Proyecto

Grupo 2

July 31, 2021

Integrantes:

- Denis Irala
- Alonso Rios
- Juan Laredo

1. Include both ASM programs extensively commented.

```
- Para MUL / UMULL / SMULL
.global _start
_start:

// Función para hallar la Fuerza ejercida sobre un objeto

    // Fuerza = Masa * Aceleración

    SUB R0, R15, R15 //Asignando 0 a los valores
SUB R1, R15, R15
SUB R2, R15, R15

ADD R0, R0, #50 // Masa
    ADD R1, R1, #10 //Aceleración 1
    ADD R2, R2, #-5 //Aceleración 2

BL FUERZA
B END

FUERZA:

    UMULL R3, R4, R0, R1 //  Fuerza 1 = Masa * Aceleración  1
MUL  R5, R0, R1 //Fuerza 1 = Masa * Aceleración 1
    SMULL R6, R7, R0, R2 // Fuerza 2 = Masa * Aceleración 2

MOV PC, LR

END:


- Para FP ADD, FP MUL
.global _start
_start:
```

```
//Programa para sumar masas y hallar el momento M=mv

//Los registros fueron inicializados mediante hardcode.

// Masa 1 = R5 = 60 kg = 0x42700000 = 0x4270
// Masa 2 = R6 = 10.5 kg = 41280000 = 0x4128
// Velocidad = R7 = 5 m/s =40a00000 = 0x40a0

BL f

B End

f:
FPADD R0, R5, R6 // Suma de masas FP 32-bits
FPMUL R1, R5, R7 // Momento = Masa 1 * Velocidad FP 32-bits
FPADD16 R2, R5, R6 // Suma de masas FP 16-bits
FPMUL16 R3, R5, R7 // Momento = Masa 1 * Velocidad FP 16-bits


End:
```

2. Show two tables with the extended functionality in control and datapath.

SINGLE CYCLE

ControlUnit SIGNALS

	PCsrc	MemToReg	MemWrite	ALUcontrol	ALUsrc	ImmSrc	RegWrite	WE4	NewSource	IsMul	FPcontrol	ResultControl
32FPmul	NC	0	0	xxxx	0	xx	1	0	1	0	11	1
32FPadd	NC	0	0	xxxx	0	xx	1	0	1	0	O1	1
16FPmul	NC	0	0	xxxx	0	xx	1	0	1	0	10	1
16FPadd	NC	0	0	xxxx	0	xx	1	0	1	0	OO	1
UMULL/SMULL	NC	0	0	1000/1100	0	xx	1	1	1	0	XX	0
MUL	NC	0	0	O100	0	xx	1	0	1	1	XX	0

CONTROLS SIGNALS FP instr

	RegSRC	ImmSRC	ALUsrc	MemToReg	RegW	MemW	Branch	ALUop	WE4w	ewSour	IsMul	ResultControl
FPinstr	OO	OO	0	0	1	0	0	1	0	1	0	1
UMULL/SMULL	OO	OO	0	0	1	0	0	1	1	1	0	0
MUL	OO	OO	0	0	1	0	0	1	0	1	1	0

MULTI CYCLE

CONTROL SIGNALS for FP instructions

	NextPC	Branch	MemW	RegW	IRWrite	AdrSrc	ResultSrc	LUSrc	ALUSrcB	ALUop	WE4	ResultControl
EXECUTER(6)	0	0	0	0	0	0	10	OO	OO	1	0	1

CONTROL SIGNALS for umull/smull

	NextPC	Branch	MemW	RegW	IRWrite	AdrSrc	ResultSrc	LUSrc	ALUSrcB	ALUop	WE4	ResultControl
ALUWB(6)	0	0	0	1	0	0	OO	OO	OO	1	1	0

CONTROL SIGNALS for mul

	NextPC	Branch	MemW	RegW	IRWrite	AdrSrc	ResultSrc	LUSrc	ALUSrcB	ALUop	WE4	ResultControl
ALUWB(6)	0	0	0	1	0	0	OO	OO	OO	1	0	0

3. Highlight the Verilog modifications to your single and multi-cycle processor (Los cambios son comentados casi en su totalidad.)

```
   //// Para tanto single-cycle como multi-cycle:

// FPU.v (Floating Point Unit)

module FPU (a, b, ALUControl, Result);
  input wire [31:0] a, b;
  input wire [1:0] ALUControl;
  output wire [31:0] Result;
  wire [31:0]Result32;
  wire [15:0]tempResult16;
  wire [31:0] Result16;

  assign Result16={16'b0000000000000000,tempResult16};
  fp fp32(
    .a(a),
    .b(b),
    .ALUControl(ALUControl[1]),
    .Result(Result32)
    );
  fp16 fp16(
    .a(a),
    .b(b),
    .ALUControl(ALUControl[1]),
    .Result(tempResult16)
  );

  mux2 #(32) decisivo(
    .d0(Result16),
    .d1(Result32),
    .s(ALUControl[0]),
    .y(Result)
  );
endmodule

// fp.v (Floating Point 32-bits)

module fp(a, b, ALUControl, Result);
  input [31:0] a, b;
  input ALUControl; // 0 = Suma // 1 = MultiplicaciÃ³n
  output reg [31:0] Result;
```

```
reg[31:0] exponente_a;
reg [31:0] mantissa_a;

reg[31:0] exponente_b;
reg[31:0] mantissa_b;

reg[31:0] temp1;
reg[999:0] temp4;
reg[31:0] temp3;
reg[31:0] temp2;
wire [31:0] inicial;
wire [31:0] exponente;
wire [31:0] matissa;

assign inicial   = 32'b00000000100000000000000000000000;
assign exponente = 32'b01111111100000000000000000000000;
assign matissa   = 32'b00000000011111111111111111111111;

always @(*)
  begin
    if(ALUControl) begin
      exponente_a = exponente & a;
      exponente_b = exponente & b;
      exponente_a = exponente_a>>23;
      exponente_b = exponente_b>>23;
      mantissa_a = a & matissa;
      mantissa_b = b & matissa;
      mantissa_a = mantissa_a | inicial;
      mantissa_b = mantissa_b | inicial;
      temp1 = exponente_a + exponente_b - 7'b1111110;
      temp4 = mantissa_a * mantissa_b;
      while(temp4 > 24'b100000000000000000000000)begin
        temp4 = temp4 >> 1;
      end
      temp4 = temp4 << 1;
      temp2 = temp4;
      temp2 = temp2 [22:0];
      temp1 = temp1<<23;
      temp2 = temp1 | temp2;
      Result = temp2;
      Result [31] = a[31]^b[31]; // Para saber si es negativo
    end
    else begin
      exponente_a=exponente & a;
```

```
        exponente_b=exponente & b;
        mantissa_a=a & matissa;
        mantissa_b=b & matissa;
        exponente_a = exponente_a>>23;
        exponente_b = exponente_b>>23;
        mantissa_a = mantissa_a | inicial;
        mantissa_b = mantissa_b | inicial;
        if(exponente_a>exponente_b)begin
            temp1=exponente_a-exponente_b;
            mantissa_b=mantissa_b>>temp1;
            temp2=mantissa_a+mantissa_b;
            temp1=exponente_a;
        end
        else begin
          temp1=exponente_b-exponente_a;
          mantissa_a=mantissa_a>>temp1;
          temp2=mantissa_a+mantissa_b;
          temp1=exponente_b;
        end
        temp3 = temp2;
        temp3 = temp3>>23;
        if(temp3>=2) begin
          temp2=temp2>>1;
          temp1=temp1+32'b00000000000000000000000000000001;
        end
      temp1=temp1<<23;
      temp2=temp2 & matissa;
      temp2=temp1 | temp2;
      Result=temp2;
        end

    end

endmodule

//fp16.v (Floating Point 16-bits)

module fp16(a,b,ALUControl,Result);
    input [15:0] a,b;
    input ALUControl;
    output reg [15:0] Result;
    reg[15:0] exponente_a;
    reg [15:0] mantissa_a;
```

```
    reg[15:0] exponente_b;
    reg[15:0] mantissa_b;

    reg[15:0] temp1;
    reg[15:0] temp2;
    reg[999:0] temp4;

    reg[15:0] temp3;
    wire [15:0] inicial;
    wire [15:0] exponente;
    wire [15:0] matissa;

    assign inicial   = 16'b0000010000000000;
    assign exponente = 16'b0111110000000000;
    assign matissa   = 16'b0000001111111111;
always @(*)
    begin
      if(ALUControl) begin
        exponente_a = exponente & a;
        exponente_b = exponente & b;
        exponente_a = exponente_a>>10;
        exponente_b = exponente_b>>10;
        mantissa_a = a & matissa;
        mantissa_b = b & matissa;
        mantissa_a = mantissa_a | inicial;
        mantissa_b = mantissa_b | inicial;
        temp1 = exponente_a + exponente_b - 4'b1110;
        temp4 = mantissa_a * mantissa_b;
        while(temp4 > 11'b10000000000)begin
          temp4 = temp4 >> 1;
        end
        temp4 = temp4 << 1;
        temp2 = temp4;
        temp2 = temp2 [9:0];
        temp1 = temp1<<10;
        temp2 = temp1 | temp2;
        Result = temp2;
        Result [15] = a[15]^b[15]; // Para saber si es negativo
      end
      else begin
        exponente_a=exponente & a;
        exponente_b=exponente & b;
        mantissa_a=a & matissa;
        mantissa_b=b & matissa;
```

```
        exponente_a = exponente_a>>10;
        exponente_b = exponente_b>>10;
        mantissa_a = mantissa_a | inicial;
        mantissa_b = mantissa_b | inicial;
        if(exponente_a>exponente_b)begin
            temp1=exponente_a-exponente_b;
            mantissa_b=mantissa_b>>temp1;
            temp2=mantissa_a+mantissa_b;
            temp1=exponente_a;
        end
        else begin
          temp1=exponente_b-exponente_a;
          mantissa_a=mantissa_a>>temp1;
          temp2=mantissa_a+mantissa_b;
          temp1=exponente_b;
        end
        temp3 = temp2;
        temp3 = temp3>>10;
        if(temp3>=2) begin
          temp2=temp2>>1;
          temp1=temp1+16'b0000000000000001;
        end
      temp1=temp1<<10;
      temp2=temp2 &matissa;
      temp2=temp1 | temp2;
      Result=temp2;
        end
    end

// alu.v
module alu(a, b, ALUControl, Result, ALUFlags, Result2);
  input [31:0] a, b;
  input [3:0] ALUControl; //nuevo
  output reg [31:0] Result;
  output wire [3:0] ALUFlags;
  output reg [31:0] Result2; //
  reg [63:0] aux;
  //nuevo
  reg signed [63:0] tempAUX;
  reg signed [31:0] tempA;
  reg signed [31:0] tempB;
  reg tempCARRY; //nuevo
  reg tempOVERFLOW; //nuevo
  wire [32:0] sum;
```

```
  wire neg, zero, carry, overflow;



  assign sum = a + (ALUControl[0]? ~b:b) + ALUControl[0]; //2's complement

  always @(*)
    begin
      casex (ALUControl[3:0])
        4'b000?: Result=sum; //suma o resta
        4'b0010: Result=a&b; //and
        4'b0011: Result=a|b; //or
        4'b 0100: Result = a*b;// mull
        4'b 1000: begin // umull
          aux = a * b;
          Result2= aux[63:32];
          Result= aux[31:0];
        end
        4'b 1100: begin // smull
          tempA=a;
          tempB=b;
          tempAUX= tempA*tempB;
          Result2= tempAUX[63:32];
          Result= tempAUX[31:0];

        end

      endcase
    end

    //nuevo
  always @(*) begin
    if(ALUControl == 4'b 0100 || ALUControl == 4'b1000 || ALUControl == 4'b1100)
 begin
    tempCARRY=0;
    tempOVERFLOW=0;
    end
    else begin
    tempCARRY = (ALUControl[1]==1'b0) & sum[32];
    tempOVERFLOW = (ALUControl[1]==1'b0) &
~(a[31] ^ b[31] ^ ALUControl[0]) & (a[31] ^ sum[31]);

    end
```

```
  end

  //flags
  assign neg = Result[31]; // para umull y smull,
  assign zero = (Result == 32'b0);
  assign carry = tempCARRY;
  assign overflow = tempOVERFLOW;


  assign ALUFlags= {neg, zero, carry, overflow};

endmodule

  ////   Exclusivo de Multi-cycle:

// decode.v

module decode (
clk,
reset,
Op,
Funct,
Rd,
FlagW,
PCS,
NextPC,
RegW,
MemW,
IRWrite,
AdrSrc,
ResultSrc,
ALUSrcA,
ALUSrcB,
ImmSrc,
RegSrc,
ALUControl,
MULL_Identifier,  // nuevo input
WE4w,
NewSource,
IsMul,
ResultControl,
FPControl,
FP_identifier,
BIT_identifier,
```

```
OP_identifier
);
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 [3:0] ALUControl; //
wire Branch;
wire ALUOp;

//nuevo
input wire [3:0] MULL_Identifier; //
output wire WE4w; //
output wire NewSource; //
output wire IsMul; //
reg [1:0] NewSignals;

//fp
output wire ResultControl;
output reg [1:0] FPControl;
input wire [4:0] FP_identifier;
input wire [3:0] BIT_identifier;
input wire [3:0] OP_identifier;

// 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),
.MULL_Identifier(MULL_Identifier),
.WE4w(WE4w),
.ResultControl(ResultControl),
.FP_identifier(FP_identifier)
);

// ADD CODE BELOW
// Add code for the ALU Decoder and PC Logic.
// Remember, you may reuse code from previous labs.

// The ALU Decoder and PC Logic are identical
 to those in the single-cycle processor.

// ALU Decoder    <------- Recibe ALUOp de MainFSM

always @(*)
if (ALUOp) begin
if (FP_identifier == 5'b 11111 & !Funct[5]) begin
case (BIT_identifier)
4'b 0000: // 32bit
if(OP_identifier == 4'b1111) //32mul
FPControl = 2'b11;
else //32add
FPControl= 2'b 01;
4'b 1111: // 16 bit
if(OP_identifier == 4'b1111) //16mul
FPControl= 2'b10;
else//16add
FPControl= 2'b00;
default: FPControl = 2'bxx;
endcase
FlagW[1]=1'b 0; // nz
FlagW[0]=1'b 0;
end
else if (MULL_Identifier== 4'b1001 & !Funct[5]) begin
```

```
case (Funct [3:1])
3'b 000: ALUControl=4'b 0100; //MULL
3'b 100: ALUControl=4'b 1000; //UMULL
3'b 110: ALUControl=4'b 1100; //SMULL
endcase
FlagW[1]=1'b0; // nz
FlagW[0]=1'b 0; //cv
end
else begin
case (Funct[4:1])
4'b0100: ALUControl = 4'b0000;
4'b0010: ALUControl = 4'b0001;
4'b0000: ALUControl = 4'b0010;
4'b1100: ALUControl = 4'b0011;
default: ALUControl = 4'bxxxx;
endcase
FlagW[1] = Funct[0];
FlagW[0] = Funct[0] & ((ALUControl == 4'b0000) | (ALUControl == 4'b0001));
end
end
else begin
ALUControl = 4'b0000; //
FlagW = 2'b00;
end

// PC Logic
// Rd es input del Decoder, Branch y RegW vienen de MainFSM

assign PCS = ((Rd == 4'b1111) & RegW) | Branch;


// Add code for the Instruction Decoder (Instr Decoder) below.
// Recall that the input to Instr Decoder is Op, and the outputs are
// ImmSrc and RegSrc. We've completed the ImmSrc logic for you.

// Instr Decoder
assign RegSrc[1] = (Op == 01);
    assign RegSrc[0] = (Op == 10);
assign ImmSrc = Op;

//Nuevo

always @(*)
casex (Op)
```

```
2'b00:
if (MULL_Identifier== 4'b1001 & Funct[3] &
 !Funct[5] || FP_identifier== 5'b11111) //umull y smull y fp
NewSignals = 2'b 10;    // nuevo
else if (MULL_Identifier== 4'b1001) //mull
NewSignals = 2'b 11;
else
NewSignals = 2'b 00;
default: NewSignals = 2'b 00;
endcase
assign {NewSource,IsMul} = NewSignals;

endmodule



// regfile.v
module regfile (
clk,
we3,
ra1,
ra2,
wa3,
wd3,
r15,
rd1,
rd2,
WE4,
WD4,
WA4
);
input wire clk;
input wire we3;
input wire [3:0] ra1;
input wire [3:0] ra2;
input wire [3:0] wa3;
input wire [31:0] wd3;
input wire [31:0] r15;
output wire [31:0] rd1;
output wire [31:0] rd2;

input wire WE4; //nuevo
input wire [31:0] WD4; //nuevo
input wire [3:0] WA4;
```

```
reg [31:0] rf [14:0];

 //// Esta parte es un hardcodeo de registros que SOLO se usa para ejecutar
 el programa, mas no es parte del procesador.

always @(clk) begin
rf[5] = 32'b 01000010011100000000000000000000;
rf[6] = 32'b 01000001001010000000000000000000;
rf[7] = 32'b 01000000101000000000000000000000;
end

////
always @(posedge clk) begin
if (we3) begin
rf[wa3] <= wd3;
end
if (WE4)
begin
rf[WA4] <= WD4;   //NUEVO
end
end
assign rd1 = (ra1 == 4'b1111 ? r15 : rf[ra1]);
assign rd2 = (ra2 == 4'b1111 ? r15 : rf[ra2]);
endmodule

// controller.v
module controller (
clk,
reset,
Instr,
ALUFlags,
PCWrite,
MemWrite,
RegWrite,
IRWrite,
AdrSrc,
RegSrc,
ALUSrcA,
ALUSrcB,
ResultSrc,
ImmSrc,
ALUControl,
WE4,
```

```
NewSource,
IsMul,
ResultControl,
FPControl
);
input wire clk;
input wire reset;
input wire [31:0] 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 [3:0] ALUControl;
wire [1:0] FlagW;
wire PCS;
wire NextPC;
wire RegW;
wire MemW;

//nuevo
output wire WE4;
output wire NewSource;
output wire IsMul;
wire WE4w;
output wire ResultControl;
output wire [1:0] FPControl;




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),
.MULL_Identifier(Instr[7:4]), // nuevo
.WE4w(WE4w),
.NewSource(NewSource),
.IsMul(IsMul),
.ResultControl(ResultControl),
.FPControl(FPControl),
.FP_identifier(Instr[24:20]),
.BIT_identifier(Instr[19:16]),
.OP_identifier(Instr[7:4])
);
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),
.WE4w(WE4w),
.WE4(WE4)
);
endmodule

// condlogic.v

module condlogic (
clk,
```

```
reset,
Cond,
ALUFlags,
FlagW,
PCS,
NextPC,
RegW,
MemW,
PCWrite,
RegWrite,
MemWrite,
WE4,
WE4w
);
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 CondEx2;
wire impro;
//nuevo
input wire WE4w;
output wire WE4;

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])
);

flopr #(1) ff(
        clk,
        reset,
        CondEx,
        CondEx2
    );


condcheck cc(
.Cond(Cond),
.Flags(Flags),
.CondEx(CondEx)
);

assign FlagWrite = FlagW & {2 {CondEx}};
assign RegWrite = RegW & CondEx2;
assign MemWrite = MemW & CondEx2;
assign impro = PCS & CondEx2;
assign PCWrite = NextPC | impro;
assign WE4= WE4w & CondEx2;



endmodule

// datapath.v
module datapath (
clk,
reset,
Adr,
WriteData,
ReadData,
Instr,
ALUFlags,
PCWrite,
RegWrite,
IRWrite,
AdrSrc,
```

```
RegSrc,
ALUSrcA,
ALUSrcB,
ResultSrc,
ImmSrc,
ALUControl,
WE4, //
NewSource,
IsMul,
ResultControl,
FPControl
);
input wire clk;
input wire reset;
output wire [31:0] Adr;
output wire [31:0] WriteData;
input wire [31:0] ReadData;
output wire [31:0] Instr;
output wire [3:0] ALUFlags;
input wire PCWrite;
input wire RegWrite;
input wire IRWrite;
input wire AdrSrc;
input wire [1:0] RegSrc;
input wire [1:0] ALUSrcA;
input wire [1:0] ALUSrcB;
input wire [1:0] ResultSrc;
input wire [1:0] ImmSrc;
input wire [3:0] ALUControl;
wire [31:0] PCNext;
wire [31:0] PC;
wire [31:0] ExtImm;
wire [31:0] SrcA;
wire [31:0] SrcB;
wire [31:0] Result;
wire [31:0] Data;
wire [31:0] RD1;
wire [31:0] RD2;
wire [31:0] A;
wire [31:0] ALUResult;
wire [31:0] ALUOut;
wire [3:0] RA1;
wire [3:0] RA2;
//nuevo
```

```
wire [31:0] ALUResult2;
wire [31:0] Result2;
wire [3:0] newRA1;
wire [3:0] newRA2;
wire [3:0] newRA3;
input wire WE4;
input wire IsMul;
input wire NewSource;
input wire ResultControl;
input wire [1:0] FPControl;
wire [31:0] FPResult;
wire [31:0] newALUResult;



// Your datapath hardware goes below. Instantiate each of the
// submodules that you need. Remember that you can reuse hardware
// from previous labs. Be sure to give your instantiated modules
// applicable names such as pcreg (PC register), adrmux
// (Address Mux), etc. so that your code is easier to understand.

// ADD CODE HERE



// Instruction Register
flopenr #(32) IRegister(
.clk(clk),
.reset(reset),
.en(IRWrite),
.d(ReadData),
.q(Instr)
);

// Data Register
flopr #(32) DataRegister(   //
.clk(clk),
.reset(reset),
.d(ReadData),
.q(Data)
);
```

```
// PC
flopenr #(32) pcreg(
.clk(clk),
.reset(reset),
.en(PCWrite),
.d(Result),
.q(PC)
);

// mux para escoger Adr
mux2 #(32) Adrmux(
.d0(PC),
.d1(Result),
.s(AdrSrc),
.y(Adr)
);


// mux para escoger RA1 (igual al single cycle)
mux2 #(4) ra1mux(
.d0(Instr[19:16]),
.d1(4'b1111),
.s(RegSrc[0]),
.y(RA1)
);

// mux para escoger RA2 (igual single cycle)
mux2 #(4) ra2mux(
.d0(Instr[3:0]),
.d1(Instr[15:12]),
.s(RegSrc[1]),
.y(RA2)
);


//new muxes // ------------------------------------------------------
mux2  #(4) ra1mux2 (
.d0(RA1),
.d1(Instr[11:8]),
.s(NewSource),
.y(newRA1)
);
mux2 #(4) ra2mux2(
```

```
.d0(RA2),
.d1(Instr[3:0]),
.s(NewSource),
.y(newRA2)
);
mux2 #(4) ra3mux(
.d0(Instr[15:12]),
.d1(Instr[19:16]),
.s(IsMul),
.y(newRA3)
);

mux2 #(32) FPmux(
.d0(ALUResult),
.d1(FPResult),
.s(ResultControl),
.y(newALUResult)

);


// Register File
regfile rf(
.clk(clk),
.we3(RegWrite),
.ra1(newRA1),
.ra2(newRA2),
.wa3(newRA3),
.wd3(Result),
.r15(Result),
.rd1(RD1),
.rd2(RD2),
.WE4(WE4),
.WD4(Result2),
.WA4(Instr[19:16])
);

// A Register
flopr #(32) Areg(
.clk(clk),
.reset(reset),
.d(RD1),
.q(A)
);
```

```
// WriteData Register
flopr #(32) WDreg(
.clk(clk),
.reset(reset),
.d(RD2),
.q(WriteData)
);


// Extend Imm    <------- same
extend ext(
.Instr(Instr[23:0]),
.ImmSrc(ImmSrc),
.ExtImm(ExtImm)
);


// mux para escoger SrcA

mux3 #(32) muxSrcA(
.d0(A),
.d1(PC),
.d2(ALUOut),
.s(ALUSrcA),
.y(SrcA)
);

// mux para escoger SrcB

mux3 #(32) muxSrcB(
.d0(WriteData),
.d1(ExtImm),
.d2(32'b00000000000000000000000000000100),
.s(ALUSrcB),
.y(SrcB)
);

// alu <------same as in single cycle

alu alu(
SrcA,
SrcB,
ALUControl,
```

```
ALUResult,
ALUFlags,
ALUResult2
);

FPU FPU(
.a(SrcA),
.b(SrcB),
.ALUControl(FPControl),
.Result(FPResult)
);

// ALUOut Register
//assign Result2=ALUResult2;
// new flopr <---------------
flopr #(32) Result2ToReg(
.clk(clk),
.reset(reset),
.d(ALUResult2),
.q(Result2)
);
flopr #(32) aluoutreg(
.clk(clk),
.reset(reset),
.d(newALUResult),
.q(ALUOut)
);

// mux para escoger Result

mux3 #(32) muxResult(
.d0(ALUOut),
.d1(Data),
.d2(newALUResult),
.s(ResultSrc),
.y(Result)
);



endmodule

// mainfsm.v
```

```
module mainfsm (
    clk,
    reset,
    Op,
    Funct,
    IRWrite,
    AdrSrc,
    ALUSrcA,
    ALUSrcB,
    ResultSrc,
    NextPC,
    RegW,
    MemW,
    Branch,
    ALUOp,
    MULL_Identifier,
    WE4w,
    ResultControl,
    FP_identifier
);
    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 [14:0] controls;
    localparam [3:0] FETCH = 0;
    localparam [3:0] BRANCH = 9;
    localparam [3:0] DECODE = 1;
    localparam [3:0] EXECUTEI = 7;
    localparam [3:0] EXECUTER = 6;
    localparam [3:0] MEMADR = 2;
    localparam [3:0] UNKNOWN = 10;
```

```
localparam [3:0] ALUWB = 8;
localparam [3:0] MEMRD = 3;
localparam [3:0] MEMWR = 5;
localparam [3:0] MEMWB = 4;

//nuevo

input wire [3:0] MULL_Identifier;
output wire WE4w;
output wire ResultControl;
input wire [4:0] FP_identifier;



// 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
        EXECUTER:
            nextstate = ALUWB;
        EXECUTEI:
```

```
            nextstate = ALUWB;
        MEMADR:
            if(Funct[0])
                nextstate=MEMRD;
            else
                nextstate=MEMWR;
        MEMRD:
            nextstate=MEMWB;

        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

//NUEVO: se anadio un bit
always @(*)
    case (state)
        FETCH: controls = 15'b100010100110000;
        DECODE: controls = 15'b000000100110000;
        // 0 0 0 0 0 0 10 01 10 0;
        EXECUTER:
        if(FP_identifier == 5'b 11111 & !Funct[5] )
            controls = 15'b000000100000101;
        else
            controls = 15'b000000100000100;
        // 0 0 0 0 0 0 10 00 00 1;
        EXECUTEI: controls = 15'b000000100001100;
        // 0 0 0 0 0 0 10 00 01 1;
        ALUWB:
            if(MULL_Identifier== 4'b1001 & Funct[3] & !Funct[5])
                controls = 15'b000100000000110;
            else
                controls = 15'b000100000000100;
        // 0 0 0 0 0 0 10 00 00 1;
        MEMADR: controls = 15'b000000100001000;
        // 0 0 0 0 0 0 10 00 01 0;
        MEMWR: controls =  15'b001001000001000;
        //0 0 1 0 0 1 00 00 01 0;
        MEMRD: controls = 15'b000001000001000;
```

```
            MEMWB: controls = 15'b000101010001000;
            // 0 0 0 0 0 1 00 00 01 0;

            BRANCH: controls = 15'b010000101001000;
            // 0 0 0 0 0 0 10 01 10 0;

            default: controls = 15'bxxxxxxxxxxxxxxx;
        endcase
    assign {NextPC, Branch, MemW, RegW, IRWrite, AdrSrc, ResultSrc,
 ALUSrcA, ALUSrcB, ALUOp, WE4w,ResultControl} = controls;
endmodule

 /// TESTVECTORS

// alu_tb.v (Para MUL)

`timescale 1ns/1ns
module alu_tb();
  reg [31:0] a,b;
  reg [7:0] ALUControl;
  reg clk, reset;
  reg [103:0] testvector[1000:0];
  wire [31:0] Result;
  reg [31:0] Result_expected; // to compare y output
  reg [31:0] vectornum; // check testvector number
  reg [31:0] errors; // error counter

  alu alu_dut(.a(a), .b(b), .ALUControl(ALUControl), .Result(Result));

  always// always execute
   begin
     clk=1; #5; clk=0; #5;   //period? Tc= 10
   end

  initial // one exec
    begin
      $readmemh("testvectormul.tv",testvector); //read in hexa
      errors=0;
      vectornum=0;
      reset=1; #17;reset=0;
    end

  always @(posedge clk)
    begin
```

```
      Result_expected = testvector[vectornum][31:0];
      a = testvector[vectornum][63:32];
      b = testvector[vectornum][95:64];
      ALUControl = testvector[vectornum][103:96];
    end

  always @(negedge clk)
    begin
      if (~reset)
        begin
          if ((Result !== Result_expected))  // ===, ==
       begin
             $display("testvector: %h",testvector[vectornum]);
             $display("Vectornum: %d",vectornum);
             $display("Error input a: %h",{a});
             $display("Error input b: %h",{b});
             $display("Error input ALUControl: %h",{ALUControl});
             $display("output Result:%h, Result_expected:%b
",Result,Result_expected);
              errors=errors+1;
            end
       vectornum=vectornum+1;

          if (testvector[vectornum][0] === 1'bx)
            begin
              $display("total errors: %d",errors);
              $finish;
            end
        end
    end

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

//testvectorMUL.tv

4_00000032_0000000A_000001F4
8_00000032_0000000A_000001F4
c_00000032_0000000A_000001F4

// fp_tb.v (32 bits)
```

```
‘timescale 1ns/1ns
module fp_tb;
  reg [31:0] a,b;
  reg ALUControl;
  reg clk, reset;
  reg [96:0] testvector[1000:0];
  wire [31:0] Result;
  reg [31:0] Result_expected; // to compare y output
  reg [31:0] vectornum; // check testvector number
  reg [31:0] errors; // error counter

  fp fp_dut(.a(a), .b(b), .ALUControl(ALUControl), .Result(Result));

  always// always execute
   begin
     clk=1; #5; clk=0; #5;   //period? Tc= 10
   end

  initial // one exec
    begin
      $readmemh("testvectorfp.tv",testvector); //read in hexa
      errors=0;
      vectornum=0;
      reset=1; #17;reset=0;
    end

  always @(posedge clk)
    begin
      Result_expected = testvector[vectornum][31:0];
      a = testvector[vectornum][63:32];
      b = testvector[vectornum][95:64];
      ALUControl = testvector[vectornum][96];
    end

  always @(negedge clk)
    begin
      if (~reset)
        begin
          if ((Result !== Result_expected))  // ===, ==
       begin
             $display("testvector: %h",testvector[vectornum]);
             $display("Vectornum: %d",vectornum);
             $display("Error input a: %h",{a});
```

```
              $display("Error input b: %h",{b});
              $display("Error input ALUControl: %h",{ALUControl});
              $display("output Result:%h, Result_expected:%b",
Result,Result_expected);
               errors=errors+1;
             end
       vectornum=vectornum+1;

          if (testvector[vectornum][0] === 1'bx)
            begin
              $display("total errors: %d",errors);
              $finish;
            end
        end
    end

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

// fp_tb16 (16-bits)

`timescale 1ns/1ns
module fp_tb;
  reg [15:0] a,b;
  reg ALUControl;
  reg clk, reset;
  reg [96:0] testvector[1000:0];
  wire [15:0] Result;
  reg [15:0] Result_expected; // to compare y output
  reg [15:0] vectornum; // check testvector number
  reg [15:0] errors; // error counter

  fp16 fp_dut(.a(a), .b(b), .ALUControl(ALUControl), .Result(Result));

  always// always execute
   begin
     clk=1; #5; clk=0; #5;   //period? Tc= 10
   end

  initial // one exec
    begin
```

```
      $readmemh("testvector2fp.tv",testvector); //read in hexa
      errors=0;
      vectornum=0;
      reset=1; #17;reset=0;
    end

always @(posedge clk)
    begin
      Result_expected = testvector[vectornum][15:0];
      a = testvector[vectornum][31:16];
      b = testvector[vectornum][47:32];
      ALUControl = testvector[vectornum][48];
    end

  always @(negedge clk)
    begin
      if (~reset)
        begin
          if ((Result !== Result_expected))  // ===, ==
       begin
             $display("testvector: %h",testvector[vectornum]);
             $display("Vectornum: %d",vectornum);
             $display("Error input a: %h",{a});
             $display("Error input b: %h",{b});
             $display("Error input ALUControl: %h",{ALUControl});
             $display("output Result:%h, Result_expected:
%b",Result,Result_expected);
              errors=errors+1;
            end
       vectornum=vectornum+1;

          if (testvector[vectornum][0] === 1'bx)
            begin
              $display("total errors: %d",errors);
              $finish;
            end
        end
    end

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

```
// testvectorfp.tv

1_40800000_40800000_41800000
1_40200000_40200000_40c80000
1_40e00000_c1080000_c26e0000
0_41280000_41280000_41a80000

// testvectofp2.tv

1_4000_4200_4600
0_4128_4128_41a8


   ////   Exclusivo de Single-cycle:


// decode .v

module decode (
Op,
Funct,
Rd,
FlagW,
PCS,
RegW,
MemW,
MemtoReg,
ALUSrc,
ImmSrc,
RegSrc,
ALUControl,
MULL_Identifier,  // nuevo input
WE4w, //
NewSource, //
IsMul, //
FP_identifier,
FPControl,
ResultControl,
BIT_identifier,
OP_identifier
);
input wire [1:0] Op;
input wire [5:0] Funct;
```

```
input wire [3:0] Rd;
input wire [3:0] MULL_Identifier; //nuevo
output reg [1:0] FlagW;
output wire PCS;
output wire RegW;
output wire MemW;
output wire MemtoReg;
output wire ALUSrc;
output wire WE4w; //nueva senal
output wire NewSource; //nuevo
output wire [1:0] ImmSrc;
output wire [1:0] RegSrc;
output reg [3:0] ALUControl;
reg [13:0] controls;
wire Branch;
wire ALUOp;
//nuevo
output wire IsMul; //
output wire ResultControl;//
output reg [1:0] FPControl;
input wire [4:0] FP_identifier;
input wire [3:0] BIT_identifier;
input wire [3:0] OP_identifier;



always @(*)
casex (Op)
2'b00:
if (Funct[5])
controls = 14'b 00001010010000;
else if(FP_identifier == 5'b11111)
controls= 14'b 00000010010101;
else if (MULL_Identifier== 4'b1001 & Funct[3]) //umull y smull
controls = 14'b 00000010011100;    // ALUOPnuevo
else if (MULL_Identifier== 4'b1001) //mull
controls = 14'b 00000010010110;
else
controls = 14'b 00000010010000;
2'b01:
if (Funct[0])
controls = 14'b 00011110000000;
else
controls = 14'b 10011101000000;
```

```
2'b10: controls = 14'b 01101000100000;
default: controls = 14'b xxxxxxxxxxxxxx;
endcase
assign {RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW, Branch,
 ALUOp,WE4w,NewSource,IsMul,ResultControl} = controls;
always @(*)
if (FP_identifier == 5'b 11111) begin
case (BIT_identifier)
4'b 0000: // 32bit
if(OP_identifier == 4'b1111) //32mul
FPControl = 2'b11;
else //32add
FPControl= 2'b 01;
4'b 1111: // 16 bit
if(OP_identifier == 4'b1111) //16mul
FPControl= 2'b10;
else//16add
FPControl= 2'b00;
default: FPControl = 2'bxx;
endcase
end
else if (MULL_Identifier== 4'b1001) begin
case (Funct [3:1])
3'b 000: ALUControl=4'b 0100; //MULL
3'b 100: ALUControl=4'b 1000; //UMULL
3'b 110: ALUControl=4'b 1100; //SMULL
endcase
FlagW[1]=Funct[0]; // nz
FlagW[0]=1'b 0; //cv
end
else if (ALUOp) begin
case (Funct[4:1])
4'b0100: ALUControl = 4'b0000;
4'b0010: ALUControl = 4'b0001;
4'b0000: ALUControl = 4'b0010;
4'b1100: ALUControl = 4'b0011;
default: ALUControl = 4'bxxxx;
endcase
FlagW[1] = Funct[0];
FlagW[0] = Funct[0] & ((ALUControl == 4'b0000) | (ALUControl == 4'b0001));
end
else begin
ALUControl = 4'b0000; //
FlagW = 2'b00;
```

```
end
assign PCS = ((Rd == 4'b1111) & RegW) | Branch;
endmodule

// controller.v
module controller (
clk,
reset,
Instr,
ALUFlags,
RegSrc,
RegWrite,
ImmSrc,
ALUSrc,
ALUControl,
MemWrite,
MemtoReg,
PCSrc,
WE4,
NewSource,
IsMul,
ResultControl,
FPControl
);
input wire clk;
input wire reset;
input wire [31:0] Instr; // paso la instr completa
input wire [3:0] ALUFlags;
output wire [1:0] RegSrc;
output wire RegWrite;
output wire [1:0] ImmSrc;
output wire ALUSrc;
output wire [3:0] ALUControl;
output wire MemWrite;
output wire MemtoReg;
output wire PCSrc;
// nuevo
output wire WE4;//
output wire NewSource;//
output wire IsMul;
//
wire [1:0] FlagW;
wire PCS;
wire RegW;
```

```
wire MemW;
//nuevo
wire WE4w;
output wire ResultControl;
output wire [1:0] FPControl;

decode dec(
.Op(Instr[27:26]),
.Funct(Instr[25:20]),
.Rd(Instr[15:12]),
.FlagW(FlagW),
.PCS(PCS),
.RegW(RegW),
.MemW(MemW),
.MemtoReg(MemtoReg),
.ALUSrc(ALUSrc),
.ImmSrc(ImmSrc),
.RegSrc(RegSrc),
.ALUControl(ALUControl),
.MULL_Identifier(Instr[7:4]), // nuevo
.WE4w(WE4w),
.NewSource(NewSource),
.IsMul(IsMul),
.FPControl (FPControl),
.ResultControl(ResultControl),
.BIT_identifier(Instr[19:16]),
.OP_identifier(Instr[7:4]),
.FP_identifier(Instr[24:20])
);
condlogic cl(
.clk(clk),
.reset(reset),
.Cond(Instr[31:28]),
.ALUFlags(ALUFlags),
.FlagW(FlagW),
.PCS(PCS),
.RegW(RegW),
.MemW(MemW),
.PCSrc(PCSrc),
.RegWrite(RegWrite),
.MemWrite(MemWrite),
.WE4w(WE4w),
.WE4(WE4)
);
```

```
endmodule

// datapath.v
module datapath (
clk,
reset,
RegSrc,
RegWrite,
ImmSrc,
ALUSrc,
ALUControl,
MemtoReg,
PCSrc,
ALUFlags,
PC,
Instr,
newALUResult, //nuevo
WriteData,
ReadData,
WE4,
NewSource,
IsMul,
ResultControl,
FPControl
);
input wire clk;
input wire reset;
input wire [1:0] RegSrc;
input wire RegWrite;
input wire [1:0] ImmSrc;
input wire ALUSrc;
input wire [3:0] ALUControl;
input wire MemtoReg;
input wire PCSrc;
output wire [3:0] ALUFlags;
output wire [31:0] PC;
input wire [31:0] Instr;
wire [31:0] ALUResult;
output wire [31:0] WriteData;
input wire [31:0] ReadData;
wire [31:0] PCNext;
wire [31:0] PCPlus4;
wire [31:0] PCPlus8;
wire [31:0] ExtImm;
```

```
wire [31:0] SrcA;
wire [31:0] SrcB;
wire [31:0] Result;
wire [3:0] RA1;
wire [3:0] RA2;
//nuevo
wire [3:0] newRA1;
wire [3:0] newRA2;
wire [3:0] newRA3;
wire [31:0] ALUResult2; // nuevo
input wire  WE4; // nuevo
wire [31:0] Result2;//
input wire NewSource; // nuevo
input wire IsMul; //
input wire ResultControl;
input wire [1:0]FPControl;
wire [31:0] FPResult;
output wire [31:0] newALUResult;
//nuevo
assign Result2 = ALUResult2;
mux2 #(32) pcmux(
.d0(PCPlus4),
.d1(Result),
.s(PCSrc),
.y(PCNext)
);
flopr #(32) pcreg(
.clk(clk),
.reset(reset),
.d(PCNext),
.q(PC)
);
adder #(32) pcadd1(
.a(PC),
.b(32'b100),
.y(PCPlus4)
);
adder #(32) pcadd2(
.a(PCPlus4),
.b(32'b100),
.y(PCPlus8)
);
mux2 #(4) ra1mux(
.d0(Instr[19:16]),
```

```
.d1(4'b1111),
.s(RegSrc[0]),
.y(RA1)
);
mux2 #(4) ra2mux(
.d0(Instr[3:0]),
.d1(Instr[15:12]),
.s(RegSrc[1]),
.y(RA2)
);


//NUEVO
mux2 #(32) FPmux(
.d0(ALUResult),
.d1(FPResult),
.s(ResultControl),
.y(newALUResult)
);
mux2 #(4) ra1mux2(
.d0(RA1),
.d1(Instr[11:8]),
.s(NewSource),
.y(newRA1)
);
mux2 #(4) ra2mux2(
.d0(RA2),
.d1(Instr[3:0]),
.s(NewSource),
.y(newRA2)
);

mux2 #(4) ra3mux(
.d0(Instr[15:12]),
.d1(Instr[19:16]),
.s(IsMul),
.y(newRA3)
);
//NUEVO


regfile rf(
.clk(clk),
.we3(RegWrite),
```

```
.ra1(newRA1),
.ra2(newRA2),
.wa3(newRA3),
.wd3(Result),
.r15(PCPlus8),
.rd1(SrcA),
.rd2(WriteData),
.WE4(WE4),
.WD4(Result2),
.WA4(Instr[19:16])
);
mux2 #(32) resmux(
.d0(newALUResult),
.d1(ReadData),
.s(MemtoReg),
.y(Result)
);
extend ext(
.Instr(Instr[23:0]),
.ImmSrc(ImmSrc),
.ExtImm(ExtImm)
);
mux2 #(32) srcbmux(
.d0(WriteData),
.d1(ExtImm),
.s(ALUSrc),
.y(SrcB)
);
alu alu(
SrcA,
SrcB,
ALUControl,
ALUResult,
ALUFlags,
ALUResult2
);

FPU FPU (
.a(SrcA),
.b(SrcB),
.Result(FPResult),
.ALUControl(FPControl)
);
endmodule
```

```
// arm.v

module arm (
clk,
reset,
PC,
Instr,
MemWrite,
newALUResult,
WriteData,
ReadData
);
input wire clk;
input wire reset;
output wire [31:0] PC;
input wire [31:0] Instr;
output wire MemWrite;
//output wire [31:0] ALUResult;
output wire [31:0] WriteData;
input wire [31:0] ReadData;
wire [3:0] ALUFlags;
wire RegWrite;
wire ALUSrc;
wire MemtoReg;
wire PCSrc;
wire [1:0] RegSrc;
wire [1:0] ImmSrc;
wire [3:0] ALUControl;
wire WE4;// nuevo
wire NewSource;// nuevo
wire IsMul; // nuevo
output wire [31:0] newALUResult;
wire ResultControl;
wire [1:0]FPControl;

controller c(
.clk(clk),
.reset(reset),
.Instr(Instr[31:0]),
.ALUFlags(ALUFlags),
.RegSrc(RegSrc),
.RegWrite(RegWrite),
.ImmSrc(ImmSrc),
```

```
.ALUSrc(ALUSrc),
.ALUControl(ALUControl),
.MemWrite(MemWrite),
.MemtoReg(MemtoReg),
.PCSrc(PCSrc),
.WE4(WE4),
.NewSource(NewSource),
.IsMul(IsMul),
.ResultControl(ResultControl),
.FPControl(FPControl)
);
datapath dp(
.clk(clk),
.reset(reset),
.RegSrc(RegSrc),
.RegWrite(RegWrite),
.ImmSrc(ImmSrc),
.ALUSrc(ALUSrc),
.ALUControl(ALUControl),
.MemtoReg(MemtoReg),
.PCSrc(PCSrc),
.ALUFlags(ALUFlags),
.PC(PC),
.Instr(Instr),
.newALUResult(newALUResult),
.WriteData(WriteData),
.ReadData(ReadData),
.WE4(WE4),
.NewSource(NewSource), //
.IsMul(IsMul),
.ResultControl(ResultControl),
.FPControl(FPControl)
);
endmodule

// top.v
module top (
clk,
reset,
WriteData,
DataAdr,
MemWrite
);
input wire clk;
```

```
input wire reset;
output wire [31:0] WriteData;
output wire [31:0] DataAdr;
output wire MemWrite;
wire [31:0] PC;
wire [31:0] Instr;
wire [31:0] ReadData;
arm arm(
.clk(clk),
.reset(reset),
.PC(PC),
.Instr(Instr),
.MemWrite(MemWrite),
.newALUResult(DataAdr), // nuevo
.WriteData(WriteData),
.ReadData(ReadData)
);
imem imem(
.a(PC),
.rd(Instr)
);
dmem dmem(
.clk(clk),
.we(MemWrite),
.a(DataAdr),
.wd(WriteData),
.rd(ReadData)
);
endmodule

// regfile.v

module regfile (
clk,
we3,
ra1,
ra2,
wa3,
wd3,
r15,
rd1,
rd2,
WE4,
WD4,
```

```
WA4
);
input wire clk;
input wire we3;
input wire [3:0] ra1;
input wire [3:0] ra2;
input wire [3:0] wa3;
input wire [31:0] wd3;
input wire [31:0] r15;
output wire [31:0] rd1;
output wire [31:0] rd2;

input wire WE4; //nuevo
input wire [31:0] WD4; //nuevo
input wire [3:0] WA4;

reg [31:0] rf [14:0];
always @(posedge clk) begin
if (we3) begin
rf[wa3] <= wd3;
end
if (WE4)
begin
rf[WA4] <= WD4;   //NUEVO
end
end
assign rd1 = (ra1 == 4'b1111 ? r15 : rf[ra1]);
assign rd2 = (ra2 == 4'b1111 ? r15 : rf[ra2]);
endmodule

// condlogic.v

module condlogic (
clk,
reset,
Cond,
ALUFlags,
FlagW,
PCS,
RegW,
MemW,
PCSrc,
RegWrite,
MemWrite,
```

```
WE4w,
WE4
);
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 RegW;
input wire MemW;
input wire WE4w;
output wire PCSrc;
output wire RegWrite;
output wire MemWrite;
output wire WE4;
wire [1:0] FlagWrite;
wire [3:0] Flags;
wire CondEx;
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 FlagWrite = FlagW & {2 {CondEx}};
assign RegWrite = RegW & CondEx;
assign MemWrite = MemW & CondEx;
assign PCSrc = PCS & CondEx;
assign WE4= WE4w & CondEx; // nuevo
endmodule
```

4. Draw a diagram for your new single and multi-cycle processor

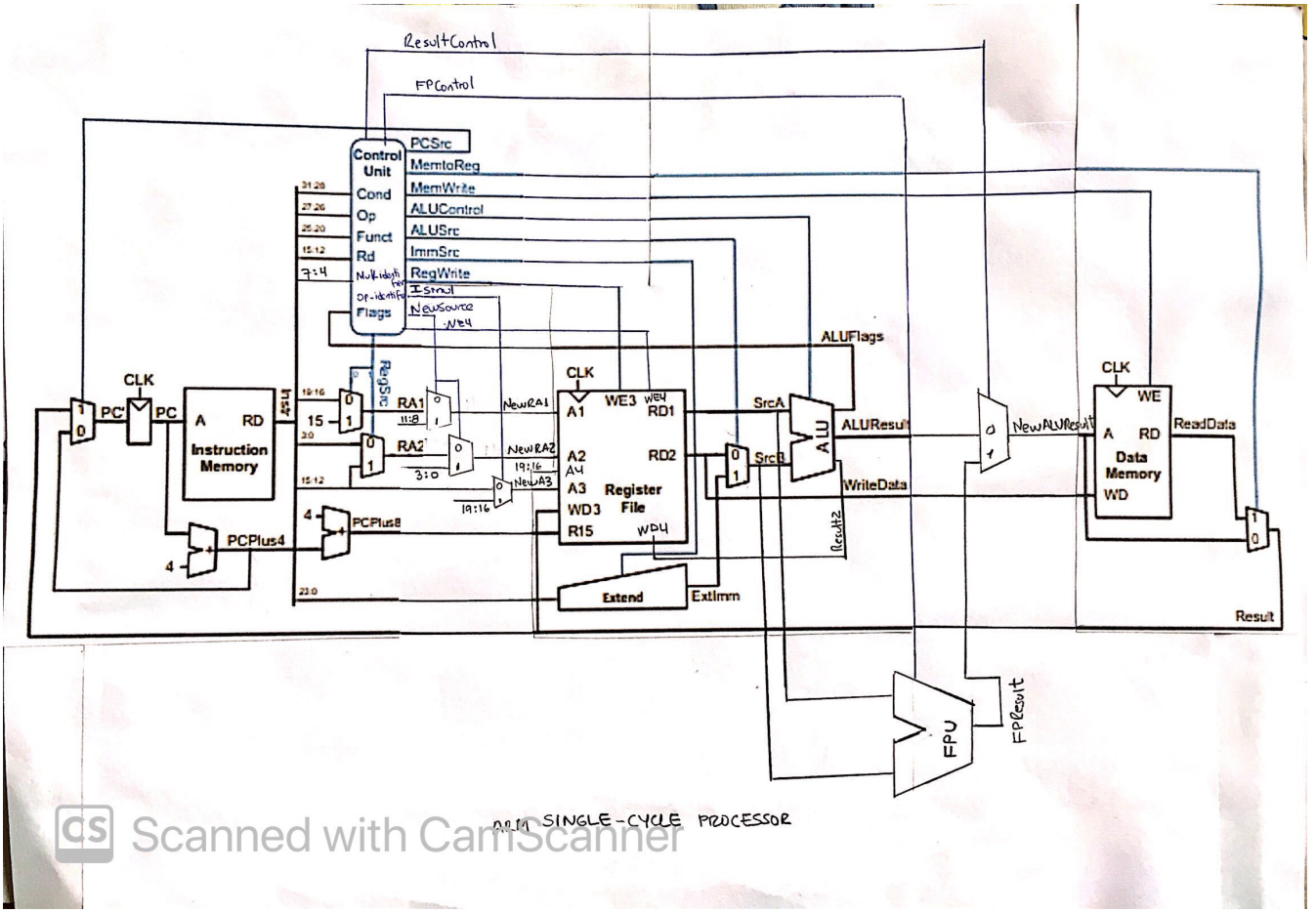

Figure 0.1: Modified Single-Cycle Processor

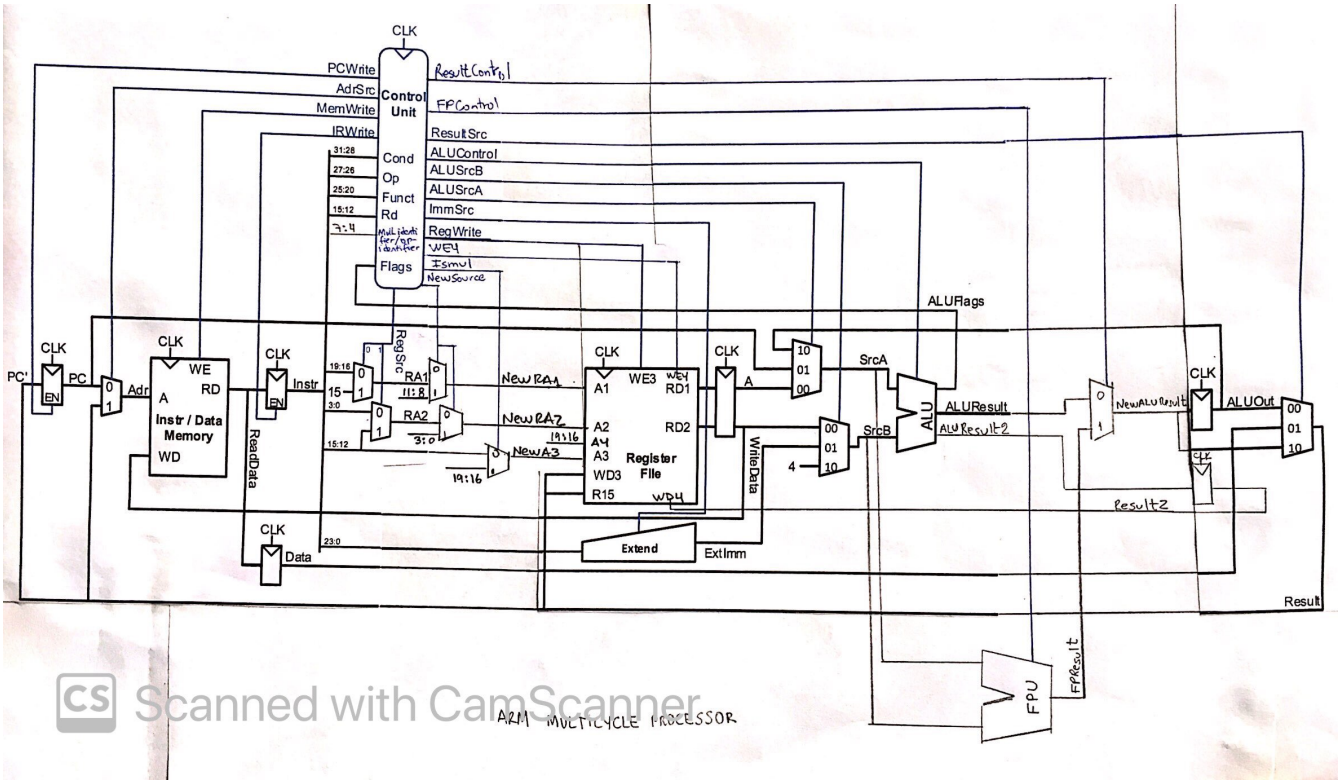

Figure 0.2: Modified Multi-Cycle Processor

5. Justify and explain your design decisions

   En ambos procesadores, el concepto del diseño es idéntico. Lo único que cambia es la implementación, específicamente en qué momento se activan las señales nuevas. Por este motivo ambos diagramas son casi idénticos.

   a) Control Unit: En el Control Unit se agregaron las señales de output de control llamadas: WE4,IsMul,NewSource, Result Control y FP Control. WE4 es un write enable que se prende cuando la instrucción tiene dos registros de destino (umull y smull). IsMul y Newsource son señales que funcionan como selector para los muxes antes de entrar al regfile. Estas señales tienen el propósito de seleccionar el número de registro adecuado cuando se trata de instrucciones de multiplicación ya que los bits en la instrucción que determinan los registros de los operandos y destino cambian de posición cuando se trata de este tipo de instrucciones. FPControl, es análogo al ALUControl y va al FPU cuando la instrucción utiliza punto flotante.Por último, ResultControl funciona como selector para el mux que tiene como input al ALUResult y FPResult. Tiene el propósito de seleccionar el resultado adecuado dependiendo si utilizo el ALU o el FPU. Cabe destacar que, a este módulo ahora ingresa la instrucción completa ya que es necesario para poder identificar cuando se trata de una multiplicación u operación de punto flotante. Se utilizaron senales como MUL_identifier o FP_identifier que cumplen con el propósito de identificar estas instrucciones nuevas.En el código se puede apreciar mejor el proceso de identificación. También, para la multiplicación, se pasa los write enables al cond logic para que solo escriba en los registros si se cumple el condcheck.

   b) Datapath: Como mencionado en la explicación del decoder, se agregaron unos cuantos muxes, que junto a las señales de control nuevas, permiten que el procesador funcione correctamente. También se agregaron unos cables salientes de los muxes tal como se muestra en el diagrama.

   c) ALU: Para poder implementar umull, smull, y mul, se añadieron dos bits al ALUControl, teniendo ahora 4 bits en total. Los dos bits originales de esta señal se mantuvieron fijos para no alterar el resto de instrucciones. Para umull, smull y mul, el encoding es 10xx, 11xx, 01xx. respectivamente. Como umull y smull utilizan dos registros de salida, se calcula el resultado de la multiplicación en un reg de bits llamado aux. Luego de la operación, se le asigna a Result2 los 32 bits más significativos y a Result los 32 bits menos significativos. El cable Result sigue su camino normal mientras que Result2, se va directamente al regfile. En caso la instrucción sea smull o umull, se activa una señal llamada WE4 para que escriba en el registro. Adicionalmente, se agregó una lógica para las nuevas ALUFlags en el caso de multiplicación, ya que estas instrucciones sólo pueden activar las flags n y z.

   d) Register file: Se añadieron tres entradas nuevas a este módulo: WE4,WA4 y WD4. WE4 es el write enable que se activa cuando la instrucción escribe

en dos registros de destino. WA4 indica en qué registro se va escribir la información. Por último, WD4 es el resultado del ALU que se guarda en el segundo registro de destino para operaciones de umull y smull.

e) FPU: El FPU es un módulo que contiene a dos módulos principales dentro: fp y fp16 para tratar con single-precision (32-bit) y half-precision (16-bit) respectivamente. El FPU recibe el FPControl el cual indica qué operación va a realizar. Para este encoding se utilizaron 2 bits: 11 (32 bit mul), 01 (32 bit add), 10 (16 bit mul) y 00(16 bit add). Como se puede observar, el primer bit indica qué operación es, 1 siendo mul y 0 siendo add. Por otro lado, el segundo bit, indica cuantos bits de precisión se van usar siendo 1 32 bits y 0 16 bits. Por último, se utiliza un mux para seleccionar el resultado correcto dependiendo de la instrucción realizada.

f) Encoding: El encodign asignado a las instrucciones FP es el siguiente:

```
1110 00011111 0000(32b) 0000(destino) 0000(op1) 1111(mul) 0000(op1) FPmul
1110 00011111 0000(32b) 0000(destino) 0000(op1) 0000(add) 0000(op2) FPadd
1110 00011111 1111(16b) 0000(destino) 0000(op1) 1111(mul) 0000(op2) FPmul16bit
1110 00011111 1111(16b) 0000 (destino) 0000(op1) 0000(add) 0000(op1) FPadd16bit
```

6. Include waveforms of the new signals from your single and multi-cycle processor that can show your work.

```
Para FP:
ADD: 60 + 10.5 = 70.5 = 428d0000 = 428d
MUL: 60 * 5 = 300 = 43960000 = 4396

Para MUL:
50 * 10 = 500 = 1F4
50 * -5 = -250 = FFFFFF06
```

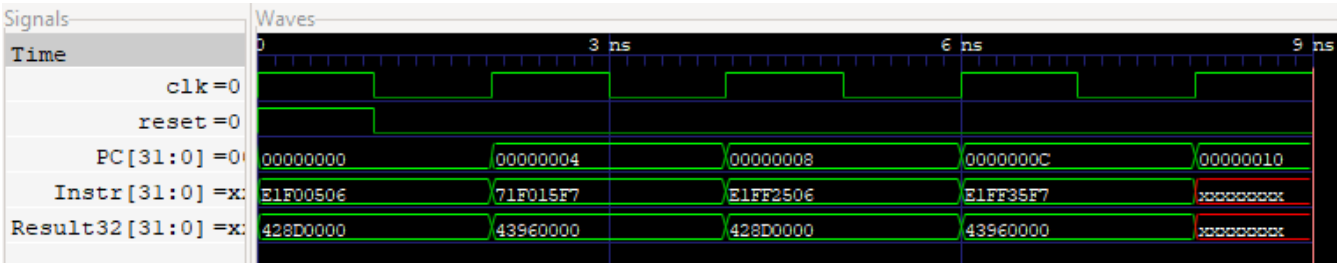

Figure 0.3: Operaciones FP Single-Cycle

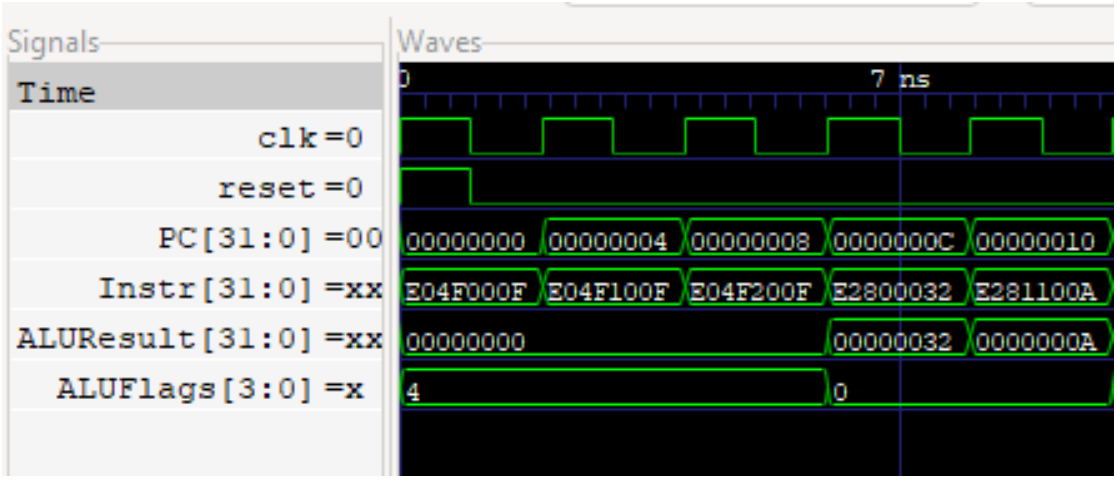

Figure 0.4: Inicialización de valores para MUL Single-Cycle

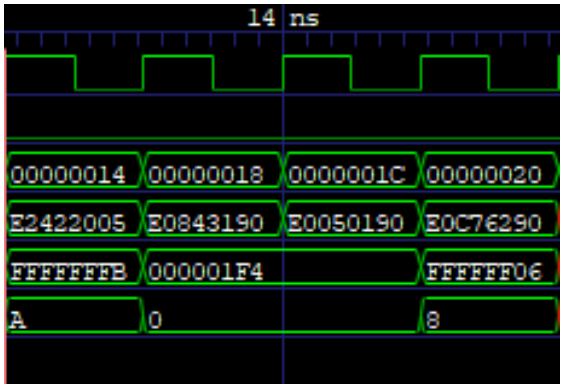

Figure 0.5: Operaciones MUL Single-Cycle

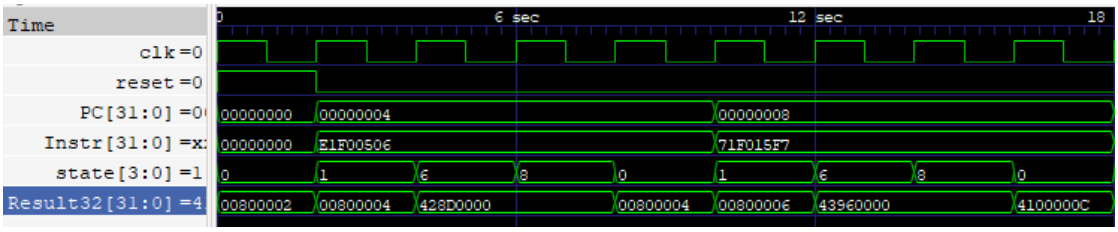

Figure 0.6: Operaciones FP Multi-Cycle (Parte 1 de 2)

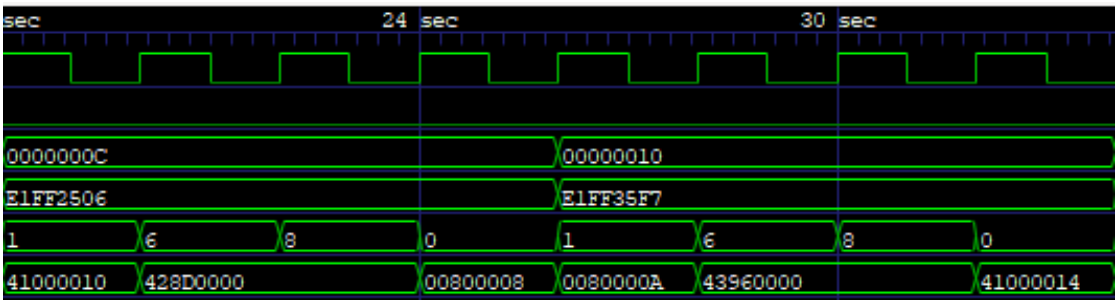

Figure 0.7: Operaciones FP Multi-Cycle (Parte 2 de 2)

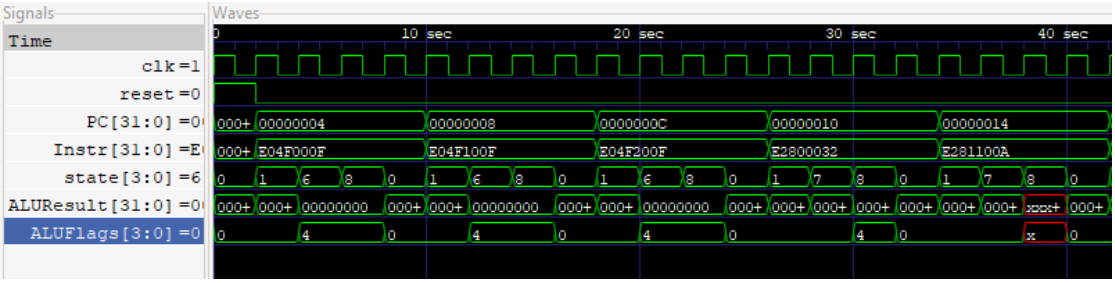

Figure 0.8: Inicialización de valores para MUL Multi-Cycle

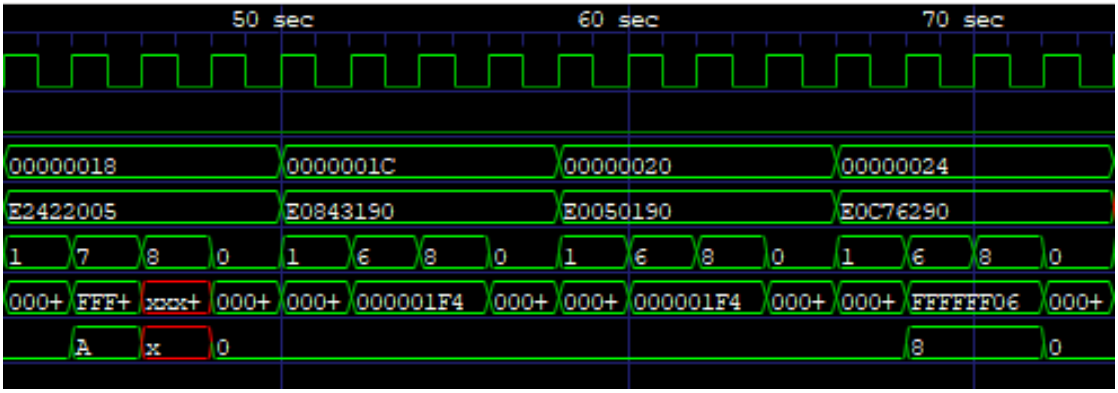

Figure 0.9: Operaciones MUL Multi-Cycle (Los valores que aparecen cortados no son necesarios para mostrar el funciomiento de las instruciones)