پروژه پایانی درس VLSI2 (فاز اول)

عنوان:

Multicycle ARM Processor Control Unit

استاد درس:

دکتر شالچیان

پدیدآورنده:

سارا قائمی 9223089

17 تیر 96

1. فایل های ارائه شده

1.1. فایل گزارش انجام پروژه

این فایل شامل موارد مقابل است. میزان زمانی که برای این پروژه صرف شده است، جدول Main FSM Output که کامل شده است، کد arm_multi که بخش هایی که به آن اضافه شده است Highlight شده است، کد controllertest و توضیحات مربوط به آن، تصاویر خروجی مدار که به درستی کار میکند.

2.1. فایل arm_multi_9223089.sv

این فایل در واقع کد اصلی برنامه میباشد که بخش های لازم تکمیل شده است. در ادامه به توضیح بیشتر این کد خواهیم پرداخت.

3.1. فایل controllertest_9223089.sv

این فایل testbench بخش کنترلر است که در ادامه به تفصیل توضیح داده خواهد شد.

4.1. فایل vsim.wlf

این فایل خروجی modelsim را شامل میشود.

2. زمان صرف شده

کد اصلی برنامه حدود 1 ساعت، کد تست بنچ حدود 30 دقیقه، دیباگ حدود 2 ساعت و گزارش نیز حدود 3 ساعت زمان برد.

مجموعا من حدود 6.5 ساعت روی این بخش از پروژه زمان گذاشتم.

3. روند انجام پروژه

من این پروژه را دقیقا به ترتیبی که در ادامه می آید انجام دادم.

1.3. تکمیل جدول Main FSM Output

ابتدا جدول زیر تکمیل گردید. برای این منظور از state transition diagram که در اختیار قرار داده شده بود استفاده شد. برای هر کدام از state ها المان های خواسته شده با توجه به آنچه در دیاگرام آمده بود مقادیر بدست آمد.

State (Name)	NextPC	Branch	MemW	RegW	IRWrite	AdrSrc	$ResultSrc_{1:0}$	$ALUSrcA_{1:0}$		$ALUSrcB_{1:0}$	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

جدول شماره 1: Main FSM Output

2.3. تکمیل کد بخش mainfsm

در ادامه با توجه به دیاگرام حالت داده شده next state logic و بخش state-dependent output logic نیز با توجه به جدول فوق تکمیل گردید. کد تکمیل شده بخش mainfsm به صورت زیر میباشد. خط های highlight شده کد های اضافه شده توسط من میباشند.

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

  typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMRD, MEMWB,
                            MEMWR, EXECUTER, EXECUTEI, ALUWB, BRANCH,
                     UNKNOWN} statetype;

  statetype state, nextstate;
  logic [12:0] controls;

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

  // next state logic
  always_comb
    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;
              else              nextstate = MEMWR;
      MEMRD:                    nextstate = MEMWB;
      default:                  nextstate = FETCH;
    endcase

  // state-dependent output logic
  always_comb
    case(state)
      FETCH:    controls = 13'b10001_010_01100;
      DECODE:   controls = 13'b00000_010_01100;
      EXECUTER: controls = 13'b00000_000_00001;
      EXECUTEI: controls = 13'b00000_000_00011;
      ALUWB:    controls = 13'b00010_000_00000;
      MEMADR:   controls = 13'b00000_000_00010;
      MEMWR:    controls = 13'b00100_100_00000;
      MEMRD:    controls = 13'b00000_100_00000;
      MEMWB:    controls = 13'b00010_001_00000;
      BRANCH:   controls = 13'b01000_010_10010;
      default:  controls = 13'bxxxxx_xxx_xxxxx;
    endcase

  assign {NextPC, Branch, MemW, RegW, IRWrite,
          AdrSrc, ResultSrc,
          ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule
```

3.3. تکمیل ادامه ی بخش decode

بخش decode شامل ALU Decoder، PC Logic و Instruction Decoder میباشد. برای دو بخش اول از کد کارگاه آخر یعنی single cycle processor استفاده شد چون این دو بخش در پروسسور های تک سیکل و چند سیکل مشابه به هم میباشند. برای بخش Instruction Decoder کد ImmSrc قبلا نوشته شده بود. RegSrc نیز با توجه به اینکه دستور پردازش داده، دستور حافظه ای و یا branch است به طوری انتخاب میگردد که مولتی پلکسر ها بخش های درستی از Instr را انتخاب کنند. در ادامه کد کامل decode را مشاهده میکنید که در آن بخش هایی که توسط من اضافه شده است highlight شده است.

```
module decode(input  logic       clk, reset,
              input  logic [1:0] Op,
              input  logic [5:0] Funct,
              input  logic [3:0] Rd,
              output logic [1:0] FlagW,
              output logic       PCS, NextPC, RegW, MemW,
              output logic       IRWrite, AdrSrc,
```

```
                output logic [1:0] ResultSrc, ALUSrcA, ALUSrcB,
                output logic [1:0] ImmSrc, RegSrc, ALUControl);

  logic       Branch, ALUOp;

  // Main FSM
  mainfsm fsm(clk, reset, Op, Funct,
              IRWrite, AdrSrc,
              ALUSrcA, ALUSrcB, ResultSrc,
              NextPC, RegW, MemW, Branch, ALUOp);

  // ALU Decoder
  always_comb
    if (ALUOp) begin                 // which DP Instr?
      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'bx;  // unimplemented
      endcase
      // update flags if S bit is set
      // (C & V only updated for arith instructions)
      FlagW[1] = Funct[0]; // FlagW[1] = S-bit
      // FlagW[0] = S-bit & (ADD | SUB)
      FlagW[0] = Funct[0] & (ALUControl == 2'b00 | ALUControl == 2'b01);
    end else begin
      ALUControl = 2'b00; // add for non-DP instructions
      FlagW      = 2'b00; // don't update Flags
    end

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

  // Instr Decoder
  assign ImmSrc    = Op;
  always_comb
      case(Op)
        2'b00:   RegSrc = 2'b00; //Data-processing
        2'b01:   RegSrc = 2'b10; //Memory-Instr
        2'b10:   RegSrc = 2'b01; // Branch
        default: RegSrc = 2'bx;  // unimplemented
      endcase
endmodule
```

4.3. تکمیل کد بخش condcheck

سپس بخش condcheck که بخشی از condcheck محسوب میگردد تکمیل گردید. این بخش نیز عینا در پروسسور تک سیکل وجود داشت. به همین دلیل از همان کد کارگاه آخر برای این بخش استفاده گردید. در ادامه کد condcheck را مشاهده میکنید که در آن بخش هایی که توسط من اضافه شده است highlight شده است.

```
module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;
```

```
  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase
endmodule
```

5.3. تکمیل دیگر بخش های condlogic

برای تکمیل این بخش به دو نوع flip flop نیاز داریم. یک نوع flipflop که enable داشته باشد که دو عدد از آن برای تولید 4 بیت Flags نیاز داریم. نوع دیگر flipflop بدون enable که در واقع برای ایجاد تاخیر در CondEx استفاده میگردد. به این منظور کد این دو flipflop را که در کارگاه آخر نیز از آن ها استفاده شد به کد اصلی اضافه میکنیم. در ادامه کد این دو رو میبینید.

```
module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

سپس از این ماژول ها استفاده کرده و کد را تکمیل میکنیم.

در این بخش یک خط کد به صورت زیر در برنامه قرار داده شده است.

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

ولی من نتوانستم به درستی این خط کد را درک کنم و به درستی از آن استفاده کنم. به همین دلیل این خط را پاک کرده و به صورت زیر پیاده سازی نموده ام که به درستی جواب میدهد.

در ادامه کد کامل مربوط به بخش condlogic را میبینید که بخش های اضافه شده توسط من highlight شده اند.

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

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx, CondExDelayed;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  condcheck cc(Cond, Flags, CondEx);
  flopr #(1)condreg(clk, reset, CondEx, CondExDelayed);

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

endmodule
```

6.3. ساماندهی کلی کد

در این بخش کل کد بخش controller را مشاهده میکنید که بخش های highlight شده در آن توسط من نوشته شده است.

```
module controller(input  logic         clk,
                  input  logic         reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic         PCWrite,
                  output logic         MemWrite,
                  output logic         RegWrite,
                  output logic         IRWrite,
                  output logic         AdrSrc,
                  output logic [1:0]   RegSrc,
                  output logic [1:0]   ALUSrcA,
                  output logic [1:0]   ALUSrcB,
                  output logic [1:0]   ResultSrc,
                  output logic [1:0]   ImmSrc,
```

```
                output logic [1:0]   ALUControl);

  logic [1:0] FlagW;
  logic       PCS, NextPC, RegW, MemW;

  decode dec(clk, reset, Instr[27:26], Instr[25:20], Instr[15:12],
             FlagW, PCS, NextPC, RegW, MemW,
             IRWrite, AdrSrc, ResultSrc,
             ALUSrcA, ALUSrcB, ImmSrc, RegSrc, ALUControl);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, NextPC, RegW, MemW,
               PCWrite, RegWrite, MemWrite);
endmodule

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

  logic       Branch, ALUOp;

  // Main FSM
  mainfsm fsm(clk, reset, Op, Funct,
              IRWrite, AdrSrc,
              ALUSrcA, ALUSrcB, ResultSrc,
              NextPC, RegW, MemW, Branch, ALUOp);

  // ALU Decoder
  always_comb
    if (ALUOp) begin                 // which DP Instr?
      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'bx;  // unimplemented
      endcase
      // update flags if S bit is set
      // (C & V only updated for arith instructions)
      FlagW[1] = Funct[0]; // FlagW[1] = S-bit
      // FlagW[0] = S-bit & (ADD | SUB)
      FlagW[0] = Funct[0] & (ALUControl == 2'b00 | ALUControl == 2'b01);
    end else begin
      ALUControl = 2'b00; // add for non-DP instructions
      FlagW      = 2'b00; // don't update Flags
    end

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

  // Instr Decoder
  assign ImmSrc    = Op;
```

```
  always_comb
      case(Op)
        2'b00:   RegSrc = 2'b00; //Data-processing
        2'b01:   RegSrc = 2'b10; //Memory-Instr
        2'b10:   RegSrc = 2'b01; // Branch
        default: RegSrc = 2'bx;  // unimplemented
      endcase
endmodule

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

  typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMRD, MEMWB,
                            MEMWR, EXECUTER, EXECUTEI, ALUWB, BRANCH,
                       UNKNOWN} statetype;

  statetype state, nextstate;
  logic [12:0] controls;

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

  // next state logic
  always_comb
    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;
              else              nextstate = MEMWR;
      MEMRD:                    nextstate = MEMWB;
      default:                  nextstate = FETCH;
    endcase

  // state-dependent output logic
  always_comb
    case(state)
      FETCH:    controls = 13'b10001_010_01100;
      DECODE:   controls = 13'b00000_010_01100;
      EXECUTER: controls = 13'b00000_000_00001;
      EXECUTEI: controls = 13'b00000_000_00011;
```

```
      ALUWB:    controls = 13'b00010_000_00000;
      MEMADR:   controls = 13'b00000_000_00010;
      MEMWR:    controls = 13'b00100_100_00000;
      MEMRD:    controls = 13'b00000_100_00000;
      MEMWB:    controls = 13'b00010_001_00000;
      BRANCH:   controls = 13'b01000_010_10010;
      default:  controls = 13'bxxxxx_xxx_xxxxx;
    endcase

  assign {NextPC, Branch, MemW, RegW, IRWrite,
          AdrSrc, ResultSrc,
          ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule

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

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx, CondExDelayed;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  condcheck cc(Cond, Flags, CondEx);
  flopr #(1)condreg(clk, reset, CondEx, CondExDelayed);

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

endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
```

```
      4'b0110: CondEx = overflow;          // VS
      4'b0111: CondEx = ~overflow;         // VC
      4'b1000: CondEx = carry & ~zero;     // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;                // GE
      4'b1011: CondEx = ~ge;               // LT
      4'b1100: CondEx = ~zero & ge;        // GT
      4'b1101: CondEx = ~(~zero & ge);     // LE
      4'b1110: CondEx = 1'b1;              // Always
      default: CondEx = 1'bx;              // undefined
    endcase
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset)   q <= 0;
    else if (en) q <= d;
endmodule

module flopr #(parameter WIDTH = 8)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] d,
               output logic [WIDTH-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= d;
endmodule
```

7.3. نوشتن کد testbench

برای اینکه کد اصلی برنامه به درستی و به طور کامل تست شود تصمیم گرفتم کد های زیر را به ترتیب به کنترلر بدهم.

ADD R1, R2, #2, ALUFlags = 4'b0000

SUB R4, R3, R5, ALUFlags = 4'b0000

ANDS R7, R2, R3, ALUFlags = 4'b0000

ORNE R2, R1, #5, ALUFlags = 4'b0000

LDR R3, [R1, #10], ALUFlags = 4'b0000

STR R5, [R2, #2], ALUFlags = 4'b0000

B, ALUFlags = 4'b0000

BNE, ALUFlags = 4'b0100

BEQ, ALUFlags = 4'b0100

سپس مقدار Instr[31:12] که وارد کنترلر میشود را برای هر کدام از خطوط بالا بدست آورده و در testbench اضافه نمودم.

همچنین برای هر کدام از دستورات به توجه به اینکه چند سیکل طول میکشند delay مناسب خود قرار داده شد. این delay با توجه به clk اصلی داده شده است که من clk را با پریود 10 در نظر گرفتم.

در ادامه کد تست بنچ را مشاهده میکنید.

```
`timescale 1ns / 1ps
module testbench();

  logic         clk;
  logic         reset;
  logic [31:12] Instr;
  logic [3:0]   ALUFlags;

  logic         PCWrite;
  logic         MemWrite;
  logic         RegWrite;
  logic         IRWrite;
  logic         AdrSrc;
  logic [1:0]   RegSrc;
  logic [1:0]   ALUSrcA;
  logic [1:0]   ALUSrcB;
  logic [1:0]   ResultSrc;
  logic [1:0]   ImmSrc;
  logic [1:0]   ALUControl;


  // instantiate device to be tested
  controller dut(clk, reset, Instr, ALUFlags, PCWrite, MemWrite, RegWrite,
IRWrite, AdrSrc,
                 RegSrc, ALUSrcA, ALUSrcB, ResultSrc, ImmSrc, ALUControl);

  // initialize test
  initial
    begin
      reset <= 1; # 15; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  //Generate test signals
  initial
    begin
      #10
      Instr = 20'b1110_00_1_0100_0_0010_0001; //ADD R1, R2, #2
      ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
      #40;
      Instr = 20'b1110_00_0_0010_0_0011_0100; //SUB R4, R3, R5
      ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
      #40;
```

```
        Instr = 20'b1110_00_0_0000_1_0010_0111; //ANDS R7, R2, R3
        ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
        #40;
        Instr = 20'b0001_00_1_1100_0_0001_0010; //ORNE R2, R1, #5
        ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
        #40;
        Instr = 20'b1110_01_0_1100_1_0001_0011; //LDR R3, [R1, #10]
        ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
        #50;
        Instr = 20'b1110_01_0_1100_0_0010_0101; //STR R5, [R2, #2]
        ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
        #40;
        Instr = 20'b1110_10_1_0_000000000000; //B
        ALUFlags = 4'b0000; //{neg, zero, carry, overflow} = Flags;
        #30;
        Instr = 20'b0001_10_1_0_000000000000; //BNE
        ALUFlags = 4'b0100; //{neg, zero, carry, overflow} = Flags;
        #30;
        Instr = 20'b0000_10_1_0_000000000000; //BEQ
        ALUFlags = 4'b0100; //{neg, zero, carry, overflow} = Flags;
        #30;
    end
endmodule
```

8.3. اجرای شبیه سازی و چک کردن خروجی ها

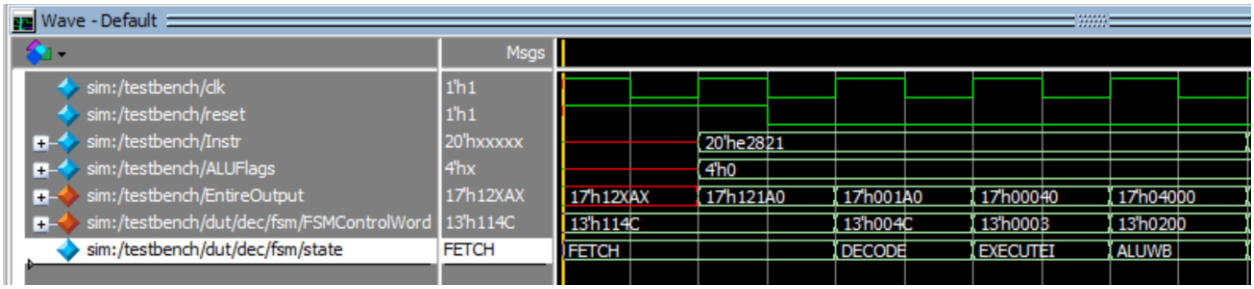
در انتها برای اینکه بتوانم به درستی خروجی برنامه خود را چک کنم جدول زیر را نوشتم که برای هر کدام از خط های کد testbench تمامی خروجی های مورد انتظار در هرکدام از state ها آورده شده است.

		Controller inputs		Controller outputs											
Test Code	State	Instr 31:12	ALUFlags 3:0	PCWritw	MemWrite	RegWrite	IRWrite	AdrSrc	RegSrc 1:0	ALUSrcA 1:0	ALUSrcB 1:0	ResultSrc 1:0	ImmSrc 1:0	ALUControl 1:0	EntireOutput
ADD R1, R2, #2	Fetch	0xE2821	0x0	1	0	0	1	0	00	01	10	10	00	00	0x121A0
	Decode			0	0	0	0	0		01	10	10		00	0x001A0
	ExecuteI			0	0	0	0	0		00	01	00		00	0x00040
	ALUWB			0	0	1	0	0		00	00	00		00	0x04000
SUB R4, R3, R5	Fetch	0xE0434	0x0	1	0	0	1	0	00	01	10	10	00	00	0x121A0
	Decode			0	0	0	0	0		01	10	10		00	0x001A0
	ExecuteR			0	0	0	0	0		00	00	00		01	0x00001
	ALUWB			0	0	1	0	0		00	00	00		00	0x04000
ANDS R7, R2, R3	Fetch	0xE0127	0x0	1	0	0	1	0	00	01	10	10	00	00	0x121A0
	Decode			0	0	0	0	0		01	10	10		00	0x001A0
	ExecuteR			0	0	0	0	0		00	00	00		10	0x00002

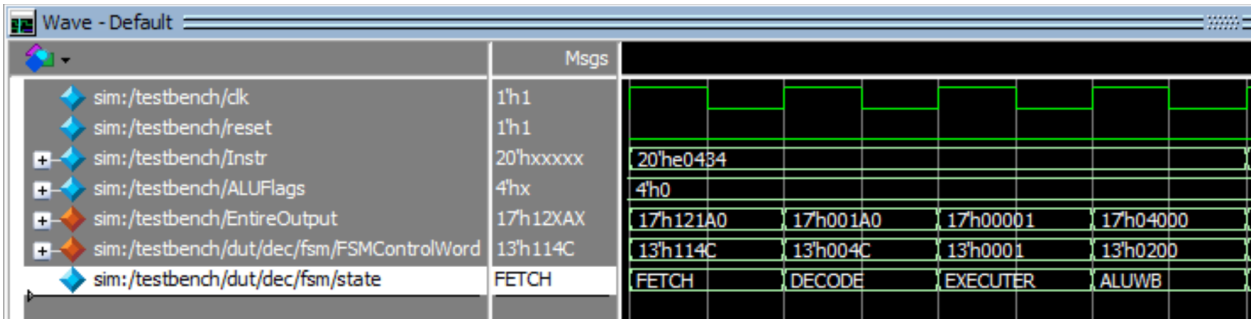
	ALUWB			0	0	1	0	0		00	00	00		00	0x04000
ORNE R2, R1, #5	Fetch	0x13812	0x0	1	0	0	1	0	00	01	10	10	00	00	0x121A0
	Decode			0	0	0	0	0		01	10	10		00	0x001A0
	ExecuteI			0	0	0	0	0		00	01	00		11	0x00043
	ALUWB			0	0	1	0	0		00	00	00		00	0x04000
LDR R3, [R1, #10]	Fetch	0xE5913	0x0	1	0	0	1	0	10	01	10	10	01	00	0x129A4
	Decode			0	0	0	0	0		01	10	10		00	0x009A4
	MemAdr			0	0	0	0	0		00	01	00		00	0x00844
	MemRD			0	0	0	0	1		00	00	00		00	0x01804
	MemWB			0	0	1	0	0		00	00	01		00	0x04814
STR R5, [R2, #2]	Fetch	0xE5825	0x0	1	0	0	1	0	10	01	10	10	01	00	0x129A4
	Decode			0	0	0	0	0		01	10	10		00	0x009A4
	MemAdr			0	0	0	0	0		00	01	00		00	0x00844
	MemWR			0	1	0	0	1		00	00	00		00	0x09804
B	Fetch	0xEA000	0x0	1	0	0	1	0	01	01	10	10	10	00	0x125A8
	Decode			0	0	0	0	0		01	10	10		00	0x005A8
	Branch			1	0	0	0	0		10	01	10		00	0x10668
BNE	Fetch	0x1A000	0x4	1	0	0	1	0	01	01	10	10	10	00	0x125A8
	Decode			0	0	0	0	0		01	10	10		00	0x005A8
	Branch			1	0	0	0	0		10	01	10		00	0x10668
BEQ	Fetch	0x0A000	0x4	1	0	0	1	0	01	01	10	10	10	00	0x125A8
	Decode			0	0	0	0	0		01	10	10		00	0x005A8
	Branch			0	0	0	0	0		10	01	10		00	0x00668

جدول شماره 2: سیگنال های کنترلر

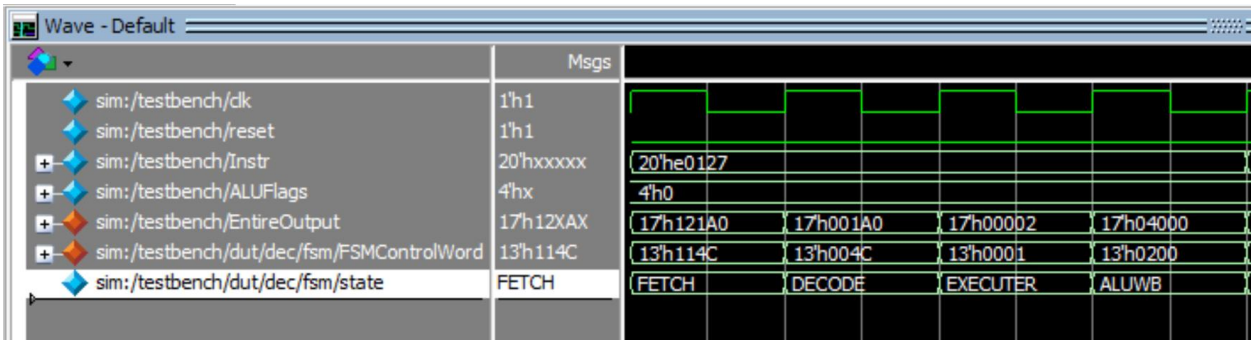
همچنین برای مقایسه ساده تر در محیط شبیه سازی modelsim سیگنال های خروجی را انتخاب کرده و با استفاده از گزینه combine signals... همگی آن ها را در یک سیگنال به نام EntireOutput نمایش دادم. همین کار را برای سیگنال های خروجی Main FSM نیز انجام داده و سیگنال FSMControlWord را تولید کردم. در ادامه تصویر خروجی تک تک خطوط کد test bench در محیط modelsim را میبینید. با چک کردن خروجی ها با جدول 2 مشاهده میشود که کد به درستی کار میکند.



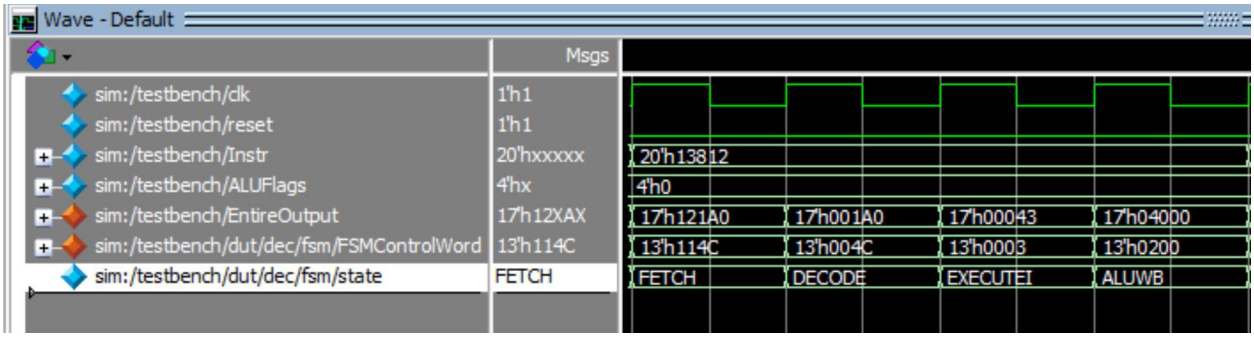
شکل شماره 1: خروجی اجرای کد ADD R1, R2, #2



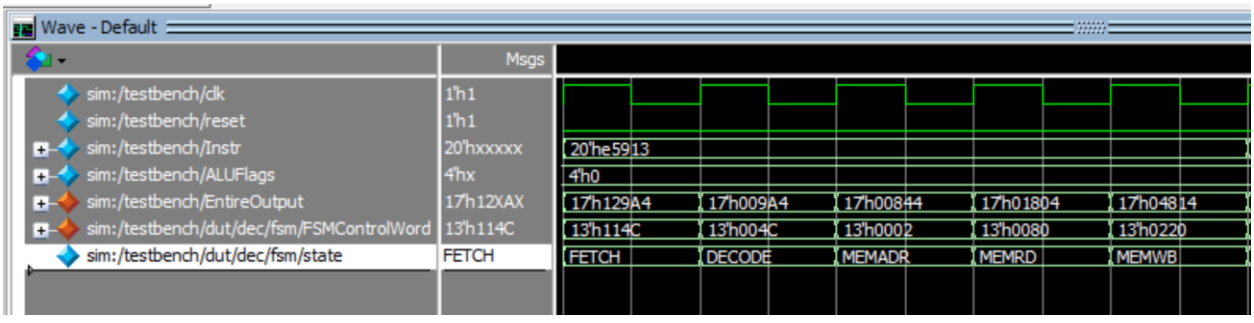
شکل شماره 2: خروجی اجرای کد SUB R4, R3, R5



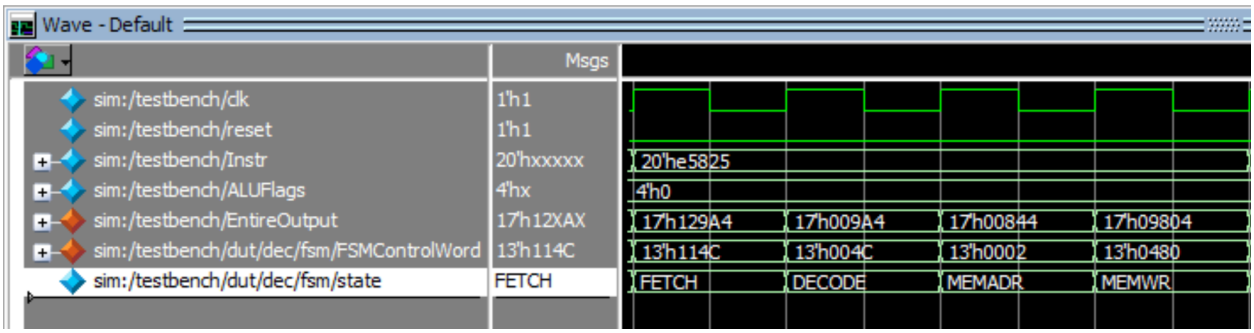
شکل شماره 3: خروجی اجرای کد ANDS R7, R2, R3



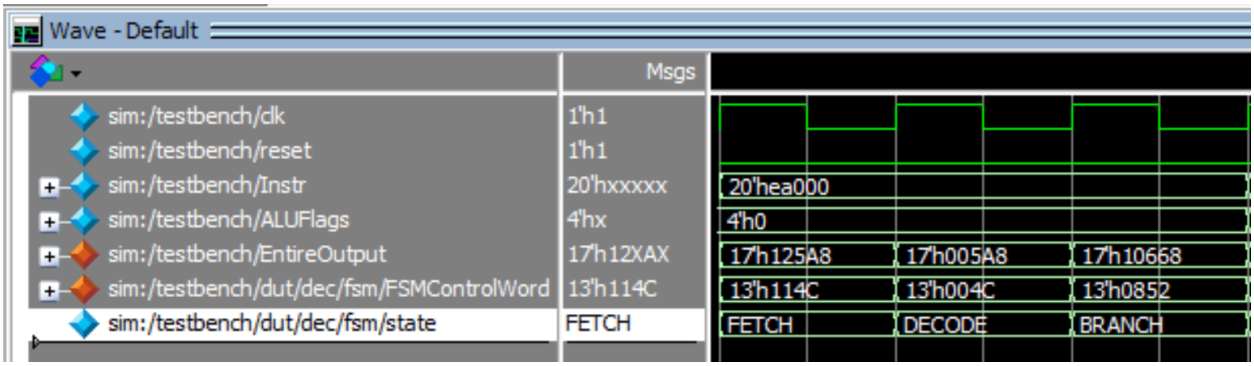
شکل شماره 4: خروجی اجرای کد ORNE R2, R1, #5



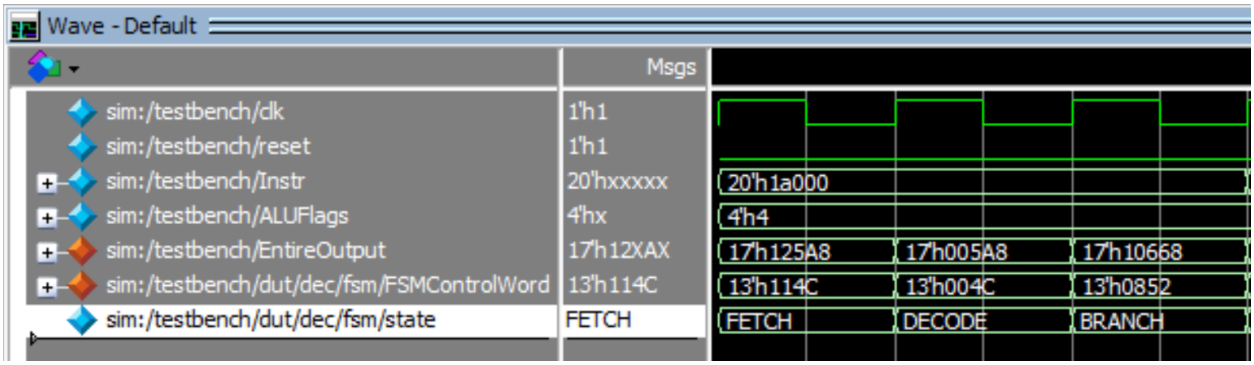
شکل شماره 5: خروجی اجرای کد LDR R3, [R1, #10]



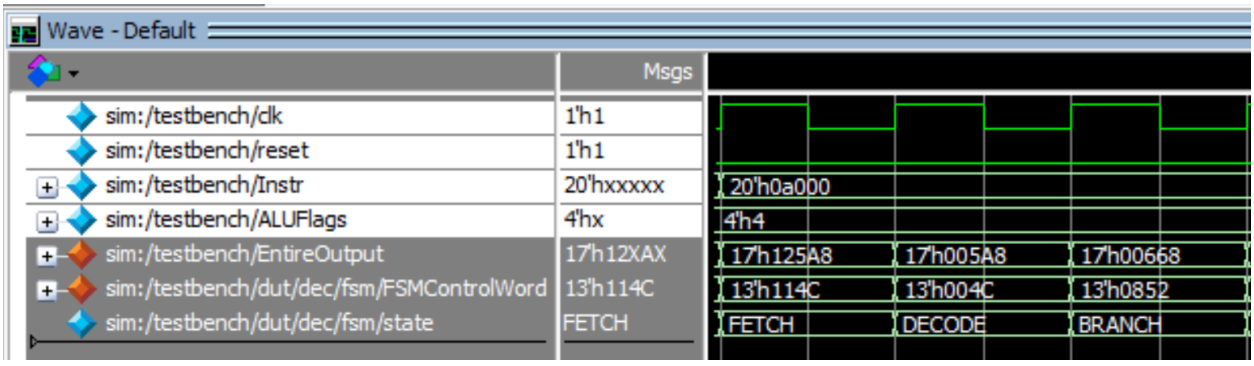
شکل شماره 6: خروجی اجرای کد STR R5, [R2, #2]



شکل شماره 7: خروجی اجرای کد B



شکل شماره 8: خروجی اجرای کد BNE با zero = 1



شکل شماره 9: خروجیاجرای کد BEQ با zero = 1