LAB 8- ARCH - 2020-I

Integrantes: Códigos:

- Calixto Rojas, Neftali 201910092
- Lapa Romero, Julisa 201910200

1. Indicaciones:

1. Fecha de entrega: 21 de Julio.
2. Entregables: Informe y archivos todo en un comprimido '`.tar`'.

2. Resumen de Contenidos:

Este informe corresponde al Proyecto Final del curso de Arquitectura de Computadores, consta de tres partes, MIPS Single-Cycle Processor, MIPS Multi-Cycle Processor 1 y MIPS Multi-Cycle Processor 2.

Esta es la segunda parte, donde diseñaremos y construiremos nuestro propio MIPS Multi-Cycle Processor(basado en el diseño de el Harris Harris). Para este laboratorio se podrá reusar todo hardware(módulos de Verilog) de previos laboratorios que se quieran.

El procesador multicycle a realizar debe coincidir con el diseño del libro, diseño que se muestra como Figura 1. Debe manejar las siguientes instrucciones: `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`, `addi` y `j`. El procesador multicycle se divide en tres unidades, el `controller datapath`, y las unidades `mem`(memory). Debemos notar que las unidades `mem` contendrán la memoria compartida usada para almacenar tanto datos como instrucciones. También, notar que la unidad `controller` comprende tanto el `Main Decoder` que toma como inputs a $OP_{5:0}$ y el `ALU Decoder` que toma com inputs al $ALUOp_{1:0}$ y el código $Funct_{5:0}$ de los 6 bits menos significantes de una instrucción. La unidad `controller` también debe incluir las compuertas necesarias para producir el la señal del write enable, `PCEn`, para el registro PC. El testbench del controller será diseñado por nosotros.

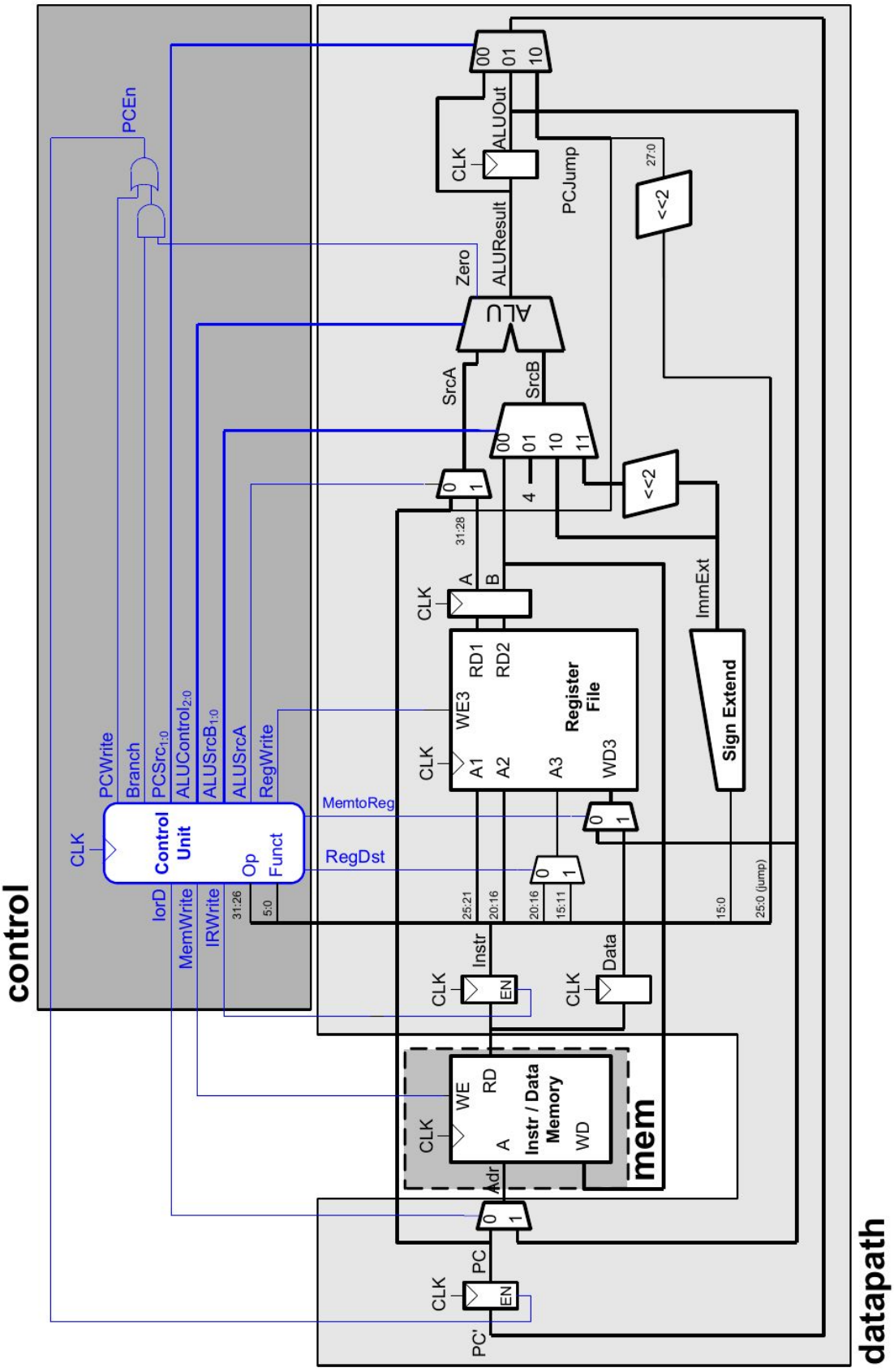


Figure 1. Multicycle Processor

3. MIPS Multi-Cycle Processor:

UNIT OVERVIEW:

Las unidades que comprenden el procesador tienen los siguientes inputs y outputs. (Las tablas tienen los nombres en mayúsculas para coincidir con el esquemático, sin embargo en el código de Verilog estos están en minúscula).

CLK		Input
Reset		Input
Op	[5:0]	Input
Funct	[5:0]	Input
Zero		Input
IorD		Output
MemWrite		Output
IRWrite		Output
RegDst		Output
MemtoReg		Output
RegWrite		Output
ALUSrcA		Output
ALUSrcB	[1:0]	Output
ALUControl	[2:0]	Output
PCSrc	[1:0]	Output
PCEn		Output

Table 1. Controller

CLK		Input
Reset		Input
PCEn		Input
IorD		Input
IRWrite		Input
RegDst		Input
MemtoReg		Input
RegWrite		Input
ALUSrcA		Input
ALUSrcB	[1:0]	Input
ALUControl	[2:0]	Input
PCSrc	[1:0]	Input
ReadData	[31:0]	Input
Op	[5:0]	Output
Funct	[5:0]	Output
Zero		Output
Adr	[5:0]	Output
WriteData	[31:0]	Output

Table 2. Datapath

Notar que *PCWrite* y *Branch* son señales internas (wires) dentro el controller.

CLK		Input
Reset		Input
MemWrite		Input
Adr	[5:0]	Input
WriteData	[31:0]	Input
ReadData	[31:0]	Output

Table 3. Memory (mem)

4. Generando Señales de Control:

Antes de empezar a desarrollar el hardware para nuestro MIPS multi-cycle processor, tendremos que determinar las correctas control signals para cada estado en el diagrama de transición de estados del procesador multi-cycle. El diagrama se muestra en la Figura 7.42 del libro. Completar la tabla de outputs del Main Decoder en la Tabla 4. Escriba la FSM control word en hexadecimal para cada estado. Las primeras dos filas están completadas como ejemplos. Debemos ser cuidadosos aquí , pues el tiempo de debug de un circuito erróneo es mucho más largo que el diseñarlo correctamente al inicio.

5. Diseño General:

Ahora se empezará la implementación hardware de nuestro procesador multicycle. Primero se copiará el mipsmulti.sv a nuestro directorio y se le llamará `mipsmulti_xx.v` y traducirlo a Verilog.

El módulo `mips` instancia el `datapath` y el control unit(módulo `controller`). El módulo controller a su vez instancia el módulo main decoder (`maindec`) y el ALU Decoder(`aludec`). Diseñaremos el controller en este laboratorio. El siguiente laboratorio se dedicará al datapath.

6. Diseño del Control Unit:

El control unit es la parte más compleja del procesador multicycle. Este consiste de dos módulos, el Main Decoder y el ALU Decoder. El Main Decoder, `maindec`, debe tomar la entrada del Opcode y producir el output descrito en la Tabla 4. En reset, el control unit debe empezar en el estado 0. El control unit debe admitir las instrucciones de la Figura 7.42 en el texto.

Diseñaremos el controller usando un FSM para el Main Decoder y el combinational logic para el ALU Decoder. También incluye cualquier lógica adicional necesaria para calcular el *PCEn* de las señales internas *PCWrite*, *Branch*, y *Zero*. Los encabezados `controller`, `maindec`, y `aludec` muestran los inputs y outputs para cada módulo.

Una porción del código de Verilog para el control unit se nos dio, este se completará con el diseño del hardware del controller y sus submódulos.

Crearemos un `controllertest_xx` testbench para el módulo `controller`. Testea cada una de las instrucciones que el procesador debe admitir(`add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`, `addi`, y `j`). Debemos asegurarnos de testear los branches tomados y no tomados. Las entradas del `controller` son: `clk`, `Reset`, `OP`, `Funct`, y `Zero`. Nuestro testbench debe aplicar las entradas. Visualmente inspeccionar los estados y outputs para verificar que concuerden con lo predicho por la Tabla 4. También verificar que *PCEn* funcione correctamente. Si se encuentra cualquier error, revise el circuito y corrija los errores. Guarde una copia de los waveforms mostrando los inputs, estados, y control outputs, y *PCEn* de cada estado.

7. **Entregables:**

Presentar los siguientes elementos en el siguiente orden. Detallar el orden.

- Una versión completa de la Tabla de Salidas del Main Decoder (Tabla 4).
- El código en Verilog de los módulos `controller`, `maindec` y `aludec`.
- El `controllertest_xx` testbench.
- Waveforms de la simulación del módulo controller, mostrando (en el orden dado): *CLK*, *Reset*, *OP*, *Funct*, *Zero*, y el *state* (esta es una señal registrada interna), *ALUControl*, *PCEn*, y todo el control word demostrando que cada instrucción (incluyendo los branches tomados y no tomados). Mostrar todas las señales en hexadecimal, **¿concuerdan con las predicciones?**

DESARROLLO:

1. Tabla 4: Tabla de Salidas del Main Decoder

State (Name)	PCWrite	MemWrite	IRWrite	RegWrite	ALUSrcA	Branch	IorD	MemtoReg	RegDst	ALUSrcB[1:0]	PCRsc[1:0]	ALUOp[1:0]	FSM Control Word
0 (Fetch)	1	0	1	0	0	0	0	0	0	01	00	00	0x5010
1 (Decode)	0	0	0	0	0	0	0	0	0	11	00	00	0x0030
2 (MemAdr)	0	0	0	0	0	1	0	0	0	01	00	00	0x0420
3 (MemRd)	0	0	0	0	0	0	0	1	0	00	00	00	0x0100
4 (MemWB)	0	0	1	0	0	0	0	1	0	00	00	00	0x0880
5 (MemWr)	0	0	0	0	1	0	0	0	1	00	00	00	0x2100
6 (RtypeEx)	0	0	0	0	1	0	0	0	0	00	00	10	0x0402
7 (RtypeWB)	0	0	0	1	0	0	0	0	1	00	00	00	0x0840
8 (BeqEx)	0	0	0	0	1	1	0	0	0	00	01	01	0x0605
9 (AddiEx)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
10 (AddiWB)	0	0	0	1	0	0	0	0	0	00	00	00	0x0800
11 (JEx)	1	0	0	0	0	0	0	0	0	00	10	00	0x4008

Table 4. Main Decoder Control output

2. CÓDIGOS EN VERILOG

➔ CONTROLLER:

```
module controller(input          clk, reset,
                  input   [5:0] op, funct,
                  input          zero,
                  output         pcen, memwrite, irwrite, regwrite,
                  output         alusrca, iord, memtoreg, regdst,
                  output  [1:0] alusrcb, pcsrc,
                  output  [3:0] alucontrol);

  wire [1:0] aluop;
  wire branch, pcwrite;

  // Main Decoder and ALU Decoder subunits.
  maindec md(clk, reset, op,
             pcwrite, memwrite, irwrite, regwrite,
             alusrca, branch, iord, memtoreg, regdst,
             alusrcb, pcsrc, aluop);
  aludec  ad(funct, aluop, alucontrol);

  // ADD CODE HERE
  // Add combinational logic (i.e. an assign statement)
  // to produce the PCEn signal (pcen) from the branch,
  // zero, and pcwrite signals
  assign pcen = pcwrite | (branch & zero);

endmodule
```

➔ MAINDEC:

```
module maindec(input         clk, reset,
               input  [5:0] op,
               output        pcwrite, memwrite, irwrite, regwrite,
               output        alusrca, branch, iord, memtoreg, regdst,
               output [1:0] alusrcb, pcsrc,
               output [1:0] aluop);

  parameter   FETCH   = 4'b0000; // State 0
  parameter   DECODE  = 4'b0001; // State 1
  parameter   MEMADR  = 4'b0010;    // State 2
  parameter   MEMRD   = 4'b0011;    // State 3
  parameter   MEMWB   = 4'b0100;    // State 4
  parameter   MEMWR   = 4'b0101;    // State 5
  parameter   RTYPEEX = 4'b0110;    // State 6
  parameter   RTYPEWB = 4'b0111;    // State 7
  parameter   BEQEX   = 4'b1000;    // State 8
  parameter   ADDIEX  = 4'b1001;    // State 9
  parameter   ADDIWB  = 4'b1010;    // state 10
  parameter   JEX     = 4'b1011;    // State 11

  parameter   LW      = 6'b100011;  // Opcode for lw
  parameter   SW      = 6'b101011;  // Opcode for sw
  parameter   RTYPE   = 6'b000000;  // Opcode for R-type
  parameter   BEQ     = 6'b000100;  // Ospcode for beq
  parameter   ADDI    = 6'b001000;  // Opcode for addi
  parameter   J       = 6'b000010;  // Opcode for j
reg [3:0]  state, nextstate;
reg [14:0] controls;

// state register
always @(posedge clk or posedge reset) begin
  if(reset) state <= FETCH;
  else state <= nextstate;
end
// ADD CODE HERE
// Finish entering the next state logic below.  We've completed the first
// two states, FETCH and DECODE, for you.
```

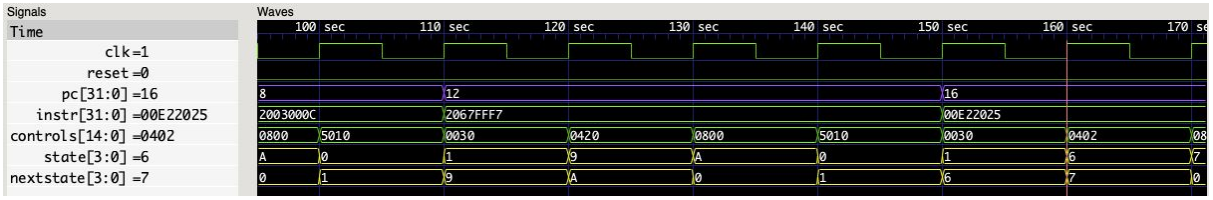
```
// next state logic
always@(*)
  case(state)
    FETCH:   nextstate <= DECODE;
    DECODE:  case(op)
               LW:        nextstate <= MEMADR;
               SW:        nextstate <= MEMADR;
               RTYPE:     nextstate <= RTYPEEX;
               BEQ:       nextstate <= BEQEX;
               ADDI:      nextstate <= ADDIEX;
               J:         nextstate <= JEX;
               default:   nextstate <= 4'bx; // should never happen
             endcase
      // Add code here
    MEMADR:   case(op)
            LW:       nextstate <= MEMRD;
            SW:       nextstate <= MEMWR;
            default:  nextstate <= 4'bx;
          endcase
    MEMRD:    nextstate <= MEMWB;
    MEMWB:    nextstate <= FETCH;
    MEMWR:    nextstate <= FETCH;
    RTYPEEX:  nextstate <= RTYPEWB;
    RTYPEWB:  nextstate <= FETCH;
    BEQEX:    nextstate <= FETCH;
    ADDIEX:   nextstate <= ADDIWB;
    ADDIWB:   nextstate <= FETCH;
    JEX:      nextstate <= FETCH;
    default:  nextstate <= 4'bx; // should never happen
  endcase
```

```
// output logic
  assign {pcwrite, memwrite, irwrite, regwrite,
          alusrca, branch, iord, memtoreg, regdst,
          alusrcb, pcsrc, aluop} = controls;

  // ADD CODE HERE
  // Finish entering the output logic below.  We've entered the
  // output logic for the first two states, S0 and S1, for you.
  always @(*)
    case(state)
      FETCH:   controls <= 15'h5010;
      DECODE:  controls <= 15'h0030;
      MEMADR:  controls <= 15'h0420;
      MEMRD:   controls <= 15'h0100;
      MEMWB:   controls <= 15'h0880;
      MEMWR:   controls <= 15'h2100;
      RTYPEEX: controls <= 15'h0402;
      RTYPEWB: controls <= 15'h0840;
      BEQEX:   controls <= 15'h0605;
      ADDIEX:  controls <= 15'h0420;
      ADDIWB:  controls <= 15'h0800;
      JEX:     controls <= 15'h4008;
      default: controls <= 15'hxxxx; // should never happen
    endcase
endmodule
```

➔ ALUDEC:

```
  // ADD CODE HERE
  // Complete the design for the ALU Decoder.
  // Your design goes here.  Remember that this is a combinational
  // module.

  // Remember that you may also reuse any code from previous labs.
module aludec(input   [5:0] funct,
              input   [1:0] aluop,
              output reg [3:0] alucontrol);

  always @* begin
    case(aluop)
      2'b00: alucontrol <= 4'b0000;  // add
      2'b01: alucontrol <= 4'b0010;  // sub
      2'b11: alucontrol <= 4'b0101;  // or

      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0000; // ADD
          6'b100010: alucontrol <= 4'b0010; // SUB
          6'b100100: alucontrol <= 4'b0100; // AND
          6'b100101: alucontrol <= 4'b0101; // OR
          6'b100111: alucontrol <= 4'b0111; // NOR
          6'b100110: alucontrol <= 4'b0110; // XOR
          6'b101010: alucontrol <= 4'b1010; // SLT
          default:   alucontrol <= 4'bxxxx; // ???
      endcase
    endcase
  end
endmodule
```

3. `controllertest_xx` testbench

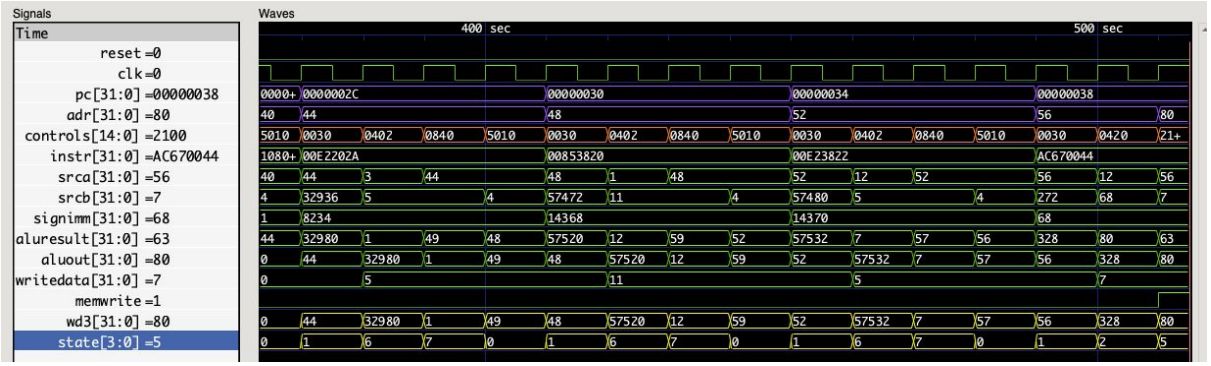


Se utilizó el memfile.dat junto con el mipstest y se observaron las señales que requería el controller.

MipsTest editado:

```
module testbench();
  reg clk;
  reg reset;

  wire [31:0]writedata, adr;
  wire memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, adr, memwrite);
  //dut dut(clk, reset, writedata, dataadr, memwrite);

  // initialize test
  initial
    begin
      reset <= 1;
      #22;
      reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1;
      #5;
      clk <= 0;
      #5;
    end

  // check that 7 gets written to address 84
  always@(negedge clk)
    begin
      if(memwrite) begin
        if(adr === 80 && writedata === 7) begin
          $display("Simulation succeeded");
          $finish;
        end else if (adr !== 80) begin
          $display("Simulation failed");
          $finish;
        end
      end
    end

    initial begin
        $dumpfile("testbenchmulti.vcd");
        $dumpvars;
    end

endmodule
```

4. WAVEFORMS:



¿Coincidió la simulación con las predicciones hechas?

no de alguna manera porque nosotros en la teoria decimos que ciertas señales no entraran o no funcionaran , pero en los wave forms vemos que si estan funcionando , pero no es de preocuparse porque tenemos un diseño de control que permitira el paso de señal o no.

GRACIAS! :D

Los demás informes estarán en la carpeta respectiva de cada laboratorio…