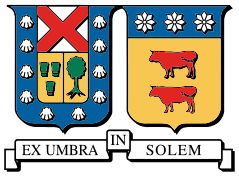

ELO325/IPD432

Informe Laboratorio # 4

Diseño de un Procesador de arquitectura μBIO

Andres Ulloa
201021003-0

Javier Marto
2704669-k

Profesor: Wolfgang Freund

27 de octubre de 2012

1. Introducción

Desde los inicios de la computación, los procesadores han sido la base de todo sistema informático, y a medida que pasan los años, se convierten en sistemas más y más complejos, con niveles de integración con billones de transistores.

La arquitectura μBIO es una arquitectura del tipo RISC, con instrucciones simples y de 16-bit. Esta arquitectura tuvo sus orígenes en las aulas de la Universidad Técnica Federico Santa María, Chile. Es una arquitectura orientada a la educación y el aprendizaje de los estudiantes de electrónica. El acrónimo μBIO hace referencia a ***micro****processor with* ***B****asic* ***I****nstructional* ***O****perations*, es decir, es una arquitectura con operaciones orientadas a la educación y el aprendizaje de los estudiantes.

El objetivo de este trabajo es diseñar, a través del lenguaje HDL Verilog, un procesador con funcionalidades básicas, de arquitectura μBIO, que luego será implementado en una FPGA para verificar su funcionalidad.

2. Diseño y Especificaciones

Este procesador posee una arquitectura de 16 bit con un diseño secuencial sin interrupciones, maneja un set reducido de 12 instrucciones, tiene 16 registros de propósito general cada uno de 16 bit, además de estos 16 registros posee 10 registros de propósitos específicos, los cuales se especifican a continuación:

- PC : Program counter
- ir : Instruction register
- rai : Index register a
- rbi : Index register b
- iv8 : Inmediate value 8 bit
- iv16 : Inmediate value 16 bit
- $data_x$: Data in/out
- addr : Address register

3. Repertorio de Instrucciones

Se dispone de tres tipos de instrucciones, instrucciones sin operando, instrucciones con dato inmediato e instrucciones con registros y operando de 16 bit, estas varían una de otra en su formato.

Las 12 instrucciones además se sub clasifican de la siguiente forma:

- 2 instrucciones de carga y almacenamiento: SW, LW.
- 2 instrucciones correspondientes a operaciones lógicas: andi, ori.
- 4 instrucciones aritméticas: add, sub, mult, addi.
- 3 instrucciones de salto: J, BZ, BNZ.
- 2 instrucciones NOP (no operation) y STOP.

4. Esquema general

La CPU consta de varios módulos, los cuales interactúan entre sí por medio de un módulo central que es el que se encarga de decodificar y cambiar los valores de los registros internos.

En el esquema se puede apreciar cómo el módulo de control de ciclo controla el estado de toda la CPU. Más adelante se tratan los módulos con más detalle.

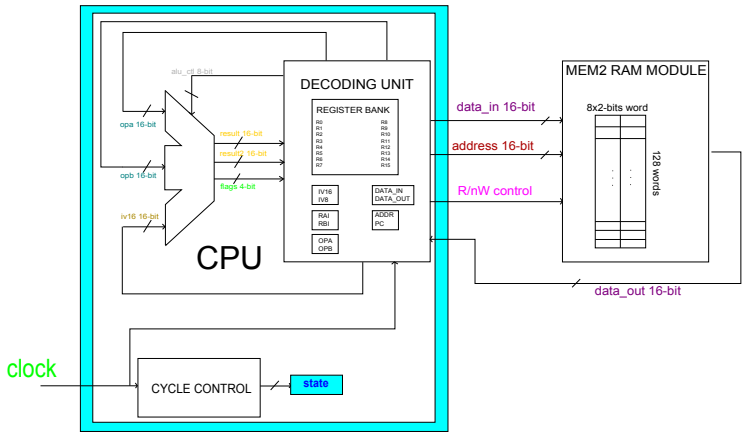

Figura 1: CPU μBIO

El código que describe este módulo es el siguiente:

```
`timescale 1ns / 1ps
`define IDLE     0
`define FETCH      1
`define DECODE   2
`define EXECUTE    3
`define WBACK      4
`define ADD      8'h12
`define SUB      8'h13
`define MUL      8'h14
`define ADDI     8'h15
`define BZ       8'h21
`define BNZ      8'h22
`define ORI      8'h0A
`define ANDI     8'h0C
`define SW       8'h02
`define LW       8'h04
`define J      8'h24
`define STP      8'hFF
`define NOP      8'h00
////////////////////////////////
//CPU uBio
////////////////////////////////
module uBio_full();
reg clk=0, rst=0;
reg [15:0] R[0:15];
reg [15:0] data_to_mem=0, opa=0, opb=0, iv16=0, address=0, pc=0;
reg [7:0] ri=0, alu_ctl=0;
reg [3:0] rai=0, rbi=0;
reg r_nWb=1;

wire [15:0] result, result2, data_from_mem;
wire [7:0] iv8;
wire [2:0] state;
wire Z, C, V, S, N;

////////////////////////////////
```

```
//SUBMODULOS
ALU ALU(opa, opb, iv16, alu_ctl, result, result2, C, V, Z, N);
CYCLE CYCLE(clk, rst, state, S);
MEM2 MEM2(r_nWb, address, data_to_mem, data_from_mem, rst);
////////////////////////////////

assign iv8={rai, rbi};
assign S=(ri==`STP);

always@(posedge clk or posedge rst)
begin
//INICIALIZACION DE REGISTROS PARA EL TEST
if(rst) begin   R[0]<=1; R[1]<=2; R[2]<=3;
          R[3]<=4; R[4]<=100; R[5]<=0;
          R[6]<=0; R[7]<=3; R[8]<=0;
          R[9]<=0; R[10]<=0; R[11]<=0;
          R[12]<=0; R[13]<=0; R[14]<=0;
          R[15]<=0; pc<=0; address<=0;
          opa<=0; opb<=0;
          end
else
//COMPORTAMIENTO DE LA LOGICA
  begin
    case(state)
    `FETCH: begin
          {rai, rbi}<=data_from_mem[7:0];
          ri<=data_from_mem[15:8];
          pc<=pc+2;
          address<=pc+2;
          end
    `DECODE:  begin
          case(ri)
          `ADD: begin opa<=R[rai]; opb<=R[rbi]; alu_ctl<=`ADD; end
          `SUB: begin opa<=R[rai]; opb<=R[rbi]; alu_ctl<=`SUB; end
          `MUL: begin opa<=R[rai]; opb<=R[rbi]; alu_ctl<=`MUL; end
          `ORI: begin opa<=R[rai]; opb<=R[rbi]; alu_ctl<=`ORI; iv16<=
               data_from_mem; pc<=pc+2; address<=pc+2; end
          `ANDI:begin opa<=R[rai]; opb<=R[rbi]; alu_ctl<=`ANDI; iv16<=
               data_from_mem; pc<=pc+2; address<=pc+2; end
          `ADDI:begin opa<=R[rai]; opb<=R[rbi]; alu_ctl<=`ADDI; iv16<=
               data_from_mem; pc<=pc+2; address<=pc+2; end
          `BZ:  begin pc<=Z?(pc + (iv8*2) - 2):pc; address<=Z?(pc + (iv8
               *2) - 2):pc; end
          `BNZ: begin pc<=!Z?(pc + (iv8*2) - 2):pc; address<=Z?(pc + (
               iv8*2) - 2):pc; end
          `J: begin iv16<=data_from_mem; end
          `NOP: begin end
          `STP: begin end
          `LW:  begin address<=data_from_mem+R[rbi]; pc<=pc+2; end
          `SW:  begin address<=data_from_mem+R[rbi]; pc<=pc+2; r_nWb<=0;
                end
          endcase
          end
    `EXECUTE:begin if(!r_nWb)data_to_mem<=R[rai]; end //ESPERAR CALCULOS
         DE LA ALU
    `WBACK: begin
          case(ri)
          `ADD: begin R[rai]<=result; end
          `SUB: begin R[rai]<=result; end
          `MUL: begin R[rbi]<=result; R[rai]<=result2; end
          `ORI: begin R[rai]<=result; end
          `ANDI:begin R[rai]<=result; end
          `ADDI:begin R[rai]<=result; end
          `BZ:  begin  end
          `BNZ: begin  end
          `J: begin pc<=(iv16 & 16'hFFFE); address<=(iv16 & 16'hFFFE);
               end
          `NOP: begin  end
          `STP: begin  end
          `LW:  begin R[rai]<=data_from_mem; address<=pc; end
          `SW:  begin r_nWb<=1; address<=pc;end
          endcase
        end
    endcase
```

```
  end
end

initial
begin
rst=0;#10;rst=1;#10;rst=0;#5;
end

always begin #10; clk=~clk; end
///////
endmodule
```

5. Módulo Control de Ciclo

Este modulo es simplemente una máquina de estados, su propósito es indicarle al procesador que hacer. Su estado cambia constantemente amenos que el flag S(stop) se encuentre en alto. Existen 5 estados para nuestro procesador los cuales se detallan a continuación:

IDLE En espera, cuando S está en alto

FETCH Capura de la instruccion

DECODE Decodificacion de la instruccion y captura del opcode si corresponde

EXECUTE Ejecuta la instruccion

WRITE-BACK Escritura de memoria o registro si corresponde

Figura 2: Módulo de cotrol de ciclos

El código que describe este módulo es el siguiente:

```
////////////////////////////////
//CYCLE uBio
////////////////////////////////
`define IDLE 0
`define FETCH 1
`define DECODE 2
`define EXECUTE 3
`define WBACK 4
module CYCLE(clk, reset, state, S);

input clk, reset, S;
output reg [2:0] state=0;

always@(posedge clk or posedge reset)
begin
  if (reset) state <= `IDLE;
  else if (S) state <= `IDLE;
  else
    case(state)
    `FETCH: state <= `DECODE;
    `DECODE: state <= `EXECUTE;
    `EXECUTE: state <= `WBACK;
    `WBACK: state <= `FETCH;
    default: state <= `FETCH;
    endcase
end

endmodule
```

6. Módulo ALU

La ALU (Arithmetic logic unit) de este procesador tiene la capacidad de realizar las operaciones lógicas antes nombradas. Además posee 4 banderas para posibles excepciones, la posible aparicion de estas banderas se detalla en la siguiente tabla.

Operacion/FLAG	add	sub	mult	andi	ori
Carry	•				
Zero	•	•	•		
Negative	•	•	•		
oVerflow		•			

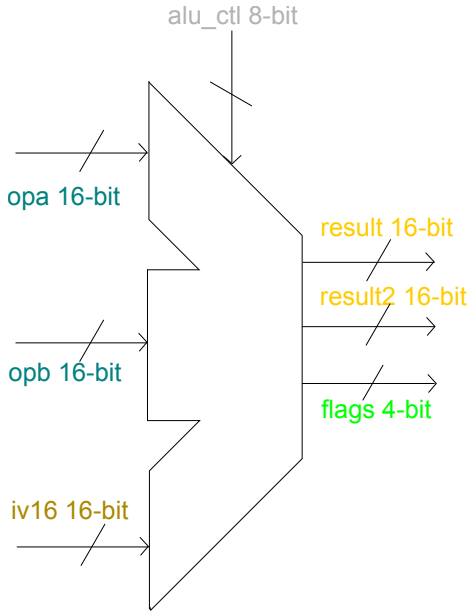

Figura 3: Módulo ALU

El código que describe este módulo es el siguiente:

```
////////////////////////////////
//ALU uBio
////////////////////////////////
`define ADD      8'h12
`define SUB      8'h13
`define MUL      8'h14
`define ORI      8'h0A
`define ANDI     8'h0C
`define NOP      8'h00
`define ADDI     8'h15

module ALU(opa, opb, iv16, alu_ctl, result, result2, C, V, Z, N);
input [15:0] opa, opb, iv16;
input [7:0] alu_ctl;
output reg [15:0] result=0, result2=0;
output reg C=0, V=0, Z=0, N=0;

always@(*)
begin
```

```
  case(alu_ctl)
    `ADD: begin
        result=opa+opb;
        C=(opa>=0 && opb>=0) ? !result[15] & (opa[15] ^ opb[15]) | (opa
            [15] & opb[15]):0;
        V=0;
        Z = (result==0);
        N = result[15];
        end
    `SUB: begin
        result = opa - opb;
        V = (opa[15] & opb[15] & ~result[15]) | ~(opa[15] & opb[15]) &
            result[15] ;
        C = 0;
        Z = (result==0);
        N = result[15];
        end
    `MUL: begin
        {result2, result}= opa * opb;
        {C, V} = 0;
        Z = (result==0);
        N = (opa[15]) || (opb[15]);
        end
    `ORI: begin
        result = opb | iv16;
        {C, V, N, Z} = 0;
        end
    `ANDI:begin
        result = opb & iv16;
        {C, V, N, Z} = 0;
        end
    `ADDI:begin
        result = opb + iv16;
        {C, V, N, Z} = 0;
        end
    default: begin end
  endcase
end
//initial begin #10;alu_ctl=`ADD; #10;opa=10; #5; opb=5; #10; alu_ctl=
    `MUL; end
endmodule
```

7. Módulo RAM "MEM2"

El módulo de memoria posee un banco de datos de 128 palabras de 16-bits, las cuales están divididas en 2 bytes, y cada uno de estos bytes es accesible a través de su dirección de memoria.

El módulo MEM2 posee un control de lectura/escritura de datos *R/nW* (1 para lectura, 0 para escritura).

Además la memoria posee un bus de entrada *data_in* que es el dato que se guarda en la dirección especificada por *address* cuando *R/nW* está en bajo, y un buffer de salida *data_out* que contiene el valor de la memoria indicada por *address* cuando *R/nW* está en alto.

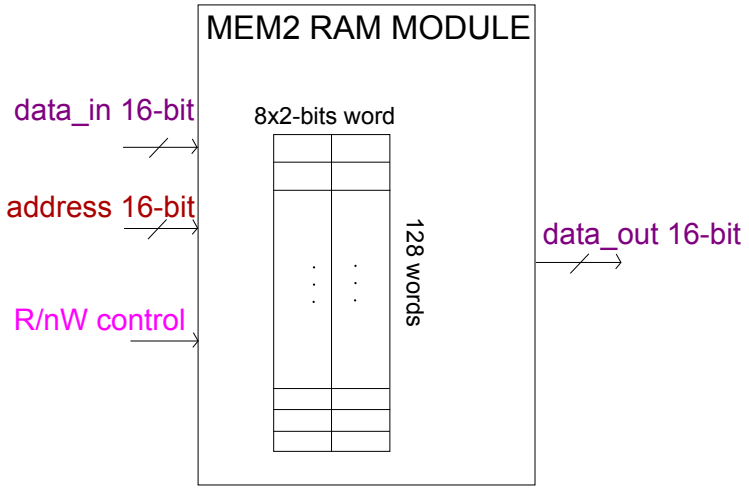

Figura 4: Módulo RAM "MEM2"

La codificación de este módulo es bastante sencilla:

```
module MEM2(r_nWb, addr, in_data, out_data, rst);
input r_nWb, rst;
input [15:0] addr, in_data;
output reg [15:0] out_data=0;
reg [7:0] mem_data [0:127];

always@(rst or r_nWb or addr or in_data)
begin
if(rst)
  begin
  //INICIALIZACION MEMORIA
  end
else
  begin
    if (r_nWb) out_data={mem_data[addr], mem_data[addr+1]};
    else {mem_data[addr], mem_data[addr+1]}=in_data;
  end
end
endmodule
```

8. Programa de prueba

Para realizar la prueba, se incializa el módulo MEM2 con el código del programa, como sigue:

```
module MEM2(r_nWb, addr, in_data, out_data, rst);
input r_nWb, rst;
input [15:0] addr, in_data;
output reg [15:0] out_data=0;
reg [7:0] mem_data [0:127];

always@(rst or r_nWb or addr or in_data)
begin
if(rst)
  //INICIALIZACION MEMORIA
  begin
    //ADD TEST
    {mem_data[0], mem_data[1]} = 'h1201; //R0+R1-> R0,{R0=3, R1=1, R2=3,
        R3=4}
    //SUB TEST
    {mem_data[2], mem_data[3]} = 'h1301; //R0-R1->R0,{R0=1, R1=1, R2=3,
        R3=4}
    //MULT TEST
    {mem_data[4], mem_data[5]} = 'h1421; //R2*R1->R2,{R2=0, R1=6, R2=3,
        R3=4}
    //ORI TEST
    {mem_data[6], mem_data[7]} = 'h0A01; //R1 | imm -> R0
    {mem_data[8], mem_data[9]} = 'h0108; //{R0=h010E, R1=6, R2=3, R3=4}
    //ANDI TEST
    {mem_data[10], mem_data[11]} = 'h0C21; //R1 & imm ->R2
    {mem_data[12], mem_data[13]} = 'hFFAA; //{R0=h010E, R1=6, R2=h0002,
        R3=4}
    //BZ TEST1
    {mem_data[14], mem_data[15]} = 'h1461; //R6*R1->{R6=0,R1=0}, {R0=
        h010E, R1=0, R2=h0002, R3=4}
    {mem_data[16], mem_data[17]} = 'h2102; //BZ->L1
    {mem_data[18], mem_data[19]} = 'h1201; //R0+R1->R0, {R0=h010E, R1=0,
        R2=h0002, R3=4}
    {mem_data[20], mem_data[21]} = 'h1203; //L1: R0+R3->R0, {R0=h0112,
        R1=0, R2=h0002, R3=4}
    //BZ TEST2
    {mem_data[22], mem_data[23]} = 'h1430; //R3*R0->{R3=0,R0=448}, {R0=
        h0448, R1=0, R2=h0002, R3=0}
    {mem_data[24], mem_data[25]} = 'h2102; //BZ->L1
    {mem_data[26], mem_data[27]} = 'h1200; //R0+R0->R0, {R0=h0890, R1=0,
        R2=h0002, R3=0}
    {mem_data[28], mem_data[29]} = 'h1200; //L1: R0+R0->R0, {R0=h1120,
        R1=0, R2=h0002, R3=0}
    //J TEST
    {mem_data[30], mem_data[31]} = 'h1210; //R1+R0->R1=R0, {R0=h1120, R1
        =h1120, R2=h0002, R3=0}
    {mem_data[32], mem_data[33]} = 'h2400; //J
    {mem_data[34], mem_data[35]} = 'h002E; //J->46
    {mem_data[36], mem_data[37]} = 'h1200; //SALTANDO
    {mem_data[38], mem_data[39]} = 'h1200; //SALTANDO
    {mem_data[40], mem_data[41]} = 'h1200; //SALTANDO
    {mem_data[42], mem_data[43]} = 'h1200; //SALTANDO
    {mem_data[44], mem_data[45]} = 'h1200; //SALTANDO
    {mem_data[46], mem_data[47]} = 'h1200; //R0+R0->R0, {R0=h2240, R1=
        h1120, R2=h0002, R3=0}
    //NOP TEST
    {mem_data[48], mem_data[49]} = 'h0000; //PC+2->PC=h32
    {mem_data[50], mem_data[51]} = 'h0011; //PC+2->PC=h34
    {mem_data[52], mem_data[53]} = 'h00FF; //PC+2->PC=h36 (54)
    //SW TEST
    {mem_data[54], mem_data[55]} = 'h0274; //R7->M[R4 + imm], {R4=100,
        imm=2, R7=h0003}
    {mem_data[56], mem_data[57]} = 'h0002; //imm
    //LW TEST
    {mem_data[58], mem_data[59]} = 'h0454; //M[R4 + imm]->R5, {R4=100,
        imm=2, R5=h0003}
    {mem_data[60], mem_data[61]} = 'h0002; //imm
    {mem_data[62], mem_data[63]} = 'h1255; //R5+R5->R5, {R5=h0006}
```

```
    //ADDI TEST
    {mem_data[64], mem_data[65]} = 'h1532; //R2+imm->R3, {R2=h0002, imm=
        h0FFF, R3=h1001}
    {mem_data[66], mem_data[67]} = 'h0FFF; //imm
    //STP TEST
    {mem_data[68], mem_data[69]} = 'hFF00; //STOP, PC=h38 (56)
    {mem_data[70], mem_data[71]} = 'h00FF; //PC NO DEBE INCREMENTARSE,
        PC=h38 (56)
    //VALORES FINALES DE REGISTROS
    //{R0=h2240 ,R1=h1120 , R2=h0002 , R3=h1001 , R4=h0064 , R5=h0006 ,
        R6=h0000 , R7=h0003 }

    //hig_mem
    {mem_data[100], mem_data[101]} = 'h0001;
    {mem_data[102], mem_data[103]} = 'h0000;//EN SW TEST, ESTE VALOR
        CAMBIA A h0003
    {mem_data[104], mem_data[105]} = 'h0004;
    {mem_data[106], mem_data[107]} = 'h0005;
    {mem_data[108], mem_data[109]} = 'h0006;
    {mem_data[110], mem_data[111]} = 'h0007;
  end
else
  begin
    if (r_nWb) out_data={mem_data[addr], mem_data[addr+1]};
    else {mem_data[addr], mem_data[addr+1]}=in_data;
  end
end
endmodule
```

Esto inicializa la memoria RAM de la siguiente forma:

	0	1	2	3	4	5	6	7
0	12	01	13	01	14	21	0A	01
8	01	08	0C	21	FF	AA	14	61
16	21	02	12	01	12	03	14	30
24	21	02	12	00	12	00	12	10
32	24	00	00	2E	12	00	12	00
40	12	00	12	00	12	00	12	00
48	00	00	00	11	00	FF	02	74
56	00	02	04	54	00	02	12	55
64	15	32	0F	FF	FF	00	00	FF
72	XX	XX	XX	XX	XX	XX	XX	XX
80	XX	XX	XX	XX	XX	XX	XX	XX
88	XX	XX	XX	XX	XX	XX	XX	XX
96	XX	XX	XX	XX	00	01	00	00
104	00	04	00	05	00	06	00	07
112	XX	XX	XX	XX	XX	XX	XX	XX
120	XX	XX	XX	XX	XX	XX	XX	XX

Figura 5: Memoria inicializada

9. Simulación

Para la simulación, primero se hace un *reset* para inicializar los registros y los valores de memoria (Ampliar la figura para ver los valores de la simulación).

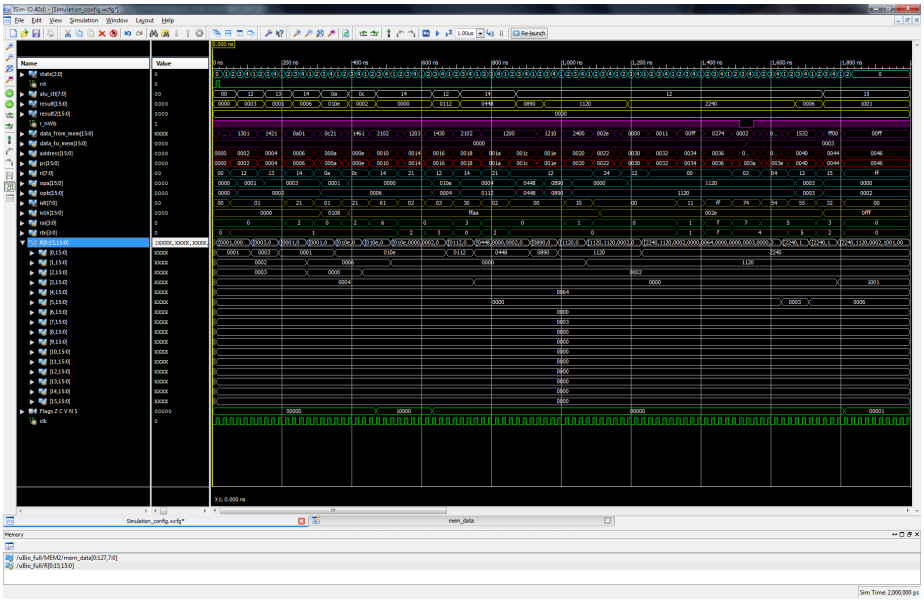

Figura 6: Simulación del programa de prueba cargado en el módulo RAM

Los resultados de la simulación coinciden con el seguimiento de las variables hecho en el programa de prueba, lo que indica que el CPU diseñado es funcional.

10. Conclusiones

Diseñar un CPU se puede tornar imposible sin un buen diseño y entendimiento previo del funcionamiento del dispositivo. Además, el código puede volverse incomprensible si se abusa de los bloques always, y de los flags para la comunicación entre estos.

Una buena idea para comenzar el diseño es hacer una CPU con una sola instrucción, probarla e ir agregando las instrucciones de manera progresiva.

Para la simulación en *ISim* es una buena idea cambiar las entradas y las salidas por registros internos temporalmente. Así, la simulación permite ver la totalidad de los registros internos y encontrar fácilmente dónde se encuentran los problemas de ejecución.

Un punto importante para la comunicación con la memoria RAM fue la estabilidad de las entradas al momento de la escritura/lectura, de hecho más del 50 % del tiempo fue invertido en lograr compatibilizar las funciones LW y SW.

La síntesis del microprocesador es posible en una FPGA Basys 1 si se reducen el número de registros y el tamaño de la memoria.