



PROYECTO

González Ramos Jorge Humberto
Jiménez Vital Francisco
Velasco Ochoa Dana

Equipo 2



30 DE MAYO DE 2020
SEMINARIO DE SOLUCION DE ARQUITECTURA DE COMPUTADORAS
Sección D14

MIPS

MIPS por las siglas de Microprocessor without Interlocked Pipeline Stages, se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

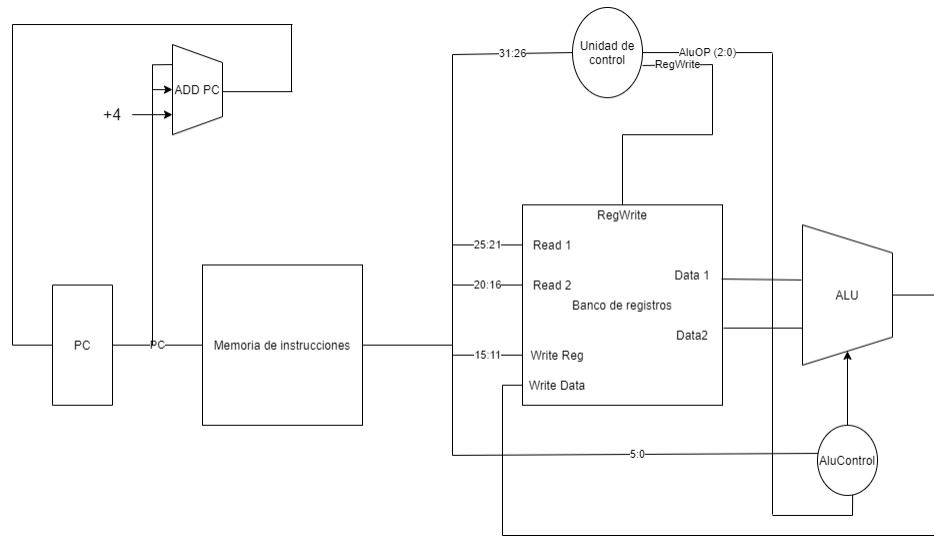
Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI; en muchos sistemas embebidos; en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable. Más recientemente, la NASA usó uno de ellos en la sonda New Horizons.

En los últimos años gran parte de la tecnología empleada en las distintas generaciones MIPS ha sido ofrecida como diseños de "IP-cores" (bloques de construcción) para sistemas embebidos. Se ofertan los núcleos básicos de 32 y 64 bits, conocidos respectivamente como 4K y 5K respectivamente, y con licencias MIPS32 y MIPS64. Estos núcleos pueden ser combinados con unidades añadidas tales como FPU's, sistemas SIMD, dispositivos de E/S, etc.

Los núcleos MIPS han sido comercialmente exitosos, siendo empleados actualmente en muchas aplicaciones industriales y de consumo. Pueden encontrarse en los más modernos routers Cisco, TP-Link y Linksys, cablemódems y módems ADSL, tarjetas inteligentes, controladoras de impresoras láser, decodificadores de TV, robots, ordenadores de mano, Sony PlayStation 2 y Sony PlayStation Portable.

En móviles y PDA's, sin embargo, el núcleo MIPS fue incapaz de desbancar a su competidor de arquitectura ARM.

Explicación DataPath Fase 1



Para la fase 1 se juntaron solo los módulos PC, Add PC, Memoria de instrucciones, Unidad de control, Banco de registros, un controlador para la Alu y la misma Alu.

```

1 module pc (
2     input clk,
3     input [31:0] entrada,
4     output reg [31:0] salida
5 );
6
7
8 initial
9 begin
10     salida <= -4;
11 end
12
13 always@(posedge clk)
14 begin
15     salida <= salida+32'd4;
16 end
17
18
19 endmodule
20
21

```

En el módulo **PC** se juntó con el **Add PC** recibiendo una entrada de 32 bits y obteniendo una salida de 32bits haciendo una suma de +4Decimal en bits, reaccionando al juego del reloj mientras está en positivo.

En el módulo de **memoria de instrucciones** se hizo una lectura en el archivo instrucciones recibiendo un dato de 32 bits que se separan más adelante.

```

1 module instrumemory(
2     input [31:0] direccion,
3     output reg [31:0] salida_datos
4 );
5
6
7 reg [7:0] mem[0:128];
8
9
10 initial begin
11     $readmemb("instrucciones.txt", mem);
12 end
13
14 always @*
15 begin
16     salida_datos <= {mem[direccion], mem[direccion+1], mem[direccion+2], mem[direccion+3]};
17 end
18
19 endmodule
20
21

```

Los módulos Unidad de control, banco de registros y Alu Control, reciben sus entradas de esta salida utilizando la separación de bits.

```

module UnidadControl (
    input [5:0] entrada,
    output [0:0] RegDst,
    output [0:0] AluSrc,
    output [2:0] AluOp,
);

```

En el módulo **Unidad de control** se reciben los últimos 6bits que tiene la salida de memoria de instrucciones la cual sirve para decidir en esta fase si serán operaciones de tipo R, otorgando la decisión al controlador de la Alu mediante 3bits y la posición donde se guardará el registro al banco de registros.

El módulo **controlAlu** recibe los primeros 6bits de la salida de memoria de instrucciones que define la operación y 3 bits que se dan del controlador para decir el tipo de operación tipo R en esta fase y ofreciendo una salida de 4 bits con la operación a realizar ya filtrada por tipo.

```

1 module controlAlu (
2     input [5:0] entrada,
3     input [2:0] op,
4     output reg [3:0] salida
5 );
6
7 always @(entrada, op)
8 begin
9     if (op == 3'b010) //2 func based
10    begin
11        if (entrada[3:0] == 0) begin //sum
12            salida <= 4'b0010;
13        end
14        else if (entrada[3:0] == 4'b0010) begin //resta
15            salida <= 4'b0110;
16        end
17        else if (entrada[3:0] == 4'b0100) begin //and
18            salida <= 4'b0000;
19        end
20        else if (entrada[3:0] == 4'b0101) begin //or
21            salida <= 4'b0001;
22        end
23        else if (entrada[3:0] == 4'b1100) begin //lessThan
24            salida <= 4'b0111;
25        end
26        else if (entrada[3:0] == 4'b0011) begin //multiplicar
27            salida <= 4'b1010;
28        end
29    end
30 end

```

```

1 module BancoRegistros (
2     input [4:0] ReadReg1,
3     input [4:0] ReadReg2,
4     input [4:0] WriteReg,
5     input [31:0] WriteData,
6     input RegWrite,
7
8     output reg [31:0] ReadData1,
9     output reg [31:0] ReadData2
10 );
11
12 reg [31:0] AlmacenReg[0:31];
13
14 initial
15 begin
16     $readmemb("bankcoreg.txt", AlmacenReg, 0, 31);
17     #10;
18 end
19
20 always @*
21 begin
22     AlmacenReg[WriteReg] <= WriteData;
23
24     ReadData1 <= AlmacenReg[ReadReg1];
25     ReadData2 <= AlmacenReg[ReadReg2];
26 end
27
28 endmodule

```

Mientras que a los anteriores módulos se les conectaron los primeros y últimos 6bits al **banco de registros** le toca recibir todos los demás bits separados de 5 en 5 los cuales contienen una dirección para guardar un dato nuevo y las direcciones de los datos que se leen en el archivo y son mandados a ser operados por la Alu.

En el módulo **Alu** se reciben las 3 entradas antes mencionadas 2 datos que interactuarán con un operador y un selector que dice que operación se tendrá que hacer dando el resultado al banco de registros para ser guardado en una posición que ya se le dio anteriormente y ofreciendo una salida más para saber si el resultado queda en 0's.

```

1 module alu (
2     input [31:0] Data1,
3     input [31:0] Data2,
4     input [3:0] selector,
5     output reg [31:0] salida,
6     output reg zflag
7 );
8
9
10 reg [31:0] tmp;
11
12 always @*
13 begin
14     case (selector)
15         4'b0000: salida <= Data1 & Data2;
16         4'b0001: salida <= Data1 | Data2;
17         4'b0010: salida <= Data1 + Data2;
18         4'b0110: salida <= Data1 - Data2;
19         4'b0111: salida = Data1 < Data2 ? 1:0;
20         4'b0101: salida <= Data1 ^ Data2; // XOR
21         4'b1100: salida <= ~(Data1 | Data2); // nor
22         4'b1000: salida <= Data1 * Data2; // MUL
23         4'b1010: salida <= Data1 / Data2; // div
24         default: salida <= Data2;
25     endcase
26 end
27
28 endmodule

```

Se agregaron 4 buffers, 1 Memoria de datos, el módulo sing extender, shift left, un adder shift PC, una compuerta and llamada Branch y 4 multiplexores y fueron editadas algunas entradas y salidas de algunos módulos para la nueva fase.

```

1 module Mux1(
2     input [31:0] ResBranch1,
3     input [31:0] ResPC0,
4     input  PcSrc,
5     output [31:0] BResult
6 );
7
8 assign
9     BResult = (PcSrc)? ResBranch1 : ResPC0;
10
11 endmodule

```

Modificando las salidas de los módulos existentes, el **primer buffer** recibe 2 entradas, una de pc y otra de la memoria de instrucciones, se recorren las salidas de la memoria de instrucciones al buffer y se hace la separación de bits en las instancias no en el módulo mismo. Las entradas del banco de registros quedan igual.

```

1 module sing_ext(
2     input [15:0] entrada,
3     output [31:0] salida
4 );
5
6 assign salida = (entrada[15] == '1'b1 ? {16'b1111111111111111, entrada}
7               : {16'b0, entrada});
8
9 endmodule

```

```

1 module UnidadControl (
2     input [1:0] entrada,
3     output reg RegDst,
4     output reg ALUSrc,
5     output reg [1:0] ALUOp,
6     output reg Branch,
7     output reg MemWrite,
8     output reg MemRead,
9     output reg Jump,
10    output reg MemtoReg,
11    output reg RegWrite
12 );
13
14 always*
15 begin
16     RegDst <= 0; // MPX (0 <= 20-16, 1-15-11) <
17     ALUSrc <= 0; // no salto
18     ALUOp <= 0; // usa reg 0 o extended/imm 1
19     Branch <= 0; // de mem a reg
20     MemWrite <= 0; // guardar en reg
21     MemRead <= 0; // leer mem
22     Jump <= 0; // guardar mem
23     MemtoReg <= 0; // no branch
24     RegWrite <= 0; // ALU funched
25 end

```

A la **unidad de control** se le agregaron instrucciones tipo I como lo son ADDI, SLTI, BEQ, LW, SW que nos ayudarán a ejecutar código en ensamblador, así mismo la unidad de control eso conlleva más salidas enviadas al segundo buffer que este a su vez lo mandará a tercero para después ser manipulado.

En el **segundo buffer** se asignaron 17 de las cuales 10 entradas que provienen de la unidad de control y otras 4 del primer buffer y 2 del banco de registros y 1 del reloj.

```

38 always @(posedge clk)
39 begin
40     sal_Regwrite <= Regwrite;
41     sal_MemtoReg <= MemtoReg;
42     sal_Memwrite <= Memwrite;
43     sal_Jump <= Jump;
44     sal_MemRead <= MemRead;
45     sal_Branch <= Branch;
46     sal_ALUOp <= ALUOp;
47     sal_ALUSrc <= ALUSrc;
48     sal_RegDst <= RegDst;
49     sal_addPc <= addPc;
50     sal_SingJump <= SingJump;
51     data1_salida <= data1;
52     data2_salida <= data2;
53     sal_SingEx <= Sing_ex;
54     salida1 <= en1;
55     salida2 <= en2;
56 end
57
58 endmodule

```

```

1 module Shift_Left(
2     input [31:0] entrada,
3     output [31:0] salida
4 );
5
6 assign salida = {entrada<<2};
7
8 endmodule

```

en el modulo **shift left** se conectó su salida de 32 bits al shift Adder PC, mientras que a su entrada se le hace un desplazamiento de 32 bits a partir de la extensión del campo inmed.

El módulo **Shift Adder PC** genera una suma de los módulos ADD PC y el shift left, este modulo nos ayuda para el salto condicional (BEQ).

```

1 module Adder(
2     input [31:0] entrPc,
3     input [31:0] entrShift,
4     output [31:0] ResBranch
5 );
6
7 assign ResBranch = entrPc + entrShift;
8
9 endmodule

```

```

1 module Mux2(
2     input [31:0] sing_ext1,
3     input [31:0] data0,
4     input alusrc,
5     output [31:0] salida
6 );
7
8 assign
9     salida = (alusrc)? sing_ext1 : data0;
10
11 endmodule

```

Este **segundo multiplexor** recibe la entrada de la Data2 del banco de registros y del shift extender y el ALUSrc es el condicional que escoge la salida tal que la salida se dirige a la ALU que se mantiene igual a la fase1 junto a su controlador especial.

El **tercer multiplexor** recibe los primero 10 bits de la separación de bits del primer buffer que se dirige hacia el segundo buffer siendo que el segundo buffer otorgó esas entradas al multiplexor.

```

1 module Mux3(
2     input [4:0] entrada1,
3     input [4:0] entrada0,
4     input RegDst,
5     output [4:0] salida
6 );
7
8 assign
9     salida = (RegDst)? entrada1 : entrada0;
10
11 endmodule

```

```

31 always @(posedge clk)
32 begin
33     sal_Regwrite <= Regwrite;
34     sal_MemtoReg <= MemtoReg;
35     sal_Jump <= Jump;
36     sal_Memwrite <= Memwrite;
37     sal_MemRead <= MemRead;
38     sal_Branch <= Branch;
39     sal_Jumpv <= Jumpv;
40     sal_OutBranch <= OutBranch;
41     sal_zflag <= zflag;
42     sal_Alures <= Alures;
43     sal_Data2 <= Data2;
44     sal_writeReg <= writeReg;
45 end
46
47 endmodule

```

En el **tercer buffer** se tienen 12 entradas y 12 salidas, mantiene y pasa los datos de las salidas anteriores para dirigirlos a la memoria de datos, a una compuerta and llamada Branch y al cuarto buffer.

El módulo **Branch** funge como and analizando el Zflag y una salida de la unidad de control de 1 bit que define lo que será el selector del ADD PC.

```

1 module branch(
2     input Branch,
3     input zflag,
4     output salida
5 );
6
7 assign
8     salida = zflag & Branch;
9
10 endmodule

```

```

1 module MemoriaDato(
2     input clk,
3
4     input [31:0] Address,
5     input [31:0] WriteData,
6     input Memwrite,
7     input MemRead,
8
9     output reg[31:0] MemRes
10 );
11
12 reg [31:0] mem[31:0];
13
14 integer i;
15 initial begin
16     for (i = 0; i < 32; i = i + 1) begin
17         mem[i] = i;
18     end
19 end
20
21 always@(posedge clk) begin
22     if(Memwrite == 1) mem[Address] <= WriteData;
23     if(MemRead == 1) MemRes <= mem[Address];
24 end
25
26
27 endmodule

```

La **memoria de datos** es una expansión del banco de registros por falta de espacio.

El **cuarto buffer** recibe los registros de la memoria de datos y el resultado de la ALU y la salida del multiplexor 3, este buffer sirve para guardar temporalmente los datos y pasarlos a sus respectivos lugares de entrada.

```

1 module buffer4 (
2     input clk,
3     input Regwrite,
4     input MemtoReg,
5     input [31:0] MemRes,
6     input [31:0] Alures,
7     input [4:0] writeRegister,
8
9     output reg sal_Regwrite,
10    output reg sal_MemtoReg,
11    output reg[31:0] sal_MemRes,
12    output reg[31:0] sal_Alures,
13    output reg[4:0] sal_writeReg
14 );
15
16 always @(posedge clk)
17 begin
18     sal_Regwrite <= Regwrite;
19     sal_MemtoReg <= MemtoReg;
20     sal_MemRes <= MemRes;
21     sal_Alures <= Alures;
22     sal_writeReg <= writeRegister;
23 end
24
25
26 endmodule

```

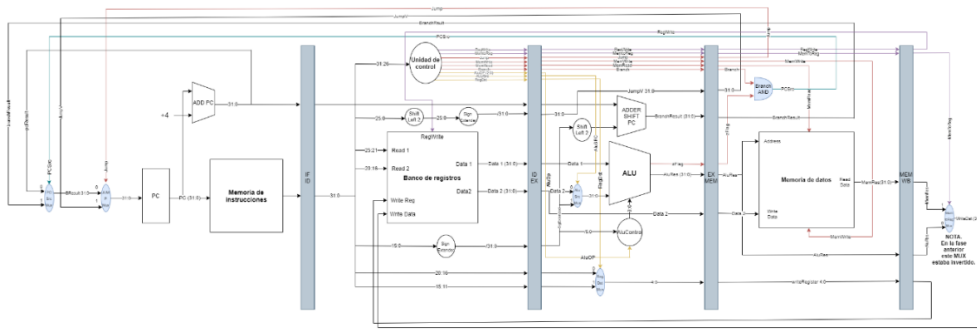
```

1 module Mux4(
2     input [31:0] Alures0,
3     input [31:0] MemRes1,
4     input MemtoReg,
5     output [31:0] WriteDat
6 );
7
8 assign
9     WriteDat = (MemtoReg)? MemRes1: Alures0;
10
11 endmodule

```

A este **cuarto multiplexor** se le asignan las salidas de la memoria de datos y el resultado de la ALU, con una entrada de la unidad de control decide qué salida tendrá el multiplexor.

DataPath Fase 3



En esta ultima etapa, se agregaron las intrucciones de salto, (tipo J) y ya con esta ultima modificación se completa lo que seria un procesador basico y funcional. Se incluyeron 3 módulos los cuales son shift_left_J, Mux_J, Sing_ext_J, y se le añadió al alu control instrucciones de tipo J

```

1 module shift_left_J(
2     input [25:0] entrada,
3     output [25:0] salida
4 );
5
6 assign salida = {entrada<<2};
7
8 endmodule
    
```

En el **shift_left_J** La entrada se multiplica por 4 moviendo 2 bits a la izquierda, dato que al pasar por los buffers que adelante se dirigirá a PC ADD

El multiplexor de instrucciones J **Mux_J** se dedica a asignarle la salida de shift_left_J y el resultado del PC ADD, con una entrada de la unidad de control decide qué salida tendrá el multiplexor.

```

1 module Mux_J(
2     input [31:0] Jumpv1,
3     input [31:0] BResult0,
4     input Jump,
5     output [31:0] Pc
6 );
7
8 assign
9     Pc = (Jump)? Jumpv1 : BResult0;
10
11 endmodule
12
13
    
```

```

1 module sing_ext_J(
2     input [25:0] entrada,
3     output [31:0] salida
4 );
5
6 assign
7     salida = (6'b0+entrada);
8
9 endmodule
    
```

El módulo **sing_ext_J** hace que el resultado dado por shift_left_J tenga el mismo valor mientras que se vuelve un dato de 32 bits utilizados en PC ADD

El modulo **controlAlu** se vio modificado agregándole el condicional que revisa si es una instrucción de tipo J junto con sus selectores de operación que se darán a la Alu

```

34
35
36 else if(entrada[3:0] == 4'b1010) begin //division
37     salida = 4'b010;
38 end
39
40 else if(entrada[3:0] == 4'b0110) begin //xor
41     salida = 4'b0101;
42 end
43
44 else if(entrada[3:0] == 4'b0111) begin //nor
45     salida = 4'b100;
46 end
47
48 else if (Op == 3'b000) begin
49     salida = 4'b0010;
50     //display("ALUCO_Suma");
51 end
52 else if (Op == 3'b001) begin
53     salida = 4'b0110;
54     //display("ALUCO_Resta");
55 end
56 else if (Op == 3'b011) begin
57     salida = 4'b0000;
58     //display("ALUCO_AND");
59 end
60 else if (Op == 3'b100) begin
61     salida = 4'b0001;
62     //display("ALUCO_OR");
63 end
64 else if (Op == 3'b101) begin
65     salida = 4'b0111;
66     //display("ALUCO_SLT");
67 end
68
    
```


Programa de ensamblador Selective Sort

```
main:
    li $t5, 85
    li $t4, 22
    li $t3, 94
    li $t2, 55
    li $t0, 105
    li $t6, 58
    li $t7, 43
    li $t8, 39
    li $t1, 69

    sw $t5, 0($zero)
    sw $t4, 1($zero)
    sw $t3, 2($zero)
    sw $t2, 3($zero)
    sw $t1, 4($zero)
    sw $t6, 5($zero)
    sw $t7, 6($zero)
    sw $t8, 7($zero)
    sw $t0, 8($zero)
    li $t9, 8 # total de nums

.sort:
    add $t0, $zero, $zero # i = 0

    .loop1:
        addi $t1, $t9, -1 # n - 1
        add $t2, $t0, $zero # index
= i
        addi $t3, $t0, 1 # j = i+1

        .loop2:
            nop
            lw $t4, 0($t3) # arr[j]
            lw $t5, 0($t2) #
arr[index]

            # if arr[j] < arr[index]
            blt $t4, $t5 .change

            j .addj
```

```
        .change:
            add $t2, $zero, $t3
# index = j

        .addj:
            addi $t3, $t3, 1 # j
= j +1
            blt $t3, $t9, .loop2
# mientras j < n

        .swap:
            lw $t4, 0($t2) #
arr[index]
            lw $t5, 0($t0) # arr[i]

            add $a1, $zero, $t4 #
temp = arr[index]

            sw $t5, 0($t2) #
arr[index] = arr[i]
            sw $a1, 0($t0) # arr[i]
= temp

        .add_i:
            addi $t0, $t0 1 # i = i
+ 1
            blt $t0, $t1 .loop1 #
mientras i < n - 1

    .end:
        lw $t1, 0($zero)
        lw $t1, 1($zero)
        lw $t1, 2($zero)
        lw $t1, 3($zero)
        lw $t1, 4($zero)
        lw $t1, 5($zero)
        lw $t1, 6($zero)
        lw $t1, 7($zero)
        lw $t1, 8($zero)
```

Este programa tiene como objetivo hacer ordenamientos de números enteros utilizando el método selective sort, fue basado en el código de la siguiente página

<https://www.geeksforgeeks.org/selection-sort/>

Programa de ensamblador Tipos de Triángulos

```
.main:
    li $t0 2 # lado 1
    li $t1 35 # lado 2
    li $t2 30 # lado 3
    sub $t4 $t0 $t1
    sub $t5 $t0 $t2
    beq $t4 $zero .f1

    .iso:

    sub $t6 $t1 $t2

    beq $t4 $zero .isocetes # a ==
b
    beq $t5 $zero .isocetes # a ==
c
    beq $t6 $zero .isocetes # b ==
c

    # si no entonces es escaleno
    j .escaleno

.f1:
    beq $t5 $zero .equilatero

    j .iso

.equilatero:
    li $t3 1
    j .end

.isocetes:
    li $t3 2
    j .end

.escaleno:
    li $t3 3

.end:
    sw $t3 0($zero) # guardar tipo
de triangulo
```

El objetivo de este programa es determinar que tipo de triángulo es, para eso utilizamos 3 registros donde guardamos la medida de sus lados y después hacemos sus cálculos.

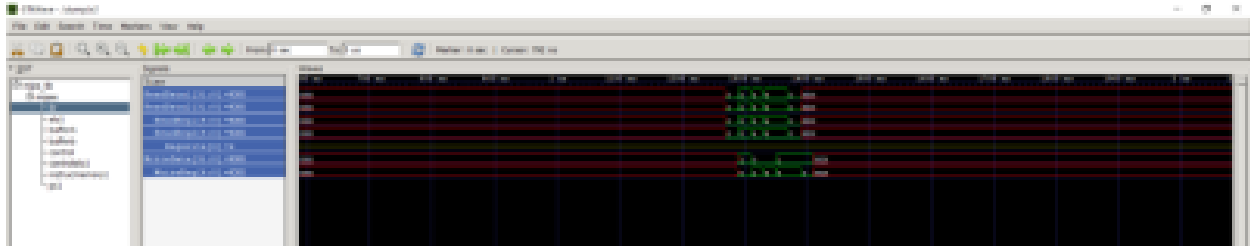
Un == es el equivalente a 2 registros restados dando 0, con esto ya podemos realizar la operación igual usando un branch con el registro \$zero

El equivalente de este código en python sería el siguiente:

```
a = 2
b = 35
c = 30
if a == b and a == c:
    resultado = 1
elif a==b or a==c or b == c:
    resultado = 2
else:
    resultado = 3
```

TestBenches

TestBench Fase 1



```
1 timescale 1ns / 1ps
2 module mips_tb;
3
4     reg clk = 0;
5
6     mips mipsx(clk);
7     always begin
8         #10;
9         clk <= ~clk;
10    end
11
12    integer i,f;
13    initial begin
14        $dumpvars(0, mips_tb);
15        #50000;
16        $finish;
17    end
18
19 endmodule
```

Se comprobó que estaba funcionando la fase 1 mediante un tb que revisaba el flujo entre el banco de registros y la memoria de instrucciones.

TestBench Fase3



```
.inicio:
    addi $t0, $zero, 1337 #reg 8 -> 1337
    j .salto2

.salto1:
    addi $t0, $t0, -724 #reg 8 -> 606
    j .final

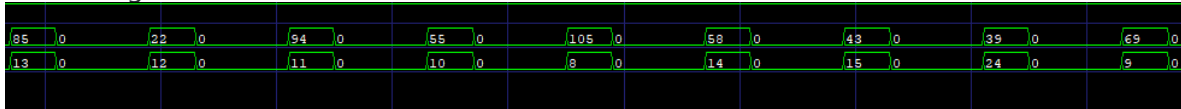
.salto2:
    addi $t0, $t0, -7 #reg 8 -> 1300
    j .salto1

.final:
    addi $t0, $t0, -605 # reg 8 -> 1
    sw $t0, 0($zero) # mem[0] -> 1
```

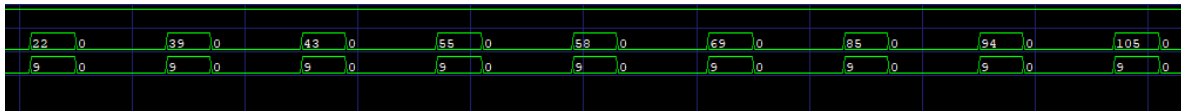
En esta tercera fase se comprobó que estuviera funcionando el salto

Testbench. Ensamblador Selective Sort

Valores ingresados: 85, 22, 94, 55, 105, 58, 43, 39, 69

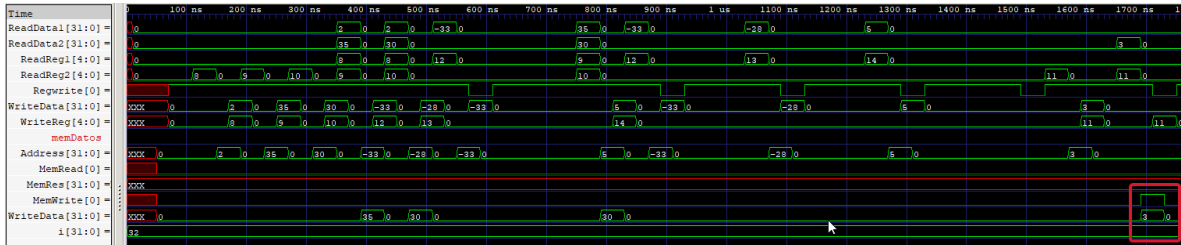


Resultados -> 22, 39, 43, 55, 58, 69, 85, 94, 105

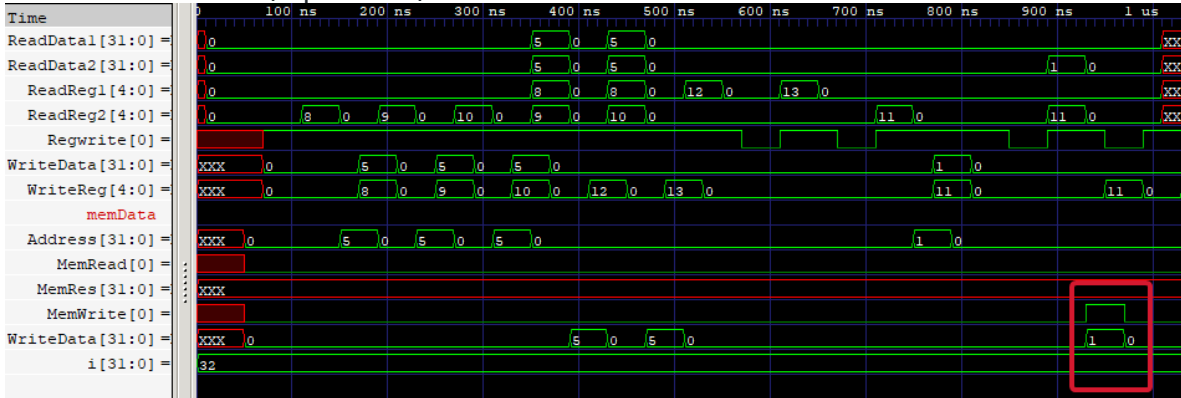


Testbench. Ensamblador Tipos de Triángulos

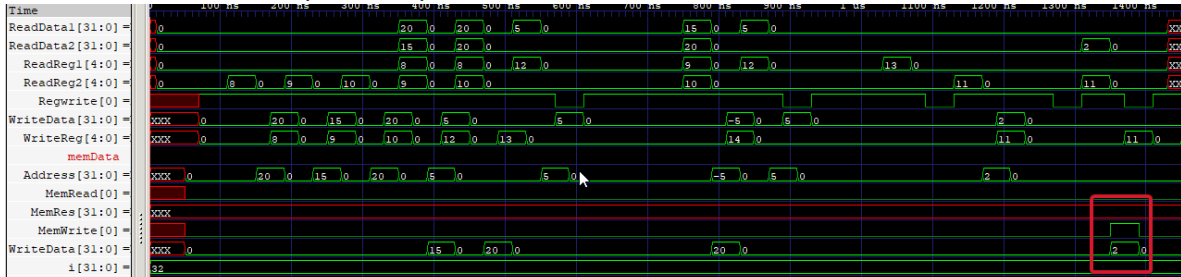
A = 2, B = 35, C = 30 (Escaleno)



A = 5, B = 5, C = 5 (Equilátero)



A = 20, B = 20, C = 5 (Isóceles)



DECODIFICAODR

El objetivo del decodificador es convertir las instrucciones de ensamblador al código de instrucciones el cual puede ser leído por el MIPS.

Las características que más definen al decodificador son:

- Soporte de labels o etiquetas
- Generar automáticamente los NOPS requeridos por las instrucciones para tener una ejecución correcta
- Detección de errores de escritura (parámetros, acceder a labels incorrectos, etc)
- Pseudo instrucciones para hacer más simple la escritura del código en ensamblador

Nuestro decodificador fue escrito en python, en 2 archivos, uno que es el principal y otro en el cual se declaran listas y diccionarios.

El archivo secundario es Diccionarios.py, en este archivo se almacenan los diccionarios y listas con las instrucciones y registros válidos, también una función para checar el contenido de estos registros.

```
registros = {  
    "$zero": [0, 0],  
    "$one": [1, 0],  
    "$t0": [8, 1],  
    "$t1": [9, 1],  
    "$t2": [10, 1],  
    "$t3": [11, 1],  
    "$t4": [12, 1],  
    "$t5": [13, 1],  
    .  
    .  
    .  
}
```

Archivo principal (Decoder.py) es el importante, es el que realiza todo el proceso de conversión a instrucciones del MIPS.

Está dividido en 6 funciones

- Cargado del archivo
- Eliminación de líneas sin código
- Mover los parámetros de ensamblador a los parámetros de instrucción y añadir NOPS
- Cálculo de posiciones de branch/jump
- Convertir los números con \$ a enteros y los registros también
- Convertir a binario y generar archivo de instrucciones

Cargado del archivo

```
• def selectfile():
•     global cfile
•     fName = str(input("Nombre del archivo: "))
•     print("")
•
•     try:
•         cfile = open(fName, 'r')
•     except FileNotFoundError:
•         print("> [ERROR] No se encontró el archivo indicado, ¿está en la
•             misma carpeta?")
•         exit()
```

Esta función simplemente pide el archivo de manera relativa el cual debe de cargar, no tiene mucha ciencia.

Eliminación de líneas sin código

```
• def torealcode():
•     global rlines
•
•     lines = cfile.readlines()
•     cline = 1
•
•     for line in lines:
•         strippedline = list(line.strip())
•
•         if strippedline:
•             while(", " in strippedline): # Remover las comas de la linea
•                 strippedline.remove(", ")
•                 strippedline = "".join(strippedline)
•                 rlines.append([strippedline, cline])
•
•     cline += 1
```

En esta función se leen todas las líneas del código, se evitan las líneas que no tienen contenido y después se eliminan las , que existen para tener una lectura más simple, ya que se eliminan se une y se añade a una lista el cual contiene el código ya retocado y la línea en la que está en el archivo.

Convertir parámetros

Esta función es una de las más complicadas y tiene bastante contenido. El código tiene comentarios le cual explica prácticamente que hace línea por línea así que se explicará lo que no podría haber quedado aquí.

addinstruction

En esta función hay una parte donde se utiliza frecuentemente la siguiente función.

```
• def addinstruction(linex, params, nops=1):
•     scriptlines_1.append([linex, params])
•     for __ in range(nops):
•         scriptlines_1.append([linex, ["nop"]])
```

Esta función se encarga de agregar a una lista la instrucción con los parámetros ya ordenados y añade la cantidad de NOPS que necesita esta función para funcionar correctamente, también guarda la posición de la línea de código original para informar si hay un error.

LW y SW

También se contempla un caso especial en la escritura del ensamblador para las instrucciones sw y lw, las cuales se escriben de la siguiente manera sw \$reg decimal(\$reg). Para no complicar el asunto usando splits se decidió utilizar una expresión regular para ver el contenido.

```
•     if rpam[0] == "sw":
•         reg = re.match(r"(\d+?)\(\$(\w+?)\)", rpam[2])
•         if reg:
•
•             arg2 = str(reg.group(1))
•             arg1 = str(reg.group(2))
•             addinstruction(linex[1], ["sw", "$" + arg1, rpam[1], arg2])
•         else:
•
•             print("> [ERROR] No cumple con los parámetros esperados en la
• línea %i (esperados: %s)" % (linex[1], dic.funcs[rpam[0]]))
•             exit()
```

Por ejemplo el SW. Primero hacemos un análisis en regex con `(\d+?)\(\$(\w+?)\)` la cual busca que cumpla con decimal(signodedinero+palabra) y almacena los datos en grupos, si no se encuentra de esa manera entonces hay un error de escritura y se termina la decodificación.

Cálculo de posiciones

```
• def calculatepositions():
•     global scriptlines_2
•
•     PC = -1
•
•     for line in scriptlines_1:
•         PC += 1
```

```

•
•     fpam = line[1][0]
•
•     if(fpam[-1] == ":"):
•         scriptlines_2.append([line[0], ["nop"]])
•         jpositions.append([PC, fpam])
•
•     else:
•         if fpam == "beq":
•             scriptlines_2.append(line)
•             scriptlines_2.append([line[0], ["nop"]])
•             jpositions.append([PC, fpam, line[1][3], 0])
•         else:
•             scriptlines_2.append(line)

```

Esta función se encarga de calcular las posiciones de los labels y almacenarlas en una lista para que en la función siguiente convierta el texto al entero que corresponde a esa línea.

En realidad los labels son interpretados como NOPS en el MIPS así de esta manera evitamos más problemas.

Conversión de palabras a enteros.

En esta sección se convierten todo lo que sea una palabra a un entero, por ejemplo los registros, los labels y algunas constantes que empiezan con \$

Para las constantes se utiliza una expresión regular que es la siguiente (^-?\d+(\.\d+)?\$?) lo cual busca números positivos o negativos que inicien con un \$.

Para la conversión de los labels a números está esta sección del código

```

•     if dic.checkKey(pam, dic.registros):
•         templist.append(dic.registros[pam][0])
•     else:
•         if pam[0] == ".": # Ajustar posiciones de salto/branching
•             for zon in jpositions:
•                 if pam == zon[1][-1]:
•                     if line[1][0] == "beq":
•                         for zonx in jpositions:
•                             if zonx[1] == "beq":
•                                 if len(zonx) > 3:
•                                     if zonx[3] == 0:
•                                         zonx[3] = 1
•                                         jpositions[jpositions.index(zonx)] =
zonx
•                                     templist.append(zon[0] - zonx[0])
•                                     break

```



```

•         else:
•             templist.append(zon[0])
•             break
•     else:
•         print("> [ERROR] Ingresaste un parámetro inexistente en la línea
•         %i (>%s) (esperados: %s)" % (line[0], pam, dic.funcs[line[1][0]]))
•         exit()

```

Se revisa que el parámetro inicie con un . (esta forma la elegimos para que sea mucho más sencillo identificar los labels) y si no entonces ya hay algo mal escrito y se informa al usuario. Checa si es una instrucción de tipo branch, si lo es entonces hace el cálculo de las posiciones que hay de diferencia `templist.append(zon[0] - zonx[0])` , si no es un branch entonces es un salto así que añade directamente el número.

Conversión a binario y guardado

Finalmente está la conversión a binario y su guardado, esta función lee todas las líneas que ya fueron procesadas y las convierte a binario. Dependiendo del tipo de instrucciones como será procesada ya que tenemos que considerar el tamaño de bits por si se utilizan números negativos.

En esta parte también se puede detectar otro error, si existe una label inexistente ya que jamás fue convertida a un número entero por no estar declarada.

Una vez terminado el proceso se convierte a binario utilizando la siguiente función que soporta tanto positivos como negativos.

```

• def dbin(number, clen, c2=0):
•     if number < 0:
•         return bin(number % (1 << clen))[2::]
•     else:
•         return "0" * (clen - len(bin(number % (1 << clen))[2::])) +
•         bin(number % (1 << clen))[2::]

```

Esta función detecta si es positivo o negativo ya que dependiendo de esto se añaden los bits sobrantes como 1 o un 0.

Y una vez convertido todo es guardado en un archivo llamado instr.mem

Objetivo

Se busca simular y demostrar en el proceso la importancia que tiene un procesador en una computadora, aprender sobre su arquitectura y desenmarañar las instrucciones y procesos que se pueden realizar mediante esta tecnología, desarrollando un procesador básico de 32 bits y ejemplificando su funcionamiento mediante instrucciones de tipo I, R, y J.

Conclusión

El procesador es la parte más importante de un ordenador. Es el cerebro, coordina todos los datos, todas las aplicaciones, y todos los procesos que pasan por el procesador. Los dos tipos de procesador más utilizados son Intel y AMD, pero no son los únicos. Se logró aprender a lo largo del semestre la arquitectura para después implementar nuestro propio procesador dummy, logrando trabajar en equipo y apoyándonos mutuamente entre todos, teniendo como base los aprendizajes y el apoyo del profesor.

Bibliografía

NEC Corporation. (2004). NEC Offers Two High Cost Performance 64-bit RISC Microprocessors. mayo 2020, de NEC Corporation Sitio web: <http://www.nec.co.jp/press/en/9801/2002.html>

Drarf Fortress. (Julio 2015). La Sonda New Horizons tiene un procesador idéntico al de la primera PlayStation. Mayo 2020, de Machine Sitio web: <https://web.archive.org/web/20150719075343/http://www.sumagamer.com/noticias/new-horizons-mismo-procesador-playstation/>

Intel. (2000). The MIPS32® Instruction Set. En MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual(475). EE.UU: -.

David A. Patterson / John L. Hennessy. (2014). COMPUTER ORGANIZATION AND DESIGN. -: FIFTH EDITION.

Se encontrará el desarrollo en GitHub: <https://github.com/kwattt/MIPS-teams>