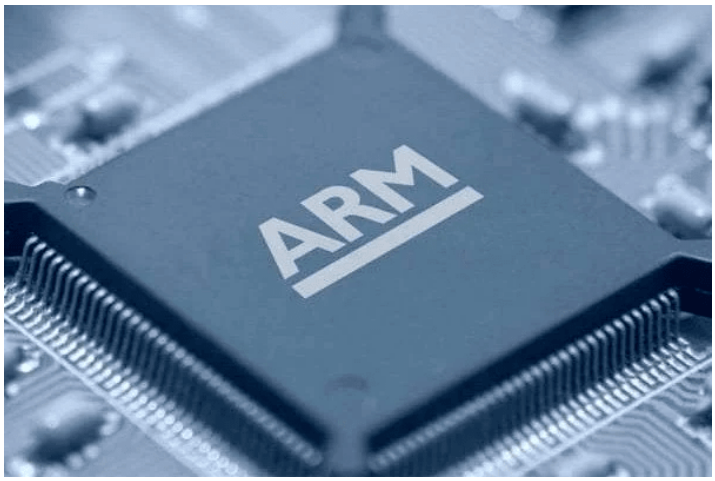
Universidad de Ingeniería y Tecnología

Arquitectura de computadoras

Multi-cycle Processor Implementation



Entrega del Laboratorio N°6

Integrantes:
Méndez Lázaro, Luis Fernando (100 % de participación)
Mora Huamanchay, Angel Obed (100 % de participación)
Wong Orrillo, Jose Francisco (100 % de participación)

5 de Noviembre de 2022

1. Main FSM output table

En esta sección se muestra la tabla de los outputs de cada estado del main FSM dentro de nuestro controller. Esta tabla fue realizada con la figura 3 de la guía del Laboratorio 6.

State (name)	NextPC	Branch	MemW	RegW	IRWrite	AdrSrc	ResultSrc	ALUSrcA		ALUSrcB	ALUOp	FSM Control Word
0 (Fetch)	1	0	0	0	1	0	10	0	1	10	0	0x114C
1 (Decode)	0	0	0	0	0	0	10	0	1	10	0	0x004C
2 (MemAdr)	0	0	0	0	0	0	00	0	0	01	0	0x0002
3 (MemRead)	0	0	0	0	0	1	00	0	0	00	0	0x0080
4 (MemWB)	0	0	0	1	0	0	01	0	0	00	0	0x0220
5 (MemWrite)	0	0	1	0	0	1	00	0	0	00	0	0x0480
6 (ExecuteR)	0	0	0	0	0	0	00	0	0	00	1	0x0001
7 (ExecuteI)	0	0	0	0	0	0	00	0	0	01	1	0x0003
8 (ALUWB)	0	0	0	1	0	0	00	0	0	00	0	0x0200
9 (Branch)	0	1	0	0	0	0	10	1	0	01	0	0x0852

Cuadro 1: Output table for main FSM

2. Multicycle Verilog code

En esta sección se muestra el código de cada módulo que compone el controller de nuestro multicycle processor. El código fuente se encuentra en el archivo controller.v en el entregable.

2.1. controller

```
1
2  module controller (
3    clk,
4    reset,
5    Instr,
6    ALUFlags,
7    PCWrite,
8    MemWrite,
9    RegWrite,
10   IRWrite,
11   AdrSrc,
12   RegSrc,
13   ALUSrcA,
14   ALUSrcB,
15   ResultSrc,
16   ImmSrc,
17   ALUControl
18 );
19   input wire clk;
20   input wire reset;
21   input wire [31:12] Instr;
22   input wire [3:0] ALUFlags;
23
24   output wire PCWrite;
25   output wire MemWrite;
26   output wire RegWrite;
27   output wire IRWrite;
28   output wire AdrSrc;
29   output wire [1:0] RegSrc;
30   output wire [1:0] ALUSrcA;
31   output wire [1:0] ALUSrcB;
32   output wire [1:0] ResultSrc;
33   output wire [1:0] ImmSrc;
34   output wire [1:0] ALUControl;
35
36   wire [1:0] FlagW;
37   wire PCS;
38   wire NextPC;
39   wire RegW;
40   wire MemW;
41   decode dec(
42     .clk(clk),
43     .reset(reset),
44     .Op(Instr[27:26]),
45     .Funct(Instr[25:20]),
46     .Rd(Instr[15:12]),
47     .FlagW(FlagW),
48     .PCS(PCS),
49     .NextPC(NextPC),
50     .RegW(RegW),
51     .MemW(MemW),
52     .IRWrite(IRWrite),
53     .AdrSrc(AdrSrc),
54     .ResultSrc(ResultSrc),
55     .ALUSrcA(ALUSrcA),
56     .ALUSrcB(ALUSrcB),
57     .ImmSrc(ImmSrc),
58     .RegSrc(RegSrc),
59     .ALUControl(ALUControl)
60   );
61   condlogic cl(
62     .clk(clk),
63     .reset(reset),
```

```
    .Cond(Instr[31:28]),
    .ALUFlags(ALUFlags),
    .FlagW(FlagW),
    .PCS(PCS),
    .NextPC(NextPC),
    .RegW(RegW),
    .MemW(MemW),
    .PCWrite(PCWrite),
    .RegWrite(RegWrite),
    .MemWrite(MemWrite)
  );
endmodule


```

2.2. decode

```

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

  // Main FSM
  mainfsm fsm(
    .clk(clk),
    .reset(reset),
    .Op(Op),
    .Funct(Funct),
    .IRWrite(IRWrite),
    .AdrSrc(AdrSrc),
    .ALUSrcA(ALUSrcA),
    .ALUSrcB(ALUSrcB),
    .ResultSrc(ResultSrc),
```

```
    .NextPC(NextPC),
    .RegW(RegW),
    .MemW(MemW),
    .Branch(Branch),
    .ALUOp(ALUOp)
  );

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

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

  // Instr Decoder
  assign ImmSrc = Op;
  always @(*)
    case (Op)
      2'b00: RegSrc = 2'b00;
      2'b01: RegSrc = 2'b10;
      2'b10: RegSrc = 2'b01;
      default: RegSrc = 2'bxx;
    endcase
endmodule


```

2.3. condlogic

```

module condlogic (
  clk,
  reset,
  Cond,
  ALUFlags,
  FlagW,
  PCS,
  NextPC,
  RegW,
  MemW,
  PCWrite,
  RegWrite,
  MemWrite
);
  input wire clk;
  input wire reset;
  input wire [3:0] Cond;
  input wire [3:0] ALUFlags;
  input wire [1:0] FlagW;
  input wire PCS;
  input wire NextPC;
  input wire RegW;
  input wire MemW;
  output wire PCWrite;
  output wire RegWrite;
  output wire MemWrite;
```

```
  wire [1:0] FlagWrite;
  wire [3:0] Flags;
  wire CondEx;

  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)
  );

  // Delay writing flags until ALUWB state
  flopr #(2) flagwritereg(
    .clk(clk),
    .reset(reset),
    .d(FlagW & {2 {CondEx}}),
    .q(FlagWrite)
  );

  assign FlagWrite = FlagW & {2 {CondEx}};
  assign RegWrite = RegW & CondEx;
  assign MemWrite = MemW & CondEx;
  assign PCWrite = (PCS & CondEx) | NextPC;

endmodule

```

2.4. mainfsm

```

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

```
  output wire NextPC;
  output wire RegW;
  output wire MemW;
  output wire Branch;
  output wire ALUOp;
  reg [3:0] state;
  reg [3:0] nextstate;
  reg [12:0] controls;

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

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

    // 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; // LDR
        else nextstate = MEMWR; // STR
      MEMRD: nextstate = MEMWB;
      default: nextstate = FETCH;
    endcase

  // state-dependent output logic
  always @(*)
    case (state)
      FETCH: controls = 13'b1000101001100;
      DECODE: controls = 13'b0000001001100;
      EXECUTER: controls = 13'b0000000000001;
      EXECUTEI: controls = 13'b0000000000011;
      ALUWB: controls = 13'b0001000000000;
      MEMADR: controls = 13'b0000000000010;
      MEMWR: controls = 13'b0010010000000; // MemWrite
      MEMRD: controls = 13'b0000010000000; // MemRead
      MEMWB: controls = 13'b0001000100000;
      BRANCH: controls = 13'b0100001010010;
      default: controls = 13'bxxxxxxxxxxxxx;
    endcase
  assign {NextPC, Branch, MemW, RegW, IRWrite, AdrSrc, ResultSrc, ALUSrcA, ALUSrcB,
    ALUOp} = controls;
endmodule

```

3. controller-tb.v testbench module

En esta sección se muestra el código del testbench que se ha diseñado para el controller del procesador multicycle de este laboratorio. El código se encuentra dentro del archivo controller_tb.v dentro del entregable.

```
1
2  `include "controller.v"
3  `timescale 1ns / 1ps
4
5  module controller_tb;
6      // inputs
7      reg clk;
8      reg reset;
9      reg [31:12] Instr;
10     reg [3:0] ALUFlags;
11
12     // outputs
13     wire PCWrite;
14     wire MemWrite;
15     wire RegWrite;
16     wire IRWrite;
17     wire AdrSrc;
18     wire [1:0] RegSrc;
19     wire [1:0] ALUSrcA;
20     wire [1:0] ALUSrcB;
21     wire [1:0] ResultSrc;
22     wire [1:0] ImmSrc;
23     wire [1:0] ALUControl;
24
25
26   controller c(
27     .clk(clk),
28     .reset(reset),
29     .Instr(Instr),
30     .ALUFlags(ALUFlags),
31     .PCWrite(PCWrite),
32     .MemWrite(MemWrite),
33     .RegWrite(RegWrite),
34     .IRWrite(IRWrite),
35     .AdrSrc(AdrSrc),
36     .RegSrc(RegSrc),
37     .ALUSrcA(ALUSrcA),
38     .ALUSrcB(ALUSrcB),
39     .ResultSrc(ResultSrc),
40     .ImmSrc(ImmSrc),
41     .ALUControl(ALUControl)
42   );
43
44
45     initial begin
46     $dumpfile("controller_tb.vcd");
47     $dumpvars;
48         reset <= 1; #10 ; reset <= 0;
49     end
50
51     always begin
52         clk <= 1; #5; clk <= 0; #5;
53     end
54
55   initial begin
56         #10;
57         // 20'b11100000010011110000
58         Instr = 20'b11100000010011110000;   // MAIN      SUB R0, R15, R15      ; R0 = 0
59         ALUFlags = 4'b0100;
60
61         #40;
62         Instr = 20'b11100010100000000010;   //           ADD R2, R0, #5        ; R2 = 5
63         ALUFlags = 4'b0000;
64         #40;
65         Instr = 20'b11100010100000000011;   //           ADD R3, R0, #12       ; R3 = 12
```

```
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100010010000110111;   //          SUB R7, R3, #9       ; R7 = 3
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100011100001110100;   //          ORR R4, R7, R2       ; R4 = 3 OR 5 = 7
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100000000000110101;   //          AND R5, R3, R4       ; R5 = 12 AND 7 = 4
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100000100001010101;   //          ADD R5, R5, R4       ; R5 = 4 + 7 = 11
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100000010101011000;   //          SUBS R8, R5, R7      ; R8 <= 11 - 3 = 8, set Flags
        ALUFlags = 4'b0010;
        #40;
        Instr = 20'b00001010000000000000;   //          BEQ END              ; shouldn't be taken
        ALUFlags = 4'b0000;
        #30;
        Instr = 20'b11100000010100111000;   //          SUBS R8, R3, R4      ; R8 = 12 - 7  = 5
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b10101010000000000000;   //          BGE AROUND           ; should be taken
        ALUFlags = 4'b0000;
        #30;
        Instr = 20'b11100010100000000101;   //          ADD R5, R0, #0       ; should be skipped
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100000010101111000;   // AROUND   SUBS R8, R7, R2      ; R8 = 3 - 5 = -2, set Flags
        ALUFlags = 4'b1000;
        #40;
        Instr = 20'b10110010100001010111;   //          ADDLT R7, R5, #1     ; R7 = 11 + 1 = 12
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100000010001110111;   //          SUB R7, R7, R2       ; R7 = 12 - 5 = 7
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100101100000110111;   //          STR R7, [R3, #84]    ; mem [12+84] = 7
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100101100100000010;   //          LDR R2, [R0, #96]    ; R2 = mem [96] = 7
        ALUFlags = 4'b0000;
        #50;
        Instr = 20'b11100000100011111111;   //          ADD R15, R15, R0   ; PC <- PC + 8 (skips next)
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100010100000000010;   //          ADD R2, R0, #14      ; shouldn't happen
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11101010000000000000;   //          B END                ; always taken
        ALUFlags = 4'b0000;
        #30;
        Instr = 20'b11100010100000000010;   //          ADD R2, R0, #13      ; shouldn't happen
```

```
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100010100000000010;   //          ADD R2, R0, #10    ; shouldn't
    happen
        ALUFlags = 4'b0000;
        #40;
        Instr = 20'b11100101100000000010;   // END      STR R2, [R0, #100]   ; mem[100]
     = 7
        ALUFlags = 4'b0000;
        #40;
        $finish;
    end
endmodule


```

4. Simulation waveform

En esta sección, primero daremos una explicación de nuestras suposiciones acerca del funcionamiento de cada tipo de instrucción ARM Assembly presente en nuestro testbench. Al final, mostraremos las formas de onda de cada instrucción.

4.1. Assumptions

4.1.1. Data Processing Instrutions

Para una instrucción de Data Processing nuestro procesador requiere de 4 ciclos para ejecutarla correctamente, y los estados por los que pasa el Main FSM dependen de si utiliza immediates o no.

Si solo toma registros, el Main FSM debería pasar por S0 (Fetch), S1 (Decode), S6 (ExecuteR) y S8 (ALUWB).

Si toma immediates, el Main FSM debería pasar por S0 (Fetch), S1 (Decode), S7 (ExecuteI) y S8 (ALUWB).

4.1.2. Memory Instructions

Los ciclos que requiere el procesador y los estados por los que pasa el Main FSM dependen de qué instrucción de memoria sea la que ejecute.

Para una instrucción STR nuestro procesador requiere de 4 ciclos para ejecutarla correctamente, y el Main FSM debería pasar por los siguientes estados: S0 (Fetch), S1 (Decode), S2 (MemAdr) y S5 (MemWrite).

Para una instrucción LDR nuestro procesador requiere de 5 ciclos para ejecutarla correctamente, y el Main FSM debería pasar por los siguientes estados: S0 (Fetch), S1 (Decode), S2 (MemAdr) y S3 (MemRead) y S4 (MemWB).

4.1.3. Branch Instruction

Para una instrucción de Branch nuestro procesador requiere de 3 ciclos para ejecutarla correctamente, y el Main FSM debería pasar por los siguientes estados: S0 (Fetch), S1 (Decode) y S9 (Branch).

Para verificar el correcto funcionamiento de nuestro controller, se verificará que cada señal mostrada en las formas de onda de cada instrucción corresponda con nuestras suposiciones guiándonos de la tabla de outputs de la primera sección.

4.2. Waveforms

A continuación, se mostrarán las waveforms producidas por el controller al ejecutar las instrucciones dentro del memfile.dat para corroborar el correcto funcionamiento de nuestro controller. No obstante, se adjunta un video dentro del entregable donde se visualiza de forma completa las señales producidas.

4.2.1. MAIN SUB R0, R15, R15

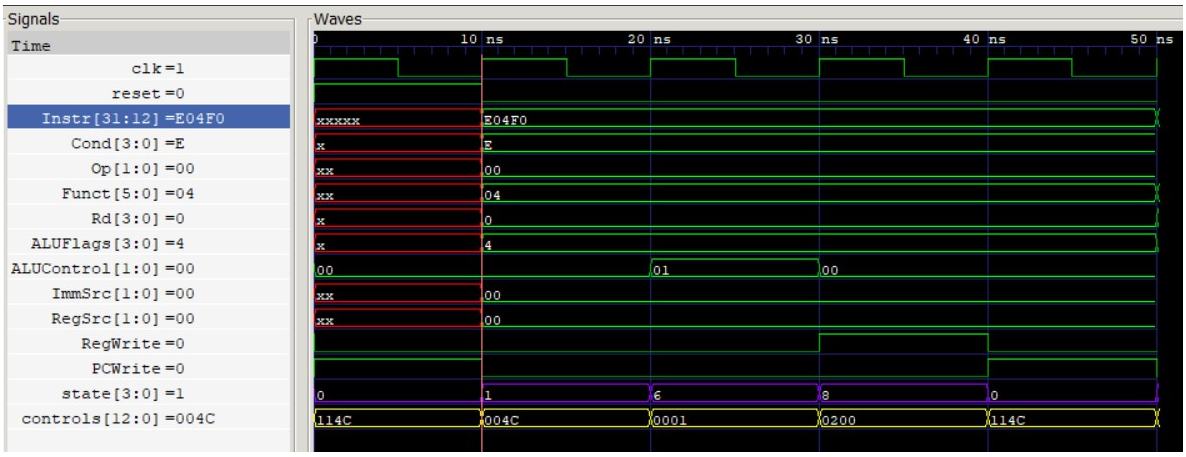


Figura 1: Simulated waveform for the instruction

4.2.2. ADD R2, R0, #5

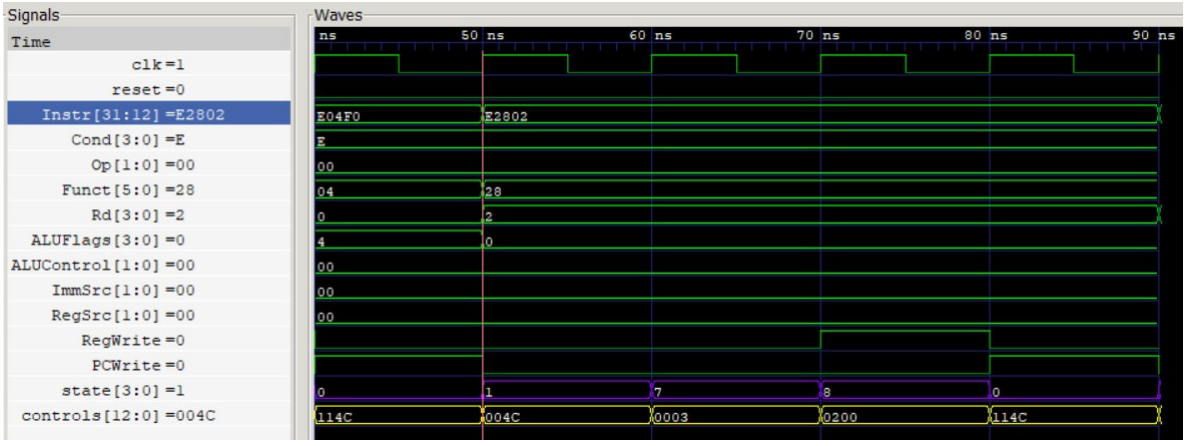


Figura 2: Simulated waveform for the instruction

4.2.3. ADD R3, R0, #12

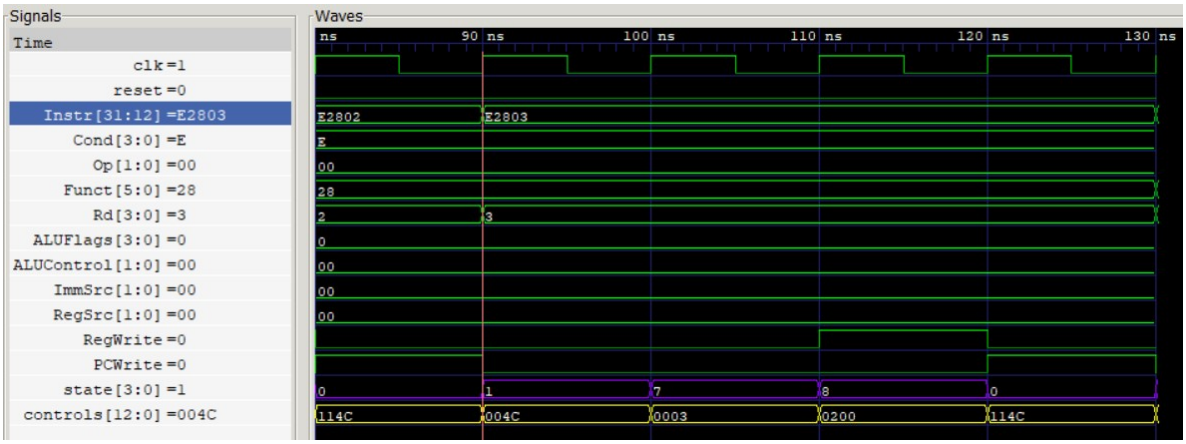


Figura 3: Simulated waveform for the instruction

4.2.4. SUB R7, R3, #9

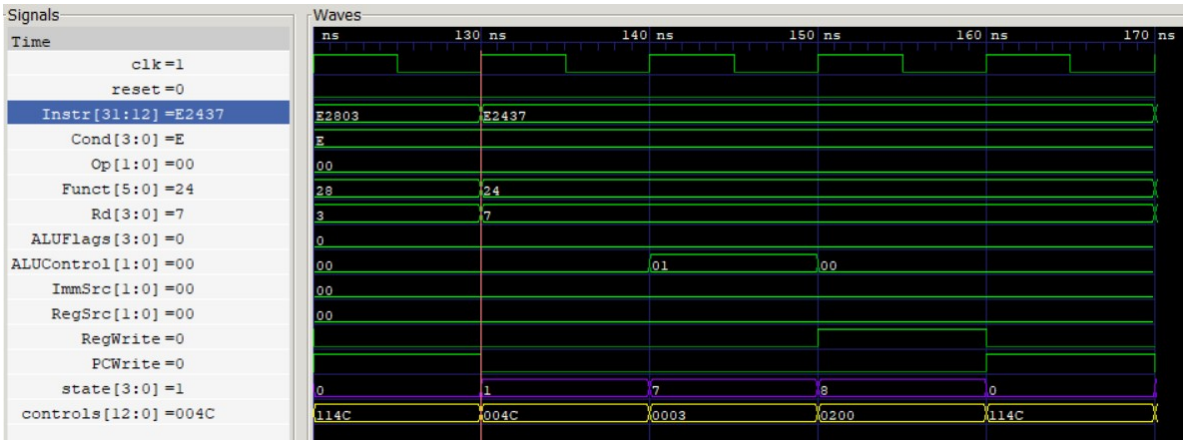


Figura 4: Simulated waveform for the instruction

4.2.5. ORR R4, R7, R2

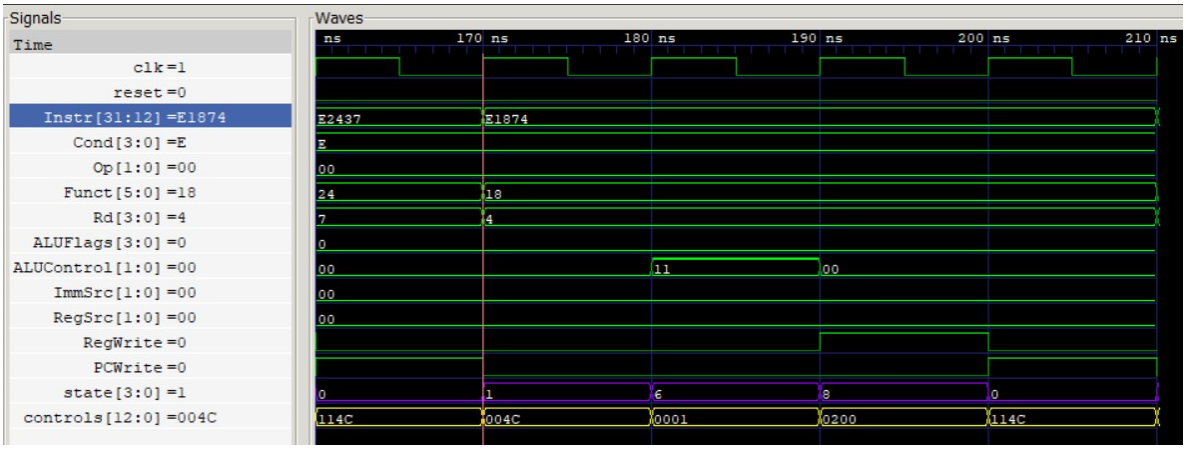


Figura 5: Simulated waveform for the instruction

4.2.6. AND R5, R3, R4

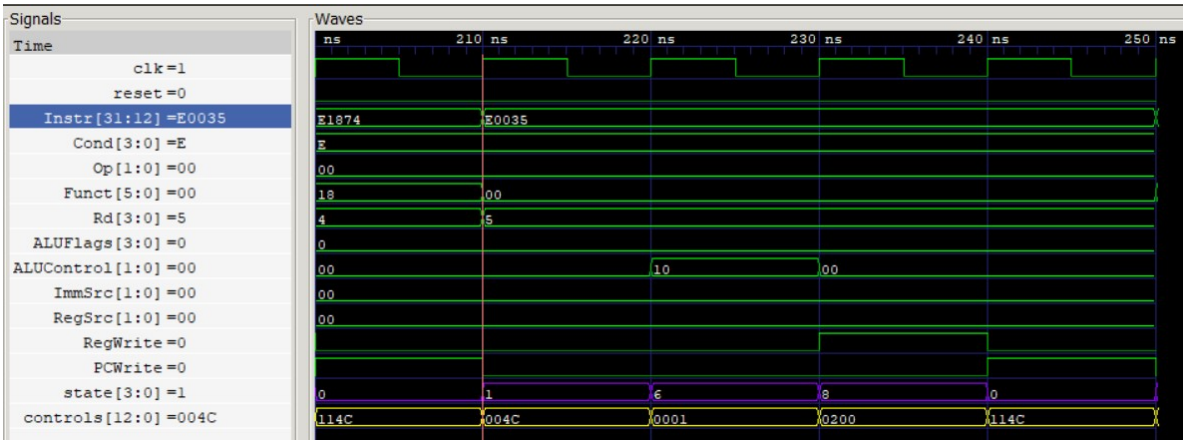


Figura 6: Simulated waveform for the instruction

4.2.7. ADD R5, R5, R4

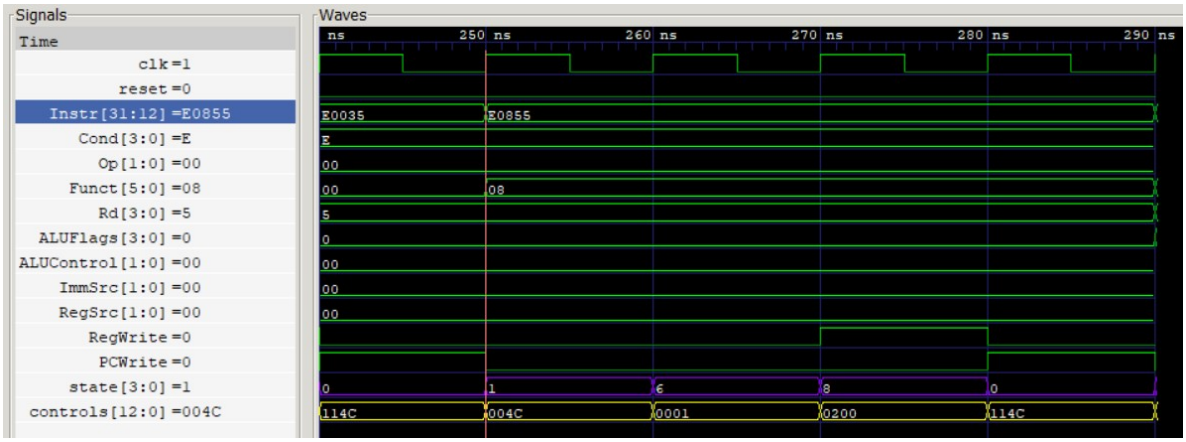


Figura 7: Simulated waveform for the instruction

4.2.8. SUBS R8, R5, R7

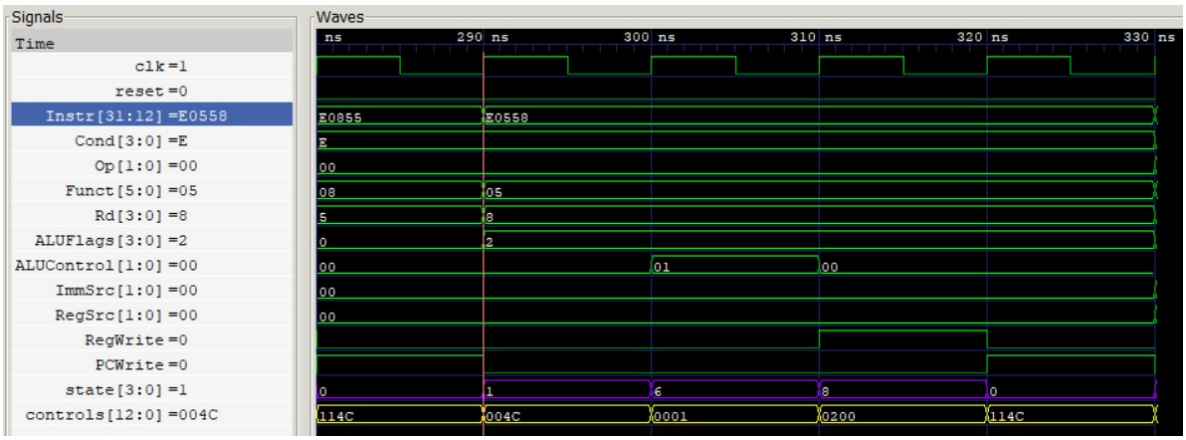


Figura 8: Simulated waveform for the instruction

4.2.9. BEQ END

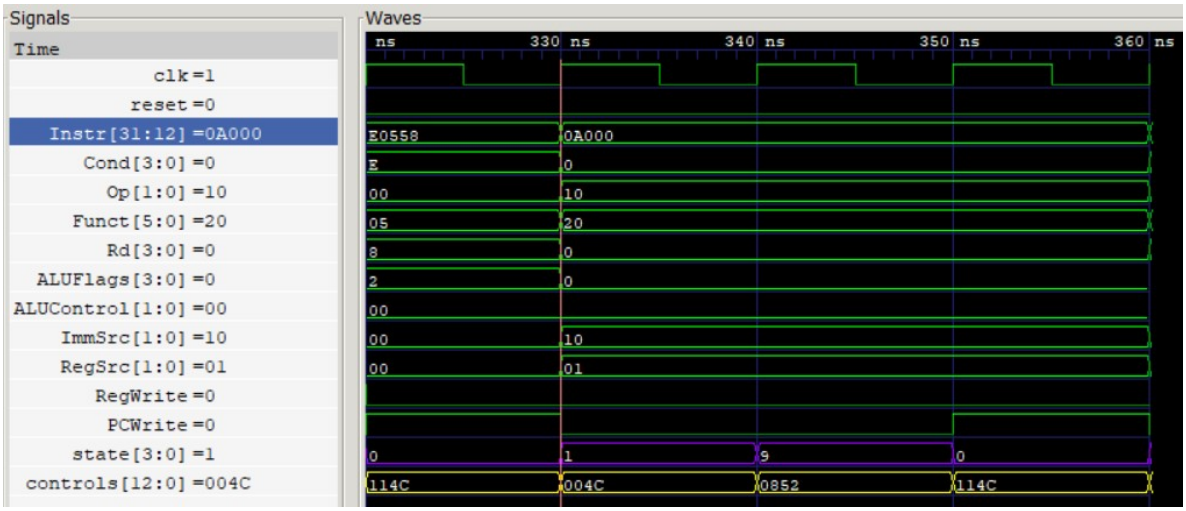


Figura 9: Simulated waveform for the instruction

4.2.10. SUBS R8, R3, R4

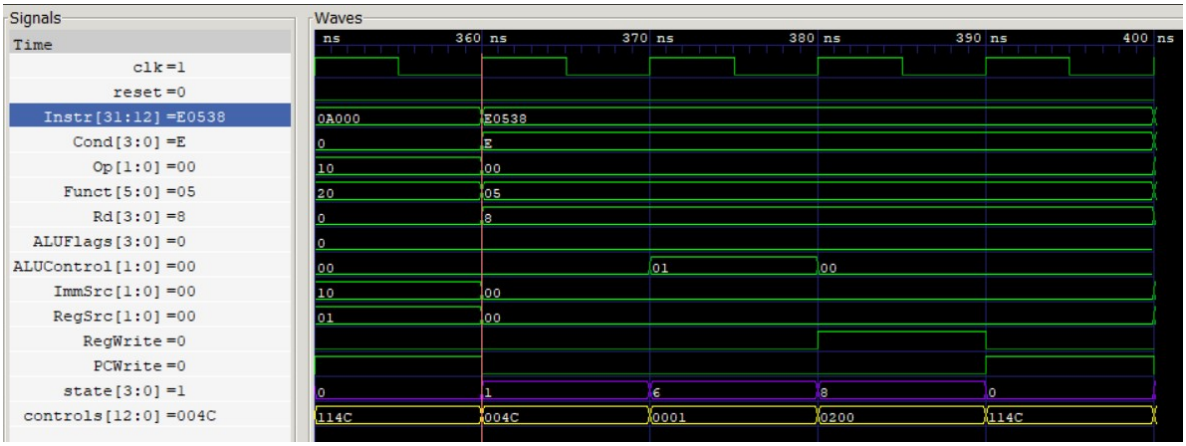


Figura 10: Simulated waveform for the instruction

4.2.11. BGE AROUND

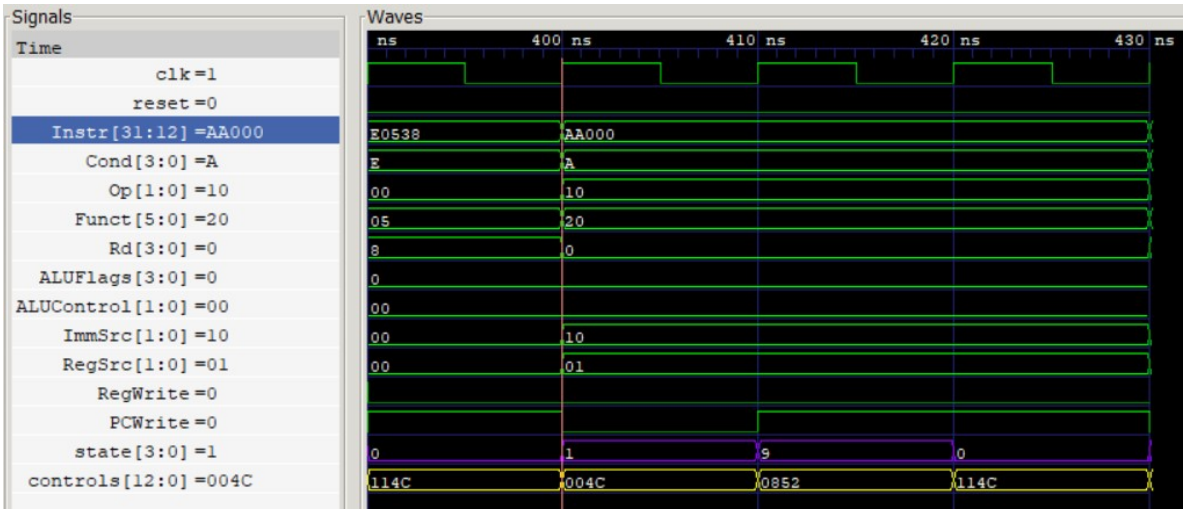


Figura 11: Simulated waveform for the instruction

4.2.12. ADD R5, R0, #0

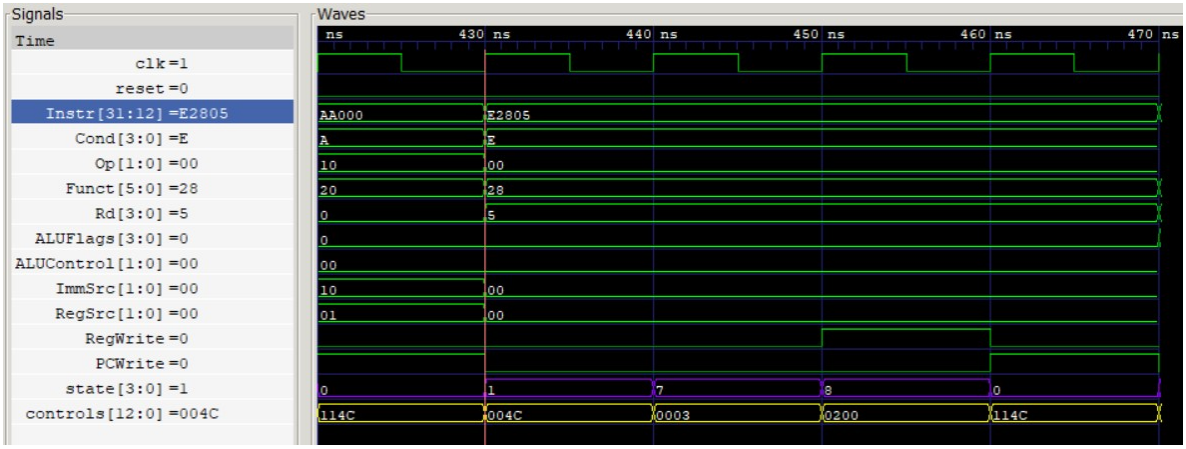


Figura 12: Simulated waveform for the instruction

4.2.13. AROUND SUBS R8, R7, R2

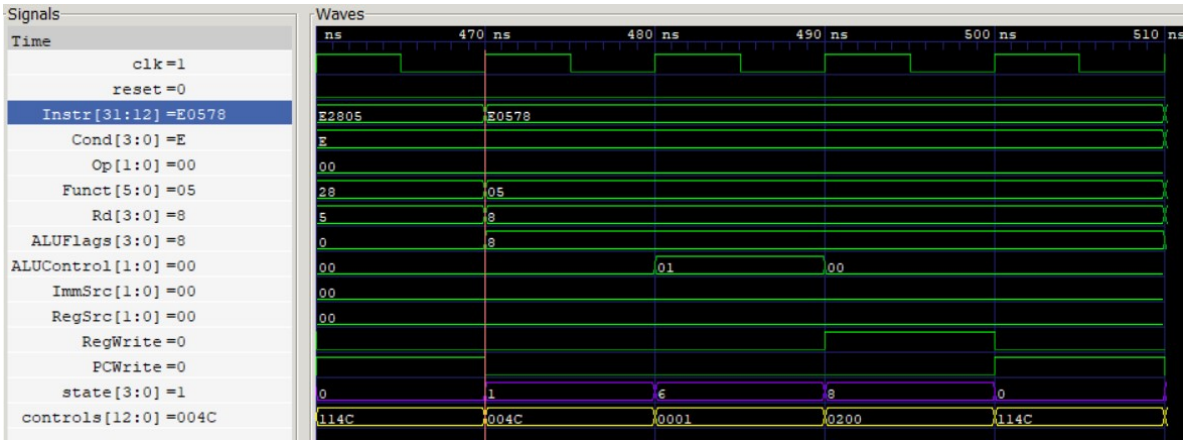


Figura 13: Simulated waveform for the instruction

4.2.14. ADDLT R7, R5, #1

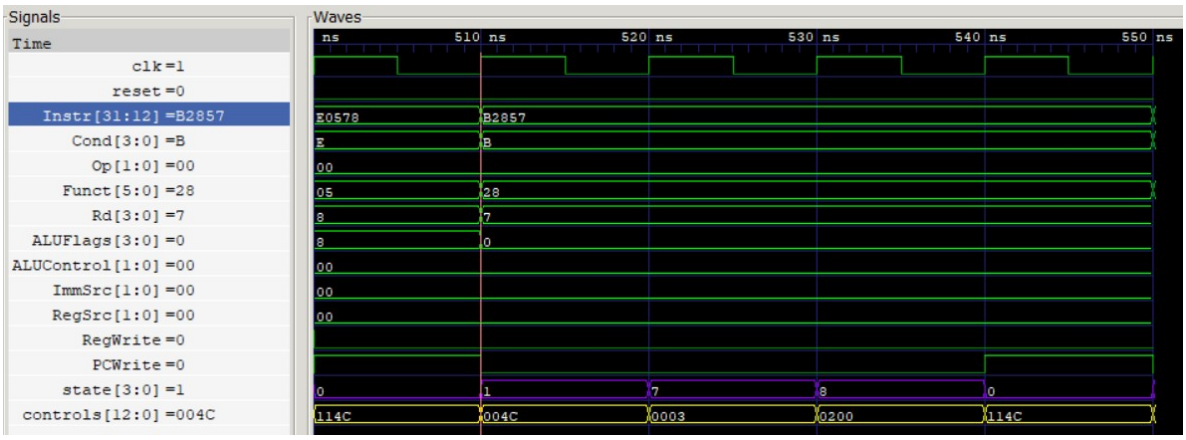


Figura 14: Simulated waveform for the instruction

4.2.15. SUB R7, R7, R2

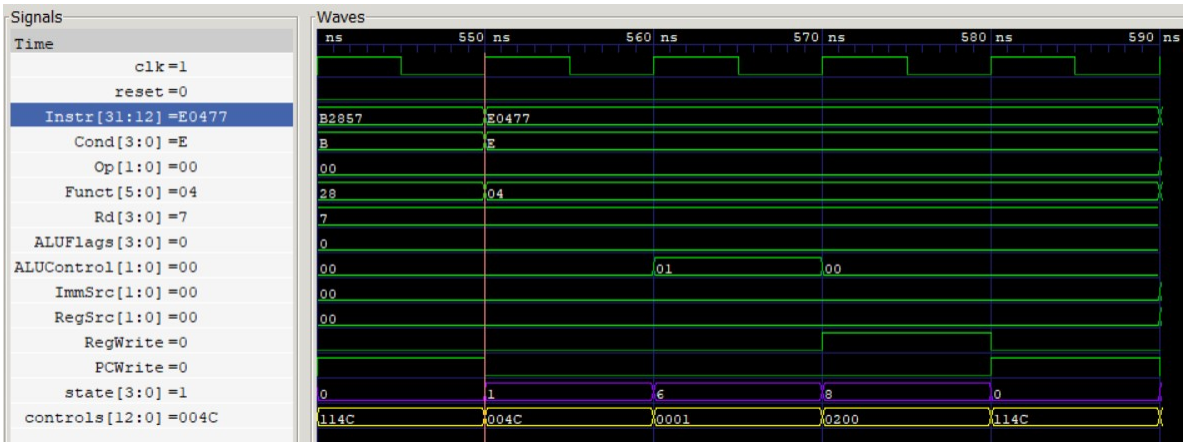


Figura 15: Simulated waveform for the instruction

4.2.16. STR R7, [R3, #84]

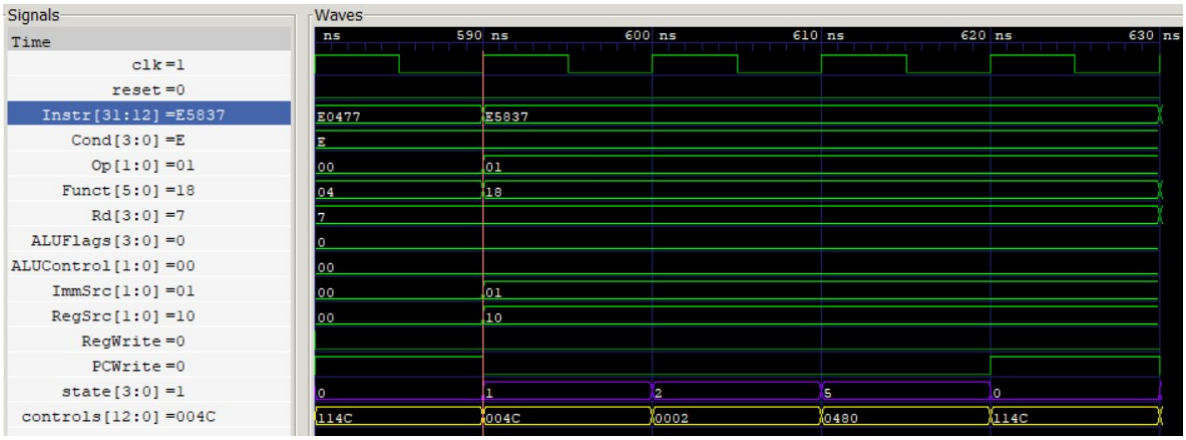


Figura 16: Simulated waveform for the instruction

4.2.17. LDR R2, [R0, #96]

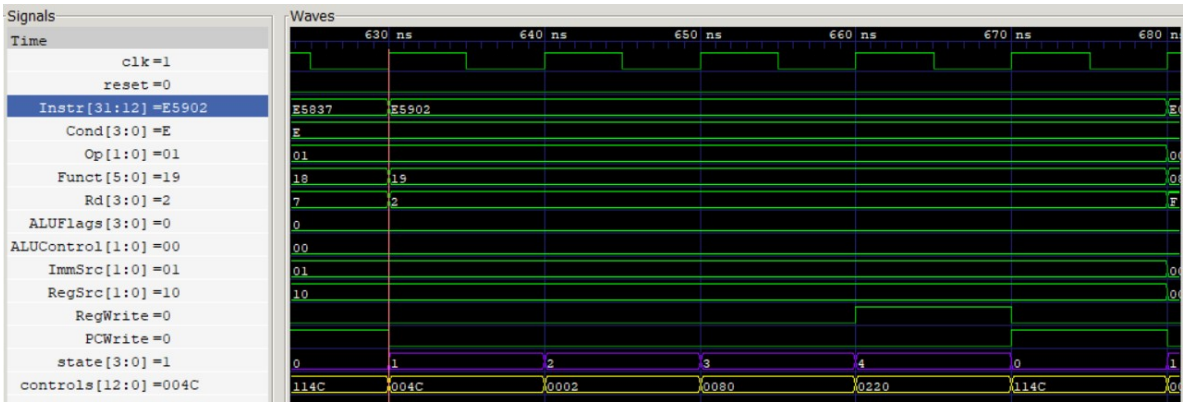


Figura 17: Simulated waveform for the instruction

4.2.18. ADD R15, R15, R0

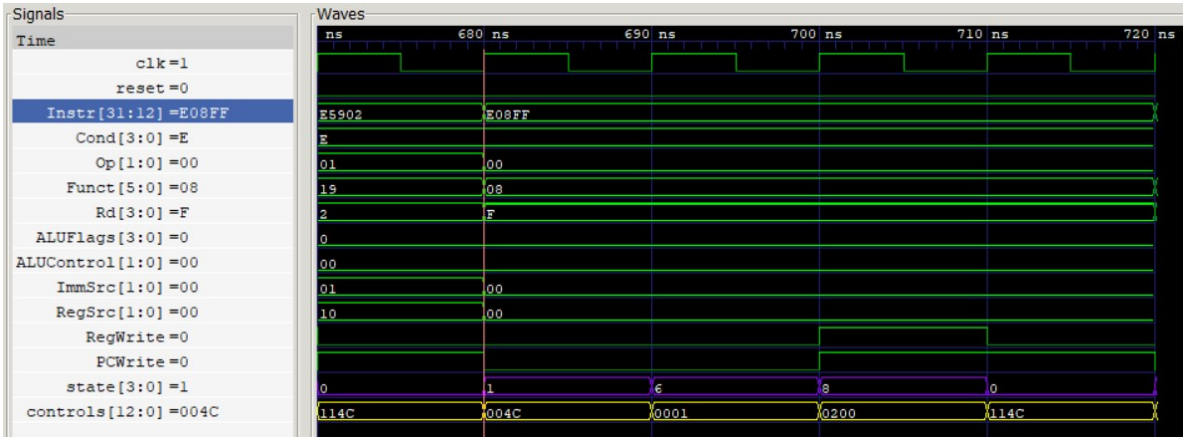


Figura 18: Simulated waveform for the instruction

4.2.19. ADD R2, R0, #14

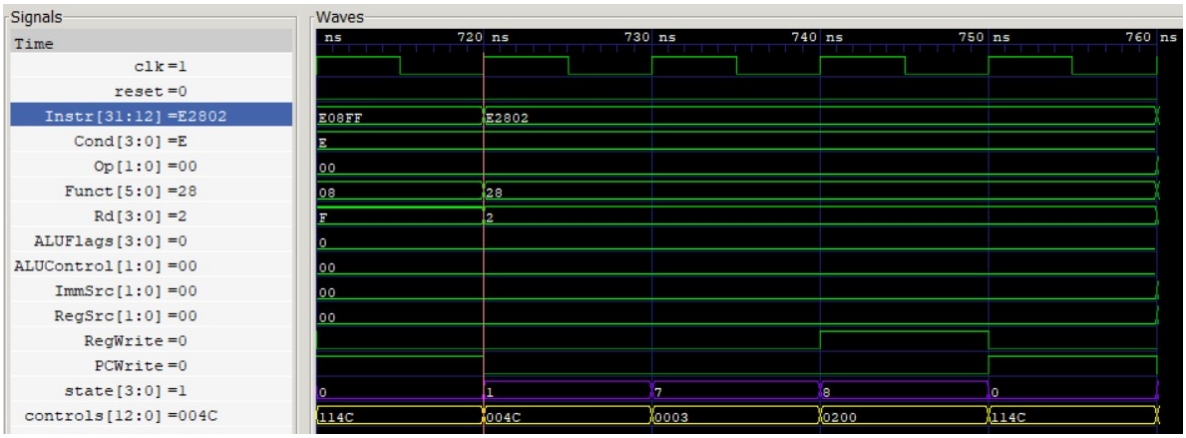


Figura 19: Simulated waveform for the instruction

4.2.20. B END

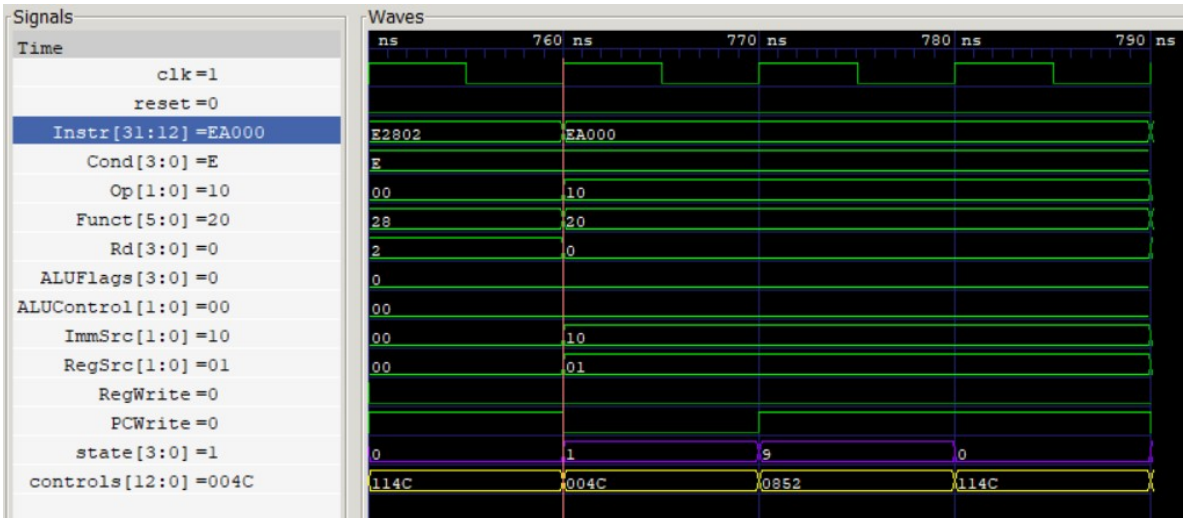


Figura 20: Simulated waveform for the instruction

4.2.21. ADD R2, R0, #13

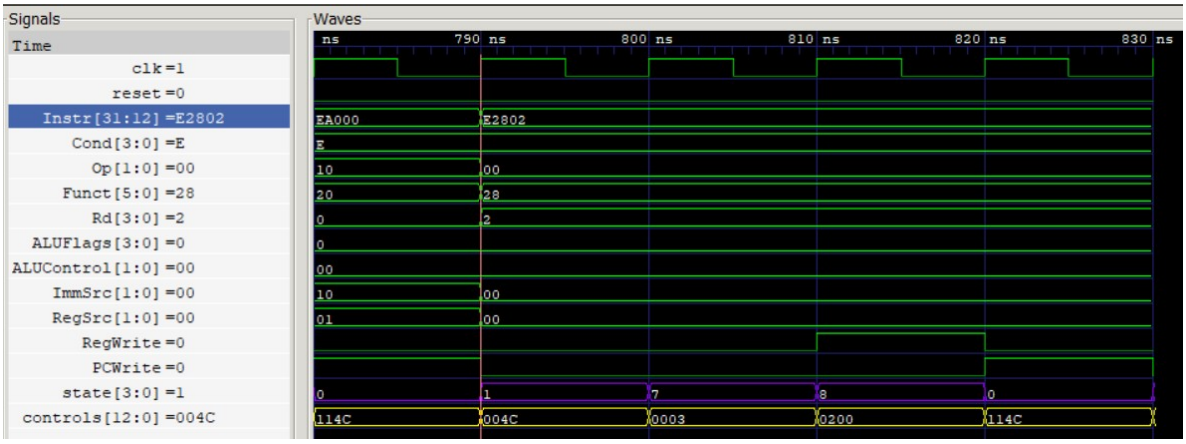


Figura 21: Simulated waveform for the instruction

4.2.22. ADD R2, R0, #10

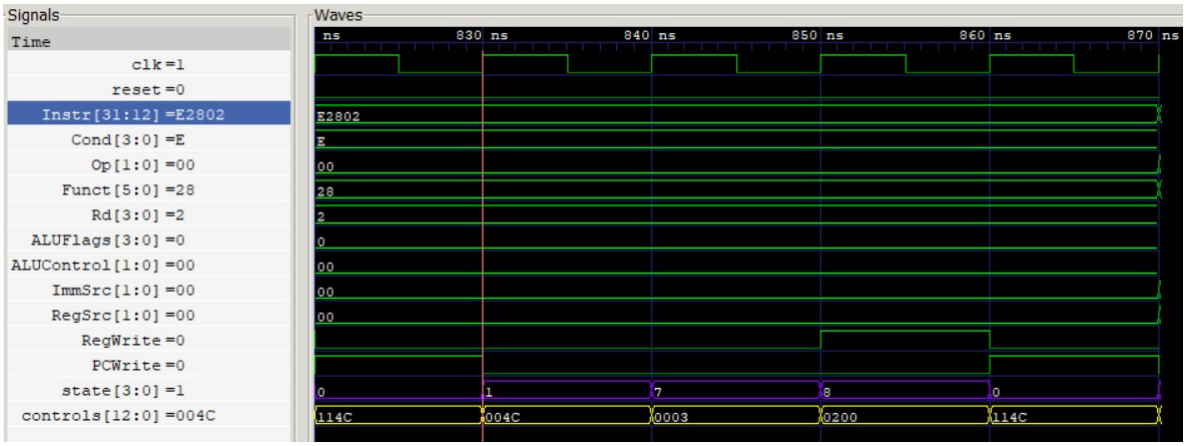


Figura 22: Simulated waveform for the instruction

4.2.23. END STR R2, [R0, #100]

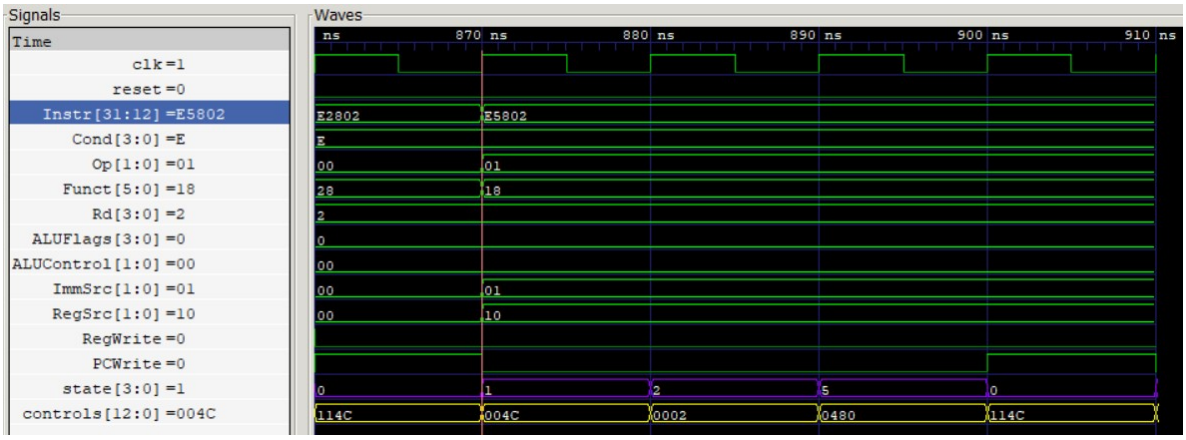


Figura 23: Simulated waveform for the instruction

5. Conclusiones

Finalmente, presentamos las siguientes conclusiones:

- Por medio de código Verilog se logró implementar un controller (unidad de control) para un multicycle processor capaz producir distintas señales para cada tipo instrucción ARM y sus respectivas etapas de ejecución.
- Se verifica el correcto funcionamiento de nuestro controller a través de un testbench que genera señales input Instr y ALUFlags.
- Las formas de onda producidas durante la simulación por cada instrucción coindicen con las suposiciones y corroboran el adecuado funcionamiento de los submódulos del controller como el Main FSM.