Universidad de Guadalajara
Departamento de Electrónica

Proyecto final

Procesador de 8 bits

Sistemas reconfigurables

Eduardo Vázquez Díaz
lalohao@gmail.com

30 de noviembre de 2014

Índice

Resumen

Se construyó un procesador de 8 bits con funcionalidades basicas, utilizando verilog y software libre para diseñar y simularlo. Posteriormente se implementó en una tarjeta fpga spartan 3.

1. Introducción

El lenguaje de descripcion de hardware es una potente herramienta que facilita la creacion de prototipos electronicos. Permite expresar en un lenguaje sintactico, entendible para los humanos funciones especificas que pueden ser asignadas a un *FPGA* y cambiarlas las veces que se desee. Un ejemplo de estos lenguajes es *Verilog*, que es ademas el elegido para llevar a cabo el diseño que se mostrará mas adelante.

Del software libre (SL) se obtiene otra gran ventaja. La mayoria del software es documentado y respaldado por una comunidad. GNU/Linux es el mejor ejemplo donde grandes comunidades se desenvuelven entorno al desarrollo y uso de SL y a la filosofia que estas transmiten.

Siendo los procesadores la unidad central de practicamente cualquier dispositivo inteligente, es imprescindible conocer su arquitectura interna y ser capaz de construir modelos de ellos.

2. Desarrollo

El procesador se puede descomponer en unidades mas pequeñas que realizan especificas tareas.

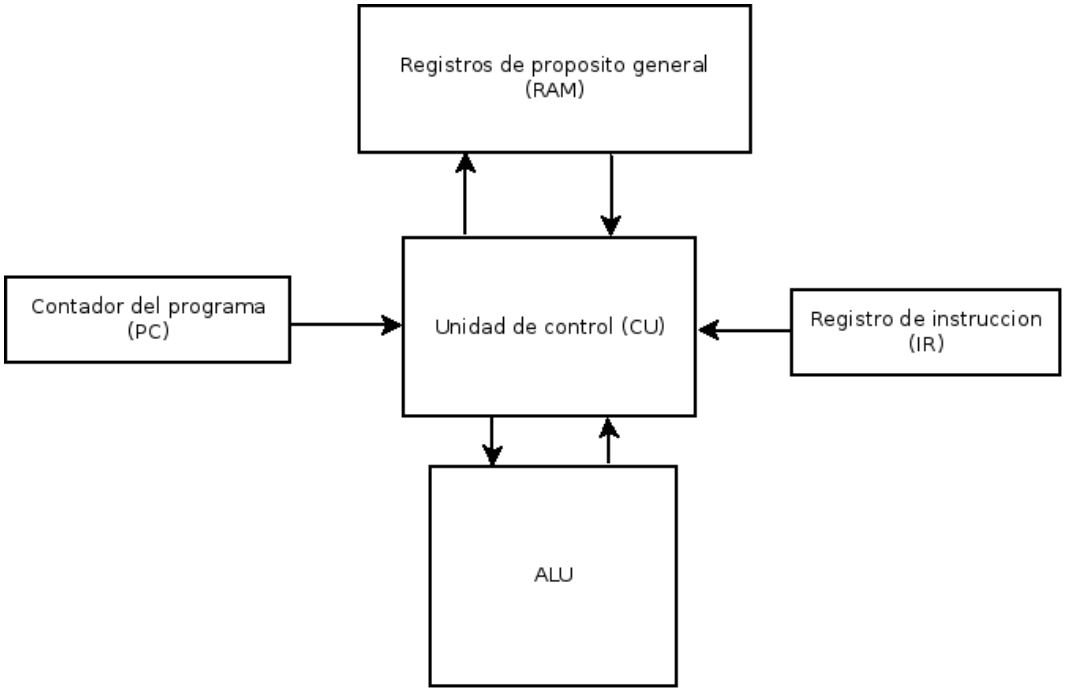

Figura 1: Diagrama de bloques del procesador

2.1. Unidad de Control

El corazon del CPU, la **unidad de control** se diseño como una maquina de estados para facilitar su implementacion en el FPGA.

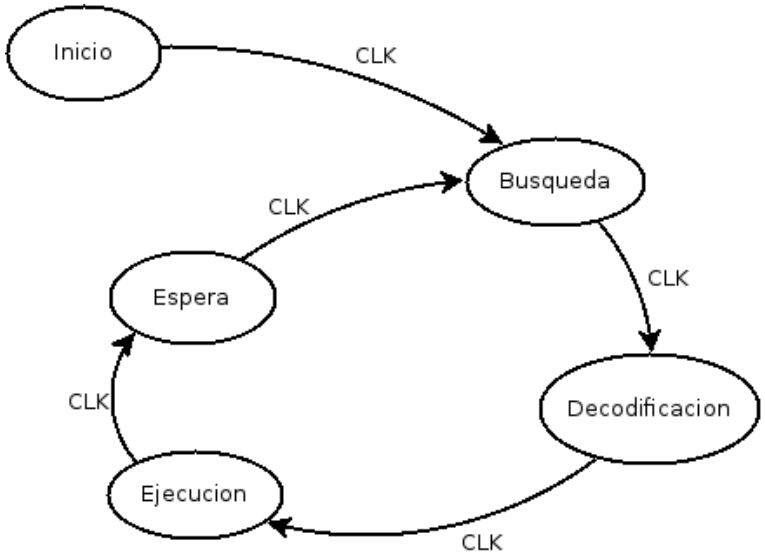

Figura 2: Diagrama de estados de la unidad de control (CU)

Estados
Inicio: Las variables y conexiones internas se (re)inicializan a 0.
Busqueda: Se pasa la instruccion completa al IR.
Decodificacion: Se prepará para la ejecucion de la instruccion.
Ejecucion: Se ejecuta la instruccion.
Espera: Se espera a que se termine de ejecutar la instruccion.

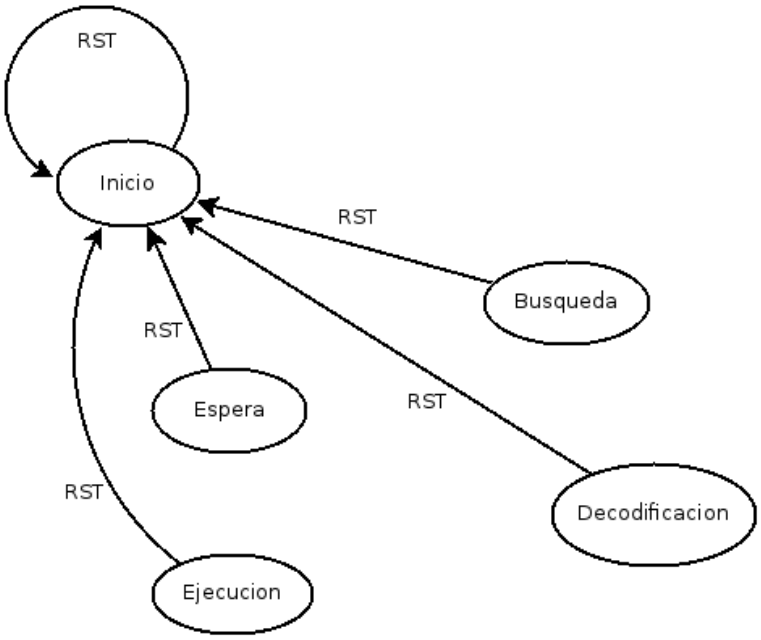

Figura 3: Al activar RST todos los estados llevan al inicial.

2.2. Registro de instruccion (IR)

La idea de enviarle instrucciones a la tarjeta spartan **xc3s200** utilizando sus DIP SWITCH conllevo al diseñó de un IR de 8bits.

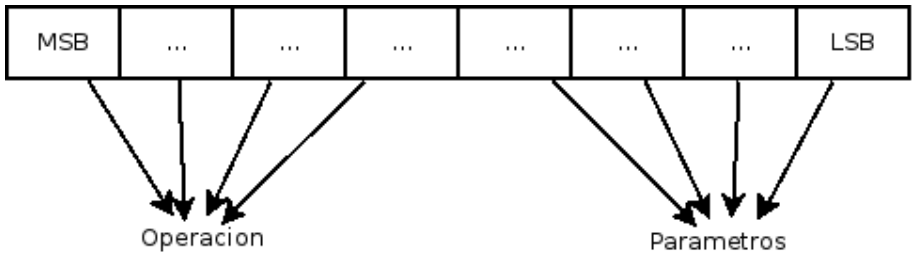

Figura 4: Los 4 bits mas significativos representan la operacion a realizar, con un total de 16 operaciones posibles. Los 4 bits menos significativos son el parametro de la operacion.

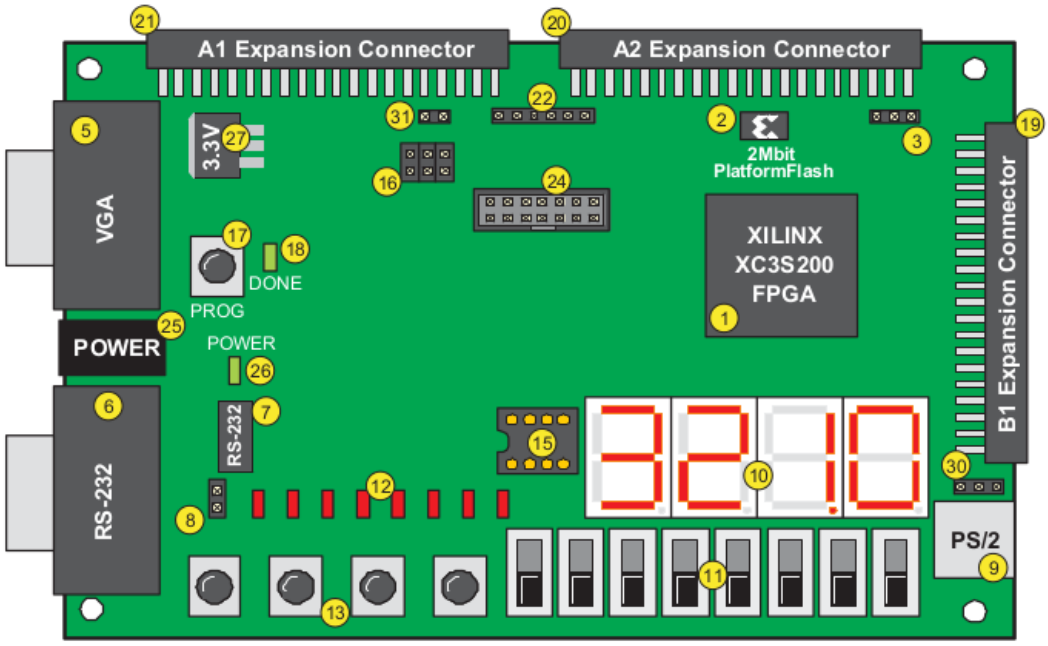

Figura 5: (11) Cada DIP-Switch representa un bit del IR. (13) CLK manual y reset.

2.3. Registros (Memoria RAM)

La memoria RAM cuenta con 16 registros de proposito general de 4 bits cada uno, lo que permite utilizar la misma longitud para el bus de datos y el bus de direccion.

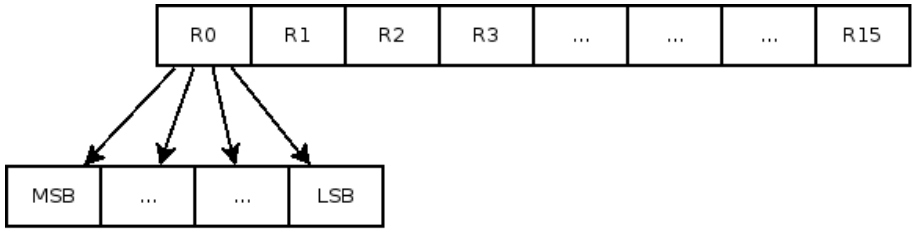

Figura 6: Arquitectura interna de la memoria RAM utilizada para guardar los registros de proposito general.

2.4. Instrucciones

Instruccion	Parametros	Descripcion
ldaxx = 0000	xxxx	Carga el dato del registro xxxx al acumulador. **a**
ldxxa = 0001	xxxx	Carga el dato **a** en el registro xxxx.
andaxx = 0010	xxxx	Aplica la operacion AND logica a nivel de bits entre **a** y el registro xxxx, el resultado se guarda en **a**.
addaxx = 0011	xxxx	Suma el contenido del registro xxxx a **a**.
subaxx = 0100	xxxx	Resta el contenido del registro xxxx a **a**.
jzxx = 0101	xxxx	Salta a la direccion xxxx del pc si **a** = 0.
jmpxx = 0110	-	Salta a la direccion xxxx del pc.
nop = 0111	-	No hace nada.
movaxx = 1000	xxxx	Pon xxxx en el acumulador.
nandaxx = 1001	xxxx	Similar a andaxx pero la operacion NAND.
oraxx = 1010	xxxx	Similar a nandaxx pero la operacion OR.
noraxx = 1011	xxxx	Similar a oraxx pero la operacion NOR.
xoraxx = 1100	xxxx	Similar a noraxx pero la operacion XOR.
xnoraxx = 1101	xxxx	Similar a xoraxx pero la operacion XNOR.
nota = 1110	-	Aplica la operacion NOT a **a**.

3. Resultados

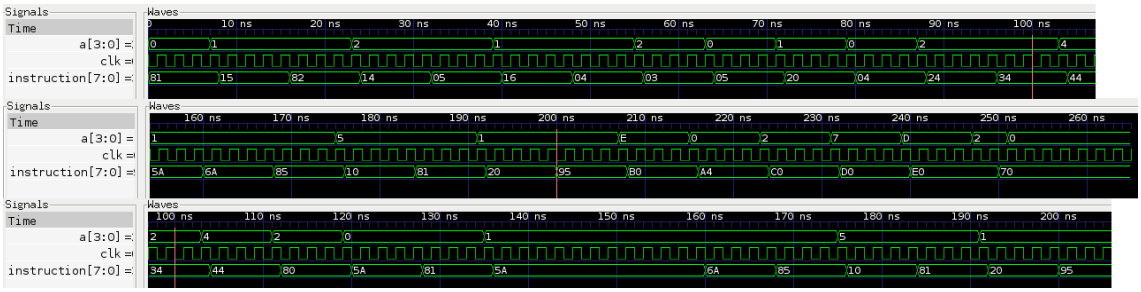

Figura 7: Simulacion de instrucciones.

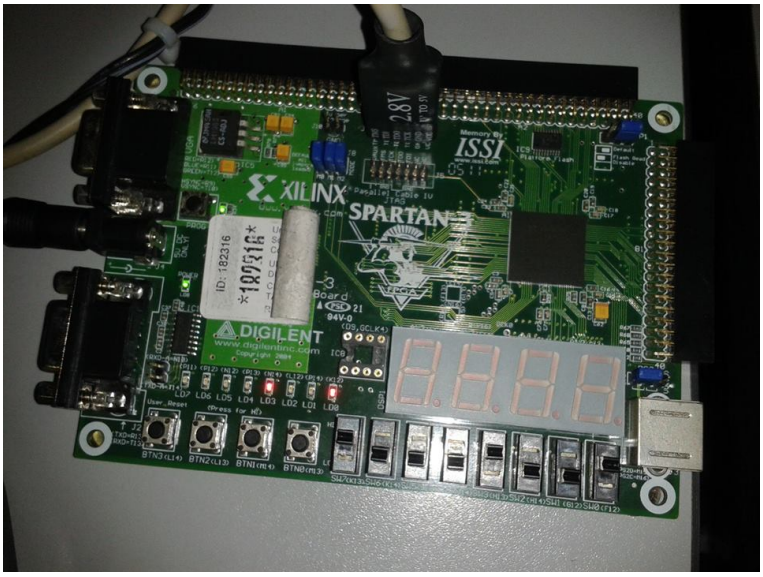

Figura 8: Instruccion 0x89.

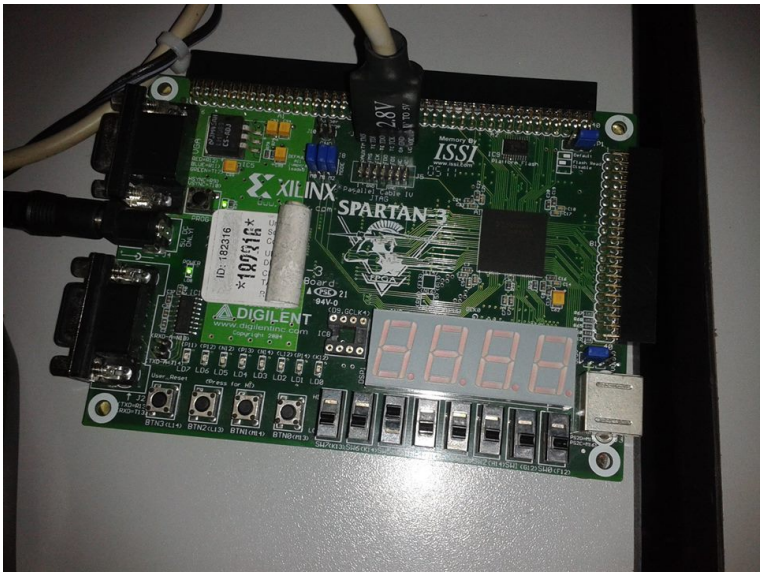

Figura 9: Instruccion 0x20.

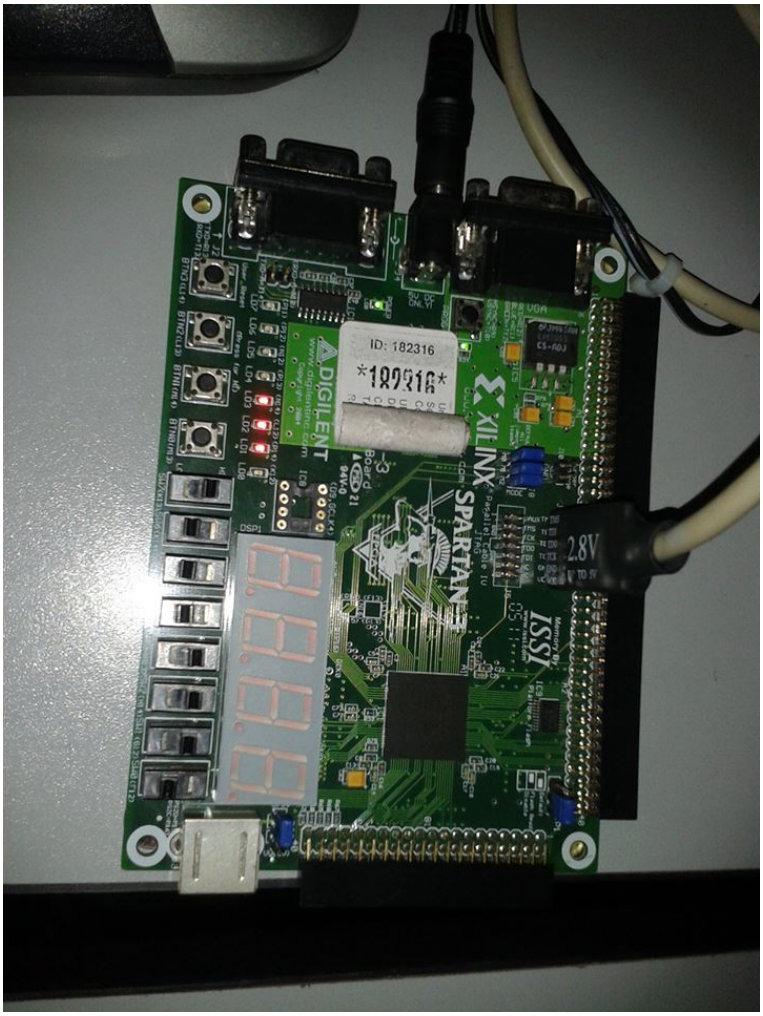

Figura 10: Instruccion 0x8E

4. Apéndice

Documento hecho en LaTeX.
Para ver los archivos sintetizables ver seccion ***Codigo sintetizable*** *más abajo.*
Los archivos descritos a continuacion son para simularse y utilizarse en conjunto:= *Linux* (o sistemas basados en Unix) + *Verilator* + *GTKWave*.

4.1. Makefile

Makefile

```
MAIN = cpu
TESTBENCH = ${MAIN}_tb.cpp

all: compile run simulate

compile:
	/usr/bin/verilator -Wall --cc --trace ${MAIN}.v --exe ${TESTBENCH}
	sed 's/VERILATOR_ROOT = \/usr\/bin/VERILATOR_ROOT = \/usr\/share\/
	rm obj_dir/V${MAIN}.mk
	mv V${MAIN}.mk obj_dir/
	make -j -C obj_dir/ -f V${MAIN}.mk V${MAIN}

run:
	obj_dir/V${MAIN}

simulate:
	gtkwave ${MAIN}.vcd
```

4.2. cpu.v

cpu.v

```
/* verilator lint_off WIDTH */
module cpu
  #(parameter data_width = 4,
    parameter addr_width = 4,
    parameter op_width   = 4,
    parameter op_param   = 4,
    parameter pc_width   = 8)
   (input wire clk, rst,
    input wire [(op_width+op_param)-1:0] instruction,
    output reg [data_width-1:0]          a
    );

   reg [data_width-1:0]                  datain;
   reg [data_width-1:0]                  dataout;
   reg [addr_width-1:0]                  addr;
```

```
reg [op_width-1:0]                      currentState, nextState;
reg [pc_width-1:0]                      pc;
reg [(op_width+op_param)-1:0]           ir;
reg                                     dir;

parameter //Ciclos del CPU
  START = 0, FETCH = 1, DECODE = 2, EXECUTE = 3, WAIT = 4;

parameter //Operaciones
  ldaxx = 0, ldxxa = 1, andaxx = 2, addaxx = 3,
  subaxx = 4, jzxx = 5, jmpxx = 6, nop = 7,
  movaxx = 8, nandaxx = 9, oraxx = 10, noraxx = 11,
  xoraxx = 12, xnoraxx = 13, nota = 14;

ram cpu_mem(.clk(clk), .rst(rst), .rw(dir), .addr(addr),
            .datain(datain), .dataout(dataout));

always @(posedge clk, posedge rst)
  if(rst) currentState <= START;
  else    currentState <= nextState;

always @(currentState)
  case (currentState)
    START:   nextState = FETCH;
    FETCH:   nextState = DECODE;
    DECODE:
      case(ir[(op_width+op_param)-1:op_width])
        nop:
          nextState = WAIT;
        default:
          nextState = EXECUTE;
      endcase // case (ir[(op_width+op_param)-1:op_width])
    EXECUTE: nextState = WAIT;
    WAIT: nextState = FETCH;
  endcase // case (currentState)

always @(posedge clk)
  case (currentState)
    START:
      begin
```

```
      dir <= 0;
      pc <= 0;
      ir <= 0;
      a <= 0;
      datain <= 0;
      addr <= 0;
    end // case: START
FETCH:
    begin
      dir <= 0;
      pc <= pc + 1;
      ir <= instruction;
      addr <= instruction[op_width-1:0];
    end // case: FETCH
EXECUTE:
    begin
      dir <= 0;
      addr <= ir[op_width-1:0];
      case(ir[(op_width+op_param)-1:op_width])
        ldaxx:
          begin
            a <= dataout;
          end
        ldxxa:
          begin
            dir <= 1;
            datain <= a;
          end
        andaxx:
          begin
            a <= (a & dataout);
          end
        addaxx:
          begin
            a <= (a + dataout);
          end
        subaxx:
          begin
            a <= (a - dataout);
          end
```

```
            jzxx:
              begin
                 if(a === 4'b0) // Importante
                   pc <= addr;
              end
            jmpxx:
              begin
                 pc <= addr;
              end
            movaxx:
              begin
                 a <= ir[op_width-1:0];
              end
            nandaxx:
              begin
                 a <= ~(a & dataout);
              end
            oraxx:
              begin
                 a <= (a | dataout);
              end
            noraxx:
              begin
                 a <= ~(a | dataout);
              end
            xoraxx:
              begin
                 a <= (a ^ dataout);
              end
            xnoraxx:
              begin
                 a <= ~(a ^ dataout);
              end
            nota:
              begin
                 a <= ~a;
              end
          endcase // case (ir[(op_width+op_param)-1:op_width])
      end // case: EXECUTE
    WAIT:
```

```
            begin
            end // case: WAIT
        endcase // case (nextState)

endmodule // cpu
```

4.3. cpu_tb.cpp

cpu_tb.cpp

```
#include "Vcpu.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

#define EXEC 8

int clk = 0;
VerilatedVcdC *gtkw = new VerilatedVcdC;
Vcpu *cpu = new Vcpu;

void exec(int instruction)
{
  static int i;
  static int t = EXEC;
  cpu->instruction = instruction;
  // t += 4*((instruction >= 0x00) && (instruction <= 0x0A));
  for (i = 0; i < t; i++)
    {
      gtkw->dump(clk);
      cpu->clk = !cpu->clk;
      cpu->eval();
      clk++;
    }
}

void end()
{
  static int i;
  cpu->rst = 1;
  for (i = 0; i < 2; i++)
```

```
    {
      gtkw->dump(clk);
      cpu->clk = !cpu->clk;
      cpu->eval();
      clk++;
    }
  cpu->rst = 0;
  cpu->instruction = 0x70;
  for (i = 0; i < EXEC*2; i++)
    {
      gtkw->dump(clk);
      cpu->clk = !cpu->clk;
      cpu->eval();
      clk++;
    }
}

int main(int argc, char **argv, char **env)
{
  Verilated::commandArgs(argc, argv);
  Verilated::traceEverOn(true);
  cpu->trace (gtkw, 99);
  gtkw->open ("cpu.vcd");

  exec(0x81); // mov a, 1
  exec(0x15); // ld r5, a
  exec(0x82); // mov a, 2
  exec(0x14); // ld r5, a
  exec(0x05); // ld a, r5
  exec(0x16); // ld r6, a
  exec(0x04); // ld a, r4
  exec(0x03); // ld a, r3
  exec(0x05); // ld a, r5
  exec(0x20); // and a, r0
  exec(0x04); // ld a, r4
  exec(0x24); // and a, r0
  exec(0x34); // add a, r4
  exec(0x44); // sub a, r4
  exec(0x80); // mov a, 0
  exec(0x5A); // jz A
```

```
  exec(0x81); // mov a, 1
  exec(0x5A); // jz A
  exec(0x5A); // jz A
  exec(0x5A); // jz A
  exec(0x6A); // jmp A
  exec(0x85); // mov a, 5
  exec(0x10); // ld r1, a
  exec(0x81); // mov a, r1
  exec(0x20); // and a, r0
  exec(0x95); // nand a, r5
  exec(0xB0); // nor a, r0
  exec(0xA4); // or a, r4
  exec(0xC0); // xor a, r0
  exec(0xD0); // xnor a, r0
  exec(0xE0); // not a
  end();      //

  if(Verilated::gotFinish())
    exit(0);
  gtkw->close();
  exit(0);

  return 0;
}
```

4.4. ram.v

ram.v

```
module ram
  #(parameter data_width = 4,
    parameter addr_width = 4)
   (input wire clk, rst, rw,
    input wire [addr_width-1:0] addr,
    input wire [data_width-1:0] datain,
    output reg [data_width-1:0] dataout
    );
   reg [data_width-1:0]          data [0:(addr_width*addr_width)-1];
   reg [data_width-1:0]          i;
```

```
    parameter
      WRITE = 1, READ = 0;

    always @(posedge clk, posedge rst)
      if (rst)
        begin
           for( i = 0 ; i < (addr_width*addr_width)-1 ; i++ )
             data[i] <= 0;
           data[(addr_width*addr_width)-1] <= 0;
           dataout <= 0;
        end //end if (rst)
      else
        case(rw)
          WRITE:
            begin
               data[addr] <= datain;
               dataout <= datain;
            end
          READ:
            begin
               dataout <= data[addr];
            end
        endcase // case (rw)

endmodule // ram
```

4.5. ram_tb.cpp

ram_tb.cpp

```
#include "Vram.h"
#include "verilated.h"
#include "verilated_vcd_c.h"

#define WRITE 1
#define READ 0

int main(int argc, char **argv, char **env)
{
  Verilated::commandArgs(argc, argv);
```

```
Verilated::traceEverOn(true);
VerilatedVcdC *gtkw = new VerilatedVcdC;

int clk, i, j;
Vram *ram = new Vram;
ram->trace(gtkw, 99);
gtkw->open("ram.vcd");

clk = 0;
ram->rw = WRITE;
i = j = 0;
while(!ram->data[15])
  {
    gtkw->dump(clk);
    if(j%2)
      i++;
    ram->datain = i;
    ram->addr = i;
    ram->clk = !ram->clk;
    ram->eval();
    clk++;
    j++;
  }
for(j = 1; j <=2; j++)
  {
    gtkw->dump(clk);
    ram->clk = !ram->clk;
    ram->eval();
    clk++;
  }
i = j = 0;
ram->rw = READ;
ram->addr = i;
ram->clk = !ram->clk;
ram->eval();
gtkw->dump(clk);
clk++;
ram->clk = !ram->clk;
while(ram->dataout != 15)
  {
```

```
        if(j%2)
          i++;
        ram->addr = i;
        ram->eval();
        gtkw->dump(clk);
        ram->clk = !ram->clk;
        clk++;
        j++;
      }

    for(i = 0; i < 6; i++)
      {
        ram->rst = i == 2;
        ram->eval();
        gtkw->dump(clk);
        ram->clk = !ram->clk;
        clk++;
      }

    if(Verilated::gotFinish())
      exit(0);
    gtkw->close();
    exit(0);

    return 0;
}
```

4.6. Codigo sintetizable

ram.v

```
module ram
  #(parameter data_width = 4,
    parameter addr_width = 4)
   (input wire clk, rst, rw,
    input wire [addr_width-1:0] addr,
    input wire [data_width-1:0] datain,
    output reg [data_width-1:0] dataout
    );
   reg [data_width-1:0]          data [0:(addr_width*addr_width)-1];
```

```
    reg [data_width-1:0]          i;

    parameter
      WRITE = 1, READ = 0;

    always @(posedge clk, posedge rst)
      if (rst)
        begin
           for( i = 0 ; i < (addr_width*addr_width)-1 ; i = i + 1 )
             data[i] <= 0;
           data[(addr_width*addr_width)-1] <= 0;
           dataout <= 0;
        end //end if (rst)
      else
        case(rw)
          WRITE:
            begin
               data[addr] <= datain;
               dataout <= datain;
            end
          READ:
            begin
               dataout <= data[addr];
            end
        endcase // case (rw)

endmodule // ram
```

cpu.v

```
module cpu
  #(parameter data_width = 4,
    parameter addr_width = 4,
    parameter op_width   = 4,
    parameter op_param   = 4,
    parameter pc_width   = 8)
   (input wire clk, rst,
    input wire [(op_width+op_param)-1:0] instruction,
    output reg [data_width-1:0]          a
    );
```

```
reg [data_width-1:0]                    datain;
wire [data_width-1:0]                    dataout;
reg [addr_width-1:0]                    addr;
reg [op_width-1:0]                      currentState, nextState;
reg [pc_width-1:0]                      pc;
reg [(op_width+op_param)-1:0]           ir;
reg                                     dir;

parameter //Unidad de control
  START = 0, FETCH = 1, DECODE = 2, EXECUTE = 3, WAIT = 4;

parameter //Operaciones
  ldaxx = 0, ldxxa = 1, andaxx = 2, addaxx = 3,
  subaxx = 4, jzxx = 5, jmpxx = 6, nop = 7,
  movaxx = 8, nandaxx = 9, oraxx = 10, noraxx = 11,
  xoraxx = 12, xnoraxx = 13, nota = 14;

ram cpu_mem(.clk(clk), .rst(rst), .rw(dir), .addr(addr),
            .datain(datain), .dataout(dataout));

always @(posedge clk, posedge rst)
  if(rst) currentState <= START;
  else    currentState <= nextState;

always @(currentState)
  case (currentState)
    START:   nextState = FETCH;
    FETCH:   nextState = DECODE;
    DECODE:
      case(ir[(op_width+op_param)-1:op_width])
        nop:
          nextState = WAIT;
        default:
          nextState = EXECUTE;
      endcase // case (ir[(op_width+op_param)-1:op_width])
    EXECUTE: nextState = WAIT;
    WAIT: nextState = FETCH;
  endcase // case (currentState)

always @(posedge clk)
```

```
case (currentState)
  START:
    begin
       dir <= 0;
       pc <= 0;
       ir <= 0;
       a <= 0;
       datain <= 0;
       addr <= 0;
    end // case: START
  FETCH:
    begin
       dir <= 0;
       pc <= pc + 1;
       ir <= instruction;
       addr <= instruction[op_width-1:0];
    end // case: FETCH
  EXECUTE:
    begin
       dir <= 0;
       addr <= ir[op_width-1:0];
       case(ir[(op_width+op_param)-1:op_width])
         ldaxx:
           begin
              a <= dataout;
           end
         ldxxa:
           begin
              dir <= 1;
              datain <= a;
           end
         andaxx:
           begin
              a <= (a & dataout);
           end
         addaxx:
           begin
              a <= (a + dataout);
           end
         subaxx:
```

```
  begin
     a <= (a - dataout);
  end
jzxx:
  begin
     if(a === 4'b0) // Importante
       pc <= addr;
  end
jmpxx:
  begin
     pc <= addr;
  end
movaxx:
  begin
     a <= ir[op_width-1:0];
  end
nandaxx:
  begin
     a <= ~(a & dataout);
  end
oraxx:
  begin
     a <= (a | dataout);
  end
noraxx:
  begin
     a <= ~(a | dataout);
  end
xoraxx:
  begin
     a <= (a ^ dataout);
  end
xnoraxx:
  begin
     a <= ~(a ^ dataout);
  end
nota:
  begin
     a <= ~a;
  end
```

```
            endcase // case (ir[(op_width+op_param)-1:op_width])
          end // case: EXECUTE
        WAIT:
          begin
          end // case: WAIT
      endcase // case (nextState)

endmodule // cpu
```