Pre-silicon Verification Spring 2022

MIPS Architecture

Team 2

Bruno Hernandez Lopez
José Alberto Gómes Diaz
Josue isaias Gomez Cosme
Christian Aaron Ortega Blanco

June 13, 2022

Content

Abstract

This paper contains an explanation of the principles and techniques used in the implementation of a single-cycle MIPS processor, with a very abstract and simplified overview. A data path and a simple version of a processor sufficient to implement a MIPS instruction set is constructed.

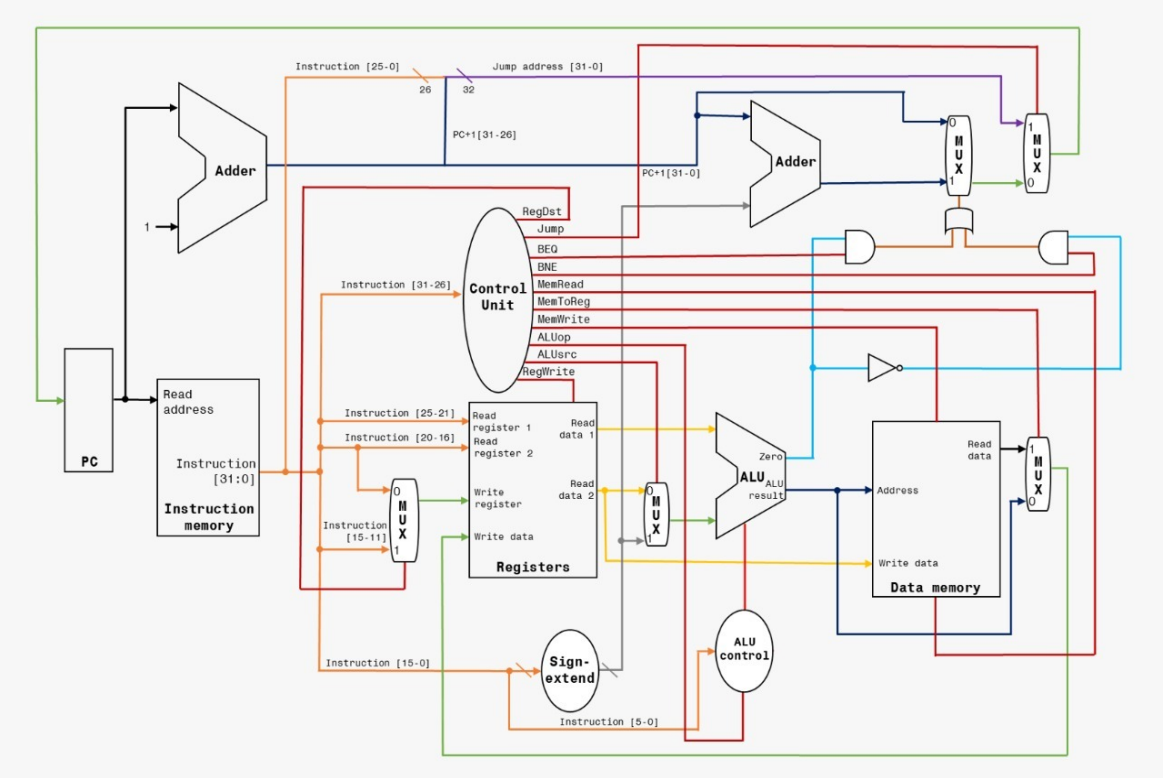
For the structure, all components were made separately and tested for serviceability. After having the components we made the union of the same with the help of a block diagram that allowed us to orient each point of union between the components. Finally, instruction-by-instruction tests were carried out and then tested completely with a program created in C++ language and converted into assembly and machine language.

Introduction

The use of a single-cycle processor refers to how it executes the instructions specified by the program and coordinates the activities of the other units of the computer.

A unicycle data path is characterized in that the execution of each instruction lasts for one clock cycle. Since the clock cycle must be adapted to the needs of the execution of all instructions and needs to adjust its duration to that of whichever instruction is longer. But the time required for execution can vary appreciably from one instruction to another. So a jump takes much less time than a load.

Description of the design



Single-Cycle Implementation

For our single-cycle implementation, we use a data memory,a file register, an instruction memory, an ALU, some adders, and five multiplexers. MIPS is based on 32 bits, so most of the buses are 32-bits wide.

The control unit is encouraged to define the datapath that the processor will execute, based on the opcode of the instruction that's currently being executed. Our processor has eleven control signals that define the datapath. The control signals can be generated by a combinational circuit with the instruction's 32-bit binary encoding as input.

In our implementation of the single-cycle structure, we will examine an implementation that includes a subset of the basic MIPS instruction set:

- The memory-reference instructions load word (lw) and store word (sw)
- The arithmetic-logical instructions add, sub, and, or, and slt
- The instructions branch equal (beq), branch not equal (bne) and jump (j), which we add last.

This list does not include all integer instructions (e.g., shift, multiply and divide are missing), nor does it include floating-point instructions.

However, they meet the elements requested for this project, where their main function is to illustrate the key principles used in the creation of a data path and control design. The implementation of the remaining instructions is similar.

HDL implementation

Design

The top file has all the physical connections required just as shown on the previous block diagram, it only needs a clock and reset signals to make the other modules start operating.

```
module Mips_machine (
      input clk,    // Clock
      input rst  // Asynchronous reset active low
);
wire [31:0] PC_In,PC_Out,Adder1_out,Adder2_out,mux1_out,InstMem_Out; //pc+4
////control
wire RegDst,Jump,BeQ,BnE,MemRead,MemToReg,MemWrite,ALUsrc,RegWrite;
wire[1:0] ALUop;
///// file register
wire [4:0] mux3_out;
wire [31 : 0]Read_Data_1,Read_Data_2;
///////sign exten
wire [31:0] signext_out;
/////// ALU control
wire [3:0]ALUci;
////// ALU
wire [31:0] mux4_out,ALU_out;
wire zero;
////// DataMem Out
wire [31:0] Ram_Out,mux5_out;

PC pc(PC_In,clk,rst,PC_Out);
Adder Adder_1(PC_Out,32'd1,Adder1_out);
Adder Adder_2(Adder1_out, signext_out ,Adder2_out);

mux2_1 mux1(mux1_out,(zero&&BeQ)|(~zero&&BnE),Adder2_out,Adder1_out);
mux2_1 mux2(PC_In,Jump,{Adder1_out[31:26],InstMem_Out[25:0]},mux1_out);

InstMem InstMem(PC_Out,InstMem_Out);
Control
Control(InstMem_Out[31:26],RegDst,Jump,BeQ,BnE,MemRead,MemToReg,MemWrite,ALUop,
ALUsrc,RegWrite);

mux5b_2to1 mux3(mux3_out,RegDst,InstMem_Out[15:11],InstMem_Out[20:16]);
```

```
  RegFile
registers(clk,RegWrite,mux3_out,mux5_out,InstMem_Out[25:21],Read_Data_1,InstMem
_Out[20:16],Read_Data_2);

SignExt SignExt(InstMem_Out[15:0],signext_out);
ALU_control ALUcontrol(InstMem_Out[5:0],ALUop,ALUci);

mux2_1 mux4(mux4_out,ALUsrc,signext_out,Read_Data_2);
ALU ALU(ALUci,Read_Data_1,mux4_out,zero,ALU_out);

  DataMem DataMem(clk,Read_Data_2,ALU_out,MemWrite,Ram_Out);

mux2_1 mux5(mux5_out,MemToReg,Ram_Out,ALU_out);

endmodule
```

PC (Program Counter)

There are four signals in the PC module, those are two single bit signals for clock (clk) and reset (rst), and two 32-bit signals of DataIn and DataOut.

DataIn comes from MUXn which allows the PC module between the value of PC+1 or the values of where PC has to go after a BNE, BEQ or Jump instruction are executed.

DataOut starts with a value of zero when the rst signal is equal to zero, and then, when the rst signal is equal to one, DataOut is equal to DataIn, so the output will be equal to the 32-bit input data coming from the MUX previously explained.

```
module PC (
      input [31:0] DataIn,
      input clk,    // Clock
      input rst, // reset
      output reg [31:0] DataOut
);
       always @ (posedge clk or negedge rst) begin
            if(~rst) begin
                  DataOut =32'h000000;
              end
            else begin
                        DataOut=DataIn;
            end
       end
endmodule
```

Adder

There are two adder modules in the implemented MIPS monocycle architecture, both have different functions and destinations.
For explanatory purposes, the first adder will be the one who interacts with PC, the purpose of this adder is to execute the PC+1 operation.

Note: The MIPS architecture implemented is by word and not by byte, that is why PC+1 is used instead of PC+4

In order to execute PC+1, the adder receives two 32-bit signals named Source A (SrcA) and Source B (SrcB), Source A will be the output values of PC, and Source B is a constant value of one, then, the result of the addition will be the output, named Result, which is also a 32-bit signal that will be used for getting to the next instruction, branch instructions or jump instructions.
Adder number two is an adder used for branch instructions (BEQ or BNE), it also has the same number of inputs as the prior adder, but the input signals are different, in this case, SrcA will be the output of the first adder which is PC+1, and SrcB will be the result of the sign extender, which represents, in the case of branch instructions, the value of imm. The sign extender function will be evaluated later when talking about that specific block.

```
module Adder (  ////WIDTH=#INSTRUCTIONS
      input [31:0]SrcA,    // Clock
      input [31:0]SrcB, // Clock Enable
      output reg [31:0] Result  // PC+4
);

 always @ ( SrcA or SrcB)
 begin
     Result = SrcA + SrcB;
 end

endmodule
```

Instruction Memory

The instruction memory gets the information from a .bin file and copies it into a register. This module gets an address as input and sends the instruction hosted at that address to the output.

```
module InstMem
(
	input [31:0] addr,
	output reg [31:0] q
);
	// Declare the ROM variable
	reg [31:0] rom[15:0];
	// Initialize the ROM with $readmemb.  Put the memory contents
	// in the file single_port_rom_init.txt.  Without this file,
	// this design will not compile.

	initial
	begin
		$readmemb("rom_init.bin", rom);
	end

	always @ (addr)
	begin
		q = {rom[addr]};
	end
endmodule
```

Registers

The register file module is where the data of the 32 registers of the MIPS will be stored and written, as inputs, it has a clock signal, which is used to look at every clock cycle if the write enable signal is activated from the control unit in order to write data to a register.

Next input is a 5-bit signal which is used to tell the address where the new data will be written, depending of the instruction, this input can come from the instruction memory from bits 15 to 11 in the case of R type instructions (rd), or from bits 20 to 16 in the case of the sw and lw instructions. It also has a 32-bit input signal to write the data into the given address, this data comes from the MUX output that selects between the result of the ALU and the Data memory.

It has also two more inputs that have the purpose of reading from the instruction memory the equivalents of rs and rt, which made them a 5-bit input signal that comes from the Instruction memory bits 21 to 25 in the case of the first register read port, and bits 16 to 20 for the second port.
The values stored in those registers are the output of the register file, which will be a 32-bit data output.
Values of the register file are initialized by a .txt file containing the 32-bits values of the 32 registers

```
//FILE REGISTER
module RegFile(
      input                    clk,
      // write port
      input                    FR_WE,
      input          [4:0]     FR_Waddr,
      input          [31:0]    FR_Wdata,
      //read port 1
      input          [4:0]     FR_RAddr_1,
      output         [31:0]    FR_Rdata_1,
      //read port 2
      input          [4:0]    FR_RAddr_2,
      output         [31:0]   FR_Rdata_2
 );
       reg     [31:0] reg_File [31:0];
  initial
    begin
      $readmemb("registers.txt",reg_File);
    end
      always @ (posedge clk ) begin

                if(FR_WE) begin
                     reg_File[FR_Waddr] <= FR_Wdata;
                      $writememb("registers.txt",reg_File);
                end
      end
  assign FR_Rdata_1 = reg_File[FR_RAddr_1];
  assign FR_Rdata_2 = reg_File[FR_RAddr_2];
 endmodule
/*--------------- FILE REGISTER -------------------*/
```

Control Unit

The control unit module is one of the most importants modules of this architecture, here, is where all the control signals will be enabled or disabled in order to execute the different instructions. You can even say that this control unit will allow to follow the different data paths, depending on the instruction.

As an input, it has a 5-bit signal (Instruction) coming from the output of the Instruction Memory, it is important to notice that, since the output of Instruction Memory is a 32-bit signal, and the input is only 5 bits wide, the input that really is inserted to the module is the values corresponding to the Opcode, which are bits from number 26 to 31.

Depending on the value of the opcode, the control unit has to determine the type of instruction that wants to be executed. There are 10 output signals, where all but one are single bit outputs, with only the ALUop being a 2-bit output. The truth table for the control unit is shown in the following table:

		Input Opcode Bits						Outputs										
Instruction	Type	Opcode 5	Opcode 4	Opcode 3	Opcode 2	Opcode 1	Opcode 0	RegDst	Jump	BEQ	BNE	MemRead	MemToReg	MemWrite	ALUop		ALUsrc	RegWrite
Add	R	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
Sub	R	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
And	R	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
Or	R	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
Slt	R	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1
Lw	I	1	0	0	0	1	1	0	0	0	0	1	1	0	0	0	1	1
Sw	I	1	0	1	0	1	1	X	0	0	0	0	X	1	0	0	1	0
BEQ	I	0	0	0	1	0	0	X	0	1	0	0	X	0	0	1	0	0
BNE	I	0	0	0	1	0	1	X	0	0	1	X	X	0	0	1	0	0
Jump	J	0	0	0	0	1	0	X	1	0	0	X	X	0	X	X	X	0

There are some outputs that do not matter if they are enabled or disabled while executing an instruction because those components are not required by the instruction or the datapath already ignores them by going to another module.

```
module Control(
  input [5:0] Instruction,
  output reg RegDst,
  output reg Jump,
  output reg BEQ,
  output reg BNE,
  output reg MemRead,
  output reg MemToReg,
  output reg MemWrite,
  output reg [1:0] ALUop,
  output reg ALUsrc,
  output reg RegWrite
);
  always @(Instruction)
    if (Instruction==6'b000000)//R Type Instruction
    begin
```

```
      $display("R Type Instruction ");

RegDst=1;Jump=0;BEQ=0;BNE=0;MemRead=0;MemToReg=0;MemWrite=0;ALUop=2'b10;
ALUsrc=0;RegWrite=1;
    end

  else if (Instruction==6'b100011)//LW Instruction
    begin
      $display("LW Instruction ");

RegDst=0;Jump=0;BEQ=0;BNE=0;MemRead=1;MemToReg=1;MemWrite=0;ALUop=2'b00;
ALUsrc=1;RegWrite=1;
    end

  else if (Instruction==6'b101011)//SW Instruction
    begin
      $display("SW Instruction ");

Jump=0;BEQ=0;BNE=0;MemRead=0;MemWrite=1;ALUop=2'b00;ALUsrc=1;RegWrite=0;
    end

  else if (Instruction==6'b000100)//BEQ Instruction
    begin
      $display("BEQ Instruction ");

Jump=0;BEQ=1;BNE=0;MemRead=0;MemWrite=0;ALUop=2'b01;ALUsrc=0;RegWrite=0;
    end

  else if (Instruction==6'b000101)//BNE Instruction
    begin
      $display("BNE Instruction ");
      Jump=0;BEQ=0;BNE=1;MemWrite=0;ALUop=2'b01;ALUsrc=0;RegWrite=0;
    end

  else if (Instruction==6'b000010)//Jump Instruction
    begin
      $display("Jump Instruction ");
      Jump=1;BEQ=0;BNE=0;MemWrite=0;RegWrite=0;
    end
  else
    begin
    $display("Opcode Incorrecto ");
RegDst=0;Jump=0;BEQ=0;BNE=0;MemRead=0;MemToReg=0;MemWrite=0;ALUop=2'b00;
ALUsrc=0;RegWrite=0;
    end
endmodule
```

ALU Control

The ALU Control is encouraged to control the Arithmetic logic unit. This is a combinational circuit that gets the Function (first 6 bits) from the instruction and is enabled by the control module through a 2 bits bus. The output is a 4 bits bus that controls the operation that ALU will do.

The following table shows the control combinations of this block.

INPUTS

ALUOp: Control signals from control module.

Funct field: Function (first 6 bits) from the instruction

OUTPUT

ALU control input: Control signals to control ALU module

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

```
module ALU_control(input [5:0]funct,
                   input [1:0]ALUOp,
                   output [3:0]ALUci);
  assign ALUci[3]=1'b0;

  assign ALUci[2]=~ALUOp[1]&ALUOp[0] |
ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0] |
ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&funct[3]&~funct[2]&funct[1]&~funct[0];

  assign ALUci[1]=~ALUOp[1]&~ALUOp[0] | ~ALUOp[1]&ALUOp[0] |
ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&~funct[3]&~funct[2]&~funct[1]&~funct[0] |
ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&~funct[3]&~funct[2]&funct[1]&~funct[0] |
ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&funct[3]&~funct[2]&funct[1]&~funct[0];
```

```
  assign
ALUci[0]=ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&~funct[3]&funct[2]&~funct
[1]&funct[0] |
ALUOp[1]&~ALUOp[0]&funct[5]&~funct[4]&funct[3]&~funct[2]&funct[1]&~funct
[0];
endmodule
```

ALU

Another important hardware component is the Arithmetic Logic Unit (ALU), this module has two 32-bit inputs, which are the ALU sources, these inputs are represented as inA and inB, it also has a 4-bit input which is the ALUop, this input comes from the ALU control module and is the selector, this signal will allow the ALU to choose which operation will be executed between both sources.

It also has two outputs, one of them is the result of the ALU, which will also be a 32-bit signal, and then, there is a single bit output named “zero”, which can be seen as a flag that is enabled when the result of the ALU is equal to zero, this signal will be useful when executing BEQ and BNE instructions, since they depend if both sources are equal or non equal respectively.

The ALU can execute five different operations between both sources, this operations are:

- And
- Or
- Add
- Sub
- Comparison (inA<inB)

As said before, the ALU control module provides the information for the ALU to choose which operation will be made.

```
module ALU
  (
    input [3:0]ALUOp,
    input [31:0]inA,inB,
    output reg Zero,
    output reg [31:0]Result
  );
```

```
  always @ (ALUOp,inA,inB) begin
      case (ALUOp)
            4'b0000: Result = inA & inB;
            4'b0001: Result = inA|inB;
          4'b0010: Result = inA+inB;
            4'b0110: Result = inA-inB;
            4'b0111: begin
                  if(inA<inB)
                    Result = 32'b1;
                  else
                    Result = 32'b0;
                  end

            default: Result = 32'bZ;
     endcase
     if(Result==0)
         Zero=1;
      else
        Zero=0;
  end
endmodule
```

Sign Extender

For some instructions it is necessary to get an imm value which is a 16 bits number and use it in an adder or in the ALU. These modules need 32 bits of input. For this reason this module gets the 16 bits number extracted from the instruction (bits 0 to 15) and extends its sign to 32 bits. When it is a negative number, it extends the 1 but when it's a positive number it extends the 0.This is done by getting the most significant bit from the input and extending it 16 times.

```
module SignExt(
  input wire [15:0] DataIn,
  output reg [31:0] DataOut);

  always @*
    begin
      if (DataIn[15]==0)
        DataOut[31:16] = 16'b0000000000000000;
      else
        DataOut[31:16] = 16'b1111111111111111;
```

```
            DataOut[15:0] = DataIn;
    end
endmodule
```

Data memory

The data memory is a RAM memory which is used for lw and sw instructions, data is initialized from a txt file, and it has two main inputs which are the address and the data, both are 32-bit wide and are provided by the ALU result in the case of the address and the Read Data 2 signal coming from the file register.
It is important to notice that in order to write any data, a write enable signal must be enabled, this signal is activated by the control unit when a sw or lw instruction is being executed. The output will always be the data from the address that can be readed.

```
module DataMem(
      input clk,
      input [31:0] WD, // 32bit data input word
      input [31:0] A,
      input we,              // Active high
      output [31:0] RD // 32bit output word
);

      // Declare A bidimensional Array for the RAM
  reg [31:0] ram [0:31];
      reg [31:0] addr_buff;

      // RAM's don't have reset
      // The default value from each location is X
      // The write is synchronous

      initial
      begin
       $readmemb("ram_init.txt", ram);
      end

      always @(posedge clk ) begin

            begin
```

```
                if(we) begin
                      ram[A] <= WD;
              //$display("write %0d from dir %0d",ram[A],A);

              //$writememb("ram_init.txt", ram);
                end

          end
        $writememb("ram_init.txt", ram);
       for(int i=0;i<31;i++)$display("%b",ram[i]);
      end

      // The read is synchronous As the Address
      // was buffered on the clk using Addr_buff
      assign RD = ram[A];

endmodule
```

MUXES

It is a device used to transmit data from different inputs to a single output, i.e., all data entering the circuit leave through the same place. This time we only needed a 2 to 1 mux, but with different input buses. Two different ones were used, one with a width of 32 bits and the other with a width of 5 bits in the inputs.

The first mux that was implemented with a width of 5 bits, was in the input of the "Registers" module where the rs [15-11] and rt [20-16] signal is connected, both connected to the Write data.

The second mux with a width of 32 bits is located at one of the ALU inputs, which receives the Read data 2 signal from "Registers" and "Sign-extend" from the bus [15-0] of the “imm” of the instruction.

The third mux with a width of 32 bits is located after the "Adder", which receives the signals of the "Adder" and the PC+1, coming from the Jump instruction. Connected to one of the inputs of the 4th mux.

The fourth mux with a width of 32 bits is located at the output of the 3rd mux, at the input it has the signal of the 3rd mux and the address of the Jump instruction, connected to the input of the PC.

The following shows the mux module, which is parameterizable.

```
module mux2_1 (Z,Sel,X,Y);
  parameter WID = 32;
  input wire [WID-1:0] X, Y;
  input wire Sel;
  output wire [WID-1:0] Z;

  assign Z= Sel ? X:Y;

endmodule
```

Detailed description of the verification process

Logic Tester

The following instruction sequence it's for a quick inspection through all the instructions on the architecture excluding the lw function which is going to be tested later. The objective in order to test an instruction is writing the register memory and the data memory with the result of the assigned addresses for each instruction. The example below is going to skip the addition instruction but everything else is going to be executed excluding the bne instruction.

```
1:00000000000000000000000000000000

2:beq 000100 00001 00001 0000000000000010   if rs==rt pc=bta -!to aovid

3:add 000000 00001 00010 00110 00000 100000 saves $1+$2 in $6
      op      $1    $2    $6   shamt  fun

4:sw 101011 00000 00110 0000000000000000  saves $6 in mem0
    dest reg addr Source reg add    imm

5:sub 000000 00011 00001 00111  00000 100010 saves $3-$1 in $7
      op     $3     $1    $0    shamt   fun

6:sw 101011  00001 00111  0000000000000000 saves $7 in mem1
      dest reg addr  Source reg add   imm

7:jump 000010 00000000000000000000001001   pc=jta  add=8 to avoid j


8:bne 000101 00001 00010 0000000000000010   if rs==rt pc=bta


9:and 000000 00001 00010 01000 00000 100100 saves $1&$2 in $8
      op      $1    $2    $8    shamt  fun

10:sw 101011 00010 01000 0000000000000000 saves $8 in mem2
    dest reg addr   Source reg add  imm

11:or 000000 00001 00010 01001 00000 100101 saves $1&$2 in $9
      op     $1     $2     $0  shamt  fun

12:sw 101011 00011 01001 0000000000000000 saves $9 in mem3
```

```
      dest reg add Source reg add   imm

13:slt 000000 00001 00011 01010 00000 101010 saves $1<$3 in $10
      op        $1   $3    $0   shamt  fun

14:sw 101011 00100 01010 0000000000000000 saves $10 in mem4
  dest reg addr Source reg add imm
```

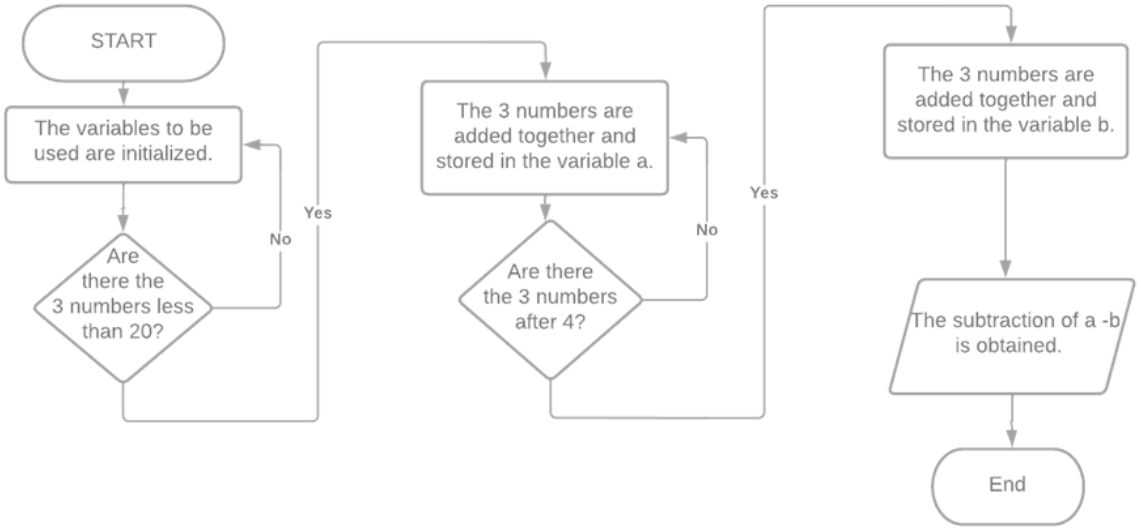
Register memory assign:

```
reg mem:
0:0
1:1
2:2
3:3
4:4
5:5
6:add
7:sub
8:and
9:or
10:slt
```

Data memory Result:

```
0:00000000000000000000000000000000  //add 2+1  #skipped
1:00000000000000000000000000000010 //sub 3-1
2:00000000000000000000000000000000 //and 10&01
3:00000000000000000000000000000011 //or 10|01
4:00000000000000000000000000000001 //slt 1<3?
5:00000000000000000000000000000000
```

Program flowchart



C code

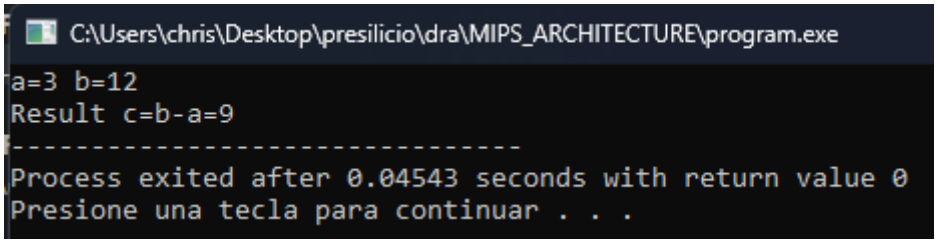
Make a C program that obtains the sum of 3 numbers before 20 and stores the result in “a”, then obtains the sum of 3 numbers after 4 and stores it in “b”, and finally obtains the subtraction of b-a .

```
#include <iostream>
int a=0;
int b=0;
int c=0;
using namespace std;

int main()
{

for (int i=0;i<3;i++){
      a=a+1;
}
for (int i=0;i<3;i++){
      b=b+4;

}
c=b-a;
cout<<c;
}
```



Assembly code

```
/////register location/////////
// $t0=8,$t1=9,$t2=10,$t3=11//
// $s0=16, $s1=17, $s2=18  //
////////////////////////////

////////register file////////
//$t0=3  //// for condition i<3
//$s0=0  //// for iterator i
//$s1=1  //// Const 1
//$s2=4  //// Const 4
//$t1=0  ///a
//$t2=0  ///b
//$t3=0  ///C
////////////////////////////

for1:bne $s0,$t0, a
j next
a: add $s0, $s1,$s0  /// $s0=$s0+1
     add $t1, $s1,$t1  /// $t1=t1+1
     j for1
next:

lw $0,0($s0)
for2:bne $s0,$t0, b
j end
b: add $s0, $s1,$s0  /// $s0=$s0+1
     add $t2, $s2,$t2  /// $t1=t1+4
     j for2
end:
```

```
sub $t3, $t2,$t1  /// $t3=$t2-$t1
sw $t3,0($0) // Result saved on the data memory
```

Machine code

Machine code translation

```
1:0:00000000000000000000000000000000
2:1:bne 000101     10000      01000   0000000000000001   if rs==rt pc=bta
            op    rs=$s0     rt=$t0          imm
3:2:jump 000010    00000000000000000000000110   pc=next
4:3:add 000000     10000      10001 10000 00000 100000  saves $s0+1 in $s0
            op       $s0       $s1   $s0  shamt  fun
5:4:add 000000     01001      10001 01001 00000 100000 saves $t1+1 in $t1
            op      $t1        $s1   $t1  shamt  fun
6:5:jump 000010    00000000000000000000000001   pc=next
7:6:lw 100011     00000   10000    0000000000000000       saves 0 in $s0
          op       $0      $s0          imm
8:7:bne 000101     10000      01000   0000000000000001   if rs==rt pc=bta
            op      rs=$s0  rt=$t0        imm
9:8:jump 000010    00000000000000000000001100   pc=end
10:9:add 000000    10000      10001 10000 00000 100000  saves $s0+1 in $s0
            op      $s0        $s1   $s0  shamt   fun
11:10:add 000000   01010      10010 01010 00000 100000  saves $t1+1 in $t1
            op      $t2        $s2   $t2  shamt  fun
12:11:jump 000010  00000000000000000000000111   pc=end
13:12:sub 000000   01010   01001    01011 00000 100010  saves $t2-$t1 in $t3
               op   $t2     $t1      $t3  shamt  fun
14:13:sw 101011    00000      01011   0000000000000000  saves $t3 in mem0
             op      $0        $t3          imm
```

Main program

```
00000000000000000000000000000000
00010110000010000000000000000001
00001000000000000000000000000110
00000010000100011000000000100000
00000001001100010100100000100000
00001000000000000000000000000001
10001100000100000000000000000000
00010110000010000000000000000001
```

```
00001000000000000000000000001100
00000100001000110000000100000
00000001010100100101000000100000
00001000000000000000000000000111
00000001010010010101100000100010
10101100000010110000000000000000
00000000000000000000000000000000
```

Register file

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000001
00000000000000000000000000000100
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

opResults

Just as the results of the C program, as we can observe on the results register ($t3), the results were equal in both cases, this is a good sign of the functionality of our architecture but there are some instructions left to try (slt,and,or,lw).

C problem translation

Register memory

```
00000000000000000000000000000000 //$0
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000011 //t0
00000000000000000000000000000011 //t1
00000000000000000000000000001100 //t2
00000000000000000000000000001001 //t3
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000011 //s0
00000000000000000000000000000001 //s1
00000000000000000000000000000100 //s2
00000000000000000000000000000000
00000000000000000000000000000000
```

Data Memory

```
00000000000000000000000000001001
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Link to the project on EDAPlayground

Logic Tester: https://www.edaplayground.com/x/MTNN

C program translation project: https://www.edaplayground.com/x/r7aE

Conclusions

Josue Isaias Gomez Cosme

Thanks to this project we visualized one way to design a data path, which is to examine the main components needed to execute each kind of MIPS instruction. We also worked with the modeling in Verilog, besides allowing us to review 10 MIPS instructions and implement them in a functional project from C++ language, to assembler and then to machine language. Where it was possible to test the functionality through the EPWave and the EDAPlayground's own console. It was quite laborious work but with a quite big reward of knowledge at the moment of realizing a single-cycle MIPS processor.

Jose Alberto Gomez Diaz

For the development of this monocycle processor it was very important to take into consideration the data path that each instruction must follow. In this way it was possible to review the correct operation of the processor by visualizing the signals that were in each segment. This type of processor is useful for performing simple instructions in a single clock cycle.

Bruno Hernandez Lopez

Implementing this MIPS monocycle architecture gave a better understanding of the hardware elements required to execute different types of instructions, seeing the different data paths that had to be followed also made clearer how instructions worked. While assembling the HDL code in EDAPlayground, the complete diagram of the MIPS structure was very useful since it made it easier to identify how the data flows into the different modules. While testing the final verilog implementation, it was a little bit difficult to understand where the data was being stored due to the lack of understanding the operation of some instructions, but when we catched up, it was easier and it made things better at the moment of implementing the application code for testing. Finally, I believe that this monocycle architecture will be a great base for implementing other types of architecture in the future, so it was very interesting.

Christian Ortega Blanco

At the beginning it was difficult to think about how to start, there were too many pieces, and we all had trouble understanding exactly what to do until the teacher gave us some hints, then immediately we looked for David A. Patterson's book and it was there. All we need to do is to distribute tasks and start programming. Programming took us a couple of hours, but understanding all the logic datapaths took us a couple of days. I really enjoyed learning about how a computer actually works and making one.

Future Work

To improve the program, we would first try to add all the missing MIPS instructions, as well as generate a simpler way to test each one of them, this can be achieved with the help of random instructions generated with "with" which allows to give specific restrictions for each one of them.

References

1. Diseño de la ruta de datos y la unidad de control (ucm.es)
2. A single-cycle MIPS processor (washington.edu)
3. https://www3.ntu.edu.sg/home/smitha/fyp_gerald/