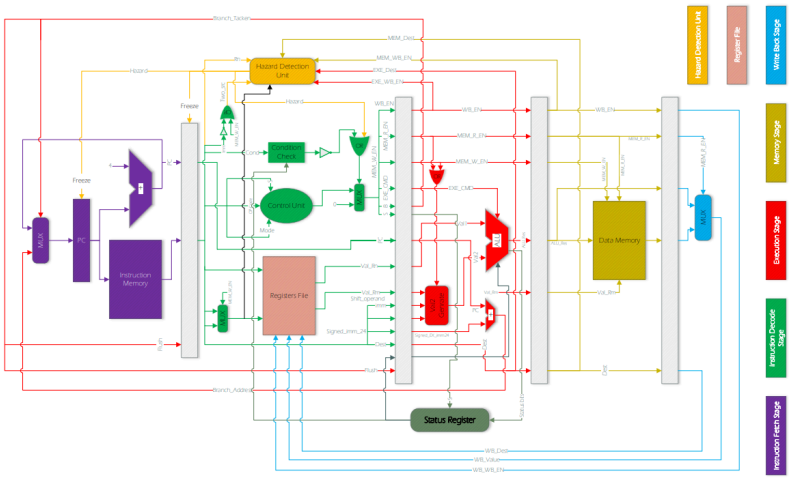
به نام خدا

گزارش کار آزمایشگاه معماری

دانشکده مهندسی برق و کامپیوتر	هستی کریمی	810100589
	علی عطاءاللهی	810199461

پردازنده ARM

در این درس می خواهیم با پیاده سازی پردازنده ARM، با ساز و کار پردازنده ها بیشتر آشنا شویم. برای این کار، ابتدا کل پروژه به چند بخش تقسیم شده که در نهایت محصول نهایی را می سازند. می دانیم که این پردازنده از 5 بخش تشکیل شده که به ترتیب: Instruction Fetch, Decode, Execution, Memory, Write Back هستند. که پس از هریک از این بخش ها یک رجیستر متنظر با آن ها قرار دارد تا اطلاعات هر استیج را در کلاک بعدی به استیج بعد منتقل کنند. در 5 قسمت اول گزارش نحوه پیاده سازی هر یک از این بخش ها توضیح داده شده و در نهایت تغییراتی که برای بهبود عملکرد پردازنده اعمال می شود مثل افزودن Forwarding و Cache و جزئیات آن ها شرح داده میشود. شکل کلی ساختار پردازنده صورت زیر است:



قسمت اول Instruction Fetch Stage:

افزودن Program Counter برای خواندن دستور بعد و همچنین خواندن دستور مورد نظر از بخش حافظه دستورات، از وظایف این قسمت هستند. برای این کار نیاز به یک جمع کننده داریم که هر بار PC مرحله قبل را 4 واحد افزایش دهد. بدلیل وجود دستورات پرشی، لزوما پس از اجرا شدن دستور فعلی نباید خط بعد آن اجرا شود و در صورتی که دستور شرطی باشد و شرط پرش آن برقرار باشد، خطی که باید PC به آن اشاره کند در قسمت های جلوتر تولید شده(Branch_Address) و به این قسمت باز می گردد تا مقدار PC به درستی ست شود. پس از خوانده شدن دستور مورد نظر از Instruction Memory، اطلاعات وارد IF_Reg میشوند تا در کلاک بعدی به استیج Decode برای ادامه روند اجرا منتقل شوند. چون معماری بصورت پایپ لاین هست، ما در مراحل بعد متوجه برقرار بودن یا نبودن شرط پرش میشویم و در این صورت باید دستوراتی که به اشتباه وارد پایپ لاین شده اند را پاک کنیم و بدین منظور یک سیگنال کنترلی flush(در کد زیر به اسم Branch_taken) قرار میدهیم که محتوای رجیستر را در صورت نیاز پاک کند تا به جلو هدایت نشوند، همچنین جلوتر خواهیم دید که به دلیل امکان وابستگی دستورات متوالی ممکن است hazard رخ دهد که در این صورت باید متوقف شویم تا داده ها کامل شوند و سپس پایپ لاین به کار خود ادامه دهد که برای این منظور نیز سیگنالی از واحد hazard detection unit (که در ادامه آورده میشود) به رجیستر حاوی PC و IF_Reg به نام freeze به منظور متوقف نمودن آن ها ارسال میشود. توضیحات فوق را بدین شکل پیاده سازی نمودیم:

```
module IF_Stage (input clk, rst, freeze, Branch_taken,
                 input [31:0] Branch_Address,
                 output [31:0] PC_ID_Stage_Reg, Ins);

  wire [31:0] PC_reg_in;
  reg  [31:0] PC_reg_out;

  Mux mux (
       PC_ID_Stage_Reg,
       Branch_Address,
       Branch_taken,
       PC_reg_in);

  Adder pcAdder (
        PC_reg_out,
        4,
        PC_ID_Stage_Reg);

  Instrucrion_Memory instruction_mem (
         PC_reg_out,
         Ins);

  always @(posedge clk, posedge rst)
  begin
    if (rst)
      PC_reg_out <= 0;
    else if (~freeze)
      PC_reg_out <= PC_reg_in;
  end
endmodule
```

ماژول های استفاده شده بصورت زیر هستند:

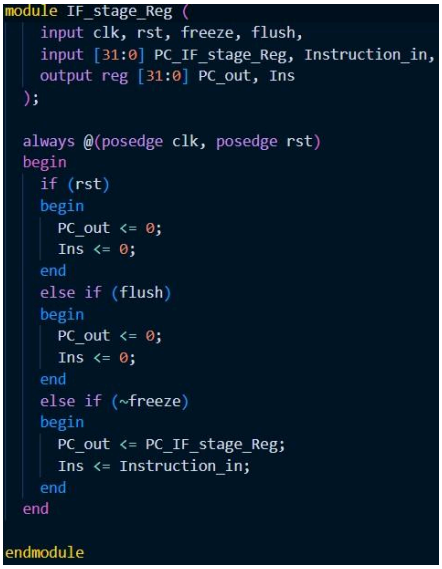
```
module Mux (
    input [31:0] a, b,
    input sel,
    output [31:0] c
  );

  assign c = (sel ? b : a);

endmodule
```

```
module Adder (
    input  [31:0] a, b,
    output [31:0] res
  );

  assign res = a + b;

endmodule
```

و واحد Instruction_Memory که حاوی دستورات قابل خواندن و اجرا برای پردازنده ARM است:

```
module Instrucrion_Memory (
    input [31:0] in,
    output [31:0] out
  );

  reg [31:0] mem[0:46];

  initial
  begin
    mem[0] = 32'b1110_00_1_1101_0_0000_0000_000000010100;
    mem[1] = 32'b1110_00_1_1101_0_0000_0001_101000000001;
    mem[2] = 32'b1110_00_1_1101_0_0000_0010_000100000011;
    mem[3] = 32'b1110_00_0_0100_1_0010_0011_000000000010;
    mem[4] = 32'b1110_00_0_0101_0_0000_0100_000000000000;
    mem[5] = 32'b1110_00_0_0010_0_0100_0101_000100000100;
    mem[6] = 32'b1110_00_0_0110_0_0000_0110_000010100000;
    mem[7] = 32'b1110_00_0_1100_0_0101_0111_000101000010;
    mem[8] = 32'b1110_00_0_0000_0_0111_1000_000000000011;
    mem[9] = 32'b1110_00_0_1111_0_0000_1001_000000000110;
    mem[10] = 32'b1110_00_0_0001_0_0100_1010_000000000101;
    mem[11] = 32'b1110_00_0_1010_1_1000_0000_000000000110;
    mem[12] = 32'b0001_00_0_0100_0_0001_0001_000000000001;
    mem[13] = 32'b1110_00_0_1000_1_1001_0000_000000001000;
    mem[14] = 32'b0000_00_0_0100_0_0010_0010_000000000010;
    mem[15] = 32'b1110_00_1_1101_0_0000_0000_101100000001;
    mem[16] = 32'b1110_01_0_0100_0_0000_0001_000000000000;
    mem[17] = 32'b1110_01_0_0100_1_0000_1011_000000000000;
    mem[18] = 32'b1110_01_0_0100_0_0000_0010_000000000100;
    mem[19] = 32'b1110_01_0_0100_0_0000_0011_000000001000;
    mem[20] = 32'b1110_01_0_0100_0_0000_0100_000000001101;
    mem[21] = 32'b1110_01_0_0100_0_0000_0101_000000010000;
    mem[22] = 32'b1110_01_0_0100_0_0000_0110_000000010100;
    mem[23] = 32'b1110_01_0_0100_1_0000_1010_000000000100;
    mem[24] = 32'b1110_01_0_0100_0_0000_0111_000000011000;
    mem[25] = 32'b1110_00_1_1101_0_0000_0001_000000000100;
    mem[26] = 32'b1110_00_1_1101_0_0000_0010_000000000000;
    mem[27] = 32'b1110_00_1_1101_0_0000_0011_000000000000;
    mem[28] = 32'b1110_00_0_0100_0_0000_0100_000100000011;
    mem[29] = 32'b1110_01_0_0100_1_0100_0101_000000000000;
    mem[30] = 32'b1110_01_0_0100_1_0100_0110_000000000100;
    mem[31] = 32'b1110_00_0_1010_1_0101_0000_000000000110;
    mem[32] = 32'b1100_01_0_0100_0_0100_0110_000000000000;
    mem[33] = 32'b1100_01_0_0100_0_0100_0101_000000000100;
    mem[34] = 32'b1110_00_1_0100_0_0011_0011_000000000001;
    mem[35] = 32'b1110_00_1_1010_1_0011_0000_000000000011;
    mem[36] = 32'b1011_10_1_0_11111111111111111110111;
    mem[37] = 32'b1110_00_1_0100_0_0010_0010_000000000001;
    mem[38] = 32'b1110_00_0_1010_1_0010_0000_000000000001;
    mem[39] = 32'b1011_10_1_0_11111111111111111110011;
    mem[40] = 32'b1110_01_0_0100_1_0000_0001_000000000000;
    mem[41] = 32'b1110_01_0_0100_1_0000_0010_000000000100;
    mem[42] = 32'b1110_01_0_0100_1_0000_0011_000000001000;
    mem[43] = 32'b1110_01_0_0100_1_0000_0100_000000001100;
    mem[44] = 32'b1110_01_0_0100_1_0000_0101_000000010000;
    mem[45] = 32'b1110_01_0_0100_1_0000_0110_000000010100;
    mem[46] = 32'b1110_10_1_0_11111111111111111111111;
  end

  assign out = mem[in>>2];

endmodule
```

رجیستر پس از این مرحله نیز بصورت زیر می باشد:



```
module IF_stage_Reg (
    input clk, rst, freeze, flush,
    input [31:0] PC_IF_stage_Reg, Instruction_in,
    output reg [31:0] PC_out, Ins
);

  always @(posedge clk, posedge rst)
  begin
    if (rst)
    begin
      PC_out <= 0;
      Ins <= 0;
    end
    else if (flush)
    begin
      PC_out <= 0;
      Ins <= 0;
    end
    else if (~freeze)
    begin
      PC_out <= PC_IF_stage_Reg;
      Ins <= Instruction_in;
    end
  end

endmodule
```

قسمت دوم Decode Stage:

در این بخش محتوای 32 بیتی خروجی استیج قبل را به قسمت های کوچکتر تبدیل کرده و آن ها را در این مرحله تفسیر میکنیم، در این قسمت سه ماژول اصلی داریم: Condition Check, Control Unit, Register File
در قسمت Control Unit مشخص میشود که دستور چیست، ADD, SUB, AND, MOVE یا از دیگر دستورات تعریف شده برای این پردازنده و بر اساس آن تمام سیگنال های کنترلی ست میشوند، مانند خواندن یا نوشتن در حافظه، WB_EN و غیره، سپس اگر شرط واحد Condition Check برقرار بود و hazard رخ نداده بود، این مقدار ها وارد رجیستر استیج دیکود میشوند و در غیر این صورت همه آن ها غیرفعال و اساین به صفر می شوند. وظیفه Register File نیز خواندن محتوای یک رجیستر یا نوشتن در آن است. پیاده سازی توضیحات فوق به شکل زیر است:

```
module ID_Stage (
    input clk, rst,
    input hazard, WB_WB_EN,
    input [3:0] Dest_wb, SR,
    input [31:0] Ins, dest_wb,

    output writeBackEn, MEM_R_en, MEM_W_en, b, S, Two_src, imm, use_src1,
    output [3:0] EXE_CMD, Dest, src1, src2,
    output [11:0] Shift_operand,
    output [23:0] Signed_imm_24,
    output [31:0] Val_Rn, Val_Rm
  );

  wire stop, WB_EN_CU, MEM_R_EN_CU, MEM_W_EN_CU, B_CU, S_CU;
  wire [3:0] EXE_CMD_CU;

  wire [3:0] cond = Ins[31:28];
  wire [1:0] mode = Ins[27:26];
  wire I = Ins[25];
  wire [3:0] opcode = Ins[24:21];
  wire S_in = Ins[20];
  wire [3:0] Rn = Ins[19:16];
  wire [3:0] Rd = Ins[15:12];
  wire [3:0] Rm = MEM_W_EN_CU ? Rd : Ins[3:0];
  assign Shift_operand = Ins[11:0];
  assign Signed_imm_24 = Ins[23:0];

  ConditionCheck CC (
                     .cond(cond),
                     .SR(SR),

                     .condition_check_result(Is_Valid)
                    );
```

```
  Register_File RF (
                    .clk(clk),
                    .rst(rst),
                    .src_1(Rn),
                    .src_2(Rm),
                    .WB_WB_EN(WB_WB_EN),
                    .Dest_wb(Dest_wb),
                    .dest_wb(dest_wb),

                    .Val_Rn(Val_Rn),
                    .Val_Rm(Val_Rm)
                   );

  ControlUnit CU (
                  .opcode(opcode),
                  .mode(mode),
                  .S_IN(S_in),

                  .EXE_CMD(EXE_CMD_CU),
                  .writeBackEn(WB_EN_CU),
                  .MEM_R_en(MEM_R_EN_CU),
                  .MEM_W_en(MEM_W_EN_CU),
                  .b(B_CU),
                  .S(S_CU)
                 );

  assign imm = I;
  assign Dest = Rd;
  assign stop = hazard || !Is_Valid;
  assign {EXE_CMD, writeBackEn, MEM_R_en, MEM_W_en, b, S} = stop ? 9'b0 : {EXE_CMD_CU, WB_EN_CU, MEM_R_EN_CU, MEM_W_EN_CU, B_CU, S_CU};

  assign Two_src = MEM_W_EN_CU || (!I && mode == 2'b00);
  assign use_src1 = opcode != 4'b1101 && opcode != 4'b1111 && mode != 2'b10;
  assign src1 = Rn;
  assign src2 = Rm;


endmodule
```

ماژول Control Unit:

در این ماژول سیگنال های کنترلی با کمک واحد Status Register تولید میشوند، در حالتی که mode برابر 00 است یعنی دستور جزو دستورات محاسباتی ست که بجز دستور compare و test در بقیه دستورات باید مقداری که در مرحله بعد تولید میشود را در رجیستر های رجیستر فایل بنویسیم که در این حالت باید writeBackEn یک شود تا در مرحله اخر این مقدار وارد رجیستر فایل شود نه محتوای مموری، EXE_CMD نیز عملیاتیست که ALU باید روی ورودی های خود انجام دهد مانند جمع و تفریق و اند منطقی و ... . حال اگر mode برابر 01 باشد، دستور کار با حافظه است و بنابراین دستور های MEM_READ و MEM_WRITE بر اساس خروجی Status Register ست میشوند و در اگر mode برابر 10 بود یعني دستور پرشیست و سیگنال b یک میشود. برای اپدیت Status Register نیز در همین قسمت سیگنال مربوطه اساین میشود.

```
module ControlUnit (
    input [3:0] opcode,
    input [1:0] mode,
    input S_IN,
    output reg [3:0] EXE_CMD,
    output reg writeBackEn, MEM_R_en, MEM_W_en, b, S
  );

`define MOV 4'b1101
`define MVN 4'b1111
`define ADD 4'b0100
`define ADC 4'b0101
`define SUB 4'b0010
`define SBC 4'b0110
`define AND 4'b0000
`define ORR 4'b1100
`define EOR 4'b0001
`define CMP 4'b1010
`define TST 4'b1000
`define LDR 4'b0100
`define STR 4'b0100

  always @(mode, opcode, S_IN)
  begin
    {EXE_CMD, writeBackEn, MEM_R_en, MEM_W_en, b, S} = 9'b0;

    case(mode)
      2'b00:
      begin
        S = S_IN;
        case(opcode)
          `MOV:
          begin
            EXE_CMD = 4'b0001;
            writeBackEn = 1'b1;
          end

          `MVN:
          begin
            EXE_CMD = 4'b1001;
            writeBackEn = 1'b1;
          end

          `ADD:
          begin
            EXE_CMD = 4'b0010;
            writeBackEn = 1'b1;
          end
```

```
          `ADC:
          begin
            EXE_CMD = 4'b0011;
            writeBackEn = 1'b1;
          end

          `SUB:
          begin
            EXE_CMD = 4'b0100;
            writeBackEn = 1'b1;
          end

          `SBC:
          begin
            EXE_CMD = 4'b0101;
            writeBackEn = 1'b1;
          end

          `AND:
          begin
            EXE_CMD = 4'b0110;
            writeBackEn = 1'b1;
          end

          `ORR:
          begin
            EXE_CMD = 4'b0111;
            writeBackEn = 1'b1;
          end

          `EOR:
          begin
            EXE_CMD = 4'b1000;
            writeBackEn = 1'b1;
          end

          `CMP:
          begin
            EXE_CMD = 4'b0100;
          end

          `TST:
          begin
            EXE_CMD = 4'b0110;
          end
```

```
            default:
              EXE_CMD = 4'b0000;
          endcase
        end


        2'b01:
        begin
          EXE_CMD = 4'b0010;
          MEM_R_en = S_IN;
          MEM_W_en = !S_IN;
          writeBackEn = S_IN;
        end


        2'b10:
        begin
          b = 1'b1;
        end

        default:;
      endcase

    end

endmodule
```

ماژول Condition Check:

بر اساس جدول 3 برقرار بودن یا نبود شرط چک میشود:

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	-	See *Condition code 0b1111*	-

جدول 3- کد شرط دستورات

```
module ConditionCheck (
    input [3:0] cond,
    input [3:0] SR,    // N Z C V
    output reg condition_check_result);

  always @ (cond, SR)
  case (cond)
    // Z set
    4'b0000:
      condition_check_result = SR[2];
    // Z clear
    4'b0001:
      condition_check_result = !SR[2];
    // C set
    4'b0010:
      condition_check_result = SR[1];
    // C clear
    4'b0011:
      condition_check_result = !SR[1];
    // N set
    4'b0100:
      condition_check_result = SR[3];
    // N clear
    4'b0101:
      condition_check_result = !SR[3];
    // V set
    4'b0110:
      condition_check_result = SR[0];
    // V clear
    4'b0111:
      condition_check_result = !SR[0];
    // C set and Z clear
    4'b1000:
      condition_check_result = SR[1] && !SR[2];
    // C clear or Z set
    4'b1001:
      condition_check_result = !SR[1] || SR[2];
    // N == V
    4'b1010:
      condition_check_result = SR[3] == SR[0];
    // N != V
    4'b1011:
      condition_check_result = SR[3] != SR[0];
    // Z==0, N==V
    4'b1100:
      condition_check_result = !SR[2] && (SR[3] == SR[0]);
    // Z==1, N!=V
    4'b1101:
      condition_check_result = SR[2] && (SR[3] != SR[0]);
    // Always
    4'b1110:
      condition_check_result = 1'b1;
    // Never
    4'b1111:
      condition_check_result = 1'b0;
  endcase

endmodule
```

ماژول Register File:

رجیستر فایل همانطور که از اسمش پیداست حاوی رجیستر ها است و با لبه بالارونده کلاک میتواند محتوای دو رجیستر را بخواند و با لبه پایین رونده در یک رجیستر بنویسد:

```
module Register_File (
    input clk, rst,
    input [3:0] Dest_wb, src_1, src_2,
    input [31:0] dest_wb,
    input WB_WB_EN,
    output [31:0] Val_Rn, Val_Rm
  );

  reg [31:0] RegFile [0:14];

  integer i;

  initial
  begin
    for (i = 0; i < 15 ; i = i + 1 )
      RegFile[i] = i;
  end

  assign Val_Rn = RegFile[src_1];
  assign Val_Rm = RegFile[src_2];

  always @(negedge clk, posedge rst)
  begin
    if (rst)
      for (i = 0; i < 15 ; i = i + 1 )
        RegFile[i] <= i;
    else if (WB_WB_EN)
      RegFile[Dest_wb] <= dest_wb;
  end

endmodule
```

در این قسمت ماژول Status Register نیز توضیح داده میشود تا استفاده های آن در قسمت بالا شفاف شود: مقدار سیگنال S در Control Unit ست شده است و مقدار status_in از خروجی ALU در مرحله بعد گرفته میشود که مقادیر N ,Z, C, V را پس از انجام عملیات محاسباتی و بر اساس نتیجه آن ست میشوند. که N نشانگر منفی بودن، Z نشانگر صفر بودن، C رقم نقلی و V اورفلو میباشد. همانطور که مشاهده میشود با لبه پایین رونده کلاک در Status_Register دیتا نوشته میشود.

```
module Status_Register (
    input clk, rst,
    input [3:0] status_in,
    input S,
    output reg [3:0] status_out
  );

  always @(negedge clk, posedge rst)
  begin
    if (rst)
      status_out <= 0;
    else if (S)
      status_out <= status_in;
  end

endmodule
```

رجیستر پس از این استیج نیز بصورت زیر میباشد که در صورت ریست شدن یا فلاش مقادیر را صفر میکند، در صورت فریز شدن همان مقدار قبلی را حفظ میکند و در غیر این صورت مقادیر جدید را به خروجی خود میفرستد.

```
module ID_Stage_Reg (
    input clk, rst, freeze, flush,
    input WB_en_in, MEM_R_en_in, MEM_W_en_in, B_IN, S_IN, imm_IN,
    input [3:0] EXE_CMD_IN, Dest_in, Status_R_in,
    input [11:0] shift_operand_IN,
    input [23:0] signed_imm_24_IN,
    input [31:0] PC_IF_stage_Reg, Val_Rn_IN, Val_Rm_IN,
    input [3:0] src1_in, src2_in,

    output reg WB_en, MEM_R_en, MEM_W_EN, B, S, imm,
    output reg [3:0] EXE_CMD, Dest, Status_R_out,
    output reg [11:0] Shift_operand,
    output reg [23:0] Signed_imm_24,
    output reg [31:0] PC_out, Val_Rn, Val_Rm,
    output reg [3:0] src1_out, src2_out
  );

  always @(posedge clk, posedge rst)
  begin
    if (rst)
    begin
      PC_out <= 0;
      {WB_en, MEM_R_en, MEM_W_EN, B, S, EXE_CMD, Val_Rn, Val_Rm, imm, Shift_operand, Signed_imm_24, Dest, Status_R_out, src1_out, src2_out} <= 126'b0;
    end
    else if (freeze)
    begin
      PC_out <= PC_out;
      {WB_en, MEM_R_en, MEM_W_EN, B, S, EXE_CMD, Val_Rn, Val_Rm, imm, Shift_operand, Signed_imm_24, Dest, Status_R_out, src1_out, src1_out} <=
      {WB_en, MEM_R_en, MEM_W_EN, B, S, EXE_CMD, Val_Rn, Val_Rm, imm, Shift_operand, Signed_imm_24, Dest, Status_R_out, src1_out, src2_out};
    end
    else if (flush)
    begin
      PC_out <= 0;
      {WB_en, MEM_R_en, MEM_W_EN, B, S, EXE_CMD, Val_Rn, Val_Rm, imm, Shift_operand, Signed_imm_24, Dest, Status_R_out, src1_out, src2_out} <= 126'b0;
    end
    else
    begin
      PC_out <= PC_IF_stage_Reg;
      {WB_en, MEM_R_en, MEM_W_EN, B, S, EXE_CMD, Val_Rn, Val_Rm, imm, Shift_operand, Signed_imm_24, Dest, Status_R_out, src1_out, src2_out} <=
      {WB_en_in, MEM_R_en_in, MEM_W_en_in, B_IN, S_IN, EXE_CMD_IN, Val_Rn_IN, Val_Rm_IN, imm_IN, shift_operand_IN, signed_imm_24_IN, Dest_in, Status_R_in, src1_in, src2_in};
    end
  end

endmodule
```

قسمت سوم EXE Stage:

در این مرحله عملیات های محاسباتی لازم انجام میشود و شامل سه ماژول میشود که ALU وظیفه انجام دستور را به عهده دارد، num2_generator که دومین مقداری که ALU برای انجام عملیات مورد نظر با مقدار اول لازم دارد را تولید میکند و در نهایت یک Adder تا ادرس برنچ را مشخص کند.

```
module EXE_Stage (
    input clk, rst,
    input[3:0] EXE_CMD,
    input MEM_R_en, MEM_W_en,
    input[31:0] PC_ID_Stage_Reg,
    input[31:0] Val_Rm_in, Val_Rn, ALU_res_f, WB_val_f,
    input imm,
    input[11:0] Shift_operand,
    input[23:0] Signed_imm_24,
    input[3:0] status_IN,
    input [1:0] sel_src1, sel_src2,

    output[31:0] address, Branch_Address, Val_Rm_out,
    output[3:0] status
  );
  wire [31:0] ALU_src_1, Val2_src;

  wire[31:0] Signed_imm_32 = { {6{Signed_imm_24[23]}}, Signed_imm_24, 2'b00};
  wire mem = MEM_R_en || MEM_W_en;
  wire[31:0] Val2;

  assign ALU_src_1 = (sel_src1 == 2'b00) ? Val_Rn :
         (sel_src1 == 2'b01) ? ALU_res_f:
         (sel_src1 == 2'b10) ? WB_val_f:
         Val_Rn;

  assign Val2_src = (sel_src2 == 2'b00) ? Val_Rm_in :
         (sel_src2 == 2'b01) ? ALU_res_f:
         (sel_src2 == 2'b10) ? WB_val_f:
         Val_Rm_in;

  assign Val_Rm_out = Val2_src;

  num2 num2_(
                .Shift_operand(Shift_operand),
                .RM_value(Val2_src),
                .imm(imm),
                .mem(mem),
                .Val2(Val2)
              );
```

```
  ALU alu(
          .input1(ALU_src_1),
          .input2(Val2),
          .carry_in(status_IN[1]),
          .command(EXE_CMD),
          .out(address),
          .carry_out(status[1]),
          .V(status[0]),
          .Z(status[2]),
          .N(status[3])
        );

  Adder adder(
          .a(PC_ID_Stage_Reg),
          .b(Signed_imm_32),
          .res(Branch_Address)
        );

endmodule
```

ماژول ALU:

بر اساس EXE_CMD و بیت C از Status Register که همان (carry_in) میباشد، عملیات مربوطه را روی دو ورودی خود انجام میدهد و سپس مقادیر N ,Z, C, V را مشخص میکند.

```
module ALU (
    input[31:0] input1, input2,
    input carry_in,
    input[3:0] command,

    output reg[31:0] out,
    output reg carry_out, V,
    output N, Z
);

    always @ (input1, input2, carry_in, command) begin
        out = 32'b0;
        carry_out = 1'b0;
        case (command)
            4'b0001: out = input2;
            4'b1001: out = ~input2;
            4'b0010: {carry_out, out} = input1 + input2;
            4'b0011: {carry_out, out} = input1 + input2 + carry_in;
            4'b0100: {carry_out, out} = input1 - input2;
            4'b0101: {carry_out, out} = input1 - input2 - 1 + carry_in;
            4'b0110: out = input1 & input2;
            4'b0111: out = input1 | input2;
            4'b1000: out = input1 ^ input2;
            default: {carry_out, out} = 33'b0;
        endcase
    end

    assign N = out[31];
    assign Z = (out == 32'b0);

    always @(command, input1, input2, N) begin
        V = 1'b0;
        case (command)
            4'b0010: V = (input1[31] & input2[31] & (~N)) || ( (~input1[31]) & (~input2[31]) & N);
            4'b0011: V = (input1[31] & input2[31] & (~N)) || ( (~input1[31]) & (~input2[31]) & N);
            4'b0100: V = (input1[31] & (~input2[31]) & (~N)) || ( (~input1[31]) & input2[31] & N);
            4'b0101: V = (input1[31] & (~input2[31]) & (~N)) || ( (~input1[31]) & input2[31] & N);
            default: V = 1'b0;
        endcase
    end

endmodule
```

ماژول num2_generator:

این ماژول ورودی دوم ALU را تولید میکند. این ورودی در دستورات LDR و STR برابر 12 بیت offset خواهد بود که با نام operand_shift وارد EXE شده است که تشخیص این حالات بر اساس سیگنال mem که نشان دهنده کار با مموری است انجام میشود(MEM_W_en || MEM_R_en). حال اگر دستور لود و یا استور نباشد بر اساس بیت immediate انتخاب میشود که آیا مقدار immediate و یا مقدار RM توسط این ماژول انتخاب شود و به عنوان ورودی دوم به ماژول ALU فرستاده شود. برای اینکه مقدار Immediate بیشتری را بتوانیم ساپورت کنیم از شیوه rotate کردن استفاده کردیم که به اندازه دو برابر مقدار rotate_imm چرخش انجام میشود. اگر RM انتخاب شود نیز بر اساس جدول زیر شیفت مورد نظر انجام میشود که پیاده سازی آن به شکل زیر می باشد:

وضعیت شیفت	توضیحات	مقدار
LSL	Logical shift left	00
LSR	Logical shift right	01
ASR	Arithmetic shift right	10
ROR	Rotate right	11

```
module num2 (
    input[11:0] Shift_operand,
    input[31:0] RM_value,
    input imm, mem,
    output reg[31:0] Val2
);

    wire[7:0] immed_8 = Shift_operand[7:0];
    wire[3:0] rotate_imm = Shift_operand[11:8];
    wire[4:0] shift_imm = Shift_operand[11:7];
    wire[1:0] shift = Shift_operand[6:5];
    reg[63:0] temp_64;

    always @ (imm, shift, Shift_operand, RM_value) begin
        temp_64 = 64'b0;
        Val2 = 32'b0;
        if (mem)
            Val2 = { {20{Shift_operand[11]}}, Shift_operand };
        else if (imm) begin
            temp_64[39:32] = immed_8;
            temp_64 = temp_64 >> (2*rotate_imm);
            Val2 = temp_64[31:0] | temp_64[63:32];
        end else begin
            temp_64[63:32] = RM_value;
            if (shift == 2'b11) begin
                temp_64[63:32] = RM_value;
                temp_64 = temp_64 >> shift_imm;
                Val2 = temp_64[31:0] | temp_64[63:32];
            end else if (shift == 2'b01)
                Val2 = RM_value >> shift_imm;
            else if (shift == 2'b10)
                Val2 = RM_value >>> shift_imm;
            else
                Val2 = RM_value << shift_imm;
        end
    end

endmodule
```

ماژول Adder:

در مراحل قبل از ادرر استفاده شده و کد آن آورده شده فقط در این مرحله از این ماژول استفاده میکنیم تا ادرس پرش را در صورت پرشی بودن دستور به دست آوریم. پس PC را با مقدار ثابت جمع زده و به استیج ابتدایی میفرستیم.

و در نهایت رجیستر پس از این استیج بصورت زیر است:

```
module EXE_Reg (
    input clk, rst, freeze, WB_en_in, MEM_R_en_in, MEM_W_en_in,
    input[31:0] ALU_result_in, ST_val_in,
    input[3:0] Dest_in,

    output reg WB_en, MEM_R_en, MEM_W_en,
    output reg[31:0] address, data,
    output reg[3:0] Dest
);

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            address <= 32'b0;
            data <= 32'b0;
            Dest <= 4'b0;
            WB_en <= 1'b0;
            MEM_R_en <= 1'b0;
            MEM_W_en <= 1'b0;
        end else if (~freeze) begin
            address <= ALU_result_in;
            data <= ST_val_in;
            Dest <= Dest_in;
            WB_en <= WB_en_in;
            MEM_R_en <= MEM_R_en_in;
            MEM_W_en <= MEM_W_en_in;
        end
    end

endmodule
```

قسمت چهارم Memory Stage:

در این قسمت تعامل با حافظه خواهد بود و طبق دستور دیکود شده یا از آن اطلاعاتی را میخوانیم و یا بر روی یک آدرس آن اطلاعاتی را ذخیره میکنیم:
لازم به ذکر است که قسمت کامنت شده برای قبل از اضافه کردن SRAM بوده که به سادگی عملیات فوق را با حافظه درونی انجام میداده و ماژول Data Memory آن بصورت زیر بود:

```
module DataMemory (
    input clk, rst, MEM_W_en, MEM_R_en,
    input[31:0] address, data,
    output[31:0] out
);

reg[31:0] memory[0:63];

wire[7:0] i = (address - 1024) >> 2;

assign out = MEM_R_en ? memory[i] : 32'b0;

always @(posedge clk)
begin
    if (MEM_W_en)
        memory[i] <= data;
end

endmodule
```

اما پس از اضافه شدن SRAM و CACHE کد به شکل زیر خواهد بود:

```
module Memory (
    input clk, rst, WB_en,
    input MEM_W_EN, MEM_R_EN,
    input[31:0] ALU_res, ST_val,
    output[31:0] mem_out,
    output WB_en_out,
    output ready,
    inout[15:0] SRAM_DQ,
    output[17:0] SRAM_ADDR,
    output SRAM_WE_N,
    output SRAM_UB_N,
    output SRAM_LB_N,
    output SRAM_CE_N,
    output SRAM_OE_N
  );

  // DataMemory data_mem(
  //             .clk(clk),
  //             .rst(rst),
  //             .MEM_W_en(MEM_W_EN),
  //             .MEM_R_en(MEM_R_EN),
  //             .address(ALU_res),
  //             .data(ST_val),
  //             .out(mem_out)
  //           );

  assign WB_en_out = ready ? WB_en : 1'b0;
```

```
  wire sram_ready;
  wire sram_mem_wen_In, sran_mem_ren_in;
  wire [63:0] sram_read_data;

  Cache_CT cache_ct(
        .clk(clk), .rst(rst),
        .wrEn(MEM_W_EN), .rdEn(MEM_R_EN),
        .address(ALU_res),
        .write_data(ST_val),
        .read_data(mem_out),
        .ready(ready),
        .sram_ready(sram_ready),
        .sram_read_data(sram_read_data),
        .sram_wr_en(sram_mem_wen_In), .sram_rd_en(sran_mem_ren_in)
  );

  Sram_CT sram_ct (
            .clk(clk),
            .rst(rst),
            .wr_en(sram_mem_wen_In),
            .rd_en(sran_mem_ren_in),
            .address(ALU_res),
            .write_data(ST_val),

            .read_data(sram_read_data),
            .ready(sram_ready),
            .SRAM_DQ(SRAM_DQ),
            .SRAM_ADDR(SRAM_ADDR),
            .SRAM_WE_N(SRAM_WE_N),
            .SRAM_UB_N(SRAM_UB_N),
            .SRAM_LB_N(SRAM_LB_N),
            .SRAM_CE_N(SRAM_CE_N),
            .SRAM_OE_N(SRAM_OE_N)
  );

endmodule
```

میدانیم که خواندن از حافظه خارجی بیشتر از یک کلاک طول میکشد و برای این ازمایش ما زمان کار با حافظه را 6 کلاک در نظر گرفته ایم که در این میان باید پایپ لاین متوقف شود که سیگنالی بنام ready تعریف کرده ایم که اگر 1 نباشد یعنی باید متوقف شویم که این تغییر را روی رجیستر های میانی اعمال کرده و ان هارا فریز میکنیم.
ماژول مربوط به اس رم در زیر اورده شده:

ساختار خود Sram به شکل زیر میباشد:

```
module Sram(
    input clk,
    input rst,
    input wr_en,
    input rd_en,
    input[31:0] address,
    input[31:0] writedata,

    output[31:0] read_data,
    output ready,

    inout[15:0] SRAM_DQ,
    output[17:0] SRAM_ADDR,
    output SRAM_WE_N,
    output SRAM_UB_N,
    output SRAM_LB_N,
    output SRAM_CE_N,
    output SRAM_OE_N
);
    reg [15:0] memory[0:511];
    assign #5 SRAM_DQ = SRAM_WE_N ? memory[SRAM_ADDR] : 16'bz;
    always@(posedge clk)
    begin
        if(~SRAM_WE_N)
        begin
        memory[SRAM_ADDR] = SRAM_DQ;
        end
    end
endmodule
```

ماژول Sram_CT:

```
module Sram_CT(
    input clk, rst,
    input wr_en, rd_en,
    input[31:0] address,
    input [31:0] write_data,

    output reg [63:0] read_data,

    output reg ready,

    inout [15:0] SRAM_DQ,
    output reg [17:0] SRAM_ADDR,
    output reg SRAM_UB_N,
    output reg SRAM_LB_N,
    output reg SRAM_WE_N,
    output reg SRAM_CE_N,
    output reg SRAM_OE_N
  );
    wire [31:0] memAddr;
    wire [17:0] sramLowAddr, sramHighAddr, sramUpLowAddess, sramUpHighAddess;
    wire [17:0] sramLowAddrWrite, sramHighAddrWrite;
    reg [15:0] dq;
    reg [2:0] ns, ps;

    assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'd0;

    assign memAddr = address - 32'd1024;

    assign sramLowAddr = {memAddr[18:3], 2'd0};
    assign sramHighAddr = sramLowAddr + 18'd1;
    assign sramUpLowAddess = sramLowAddr + 18'd2;
    assign sramUpHighAddess = sramLowAddr + 18'd3;

    assign sramLowAddrWrite = {memAddr[18:2], 1'b0};
    assign sramHighAddrWrite = sramLowAddrWrite + 18'd1;

    assign SRAM_DQ = wr_en ? dq : 16'bz;

    localparam Idle = 3'd0, DataLow = 3'd1, DataHigh = 3'd2, DataUpLow = 3'd3, DataUpHigh = 3'd4, Done = 3'd5;

    always @(ps, wr_en, rd_en) begin
        case (ps)
            Idle: ns = (wr_en == 1'b1 || rd_en == 1'b1) ? DataLow : Idle;
            DataLow: ns = DataHigh;
            DataHigh: ns = DataUpLow;
            DataUpLow: ns = DataUpHigh;
            DataUpHigh: ns = Done;
            Done: ns = Idle;
        endcase
    end
```

```
    always @(*) begin
        SRAM_ADDR = 18'b0;
        SRAM_WE_N = 1'b1;
        ready = 1'b0;

        case (ps)
            Idle: ready = ~(wr_en | rd_en);
            DataLow: begin
                SRAM_WE_N = ~wr_en;
                if (rd_en) begin
                    SRAM_ADDR = sramLowAddr;
                    read_data[15:0] <= SRAM_DQ;
                end
                else if (wr_en) begin
                    SRAM_ADDR = sramLowAddrWrite;
                    dq = write_data[15:0];
                end
            end
            DataHigh: begin
                SRAM_WE_N = ~wr_en;
                if (rd_en) begin
                    read_data[31:16] <= SRAM_DQ;
                    SRAM_ADDR = sramHighAddr;
                end
                else if (wr_en) begin
                    SRAM_ADDR = sramHighAddrWrite;
                    dq = write_data[31:16];
                end
            end
            DataUpLow: begin
                SRAM_WE_N = 1'b1;
                if (rd_en) begin
                    read_data[47:32] <= SRAM_DQ;
                    SRAM_ADDR = sramUpLowAddess;
                end
            end
            DataUpHigh: begin
                SRAM_WE_N = 1'b1;
                if (rd_en) begin
                    read_data[63:48] <= SRAM_DQ;
                    SRAM_ADDR = sramUpHighAddess;
                end
            end
            Done: ready = 1'b1;
        endcase
    end

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            ps <= Idle;
        end
        else begin
            ps <= ns;
        end
    end
endmodule
```

در این قسمت طبق صورت پروژه برای نوشتن 4 استیت در نظر گرفته شده که بصورت زیر میباشند:

IDLE: در استیت IDLE، منتظر آنیم که دیتای مورد نظر به درستی شکل بگیرد و سپس سیگنال READY برابر با صفر میشود

WRITE_1: در این استیت، آدرس 16 بیت کم ارزش)0 تا 15(فرستاده شده و سپس دیتای کم ارزش بر روی باس مربوطه گذاشته میشود و سیگنال N_WE_SRAM برابر با صفر میشود تا دیتا در ماژول SRAM نوشته شود.

WRITE_2: در این استیت، آدرس 16 بیت پرارزش)16 تا 31(فرستاده شده و سپس دیتای پرارزش بر روی باس مربوطه گذاشته میشود و سیگنال N_WE_SRAM برابر با صفر میشود تا دیتا در ماژول SRAM نوشته شود.

END_WRITE: در این سیکل، سیگنال N_WE_SRAM را برابر با یک قرار میدهیم تا دیتا به اشتباه در آدرس موردنظر ننویسد و سیگنال READY برابر یک میشود تا نشان دهد دیتای مورد نظر در ماژول SRAM نوشته شده است و این عملیات تکمیل شده است.

و برای خواندن 5 استیت بصورت:

IDLE: در استیت IDLE، منتظر آنیم که دیتای مورد نظر به درستی شکل بگیرد و سپس سیگنال READY برابر با صفر میشود

READ_1: در این استیت، آدرس 16 بیت کم ارزش) 0 تا 15(فرستاده شده تا در استیت بعد بتوانیم دیتای مورد نظر را از باس بخوانیم.

READ_2: در این استیت، ابتدا 16 بیت کم ارزش از باس خوانده میشود و سپس آدرس 16 بیت پرارزش)16 تا 31(فرستاده شده است تا در استیت بعد آن را بخوانیم.

READ_3: در این سیکل، دیتای پرارزش از باس مربوطه خوانده میشود.

سیکل :5 در این استیت، سیگنال READY برابر یک میشود تا نشان دهد دیتای مورد نظر از ماژول SRAM خوانده شده است و این عملیات تکمیل شده است.

رجیستر پس از این استیج نیز به شکل زیر میباشد:

```
module MEM_Reg (
    input clk, rst, freeze, WB_en_in, MEM_R_en_in,
    input[31:0] ALU_result_in, Mem_read_value_in,
    input[3:0] Dest_in,

    output reg WB_en, MEM_R_en,
    output reg[31:0] address, MEM_result,
    output reg[3:0] Dest

);

    always @(posedge clk, posedge rst) begin
        if (rst) begin
            Dest <= 0;
            WB_en <= 0;
            MEM_R_en <= 0;
            address <= 0;
            MEM_result <= 0;
        end else if(~freeze) begin
            WB_en <= WB_en_in;
            MEM_R_en <= MEM_R_en_in;
            Dest <= Dest_in;
            address <= ALU_result_in;
            MEM_result <= Mem_read_value_in;
        end
    end

endmodule
```

قسمت پنجم Write Back:

در این مرحله محتوای خوانده شده از مموری یا محتوای آماده شده از بخش اجرا، بر اساس نوع دستور سلکت شده و برای نوشته شدن در یک رجیستر، به Register File فرستاده میشود.

```
module WB_Stage (
    input clk, rst, MEM_R_en,
    input[31:0] address, MEM_result,

    output[31:0] WB_value
);

    Mux mux(
        .a(address),
        .b(MEM_result),
        .sel(MEM_R_en),
        .c(WB_value)
    );

endmodule
```

بخش Hazard Unit:

به طور کلی در پردازنده ها سه نوع مخاطره وجود دارد:
الف) مخاطره ساختاری: این مخاطره در بطن ساختار خط لوله وجود دارد به همین دلیل ساختاری نا گرفته است. این مخاطره بین مرحله WB و ID به دلیل همزمانی خواندن و نوشتن از ثبات های عمومی ناشی می شود. برای رفع این مخاطره نوشتن در ثباتهای عمومی را به لبه پایین رونده منتقل کردیم. بنابراین این مخاطره در پردازنده رفع شده است.
ب) مخاطره کنترلی: این مخاطره ناشی از دستورات پرش است. در صورتی که دستور پرش وارد خط لوله شود به دلیل تأخیر در تشخیص و محاسبه آدرس پرش دو دستور به اشتباه وارد خط لوله میشود. برای رفع این مشکل سیگنال های Flush به پردازنده افزوده شد. بنابراین این مخاطره نیز رفع شده است.
ج) مخاطره داده ای: مخاطره داده‌ای به صورت زیر دسته بندی می گردد:
1- خواندن پس از نوشتن
2- نوشتن پس از خواندن
3- نوشتن پس از نوشتن
در این واحد، منابع 1Src و 2Src در مرحله ID با مقصدهای مراحل EXE و MEM به صورت مجزا مقایسه میشود و در صورت برابر بودن یکی از منابع با مقصدها، سیگنال کنترلی Signal_Detected_Hazard را برابر 1 قرار

میدهد. این سیگنال دستورات درون IF و رجیسترهای پس از آن را متوقف می نماید و حبابی را به خط لوله تزریق نماید.

حالات فوق به صورت زیر هستند:

برابری 1src با مقصد EXE در صورت یک بودن EN_WB در مرحله اجرا

برابری 1src با مقصد MEM در صورت یک بودن EN_WB در مرحله حافظه

برابری 2src با مقصد EXE در صورت یک بودن EN_WB در مرحله اجرا و دو منبعی بودن دستور

برابری 2src با مقصد MEM در صورت یک بودن EN_WB در مرحله حافظه و دو منبعی بودن دستور

پیاده سازی آن بصورت زیر می باشد:

```
module HazardDetector (input [3:0] src1,
                       input [3:0] src2,
                       input [3:0] Exe_Dest,
                       input Exe_WB_EN,
                       input [3:0] Mem_Dest,
                       input Mem_WB_EN,
                       input Two_src,
                       input use_src1,
                       input forward_mode,
                       input Exe_Mem_R_EN,
                       output hazard_Detected
                      );

 assign hazard_Detected = !forward_mode && ((Exe_WB_EN && (use_src1 && src1 == Exe_Dest)) ||
        (Exe_WB_EN && (Two_src  && src2 == Exe_Dest)) ||
        (Mem_WB_EN && (use_src1 && src1 == Mem_Dest)) ||
        (Mem_WB_EN && (Two_src  && src2 == Mem_Dest))) || Exe_Mem_R_EN && (src1 == Exe_Dest || src2 == Exe_Dest);

endmodule
```

بخش Forwarding Unit:

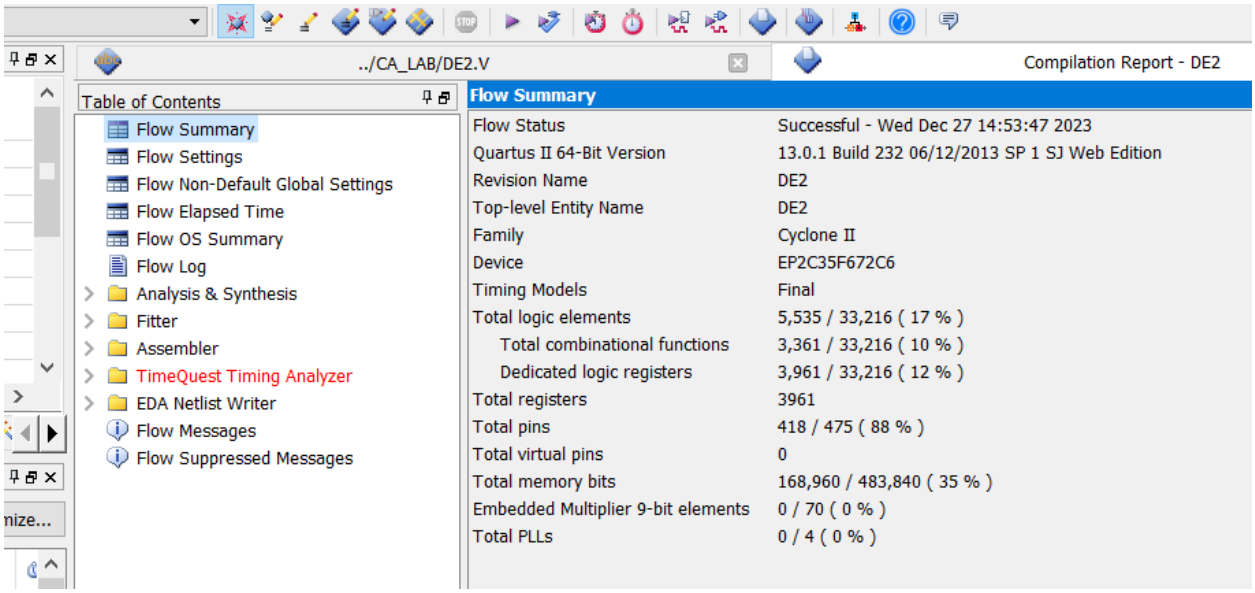
پس از کامل کردن 5 استیج مذکور، همانطور که قبل تر اشاره شد ممکن است یک دستور به محتوای رجیستر هایی نیاز داشته باشد که هنوز اطلاعات آن ها آپدیت نشده، چون می دانیم پس از استیج آخر دیتا به رجیستر فایل منتقل میشود. اما با این وجود می دانیم که دیتا مورد نظرمان پس از مرحله EXE آماده هست و فقط در دو استیج دیگر در رجیستر مربوطه نوشته میشود، ما در حالت عادی برای این کار باید پایپ لاین را چند سیکل متوقف کنیم و کارایی سیستم ما بسیار پایین می آید. برای حل این مشکل ما بجای خواندن محتوا مستقیما از خود رجیستر، دیتایی که قرار است در آن نوشته شود را پس از مرحله EXE مستقیما دریافت میکنیم و اصطلاحا آن را forward میکنیم. برای این کار ماژولی به اسم forwarding unit ایجاد کردیم تا عملیات فوق را برای ما انجام دهد:

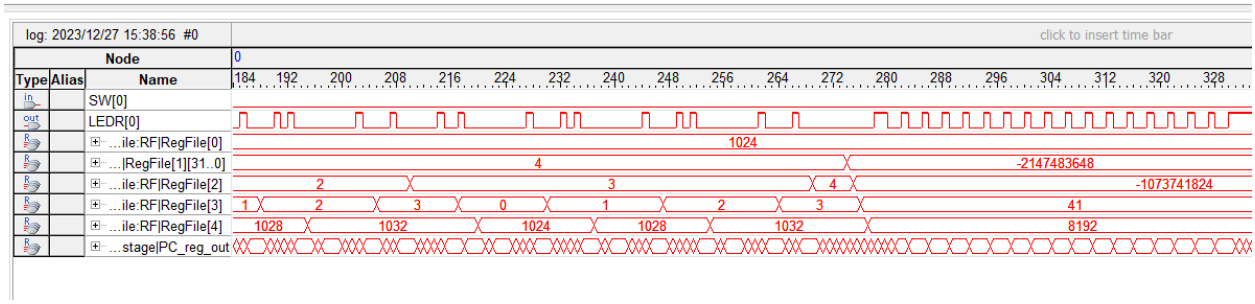
```
module Forwarding_unit (input forward_en,
                        input [3:0] src1,
                        input [3:0] src2,
                        input [3:0] WB_dest,
                        input [3:0] MEM_dest,
                        input WB_WB_en,
                        input MEM_WB_en,
                        output reg [1:0] sel_src1,
                        output reg [1:0] sel_src2
                        );


  always @(forward_en, src1, src2, MEM_WB_en, MEM_dest, WB_WB_en, WB_dest)
  begin
    sel_src1 = 2'b00;
    if (forward_en)
    begin
      if (MEM_WB_en && (src1 == MEM_dest))
      begin
        sel_src1 = 2'b01;
      end
      else if (WB_WB_en && (src1 == WB_dest))
      begin
        sel_src1 = 2'b10;
      end
      else
      begin
        sel_src1 = 2'b00;
      end
    end
  end
```

```
  always @(forward_en, src1, src2, MEM_WB_en, MEM_dest, WB_WB_en, WB_dest)
  begin
    sel_src2 = 2'b00;
    if (forward_en)
    begin
      if (MEM_WB_en && (src2 == MEM_dest))
      begin
        sel_src2 = 2'b01;
      end
      else if (WB_WB_en && (src2 == WB_dest))
      begin
        sel_src2 = 2'b10;
      end
      else
      begin
        sel_src2 = 2'b00;
      end
    end
  end

endmodule
```

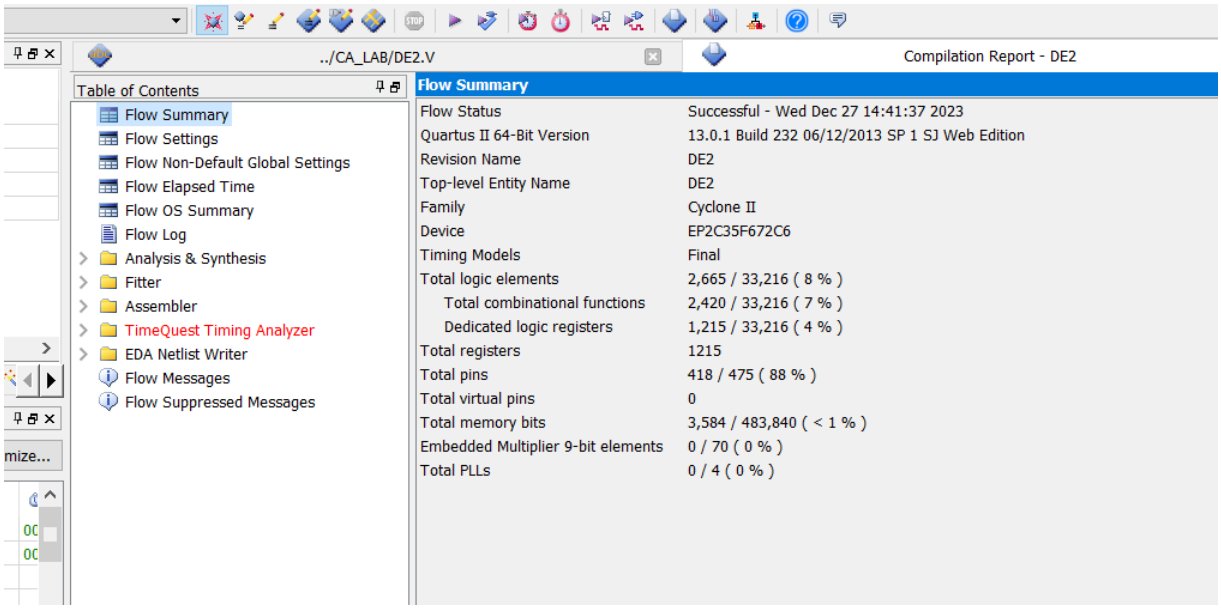
با آوردن کد بر روی Quartus مقدار بهبود هزینه سخت افزاری و پرفورمنس قابل مشاهده میشود:

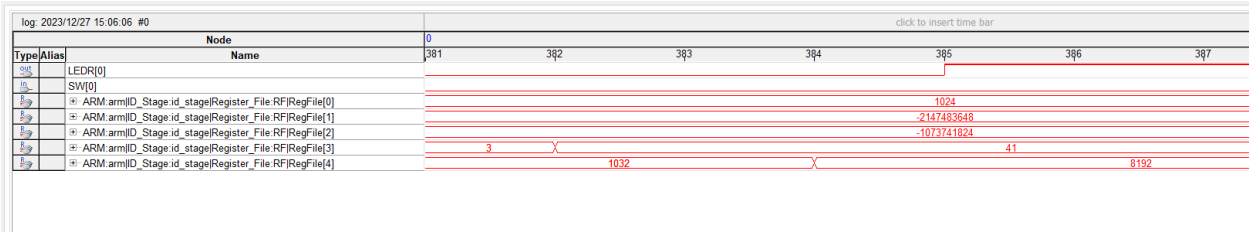
قبل از Forward:





پس از افزودن Forward:





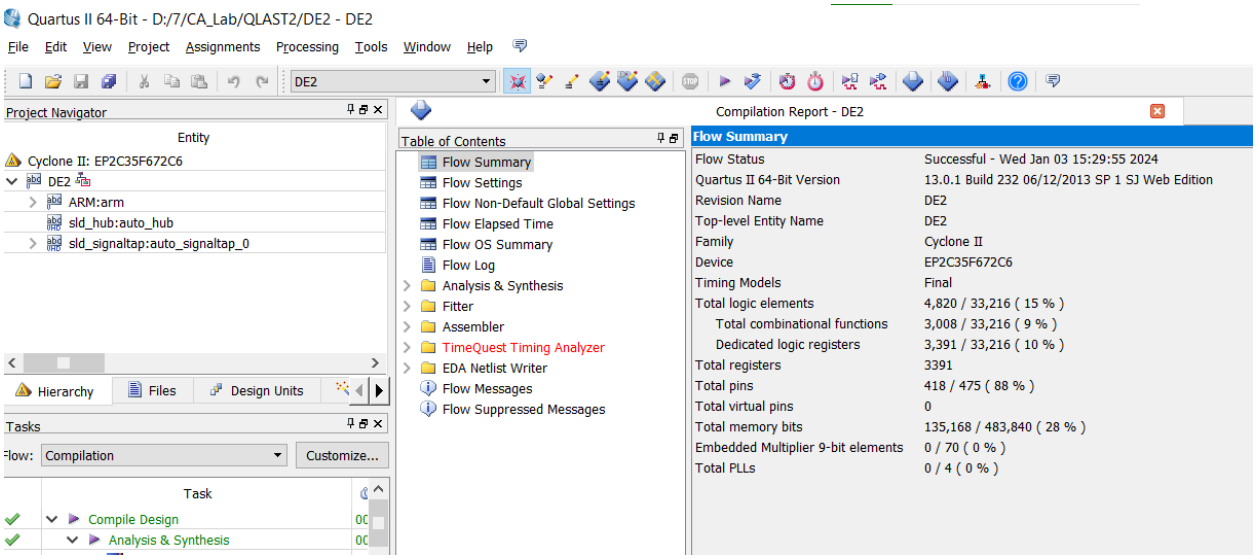
در قسمت اول چون مجبور به استفاده از frequency divider شدیم پس زمان کل برابر 280*2 = 560 و در حالت با فوروارد برابر 384 میباشد که همانطور که واضح است حدود 1.5 برابر افزایش سرعت داشته ایم. و همینطور مقدار سخت افزار مورد نیاز نیز تقریبا نصف شده است.

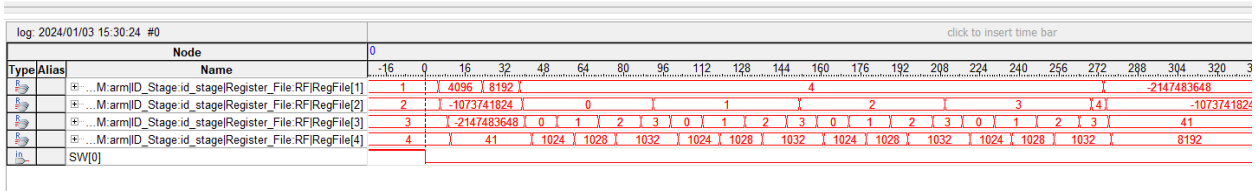
قابل ذکر است که نتایج سنتز مراحل مختلف را به جمع بندی نهایی ARM موکول کردیم چون دیتای آن ها را از دست دادیم اما چون در نهایت پس از forwarding همانطور که میبینم عملیات سورت مورد نظر انجام میشود نشان از کارایی مناسب بخش های قبل دارد.

بخش SRAM:

ما برای حافظه در قسمت MEMORY از تعدادی رجیستر محدود استفاده کرده ایم که در حالت واقعی پردازنده ما محتوا را از یک حافظه خارجی می خواند و روی آن می نویسد، پس در این بخش برای تکمیل کار خود، فضای تعامل با یک حافظه رم بر روی FPGA را فراهم میکنیم. تغییرات لازم بر روی قسمت Memory انجام شده و در بخش ها قبل آورده شده است.

نتایج کوارتس بصورت زیر میباشد:





همانطور که دیده میشود عملکرد افت شدیدی میکند که بدلیل تاخیر در پاسخگویی پس از استفاده از حافظه خارجی میباشد.

قسمت امتیازی SRAM:

برای بهبود کارایی بجز استفاده از کش که در بخش بعد ذکر شده است، میتوان از تکنیک out-of-order execution استفاده کرد، تکنیکی است که به پردازنده اجازه می‌دهد تا دستورالعمل‌ها را به ترتیبی غیر از ترتیب برنامه اجرا کند.بدین صورت که عملیات مربوط به یک سری داده که بسیار با هم ارتباط دارند را در سرتاسر کد

بررسی کرده و عملیات های مربوط به ان ها را یه یکباره و پشت هم انجام داد تا اگر در این میان مجبور به حذف بعضی از این داده ها از کش شدیم بدلیل اجرای بقیه عملیات ها، اگر وابستگی داده ای نداشته باشیم، در این حالت مجبور نیستیم که چند بار داده ها را وارد کش کنیم و دوباره ان ها را اوررایت کنیم. استفاده از پردازنده های موازی نیز راهکار دیگری برای بهبود سرعت میباشد.

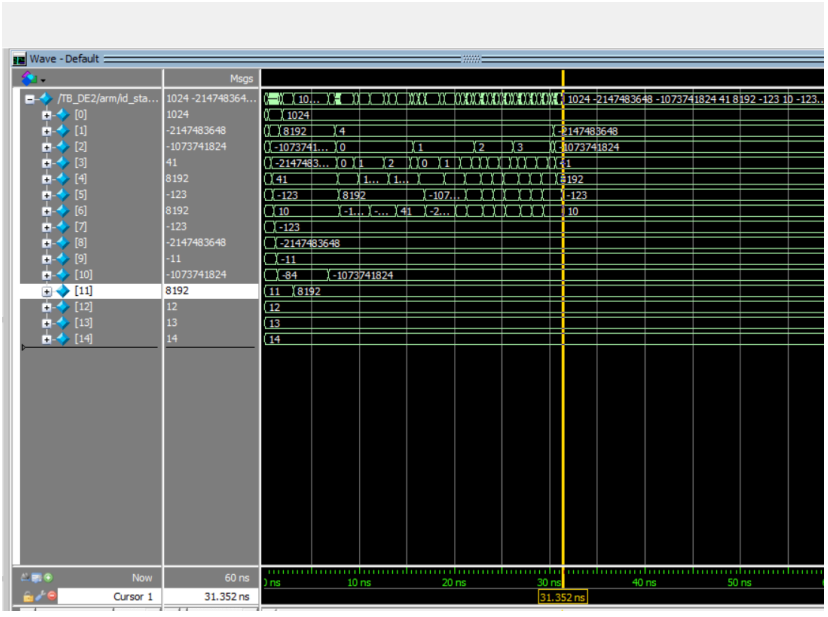
بخش CACHE:

مشاهده کردیم که با اضافه کردن حافظه خارجی بطور مشهودی سرعت پردازنده پایین می آید زیرا کار با حافظه چندین سیکل طول میکشد و زمان بر است، برای رفع این مشکل یک حافظه کوچکتر و سریع تر بنام Cache اضافه میشود که با استفاده از آن کمتر به سراغ حافظه رم می رویم و در یک سیکل دیتا ما آماده میشود و دوباره کارایی حافظه به حالت قبل خود باز میگردد. این قسمت در ممووری پیاده سازی شده که کد ان به شکل زیر است:

```
module Cache_CT(
        input clk, rst,
        input rdEn, wrEn,
        input [31:0] address,
        input [31:0] write_data,
        input sram_ready,
        input [63:0] sram_read_data,
        output [31:0] read_data,
        output ready,
        output sram_wr_en, sram_rd_en
    );

    // Cache
    reg [31:0] way0First  [0:63];
    reg [31:0] way0Second [0:63];
    reg [31:0] way1First  [0:63];
    reg [31:0] way1Second [0:63];
    reg [9:0] way0Tag [0:63];
    reg [9:0] way1Tag [0:63];
    reg [63:0] way0Valid;
    reg [63:0] way1Valid;
    reg [63:0] indexLru;

    // Address Decode
    wire [2:0] offset;
    wire [5:0] index;
    wire [9:0] tag;
    assign offset = address[2:0];
    assign index = address[8:3];
    assign tag = address[18:9];

    // Way Decode
    wire [31:0] dataWay0, dataWay1;
    wire [9:0] tagWay0, tagWay1;
    wire validWay0, validWay1;
    assign dataWay0 = (offset[2] == 1'b0) ? way0First[index] : way0Second[index];
    assign dataWay1 = (offset[2] == 1'b0) ? way1First[index] : way1Second[index];
    assign tagWay0 = way0Tag[index];
    assign tagWay1 = way1Tag[index];
    assign validWay0 = way0Valid[index];
    assign validWay1 = way1Valid[index];

    // Hit Controller
    wire hit;
    wire hitWay0, hitWay1;
    assign hitWay0 = (tagWay0 == tag && validWay0 == 1'b1);
    assign hitWay1 = (tagWay1 == tag && validWay1 == 1'b1);
    assign hit = hitWay0 | hitWay1;
```

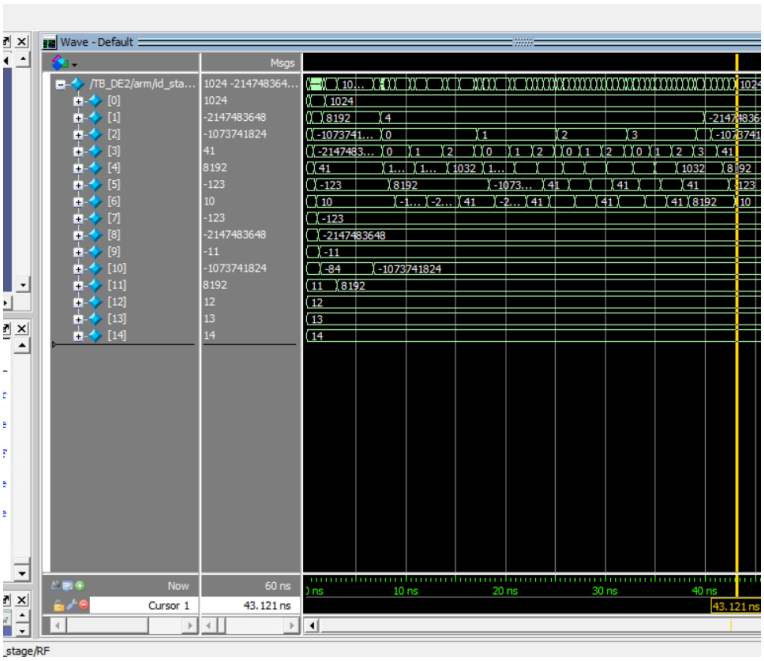
```
// Data Controller
wire [31:0] data;
wire [31:0] readDataQ;
assign data = hitWay0 ? dataWay0 :
              hitWay1 ? dataWay1 : 32'dz;
assign readDataQ = hit ? data :
                   sram_ready ? (offset[2] == 1'b0 ? sram_read_data[31:0] : sram_read_data[63:32]) : 32'bz;
assign read_data = rdEn ? readDataQ : 32'bz;
assign ready = sram_ready;

// Sram Controller
assign sram_rd_en = ~hit & rdEn;
assign sram_wr_en = wrEn;

always @(posedge clk) begin
    if (wrEn) begin
        if (hitWay0) begin
            way0Valid[index] = 1'b0;
            indexLru[index] = 1'b1;
        end
        else if (hitWay1) begin
            way1Valid[index] = 1'b0;
            indexLru[index] = 1'b0;
        end
    end
end

always @(posedge clk) begin
    if (rdEn) begin
        if (hit) begin
            // read_data = data;
            indexLru[index] = hitWay1;
        end
        else begin
            if (sram_ready) begin
                if (indexLru[index] == 1'b1) begin
                    {way0Second[index], way0First[index]} = sram_read_data;
                    way0Valid[index] = 1'b1;
                    way0Tag[index] = tag;
                    indexLru[index] = 1'b0;
                end
                else begin
                    {way1Second[index], way1First[index]} = sram_read_data;
                    way1Valid[index] = 1'b1;
                    way1Tag[index] = tag;
                    indexLru[index] = 1'b1;
                end
                // read_data = (offset[2] == 1'b0) ? sram_read_data[31:0] : sram_read_data[63:32];
            end
        end
    end
end

always @(posedge clk or posedge rst) begin
    if (rst) begin
        way0Valid = 64'd0;
        way1Valid = 64'd0;
        indexLru = 64'd0;
    end
end
endmodule
```

با Cache و Sram و Forwarding (دوره تناوب 0.1 نانو ثانیه)



تعداد سیکل ها: 310

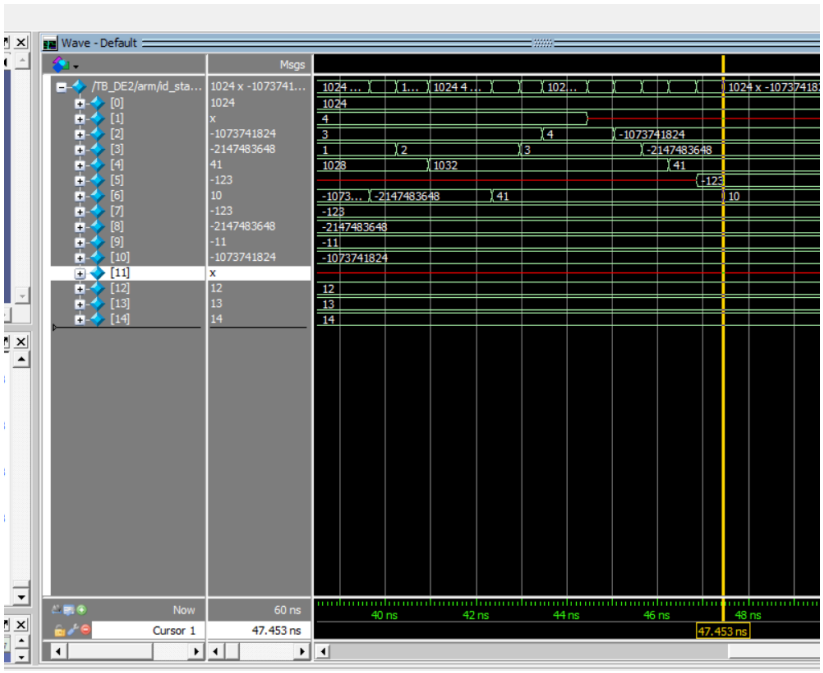
با Sram و Forwarding بدون Cache



تعداد سیکل ها: 430

همانطور که دیده میشود با استفاده از کش تعداد سیکل ها از 430 به 310 رسیده که نشانگر بهبود عملکرد خوبی میباشد.

با Sram و بدون Cache و Forwarding:



تعداد سیکل ها: 470
همانطور که انتظار میرفت این حالت تاخیری که ذکر شد را کاملا نشان میدهد و سیکل ها به 470 رسیده است.

قسمت نمره اضافی cache:

در حافظه نهان، الگوریتم LRU به کار می‌رود. در این الگوریتم، هنگامی که نیاز به جایگزینی داده با داده دیگری پیش می‌آید، سطری انتخاب می‌شود که به مدت زمان بیشتری استفاده نشده است. برای مدیریت این مسئله، در زمانی که داده‌ها از حافظه نهان خوانده می‌شوند، شمارنده LRU برابر با سطری قرار می‌گیرد که از آن داده‌ای خوانده شده است. وقتی نیاز به نوشتن داده جدید در حافظه نهان پیش می‌آید، ابتدا به بیت LRU نگاه می‌شود. اگر بیت LRU برابر با صفر باشد، یکی از دو سطر را برای جایگزینی انتخاب می‌کنیم؛ در غیر این صورت، یکی از دیگری را جایگزین می‌کنیم. سپس پس از جایگزینی، شمارنده LRU را برابر با سطری قرار می‌دهیم که در آن داده را نوشته‌ایم. برای به‌روزرسانی داده‌های موجود در حافظه نهان، تنها بیت معتبر را به صفر تغییر می‌دهیم و داده را مستقیماً در حافظه اصلی به‌روزرسانی می‌کنیم.

در رابطه با LRU لازم به ذکر است که این الگوریتم از دیگر الگوریتم‌های مدیریت حافظه نهان مانند FIFO و LFU نیز متمایز است. در LRU، تا زمانی که داده‌ای به کاربر داده نشود، این داده در حافظه نهان حفظ می‌شود تا به عنوان داده فعال باقی بماند.