Pathline of 32-RISC ISA

Martin Carrasco
Computer Science
UTEC
Lima, Peru
martin.carrasco@utec.edu.pe

Andrea Diaz
Computer Science
UTEC
Lima, Peru
andrea.diaz@utec.edu.pe

***Abstract*—This document describes the procedure of creating a 32 bits MIPS processor in Verilog, the program is designed to execute basic algorithms with a pre-defined short set of instructions. We will describe the methods we used to create it, the input and outputs of every module and a brief description of the individual function of each of them.**

***Index Terms*—MIPS processor, Verilog, ISA, Datapath, RISC**

I. Introduction

Everything in the processor starts with the decomposition of high-level languages to something more understandable to a computer, this happens with the help of the instruction set architecture (ISA). The ISA act as a main schematic for how the machine will work. To improve the process of translation to machine language was first introduced the reduced instruction set computer (RISC) technology. Enlisting the most used operations and minimizing them in a way that simple instructions can be combined to form complex statements. This architecture can run three times faster than others because of this. language. The assembler-compiler ratio is the execution time for an assembler program divided by the exe? cution time for a compiled version of the program. This ratio is less than 50 percent for CISCs. For the RISC II, however, it is 90 percent. [1] Nowadays, most universities and research labs use MIPS, one of the first RISC architectures, as their favorite; the simplicity of it makes it easier to manage the size of the RISC code. With simplicity

A. MIPS - instructions

Every computer executes a pre-established set of instructions. These instructions are organized into five different types. The three main ones that we will use in this paper are R-type[Table I] performs simple arithmetic and logic operations, the I-type[Table II], immediate values and a register values; and the J-type[Table III], jumps. MIPS has a vast number of instructions, but for the purpose of simplification, we will only use the following in the Table IV.

TABLE I
R-TYPE FORMAT

R-type fields					
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

TABLE II
I-TYPE FORMAT

I-type fields			
op	rs	rt	inmediate
6 bits	5 bits	5 bits	16 bits

TABLE III
J-TYPE FORMAT

J-type	
op	offset

B. Datapath

The MIPS implementation[Fig. 1] goes into a graphic model called datapath, every component works in a different way, between decoding the instruction and passing it to the next one, the processor can have a big amount of tasks that influence on the final performance.

II. Methodology

For the construction and designing of the processor we used Verilog as our hardware description language, Icarus Verilog as simulation and synthesis tool, and Scansion as our main simulator with the alternated use of GTKWave and ModelSim.

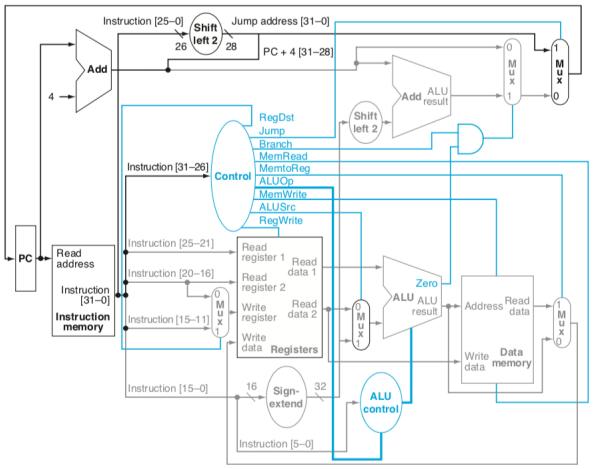
A. The coding

We started with the simple modules (mux, shift left, sign extend, the adders and the AND used for the branch) then we continued to the more complex sections. All the modules were first made without linking and the variables wrote to resemble the input and outputs of the datapath show on the Figure 1, but after more implementations some of them alternated.

TABLE IV
INSTRUCTIONS SUPPORTED BY OUR MIPS

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

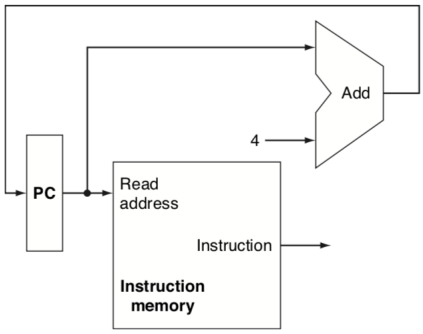
Fig. 1. Control and datapath that handles the jump instruction



B. Principal Modules

The biggest code was the control, but the hardest procedure was the code for the implementation of the datapath file that we named Datapath.v; this is in charge of delivering the wires for each module, setting up the clock of the processor and linking every variable that needs to be send between units.

Fig. 2. Fetching instruction. The Program counter, Instruction Memory and a Add 4



III. Experimental Setup

A. Program Counter

Every instruction is 32 bits, those go to the instruction memory and to the 4 adder via wires[Fig. 2]. The program counter is in charge of handing the previous and next instruction, including delivering the branch when the jump is necessary. The program counter equation when jump is used

$$PC = Register + BranchAddress$$

In our first Verilog program of the PC received a clock, the address and the add 4 result.

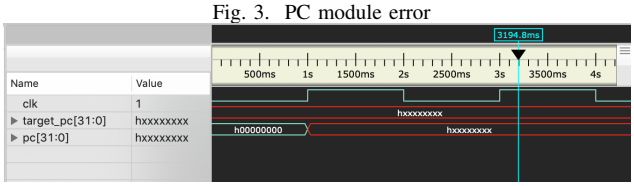
```
module PC(clk, pc, target_pc);
    input clk;
    input [31:0] target_pc;
    output reg [31:0] pc;

    initial begin
        pc = 32'b0;
    end

    always @(posedge clk) begin
        pc = target_pc;
    end
endmodule
```

One of the main struggles we had was the *always* instruction, initially we thought that writing *always @ (posedge clk)* would make the pc change continuously as the clk cycles were running but that's not what happened at the time of the simulation, the pc was only changing at the first clk posedge but modifying the next values to undefined; as the code continue to simulate with this pc value the some of the variables would change to undefined as well.

Fig. 3. PC module error



The temporary solution for this was the *$display* command of the *target_pc* (the variable that was allegedly changing the pc) but now the issues was that *$display* only showed the instant values without waiting for changes to be applied, that quite confused us for a while, after some research and we continued to this command but only to proved the right functionality of the module not anymore to test variable values.

PC module fixed

```
module PC (clk, pc);
    input clk;
    output reg [31:0] pc;

    initial begin
        pc = 32'b0;
    end

    always @(posedge clk) begin
        pc = pc + 32'h4;
    end
endmodule
```

We fixed this error by replacing the *target_pc* by just adding 4 to our pc variable, as a result we removed the *target_pc* element as well.

B. Clock

Originally the clock was introduced and defined at our main controller, previously mentioned as DataPath.v, and delivered to every module that need it, the predicament with this way of declaration it was that some units were having their own clocks

TABLE V
INPUTS AND OUTPUTS OF THE FIRST PC

inputs	
clk	1 bit
target_pc	32 bits
outputs	
pc	32 bits

running in their own pace. The best fix for this inconsistency was to create a new file that is in charge of all the changes in the wave; it had to be changed the inputs and outputs of the modules that used the clk

This is the code for an processor that never stops, for the purpose of only demonstration the clock will run for 50 clicks.

```
module Clock(clk);
    output reg clk;

    initial begin
        clk = 0;
    end

    always begin
        #5 clk <= ~clk;
    end
endmodule
```

Code with 50 click, changing the clock every 5.

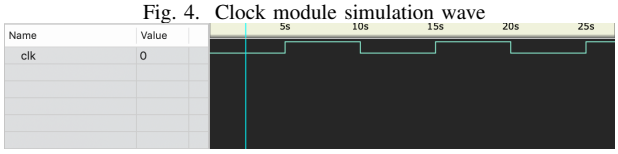
```
module Clock(clk);
    output reg clk;

    initial begin
        clk = 0;
    end

    initial #50 $finish();

    always begin
        #5 clk = ~clk;
    end
endmodule
```

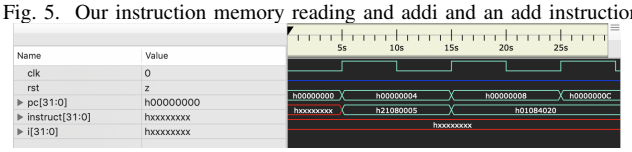
Fig. 4. Clock module simulation wave



C. Instruction Memory

In this module the binary code is divided, the bits from 31 to 26 go to the control and depending of the instruction the rest if bits will be distribute to the register or the sign extend.[Fig. 5]

Fig. 5. Our instruction memory reading and addi and an add instruction



D. The Data Memory

This module represents how we save data to a bigger memory register than the usual register. That is memory we don't use as often as the register but where we need to store larger amounts of data. We can access it in a few different ways. We can either load or save. The sizes we use are word (4bytes), half word(2bytes) or a byte. This works by passing it a a reference to the start of the the array that is memory and and offset for the amount of data.

E. The Registers

Here is where all the quasi temporal data is stored and handled. Arguably one of the most important units, here is where every instruction makes a stop to collect or store data. The register is made up of 32 addresses with 1 byte long. Here we store our temporal values, the saved register for the jal instruction, the values we read from memory and everything else we want to operate on to.

Each set of registers here serves a function some save space for a constant zero ($0) temporal values ($t), permanent values ($s) and some other valuable data we will talk about later on. [Fig. 6]

Fig. 6. The register in a simulation of an addi instruction next to a jump one

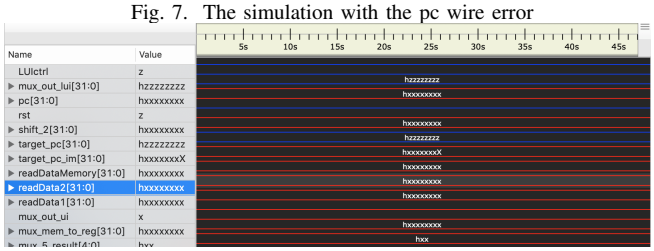
F. The J-type instructions

One of our biggest problems was the J-type instructions. When creating the last mux (the one that changed the PC to the address we were jumping) where the output was our *pc wire*, the same one we were passing to the PCModule and, as you can see in the PC module fixed code, by giving the pc wire with a changed value to this module it damaged the rest of the modules that depended of the PC module output [Fig. 7].

To repair this crucial mistake we had to pass three auxiliary variables, again *pc_target* and the *Jump* and *Branch* signal form the Control editing the code like this

```
module PC ( clk, pc, target_pc,
            Jump, Branch);
    input clk;
    input [31:0] target_pc;
    input Jump, Branch;
```

Fig. 7. The simulation with the pc wire error




```
    output reg [31:0] pc;

    initial  begin
             pc = 32'b0;
    end

    always @(posedge clk) begin
             pc = pc + 32'h4;
             if (Jump || Branch)
                    pc = target_pc;
    end
endmodule
```

As this was mended everything worked just fine, we only needed to check if every instruction was working fine.

IV. Evaluation

After creating each module we continued to the testing of every element individually, for this part we used ModelSim as the tester, this helped us to correct some syntax errors and other minor issues. The formulation of the test benches was after we had applied the necessary adjustments, simple values where logged in to the inputs to verify the veracity of the operation conducted by the component.Even with this tool some of the simple files couldn't be tested so along the $display command we confirmed the results.

The next step was to pass the simulation, here everything got a lot harder, as the final phase of the evaluation things were suppose to work perfectly but the code was far from it; even if it did not have syntax errors and compiled fine, the processor did not carry out the task, initializing values to variables or losing data along the path.

In the *Experimental Setup* there is a more detailed explanation of our principal mistakes in the process of creation.

V. Conclusions

Being students of Computer Science and adding to the equation that is our second year, we realize with this project the relevance of knowing what's really behind the compiler, how each instruction is executed and how much it cost to our computers to perform it by means of choosing instructions that now we know are the best for particular cases.

For us the concept of programming wires was hard to grasp. However, once we understood it, we were able to cleanup our design and make it work. It was imperative to shift our usual thinking process for this new programming paradigm. In many instances we had to reuse wires and pass the same wires but with different values to modules which in regular programming would be nonsense.

The connection between the hardware and software was difficult to achieve, to the point where we needed to re-think simple processes and add new ones that we used to ignore, nevertheless, it's important to point out that this project was difficult to execute.

VI. Comments

We started to code the modules and compiling them separately without taking into consideration the other parts of the processor. Ergo, we underestimated the difficulty of the project.

We found it rewarding to learn new ways to approach at the hardware components using low level machine language. Also, we learned what was behind the system we were accustomed to use and why they worked the way they did.

References

[1] P. J. Denning, "The science of computing: Risc architecture," *American Scientist*, vol. 81, no. 1, pp. 7–10, 1993.

[2] S. Mukherjee, "If anything in this life is certain, it's that you can kill any isa," *Computer*, vol. 44, pp. 87–88, 11 2011.

[3] X. Chen and Z. Yu, "A flexible and energy-efficient convolutional neural network acceleration with dedicated isa and accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, pp. 1408–1412, July 2018.

[4] D. Kumar and K. Singh, "Design of high performance mips-32 pipeline processor," 04 2012.

[2] [1] [3] [4]