32 bits MIPS RISC Processor

Herless Alvarado
Computer Science 4th semester
UTEC
Lima, Peru
herless.alvarado@utec.edu.pe
https://github.com/herlessjap/32-bits-MIPS-RISC-Processor

***Abstract*—The project for the course of Computer Architecture at UTEC is to develop a 32 bits MIPS RISC Processor with 24 instructions among R-type, I-type and J-type. All of these were develop and exceuted in three different testbenches and the results are shown in the paper.**

***Index Terms*—Computer Architecture, mips, verilog, processor, 32bits**

I. Introduction

Early in the 1985 MIPS was introduced which stands for Microprocessor without Interlocked Pipelined Stages. MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA). The final project for the course CS2201 - Computer Architecture is to develop a 32 Bits Pathline RISC MIPS in verilog which supports r-type, i-type and j-type instructions. The instructions are the following:

add	addi	lb	beq
sub	subi	lh	bneq
and	andi	lw	bgez
nor	ori	sb	j
or	slti	sh	jr
slt		sw	jal
		lui	

The datapath for this project is image shown in Figure 1. To see the datapath in full image you can check the README.md in the github source code.

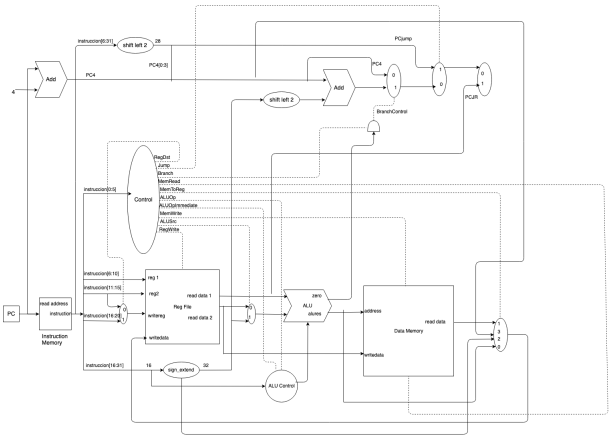

Fig. 1. MIPS Datapath

The size of the Instruction Memory and Data Memory are of 256 bytes, both memories are byte-addressable. Furthermore, the endianness of the datapath is big-endian.

II. Methodology

This paper aims to explain the development, analysis and results of a 32 bits MIPS datapath though MIPS is a very well known instruction set architecture in which most of the gathered information was quantitative found on the internet. The base knowledge to build this project is being familiarized with MIPS assembly instructions and also know how a datapath works and its elements work together, all of this information is on the book *"Computer Organization and Design by Patterson and Hennessy"*

III. Experimental Setup

The base to begin the project was the knowledge from the Computer Organization and Design and some verilog tutorials from the web.

R-type

The R-type instructions were done first. The modules for this were created as follow, Program Counter, Instruction Memory, Register File and ALU. R-type instructions were not that complex and done in a very short time after of the first gathered of knowledge.

For the R instructions all have the same control Operation which is shown in Figure 2.

```
case (Opcode)
    6'b000000: begin //r-instructions and jr
    RegDst = 1'b0;
    Jump = 1'b0;
    Branch = 1'b0;
    MemRead = 3'b000;
    MemToReg = 2'b00;
    ALUOp = 2'b10;
    ALUOpImmmediate = 3'b000;
    MemWrite = 2'b00;
    ALUSrc = 1'b0;
    RegWrite = 2'b01;
    end
```

Fig. 2. R-type-OpCode

The operation is calculated depending on the ALUOp and func which is shown in Figure 3.

```
always @(ALUOp & func)
begin
    if (ALUOp == 2'b10 & func == 6'b100000) //add
        ALUControl = 4'b0010;
    if (ALUOp == 2'b10 & func == 6'b100010) //sub
        ALUControl = 4'b0110;
    if (ALUOp == 2'b10 & func == 6'b100100) //and
        ALUControl = 4'b0000;
    if (ALUOp == 2'b10 & func == 6'b100111) //nor
        ALUControl = 4'b1100;
    if (ALUOp == 2'b10 & func == 6'b100101) //or
        ALUControl = 4'b0001;
    if (ALUOp == 2'b10 & func == 6'b101010) //slt
        ALUControl = 4'b0111;
end
```

Fig. 3. R-type-Operation

I-TYPE

The next instructions done were the first column of I-type shown in Introduction. As in the R-type instructions was missing the Control Unit it was a need for this instruction. The I-type instructions created the following modules, Sign Extend, Control, ALUControl, Multiplexer InsReg and Multiplexer RegALU. The I-type instruction were difficult and took some time to finish.

The main difference of Immediate instructions is that they use a sign_extend to extend 16 bits to 32 bits shown in Figure 4.

```
module sign_extend(sign_in,sign_out);

input [0:15] sign_in;
output [0:31] sign_out;

assign sign_out[16:31] = sign_in[0:15];
assign sign_out[0:15] = 1'b0;

endmodule
```

Fig. 4. I-type-SignExtend

Also to decide which instruction to execute depends on the ALUOp and ALUOpImmediate shown in Figure 5.

J-TYPE

Lastly the J-type instructions created the modules And, Multiplexer ControlPC, ImmediateShiftLeft,Multiplexer PC-ShiftAdd, PCAddShift, PCShiftLeft,Multiplexer Rad1PC and ShiftPlusPC.

This was the most complicated one because of the amount of modules created. The most important are the two shift lefts

```
always @(ALUOp & ALUOpImmmediate)
begin
    if (ALUOp == 2'b11 & ALUOpImmmediate == 3'b001) //addi
        ALUControl = 4'b0010;
    if (ALUOp == 2'b11 & ALUOpImmmediate == 3'b010) //subi
        ALUControl = 4'b0110;
    if (ALUOp == 2'b11 & ALUOpImmmediate == 3'b011) //andi
        ALUControl = 4'b0000;
    if (ALUOp == 2'b11 & ALUOpImmmediate == 3'b100) //ori
        ALUControl = 4'b0001;
    if (ALUOp == 2'b11 & ALUOpImmmediate == 3'b101) //slti
        ALUControl = 4'b0111;
end
```

Fig. 5. I-type-Operation

```
module immediate_shift_left(in,out);

input [0:31] in;
output [0:31] out;

assign out = {in[2:31], 1'b0, 1'b0};

endmodule
```

Fig. 6. Immediate ShiftLeft

used one for 26 bits code and another for immediate 16 bits numbers shown in Figure 6 and 7.

Another important module is the sum between pc+4 and Immediate shift left, this is shown in Figure 8.

Last but not least the is an and operator which compares branch signal with zero signal, this is shown in Figure 9.

This were most of the modules used for the j-type instructions, there are also some multiplexers but are not going to be shown.

IV. EVALUATION

There are 3 testbenches for this datapath. The first testbench contains the first and second columns of the instructions shown in the Introduction table. The second testbench contains the third column and lastly the third testbench contains the fourth column of instructions.

The first testbench executes the following instructions:

- add t1, t2, t3
- sub t1, t2, t3
- and t1, t2, t3
- nor t1, t2, t3
- or t1, t2, t3
- slt t1, t2, t3
- addi t1, t2, 1
- subi t1, t2, 1
- andi t1, t2, 1
- ori t1, t2, 1
- slti t1, t2, 1

The first testbench modules are shown in Fig 10.

```
module pc_shift_left(in,out);

input [0:25] in;
output [0:27] out;

assign out = {in[0:25], 1'b0, 1'b0};

endmodule
```

Fig. 7. 26 bits ShiftLeft

```
module shift_plus_pc(shiftin,pcin,jaddress);

input [0:27] shiftin;
input [0:31] pcin;
output reg [0:31] jaddress;

always @(*)begin
    jaddress[0:3] <= pcin[0:3];
    jaddress[4:31] <= shiftin;
end

endmodule
```

Fig. 8. PC+4 add ImmediateShift

The first testbench is compiled with the following command *iverilog ri_instructions_tb.v alu_control.v alu.v control.v data_memory.v ins_mem.v ins_mux_reg.v pc.v reg_file.v reg_mux_alu.v sign_extend.v data_mux_reg.v* and the output is in Fig 11.

The second testbench executes the following instructions:

- lw t1, 1(s1)
- lh t2, 1(s2)
- lb t5, 1(s5)
- sw t7, 0(s1)
- sh t7, 6(s1)
- sb t7, 9(s1)
- lui s0, ffff

The second testbench modules are shown in Fig 12.

The second testbench is compiled with the following command *iverilog ls_instructions_tb.v alu_control.v alu.v control.v data_memory.v ins_mem.v ins_mux_reg.v pc.v reg_file.v reg_mux_alu.v sign_extend.v data_mux_reg.v* and the output is in Fig 13.

The third testbench executes the following instructions:

- jal 1
- j 11
- jr t0
- beq t0, t1, 1
- bne t0, t1, 1
- bgez t1, 1

The third testbench modules are shown in Fig 14.

```
module andm(in1,in2,out);

input in1,in2;
output out;

assign out = in1&in2;

endmodule
```

Fig. 9. Module AND

```
pc pc1(clk,reset,out);
InstructionMemory in1(clk,out,instruction);
control con1(Opcode,RegDst,Jump,Branch,MemRead,MemToReg,ALUOp,ALUOpImmediate,MemWrite,ALUSrc,RegWrite);
ins_mux_reg mux1(rad2,writeadd,RegDst,writereg);
sign_extend sign1(sign_in,sign_out);
alu_control aluc1(ALUOp,ALUOpImmediate,func,ALUControl);
reg_file reg1(clk,rad1,rad2,writereg,writedata,RegWrite,readdata1,readdata2);
reg_mux_alu mux2(readdata2,sign_out,ALUSrc,resmux);
alu alu1(clk,readdata1,resmux,ALUControl,alures,zero);
data_memory data1(clk,alures,readdata2,MemRead,MemWrite,data);
data_mux_reg mux3(data,alures,sign_out,pcaddress,MemToReg,writedata);
```

Fig. 10. First testbench modules

```
R-type 00000000000000000000000000000100 + 00000000000000000000000000000011 = 00000000000000000000000000000111, time=                21
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
R-type 00000000000000000000000000000100 + 00000000000000000000000000000011 = 00000000000000000000000000000111, time=                42
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
R-type 00000000000000000000000000000100 - 00000000000000000000000000000011 = 00000000000000000000000000000001, time=                63
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
R-type 00000000000000000000000000000100 & 00000000000000000000000000000011 = 00000000000000000000000000000000, time=                84
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
R-type ~(00000000000000000000000000000100 | 00000000000000000000000000000011) = 11111111111111111111111111111000, time=               105
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
R-type 00000000000000000000000000000100 | 00000000000000000000000000000011 = 00000000000000000000000000000111, time=               126
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
R-type 00000000000000000000000000000100 < 00000000000000000000000000000011 = 00000000000000000000000000000000, time=               147
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
I-type 00000000000000000000000000000100 - 00000000000000000000000000000001 = 00000000000000000000000000000011, time=               160
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
I-type 00000000000000000000000000000100 + 00000000000000000000000000000001 = 00000000000000000000000000000101, time=               180
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
I-type 00000000000000000000000000000100 & 00000000000000000000000000000001 = 00000000000000000000000000000000, time=               200
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
I-type 00000000000000000000000000000100 | 00000000000000000000000000000001 = 00000000000000000000000000000101, time=               220
WARNING: reg_file.v:54: $writememb: Standard inconsistency, following 1364-2005.
I-type 00000000000000000000000000000100 < 00000000000000000000000000000001 = 00000000000000000000000000000000, time=               240
```

Fig. 11. First testbench output

The third tesbench is compiled with the following command *iverilog j_instructions_tb.v alu_control.v alu.v and.v control_mux_pc.v control.v data_memory.v data_mux_reg.v immediate_shift_left.v ins_mem.v ins_mux_reg.v pc_add_shift2.v pc_shift_left.v pc.v pc4_mux_sl2add.v rad1_mux_pc.v reg_file.v reg_mux_alu.v shift_plus_pc.v sign_extend.v* and the output is in Fig 15.

V. Conclusions

MIPS is an understandable Instruction Set Architecture which can easily learned from the Computer Organization and Design by Patterson and Hennessy book. Furthermore, verilog is also an easy HDL to learn specially if you are familiarized with C-language. The datapath was mostly challenging in the immediate instructions in the way of getting to understand how the instructions divides and goes through the datapath.

The datapath itself is challenging because of the knowledge required to know what each module does. The jump instructions were a little difficult because of the big amount of multiplexers and modules to each one.

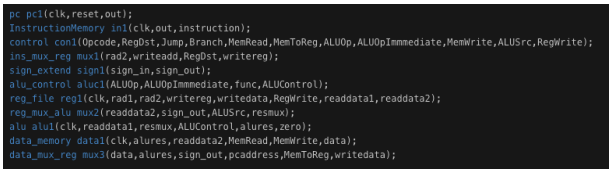

Fig. 12. Second testbench modules

Fig. 13. Second testbench output

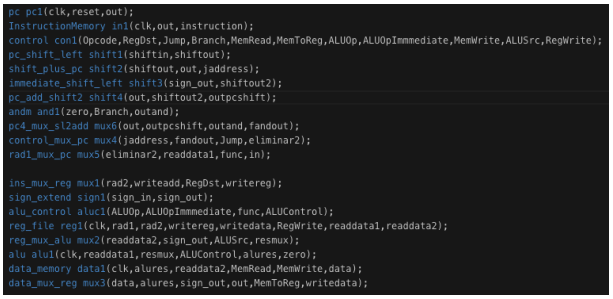

Fig. 14. Third testbench modules

The final conclusions is that this project can be done in two weeks with the previous knowledge of MIPS and verilog.

VI. Comments

The first difficult I found is that most of the implementations of the datapath were little endian and it was given to make it big endian. Another difficult is that there is no much information about some instructions and how they go through the datapath because when you dont know you have to manage to get to the correct result. Last difficult found is that some instructions needed more wires to get to the right answer and you have to create new wires and even extend the bits of some wire to get the right answer. I would really suggest to have some labs in class to teach how to begin with the datapath and the teacher going along with the students

References

[1] D. Patterson and J. Hennessy, “Computer Organization and Design,” 4th ed., San Mateo, CA:Morgan Kaufman, 2006.

[2] MIPS, “MIPS32 Architecture,” Accessed on: June. 15, 2019. [Online]. Available: https://www.mips.com/products/architectures/mips32-2/

[3] University of Minnesota Duluth, “MIPS R-Type Instruction Coding,” Accessed on: June. 16, 2019. [Online]. Available: https://www.d.umn.edu/ gshute/mips/rtype.xhtml

[4] University of Minnesota Duluth, “MIPS I-Type Instruction Coding,” Accessed on: June. 20, 2019. [Online]. Available: https://www.d.umn.edu/ gshute/mips/itype.xhtml

[5] University of California, “MIPS Reference Sheet,” Accessed on: June. 20, 2019. [Online]. Available: https://www.d.umn.edu/ gshute/mips/itype.xhtml

[6] University of Idaho, “MIPS Instruction Reference,” Accessed on: June. 20, 2019. [Online]. Available: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

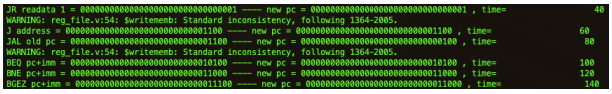

Fig. 15. Third testbench output

[7] University of Cambridge, “MIPS-Datapath,” Accessed on: June. 24, 2019. [Online]. Available: http://mi.eng.cam.ac.uk/ ahg/MIPS-Datapath/