Solución del Laboratorio 4

Angeles Barazorda, Jean Pier (100%); Mori Ortiz, Pedro Enrique (100%); Neira Riveros, Jorge Luis (100%)

I. Ejercicio 1: Testeo del procesador ARM single-cycle

Probaremos el procesador ARM single-cycle con el programa de ejemplo del libro. Este procesador está implementado con el código en Verilog base dado para este laboratorio junto a nuestra implementación del ALU. Esta versión del procesador reconoce las siguientes instrucciones: ADD, SUB, AND, ORR, LDR, STR y B. Las instrucciones del programa están en el archivo $memfile.asm$ en hexadecimal.

A. Programa de testeo 1

El programa a testear es el siguiente:

```
.global _start
_start:
MAIN: SUB R0, R15, R15     // R0 = 0
      ADD R2, R0, #5       // R2 = 5
      ADD R3, R0, #12      // R3 = 12
      SUB R7, R3, #9       // R7 = 3
      ORR R4, R7, R2       // R4 = 3 OR 5 = 7
      AND R5, R3, R4       // R5 = 12 AND 7 = 4
      ADD R5, R5, R4       // R5 = 4 + 7 = 11
      SUBS R8, R5, R7      // R8 = 11 - 3 = 8, set flags
      BEQ END              //; shouldn't be taken
      SUBS R8, R3, R4      // R8 = 12 - 7 = 5
      BGE AROUND           // should be taken
      ADD R5, R0, #0       // should be skipped
AROUND: SUBS R8, R7, R2    // R8 = 3 - 5 = -2, set flags
      ADDLT R7, R5, #1     // R7 = 11 + 1 = 12
      SUB R7, R7, R2       // R7 = 12 - 5 = 7
      STR R7, [R3, #84]    // mem[12+84] = 7
      LDR R2, [R0, #96]    // R2 = mem[96] = 7
      ADD R15, R15, R0     // PC = PC+8 (skips next)
      ADD R2, R0, #14      // shouldn't happen
      B END                // always taken
      ADD R2, R0, #13      // shouldn't happen
      ADD R2, R0, #10      // shouldn't happen
END: STR R2, [R0, #100]    // mem[100] = 7
```

B. Señales del procesador para el programa 1

La ejecución de las instrucciones se realiza ciclo por ciclo. Al final del laboratorio mostramos la tabla de resultados esperados. A continuación presentamos los resultados de la simulación con GTKWave. Las señales mostradas corresponden a: clk, $reset$, PC, $Instr$, $ALUResult$, $WriteData$, $MemWrite$ y $ReadData$.

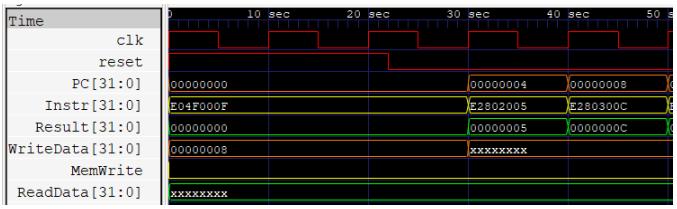

Fig 1.1 Ciclos 1 - 3

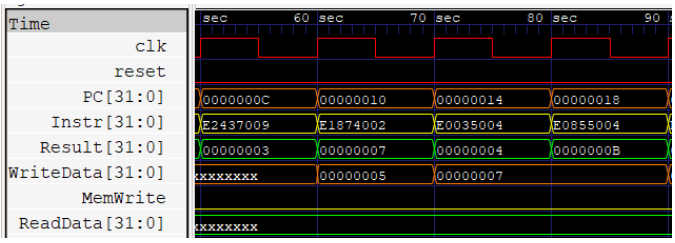

Fig 1.2 Ciclos 4 - 7

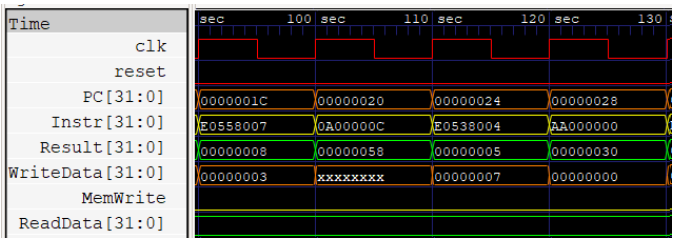

Fig 1.3 Ciclos 8 - 11

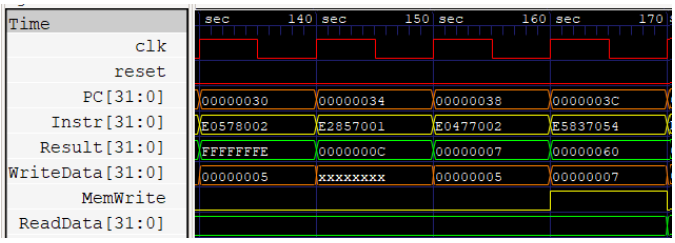

Fig 1.4 Ciclos 12 - 15

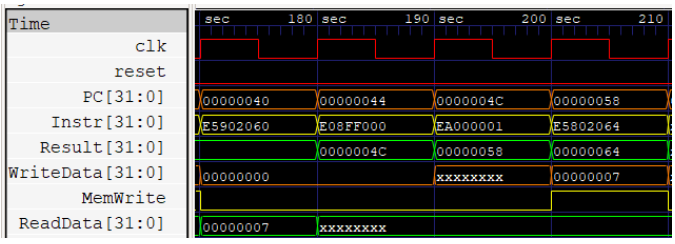

Fig 1.5 Ciclos 16 - 20

1) Final de simulación: El programa termina exitosamente con las condiciones del testbench.v.

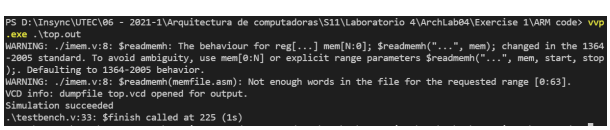

Fig 1.6 Fin de ejecución.

II. Ejercicio 2: Modificación del procesador ARM single-cycle

Realizaremos modificaciones en el código Verilog del procesador para que reconozca adicionalmente a las instrucciones EOR y $LDRB$. Las instrucciones del programa de testeo 2 están en el archivo $memfile2.asm$ en hexadecimal.

A. Modificaciones realizadas en el procesador ARM single cycle

1) Modificaciones para el procesamiento de EOR:

- Debido a que nuestro ALU realiza las operaciones ADD, SUB, AND y ORR, la señal $ALUControl$ cuenta con 2 bits. Si queremos agregar una nueva operación, entonces debemos añadir un bit a nuestro $ALUControl$. Esta modificacion afecta también a los bloques de control del procesador, específicamente al $ALUDecoder$.

TABLE I
DECODER MODIFICADO

$ALUOp$	$Funct_{4:1}$ (cmd)	$Funct_0$ (S)	$Type$	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	000	00
1	0100	0	ADD	000	00
1	0100	1	ADD	000	11
1	0010	0	SUB	001	00
1	0010	1	SUB	001	11
1	0000	0	AND	010	00
1	0000	1	AND	010	10
1	1100	0	ORR	011	00
1	1100	1	ORR	011	10
1	0001	0	EOR	110	00
1	0001	1	EOR	110	10

- Note que el $ALUControl$ para el EOR es 110, mantenemos el 2do bit en 1 para que tenga relación con los otros dos operadores lógicos. Código modificado del decode:

```
always @(*)
    if (ALUOp) begin
        case (Funct[4:1])
            4'b0100: ALUControl = 3'b000;
            4'b0010: ALUControl = 3'b001;
            4'b0000: ALUControl = 3'b010;
            4'b1100: ALUControl = 3'b011;
            4'b0001: ALUControl = 3'b110;
            default: ALUControl = 3'bxxx;
        endcase
        FlagW[1] = Funct[0];
        FlagW[0] = Funct[0] & ((ALUControl == 3'b000) | (ALUControl == 3'b001));
    end
    else begin
        ALUControl = 3'b000;
        FlagW = 2'b00;
    end
```

Fig 2.1 Modificación en decode.v

- También, modificamos el ALU para que realice la operación EOR. Como es representado por 110 entonces no afectará ni al $Carry$ ni al $Overflow$.

```
always @(*)
  begin
    casex (ALUControl[2:0])
      3'b00?: Result=sum;
      3'b010: Result=a&b;
      3'b011: Result=a|b;
      3'b110: Result=a^b;
    endcase
  end
```

Fig 2.2 Modificación en alu.v

- Por último, modificamos cada llamada a AluControl en todo el procesador, y la cambiamos de 2 a 3 bits.

2) Modificaciones para el procesamiento de LDRB:

- Añadimos un *logic block* propio llamado $ByteController$, cuyos inputs son ExtImm y Result; y como output, $ByteResult$, tiene al byte requerido. Internamente, este bloque extrae el byte especificado en el ExtImm y realiza un *zero-extend* de 24 bits.

- Luego de este procedimiento, colocamos un mux 2:1 para poder elegir entre Result, el cual sería el mismo que para una instrucción cualquiera, y el byte extendido a 32 bits.

- Además, necesitamos de otro mux 2:1 en la entrada del $SrcB$ del ALU, ya que, en caso de realizar un $LDRB$, solo debemos considerar a la dirección del $RD2$ del $RegFile$, ya que el $immediate$ procesará el resultado en $ByteController$. El mux escogerá entre la entrada normal a $SrcB$ y 0.

- La señal de control para ambos mux proviene de una nueva salida que agregamos al Control Unit, específicamente en el $Decoder$. Allí colocamos una nueva validación del Funct[2] (B o bit 22 en la instrucción original), el cual nos indica si la operación tomará solo un byte ($LDRB$) o todo el word (LDR). A continuación, se muestra el nuevo Schematic del *Single-Cycle processor* para procesar esta instrucción:

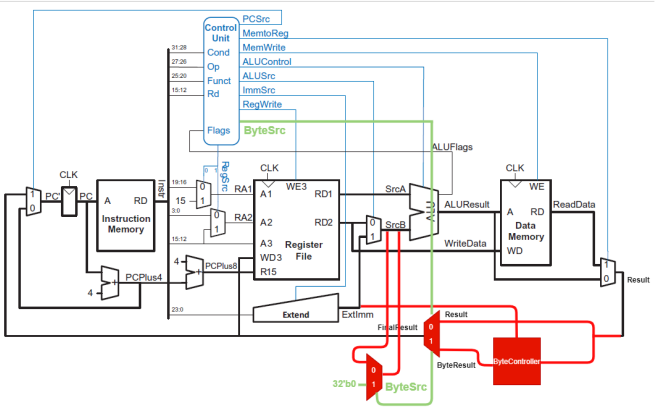

Fig 2.3 Nuevo Schematic del Single-Cycle processor

- Por último, agregamos estos módulos al código en Verilog y cambiamos el testbench, el cual será exitoso para el programa 2 cuando $DataAdr$ es 128 y $WriteData$, $32'hFE$.

```
S11 > Laboratorio 4 > ArchLab04 > Exercise 2 > ARM code > byteController.v > byteController
module byteController(
    Result,
    BytePosition,
    ByteResult,
    enable
);
    input wire [31:0] Result;
    input wire [31:0] BytePosition;
    input wire enable;
    output reg [31:0] ByteResult;

    always @(*)
        if (enable)
        case (BytePosition[1:0])
            2'b00: ByteResult = {24'b000000000000000000000000, Result[7:0]};
            2'b01: ByteResult = {24'b000000000000000000000000, Result[15:8]};
            2'b10: ByteResult = {24'b000000000000000000000000, Result[23:16]};
            2'b11: ByteResult = {24'b000000000000000000000000, Result[31:24]};
            default: ByteResult = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
        endcase
endmodule
```

Fig 2.4 Módulo agregado.

```
125     mux2 #(32) byteMuxAlu(
126         .d0(SrcB),
127         .d1(32'b00000000000000000000000000000000),
128         .s(ByteSrc),
129         .y(SrcBB)
130     );
131     byteController byteC(
132         .Result(Result),
133         .BytePosition(ExtImm),
134         .ByteResult(ByteResult),
135         .enable(ByteSrc)
136     );
137     mux2 #(32) byteMuxResult(
138         .d0(Result),
139         .d1(ByteResult),
140         .s(ByteSrc),
141         .y(FinalResult)
142     );
143 endmodule
```

Fig 2.5 Modificación en datapath.

```
32  always @(*)
33      casex (Op)
34          2'b00:
35              if (Funct[5])
36                  controls = 11'b00001010010;
37              else
38                  controls = 11'b00000010010;
39          2'b01:
40              if (Funct[0])
41                  begin
42                      if (Funct[2])
43                          controls = 11'b00011110001;
44                      else
45                          controls = 11'b00011110000;
46                  end
47              else
48                  controls = 11'b10011101000;
49          2'b10: controls = 11'b0110100100;
50          default: controls = 11'bxxxxxxxxxxx;
51      endcase
52  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg, RegW, MemW, Branch, ALUOp, ByteSrc} = controls;
53  always @(*)
```

Fig 2.6 Modificación en decode

```
29      always @(negedge clk)
30          if (MemWrite)
31              if ((DataAdr === 128) & (WriteData === 32'hFE)) begin
32                  $display("Simulation succeeded");
33                  #10;
34                  $finish;
35              end
36              else if ((DataAdr === 128) & (WriteData !== 32'hFE)) begin
37                  $display("Simulation failed");
38                  #10;
39                  $finish;
40              end
41      initial begin
42          $dumpfile("top.vcd");
43          $dumpvars;
44      end
```

Fig 2.7 Modificación en testbench

B. Programa de testeo 2

El programa a testear es el siguiente:

```
.global _start
_start:
MAIN: SUB R0, R15, R15    // R0 = 0
      ADD R1, R0, #255    // R1 = 0 + 255 = 255
      ADD R2, R1, R1      // R2 = 255 + 255 = 510
      STR R2, [R0, #196]  // mem[0 + 196] = 510
      EOR R3, R1, #77     // R3 = 255 ^ 77 = 178
      AND R4, R3, #0x1F   // R4 = 178 & 31 = 18
      ADD R5, R3, R4      // R5 = 178 + 18 = 196
      LDRB R6, [R5]       // R6 = mem[196] (1 byte) = 254
      LDRB R7, [R5, #1]   // R7 = mem[197] (1 byte) = 1
      SUBS R0, R6, R7     // R0 = 254 - 1 = 253, set flags
      BLT MAIN            // shouldn't be taken
      BGT HERE            // should be taken
      STR R1, [R4, #110]  // mem[18 + 110] = 255
      B MAIN              // shouldn't happen
HERE: STR R6, [R4, #110]  // mem[18 + 110] = 254
```

C. Señales del procesador para el programa 2

La ejecución de las instrucciones se realiza ciclo por ciclo. Al final del laboratorio mostramos la tabla de resultados esperados. A continuación presentamos los resultados de la simulación con GTKWave. Las señales mostradas corresponden a: clk, $reset$, PC, $Instr$, $ALUResult$, $WriteData$, $MemWrite$ y $ReadData$.

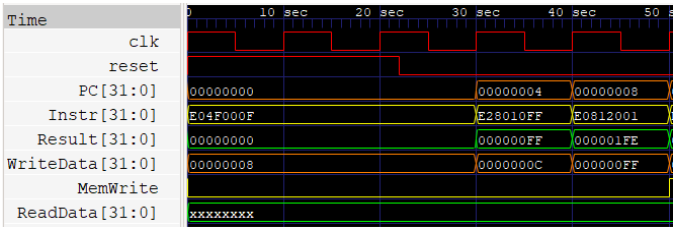

Fig 2.8 Ciclos 1 - 3

Fig 2.9 Ciclos 4 - 7

Fig 2.10 Ciclos 8 - 11

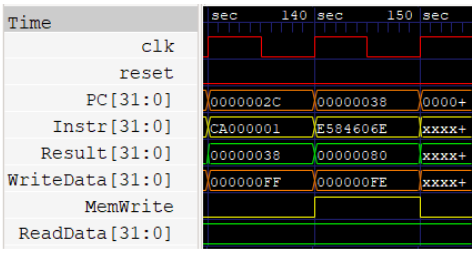

Fig 2.11 Ciclos 12 - 13

III. Tabla del programa 1 ’

TABLE II
EJECUCIÓN DEL PROGRAMA 1

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	NZCV	CondEx	WriteData	MemWrite	ReadData
1	1	00	SUB R0, R15, R15 E04F000F	8	8	0	0	0100	1	8	0	X
2	0	04	ADD R2, R0, #5 E2802005	0	5	0	5	0000	1	X	0	X
3	0	08	ADD R3, R0, #12 E280300C	0	C	0	C	0000	1	X	0	X
4	0	0C	SUB R7, R3, #9 E2437009	C	9	0	3	0000	1	X	0	X
5	0	10	ORR R4, R7, R2 E1874002	3	5	0	7	0000	1	5	0	X
6	0	14	AND R5, R3, R4 E0035004	C	7	0	4	0000	1	7	0	X
7	0	18	ADD R5, R5, R4 E0855004	4	7	0	B	0000	1	7	0	X
8	0	1C	SUBS R8, R5, R7 E0558007	B	3	0	8	0000	1	3	0	X
9	0	20	BEQ END 0A00000C	28	30	1	58	0000	0	X	0	X
10	0	24	SUBS R8, R3, R4 E0538004	C	7	0	5	0000	1	7	0	X
11	0	28	BGE AROUND AA000000	30	0	1	30	0000	1	0	0	X
12	0	30	SUBS R8, R7, R2 E0578002	3	5	0	-2	1011	1	5	0	X
13	0	34	ADDLT R7, R5, #1 B2857001	B	1	0	C	0000	1	X	0	X
14	0	38	SUB R7, R7, R2 E0477002	C	5	0	7	0000	1	5	0	X
15	0	3C	STR R7, [R3, #84] E5837054	C	54	0	60	0000	1	7	1	X
16	0	40	LDR R2, [R0, #96] E5902060	0	60	0	60	0000	1	0	0	7
17	0	44	ADD R15, R15, R0 E08FF000	4C	0	0	4C	0000	1	0	0	X
18	0	4C	B END EA000001	0	64	1	64	0000	1	X	0	X
19	0	58	STR R2, [R0, #100] E5802064	0	64	0	64	0000	1	7	1	X

IV. Tabla del programa 2

TABLE III
EJECUCIÓN DEL PROGRAMA 2

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	NZCV	CondEx	WriteData	MemWrite	ReadData
1	1	00	SUB R0, R15, R15 E04F000F	8	8	0	0	0100	1	8	0	X
2	0	04	ADD R1, R0, #255 E28010FF	0	FF	0	FF	0000	1	C	0	X
3	0	08	ADD R2, R1, R1 E0812001	FF	FF	0	1FE	0000	1	FF	0	X
4	0	0C	STR R2, [R0, #196] E58020C4	0	C4	0	C4	0000	1	1FE	1	X
5	0	10	EOR R3, R1, #77 E221304D	FF	4D	0	B2	0000	1	X	0	X
6	0	14	AND R4, R3, #0x1F E203401F	B2	1F	0	12	0000	1	1C	0	X
7	0	18	ADD R5, R3, R4 E0835004	B2	12	0	C4	0000	1	12	0	X
8	0	1C	LDRB R6, [R5] E5D56000	C4	0	0	C4	0000	1	0	0	FE
9	0	20	LDRB R7, [R5, #1] E5D57001	C4	1	0	C5	0000	1	FF	0	1
10	0	24	SUBS R0, R6, R7 E0560007	FE	1	0	FD	0010	1	1FE	0	X
11	0	28	BLT MAIN BAFFFFF4	30	-48	0	0	0010	0	12	0	X
12	0	2C	BGT HERE CA000001	34	4	1	38	0010	1	FF	0	X
13	0	38	STR R6, [R4, #110] E584606E	12	6E	0	80	0010	1	FE	1	X