# GEBZE TECHNICAL UNIVERSİTY

# COMPUTER ENGINEERING

## COMPUTER ORGANIZATION

## CSE 331 – 2019 FALL

## HW3 REPORT

*SILA BENGİSU YÜKSEKLİ*

*1801042877*

## ➢ *Explanation of Homework and Codes*

In this homework, I implemented a single cycle datapath that provides load byte, load byte unsigned, load halfword, load halfword unsigned, load upper immediate, load word, store byte, store halfword and store word. This implementation is made by using structural verilog.Only, in instruction memory, data memory and register file is made by using behavioral and dataflow verilog.To support these operations, 32-Bit 2x1 MUX Module, 32-bit full adder, buffer gate, sign and zero extend modules, control unit, register file, data memory, instruction memory and some other units called Change_16Bit, Change_8Bit and luiImmediate are used . Using of gate except and,or,not is forbidden. That's way I implemented my own xor and buffer gate.

Lui_Immediate module is used for only lui instruction. It provides shifting left and filling lower 16 bits with zero. Change_8Bit module is used for lb and lh instructions. It takes 8 bits and fills upper 24 bit with zeros. Change _16Bit is used for lh, lhu instructions. It takes 16 bits and fills upper 16 bit with zeros.

For address calculation, full adder is used, not ALU. Because single cycle datapath that I created, only supports load and store operations. So all of them will calculate some address by adding rs content and offset to each other. I use three xor gates, two and gates and one or gate in adder implementation. Also, xor gate has two not and two and gates, one or gate. So I used 18x32(For 32 bit adder), 576 gates in total.

In this homework, I decided to use seven control signals that are MemRead, MemWrite, Extend, Lui, RegWrite, BitNumSelection and lw. I made a truth table for all instructions and wrote logic equations for all instructions that is shown in below:

| ********** | lb | lbu | lh | lhu | lui | lw | sb | sh | Sw |
|---|---|---|---|---|---|---|---|---|---|
| RegWrite | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| MemRead | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Lui | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Extend | 0 | 1 | 0 | 1 | x | 1 | 1 | 1 | 1 |
| lw | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| BitNumSelection | 0 | 0 | 1 | 1 | x | 0 | x | x | x |

- *BitNumSelection = (op1 xor op0) . op3'*
- *Lui = op5'*
- *MemWrite = op5.op4'.op3*
- *MemRead = op3'*
- *RegWrite = op5' + op3'*
- *Exdend = op5.(op1 + op2 + op3)*
- *Lw = op5.op3'.op1*

## ➢ *Testbench*

I wrote testbenchs for all modules to understand all of them works correctly. But to keep it small, I will mention about some of them.

- ***Testbench for full adder ->*** *I display results on the screen in decimal format, because of simplicity.*

```
module _32Bit_Full_Adder_testbench();

  reg [31:0] A,B;
  reg Ci;
  wire [31:0] R;
  wire C0;
_32Bit_Adder adder (R,Co,A,B,Ci);

initial begin
        A = 4'd15;  B = 4'd15; Ci = 1'b0;
        #`DELAY;
        A = 4'd5;  B = 4'd15; Ci = 1'b0;
        #`DELAY;
        A = 4'd12;  B = 4'd1; Ci = 1'b0;
        #`DELAY;
        A = 4'd11;  B = 4'd10; Ci = 1'b0;
        #`DELAY;
end

initial begin
        $monitor("time =%2d, A =%2d, B =%2d, Ci =%2d,Co =%2d, Result=%2d",$time, A, B, Ci,Co,R);
end

endmodule
```

- ***Testbench for control unit ->*** *To understand if the control signals are generated correctly for all instructions, I wrote this testbench.*

```
module controlUnit_testbench();
wire memRead,memWrite,Extend,Lui,RegWrite,BitNumSelection,lw;
reg [5:0] opcode;
Control_Unit c(MemRead,MemWrite,Extend,Lui,RegWrite,BitNumSelection,lw,opcode);

initial begin
        opcode = 6'b100000; //lb
        #`DELAY;
        opcode = 6'b100100; //lbu
        #`DELAY;
        opcode = 6'b100001; //lh
        #`DELAY;
        opcode = 6'b100101; //lhu
        #`DELAY;
        opcode = 6'b001111; //lui
        #`DELAY;
        opcode = 6'b100011; //lw
        #`DELAY;
        opcode = 6'b101000; //sb
        #`DELAY;
        opcode = 6'b101001; //sh
        #`DELAY;
```

```
        opcode = 6'b101011; //sw
        #`DELAY
end

initial begin
        $monitor("time =%2d, MemRead = %1b, MemWrite= %1b, Extend = %1b, Lui= %1b, RegWrite = %1b,
BitNumSelection= %1b, lw= %1b",$time,MemRead,MemWrite,Extend,Lui,RegWrite,BitNumSelection,lw);
end
endmodule
```

- ***Testbench for Data Memory*** ***->*** *I both try read and write operation. To try write operation, you must assign memWrite to one and memRead to zero. You have to give clock also, At the positive edge of clock write operations are made for all units that has write operation, if the write signal is asserted.*

```
module dataMem_testbench();

wire [31:0] readData;
reg [31:0] address, writeData;
reg memWrite,memRead,clock;
reg [5:0] opcode;
Data_Memory mem(readData,address,writeData,memWrite,memRead,opcode,clock);

initial clock = 0;
always
        #50 clock = ~clock;
initial begin
        #2000 $finish;
end

initial begin

writeData=32'b01111111111111111111111111111111; address=32'b00000000000000000000000000000000;
opcode = 6'b100000; memWrite=1'b0; memRead=1'b1;
#`DELAY;
writeData=32'b01111111111111111111111111111111; address=32'b00000000000000000000000000000001;
opcode = 6'b100000; memWrite=1'b0; memRead=1'b1;
#`DELAY;
writeData=32'b01111111111111111111111111111111; address=32'b00000000000000000000000000000010;
opcode = 6'b100000; memWrite=1'b0; memRead=1'b1;
#`DELAY;
writeData=32'b01111111111111111111111111111111; address=32'b00000000000000000000000000000011;
opcode = 6'b100000; memWrite=1'b0; memRead=1'b1;
#`DELAY;

end

initial begin
        $monitor("readData = %32b",readData);
end

endmodule
```

- ***Testbench for Instruction Memory ->*** *Instruction memory should fetch next instruction by increasing pc counter after reads the current instruction. To understand fetching operation works fine, I display instructions that are read from file, on the screen. At the end of the line, It always prints something like xxx……xxxx. Just ignore it. I didn't understand why it does it, but it does not about the instructions that I read, it reads all eighteen instructions correctly.*

```
module instructionMem_testbench();

reg clock;
wire [31:0] curIns;
Instruction_Memory insMem(curIns,clock);

initial clock = 0;
always
	#50 clock = ~clock;
initial begin
	#2000 $finish;
end

initial begin
	$monitor("instr = %32b",curIns);
end

endmodule
```

- ***Testbench for mips unit ->*** *mips32 unit set as top level entity and this the testbench for it. It takes only one parameter which is clock. This clock will be used in register, data memory and instruction memory units that are inside mips 32 unit. Updating the files, after write operation is done in this testbench.*

```
module mips32_testbench();

reg clock;
mips32 singleCycleMips(clock);

initial clock = 0;
always
	#50 clock = ~clock;
initial begin
	#3200 $finish;
end

always @(*) begin
	$writememb("register.txt",singleCycleMips.register.registers);
	$writememb("data.txt", singleCycleMips.dataMem.data_memory);
end
```

## ➢ *Results for Testbenchs*

In order to avoid tangling, first I try all instructions one by one. I checked the register, data memory, instruction files to understand results are true.

After I control for every operation, I fill instruction memory with 18 instructions which is shown below:

```
10000000000000010000000000000001    // lb
10010000000000010000000000000010    // lbu
10000100000000110000000000000011    // lh
10010100000001000000000000000100    // lhu
00111100000001010000000000000101    // lui
10001100000001100000000000000110    // lw
10100000000001110000000000000111    // sb
10100100000010000000000000001000    // sh
10101100000010010000000000001001    // sw
10000000000010100000000000001010    // lb
10010000000010110000000000001011    // lbu
10000100000011000000000000001100    // lh
10010100000011010000000000001101    // lhu
00111100000011100000000000001110    // lui
10001100000011110000000000001111    // lw
10100000000100000000000000010000    // sb
10100100000100010000000000010001    // sh
10101100000100100000000000010010    // sw
```

Then I try my single cycle datapath for all eighteen instructions , and display some variables on the screen which are current instruction, control signals, rs content, rt content, addres result and the data that will be written to register, but if the instruction is store, then the last variable called writeData, is shown like xx……xxx, they don't write in register.

```
initial  begin
        $monitor("instruction=%32b, MemRead=%1b, MemWrite=%1b, Extend=%1b, Lui=%1b, RegWrite=%1b,
        BitNumSelection=%1b, lw=%1b, rsData=%32b, rtData=%32b, addres=%32b,
        writeData=%32b",currentInstruction,MemRead,MemWrite,Extend,Lui,RegWrite,BitNumSelection,lw,readDat
        a1,readData2,adderOut,muxOut4);
end
```

```
VSIM 4> step -current
# instruction=10000000000000110000000000000011, MemRead=1, MemWrite=0, Extend=0, Lui=0, RegWrite=1, BitNumSelection=0, lw=0, rsData=00000000000000000000000000001000, rtData=00000
000000000000000000000000011, addres=00000000000000000000000000001011, muxout=00000000000000000000000000001011
```

***Result for lb***

```
VSIM 4> step -current
# instruction=10010000000001000000000000000011, MemRead=1, MemWrite=0, Extend=1, Lui=0, RegWrite=1, BitNumSelection=0, lw=0, rsData=00000000000000000000000000001000, rtData=00000
000000000000000000000000010, addres=00000000000000000000000000001011, muxout=00000000000000000000000000001011
```

***Result for lbu***

```
VSIM 7> step -current
# instruction=10000100000000000000000000000011, MemRead=1, MemWrite=0, Extend=0, Lui=0, RegWrite=1, BitNumSelection=1, lw=0, rsData=00000000000000000000000000001000, rtData=00000
000000000000000000000001000, addres=00000000000000000000000000001011, muxout=00000000000000000000000000001011
```

***Result for lh***

```
VSIM 10> step -current
# instruction=10010100000001000000000000000011, MemRead=1, MemWrite=0, Extend=1, Lui=0, RegWrite=1, BitNumSelection=1, lw=0, rsData=00000000000000000000000000001011, rtData=00000
000000000000000000000000100, addres=00000000000000000000000000001110, muxout=00000000000000000000000000001110
```

***Result for lhu***

```
VSIM 13> step -current
# instruction=00111100000001010000000000000011, MemRead=0, MemWrite=0, Extend=0, Lui=1, RegWrite=1, BitNumSelection=0, lw=0, rsData=00000000000000000000000000001011, rtData=00000
000000000000000000000000101, addres=00000000000000000000000000001110, muxout=00000000000000110000000000000000
```

***Result for lui***

```
VSIM 4> step -current
# instruction=10001100000000010000000000000011, MemRead=1, MemWrite=0, Extend=1, Lui=0, RegWrite=1, BitNumSelection=0, lw=1, rsData=00000000000000000000000000001000, rtData=01111
111111111111111111111111, addres=00000000000000000000000000001011, muxout=00000000000000000000000000001011
```

***Result for lw***

```
VSIM 10> step -current
# instruction=10100000000001000000000000000011, MemRead=0, MemWrite=1, Extend=1, Lui=0, RegWrite=0, BitNumSelection=0, lw=0, rsData=00000000000000000000000000001011, rtData=00000
000000000000000000000001110, addres=00000000000000000000000000001110, muxout=00000000000000000000000xxxxxxxx
```

***Result for sb***

```
VSIM 13> step -current
# instruction=10100100000001000000000000000011, MemRead=0, MemWrite=1, Extend=1, Lui=0, RegWrite=0, BitNumSelection=0, lw=0, rsData=00000000000000000000000000001011, rtData=00000
000000000000000000000001110, addres=00000000000000000000000000001110, muxout=00000000000000000000000xxxxxxxx
```

***Result for sh***

```
VSIM 4> step -current
# instruction=10101100000001010000000000000011, MemRead=0, MemWrite=1, Extend=1, Lui=0, RegWrite=0, BitNumSelection=0, lw=0, rsData=00000000000000000000000000001011, rtData=00000
000000001100000000000000000, addres=00000000000000000000000000001110, muxout=00000000000000000000000xxxxxxxx
```

***Result for sw***


***Eighteen***

***instruction in***

***one***