# Gebze Technical University
# CSE331 - Computer Organization

## Homework 4


## Ahmet Tuğkan Ayhan
## 1901042692

# Modules and Their Testbench Results

## nextPC(branch, branchAddress, nextPC, clock)

* With every positive edge of clock, it calculates the next instruction address and returns it as nextPC.
* If branch signal is 1, then instead of adding 1 to current program count it adds branchAddres to nextPC with +1 and returns nextPC
* For testbench, I started program count from 0 and initially gave a branchAddress
* With every positive clock signal it incremented itself with 1.
* After waiting 6ns I changed branch signal to 1 and it added branch address to nextPC with +1.
* In parentheses I showed their hexadecimal values
* Everything worked exactly as it should be

```
# vsim work.nextPC_tb
# Loading work.nextPC_tb
# Loading work.nextPC
VSIM 11> step -current
# Time:   2 | Clock: 1 | Branch: 0
# Branch Address: 00000000000000000000000000011000 (18)
# Next PC       : 00000000000000000000000000000000 (00)
#
# Time:   6 | Clock: 1 | Branch: 0
# Branch Address: 00000000000000000000000000011000 (18)
# Next PC       : 00000000000000000000000000000001 (01)
#
# Time:  10 | Clock: 1 | Branch: 1
# Branch Address: 00000000000000000000000000011000 (18)
# Next PC       : 00000000000000000000000000000010 (02)
#
# Time:  14 | Clock: 1 | Branch: 0
# Branch Address: 00000000000000000000000000011000 (18)
# Next PC       : 00000000000000000000000000011011 (1b)
#
# Time:  18 | Clock: 1 | Branch: 0
# Branch Address: 00000000000000000000000000011000 (18)
# Next PC       : 00000000000000000000000000011100 (1c)
#
# Time:  22 | Clock: 1 | Branch: 0
# Branch Address: 00000000000000000000000000011000 (18)
# Next PC       : 00000000000000000000000000011101 (1d)
#
```

## InstructionMemory(read_address, instruction)

* With every clock signal(positive or negative), it reads the instruction from the instruction memory.
* The address it reads from instruction memory is given with read_address
* Then it finds the instruction at read_address value and returns it
* In testbench I initially gave 3 read addresses and placed instructions inside to these addresses.
* It successfully returned the correct instructions

```
# Loading work.InstructionMemory_tb
# Loading work.InstructionMemory
VSIM 12> step -current
# Time         :  0
# Read Address: 00000000000000000000000000000000
# Instruction : 1111111111111111
#
# Time         : 10
# Read Address: 00000000000000000000000000000001
# Instruction : 0000000011111111
#
# Time         : 20
# Read Address: 00000000000000000000000000000010
# Instruction : 1010101010101010
#
```

**Control(Opcode, RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp)**

* Control module takes only one input, which is Opcode.
* According to 4bit Opcode value, it calculates rest of the parameters.
* To calculate all signals, first I drew a truth table for control unit.

| Control Signals | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Instructions | Opcode | RegDst | ALUSrc | MemtoReg | RegWrite | MemRead | MemWrite | Branch | ALUOp |
| R - Types | 0000 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 000 |
| ADDI | 0001 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 001 |
| ANDI | 0010 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 010 |
| ORI | 0011 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 011 |
| NORI | 0100 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 100 |
| BEQ | 0101 | x | 0 | x | 0 | 0 | 0 | 1 | 101 |
| BNE | 0110 | x | 0 | x | 0 | 0 | 0 | 1 | 101 |
| SLTI | 0111 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 110 |
| LW | 1000 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 001 |
| SW | 1001 | x | 1 | x | 0 | 0 | 1 | 0 | 001 |

| Truth Table | |
|---|---|
| **RegDst =** | R-Types |
| **ALUSrc =** | ADDI + ANDI + ORI + NORI + SLTI + LW + SW |
| **MemtoReg =** | LW |
| **RegWrite =** | R-Types + ADDI + ANDI + ORI + NORI + SLTI + LW |
| **MemRead =** | LW |
| **MemWrite =** | SW |
| **Branch =** | BEQ + BNE |
| **ALUOp_2 =** | NORI + BEQ + BNE + SLTI |
| **ALUOp_1 =** | ANDI + ORI + SLTI |
| **ALUOp_0 =** | ADDI + ORI + BEQ + BNE + LW + SW |

* After drawing truth table, I derived every signal's boolean expression, then I used them in the module.

```
# vsim work.control_tb
# Loading work.control_tb
# Loading work.control
VSIM 9> step -current
# Time:   0 | Opcode: 0000 R-types
# RegDst: 1 | ALUSrc: 0 | MemtoReg: 0 | RegWrite: 1 | MemRead: 0 | MemWrite: 0 | Branch: 0 | ALUOp:  000
#
# Time:  20 | Opcode: 0001 ADDI
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 0 | RegWrite: 1 | MemRead: 0 | MemWrite: 0 | Branch: 0 | ALUOp:  001
#
# Time:  40 | Opcode: 0010 ANDI
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 0 | RegWrite: 1 | MemRead: 0 | MemWrite: 0 | Branch: 0 | ALUOp:  010
#
# Time:  60 | Opcode: 0011 ORI
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 0 | RegWrite: 1 | MemRead: 0 | MemWrite: 0 | Branch: 0 | ALUOp:  011
#
# Time:  80 | Opcode: 0100 NORI
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 0 | RegWrite: 1 | MemRead: 0 | MemWrite: 0 | Branch: 0 | ALUOp:  100
#
# Time: 100 | Opcode: 0101 BEQ
# RegDst: 0 | ALUSrc: 0 | MemtoReg: 0 | RegWrite: 0 | MemRead: 0 | MemWrite: 0 | Branch: 1 | ALUOp:  101
#
# Time: 120 | Opcode: 0110 BNE
# RegDst: 0 | ALUSrc: 0 | MemtoReg: 0 | RegWrite: 0 | MemRead: 0 | MemWrite: 0 | Branch: 1 | ALUOp:  101
#
# Time: 140 | Opcode: 0111 SLTI
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 0 | RegWrite: 1 | MemRead: 0 | MemWrite: 0 | Branch: 0 | ALUOp:  110
#
# Time: 160 | Opcode: 1000 LW
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 1 | RegWrite: 1 | MemRead: 1 | MemWrite: 0 | Branch: 0 | ALUOp:  001
#
# Time: 180 | Opcode: 1001 SW
# RegDst: 0 | ALUSrc: 1 | MemtoReg: 0 | RegWrite: 0 | MemRead: 0 | MemWrite: 1 | Branch: 0 | ALUOp:  001
#
```

**Register(read_reg1, read_reg2, write_reg, write_data, RegWrite, read_data1, read_data2, clock)**

* Register module takes 6 input, which are
    - **read_reg1, read_reg2:** They hold 3 bit register addresses. Since there are 8 registers and instruction width(16bit) allows to only 3 bit addresses I used them like that. read_reg1 represents **rs** register's address and read_reg2 represents **rt** register's address.
    - **write_reg:** It holds write register's address. It doesn't care which register's(**rt** or **rd**) address comes since it directly takes the address. Which register's address will go through is decided in MiniMIPS module using 3 bit 2x1 MUX.
    - **write_data:** It holds what data will be written inside the write_reg. The write_data value comes from DataMemory module
    - **RegWrite:** Control signal for Register module. If it is 1, then write_data value is going to be written into the register with write_reg address. If it is 0, then there will be no write operation, instead it will read register values using read_reg1 and read_reg2 values and return them with read_data1(**rs**) and read_data2(**rt**)
    - **clock:** Clock basically decides when the operation will be done(reading or writing). When it's posedge comes it will write if write signal is also 1. Reading operation is constantly done with every clock signal(positive or negative).

* There are only 2 outputs:
    - **read_data1, read_data2:** read_data1 holds read_reg1 register's value and read_data2 holds read_reg2 register's value.

```
# Loading work.register_tb
# Loading work.register
VSIM 29> step -current
# Time:    0 | Clock: 0 | RegWrite: 1 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 001
# Write Data : 11111111111111110001000100010001
# Read Data 1: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:    2 | Clock: 1 | RegWrite: 1 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 001
# Write Data : 11111111111111110001000100010001
# Read Data 1: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:    4 | Clock: 0 | RegWrite: 0 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 001
# Write Data : 11111111111111110001000100010001
# Read Data 1: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:    6 | Clock: 1 | RegWrite: 0 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 001
# Write Data : 11111111111111110001000100010001
# Read Data 1: 11111111111111110001000100010001
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:    8 | Clock: 0 | RegWrite: 1 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 010
# Write Data : 11111111111111110000000000000000
# Read Data 1: 11111111111111110001000100010001
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   10 | Clock: 1 | RegWrite: 1 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 010
# Write Data : 11111111111111110000000000000000
# Read Data 1: 11111111111111110001000100010001
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   12 | Clock: 0 | RegWrite: 0 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 010
# Write Data : 11111111111111110000000000000000
# Read Data 1: 11111111111111110001000100010001
# Read Data 2: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   14 | Clock: 1 | RegWrite: 0 | Read Register 1: 001 | Read Register 2: 010 | Write Register: 010
# Write Data : 11111111111111110000000000000000
# Read Data 1: 11111111111111110001000100010001
# Read Data 2: 11111111111111110000000000000000
```

## ALUcontrol(ALUOp, func, ALUctr)

* ALU control module takes 2 input(ALUop and func) and returns an output (ALUctr).
* According to ALUop and func values it decides to which operation will be done inside the ALU.
* The use ALUop and func values, I first created a truth table and then used truth table to extract boolean expressions.

| ALU Control | | | | |
|---|---|---|---|---|
| Instructions | ALUOp (P) | Function (F) | Desired Alu Action | ALUctr |
| AND | 000 | 000 | and | 000 |
| ADD | 000 | 001 | add | 001 |
| SUB | 000 | 010 | sub | 010 |
| XOR | 000 | 011 | xor | 011 |
| NOR | 000 | 100 | nor | 100 |
| OR | 000 | 101 | or | 101 |
| ADDI | 001 | xxx | add | 001 |
| ANDI | 010 | xxx | and | 000 |
| ORI | 011 | xxx | or | 101 |
| NORI | 100 | xxx | nor | 100 |
| BEQ | 101 | xxx | sub | 010 |
| BNE | 101 | xxx | sub | 010 |
| SLTI | 110 | xxx | slt | 110 |
| LW | 001 | xxx | add | 001 |
| SW | 001 | xxx | add | 001 |
| Truth Table | | | | |
| **ALUctr_2 =** | P2'.P1'.P0'.F2.F1' + P2'.P1.P0 + P2.P1'.P0' + P2.P1.P0' | | | |
| **ALUctr_1 =** | P2'.P1'.P0'.F2'.F1 + P2.P1'.P0 + P2.P1.P0' | | | |
| **ALUctr_0 =** | P2'.P1'.P0'.F0 + P2'.P0 | | | |

```
# Loading work.ALUcontrol
VSIM 4> step -current
# Time:   0 | ALUOp: 000 | Function: 000 And
# ALUctr: 000
#
# Time:  20 | ALUOp: 000 | Function: 001 Add
# ALUctr: 001
#
# Time:  40 | ALUOp: 000 | Function: 010 Sub
# ALUctr: 010
#
# Time:  60 | ALUOp: 000 | Function: 011 Xor
# ALUctr: 011
#
# Time:  80 | ALUOp: 000 | Function: 100 nor
# ALUctr: 100
#
# Time: 100 | ALUOp: 000 | Function: 101 Or
# ALUctr: 101
#
# Time: 120 | ALUOp: 001 | Function: zzz addi
# ALUctr: 001
#
# Time: 140 | ALUOp: 010 | Function: zzz andi
# ALUctr: 000
#
# Time: 160 | ALUOp: 011 | Function: zzz ori
# ALUctr: 101
#
# Time: 180 | ALUOp: 100 | Function: zzz nori
# ALUctr: 100
#
# Time: 200 | ALUOp: 101 | Function: zzz beq-bne
# ALUctr: 010
#
# Time: 240 | ALUOp: 110 | Function: zzz slti
# ALUctr: 110
#
# Time: 260 | ALUOp: 001 | Function: zzz lw-sw
# ALUctr: 001
```

## alu_32bit(value1, value2, select, result)

* This part of the project is already done with homework3 but I changed the operation order according to homework 4 and recompiled the testbench.
* ALU takes 2  32-bit value and makes an operation with these values according to 3-bit select input and returns it as 32-bit result.

```
VSIM 31> step -current
# Time  : 0 - Select: 000
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 00000000000000000000000000100001 And
#
# Time  : 20 - Select: 001
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 00000000000000000000000001011100 Add
#
# Time  : 40 - Select: 010
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 00000000000000000000000000001010 Sub
#
# Time  : 60 - Select: 011
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 00000000000000000000000000011010 Xor
```

```
#
# Time  : 80 - Select: 100
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 11111111111111111111111111000100 Nor
#
# Time  : 100 - Select: 101
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 00000000000000000000000000111011 Or
#
# Time  : 120 - Select: 110
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: 00000000000000000000000000000000 Slt
#
# Time  : 140 - Select: 111
# value1: 00000000000000000000000000110011
# value2: 00000000000000000000000000101001
# result: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx mult
```

## DataMemory(MemWrite, MemRead, address, write_data, read_data, clock)

* Data Memory module takes 6 parameters which are;
  - **MemWrite, MemRead:** These are 1-bit signals coming from Control unit. When MemWrite is 1, module writes 32-bit write_data into the memory block with given address. If MemRead is 1, then module reads the data at memory[address] and assigns the value into read_data(which is the only output).
  - **address, write_data, read_data:** There only 2 operations in the module. Either it will read the value from memory with memory[address] and assign it into the read_data or it will write the write_data value into memory[address]. All of these parameters are 32-bit.
  - **clock:** with every positive edge of the clock write operation occurs if MemWrite is also 1. Reading operation happens with every clock signal(positive or negative) if MemRead is also 1.

```
VSIM 20> run
# Time:   0 | Clock: 0 | MemWrite: 1 | MemRead: 0
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   2 | Clock: 1 | MemWrite: 1 | MemRead: 0
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   4 | Clock: 0 | MemWrite: 1 | MemRead: 0
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   6 | Clock: 1 | MemWrite: 1 | MemRead: 0
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:   8 | Clock: 0 | MemWrite: 1 | MemRead: 0
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# Time:  10 | Clock: 1 | MemWrite: 0 | MemRead: 1
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : 11111111111111111111111111110001
# Time:  12 | Clock: 0 | MemWrite: 0 | MemRead: 1
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : 11111111111111111111111111110001
# Time:  14 | Clock: 1 | MemWrite: 0 | MemRead: 1
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : 11111111111111111111111111110001
# Time:  16 | Clock: 0 | MemWrite: 0 | MemRead: 1
# Address   : 00000000000000000000000000000001
# Write Data: 11111111111111111111111111110001
# Read Data : 11111111111111111111111111110001
```

## signExtend_6to32(value, result)

* Takes a 6-bit input and extends it to 32bit.
* Most significant bit is important

```
ModelSim> vsim work.signExtend_6to32_tb
# vsim work.signExtend_6to32_tb
# Loading work.signExtend_6to32_tb
# Loading work.signExtend_6to32
VSIM 4> step -current
# Time  :  0
# Value :                              011101
# Result: 00000000000000000000000000011101
#
# Time  : 20
# Value :                              100010
# Result: 11111111111111111111111111100010
#
```

## shiftLeft_32bit(value, result)

* Takes a 32-bit input and shifts it to the left 1-bit

```
# vsim work.shiftLeft_32bit_tb
# Loading work.shiftLeft_32bit_tb
# Loading work.shiftLeft_32bit
VSIM 12> step -current
# Time  :  0
# Value : 10101010101010101010101010101010
# Result: 01010101010101010101010101010100
#
# Time  : 20
# Value : 11001100110011000001001000110100
# Result: 10011001100110000010010001101000
#
```

## MiniMIPS(clock)

* We now came to the real part. This is where the magic happens :)
* MiniMIPS module only takes 1-bit clock input and it calls other modules in this order:

1. **SignExtend:** Since branchAddress(which comes from instruction's least 6 bit) is 6 bit, we need to extend it from 6 to 32. So we can calculate which instruction to go when **beq** or **bne** instructions used.
2. **nextPC:** Calculates next instruction address.
3. **InstructionMemory:** Gets next instruction from memory.
4. **Control:** Creates all control signals according to the instruction's opcode
5. **Register:** Reads/writes the data from/to registers according to instruction type and control signals
6. **ALUcontrol:** Decides which operation ALU is going to do
7. **ALU:** Executes the operation and returns the result
8. **DataMemory:** Uses result of the ALU operation to write. Or it can read, depends to the control signal.

* Before showing the testbench result of the MiniMIPS, first I will show before - after status of the txt files.
* **Input txt files:** instruction.txt, data.txt, register.txt
* **Output txt filex:** registerOut.txt, dataOut.txt

### instruction.txt

* Readmemb works fine even after you add comments. So I added as much as I can to explain it better.

```
// R-Types: opcode-rs-rt-rd-funct
0000_011_001_010_000 // AND $2, $0, $1
0000_010_011_100_000 // AND $4, $2, $3
0000_000_001_010_001 // ADD $2, $0, $1
0000_010_011_100_001 // ADD $4, $2, $3
0000_000_001_010_010 // SUB $2, $0, $1
0000_010_011_100_010 // SUB $4, $2, $3
0000_000_001_010_011 // XOR $2, $0, $1
0000_010_011_100_011 // XOR $4, $2, $3
0000_000_001_010_100 // NOR $2, $0, $1
0000_010_011_100_100 // NOR $4, $2, $3
0000_000_001_010_101 // OR  $2, $0, $1
0000_010_011_100_101 // OR, $4, $2, $3
0001_000_101_000111  // addi $5, $1, 000111
0001_101_110_010101  // addi $6, $5, 010101
0010_000_101_000111  // andi $5, $0, 000111
0010_101_110_010101  // andi $6, $5, 010101
0011_000_101_000111  // ori  $5, $0, 000111
0011_101_110_010101  // ori  $6, $5, 010101
0100_000_101_000111  // nori $5, $0, 000111
0100_101_110_010101  // nori $6, $5, 010101
0101_111_101_000011  // beq  $5, $7, 000001
0101_101_110_010101  // beq  $6, $5, 010101
0110_000_101_000111  // bne  $5, $0, 000111
0110_101_110_010101  // bne  $6, $5, 010101
0111_000_101_100111  // slti $5, $0, 100111
0111_101_110_010101  // slti $6, $5, 010101
1000_000_110_000001  // lw   $6, (0)$0
1000_000_111_000100  // lw   $7, (4)$0
1001_000_110_010100  // sw   $6, (20)$0
1001_000_111_011000  // sw   $7, (24)$0
```

**data.txt**

```
1  00000000000000000000000000000001
2  00000000000000000000000000000011
3  00000000000000000000000000000111
4  00000000000000000000000000001111
5  00000000000000000000000000011111
6  00000000000000000000000000111111
7  00000000000000000000000001111111
8  00000000000000000000000011111111
9  00000000000000000000000111111111
10 00000000000000000000001111111111
11 00000000000000000000011111111111
12 00000000000000000000111111111111
13 00000000000000000001111111111111
14 00000000000000000011111111111111
15 00000000000000000111111111111111
16 00000000000000001111111111111111
17 11111111111111110000000000000000
18 11111111111111110000000000000000
19 11111111111111110000000000000000
20 11111111111111110000000000000000
21 11111111111111110000000000000000
22 11111111111111110000000000000000
23 11111111111111110000000000000000
24 11111111111111110000000000000000
25 11111111111111110000000000000000
26 11111111111111110000000000000000
27 11111111111111110000000000000000
28 11111111111111110000000000000000
29 11111111111111111111111111111111
30 11111111111111111111111111111111
31 11111111111111111111111111111111
32 11111111111111111111111111111111
```

**dataOut.txt**

```
1  00000000000000000000000000000001
2  00000000000000000000000000000011
3  00000000000000000000000000000111
4  00000000000000000000000000001111
5  00000000000000000000000000011111
6  00000000000000000000000000111111
7  00000000000000000000000001111111
8  00000000000000000000000011111111
9  00000000000000000000000111111111
10 00000000000000000000001111111111
11 00000000000000000000011111111111
12 00000000000000000000111111111111
13 00000000000000000001111111111111
14 00000000000000000011111111111111
15 00000000000000000111111111111111
16 00000000000000001111111111111111
17 11111111111111110000000000000000
18 11111111111111110000000000000000
19 11111111111111110000000000000000
20 11111111111111110000000000000000
21 00000000000000000000000000000011
22 11111111111111110000000000000000
23 11111111111111110000000000000000
24 11111111111111110000000000000000
25 00000000000000000000000000011111
26 11111111111111110000000000000000
27 11111111111111110000000000000000
28 11111111111111110000000000000000
29 11111111111111111111111111111111
30 11111111111111111111111111111111
31 11111111111111111111111111111111
32 11111111111111111111111111111111
```

* Most of the data is unused because only lw and sw instructions access to the memory and only sw changes it. Since I used 2 sw instruction there are only 2 changes on the data memory(which are on the line 21 and 25).

**register.txt**

```
00000000000000000000000000000000 // $0
00000000000000001010101010101010 // $1
00000000000000001111111100000000 // $2
00000000000000000000000011111111 // $3
00000000000000000001111100001111 // $4
00000000000000000000000000000000 // $5
00000000000000000000000000000000 // $6
11111111111111111111111111111000 // $7
```

**registerOut.txt**

```
00000000000000000000000000000000 // $0
00000000000000001010101010101010 // $1
00000000000000001010101010101010 // $2
00000000000000000000000011111111 // $3
00000000000000001010101011111111 // $4
00000000000000000000000000000000 // $5
00000000000000000000000000000011 // $6
00000000000000000000000000011111 // $7
```

***** Since I finished showing txt files now I will show testbench results.

* Before starting, this is the last explanation for the report and I want to mention that beq command works fine but bne works same as beq so there are a little problem with bne. I would solve it with a control signal but I have unfortunately no time for that. In the testbench, **andi** operation looks like does nothing but actually it does. If you give different values or immediate values you can see that it works correctly.

* Lastly, beq operation in the testbench subtracts $7 from $5 and finds 0, then it jumps 4 instruction ahead, since I gave immediate value as 000011 (+1 from nextPC). You can see the difference by looking to the PC count. LW and SW operations works fine as like as other operations except bne. The only problem I can see in the project is BNE.

**Thanks for reading**

**Testbench Results of MiniMIPS**

```
VSIM 13> vsim work.MiniMIPS_tb
# vsim work.MiniMIPS_tb
# Loading work.MiniMIPS_tb
# Loading work.MiniMIPS
# Loading work.signExtend_6to32
# Loading work.nextPC
# Loading work.InstructionMemory
# Loading work.control
# Loading work.mux2x1_3bit
# Loading work.mux2x1
# Loading work.register
# Loading work.ALUcontrol
# Loading work.mux2x1_32bit
# Loading work.alu_32bit
# Loading work.adder_32bit
# Loading work.adder_4bit
# Loading work.full_adder
# Loading work.half_adder
# Loading work.xor_32bit
# Loading work.sub_32bit
# Loading work.not_32bit
# Loading work.slt_32bit
# Loading work.nor_32bit
# Loading work.and_32bit
# Loading work.or_32bit
# Loading work.mux8x1_32bit
# Loading work.mux8x1
# Loading work.DataMemory
```

```
VSIM 5> step -current
# Time  :    2 | Clock: 1    | PC(int):  0 | Instruction: 0000011001010000 | ALUctr: 000
# Opcode: 0000 | Rs: 011($3) | Rt: 001($1) | Rd: 010($2) | Func: 000
# Operation: AND
# Value 1: 00000000000000000000000011111111
# Value 2: 00000000000000001010101010101010
# Result : 00000000000000000000000010101010
# Time  :    6 | Clock: 1    | PC(int):  1 | Instruction: 0000010011100000 | ALUctr: 000
# Opcode: 0000 | Rs: 010($2) | Rt: 011($3) | Rd: 100($4) | Func: 000
# Operation: AND
# Value 1: 00000000000000000000000010101010
# Value 2: 00000000000000000000000011111111
# Result : 00000000000000000000000010101010
# Time  :   10 | Clock: 1    | PC(int):  2 | Instruction: 0000000001010001 | ALUctr: 001
# Opcode: 0000 | Rs: 000($0) | Rt: 001($1) | Rd: 010($2) | Func: 001
# Operation: ADD
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000001010101010101010
# Result : 00000000000000001010101010101010
# Time  :   14 | Clock: 1    | PC(int):  3 | Instruction: 0000010011100001 | ALUctr: 001
# Opcode: 0000 | Rs: 010($2) | Rt: 011($3) | Rd: 100($4) | Func: 001
# Operation: ADD
# Value 1: 00000000000000001010101010101010
# Value 2: 00000000000000000000000011111111
# Result : 00000000000000001010101110101001
# Time  :   18 | Clock: 1    | PC(int):  4 | Instruction: 0000000001010010 | ALUctr: 010
# Opcode: 0000 | Rs: 000($0) | Rt: 001($1) | Rd: 010($2) | Func: 010
# Operation: SUB
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000001010101010101010
# Result : 11111111111111111010101010101010
# Time  :   22 | Clock: 1    | PC(int):  5 | Instruction: 0000010011100010 | ALUctr: 010
# Opcode: 0000 | Rs: 010($2) | Rt: 011($3) | Rd: 100($4) | Func: 010
# Operation: SUB
# Value 1: 11111111111111111010101010101010
# Value 2: 00000000000000000000000011111111
# Result : 11111111111111111010101000101011
```

```
# Time  :   26 | Clock: 1    | PC(int):  6 | Instruction: 0000000001010011 | ALUctr: 011
# Opcode: 0000 | Rs: 000($0) | Rt: 001($1) | Rd: 010($2) | Func: 011
# Operation: XOR
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000001010101010101010
# Result : 00000000000000001010101010101010
# Time  :   30 | Clock: 1    | PC(int):  7 | Instruction: 0000010011100011 | ALUctr: 011
# Opcode: 0000 | Rs: 010($2) | Rt: 011($3) | Rd: 100($4) | Func: 011
# Operation: XOR
# Value 1: 00000000000000001010101010101010
# Value 2: 00000000000000000000000011111111
# Result : 00000000000000001010101001010101
# Time  :   34 | Clock: 1    | PC(int):  8 | Instruction: 0000000001010100 | ALUctr: 100
# Opcode: 0000 | Rs: 000($0) | Rt: 001($1) | Rd: 010($2) | Func: 100
# Operation: NOR
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000001010101010101010
# Result : 11111111111111111010101010101010
# Time  :   38 | Clock: 1    | PC(int):  9 | Instruction: 0000010011100100 | ALUctr: 100
# Opcode: 0000 | Rs: 010($2) | Rt: 011($3) | Rd: 100($4) | Func: 100
# Operation: NOR
# Value 1: 11111111111111111010101010101010
# Value 2: 00000000000000000000000011111111
# Result : 00000000000000001010101000000000
# Time  :   42 | Clock: 1    | PC(int): 10 | Instruction: 0000000001010101 | ALUctr: 101
# Opcode: 0000 | Rs: 000($0) | Rt: 001($1) | Rd: 010($2) | Func: 101
# Operation: OR
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000001010101010101010
# Result : 00000000000000001010101010101010
# Time  :   46 | Clock: 1    | PC(int): 11 | Instruction: 0000010011100101 | ALUctr: 101
# Opcode: 0000 | Rs: 010($2) | Rt: 011($3) | Rd: 100($4) | Func: 101
# Operation: OR
# Value 1: 00000000000000001010101010101010
# Value 2: 00000000000000000000000011111111
# Result : 00000000000000001010101011111111
```

```
# Time  :   50 | Clock: 1    | PC(int): 12 | Instruction: 0001000101000111 | ALUctr: 001
# Opcode: 0001 | Rs: 000($0) | Rt: 101($5) | Immediate  : 000111
# Operation: ADDI
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000000111
# Result : 00000000000000000000000000000111
# Time  :   54 | Clock: 1    | PC(int): 13 | Instruction: 0001101110010101 | ALUctr: 001
# Opcode: 0001 | Rs: 101($5) | Rt: 110($6) | Immediate  : 010101
# Operation: ADDI
# Value 1: 00000000000000000000000000000111
# Value 2: 00000000000000000000000000010101
# Result : 00000000000000000000000000011100
# Time  :   58 | Clock: 1    | PC(int): 14 | Instruction: 0010000101000111 | ALUctr: 000
# Opcode: 0010 | Rs: 000($0) | Rt: 101($5) | Immediate  : 000111
# Operation: ANDI
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000000111
# Result : 00000000000000000000000000000000
# Time  :   62 | Clock: 1    | PC(int): 15 | Instruction: 0010101110010101 | ALUctr: 000
# Opcode: 0010 | Rs: 101($5) | Rt: 110($6) | Immediate  : 010101
# Operation: ANDI
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000010101
# Result : 00000000000000000000000000000000
# Time  :   66 | Clock: 1    | PC(int): 16 | Instruction: 0011000101000111 | ALUctr: 101
# Opcode: 0011 | Rs: 000($0) | Rt: 101($5) | Immediate  : 000111
# Operation: ORI
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000000111
# Result : 00000000000000000000000000000111
# Time  :   70 | Clock: 1    | PC(int): 17 | Instruction: 0011101110010101 | ALUctr: 101
# Opcode: 0011 | Rs: 101($5) | Rt: 110($6) | Immediate  : 010101
# Operation: ORI
# Value 1: 00000000000000000000000000000111
# Value 2: 00000000000000000000000000010101
# Result : 00000000000000000000000000010111
```

```
# Time  :   74 | Clock: 1    | PC(int): 18 | Instruction: 0100000101000111 | ALUctr: 100
# Opcode: 0100 | Rs: 000($0) | Rt: 101($5) | Immediate  : 000111
# Operation: NORI
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000000111
# Result : 11111111111111111111111111111000
# Time  :   78 | Clock: 1    | PC(int): 19 | Instruction: 0100101110010101 | ALUctr: 100
# Opcode: 0100 | Rs: 101($5) | Rt: 110($6) | Immediate  : 010101
# Operation: NORI
# Value 1: 11111111111111111111111111111000
# Value 2: 00000000000000000000000000010101
# Result : 00000000000000000000000000000010
# Time  :   82 | Clock: 1    | PC(int): 20 | Instruction: 0101111101000011 | ALUctr: 010
# Opcode: 0101 | Rs: 111($7) | Rt: 101($5) | Immediate  : 000011
# Operation: BEQ
# Value 1: 11111111111111111111111111111000
# Value 2: 11111111111111111111111111111000
# Result : 00000000000000000000000000000000
# Time  :   86 | Clock: 1    | PC(int): 24 | Instruction: 0111000101100111 | ALUctr: 110
# Opcode: 0111 | Rs: 000($0) | Rt: 101($5) | Immediate  : 100111
# Operation: SLTI
# Value 1: 00000000000000000000000000000000
# Value 2: 11111111111111111111111111100111
# Result : 00000000000000000000000000000000
# Time  :   90 | Clock: 1    | PC(int): 25 | Instruction: 0111101110010101 | ALUctr: 110
# Opcode: 0111 | Rs: 101($5) | Rt: 110($6) | Immediate  : 010101
# Operation: SLTI
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000010101
# Result : 11111111111111111111111111111111
```

```
# Time  :   94 | Clock: 1    | PC(int): 26 | Instruction: 1000000110000001 | ALUctr: 001
# Opcode: 1000 | Rs: 000($0) | Rt: 110($6) | Immediate  : 000001
# Operation: LW
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000000001
# Result : 00000000000000000000000000000001
# Time  :   98 | Clock: 1    | PC(int): 27 | Instruction: 1000000111000100 | ALUctr: 001
# Opcode: 1000 | Rs: 000($0) | Rt: 111($7) | Immediate  : 000100
# Operation: LW
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000000100
# Result : 00000000000000000000000000000100
# Time  :  102 | Clock: 1    | PC(int): 28 | Instruction: 1001000110010100 | ALUctr: 001
# Opcode: 1001 | Rs: 000($0) | Rt: 110($6) | Immediate  : 010100
# Operation: SW
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000010100
# Result : 00000000000000000000000000010100
# Time  :  106 | Clock: 1    | PC(int): 29 | Instruction: 1001000111011000 | ALUctr: 001
# Opcode: 1001 | Rs: 000($0) | Rt: 111($7) | Immediate  : 011000
# Operation: SW
# Value 1: 00000000000000000000000000000000
# Value 2: 00000000000000000000000000011000
# Result : 00000000000000000000000000011000
# ** Note: $finish    : C:/altera/13.1/workspace/hw4/MiniMIPS_tb.v(95)
#    Time: 110 ps  Iteration: 1  Instance: /MiniMIPS_tb
# 1
# Break in Module MiniMIPS_tb at C:/altera/13.1/workspace/hw4/MiniMIPS_tb.v line 95
```