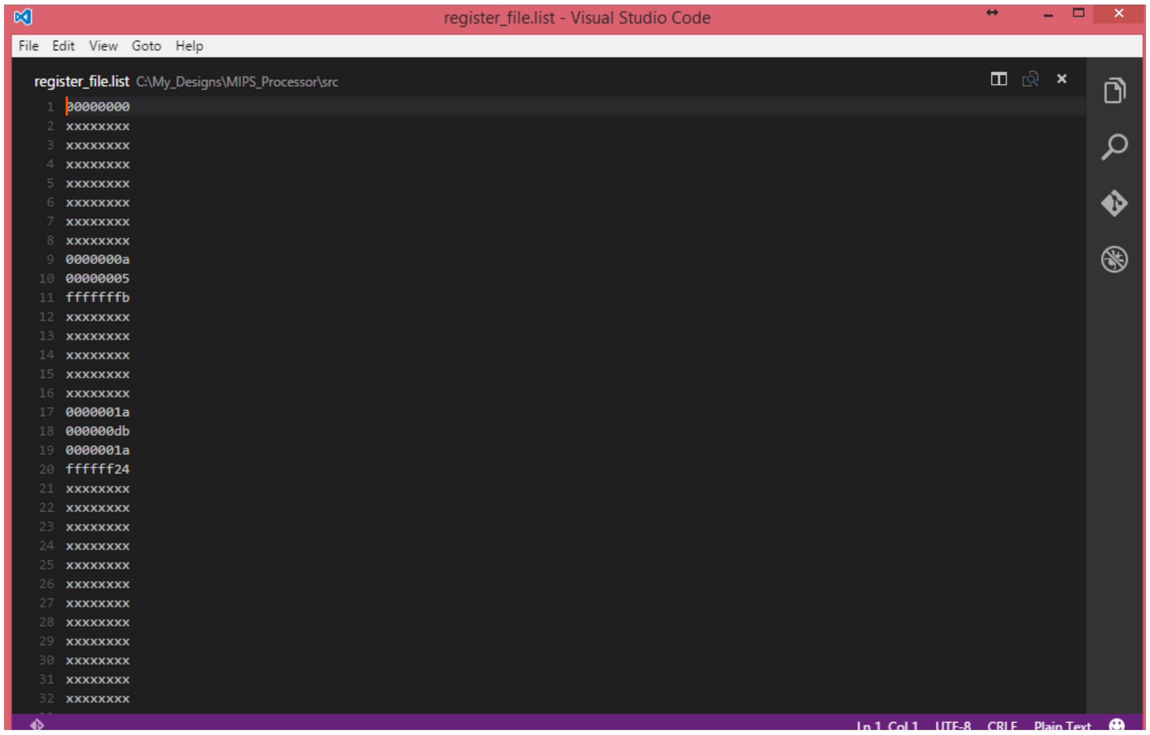
- As expected after executing this instruction register \$s2 has the value $(1A)_{16}$ which is equal to $(1A)_{16}$ && $(DB)_{16}$ as shown in Figure 8.
- Figure 7 shows wave forms of all the internal signal of the datapath after the instruction has finished executing.

➢ <u>Instruction #4:</u> [nor \$s3, \$s1, \$s0]



Signal name	Value
clk	0 to 1
address	0C
next_address	10
instruction	02309827
shifted_extended_si...	9C
current_address_pl...	10
branch_address	10
jump_address	9C
ALU_add_result	AC
branch_jump_addre...	10
read_data_1	000000DB
read_data_2	0000001A
write_data	FFFFFF24
extended_signal	FFFF9827
ALU_2nd_input	0000001A
ALU_result	FFFFFF24
read_data	0000001A
current_address_pl...	00000010
RegDst	1
ALUOp	2
MemtoReg	0
write_register	13
shift	xx
ALU_Control	3
Branch	0
MemRead	0
MemWrite	0
AluSrc	0
RegWrite	1
Jump	0
JumpRegister	0
zero_port	0
branch_if_equal	0

Figure 9 Instruction #4 waveform

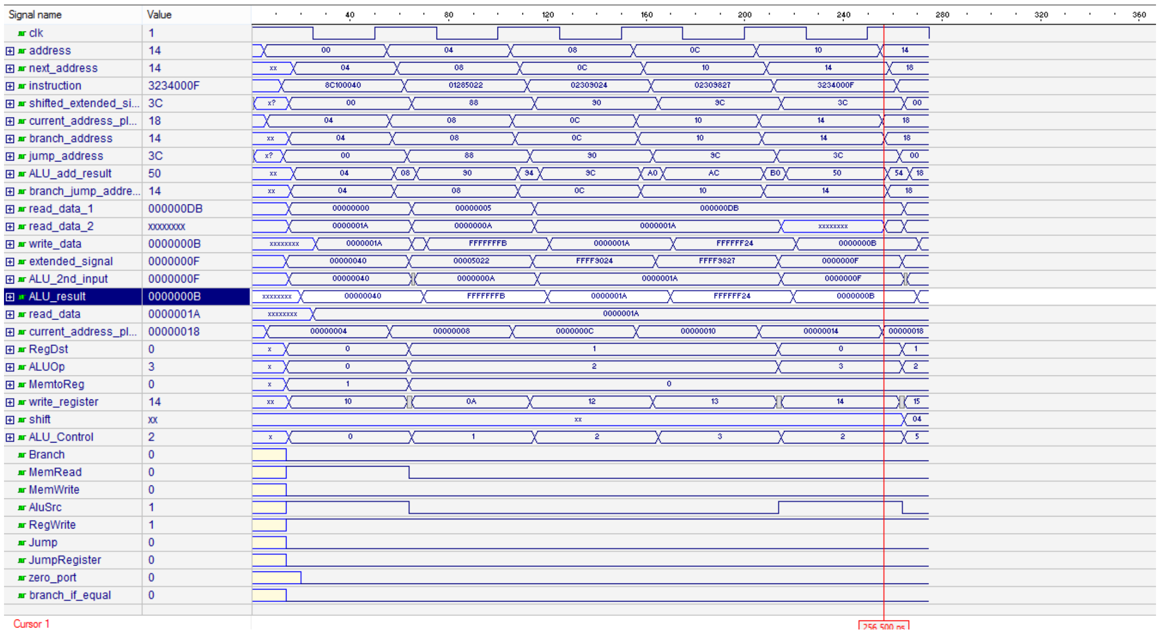

register_file.list - Visual Studio Code
File Edit View Goto Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 1, Col 1 UTF-8 CRLF Plain Text

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000a
00000005
fffffffb
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000001a
000000db
0000001a
ffffff24
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Figure 10 instruction #4 register file

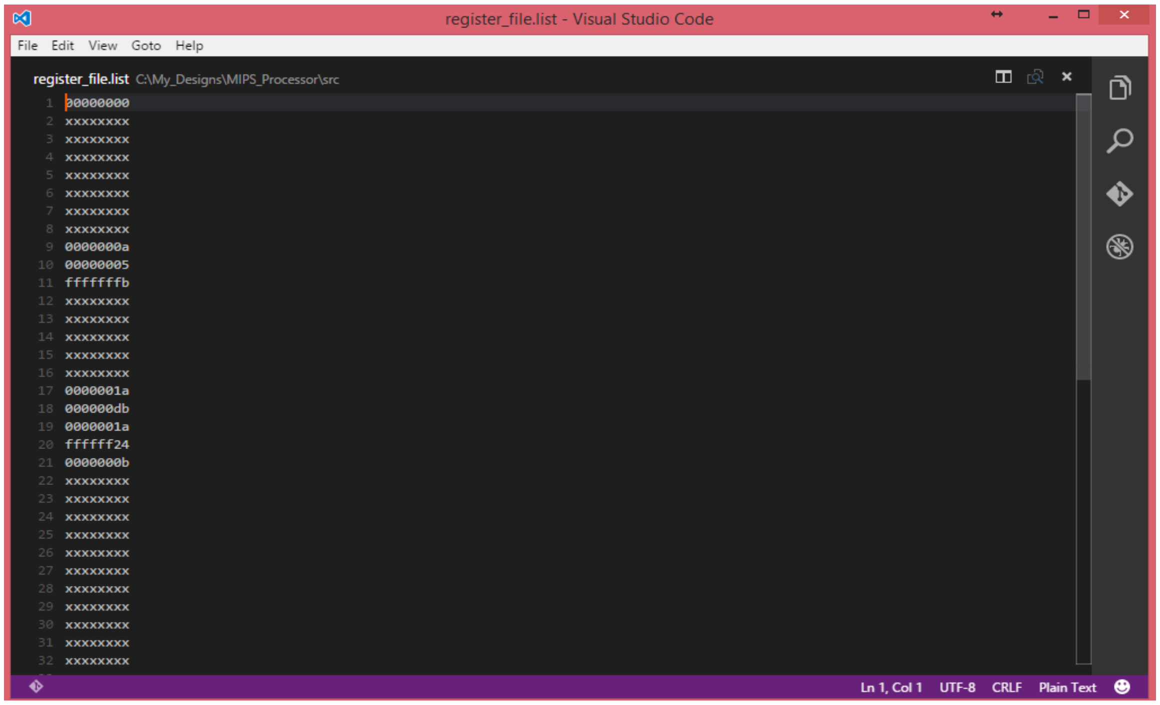
- As expected after executing this instruction register $s3 has the value $(FFFFFF24)_{16}$ which is equal to $(1A)_{16}$ ~| $(DB)_{16}$ as shown in Figure 10.
- Figure 9 shows wave forms of all the internal signal of the datapath after the instruction has finished executing.

➢ <u>Instruction #5:</u> [andi $s4, $s1, 15]



Signal name	Value
clk	1
address	14
next_address	14
instruction	3234000F
shifted_extended_si...	3C
current_address_pl...	18
branch_address	14
jump_address	3C
ALU_add_result	50
branch_jump_addre...	14
read_data_1	000000DB
read_data_2	xxxxxxxx
write_data	0000000B
extended_signal	0000000F
ALU_2nd_input	0000000F
ALU_result	0000000B
read_data	0000001A
current_address_pl...	00000018
RegDst	0
ALUOp	3
MemtoReg	0
write_register	14
shift	xx
ALU_Control	2
Branch	0
MemRead	0
MemWrite	0
AluSrc	1
RegWrite	1
Jump	0
JumpRegister	0
zero_port	0
branch_if_equal	0

Figure 11 Instruction #5 waveform



register_file.list - Visual Studio Code

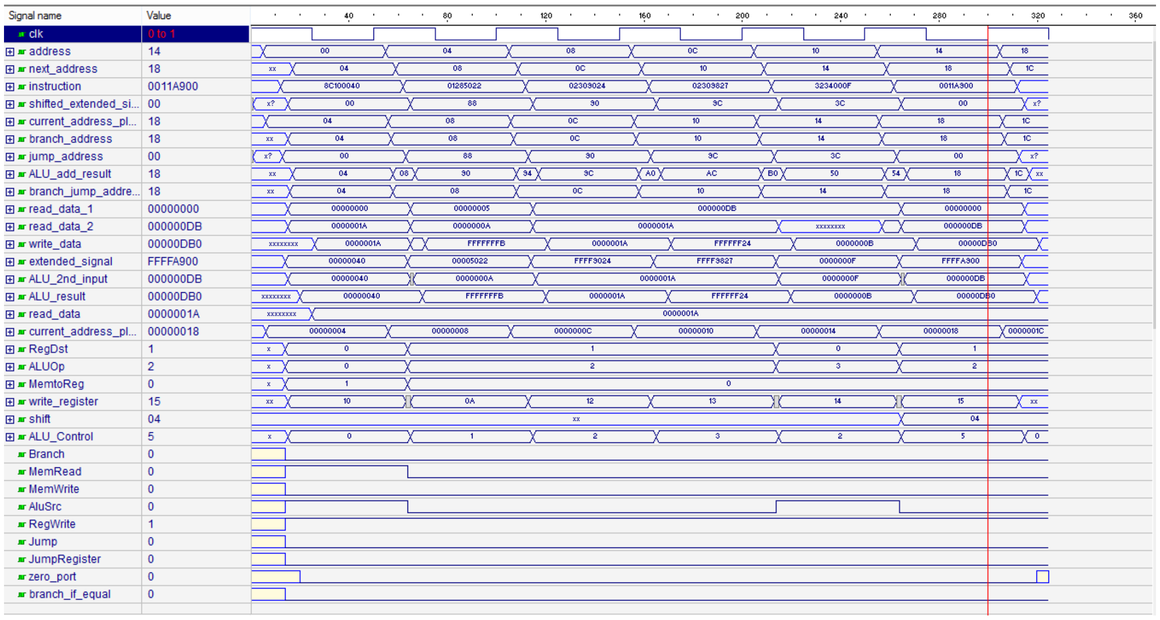
File Edit View Goto Help
register_file.list C:\My_Designs\MIPS_Processor\src
Ln 1, Col 1 UTF-8 CRLF Plain Text

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000a
00000005
fffffffb
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000001a
000000db
0000001a
ffffff24
0000000b
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Figure 12 instruction #5 register file

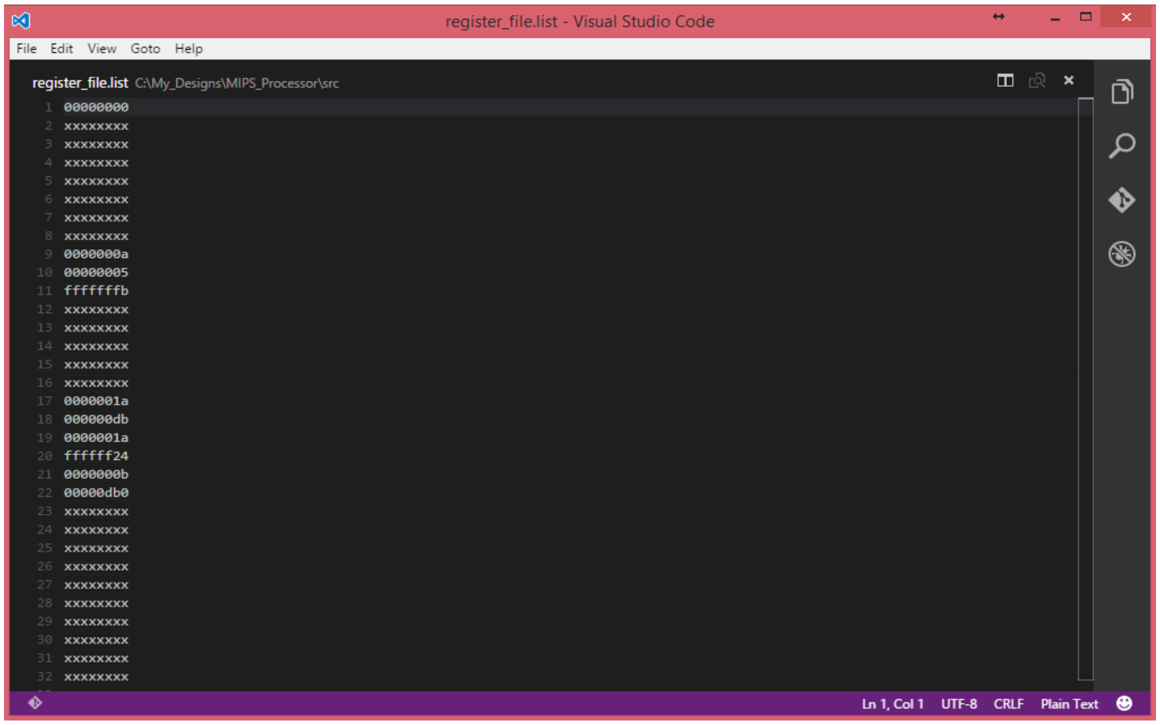
- As expected after executing this instruction register \$s4 has the value $(0B)_{16}$ which is equal to $(1A)_{16}$ && $(15)_{10}$ as shown in Figure 12.
- Figure 11 shows wave forms of all the internal signal of the datapath after the instruction has finished executing.

➢ <u>Instruction #6:</u> [sll \$s5, \$s1, 4]



Signal name	Value
clk	0 to 1
address	14
next_address	18
instruction	0011A900
shifted_extended_si...	00
current_address_pl...	18
branch_address	18
jump_address	00
ALU_add_result	18
branch_jump_addre...	18
read_data_1	00000000
read_data_2	000000DB
write_data	00000DB0
extended_signal	FFFFA900
ALU_2nd_input	000000DB
ALU_result	00000DB0
read_data	0000001A
current_address_pl...	00000018
RegDst	1
ALUOp	2
MemtoReg	0
write_register	15
shift	04
ALU_Control	5
Branch	0
MemRead	0
MemWrite	0
AluSrc	0
RegWrite	1
Jump	0
JumpRegister	0
zero_port	0
branch_if_equal	0

Figure 13 Instruction #6 waveform



register_file.list - Visual Studio Code

register_file.list C:\My_Designs\MIPS_Processor\src

```
00000000
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000000a
00000005
fffffffb
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
0000001a
000000db
0000001a
ffffff24
0000000b
00000db0
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

Ln 1, Col 1 UTF-8 CRLF Plain Text

Figure 14 instruction #6 register file

- As expected after executing this instruction register \$s5 has the value $(DB0)_{16}$ which is equal to $(DB)_{16}$ shifted by 4 bits as shown in Figure 14.
- Figure 13 shows wave forms of all the internal signal of the datapath after the instruction has finished executing.

6.0 References

[1] http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

[2] http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html

[3] https://drive.google.com/open?id=0B6S4xCqu_xTDUzZOX0VjV3p0SjQ