

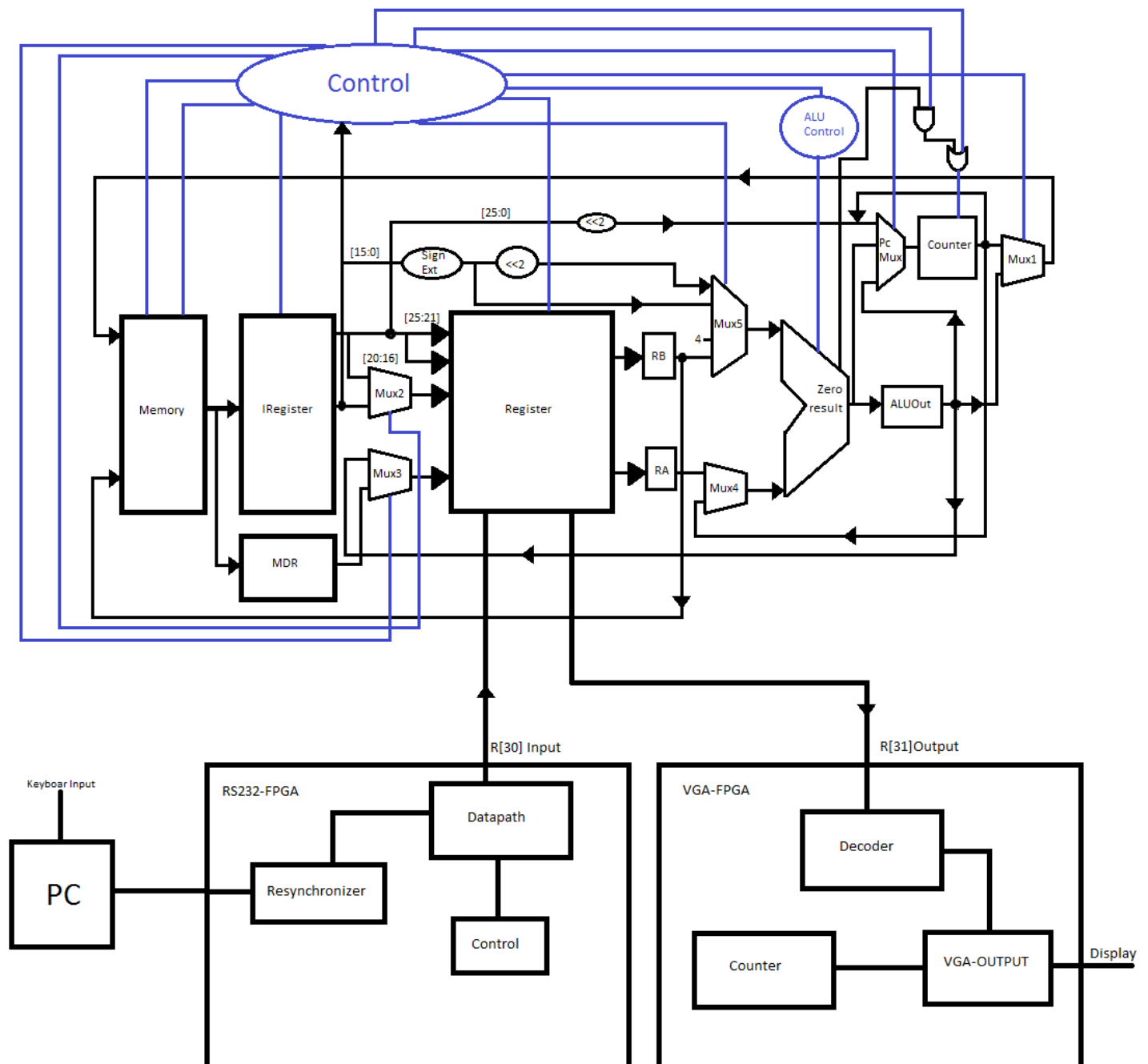
CINVESTAV

Alumno: Efraín Arrambide Barrón

Proyecto Final

Docente: Doc. Mariano Aguirre

PADTS 19



Se realizó implementación en mi MIPS multiciclo porque su frecuencia de operación de más alta que uniciclo, comprado con el pipeline es menos complejo y más practico para realizar está implementación, además por su velocidad de operación resolvía la detección del bit READY. El programa(software) que se implementó, son unos contadores para el Add(que columna) y Color para el color de cada columna de manera que si presiona flecha arriba el contador de Color se incrementa 1, en caso de presionar hacia los lados el contador Add se suma o se resta dependiendo si fue hacia la izquierda o derecha.

Primero se hizo un análisis de la duración del bit Ready del protocolo RS232 sabemos que corre a una velocidad de 9600baud esto nos da como resultado un periodo de $T=104166.667$ nano segundo por bit. Y en nuestro procesador tiene un periodo de $T=20$ nano-segundo, por lo que es tiempo suficiente para poder detectar el bit de ready.

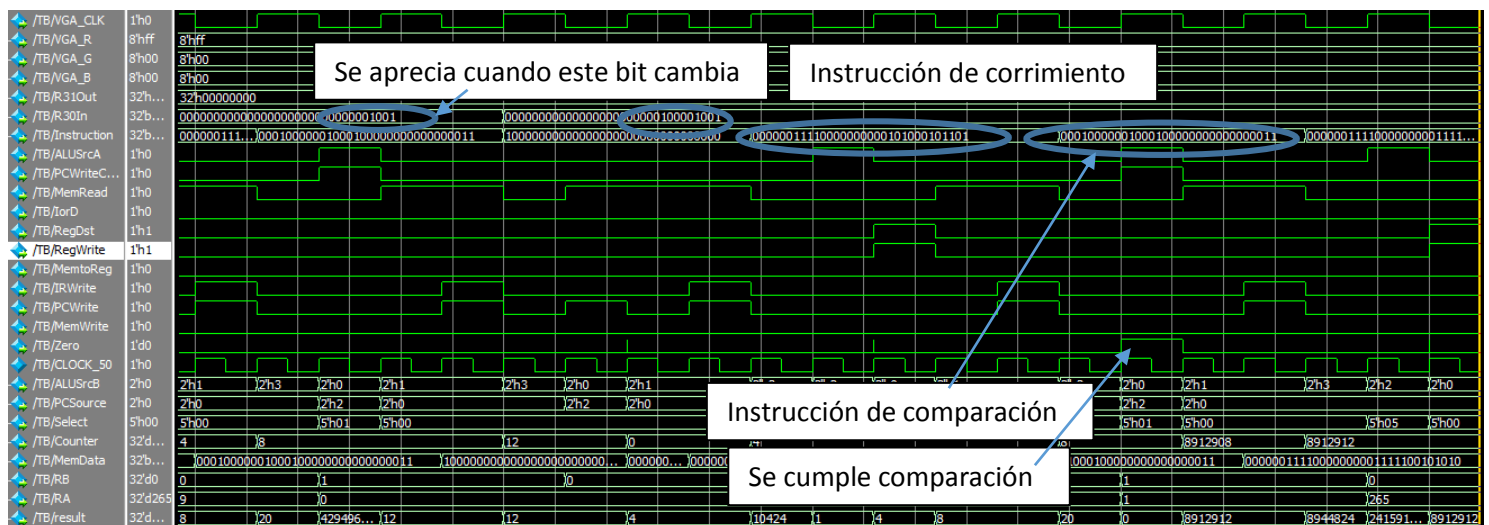
El bit de Ready se activa cada vez que hay un nuevo dato es transferido a través de nuestro protocolo RS232.

Con las siguientes 3 instrucciones se monitorea este bit[8], primero se hace un corrimiento, se compara y si resulta igual a R[2], por default el registro 2 es un 1, entonces si se cumple esta condición sale de este loop, en caso contrario seguirá haciendo esta acción infinitamente.

00000011110000000000101000101101 //R[1]=R[30]>>8 Hace corrimiento para ver el bit rdy

00010000001000100000000000000011 //R[1]=R[2] Jump 3 compara con el R[2] para saber si esta enable

10000000000000000000000000000000 //Jump 0 Regresa a dirección 0 para verificar nuevamente si rdy no ha cambiado

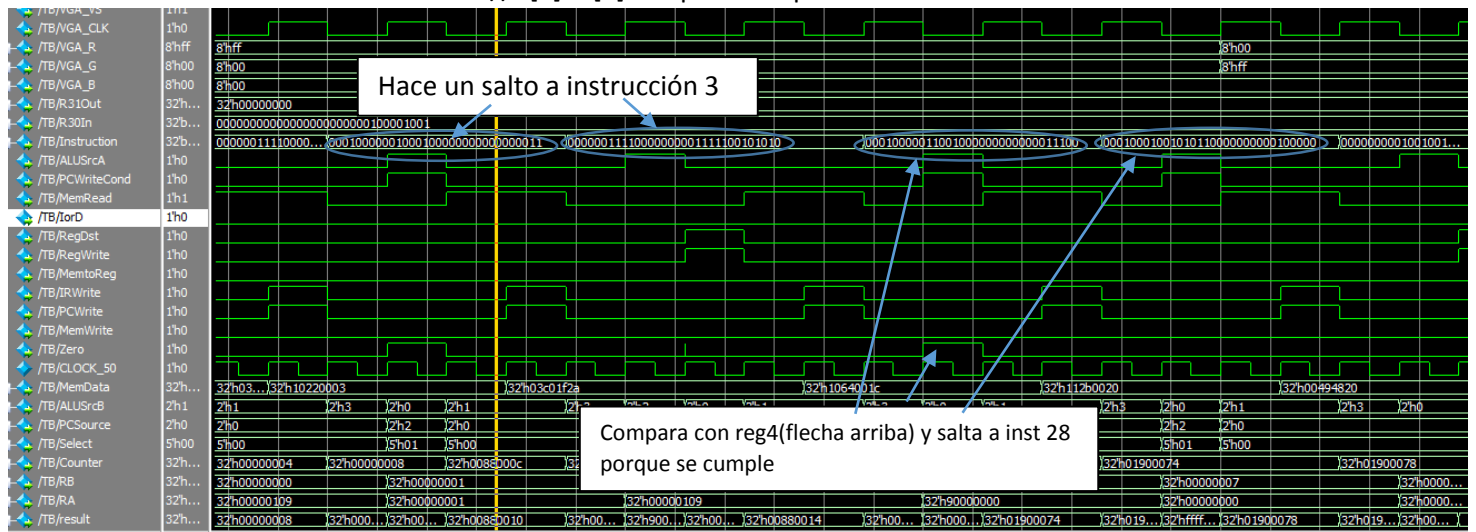


00000011110000000000111100101010 //R[3]=R[30]<<28 Corr verf MSB y saber tipo UP,DOWN, LEFT,RIGHT,

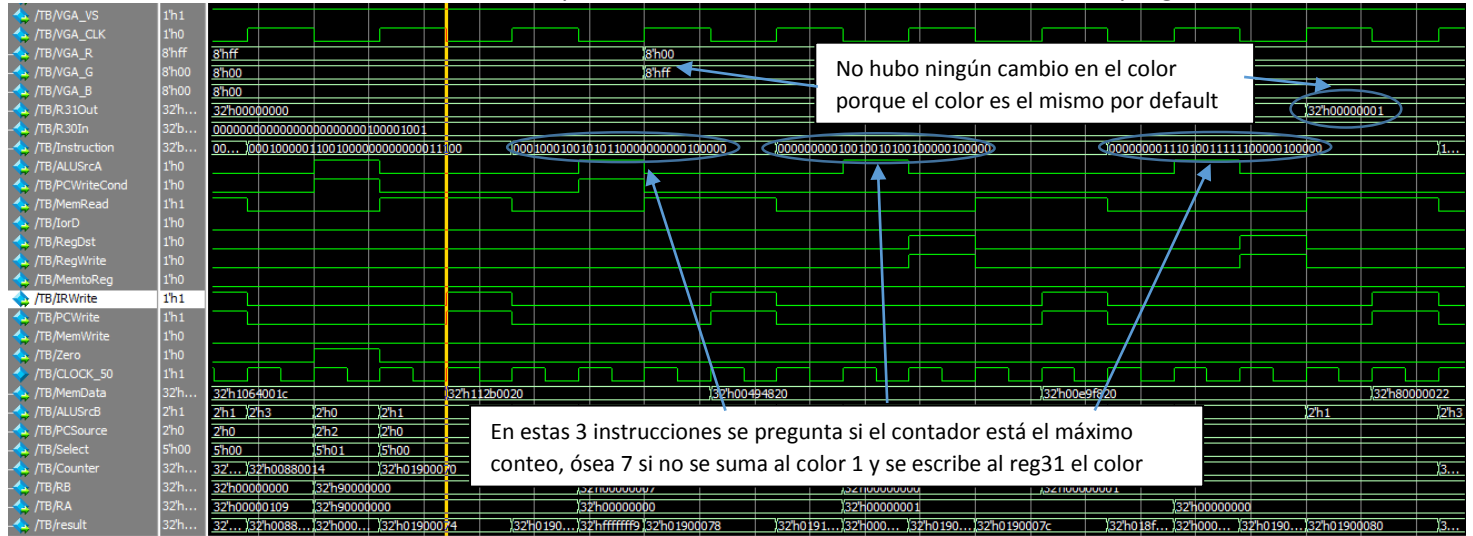
00010000011001000000000000001100 //R[3]=R[4] Jump 28 Compara si es 1001 Color+

000100000110010100000000000010101 //R[3]=R[5] Jump 21 Compara si es 1011 Color-

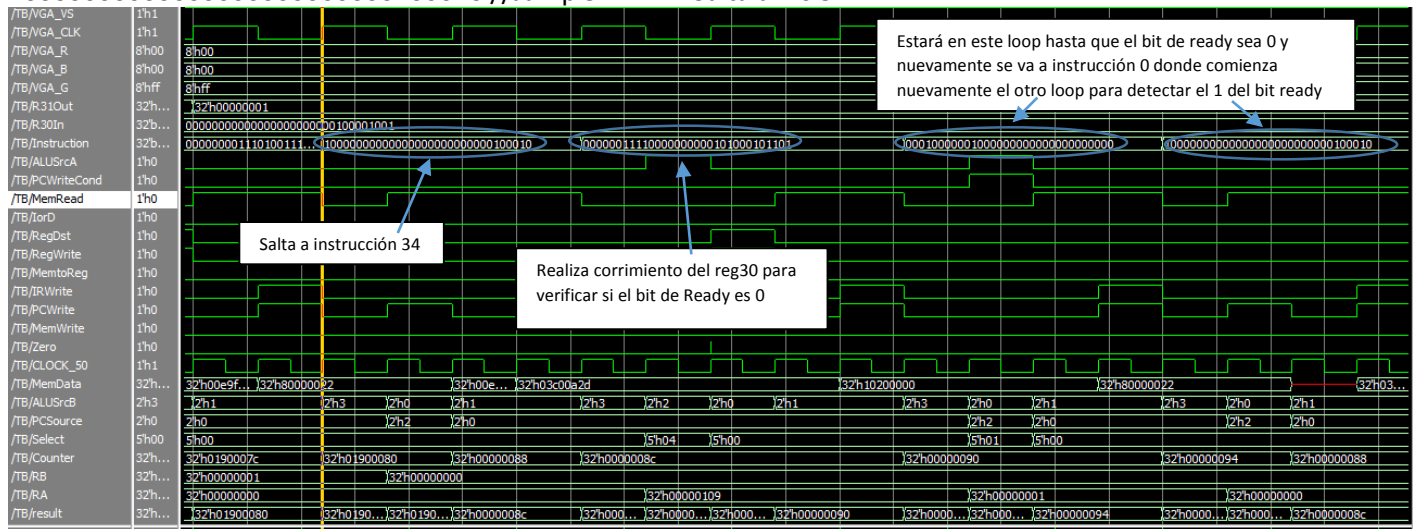
0001000001100110000000000000001110 //R[3]=R[6] Jump 14 Compara si es 1100 Add



00010001001010110000000000100000 //R[9]=R[11] Jump 32 Pregunta si es Color es 7
 00000000010010010100100000100000 //R[9]=R[9]+R[2] Suma al color 1
 00000000111010011111100000100000 //R[31]=R[7]+R[9] Escribe al Reg31 Add+Color
 1000000000000000000000000100010 //Jump 34 Salta al fin de programa



00000011110000000000101000101101 //R[1]=R[30]>>8 Corrimiento para ver bit de ready
 00010000001000000000000000000000 //R[1]=R[0] Jump 0 Salta a Instr 0
 1000000000000000000000000100010 //Jump 34 Salta a Ins 34



Jerarquía

Interfaces	2694 (0)	1287 (0)	0 (0)	1184	1	0	0	0	36	0
↳ Multibio:In1	2482 (0)	1151 (0)	0 (0)	1184	1	0	0	0	0	0
↳ DataPath:Mult1	2463 (0)	1138 (0)	0 (0)	1184	1	0	0	0	0	0
↳ ProgramCounter:M0	65 (65)	32 (32)	0 (0)	0	0	0	0	0	0	0
↳ Mux1:M1	6 (6)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ Memory:M2	0 (0)	0 (0)	0 (0)	1184	1	0	0	0	0	0
↳ altsyncram:mem_rtl_0	0 (0)	0 (0)	0 (0)	1184	1	0	0	0	0	0
↳ altsyncram_eki1:auto_generated	0 (0)	0 (0)	0 (0)	1184	1	0	0	0	0	0
↳ MDR:M3										
↳ IRegister:M4	32 (32)	32 (32)	0 (0)	0	0	0	0	0	0	0
↳ Mux2:M5	4 (4)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ Mux3:M6										
↳ Registro:M7	1839 (1839)	978 (978)	0 (0)	0	0	0	0	0	0	0
↳ RA:M8	32 (32)	32 (32)	0 (0)	0	0	0	0	0	0	0
↳ RB:M9	32 (32)	32 (32)	0 (0)	0	0	0	0	0	0	0
↳ Mux4:M10	32 (32)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ Mux5:M11	47 (47)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ PCMux:M12	7 (7)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ AluOut:M13	32 (32)	32 (32)	0 (0)	0	0	0	0	0	0	0
↳ ALU:M14	447 (447)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ MainControl:Mult2	32 (0)	13 (0)	0 (0)	0	0	0	0	0	0	0
↳ Control:C1	26 (26)	13 (13)	0 (0)	0	0	0	0	0	0	0
↳ ALUControl:C2	6 (6)	0 (0)	0 (0)	0	0	0	0	0	0	0
↳ VGA:In2	126 (0)	76 (0)	0 (0)	0	0	0	0	0	0	0
↳ VGAOK:M1	68 (68)	27 (27)	0 (0)	0	0	0	0	0	0	0
↳ Clk:M2	2 (2)	1 (1)	0 (0)	0	0	0	0	0	0	0
↳ Counter:M3	29 (29)	21 (21)	0 (0)	0	0	0	0	0	0	0
↳ decode:M4	33 (33)	27 (27)	0 (0)	0	0	0	0	0	0	0
↳ Wrapper_Receiver:In3	92 (0)	60 (0)	0 (0)	0	0	0	0	0	0	0
↳ UART_Clock:C_76800	41 (41)	27 (27)	0 (0)	0	0	0	0	0	0	0
↳ Resynchronizer:Rsync	2 (2)	2 (2)	0 (0)	0	0	0	0	0	0	0
↳ UART_Receiver:U_Receiver	51 (0)	31 (0)	0 (0)	0	0	0	0	0	0	0
↳ Receiver_Controller:C	38 (38)	13 (13)	0 (0)	0	0	0	0	0	0	0
↳ Receiver_Datapath:D	20 (20)	18 (18)	0 (0)	0	0	0	0	0	0	0

Programa

```

00000011110000000000101000101101 //R[1]=R[30]>>8 Hacer corrimiento para ver el bit rdy
00010000001000100000000000000011 //R[1]=R[2] Jump 3 compara con el R[2] para saber si esta enable
10000000000000000000000000000000 //Jump 0 Regresa a direccion 0 para verificar nuevamente si rdy no ha cambiado
00000011110000000001111100101010 //R[3]=R[30]<<28 Corr verf MSB y saber tipo UP,DOWN, LEFT,RIGHT,
00010000011001000000000000011100 //R[3]=R[4] Jump 28 Compara si es 1001 Color+
00010000011001010000000000010101 //R[3]=R[5] Jump 21 Compara si es 1011 Color-
0001000001100110000000000001110 //R[3]=R[6] Jump 14 Compara si es 1100 Add+
0001000011100000000000000001011 //R[7]=R[0] Jump 11 Entra etapa Add- y pregunta si es 0
00000000111010100011100000100010 //R[7]=R[7]-R[10] Resta 1 al Add
00000000111010011111100000100000 //R[31]=R[7]+R[9] Escribe al Reg31 Add+Color
1000000000000000000000000100010 //Jump 34 Salta al fin de programa
00000001000010011111100000100000 //R[31]=R[8]+R[9] Escribe al Reg31 111+Color
00000000111010000011100000100000 //R[7]=R[7]+R[8] Iguala R7 a R8
1000000000000000000000000100010 //Jump 34 Salta al fin de programa
00010000111010000000000000010010 //R[7]=R[8] Jump 18 Pregunta si es un Add es 7
00000000111010100011100000100000 //R[7]=R[7]+R[8] Suma 1 al Add
00000000111010011111100000100000 //R[31]=R[7]+R[9] Escribe al Reg31 Add+Color
1000000000000000000000000100010 //Jump 34 Salta al fin de programa
00000000000010011111100000100000 //R[31]=R[0]+R[9] Escribe al Reg31 0+Color
00000011011000000011100000100000 //R[7]=R[27]+R[0] Suma 0+0=0
1000000000000000000000000100010 //Jump 34 Salta al fin de programa
00010001001000000000000000011001 //R[9]=R[0] Jump 25 Pregunta si Color = 0
00000001001000100100100000100010 //R[9]=R[9]-R[2] Resta a Color 1
00000000111010011111100000100000 //R[31]=R[7]+R[9] Escribe al Reg31 Su color y add
1000000000000000000000000100010 //Jump 34 Salta al fin de programa
00000000111010111111100000100000 //R[31]=R[7]+R[11] Escribe al Reg31 Su Add y 7

```

[illegible]

Inicialización de Registro

[illegible]

```
module Interfaces(input [3:0]KEY,output [0:0]LEDG, output VGA_CLK,output [7:0] VGA_R,VGA_G,VGA_B, input  
CLOCK_50,UART_RXD, output VGA_BLANK_N,VGA_SYNC_N,VGA_HS,VGA_VS);
```

```
logic clk,start;
```

```
assign clk=CLOCK_50;  
assign start=~KEY[2];
```

```
logic [31:0]R31Out;  
logic [31:0]R30In;
```

```
Multiciclo      In1(.*);  
VGA             In2(.*);  
Wrapper_Receiver In3(.*);
```

```
Endmodule
```

```
module Clk (input CLOCK_50,output reg clk );
```

```
initial clk = 1;  
always@(posedge CLOCK_50)  
    clk <= !clk;  
endmodule
```

```
module VGAOK(output logic [7:0] VGA_R,VGA_G,VGA_B,output logic  
VGA_BLANK_N,VGA_SYNC_N,VGA_HS,VGA_VS,input [9:0] hcont,vcont,input clk,input[191:0]Colum);
```

```
    logic blank;  
    logic hsync;  
    logic vsync;
```

```
initial begin  
    VGA_SYNC_N=1'b1;  
    VGA_G=8'd0;  
    VGA_R=8'd0;  
    VGA_B=8'd0;  
    blank=1'b0;  
    hsync=1'b1;  
    vsync=1'b1;  
end
```

```
assign VGA_BLANK_N = blank;  
assign VGA_HS = hsync;  
assign VGA_VS = vsync;
```

```
always@(posedge clk )  
    begin  
  
        if( hcont <= 640 && vcont <= 480 )  
            begin  
                blank <= 1;  
                hsync <= 1;  
                if( hcont<=80 ){VGA_R,VGA_B,VGA_G}<=Colum[23:0];
```

```

else if (hcont<=160){VGA_R,VGA_B,VGA_G}<=Colum[47:24];
else if (hcont<=240){VGA_R,VGA_B,VGA_G}<=Colum[71:48];
else if (hcont<=320){VGA_R,VGA_B,VGA_G}<=Colum[95:72];
else if (hcont<=400){VGA_R,VGA_B,VGA_G}<=Colum[119:96];
else if (hcont<=480){VGA_R,VGA_B,VGA_G}<=Colum[143:120];
else if (hcont<=560){VGA_R,VGA_B,VGA_G}<=Colum[167:144];
else {VGA_R,VGA_B,VGA_G}<=Colum[191:168];
end
else if ( hcont <= 660) blank <= 0;
else if ( hcont <= 755 ) hsync <= 0;
else hsync <= 1;

if ( vcont <= 494 )vsync <= 1'b1;
else if ( vcont <= 496 ) vsync <= 0;
else vsync <= 1;

```

end

endmodule

module VGA(output VGA_CLK,output [7:0] VGA_R,VGA_G,VGA_B, input CLOCK_50, output
VGA_BLANK_N,VGA_SYNC_N,VGA_HS,VGA_VS,input [31:0]R31Out);

```

logic [9:0]hcont;
logic [9:0]vcont;
logic [191:0]Colum;
logic clk;

```

assign VGA_CLK = clk;

```

VGAOK      M1(. *);
Clk        M2(. *);
Counter    M3(. *);
decode M4(. *);

```

endmodule

module Registro (
output [31:0]data1,data2,
input [31:0]Mux3,Instruction,
input [4:0]Mux2,
input RegWrite,clk,
input [31:0]R30In,
output [31:0]R31Out
);

```
reg [31:0] Registro [0:31];
```

```
initial $readmemb("Registro.dat",Registro);
```

```
        assign R31Out = Registro[31];
    assign data1 = Registro[Instruction[25:21]];
    assign data2 = Registro[Instruction[20:16]];
```

```
always@(posedge clk) begin
    Registro[30]=R30In;
    if ((RegWrite==1) && (Mux2!=0) && (Mux2!=30))
        Registro[Mux2] <= Mux3;
    else if(Mux2==0)
        Registro[Mux2] <= 0;
    else if(Mux2==30)
        Registro[Mux2] <= R30In;
end
```

```
end
```

```
endmodule
```

```
module RB(input [31:0]data2, input clk,start, output logic[31:0]RB);
```

```
always@(posedge clk or posedge start)
    if(start)
        RB <= 0;
    else RB <= data2;
```

```
endmodule
```

```
module RA(input [31:0]data1, input clk,start,output logic [31:0]RA);
```

```
always@(posedge clk or posedge start)
    if(start)
        RA <= 0;
    else    RA <= data1;
```

```
endmodule
```

```
module ProgramCounter(input PCWriteCond,PCWrite,Zero,clk,start, input [31:0]PCMux, output logic [31:0]Counter);
```

```
logic PCEnable;
```

```
assign PCEnable=((Zero&PCWriteCond)|(PCWrite));
```

```
always@(posedge clk or posedge start)
    if(start)
```

```

        Counter <= 0;
    else if(PCEnable)
        Counter <= PCMux;

```

Endmodule

module PCMux(input [31:0]Instruction,result,ALUOut,Counter, input [1:0]PCSource, output logic [31:0]PCMux);

```

    always@(PCSource or result or ALUOut or Instruction or Counter)
    case(PCSource)
    0:    PCMux = result;
    1:    PCMux = ALUOut;
    2: PCMux = {Counter[31:28],(Instruction[26:0]<<2)};
    default PCMux = result;
    endcase

```

endmodule

module Mux5(input [31:0]Instruction,RB, input [1:0]ALUSrcB, output logic [31:0]Mux5);

```

    always@(ALUSrcB or RB or Instruction)

    case(ALUSrcB)
    0:    Mux5 = RB;
    1: Mux5 = 4;
    2:    Mux5 = {{16{Instruction[15]}},Instruction[15:0]};
    3:    Mux5 = {{16{Instruction[15]}},Instruction[15:0]}<<2;
    default Mux5 = RB;
    endcase

```

endmodule

module Mux4(input [31:0]Counter,RA, input ALUSrcA, output [31:0]Mux4);

```

assign Mux4=(ALUSrcA)?RA:Counter;

```

endmodule

module Mux3(input [31:0]ALUOut,MDROut, input MemtoReg, output [31:0]Mux3);

```

assign Mux3=(MemtoReg)?MDROut:ALUOut;

```

endmodule

module Mux2(input [31:0]Instruction, input RegDst, output [4:0]Mux2);

```

assign Mux2=(RegDst)?Instruction[15:11]:Instruction[20:16];

```

endmodule

```
module Mux1(input [31:0]Counter,ALUOut, input lorD, output [31:0]Mux1);
```

```
assign Mux1=(lorD)?ALUOut:Counter;
```

```
endmodule
```

```
module Multiciclo(input clk,start,output logic [31:0]R31Out,input [31:0]R30In);
```

```
logic [31:0]Instruction;
```

```
logic ALUSrcA,PCWriteCond,MemRead,lorD,RegDst,RegWrite,MemtoReg,IRWrite,PCWrite,MemWrite;
```

```
logic [1:0]ALUSrcB;
```

```
logic [1:0]PCSource;
```

```
logic [4:0]Select;
```

```
DataPath          Mult1(.*);
```

```
MainControl      Mult2(.*);
```

```
Endmodule
```

```
module Memory(input MemRead,MemWrite,clk, output logic [31:0]MemData,input [31:0]RB,Mux1);
```

```
reg [31:0] mem [0:36];
```

```
logic [4:0]Add_Read;
```

```
initial $readmemb("Memoria.dat",mem);
```

```
always@(posedge clk)
    if(MemWrite) mem[Mux1[7:2]]<= RB;
```

```
always@(posedge clk)
    if(MemRead)MemData=mem[Mux1[7:2]];
```

```
endmodule
```

```
module MDR(input [31:0]MemData, input clk,start, output logic [31:0]MDROut);
```

```
always@(posedge clk or posedge start)
```

```
    if(start)
```

```
        MDROut <= 0;
```

```
    else MDROut <= MemData;
```

```
endmodule
```

```
module MainControl(output logic
```

```
ALUSrcA,PCWriteCond,MemRead,lorD,RegDst,RegWrite,MemtoReg,IRWrite,PCWrite,MemWrite,
```

```
output logic [1:0]ALUSrcB,PCSource,
```

```
input [31:0]Instruction,
```

```
input clk,start,
```

```
output logic [4:0]Select);
```

```
logic [1:0]ALUOp;
```

```
Control      C1(. *);  
ALUControl   C2(. *);
```

```
Endmodule
```

```
module IRegister(input [31:0]MemData, input clk,start,IRWrite, output logic [31:0]Instruction);
```

```
always@(posedge clk or posedge start)  
    if(start)  
        Instruction <= 0;  
    else if(IRWrite)  
        Instruction <= MemData;
```

```
Endmodule
```

```
module decode(input [31:0]R31Out, output logic [191:0]Colum, input clk);
```

```
parameter [23:0]                                Negro=24'hFF0000,  
                                                    Amarillo=24'h0000FF,  
                                                    Rojo=24'hFFFFFF,  
                                                    Verde=24'hFFFF00,  
                                                    Azul=24'h000000,  
                                                    Blanco=24'hFF00FF,  
                                                    Celeste=24'h00FF00,  
                                                    Rosa=24'h00FFFF;
```

```
logic [23:0]Color;
```

```
initial begin
```

```
    Colum[23:0]=Negro;  
    Colum[47:24]=Amarillo;  
    Colum[71:48]=Rojo;  
    Colum[95:72]=Azul;  
    Colum[119:96]=Verde;  
    Colum[143:120]=Rosa;  
    Colum[167:144]=Celeste;  
    Colum[191:168]=Blanco;  
end
```

```
always@(posedge clk)  
case(R31Out[2:0])
```

```
0:      Color <=Negro;  
1:      Color <=Amarillo;  
2:      Color <=Rojo;  
3:      Color <=Azul;  
4:      Color <=Verde;  
5:      Color <=Rosa;  
6:      Color <=Celeste;  
7:      Color <=Blanco;
```

```
endcase
```

```

always@(posedge clk)
    case(R31Out[5:3])
        0:      Colum[23:0]<=Color;
        1:      Colum[47:24]<=Color;
        2:      Colum[71:48]<=Color;
        3:      Colum[95:72]<=Color;
        4:      Colum[119:96]<=Color;
        5:      Colum[143:120]<=Color;
        6:      Colum[167:144]<=Color;
        7:      Colum[191:168]<=Color;
    endcase

```

```
endmodule
```

module DataPath(

```

input PCWriteCond,PCWrite,clk,IorD,MemRead,MemWrite,IRWrite,RegDst,MemtoReg,RegWrite,ALUSrcA,start,
input [1:0]ALUSrcB,
input [31:0]R30In,
input [1:0]PCSource,
input [4:0]Select,
output logic [31:0]R31Out,
output logic [31:0]Instruction
);

```

```

logic [31:0]Counter;
logic [31:0]MemData;

```

```

logic [31:0]RB,RA;
logic [31:0]result;
logic [31:0]ALUOut;
logic [31:0]MDROut;
logic [31:0]Mux4;
logic [31:0]Mux5;
logic Zero;
logic [31:0]PCMux;
logic [31:0]Mux1;
logic [4:0]Mux2;
logic [31:0]Mux3;
logic [31:0]data1,data2;

```

ProgramCounter	M0(.*);
Mux1	M1(.*);
Memory	M2(.*);
MDR	M3(.*);
IRegister	M4(.*);
Mux2	M5(.*);
Mux3	M6(.*);
Registro	M7(.*);
RA	M8(.*);
RB	M9(.*);
Mux4	M10(.*);
Mux5	M11(.*) ;

PCMux	M12(.*);
AluOut	M13(.*);
ALU	M14(.*);

Endmodule

module Counter (input clk,output reg [9:0]hcont,vcont);

logic rst;

initial begin hcont = 0; vcont = 0; rst = 1'b1; end

always@(posedge clk or posedge rst)

if(rst)begin

hcont <= 0;

rst <= 0;

end

else if(hcont >= 800)begin

vcont <= vcont + 1;

rst <= 1;

end

else if (vcont >= 525) vcont <= 0;

else hcont <= hcont + 1;

endmodule

module Control(output logic

ALUSrcA,PCWriteCond,MemRead,IorD,RegDst,RegWrite,MemtoReg,IRWrite,PCWrite,MemWrite, output logic

[1:0]ALUSrcB,ALUOp,PCSource, input [31:0]Instruction, input clk,start);

enum bit [3:0] {Fetch,Fetc,Fe,DI,Rty,Decode,MemAdd,Exe,Branch,Jump,MemAcc1,MemAcc2,Rtype,MemoryR}NameState;

parameter [5:0]LW = 6'b100011,SW = 6'b101011,R_Type = 6'b000000,Beq = 6'b000100, J = 6'b100000, ADDI = 6'b110000;

reg [3:0]state,next;

always@(posedge clk or posedge start)

if(start)state<=Fetch;

else state <=next;

always@(state) begin

next = 4'bxxxx;

ALUSrcA = 0;

IorD = 0;

ALUSrcB = 2'b00;

ALUOp = 2'b00;

PCSource = 2'b00;

PCWrite = 1'b0;

IRWrite = 1'b0;

MemRead = 1'b0;

RegDst = 1'b0;

RegWrite = 1'b0;

MemtoReg = 1'b0;

MemWrite = 1'b0;

```

PCWriteCond = 1'b0;

unique case(state)
  Fetch:begin
    MemRead      = 1;
    ALUSrcA      = 0;
    lorD         = 0;
    IRWrite      = 1;
    ALUSrcB      = 2'b01;
    ALUOp        = 2'b00;
    PCWrite      = 1;
    PCSource     = 2'b00;
    next = Decode;
  end

  Decode: begin
    ALUSrcA = 0;
    ALUSrcB = 2'b11;
    ALUOp   = 2'b00;
    if((Instruction[31:26] == LW) || (Instruction[31:26] == SW))
      next = MemAdd;
    else if(Instruction[31:26] == R_Type)
      next = Exe;
    else if(Instruction[31:26] == Beq )
      next = Branch;
    else if(Instruction[31:26] == J)
      next = Jump;
    else if(Instruction[31:26] == ADDI)
      next = DI;
  end

  MemAdd: begin
    ALUSrcA = 1;
    ALUSrcB = 2'b11;
    ALUOp   = 2'b00;
    if(Instruction[31:26] == LW)
      next = MemAcc1;
    else next = MemAcc2;
  end

  Exe:      begin
    ALUSrcA = 1;
    ALUSrcB =
      ((Instruction[5:0]==6'b101101) || (Instruction[5:0]==6'b101010)) ? 2'b10 : 2'b00;
    ALUOp   = 2'b10;
    next = Rtype;
  end

  Branch: begin
    ALUSrcA = 1;
    ALUSrcB = 2'b00;
    ALUOp   = 2'b01;
    PCWriteCond = 1'b1;
    PCSource   = 2'b10;
    next = Fetc;
  end

  Jump: begin

```

```

        PCWrite = 1'b1;
        PCSource = 2'b10;
        MemRead = 1;
        next = Fetc;
    end
MemAcc1: begin
        MemRead = 1'b1;
        lorD = 1;
        next = MemoryR;
    end
MemAcc2: begin
        MemWrite = 1'b1;
        lorD = 1;
        next = Fetc;
    end
Rtype: begin
        RegDst = 1;
        RegWrite = 1'b1;
        MemtoReg = 0;
        next = Fetc;
    end
MemoryR: begin
        RegDst = 1;
        RegWrite = 1'b1;
        MemtoReg = 0;
        next = Fetc;
    end
Fetc:begin
        MemRead = 1;
        ALUSrcA = 0;
        lorD = 0;
        IRWrite = 0;
        ALUSrcB = 2'b01;
        ALUOp = 2'b00;
        PCWrite = 0;
        PCSource = 2'b00;
        next = Fetch;
    end

DI:    begin
        ALUSrcA = 1;
        ALUSrcB = 2'b10;
        ALUOp = 2'b10;
        next = Rty;
    end
Rty:    begin
        RegDst = 0;
        RegWrite = 1'b1;
        MemtoReg = 0;
        next = Fetch;
    end
endcase
endcase

```

end

endmodule

module AluOut(input [31:0]result, input clk,start, output logic [31:0]ALUOut);

always@(posedge clk or posedge start)

if(start)

ALUOut <= 0;

else ALUOut <= result;

endmodule

module ALUControl (

input [31:0]Instruction,

input [1:0]ALUOp,

output logic[4:0]Select

);

always@(ALUOp or Instruction[5:0])

begin

if (ALUOp == 2'b10)

begin

unique case(Instruction[5:0])

6'b100000: Select = 0;//Suma ALU

6'b100010: Select = 1;//Resta ALU

6'b100100: Select = 2;//AND

6'b100101: Select = 3;//XOR

6'b101101: Select = 4;//Corrimiento Dere

6'b101010: Select = 5;//Corrimiento Izq

endcase

end

else if (ALUOp == 2'b01)

Select = 1;

else

Select = 0;

end

endmodule

module ALU (input [4:0]Select,input [31:0]Mux4,Mux5, output logic [31:0]result,output Zero);

assign Zero=(result==0)?1'b1:1'b0;

always@(Select,Mux4,Mux5)

begin

case (Select)

0: result = Mux4+Mux5;

1: result = Mux4-Mux5;

2: result = Mux4&Mux5;

3: result = Mux4^Mux5;

```

        4:          result = Mux4>>Mux5[10:6];
        5:          result = Mux4<<Mux5[10:6];
        default: result = Mux4+Mux5;
    endcase
end

endmodule:ALU

module Receiver_Controller(
input Serial_in,Sample_Clk,Rst,output Shift,Load,R_Ready,
output logic[4:0]Bit_Counter);

    logic          Clr_Sample_Counter,
                  Clr_Bit_Counter,
                  Inc_Sample_Counter,
                  Inc_Bit_Counter;

    logic [3:0]    Sample_Counter;

    enum logic [2:0] {IDLE = 3'b001,
                    STRT = 3'b010,
                    RCVG = 3'b011,
                    STOP = 3'b100} state, next_state;

    always_ff@(posedge Sample_Clk or negedge Rst) begin: State_Transitions
        if(!Rst)begin
            Bit_Counter <= 0;
            Sample_Counter <= 0;
            state <= IDLE;
        end
        else begin
            state <= next_state;

            if(Clr_Sample_Counter)
                Sample_Counter <= 0;
            else if(Inc_Sample_Counter)
                Sample_Counter <= Sample_Counter + 1'b1;

            if(Clr_Bit_Counter)
                Bit_Counter <= 0;
            else if(Inc_Bit_Counter)
                Bit_Counter <= Bit_Counter + 1'b1;
        end
    end : State_Transitions

    always_comb begin : Output_and_Next_state
        Shift = 1'b0;
        Load = 1'b0;
        R_Ready = 1'b0;
        next_state = state;
        Clr_Sample_Counter = 1'b0;
        Clr_Bit_Counter = 1'b0;
        Inc_Sample_Counter = 1'b0;
        Inc_Bit_Counter = 1'b0;
    end

```

```

case (state)
    IDLE: if (!Serial_in)                                //Start bit detected
        next_state = STRT;

    STRT: if (Serial_in) begin                            //First bit change before 4 samples
        next_state = IDLE;
        Clr_Sample_Counter = 1;
    end else

        if(Sample_Counter == 3)begin //Start bit confirmed
            next_state = RCVG;
            Clr_Sample_Counter = 1;
        end else
            Inc_Sample_Counter = 1;

    RCVG: if(Sample_Counter < 7)
        Inc_Sample_Counter = 1;
    else begin
        Clr_Sample_Counter = 1;
        if(Bit_Counter != 9) begin
            Shift = 1;
            Inc_Bit_Counter = 1;
        end
        else begin
            next_state = STOP;                                //Data recived completed
        end
    end

    STOP: if(Sample_Counter < 7)
        Inc_Sample_Counter = 1;
    else begin
        Clr_Sample_Counter = 1;
        if(Bit_Counter != 10) begin
            Shift = 1;
            Inc_Bit_Counter = 1;
        end
        else begin
            next_state = IDLE;
            Clr_Bit_Counter = 1;
            Load = 1;
            R_Ready = 1;
        end
    end

    default: next_state = IDLE;
endcase
end : Output_and_Next_state

endmodule

```

module Receiver_Datapath

```
(  
    input                Serial_in,  
                        Shift,  
                        Load,  
  
                        Sample_Clk,  
                        Rst,  
  
    input  [4:0]         Bit_Counter,  
    output                Error,  
    output[7:0]         Data_Bus  
);  
  
logic                Parity_in,  
                    Parity_out;  
  
logic[7:0]          RCV_DataReg;  
logic[9:0]          RCV_ShiftReg;  
  
assign              Data_Bus = RCV_DataReg;  
  
always_ff@(posedge Sample_Clk, negedge Rst) begin  
    if(!Rst)begin  
        RCV_DataReg <= 0;  
        RCV_ShiftReg <= 0;  
    end  
    else begin  
        if(Shift)  
            RCV_ShiftReg <= {Serial_in, RCV_ShiftReg[9:1]};  
        if(Load)  
            RCV_DataReg <= RCV_ShiftReg[7:0];  
    end  
end  
  
assign Parity_in = ^RCV_ShiftReg[7:0];  
assign Parity_out = RCV_ShiftReg[8];  
assign Error = ((Bit_Counter == 0) && (Parity_in != Parity_out)) ? 1'b1 : 1'b0;  
  
endmodule
```

module Resynchronizer

```
(input        Button_in, Clk,  
output Button_out);  
  
logic Sync_1;  
  
always @(posedge Clk) begin  
    Sync_1 <= Button_in;  
    Button_out <= Sync_1;  
end  
  
endmodule
```

```

module UART_Clock #(COUNT = 5028)(
input          Sys_Clk, Rst, en,
output reg     Clk
);

localparam LIMIT_C = ((COUNT/2)-1);

reg [25:0]      Count_C;

wire RCO_C;

assign RCO_C = (Count_C == LIMIT_C) ? 1'b1 : 1'b0;

always @(posedge Sys_Clk, negedge Rst) begin : Counter_Clock
    if(!Rst)
        Count_C <= 0;
    else
        if(en)
            begin
                if(Count_C == LIMIT_C)
                    Count_C <= 0;
                else
                    Count_C <= Count_C + 1'b1;
            end
    end : Counter_Clock

always@(posedge Sys_Clk)
begin
    if(RCO_C)
        Clk <= ~Clk;
    else
        Clk <= Clk;
end

endmodule

```

```

module UART_Receiver(
input          Serial_in, Sample_Clk, Rst,

output         Error,

                                R_Ready,

output[7:0]     Data_Bus
);

logic          Shift,
                                Load;

logic[4:0]      Bit_Counter;

Receiver_Datapath D (. *);
Receiver_Controller C (. *);

Endmodule

```

module Wrapper_Receiver

```
(
    input                CLOCK_50,
                        UART_RXD,

    input [3:0]          KEY,
    output               [0:0]LEDG,
    output logic [31:0] R30In
);

    logic                Sample_Clk,
                        rdy,
                        Serial_in;

    logic  [7:0]         Data_Bus;

    assign R30In={23'd0,rdy,Data_Bus};

    Resynchronizer Rsync (.Button_in(UART_RXD), .Clk(Sample_Clk), .Button_out(Serial_in));

    UART_Clock #(.COUNT(651)) C_76800 (.Sys_Clk(CLOCK_50), .Rst(KEY[0]), .en(1'b1), .Clk(Sample_Clk));
    UART_Receiver U_Receiver(.Rst(KEY[0]), .Data_Bus(Data_Bus), .Error(LEDG[0]), .R_Ready(rdy), .*);

Endmodule
```

Slow 1200mV 0C Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	80.28 MHz	80.28 MHz	CLOCK_50	
2	219.35 MHz	219.35 MHz	VGA:In2 Clk:M2 clk	
3	298.15 MHz	298.15 MHz	Wrapper_Receiver:In3 UART_Clock:C_76800 Clk	

Flow Summary

Flow Status	Successful - Sun Nov 02 23:34:33 2014
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	Interfaces
Top-level Entity Name	Interfaces
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	2,694 / 114,480 (2 %)
Total combinational functions	2,213 / 114,480 (2 %)
Dedicated logic registers	1,287 / 114,480 (1 %)
Total registers	1287
Total pins	36 / 529 (7 %)
Total virtual pins	0
Total memory bits	1,184 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)