Universidad de Guadalajara

CUCEI

Seminario de Solución de Problemas de Arquitectura de Computadoras.

D14

Profesor: López Arce Delgado Jorge Ernesto.

Proyecto final

Fase 2

Integrantes:

- Rubio Andrade Athziri Magdalena.
- Casas Chavarría Diego Maximiliano.
- García Ramírez José Manuel.

Actividades a realizar

1. *Código Verilog* - Implementación del “single datapath” de MIPS de 32 bits para ejecutar instrucciones tipo I.

2. *Reporte* - Redacción y descripción del desarrollo de los módulos que tienen el datapath y los elementos nuevos para instrucciones tipo I.

3. *Programa ensamblador* - Investigar y crear propuesta de algoritmo a implementar para evaluación y posible aceptación por parte del profesor.

4. *Presentación* *(subirla a Moodle)*, esta debe ser un resumen de los tres puntos anteriores, se debe de mostrar que han investigado de la parte teórica del algoritmo a implementar en ensamblador, que han codificado en esta fase 2 así como la validación de el datapath para dicha fase.

Bitácora (días fase 2):

- 28 de Mayo del 2021
- 31 de Mayo del 2021
- 1 y 2 de Junio del 2021
- 03 de Junio del 2021
- 04 de Junio del 2021
- 5, 6 y 7 de Junio del 2021
- 10 de Junio del 2021

28 de Mayo de 2021

- Horario:

  N/A

- Descripción:

  Este dia fue la presentación de nuestra Fase 1 del proyecto, Después se platico de los errores que tuvimos en el código de verilog y cómo solucionarlos, Tambien platicamos de los errores del reporte y la extensión de nuestra presentación.

31 de Mayo de 2021

- Horario:

  Todo el día

- Descripción:

  Se soluciono un warning que había en el módulo de BankReg, ya que anteriormente nos causaba mucho conflicto y no supimos arreglarlo, El unico problema era la condicional if estaba hasta el último y no se estaba indicando que se tenía que escribir antes de leer.

  Tambien este dia se realizo el módulo del primer buffer.

1 y 2 de Junio de 2021

- Horario:

  Todo el día

- Descripción:

  Estos días fueron solo de investigación sobre los nuevos módulos e instrucciones, Nuestros horarios escolares nos complica mucho conectarnos para aclarar y ver nuestros adelantos de cada uno.

03 de Junio de 2021

- Horario:

  De 7:30 pm a 01:00 am

- Descripción:

  Este dia fue la primera vez que nos juntamos en llamada, Intentamos con discord pero Ahtziri tuvo muchos problemas para transmitir así que decidimos irnos a Google meet, En esta llamada se inició las conexiones de todos los módulos en el Datapath, Cambiamos los nombres de cada cable por sus nombres definidos en ves de “C1, C2, C3”...

04 de Junio de 2021

- Horario:

  Hora clase.

- Descripción:

  Este dia fue la clase de la explicación de algunas cosas que quedaron pendientes por falta de tiempo en la anterior clase, Se definió el algoritmo que hará nuestro lenguaje en ensamblador para que lo realice nuestro código.

5, 6 y 7 de Junio de 2021

- Horario:

  Todo el dia.

- Descripción:

  Se continuó arduamente con el código para dejarlo sin errores para darle prioridad al lenguaje en ensamblador y el algoritmo factorial ya definido el 04 de junio.

10 de Junio de 2021

- Horario:

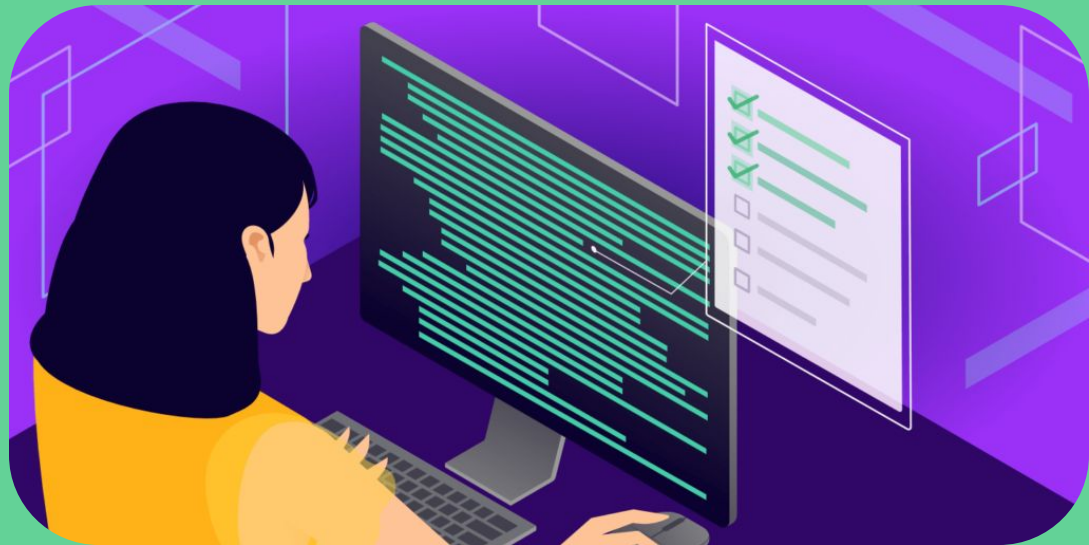
  Todo el dia.

- Descripción:

  Este dia nos logramos conectar en la tarde para por fin terminar el código verilog con algunas dudas, Las conexiones y lograr terminar el lenguaje ensamblador con el algoritmo factorial y arreglar muchos errores que aun había, modificar el texto de la memoria de instrucciones para que de los resultados esperados.

Código verilog

nuevos módulos.



ALU

- X(entrada de 32 bits)
- Y(entrada de 32 bits)
- SEL(entrada de 3 bits)
- R(salida de 32 bits)
- Z_flag(salida de 1 bit)

Se agregaron las operaciones correspondientes a la fase 2, en este caso las de tipo I solicitadas (LW, SW, BEQ, ADDI, ANDI, ORI)

```
1  `timescale 1ns/1ns
2
3  module ALU(
4      input [31:0] X,
5      input [31:0] Y,
6      input [2:0] SEL,
7      output reg [31:0] R,
8      output reg Z_flag
9  );
10
11 always@* begin
12     case(SEL)
13     //R operando destino, X operando fuente 1,Y operando fuente 2
14         3'd0: R = X + Y; //ADD ADDI LW SW
15         3'd1: R = X - Y; //SUB BEQ
16         3'd2: R = X & Y; //AND ANDI
17         3'd3: R = X | Y; //OR ORI
18         3'd4: R = X < Y; //SLT SLTI
19         3'd5: R = X * Y; //Mul
20         3'd6: R = X / Y; //Div
21         3'd7: R = X << 0; //NOP
22         default: R <= 32'bx;
23     endcase
24
25     Z_flag <=(R) ? 0:1;
26
27 end
28 endmodule
```

ALU Control

- FUNCTION(entrada de 6 bits)
- ALUOP(entrada de 3 bits)
- OP(salida de 3 bits)

Se agregaron las opciones para ejecutar las operaciones correspondientes a la fase 2, en este caso las de tipo I solicitadas (LW, SW, BEQ, ADDI, ANDI, ORI)

```
1  `timescale 1ns/1ns
2
3  module alu_ctrl(
4      input[5:0] FUNCTION,
5      input[2:0] ALUOP,
6      output reg [2:0] OP
7  );
8
9      always @(*) begin
10         case (ALUOP)
11             3'b010: begin
12                 case (FUNCTION)
13                 6'b100000: OP = 3'd0; //ADD
14                 6'b100010: OP = 3'd1; //SUB
15                 6'b100100: OP = 3'd2; //AND
16                 6'b100101: OP = 3'd3; //OR
17                 6'b101010: OP = 3'd4; //slt
18                 6'b011000: OP = 3'd5; //mul
19                 6'b011010: OP = 3'd6; //div
20                 6'b000000: OP = 3'd7; //nop
21                 endcase
22             end
23             3'b000: OP = 3'd0; //addi...
24             3'b001: OP = 3'd4; //slti...
25             3'b100: OP = 3'd2; //andi...
26             3'b011: OP = 3'd3; //ori...
27         endcase
28     end
29 endmodule
```

Unidad de Control

Se agregaron las opciones para ejecutar las operaciones correspondientes a la fase 2, en este caso las de tipo I solicitadas (LW, SW, BEQ, ADDI, ANDI, ORI) con sus respectivas salidas hacia buffers.

```
3  module unit_control(
4      input[5:0] OPCODE,
5      output reg MemREG, RegWRITE, MemWRITE
6      output reg Branch, MemRead, ALUSrc, F
7      output reg [2:0] ALUOP
8  );
9
10     always @* begin
11         case (OPCODE)
12             6'b000000: begin //TIPO R
13                 MemREG = 0;
14                 RegWRITE = 1;
15                 MemWRITE = 0;
16                 Branch = 0;
17                 MemRead = 0;
18                 ALUSrc = 0;
19                 RegDst = 1;
20                 ALUOP = 3'b010;
21             end
22             6'b100011: begin //lw
23                 MemREG = 1;
24                 RegWRITE = 1;
25                 MemWRITE = 0;
26                 Branch = 0;
27                 MemRead = 1;
28                 ALUSrc = 1;
29                 RegDst = 0;
30                 ALUOP = 3'b000;
31             end
```

```
6'b101011: begin //sw
    MemREG = 1'bx;
    RegWRITE = 0;
    MemWRITE = 1;
    Branch = 0;
    MemRead = 0;
    ALUSrc = 1;
    RegDst = 1'bx;
    ALUOP = 3'b000;
end
6'b000100: begin //beq
    MemREG = 1'bx;
    RegWRITE = 0;
    MemWRITE = 0;
    Branch = 1;
    MemRead = 0;
    ALUSrc = 0;
    RegDst = 1'bx;
    ALUOP = 3'b000;
end
6'b001000: begin //ADDI
    MemREG = 0;
    RegWRITE = 1;
    MemWRITE = 0;
    Branch = 0;
    MemRead = 0;
    ALUSrc = 1; //1 TIPO I
    RegDst = 1;
    ALUOP = 3'b000; //dependende de cada op
end
6'b001100: begin //ANDI
    MemREG = 0;
    RegWRITE = 1;
    MemWRITE = 0;
```

Buffers

Se elaboraron 4 buffers, con la descripción IF/ID, ID/EX, EX/MEM y MEM/WB, los cuales están en ese orden y en un cierto acomodo de cables para llevar a cabo la técnica conocida como “pipeline” que nos ayuda a optimizar tiempos en relación a las operaciones a realizar.

```
4  module buffer1
5      (
6      input clk,
7      input [31:0] inputAdd,
8      input [31:0] inputInsMem,
9      output reg [31:0] outputAdd,
10     output reg [31:0] OutputInsMem
11     );
12     always @(posedge clk)
13     begin
14         outputAdd = inputAdd;
15         OutputInsMem = inputInsMem;
16     end
17 endmodule
```

Buffer 2

```
module buffer2
    (
    //UNIDAD DE CONTROL
    //ENTRADAS
    //WB
    input INB2MemREG,
    input INB2RegWRITE,
    //M
    input INB2Branch,
    input INB2MemWRITE,
    input INB2MemRead,
    //EX
    input INB2RegDst,
    input [2:0]INB2ALUOP,
    input INB2ALUSrc,
    //RESTO
    input clk,
    input [31:0] inputAddB2,
    input [31:0] inputRD1,
    input [31:0] inputRD2,
    input [31:0] inputSinex,
    input [4:0] inputRr1,
    input [4:0] inputRr2,
    //UNIDAD DE CONTROL
    //SALIDAS
    //WB
    output reg OTB2MemREG,
    output reg OTB2RegWRITE,
    //M
    output reg OTB2Branch,
    output reg OTB2MemWRITE,
    output reg OTB2MemRead,
    output reg OTB2RegDst,
    output reg [2:0]OTB2ALUOP,
    output reg OTB2ALUSrc,
    //RESTO
    output reg [31:0] outputAddB2,
    output reg [31:0] outputRD1,
    output reg [31:0] outputRD2,
    output reg [31:0] outputSinex,
    output reg [4:0] outputRr1,
    output reg [4:0] outputRr2
    );

    always @(posedge clk)
    begin
        OTB2MemREG = INB2MemREG;
        OTB2RegWRITE = INB2RegWRITE;
        OTB2Branch = INB2Branch;
        OTB2MemWRITE = INB2MemWRITE;
        OTB2MemRead = INB2MemRead;
        OTB2RegDst = INB2RegDst;
        OTB2ALUOP = INB2ALUOP;
        OTB2ALUSrc = INB2ALUSrc;
        outputAddB2 = inputAddB2;
        outputRD1 = inputRD1;
        outputRD2 = inputRD2;
        outputSinex = inputSinex;
        outputRr1 = inputRr1;
        outputRr2 = inputRr2;
    end
endmodule
```

Buffer 3

```
module buffer3
    (
    //UNIDAD DE CONTROL
    //ENTRADAS
    //WB
    input INB3MemREG,
    input INB3RegWRITE,
    //M
    input INB3Branch,
    input INB3MemWRITE,
    input INB3MemRead,
    //RESTO
    input clk,
    input [31:0] inputAddB3,
    input Z_flag,
    input [31:0] inputAddrst,
    input [31:0] inputRD2B3,
    input [4:0] inputmux,
    //UNIDAD DE CONTROL
    //SALIDAS
    //WB
    output reg OTB3MemREG,
    output reg OTB3RegWRITE,
    //M
    output reg OTB3Branch,
    output reg OTB3MemWRITE,
    output reg OTB3MemRead,
    //RESTO
    output reg [31:0] outputAddB3,
    output reg [31:0] outputAddrst,
    output reg [31:0] outputRD2B3,
    output reg output_Z_flag,
    output reg [4:0] outputmux
```

```
    );

    always @(posedge clk)
    begin
        OTB3MemREG = INB3MemREG;
        OTB3RegWRITE = INB3RegWRITE;
        OTB3Branch = INB3Branch;
        OTB3MemWRITE = INB3MemWRITE;
        OTB3MemRead = INB3MemRead;
        outputAddB3 = inputAddB3;
        outputAddrst = inputAddrst;
        outputRD2B3 = inputRD2B3;
        outputmux = inputmux;
    end
endmodule
```

Buffer 4

```
module buffer4
    (
    //UNIDAD DE CONTROL
    //ENTRADAS
    //WB
    input INB4MemREG,
    input INB4RegWRITE,
    //RESTO
    input clk,
    input [31:0] inputDatmem,
    input [31:0] inputAddrstB4,
    input [4:0] inputmuxB4,
    //UNIDAD DE CONTROL
    //SALIDAS
    //WB
    output reg OTB4MemREG,
    output reg OTB4RegWRITE,
    //RESTO
    output reg [31:0] outputDatmem,
    output reg [31:0] outputAddrstB4,
    output reg [4:0] outputmuxB4
    );
    always @(posedge clk)
    begin
        OTB4MemREG = INB4MemREG;
        OTB4RegWRITE = INB4RegWRITE;
        outputDatmem = inputDatmem;
        outputAddrstB4 = inputAddrstB4;
        outputmuxB4 = inputmuxB4;
    end
endmodule
```

Datapath

- clk (reg)

Se reorganizaron los cables conectando buffers y módulos entre sí.

```
module singledptR(
    input clk

);

    //BUFFER 1 IF/ID
    wire [31:0] inputAdd;
    wire [31:0] inputInsMem;
    wire [31:0] outputAdd;
    wire [31:0] OutputInsMem;
    //BUFFER 2 ID/EX
    //UNIDAD DE CONTROL
    //ENTRADAS
    //WB
    wire INB2MemREG;
    wire INB2RegWRITE;
    //M
    wire INB2Branch;
    wire INB2MemWRITE;
    wire INB2MemRead;
    //EX
    wire INB2RegDst;
    wire [2:0]INB2ALUOP;
    wire INB2ALUSrc;
    //RESTO
    wire [31:0] inputRD1;
    wire [31:0] inputRD2;
    wire [31:0] inputSinex;
    //UNIDAD DE CONTROL
    //SALIDAS
    //WB
    wire OTB2MemREG;
    wire OTB2RegWRITE;
    //M
    wire OTB2Branch;
    wire OTB2MemWRITE;
    wire OTB2MemRead;
    //EX
    wire OTB2RegDst;
    wire [2:0]OTB2ALUOP;
    wire OTB2ALUSrc;
    //RESTO
    wire [31:0] outputAddB2;
    wire [31:0] outputRD1;
    wire [31:0] outputRD2;
    wire [31:0] outputSinex;
    wire [4:0] outputRr1;
    wire [4:0] outputRr2;
    //BUFFER 3 EX/MEM
    //RESTO
    wire [31:0] inputAddB3;
    wire Z_flag;
    wire [31:0] inputAddrst;
    wire [4:0] inputmux;
    //UNIDAD DE CONTROL
    //SALIDAS
    //WB
    wire OTB3MemREG;
    wire OTB3RegWRITE;
    //M
    wire OTB3Branch;
    wire OTB3MemWRITE;
    wire OTB3MemRead;
    //RESTO
    wire [31:0] outputAddB3;
    wire [31:0] outputAddrst;
    wire [31:0] outputRD2B3;
```

Test Bench

- clk (reg)

Se inicializa el “reloj”, que va a hacer que el single datapath entre en un bucle hasta el tiempo que le demos.

Se modifico los tiempos para los pulsos de reloj y que entren todas las instrucciones y operaciones.

```
3   module tbsingled();
4       reg clk;
5
6   singledptR datapath(clk);
7
8   always
9
10      #50 clk = ~ clk;
11
12  initial
13  begin
14      clk <= 0;
15      #50000
16      $stop;
17
18  end
19  endmodule
20
```

Investigación

Instrucciones de tipo “I”

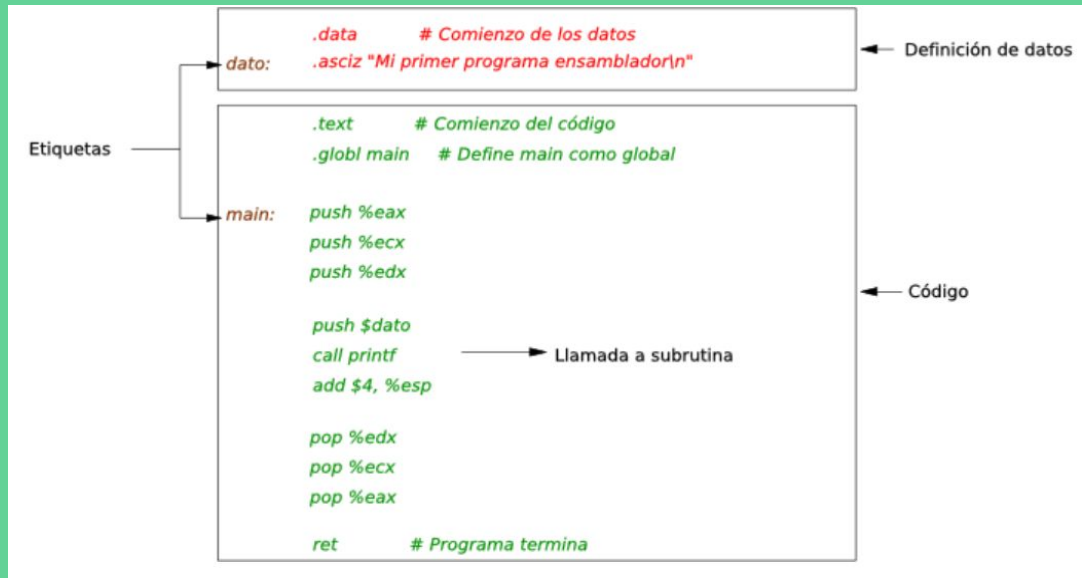
- Se divide en las siguientes secciones:
- · Bits 31-26:
- En estos primeros seis bits encontramos el espacio para el código de operación que identifica a cada instrucción, en este caso por ser el tipo de instrucción I será el código correspondiente a LW, SW, BEQ,SLTI,ADDI,ORI,ANDI
- · Bits 25-21:
- Los siguientes cinco bits son los destinados a el registro base.
- · Bits 20-16:
- Los siguientes cinco bits son los que representan al registro destino.
- · Bits 15-0:
- Estos últimos dieciséis bits son los destinados al apartado conocido como “desplazamiento”.

Set de instrucciones

Instrucción	Tipo	Sintaxis
Addi	I	addi $rt $rs $inmediate
Subi	I	subi $rt $rs $inmediate
Ori	I	ori $rt $rs $inmediate
Andi	I	andi $rt $rs $inmediate
Lw	I	lw $rt #offset(base)
Sw	I	sw $rt #offset(base)

Bgtz	I	bgtz $rs $etiqueta
Beql	I	beql $rs $rt #offset(base)
Slti	I	slti $rt $rs $inmediate
Beq	I	beq $rs $rt #offset(base)
Bne	I	bne $rs $rt #offset(base)

Programa ensamblador



Lenguaje ensamblador

El lenguaje ensamblador trabaja con nemónicos, que son grupos de caracteres alfanuméricos que simbolizan las órdenes o tareas a realizar. La traducción de los nemónicos a código máquina entendible por el microcontrolador la lleva a cabo un programa ensamblador. El programa escrito en lenguaje ensamblador se denomina código fuente (*.asm). El programa ensamblador proporciona a partir de este fichero el correspondiente código máquina, que suele tener la extensión *.hex.

Propuesta de algoritmo

```
#Calculara un numero factorial
#Algoritmo en lenguaje ensamblador

.data
uno:     .word 1        # uno inicializara un registrador en el valor uno
cadena:  .ascii         "Favor ingrese numero a calcular factorial: "
line:    .asciiz        "\n"    # line tomara un salto de línea

.text
.globl main

main:

lw $s1 uno              # inicializa registrador $s1 en uno (1)
lw $s2 uno              # inicializa registrador $s2 en uno (1)

la $a0 cadena           # carga cadena en $a0
li $v0 4                # carga $v0 a 4: permite mostra una cadena
syscall

li $v0 5                # carga $v0 a 5: permite leer un entero
syscall
move $t0 $v0
move $s0 $t0

beq $s0 $zero printCero # el factorial de cero es 1


loop:
```

```
loop:

beq $s0 $s1 print
mul $s2 $s2 $s0         # $s2 toma el valor del producto entre $s2 y $s0
sub $s0 $s0 $s1         # $s0 se decrementa para continuar con el ciclo
j loop                  # jump: salto

print:

la $a0 line             # carga salto de linea en $a0 para imprimirla
li $v0 4
syscall

move $a0 $s2
li $v0 1                # carga $v0 a 1: permite mostrar un entero
syscall
j fin

printCero:

la $a0 line             # carga salto de linea en $a0 para imprimirla
li $v0 4                # carga $v0 a 4: permite mostrar una cadena
syscall

move $a0 $s1
li $v0 1                # carga $v0 a 1: permite mostrar un entero
syscall

fin:

li $v0 10
syscall                 # salir
```