Arquitectura de Computadoras
LAB 6 parte 1

Docente: Jorge Gonzales Reaño

Integrantes:

Nombre y apellidos	Código	Correo
Jose Leandro Machaca	202110202	jose.machaca.s@utec.edu,pe
Diego Pacheco Ferrell	202010159	diego.pacheco@utec.edu.pe
Dimael Rivas Chavez	202110307	dimael.rivas@utec.edu.pe
Anderson Carcamo Vargas	202020025	anderson.carcamo@utec.edu.pe

MAIN FSM OUTPUT:

State	NextPC	Branch	MemeW	RegW	IRWrite	AdrSrc	ResultSrc	ALUSrcA	ALUSrcB	ALUSrcB	AluOp	FSM Control Word
Fetch	1	0	0	0	1	0	10	0	1	10	0	0x114C
Decode	0	0	0	0	0	0	10	0	1	10	0	0x004C
MemAdr	0	0	0	0	0	0	10	0	0	01	0	0x0042
MemRead	0	0	0	0	0	1	00	0	0	01	0	0x0082
MemWB	0	0	1	0	0	1	00	0	0	01	0	0x0482
MemWrite	0	0	1	0	0	1	00	0	0	01	0	0x0482
ExecuteR	0	0	0	0	0	0	10	0	0	00	1	0x0041
ExecuteI	0	0	0	0	0	0	10	0	0	01	1	0x0043
ALUWB	0	0	0	1	0	0	00	0	0	0x	0	0x0203
Branch	0	1	0	0	0	0	10	1	0	01	0	0x0852

Controller

```
module controller (
clk,
reset,
Instr,
ALUFlags,
PCWrite,
MemWrite,
RegWrite,
IRWrite,
AdrSrc,
RegSrc,
ALUSrcA,
ALUSrcB,
ResultSrc,
```

```
ImmSrc,
ALUControl
);
input wire clk;
input wire reset;
input wire [31:12] Instr;
input wire [3:0] ALUFlags;
output wire PCWrite;
output wire MemWrite;
output wire RegWrite;
output wire IRWrite;
output wire AdrSrc;
output wire [1:0] RegSrc;
output wire [1:0] ALUSrcA;
output wire [1:0] ALUSrcB;
output wire [1:0] ResultSrc;
output wire [1:0] ImmSrc;
output wire [1:0] ALUControl;
wire [1:0] FlagW;
wire PCS;
wire NextPC;
wire RegW;
wire MemW;
decode dec(
.clk(clk),
.reset(reset),
.Op(Instr[27:26]),
.Funct(Instr[25:20]),
.Rd(Instr[15:12]),
.FlagW(FlagW),
.PCS(PCS),
.NextPC(NextPC),
.RegW(RegW),
.MemW(MemW),
.IRWrite(IRWrite),
.AdrSrc(AdrSrc),
.ResultSrc(ResultSrc),
.ALUSrcA(ALUSrcA),
.ALUSrcB(ALUSrcB),
.ImmSrc(ImmSrc),
.RegSrc(RegSrc),
.ALUControl(ALUControl)
);
```

```
condlogic cl(
.clk(clk),
.reset(reset),
.Cond(Instr[31:28]),
.ALUFlags(ALUFlags),
.FlagW(FlagW),
.PCS(PCS),
.NextPC(NextPC),
.RegW(RegW),
.MemW(MemW),
.PCWrite(PCWrite),
.RegWrite(RegWrite),
.MemWrite(MemWrite)
);
endmodule
module controller (
clk,
reset,
Instr,
ALUFlags,
PCWrite,
MemWrite,
RegWrite,
IRWrite,
AdrSrc,
RegSrc,
ALUSrcA,
ALUSrcB,
ResultSrc,
ImmSrc,
ALUControl
);
input wire clk;
input wire reset;
input wire [31:12] Instr;
input wire [3:0] ALUFlags;
output wire PCWrite;
output wire MemWrite;
output wire RegWrite;
output wire IRWrite;
output wire AdrSrc;
output wire [1:0] RegSrc;
output wire [1:0] ALUSrcA;
```

```
output wire [1:0] ALUSrcB;
output wire [1:0] ResultSrc;
output wire [1:0] ImmSrc;
output wire [1:0] ALUControl;
wire [1:0] FlagW;
wire PCS;
wire NextPC;
wire RegW;
wire MemW;
decode dec(
.clk(clk),
.reset(reset),
.Op(Instr[27:26]),
.Funct(Instr[25:20]),
.Rd(Instr[15:12]),
.FlagW(FlagW),
.PCS(PCS),
.NextPC(NextPC),
.RegW(RegW),
.MemW(MemW),
.IRWrite(IRWrite),
.AdrSrc(AdrSrc),
.ResultSrc(ResultSrc),
.ALUSrcA(ALUSrcA),
.ALUSrcB(ALUSrcB),
.ImmSrc(ImmSrc),
.RegSrc(RegSrc),
.ALUControl(ALUControl)
);
condlogic cl(
.clk(clk),
.reset(reset),
.Cond(Instr[31:28]),
.ALUFlags(ALUFlags),
.FlagW(FlagW),
.PCS(PCS),
.NextPC(NextPC),
.RegW(RegW),
.MemW(MemW),
.PCWrite(PCWrite),
.RegWrite(RegWrite),
.MemWrite(MemWrite)
);
```

```
endmodule
```

Decode:

El decode tiene la misma estructura del single cycle con respecto al aludecoder y al PC Logic, con la diferencia que ImmSrc y RegSrc son determinados por las transiciones de op. Por eso se le añade un decoder a Instruction.

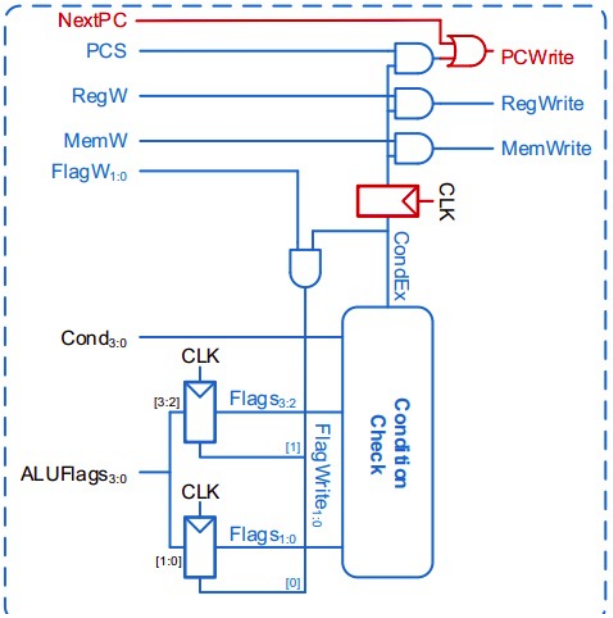
```
module decode (
clk,
reset,
Op,
Funct,
Rd,
FlagW,
PCS,
NextPC,
RegW,
MemW,
IRWrite,
AdrSrc,
ResultSrc,
ALUSrcA,
ALUSrcB,
ImmSrc,
RegSrc,
ALUControl
);
input wire clk;
input wire reset;
input wire [1:0] Op;
input wire [5:0] Funct;
input wire [3:0] Rd;
output reg [1:0] FlagW;
output wire PCS;
output wire NextPC;
output wire RegW;
output wire MemW;
output wire IRWrite;
output wire AdrSrc;
output wire [1:0] ResultSrc;
output wire [1:0] ALUSrcA;
output wire [1:0] ALUSrcB;
output wire [1:0] ImmSrc;
output wire [1:0] RegSrc;
```

```
output reg [1:0] ALUControl;
wire Branch;
wire ALUOp;

// Main FSM
mainfsm fsm(
.clk(clk),
.reset(reset),
.Op(Op),
.Funct(Funct),
.IRWrite(IRWrite),
.AdrSrc(AdrSrc),
.ALUSrcA(ALUSrcA),
.ALUSrcB(ALUSrcB),
.ResultSrc(ResultSrc),
.NextPC(NextPC),
.RegW(RegW),
.MemW(MemW),
.Branch(Branch),
.ALUOp(ALUOp)
);

// ALU Decoder
always @(*)
if (ALUOp) begin
case (Funct[4:1])
4'b0100: ALUControl = 2'b00;
4'b0010: ALUControl = 2'b01;
4'b0000: ALUControl = 2'b10;
4'b1100: ALUControl = 2'b11;
default: ALUControl = 2'bxx;
endcase
FlagW[1] = Funct[0];
FlagW[0] = Funct[0] & ((ALUControl == 2'b00) | (ALUControl == 2'b01));
end
else begin
ALUControl = 2'b00;
FlagW = 2'b00;
end
assign PCS = ((Rd == 4'b1111) & RegW) | Branch;

// Instruction decoder for ImmSrc and RegSrc
assign ImmSrc = Op;
```

```
assign RegSrc[0] = Op == 2'b10;
assign RegSrc[1] = Op == 2'b01;
endmodule
```

CondLogic:

Para el conditional logic, retardamos el CondEx con el uso de 1 flip-flop, debido a que CondEx es 1 bit. La variable que tendrá su valor es AfterCondEx. Por otro lado, guiándonos del esquema del output del condicional logic, cambiamos el atrasamiento de Flagwrite debido a que no es necesario.



- **PCWrite** asserted in Fetch state
- **Executel/ExecuteR** state:
  CondEx asserts
  ALUFlags generated
- **ALUWB** state:
  Flags updated
  CondEx changes
  PCWrite, ***RegWrite***, and ***MemWrite*** don't see change till new instruction (Fetch state)

```
// reuse code from prior labs.
module condlogic (
clk,
reset,
Cond,
ALUFlags,
FlagW,
PCS,
NextPC,
RegW,
MemW,
PCWrite,
RegWrite,
MemWrite
);
input wire clk;
input wire reset;
input wire [3:0] Cond;
input wire [3:0] ALUFlags;
```

```
input wire [1:0] FlagW;
input wire PCS;
input wire NextPC;
input wire RegW;
input wire MemW;
output wire PCWrite;
output wire RegWrite;
output wire MemWrite;
wire [1:0] FlagWrite;
wire [3:0] Flags;
wire CondEx;
wire AfterCondEx;

// Delay writing flags until ALUWB state
flopr #(1) flagwritereg(
clk,
reset,
CondEx,
AfterCondEx
);

// ADD CODE HERE
flopenr #(2) flagreg1(
.clk(clk),
.reset(reset),
.en(FlagWrite[1]),
.d(ALUFlags[3:2]),
.q(Flags[3:2])
);
flopenr #(2) flagreg0(
.clk(clk),
.reset(reset),
.en(FlagWrite[0]),
.d(ALUFlags[1:0]),
.q(Flags[1:0])
);
condcheck cc(
.Cond(Cond),
.Flags(Flags),
.CondEx(CondEx)
);
assign PCWrite = NextPC | (AfterCondEx & PCS);
assign RegWrite = AfterCondEx & RegW;
```

```
assign MemWrite = AfterCondEx & MemW;
assign FlagWrite = FlagW & {2 {CondEx}};

endmodule
```

MainFsm:

Para el fsm nos guiamos del diagrama de transiciones del enunciado, y para los valores no determinamos, los mantenemos de su predecesor, es decir, los valores que no han sido modificados de un nodo, son los mismos que los de su padre, puedes de tomas maneras los valores de estos no van a servinos. Para los otros valores como Branch, MemWrite, etc, donde solo se van a considerar en ciertos estados, hemos considerado 0 en otros estados.

```
module mainfsm (
clk,
reset,
Op,
Funct,
IRWrite,
AdrSrc,
ALUSrcA,
ALUSrcB,
ResultSrc,
NextPC,
RegW,
MemW,
Branch,
ALUOp
);
input wire clk;
input wire reset;
input wire [1:0] Op;
input wire [5:0] Funct;
output wire IRWrite;
output wire AdrSrc;
output wire [1:0] ALUSrcA;
output wire [1:0] ALUSrcB;
output wire [1:0] ResultSrc;
output wire NextPC;
output wire RegW;
output wire MemW;
output wire Branch;
output wire ALUOp;
reg [3:0] state;
reg [3:0] nextstate;
reg [12:0] controls;
```

```
localparam [3:0] FETCH = 0;
localparam [3:0] DECODE = 1;
localparam [3:0] MEMADR = 2;
localparam [3:0] MEMREAD = 3;
localparam [3:0] MEMWB = 4;
localparam [3:0] MEMWRITE = 5;
localparam [3:0] EXECUTER = 6;
localparam [3:0] EXECUTEI = 7;
localparam [3:0] ALUWB = 8;
localparam [3:0] BRANCH = 9;
localparam [3:0] UNKNOWN = 10;

// state register
always @(posedge clk or posedge reset)
if (reset)
state <= FETCH;
else
state <= nextstate;

// ADD CODE BELOW
// Finish entering the next state logic below. We've completed the
// first two states, FETCH and DECODE, for you.

// next state logic
always @(*)
casex (state)
FETCH: nextstate = DECODE;
DECODE:
case (Op)
2'b00:
if (Funct[5])
nextstate = EXECUTEI;
else
nextstate = EXECUTER;
2'b01: nextstate = MEMADR;
2'b10: nextstate = BRANCH;
default: nextstate = UNKNOWN;
endcase
MEMADR:
case (Funct[0])
1'b0:
nextstate = MEMWRITE;
```

```
1'b1:
nextstate = MEMREAD;
endcase
MEMREAD: nextstate = MEMWB;
EXECUTER: nextstate = ALUWB;
EXECUTEI: nextstate = ALUWB;
default: nextstate = FETCH;
endcase

// ADD CODE BELOW
// Finish entering the output logic below. We've entered the
// output logic for the first two states, FETCH and DECODE, for you.

// state-dependent output logic
always @(*)
case (state)
FETCH: controls = 13'b1000101001100;
DECODE: controls = 13'b0000001001100;
EXECUTER: controls = 13'b0000001000001;
EXECUTEI: controls = 13'b0000001000011;
ALUWB: controls = 13'b0001000000001;
MEMADR: controls = 13'b0000001000010;
MEMWRITE:controls = 13'b0010010000010;
MEMREAD: controls = 13'b0000010000010;
MEMWB: controls = 13'b0010010000010;
BRANCH: controls = 13'b0100001010010;
default: controls = 13'bxxxxxxxxxxxxx;
endcase
assign {NextPC, Branch, MemW, RegW, IRWrite, AdrSrc, ResultSrc,
ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule
```

Controller Test Bench

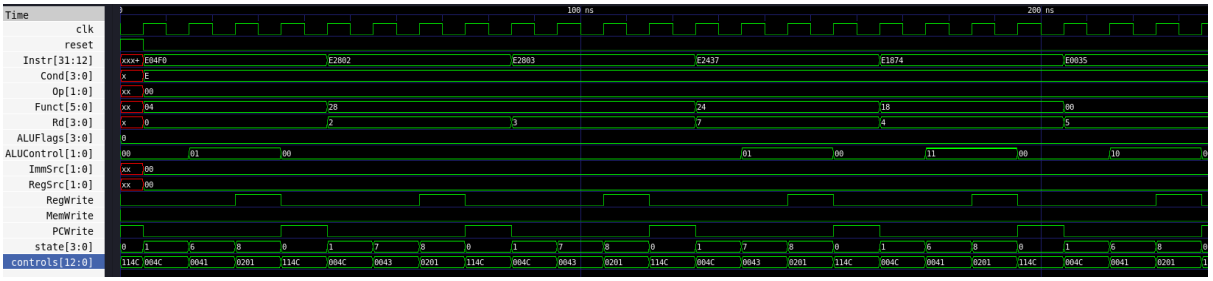
Para probar el controller, hemos usado el archivo de memfile.dat y lo hemos recorrido por instrucción que se ha pasado al controller, obteniendo los resultados esperados según la instrucción. Para el recorrimiento del memfile.dat se ha utilizado RAM que es donde se ha almacenado la linea de instrucción y ¨a¨ para poder iterar en el archivo.

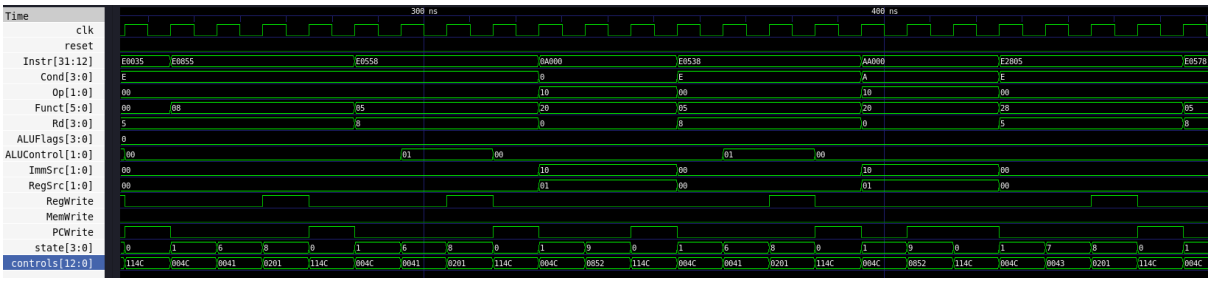
```
`timescale 1ns/1ns

module controller_tb;
reg clk;
reg reset;
```

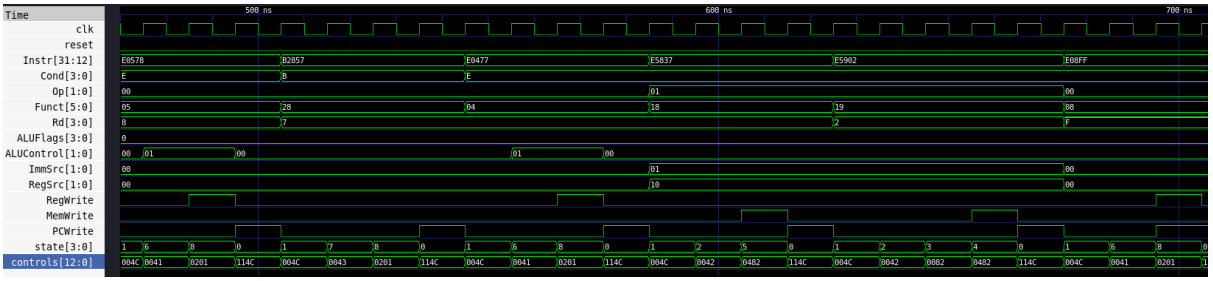
```
reg [31:12] Instr;
reg [3:0] ALUFlags;
wire PCWrite;
wire MemWrite;
wire RegWrite;
wire IRWrite;
wire AdrSrc;
wire [1:0] RegSrc;
wire [1:0] ALUSrcA;
wire [1:0] ALUSrcB;
wire [1:0] ResultSrc;
wire [1:0] ImmSrc;
wire [1:0] ALUControl;
reg [31:0] RAM [63:0]; // Para leer el memfile.dat
reg [31:0] a;
controller dut (
.clk(clk),
.reset(reset),
.Instr(Instr),
.ALUFlags(ALUFlags),
.PCWrite(PCWrite),
.MemWrite(MemWrite),
.RegWrite(RegWrite),
.IRWrite(IRWrite),
.AdrSrc(AdrSrc),
.RegSrc(RegSrc),
.ALUSrcA(ALUSrcA),
.ALUSrcB(ALUSrcB),
.ResultSrc(ResultSrc),
.ImmSrc(ImmSrc),
.ALUControl(ALUControl)
);
always begin
clk <= 0;
#(5);
clk <= 1;
#(5);
end
initial begin
$readmemh("memfile.dat", RAM);
a = 0;
ALUFlags = 4'b0000; // empieza sin activar ningun flag
reset = 1;
```

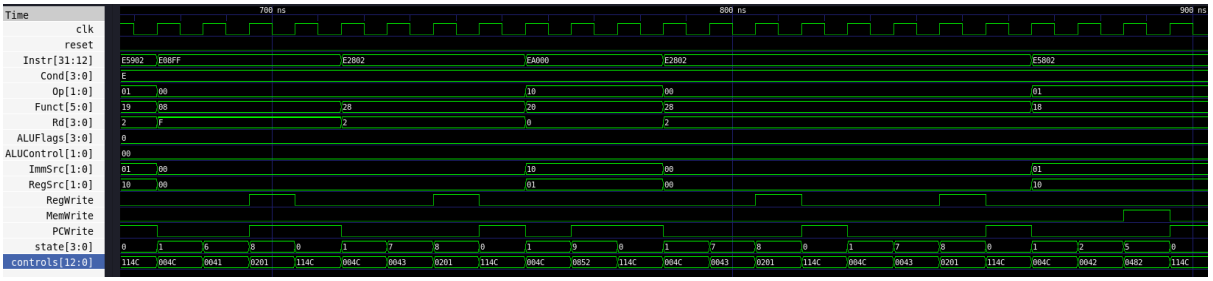
```
#5; reset = 0;
end
always @(posedge clk) begin
if(~reset & IRWrite) begin
if (RAM[a] == 0)
$finish;
Instr = RAM[a][31:12];
if (Instr == 20'b11100010010100100011)
ALUFlags = 4'b0110;
a = a+1; // me sirve para recorrer las instrucciones en el memfile.dat
end
end

initial begin
$dumpfile("controller_tb.vcd");
$dumpvars;
end
endmodule
```

Waveform:









Los resultados concuerdan con lo propuesto debido a que se ve el desfase producido por el atrasamiento de CondEx y la transición de los estados en función a las instrucciones. Por otro lado, el memwrite responde a las instrucciones que necesitan el acceso a la escritura en la memoria, esto también sucede con las señales de regwrite y pcwrite. Sin embargo, se puede apreciar de que los aluflags no son modificados en ningún momento, esto se debe a la falta del alu, pero como en esta entrega solo consiste en el controller, los aluflags no son modificados.