به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده مهندسی برق و کامپیوتر

آزمایشگاه معماری کامپیوتر دانشکده مهندسی برق و کامپیوتر دانشگاه تهران

گزارش کار پیاده سازی پردازنده ARM

سجاد علیزاده **810197547**

حمیدرضا خدادادی **810197499**

بهار ۱۴۰۰

فهرست مطالب:

مرحله واکشی (Instruction Fetch)

در این مرحله واکشی دستور انجام می شود. ابتدا به معرفی زیر ماژول های استفاده شده می پردازیم:

- **ماژول MUX2_32:**

این ماژول برای انتخاب بین دو ورودی استفاده می شود. ورودی ها و خروجی آن به شرح زیر هستند:

1. ورودی های in1 و in2 که از بین آنها یکی برای قرار گرفتن روی خروجی انتخاب می شود.
2. ورودی sel که تعیین می کند کدام ورودی روی خروجی قرار گیرد.
3. خروجی out که خروجی این ماژول است.

توجه کنید که ورودی های in1 و in2 و خروجی out هر سه 32 بیتی هستند. پیاده سازی این ماژول به شکل زیر است:

همانطور که مشاهده می شود این ماژول توسط یک assign نوشته شده است، که اگر سیگنال sel صفر بود ورودی اول و اگر یک بود ورودی دوم روی خروجی قرار داده شود.

```
module MUX2_32(
    input [31:0] in1, in2,
    input sel,
    output [31:0] out
);
    assign out = sel ? in1 : in2;
endmodule
```

- **ماژول Adder:**

این ماژول برای جمع کردن دو عدد 32 بیتی استفاده می شود. به عنوان مثال در مرحله واکشی برای جمع کردن PC با 4 استفاده شده است. ورودیها و خروجی این ماژول به شرح زیر هستند:

1. ورودی های in1 و in2 که این دو مقدار با هم جمع می شوند.
2. خروجی result که برابر مجموع مقدار دو ورودی است.

پیاده سازی این ماژول به شکل زیر است:

```
module Adder_32bit(
    input [31:0] in1, in2,
    output [31:0] result
);
    assign result = in1 + in2;
endmodule
```

همانطور که مشاهده می شود این ماژول توسط یک assign نوشته شده است، که خروجی را به مجموع دو ورودی assign می کند.

- **ماژول InstructionMemory:**

این ماژول، حافظه ای است که دستورات از آن واکشی می شوند. ورودی و خروجی این ماژول به شرح زیر است:

1. ورودی address که آدرس دستور مورد نظر است.
2. خروجی instruction که دستور متناظر به آدرس ورودی است.

پیاده سازی این ماژول به شرح زیر است:

```
module InstructionMemory(
    input [31:0] address,
    output [31:0] instruction
);
    reg [31:0] MemByte[0:255];
    initial begin
```

```
        $readmemb("Program.data", MemByte);
    end
    assign instruction = MemByte[{2'b0, address[31:2]}];
endmodule
```

همانطور که مشاهده می شود یک حافظه درونی به نام MemByte ساخته میشود. سپس همه دستورات با استفاده از initial از یک فایل خوانده و در این حافظه درونی نوشته میشود. پس از آن خروجی با استفاده از یک assign تعیین میشود. توجه کنید آدرس دستورات در پردازنده به گونه ای هستند که باید آنها را دو واحد شیفت داد. در اصل همه مضرب چهار هستند پس برای دسترسی در حافظه درونی دو صفر سمت راست آنها در نظر گرفته نمیشود. همچنین توجه داشته باشید این مدار یک مدار ترکیبی است.

- **ماژول PCRegister:**

این ماژول در اصل یک رجیستر است که آدرس دستور بعدی را که قرار است اجرا شود در خود ذخیره می کند. ورودی ها و خروجی آن به شرح زیر هستند:

1. ورودی clk که کلاک پردازنده است.
2. ورودی rst که سیگنال reset پردازنده است.
3. ورودی load که تعیین میکند آیا ورودی روی خروجی قرار گیرد یا خیر.
4. ورودی in که ورودی رجیستر است.
5. خروجی out که خروجی رجیستر است.

پیاده سازی این ماژول به شکل زیر است:

```
module PCRegister(
    input clk, rst, load,
    input [31:0] in,
    output reg [31:0] out
);
```

```
    always @(posedge clk, posedge rst)
    begin
        if (rst) out <= 0;
        else if (load) out <= in;
    end
endmodule
```

همانطور که مشاهده می شود این ماژول با استفاده از یک always که به لبه بالا رونده سیگنال های clk و rst حساس است نوشته شده است. اگر سیگنال rst یک بود خروجی برابر صفر قرار داده می شود و رجیستر reset می شود. در غیر این صورت اگر سیگنال load یک بود مقدار خروجی برابر ورودی قرار داده می شود و اگر سیگنال load صفر بود خروجی مقدار قبلی خود را حفظ می کند.

- **ماژول IF_Stage:**

این ماژول بیانگر شمای کلی مرحله واکشی دستور است که در آن از زیر ماژول هایی که در قسمتهای قبل معرفی کردیم نمونه گیری میکنیم. ورودی ها و خروجی های آن به شرح زیر هستند:

1. ورودی های clk و rst که برای رجیستر PC مورد نیاز هستند.
2. ورودی freeze برای مواقعی که میخواهیم دستور جدید وارد پردازنده نشود. منبع این ورودی در ادامه بیان خواهد شد.
3. ورودی branch_taken که توسط واحد کنترل تولید میشود و بیانگر این است که آیا دستور ورودی از نوع پرش هست یا خیر که به تناسب آن آدرس صحیح در رجیستر PC قرار گیرد.
4. ورودی branch_address که بیانگر آدرس پرش است.
5. خروجی PC که به قسمت بعد فرستاده می شود.
6. خروجی instruction که دستور خوانده شده از حافظه دستور است و به مرحله بعد فرستاده می شود.

پیاده سازی این ماژول به شکل زیر است:

```
module IF_Stage (
    input clk, rst, freeze, Branch_taken,
    input [31:0] BranchAddr,
    output [31:0] PC, Instruction
);
    wire[31:0] pcIn, pcOut;

    MUX2_32 mux(BranchAddr, PC, Branch_taken, pcIn);
    PCRegister PCReg(clk, rst, ~freeze, pcIn, pcOut);
    Adder_32bit adder(pcOut, 32'd4, PC);
    InstructionMemory inst_mem(pcOut, Instruction);
endmodule
```

همانطور که مشاهده می شود، از سیگنال Branch_taken برای تشخیص پرش استفاده می شود. در صورتی که دستور پرش بود ورودی رجیستر PC به BranchAddr تغییر می کند. همچنین از سیگنال freeze به عنوان سیگنال load رجیستر استفاده شده است که در صورتی که این سیگنال یک شد در رجیستر PC داده جدید نوشته نشود. خروجی این رجیستر به ماژول حافظه دستور داده شده است و خروجی این ماژول به مرحله بعد داده می شود. همچنین مقدار ورودی رجیستر PC در شرایطی که پرش نداریم برابر مقدار PC به علاوه چهار قرار داده شده است که وظیفه عملیات جمع کردن را ماژول adder بر عهده دارد.

- **ماژول IFStageRegister:**

این ماژول یک نوع رجیستر است که در رجیسترهای بین مراحل واکشی و کدگشایی استفاده می شود. ورودیها و خروجی آن به شرح زیر هستند:

1. ورودی clk که کلاک پردازنده است.
2. ورودی rst که سیگنال reset پردازنده است.
3. ورودی flush که در صورت یک بودن خروجی رجیستر را برابر صفر می کند.

4. ورودی load که تعیین میکند آیا ورودی روی خروجی قرار گیرد یا خیر.
5. ورودی in که ورودی رجیستر است.
6. خروجی out که خروجی رجیستر است.

```
module IFStageRegister(
    input clk, rst, flush, load,
    input [31:0] in,
    output reg [31:0] out
);
    always @(posedge clk, posedge rst) begin
        if (rst) out <= 0;
        else if (load && flush) out <= 0;
        else if (load) out <= in;
    end
endmodule
```

همانطور که مشاهده می شود این ماژول با استفاده از یک always که به لبه بالا رونده سیگنال های clk و rst حساس است نوشته شده است. اگر سیگنال rst یک بود خروجی برابر صفر قرار داده می شود و رجیستر reset می شود. همچنین اگر سیگنالهای load و flush هر دو یک بودند خروجی flush می شود یعنی برابر صفر قرار داده می شود. در نهایت اگر سیگنال load یک بود و سیگنال های flush و rst هر دو صفر بودند خروجی برابر ورودی قرار داده می شود. با این نوع پیاده سازی ابتدا به rst و سپس به ترتیب به flush و load اولویت داده شده است.

- **ماژول IF_Stage_Reg:**

این ماژول واسط بین مراحل واکشی و کدگشایی است. ورودیها و خروجیهای این ماژول به شرح زیر هستند:

1. ورودیهای clk و rst که برای رجیستر PC مورد نیاز هستند.
2. ورودی freeze که برای جلوگیری از ورود دستور جدید به پردازنده استفاده می شود.

3. ورودی flush که برای خالی کردن رجیسترهای میانی به کار می رود.
4. ورودی های PC_in و Instruction_in که از سمت ماژول IF_Stage می آیند.
5. خروجی های PC و Instruction که به سمت مرحله واکشی می روند.

پیاده سازی این ماژول به شکل زیر است.

```
module IF_Stage_Reg (
    input clk, rst, freeze, flush,
    input [31:0] PC_in, Instruction_in,
    output [31:0] PC, Instruction
);
    IFStageRegister R1(clk, rst, flush, ~freeze, PC_in, PC);
    IFStageRegister R2(clk, rst, flush, ~freeze, Instruction_in, Instruction);
endmodule
```

همانطور که مشاهده می شود ورودیهای PC_in و Instruction_in به رجیستر دارای flush و freeze که در مرحله قبل اشاره شد وارد می شوند و خروجی این رجیسترها به عنوان خروجی ماژول به سمت مرحله کدگشایی می رود. توجه کنید هنگامی که سیگنال freeze صفر باشد، نوشتن در رجیسترها انجام نمی شود و پردازنده freeze می شود.

- **ماژول IF:**

در این ماژول اتصال بین ماژول های IF_Stage و IF_Stage_Reg انجام می شود و نکته خاصی در این ماژول وجود ندارد.

پیاده سازی آن به شکل زیر است:

```
module IF(
    input clk, rst, freeze, Branch_taken, flush,
    input [31:0] BranchAddr,
    output [31:0] PC, Instruction
);
    wire[31:0] PC_if, Instruction_if;

    IF_Stage instruction_fetch(clk, rst, freeze, Branch_taken, BranchAddr, PC_if, Instruction_if);
    IF_Stage_Reg instruction_fetch_registers(clk, rst, freeze, flush, PC_if, Instruction_if, PC, Instruction);
endmodule
```

همانطور که مشاهده می شود صرفا از ماژولهای IF_Stage و IF_Stage_Reg نمونه برداری شده است و سیگنال های مورد نیاز این دو ماژول به عنوان ورودی و سیگنال های مورد نیاز در مرحله کدگشایی به عنوان خروجی در نظر گرفته شده است که در اصل ورودی ها و خروجی های همین دو ماژول است.

مرحله کدگشایی (Instruction Decode)

در این مرحله کدگشایی دستورها انجام می شود. ابتدا به معرفی ماژولهای استفاده شده در این مرحله می پردازیم.

- **ماژول RegisterFile:**

این ماژول حاوی رجیسترهای عمومی پردازنده است که امکان خواندن از آنها و نوشتن در آنها را فراهم می کند. سیگنال های ورودی و خروجی آن به شرح زیر هستند:

1. ورودی clk و rst
2. ورودی های src1 و src2 که محتوای رجیستر متناظر با این شماره در خروجی قرار می گیرد.
3. ورودی Dest_wb که شماره رجیستری است که در آن نوشتن انجام شود.

4. ورودی Result_WB که محتوایی است که در رجیستر نوشته شود.
5. ورودی writeBackEn که در صورت یک بودن آن مقدار Result_WB در رجیستر متناظر با Dest_wb نوشته می شود.
6. خروجی های reg1 و reg2 که به ترتیب محتوای متناظر با رجیسترها با شماره src1 و src2 روی آن قرار می گیرد.

```
module RegisterFile (
    input clk, rst,
    input [3:0] src1, src2, Dest_wb,
    input [31:0] Result_WB,
    input writeBackEn,
    output [31:0] reg1, reg2
);
```

حال به بررسی پیاده سازی این ماژول می پردازیم:

```
    reg [31:0] R[0:14];
    integer i;
    initial begin
        for (i = 0; i < 15; i = i + 1)
            R[i] = i;
        end
```

در ابتدای پیاده سازی یک حافظه 15 تایی در نظر گرفته شده است که به عنوان رجیستر استفاده می شود و در ابتدا محتوای هر کدام از آنها برابر شماره رجیستر قرار داده می شود.

```
    assign reg1 = R[src1];
    assign reg2 = R[src2];
```

سپس مقادیر خروجی برابر شماره رجیسترهای ورودی قرار داده می شود. توجه کنید که این عملیات به صورت ترکیبی انجام می شود.

```
always @(negedge clk, posedge rst) begin
    if (rst) begin
        for (i = 0; i < 15; i = i + 1)
        R[i] = i;
    end
    if (writeBackEn)
        R[Dest_wb] <= Result_WB;
    end
```

در نهایت یک always نوشته شده که به لبه پایین رونده کلاک و لبه بالا رونده ریست حساس است و در آن عملیات نوشتن در رجیستر انجام می شود. ابتدا چک می شود اگر سیگنال ریست یک باشد مقادیر تمام رجیسترها برابر شماره آنها قرار داده می شود. سپس اگر سیگنال writeBackEn برابر یک باشد مقدار رجیستر شماره Dest_wb برابر Result_WB قرار داده می شود. توجه کنید برای جلوگیری از مخاطره داده ای نوشتن در رجیستر فایل با لبه پایین رونده کلاک انجام می شود.

- **ماژول ControlUnit:**

این ماژول برای تولید سیگنالهای کنترلی هر دستور به کار میرود. سیگنالهای ورودی و خروجی آن به شرح زیر هستند:

1. ورودی Sin که بیت بیستم دستور است.
2. ورودی opcode که بیتهای 21 تا 24 دستور است و نوع دستور را مشخص می کند.
3. ورودی mode که بیتهای 26 و 27 دستور است و بیانگر دسته دستور است.
4. خروجی WB_EN که نشان می دهد آیا در این دستور باید در خروجی نوشته شود یا خیر.
5. خروجی MEM_R_EN که نشان می دهد آیا در این دستور باید از حافظه اصلی خوانده شود یا خیر.
6. خروجی MEM_W_EN که نشان می دهد آیا در این دستور باید در حافظه اصلی نوشته شود یا خیر.
7. خروجی B که نشان میدهد دستور از نوع پرش هست یا خیر.

8. خروجی S که تعیین میکند آیا رجیستر وضعیت باید بروزرسانی شود یا خیر.
9. خروجی hasSrc1 که نشان می دهد در دستور src1 وجود دارد یا خیر.
10. خروجی EXE_CMD که عملیاتی را که باید در ALU انجام شود تعیین می کند.

توجه کنید این ماژول به صورت ترکیبی پیاده سازی می شود.

```
module ControlUnit (
    input SIn,
    input [3:0] opcode,
    input [1:0] mode,
    output reg WB_EN, MEM_R_EN, MEM_W_EN, B, S, hasSrc1,
    output reg [3:0] EXE_CMD
);
```

حال به پیاده سازی این ماژول می پردازیم:

```
    parameter STR = 1'b0;
    parameter LDR = 1'b1;

    parameter [1:0] ARTHMETIC_LOGIC = 2'b00;
    parameter [1:0] STR_LDR = 2'b01;
    parameter [1:0] BRANCH = 2'b10;

    parameter [3:0] AND = 4'b0000;
    parameter [3:0] EOR = 4'b0001;
    parameter [3:0] SUB = 4'b0010;
    parameter [3:0] ADD = 4'b0100;
```

```
parameter [3:0] ADC = 4'b0101;
parameter [3:0] SBC = 4'b0110;
parameter [3:0] TST = 4'b1000;
parameter [3:0] CMP = 4'b1010;
parameter [3:0] ORR = 4'b1100;
parameter [3:0] MOV = 4'b1101;
parameter [3:0] MVN = 4'b1111;
```

در ابتدا تمام پارامترها را تعیین کردیم تا خوانایی کد بیشتر شود.

```
always @(SIn, opcode, mode) begin
    {WB_EN, MEM_R_EN, MEM_W_EN, B, S, EXE_CMD, hasSrc1} = 10'b0;
```

سپس با استفاده از یک always که به ورودیها حساس است. در ابتدا تمام سیگنال ها را صفر می کنیم.

```
    case (mode)
        ARTHMETIC_LOGIC: begin
            case (opcode)
                MOV: begin
                    WB_EN = 1'b1;
                    EXE_CMD = 4'b0001;
                    S = SIn;
                end
                MVN: begin
                    WB_EN = 1'b1;
                    EXE_CMD = 4'b1001;
                    S = SIn;
```

```
            end
            ADD: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b0010;
                S = SIn;
                hasSrc1 = 1'b1;
            end
            ADC: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b0011;
                S = SIn;
                hasSrc1 = 1'b1;
            end
            SUB: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b0100;
                S = SIn;
                hasSrc1 = 1'b1;
            end
            SBC: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b0101;
                S = SIn;
                hasSrc1 = 1'b1;
            end
```

```
            AND: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b0110;
                S = SIn;
                hasSrc1 = 1'b1;
            end
            ORR: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b0111;
                S = SIn;
                hasSrc1 = 1'b1;
            end
            EOR: begin
                WB_EN = 1'b1;
                EXE_CMD = 4'b1000;
                S = SIn;
                hasSrc1 = 1'b1;
            end
            CMP: begin
                EXE_CMD = 4'b0100;
                S = 1'b1;
                hasSrc1 = 1'b1;
            end
            TST: begin
                EXE_CMD = 4'b0110;
```

```
                S = 1'b1;
                hasSrc1 = 1'b1;
            end
        endcase
    end
     STR_LDR: begin
        case (SIn)
            STR: begin
                MEM_W_EN = 1'b1;
                EXE_CMD = 4'b0010;
                S = 1'b0;
                hasSrc1 = 1'b1;
            end
            LDR: begin
                MEM_R_EN = 1'b1;
                WB_EN = 1'b1;
                EXE_CMD = 4'b0010;
                S = 1'b1;
                hasSrc1 = 1'b1;
            end
        endcase
    end
     BRANCH: begin
        B = 1'b1;
    end
```

سپس بر اساس mode تصمیم گیری می کنیم. اگر mode دستور از نوع اعمال ریاضی بود بر اساس opcode آنها سیگنال های خروجی را تعیین می کنیم. در تمام آنها WB_EN و hasSrc1 یک می شوند و مقدار EXE_CMD بر اساس نوع دستور تعیین می شود. همچنین مقدار لود رجیستر وضعیت برابر ورودی موجود در دستور قرار داده می شود. در دستورات MOV و MVN سیگنال hasSrc1 یک نمی شود زیرا این دستورات یک اپرند دارند.

اگر mode دستور از نوع کار با حافظه بود بر اساس ورودی Sin تعیین می شود از نوع LDR است یا STR و در هر کدام سیگنال های مورد نظر مقداردهی می شوند. توجه کنید در LDR سیگنال WB_EN نیز یک می شود زیرا نتیجه باید در رجیستر فایل نوشته شود ولی این مقدار در STR یک نمی شود زیرا نوشتن در رجیستر فایل ندارد. در نهایت نیز اگر دستور از نوع پرش بود سیگنال B یک می شود.

- **ماژول ConditionCheck:**

این ماژول برای بررسی برقرار بودن شرط اجرای دستورات در پردازنده به کار می رود. سیگنال های ورودی و خروجی به شرح زیر هستند:

1. ورودی condition که بیانگر بیت های مربوط به شرط از دستور ورودی است.
2. ورودی SR که شامل بیت های مربوط به ثبات وضعیت است.
3. خروجی out که نشان می دهد شرط برقرار بوده یا خیر.

```
module ContidionCheck (
    input[3:0] condition, SR,
    output reg out
);
```

جدول حالات شرط ها بدین صورت است:

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 ۳- کد شرط دستورات

حال به پیاده سازی این ماژول می پردازیم:

```
parameter [3:0] EQ = 4'b0000;
parameter [3:0] NE = 4'b0001;
parameter [3:0] CS_HS = 4'b0010;
parameter [3:0] CC_LO = 4'b0011;
parameter [3:0] MI = 4'b0100;
parameter [3:0] PL = 4'b0101;
parameter [3:0] VS = 4'b0110;
```

```
parameter [3:0] VC = 4'b0111;
parameter [3:0] HI = 4'b1000;
parameter [3:0] LS = 4'b1001;
parameter [3:0] GE = 4'b1010;
parameter [3:0] LT = 4'b1011;
parameter [3:0] GT = 4'b1100;
parameter [3:0] LE = 4'b1101;
parameter [3:0] AL = 4'b1110;
```

ابتدا پارامترهای مربوط را تعیین می کنیم. این کار در افزایش خوانایی کمک خواهد کرد.

```
wire Z, C, N, V;
assign {Z, C, N, V} = SR;
```

سپس wire هایی تعریف کردیم و مقدار رجیستر وضعیت را در آن ریختیم تا خوانایی کد افزایش یابد.

```
always @(condition, SR) begin
    out = 1'b0;
```

سپس یک always جهت تعیین خروجی قرار می دهیم. ابتدای این always مقدار خروجی را صفر قرار می دهیم.

```
    case(condition)
      EQ: out = Z;
      NE: out = ~Z;
      CS_HS: out = C;
      CC_LO: out = ~C;
      MI: out = N;
      PL: out = ~N;
```

```
        VS: out = V;
        VC: out = ~V;
        HI: out = C & ~Z;
        LS: out = ~C & Z;
        GE: out = (N & V) | (~N & ~V);
        LT: out = (N & ~V) | (~N & V);
        GT: out = ~Z & ((N & V) | (~N & ~V));
        LE: out = Z & ((N & ~V) | (~N & V));
        AL: out = 1'b1;
        default: out = 1'b0;
    endcase
```

سپس یک case قرار داده و به ازای شروط مختلف بررسی می کنیم که آیا آن شرط برقرار است یا خیر. در صورت برقرار نبودن شرط باعث می شود یک حباب در پردازنده ایجاد شود.

- **ماژول ID_Stage:**

این ماژول بیانگر شمای کلی مرحله کدگشایی است که از ماژولهایی که در قسمتهای قبل گفته شد نمونه گیری می شود. سیگنال های ورودی و خروجی این ماژول به شرح زیر هستند:

1. ورودی های clk و rst
2. ورودی Instruction که دستور خوانده شده از حافظه است و از مرحله واکشی دستور به این مرحله رسیده است.
3. ورودی PCin که شمارنده برنامه است و از مرحله واکشی دستور به این مرحله رسیده است.
4. ورودی Result_WB که برابر مقداری است که قرار است در رجیستر فایل نوشته شود.
5. ورودی writeBackEn که تعیین می کند آیا مقداری باید در رجیستر فایل نوشته شود یا خیر.
6. ورودی Dest_wb که برابر است با شماره رجیستری که در آن داده نوشته می شود.

7. ورودی hazard که در صورتی که مخاطره داشته باشیم یک میشود و از آن استفاده میکنیم تا پردازنده را فریز کنیم.
8. ورودی SR که برابر مقدار رجیستر وضعیت است.
9. خروجی های WB_EN, MEM_R_EN, MEM_W_EN, B, S, hasSrc1, EXE_CMD که سیگنال های کنترلی تولید شده مربوط به دستور هستند. این سیگنال ها توسط کنترلر تولید می شوند و در قسمت کنترلر توضیح داده شده اند.
10. خروجی های Val_Rn و Val_Rm که مقادیر خروجی از رجیستر فایل هستند.
11. خروجی PC که شمارنده دستور است که به مرحله اجرا فرستاده می شود.
12. خروجی imm که فوری بودن دستور را مشخص می کند.
13. خروجی Shift_operand که بیانگر بیت های 0 تا 11 دستور است.
14. خروجی Signed_imm_24 که بیانگر بیت های 0 تا 23 دستور است.
15. خروجی Dest که برابر شماره رجیستری است که نتیجه دستور در آن نوشته می شود.
16. خروجی های src1 و src2 که برابر شماره رجیستر هایی هستند که داده آنها خوانده شده است.
17. خروجی Two_src که نشان می دهد دستور دارای یک اپرند ورودی است یا دو اپرند ورودی.

```
module ID_Stage (
    input clk, rst,
    input [31:0] Instruction, PCin,
    input [31:0] Result_WB,
    input writeBackEn,
    input [3:0] Dest_wb,
    input hazard,
    input [3:0] SR,
    output WB_EN, MEM_R_EN, MEM_W_EN, B, S, hasSrc1,
    output [3:0] EXE_CMD,
    output [31:0] Val_Rn, Val_Rm, PC,
```

```
    output imm,
    output [11:0] Shift_operand,
    output [23:0] Signed_imm_24,
    output [3:0] Dest,
    output [3:0] src1, src2,
    output Two_src
);
```

حال به پیاده سازی این ماژول می پردازیم:

```
    assign PC = PCin;
    assign imm = Instruction[25];
    assign Shift_operand = Instruction[11:0];
    assign Signed_imm_24 = Instruction[23:0];
    assign Dest = Instruction[15:12];
```

در ابتدا خروجی های مشخص را به شکل بالا مقدار دهی می کنیم.

```
    wire condition_check, control_unit_mux_select;
    wire WB_EN_temp, MEM_R_EN_temp, MEM_W_EN_temp, B_temp, S_temp, hasSrc1_temp;
    wire [3:0] EXE_CMD_temp;
```

سپس وایرهایی را به شکل بالا تعریف می کنیم. دلیل استفاده از temp برای سیگنال های کنترلی این است که در صورتی که شرط دستور برقرار نبود بتوانیم خروجی های اصلی را صفر کنیم و یک حباب وارد پردازنده کنیم.

```
    ControlUnit CU(Instruction[20], Instruction[24:21],
        Instruction[27:26], WB_EN_temp, MEM_R_EN_temp,
        MEM_W_EN_temp, B_temp, S_temp, hasSrc1_temp, EXE_CMD_temp);
```

```
or twoSrcGen(Two_src, MEM_W_EN, ~imm);
ContidionCheck CC(Instruction[31:28], SR, condition_check);
```

سپس یک نمونه از کنترلر گرفتیم تا سیگنال های کنترلی را تولید کند. بعد از آن توسط یک ماژول or خروجی Two_src را تولید کردیم. این خروجی در صورتی یک است که دستور از نوع نوشتن باشد یا بیت فوری برابر صفر باشد. در نهایت نیز یک نمونه از ماژول ConditionCheck گرفتیم و با دادن ورودی های مناسب به آن سیگنال condition_check را تولید کردیم که در صورت صفر بودن بیان می کند باید حباب در پردازنده ایجاد شود.

```
or mux_select(control_unit_mux_select, ~condition_check, hazard);
```

سپس سیگنال select برای تصمیم گیری اینکه حباب وارد سیستم شود یا خیر را تولید می کنیم. اگر شرط برقرار نباشد یا مخاطره رخ داده باشد این سیگنال یک می شود و باعث میشود یک حباب در پردازنده ایجاد شود.

```
assign WB_EN = control_unit_mux_select ? 0 : WB_EN_temp;
assign MEM_R_EN = control_unit_mux_select ? 0 : MEM_R_EN_temp;
assign MEM_W_EN = control_unit_mux_select ? 0 : MEM_W_EN_temp;
assign B = control_unit_mux_select ? 0 : B_temp;
assign S = control_unit_mux_select ? 0 : S_temp;
assign EXE_CMD = control_unit_mux_select ? 0 : EXE_CMD_temp;
assign hasSrc1 = control_unit_mux_select ? 0 : hasSrc1_temp;
```

حال خروجی های کنترلی مقدار دهی می شوند. اگر سیگنال تولید شده در مرحله قبل یک باشد یعنی باید حباب وارد سیستم شود. زیرا یا شرط دستور برقرار نبوده یا دچار مخاطره بوده ایم. در غیر این صورت همان سیگنال های تولید شده در کنترلر به عنوان خروجی تعیین می شود.

```
assign src1 = Instruction[19:16];
assign src2 = MEM_W_EN ? Instruction[15:12]:Instruction[3:0];
```

در این قسمت سیگنالهای src1 و src2 تعیین می شود. توجه کنید مقدار src2 به اینکه دستور از نوع نوشتن در رجیستر است ربط دارد و در صورتی که دستور از نوع نوشتن نباشد این مقدار برابر بیت های صفر تا سه دستور قرار داده می شود تا در قسمت اجرا مورد استفاده قرار گیرد.

```
RegisterFile RF(clk, rst, src1, src2, Dest_wb, Result_WB, writeBackEn, Val_Rn, Val_Rm);
```

در نهایت نیز یک نمونه از رجیستر فایل گرفته می شود و ورودی ها و خروجی های متناظر آن به آن متصل می شود تا خروجی های مورد نیاز در مراحل بعدی تولید شود.

- **ماژولهای IDStageRegisterXbit:**

این ماژول ها رجیستر هستند که در میان مرحله کدگشایی و مرحله اجرا استفاده می شوند. تفاوت آنها در تعداد بیت است که به ازای X های 1 و 4 و 12 و 24 و 32 تعریف شده هستند. ورودی و خروجی این ماژول ها به شرح زیر هستند:

1. ورودی clk که کلاک پردازنده است.
2. ورودی rst که سیگنال reset پردازنده است.
3. ورودی flush که در صورت یک بودن خروجی رجیستر را برابر صفر می کند.
4. ورودی load که تعیین میکند آیا ورودی روی خروجی قرار گیرد یا خیر.
5. ورودی in که ورودی رجیستر است.
6. خروجی out که خروجی رجیستر است.

```
module IDStageRegister1bit(
    input clk, rst, flush, load,
    input in,
    output reg out
);
    always@(posedge clk, posedge rst) begin
        if (rst) out <= 0;
```

```
        else if (load && flush) out <= 0;
        else if (load) out <= in;
    end
endmodule
```

به عنوان نمونه رجیستر یک بیتی آورده شده است. همانطور که مشاهده می شود این ماژول با استفاده از یک always که به لبه بالا رونده سیگنال های clk و rst حساس است نوشته شده است. اگر سیگنال rst یک بود خروجی برابر صفر قرار داده می شود و رجیستر reset می شود. همچنین اگر سیگنالهای load و flush هر دو یک بودند خروجی flush می شود یعنی برابر صفر قرار داده میشود. در نهایت اگر سیگنال load یک بود و سیگنال های flush و rst هر دو صفر بودند خروجی برابر ورودی قرار داده می شود. با این نوع پیاده سازی ابتدا به rst و سپس به ترتیب به flush و load اولویت داده شده است.

- **ماژول ID_Stage_Reg:**

این ماژول واسط بین مراحل کدگشایی و اجرا است. ورودی ها و خروجی های این ماژول به شرح زیر هستند:

1. ورودی های clk و rst
2. ورودی flush که در صورت یک بودن خروجی این رجیسترهای میانی را صفر می کند.
3. ورودی freeze که در صورت یک بودن باعث فریز شدن این رجیسترها می شود و باعث می شود دستور به جلو حرکت نکند.
4. سایر ورودی ها و خروجی ها همان سیگنال های تولید شده در ماژول ID_Stage هستند که صرفا در رجیستر قرار گرفته اند. یعنی هر ورودی وارد یک رجیستر شده است و خروجی ماژول خروجی متناظر با آن ورودی است که از رجیستر خارج می شود.

```
module ID_Stage_Reg (
    input clk, rst, flush, freeze,
    input WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,
    input B_IN, S_IN, hasSrc1_IN,
```

```
    input [3:0] EXE_CMD_IN, src1_IN, src2_IN,
    input [31:0] PC_IN,
    input [31:0] Val_Rn_IN, Val_Rm_IN,
    input imm_IN,
    input [11:0] Shift_operand_IN,
    input [23:0] Signed_imm_24_IN,
    input [3:0] Dest_IN,
    output WB_EN, MEM_R_EN, MEM_W_EN, B, S, hasSrc1,
    output [3:0] EXE_CMD, src1, src2,
    output [31:0] PC,
    output [31:0] Val_Rn, Val_Rm,
    output imm,
    output [11:0] Shift_operand,
    output [23:0] Signed_imm_24,
    output [3:0] Dest
);
```

پیاده سازی این ماژول به شکل زیر است:

```
    IDStageRegister32bit R1(clk, rst, flush, freeze, PC_IN, PC);
    IDStageRegister32bit R2(clk, rst, flush, freeze, Val_Rm_IN, Val_Rm);
    IDStageRegister32bit R3(clk, rst, flush, freeze, Val_Rn_IN, Val_Rn);
    IDStageRegister1bit R4(clk, rst, flush, freeze, WB_EN_IN, WB_EN);
    IDStageRegister1bit R5(clk, rst, flush, freeze, MEM_R_EN_IN, MEM_R_EN);
    IDStageRegister1bit R6(clk, rst, flush, freeze, MEM_W_EN_IN, MEM_W_EN);
    IDStageRegister1bit R7(clk, rst, flush, freeze, B_IN, B);
```

```
IDStageRegister1bit R8(clk, rst, flush, freeze, S_IN, S);
IDStageRegister1bit R16(clk, rst, flush, freeze, hasSrc1_IN, hasSrc1);
IDStageRegister4bit R9(clk, rst, flush, freeze, EXE_CMD_IN, EXE_CMD);
IDStageRegister1bit R10(clk, rst, flush, freeze, imm_IN, imm);
IDStageRegister12bit R11(clk, rst, flush, freeze, Shift_operand_IN, Shift_operand);
IDStageRegister24bit R12(clk, rst, flush, freeze, Signed_imm_24_IN, Signed_imm_24);
IDStageRegister4bit R13(clk, rst, flush, freeze, Dest_IN, Dest);
IDStageRegister4bit R14(clk, rst, flush, freeze, src1_IN, src1);
IDStageRegister4bit R15(clk, rst, flush, freeze, src2_IN, src2);
```

همانطور که مشاهده می شود هر ورودی وارد یک رجیستر شده است و خروجی متناظر از آن از رجیستر خارج شده است. همچنین هر استفاده از هر رجیستر با توجه به تعداد بیتهای آن انجام شده است. از سیگنال freeze نیز برای load رجیستر استفاده شده است تا فریز شدن پردازنده کنترل شود.

- **ماژول ID:**

در این ماژول اتصال بین ماژول های ID_Stage و ID_Stage_Reg انجام می شود و نکته خاصی در این ماژول وجود ندارد. پیاده سازی آن به شکل زیر است:

```
module ID(
  input clk, rst, flush, freeze,
  // From IF
  input[31:0] Instruction, PCin,
  // From WB
  input writeBackEn,
  input [3:0] Dest_wb,
  input [31:0] Result_WB,
```

```
    // From hazard
    input hazard,
    // From Status Register
    input [3:0] SR,
    output [3:0] src1, src2, src1_reg, src2_reg,
    output Two_src,
    output WB_EN, MEM_R_EN, MEM_W_EN, B, S, hasSrc1,
    output [3:0] EXE_CMD,
    output [31:0] PC,
    output [31:0] Val_Rn, Val_Rm,
    output imm,
    output [11:0] Shift_operand,
    output [23:0] Signed_imm_24,
    output [3:0] Dest
);
    wire WB_EN_temp, MEM_R_EN_temp, MEM_W_EN_temp, B_temp, S_temp, imm_temp,
hasSrc1_temp;
    wire [3:0] EXE_CMD_temp, Dest_temp;
    wire [31:0] Val_Rn_temp, Val_Rm_temp, PC_temp;
    wire [11:0] Shift_operand_temp;
    wire [23:0] Signed_imm_24_temp;

    ID_Stage instruction_decode(
        clk, rst,
        Instruction, PCin,
```

```
    Result_WB,
    writeBackEn,
    Dest_wb,
    hazard,
    SR,
    WB_EN_temp, MEM_R_EN_temp, MEM_W_EN_temp, B_temp, S_temp, hasSrc1_temp,
    EXE_CMD_temp,
    Val_Rn_temp, Val_Rm_temp, PC_temp,
    imm_temp,
    Shift_operand_temp,
    Signed_imm_24_temp,
    Dest_temp,
    src1, src2,
    Two_src
);

ID_Stage_Reg instruction_decode_registers(
    clk, rst, flush, freeze,
    WB_EN_temp, MEM_R_EN_temp, MEM_W_EN_temp,
    B_temp, S_temp, hasSrc1_temp,
    EXE_CMD_temp, src1, src2,
    PC_temp, Val_Rn_temp, Val_Rm_temp,
    imm_temp,
    Shift_operand_temp,
    Signed_imm_24_temp,
```

```
    Dest_temp,
    WB_EN, MEM_R_EN, MEM_W_EN, B, S, hasSrc1,
    EXE_CMD, src1_reg, src2_reg,
    PC,
    Val_Rn, Val_Rm,
    imm,
    Shift_operand,
    Signed_imm_24,
    Dest
  );
endmodule
```

همانطور که مشاهده می شود صرفا از ماژول های ID_Stage و ID_Stage_Reg نمونه برداری شده است و سیگنال های مورد نیاز این دو ماژول به عنوان ورودی و سیگنال های مورد نیاز در مرحله اجرا به عنوان خروجی در نظر گرفته شده است که در اصل ورودی ها و خروجی های همین دو ماژول است. نقش تمام این سیگنال های ورودی و خروجی در قسمت توضیحات ماژول های ID_Stage و ID_Stage_Reg مفصلا شرح داده شده است.

مرحله اجرا (Execution)

EXE.v

```
1   `timescale 1ns/1ns
2
3   module EXE(
4       input clk, rst, freeze,
5       input imm, MEM_R_EN_in, MEM_W_EN_in, WB_EN_in,
6       input [1:0] Sel_src1, Sel_src2,
7       input [3:0] EXE_CMD, dst_in, SRin,
8       input [11:0] Shift_operand,
9       input [23:0] Signed_imm_24,
10      input [31:0] PC_in, Val_Rn, Val_Rm, ALU_MEM_val, WB_Val,
11      output MEM_R_EN, MEM_W_EN, WB_EN,
12      output [3:0] Dest, SR,
13      output [31:0] ALU_result, Branch_Address, Val_Rm_out
14  );
```

ماژول EXE در این بخش از دو زیر ماژول EXE_Stage و EXE_reg تشکیل شده است.

ورودی های ماژول EXE همان ورودی های ماژول EXE_Stage خواهد بود که خروجی زیر ماژول EXE_Stage به ورودی زیر ماژول EXE_reg وصل می شود. و در نهایت خروجی زیر ماژول EXE_reg همان خروجی نهایی بخش Execution خواهد بود.

EXE.v

```
15
16      wire [31:0] ALU_result_temp, Val_Rm_temp;
17      wire [3:0] dest_temp;
18
19      EXE_Stage execute(
20          clk, rst, imm, MEM_R_EN_in, MEM_W_EN_in,
21          Sel_src1, Sel_src2,
22          EXE_CMD, dst_in, SRin,
23          Shift_operand,
24          Signed_imm_24,
25          PC_in, Val_Rn, Val_Rm, ALU_MEM_val, WB_Val,
26          SR, dest_temp,
27          ALU_result_temp, Branch_Address, Val_Rm_temp
28      );
29
30      EXE_reg execute_registers(
31          clk, rst, freeze,
32          WB_EN_in, MEM_R_EN_in, MEM_W_EN_in,
33          dest_temp,
34          ALU_result_temp, Val_Rm_temp,
35          WB_EN, MEM_R_EN, MEM_W_EN,
36          Dest,
37          ALU_result, Val_Rm_out
38      );
39
40  endmodule
```

در ادامه به توضیح هر کدام از زیر ماژول ها خواهیم پرداخت:

- **ماژول EXE_Stage:**

EXE_Stage.v

```
1   `timescale 1ns/1ns
2
3   module EXE_Stage(
4       input clk, rst, imm, MEM_R_EN, MEM_W_EN,
5       input [1:0] Sel_src1, Sel_src2,
6       input [3:0] EXE_CMD, dst_in, SR_in,
7       input [11:0] Shifte_operand,
8       input [23:0] Signed_imm_24,
9       input [31:0] PC_in, Val_Rn, Val_Rm, ALU_MEM_val, WB_Val,
10      output [3:0] SR_out, dst_out,
11      output [31:0] ALU_result, Branch_Address, Val_Rm_out
12  );
```

مقدار imm که از بخش Decode آمده، در صورت صفر بودن، برای shift_operand نیاز به شیفت فوری است و در صورت یک بودن نیاز به 32 بیت عدد فوری است. دو مقدار Sel_src1 و Sel_src2 برای انتخاب ورودی دو تا عدد ورودی ALU استفاده می شود و مقدار این دو selector از ForwardingUnit خروجی می آید. دو مقدار MEM_R_EN و MEM_W_EN که به ترتیب برای خواندن از حافظه و نوشتن در حافظه استفاده می شوند و از ControlUnit مرحله Decode خروجی می آید و به مرحله Memory می رود. ورودی EXE_CMD که مشخص کننده case در ALU است. مقدار SR_in مقدار بیت های Status_Reg را در خود ذخیره دارد. مقدار SR_out مقدار های به روز شده در ALU برای بیت های Status_Reg را به خروجی می برد. مقدار Shift_opernad بیانگر عملوند عملیات شیفت است. مقدار Signed_imm_24 بیانگر مقدار فوری است. مقدار PC_in شمارنده برنامه است و از بخش Decode به خروجی آمده است. مقدار dst_in بیانگر مقصد نوشتن برای حافظه است و از مرحله Decode به خروجی آمده است و از طریق dst_out به مرحله Memory می رود. مقدار Val_Rn و Val_Rm بیانگر عملوند های دستور هستند و به ترتیب مقدار های خروجی اول و دوم رجیستر فایل را در خود دارند. مقدار ALU_MEM_Val بیانگر مقداری است که از مرحله Memory به این بخش آمده است و مقدار WB_Val بیانگر مقداری است که از مرحله Write Back به این بخش آمده است؛ این دو مقدار در دو تا مالتی پلکسر های پشت ورودی های اول و دوم ALU به کار رفته اند. مقدار ALU_result بیانگر خروجی ALU است. مقدار Branch_Address بیانگر آدرس پرش است.

```
17      assign dst_out = dst_in;
18      assign C = SR_in[2];
19      assign Signed_imm_32 = {{6{Signed_imm_24[23]}}, Signed_imm_24, 2'b0};
20
21      MUX3_32 mux3_32_1(Val_Rn, ALU_MEM_val, WB_Val, Sel_src1, mux3_32_1_out);
22      MUX3_32 mux3_32_2(Val_Rm, ALU_MEM_val, WB_Val, Sel_src2, mux3_32_2_out);
23
24      or is_memory(is_memory_command, MEM_R_EN, MEM_W_EN);
25      Val2Generator val2_generator(imm, is_memory_command, Shifte_operand, mux3_32_2_out, Val2);
26      ALU alu(mux3_32_1_out, Val2, C, EXE_CMD, SR_out, ALU_result);
27      Adder_32bit pc_adder(PC_in, Signed_imm_32, Branch_Address);
28
29      assign Val_Rm_out = mux3_32_2_out;
30
31  endmodule
```

این ماژول از یک ALU و یک Val2Generator و یک گیت or و یک adder تشکیل شده است.

آدرس Branch توسط Adder با جمع مقدار های PC_in و sign extend شده ی Signed_imm_24 به دست می آید.

گیت or برای مشخص شدن مقدار سیگنال is_memory_command استفاده می شود.

- **ماژول ALU:**

```
ALU.v
1   `timescale 1ns/1ns
2
3   module ALU(
4       input signed [31:0] ALU_in1, ALU_in2,
5       input C_in,
6       input [3:0] ALU_command,
7       output wire [3:0] SR,
8       output reg [31:0] ALU_result
9   );
```

این ماژول دو ورودی 32 بیت علامت دار دریافت می کند که ورودی اولش همان Val_Rn خروجی از بخش Decode است و ورودی دومش توسط ماژول Val2Generator در همین بخش Execution تولید می شود.

ورودی دیگرش بیت C_in یا همان بیت carry است که از ماژول Status_Reg تولید می شود و به بخش Decode وارد می شود و از آن جا به بخش Execution وارد می شود.

ورودی دیگر 4 بیت ALU_command است که بر اساس آن عملیات مورد نظر انجام می گیرد.

این ماژول دو خروجی هم دارد که خروجی محاسبه شده ALU و 4 بیت بدست آمده برای Status_Rge است.

```
27
28      assign SR = {Z, C_out, N, V};
29
30      assign Z = (ALU_result == 32'b0) ? 1'b1 : 1'b0;
31      assign N = ALU_result[31];
32
```

4 بیت Z و C_out و N و V مقدار رجیستر Status_Reg را مشخص می کنند. بیت Z بر این اساس مشخص می شود که در صورتی که مقدار خروجی ALU برابر صفر باشد این بیت یک خواهد شد، در غیر این صورت مقدار صفر خواهد گرفت. بیت N بر این اساس مشخص می شود که پر ارزش ترین بیت خروجی ALU چگونه است؛ اگر این بیت برابر صفر بود یعنی این عدد مثبت است و اگر برابر یک بود یعنی این عدد منفی است. دو بیت C_out و V که برای Carry و overflow هستند در always با مقدار اولیه صفر مقداردهی شده اند و در عملیات های جمع و تفریق ممکن است تولید شده و مقدار یک بگیرند.

در ادامه به توضیح هر کدام از کیس های این always می پردازیم.

Instruction	ALU Command	Operation
MOV	0001	result = in2
MVN	1001	result = ~in2
ADD	0010	result = in1 + in2
ADC	0011	result = in1 + in2 + C
SUB	0100	result = in1 - in2
SBC	0101	result = in1 - in2 – ~C
AND	0110	result = in1 & in2
ORR	0111	result = in1 | in2
EOR	1000	result = in1 ^ in2
CMP	0100	result = in1 - in2
TST	0110	result = in1 & in2
LDR	0010	result = in1 + in2
STR	0010	result = in1 + in2
B	XXXX	

جدول ۵– ریز دستورهای واحد حساب و منطق

```
always @(ALU_command, ALU_in1, ALU_in2) begin

    {ALU_result, C_out, V} = 34'b0;

    case (ALU_command)
        MOV: ALU_result = ALU_in2;
        MVN: ALU_result = ~ALU_in2;
        ADD: begin
            {C_out, ALU_result} = ALU_in1 + ALU_in2;
            V = (ALU_in1[31] == ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
        end
        ADC: begin
            {C_out, ALU_result} = ALU_in1 + ALU_in2 + {32'b0, C_in};
            V = (ALU_in1[31] == ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
        end
        SUB: begin
            {C_out, ALU_result} = {ALU_in1[31], ALU_in1} - {ALU_in2[31], ALU_in2};
            V = (ALU_in1[31] == ~ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
        end
        SBC: begin
            {C_out, ALU_result} = {ALU_in1[31], ALU_in1} - {ALU_in2[31], ALU_in2} -  {32'b0, ~C_in};
            V = (ALU_in1[31] == ~ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
        end
```

ALU.v

```
45              ADC: begin
46                  {C_out, ALU_result} = ALU_in1 + ALU_in2 + {32'b0, C_in};
47                  V = (ALU_in1[31] == ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
48              end
49              SUB: begin
50                  {C_out, ALU_result} = {ALU_in1[31], ALU_in1} - {ALU_in2[31], ALU_in2};
51                  V = (ALU_in1[31] == ~ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
52              end
53              SBC: begin
54                  {C_out, ALU_result} = {ALU_in1[31], ALU_in1} - {ALU_in2[31], ALU_in2} -  {32'b0, ~C_in};
55                  V = (ALU_in1[31] == ~ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
56              end
57              AND: ALU_result = ALU_in1 & ALU_in2;
58              ORR: ALU_result = ALU_in1 | ALU_in2;
59              EOR: ALU_result = ALU_in1 ^ ALU_in2;
60              CMP: begin
61                  {C_out, ALU_result} = {ALU_in1[31], ALU_in1} - {ALU_in2[31], ALU_in2};
62                  V = (ALU_in1[31] == ~ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
63              end
64              TST: ALU_result = ALU_in1 & ALU_in2;
65              LDR: begin
66                  {C_out, ALU_result} = ALU_in1 + ALU_in2;
67                  V = (ALU_in1[31] == ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
68              end
69              STR: begin
70                  {C_out, ALU_result} = ALU_in1 + ALU_in2;
71                  V = (ALU_in1[31] == ALU_in2[31]) & (ALU_in1[31] != ALU_result[31]);
72              end
73          endcase
74      end
75
76  endmodule
```

- کامند MOV: در این حالت ورودی دوم ALU را در خروجی ALU قرار می دهیم.
- کامند MVN: در این حالت نتیجه not شده ورودی دوم ALU را در خروجی ALU قرار می دهیم.
- کامند ADD: در این حالت نتیجه حاصل جمع ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم. بیت 33ام حاصل از جمع دو عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت باشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.
- کامند ADC: در این حالت نتیجه حاصل جمع ورودی اول و ورودی دوم ALU و بیت C_in ورودی را در خروجی ALU قرار می دهیم. بیت 33ام حاصل از جمع دو عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت باشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.
- کامند SUB: در این حالت نتیجه تفریق جمع ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم. و برای اینکه بیت carry حاصل از تفریق دو عدد از هم به درستی محاسبه شود، دو عملوند را یک بیت sign extend می کنیم. بیت 33ام حاصل از جمع دو

عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت نباشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.

- کامند SBC: در این حالت نتیجه حاصل تفریق ورودی اول و ورودی دوم ALU منهای not بیت C_in ورودی را در خروجی ALU قرار می دهیم.و برای اینکه بیت carry حاصل از تفریق دو عدد از هم به درستی محاسبه شود، دو عملوند را یک بیت sign extend می کنیم. بیت 33ام حاصل از جمع دو عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت نباشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.
- کامند AND: در این حالت نتیجه bitwise and ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم.
- کامند ORR: در این حالت نتیجه bitwise or ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم.
- کامند EOR: در این حالت نتیجه bitwise xor ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم.
- کامند CMP: در این حالت برای مقایسه دو عدد ورودی ALU، نتیجه تفریق جمع ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم. و برای اینکه بیت carry حاصل از تفریق دو عدد از هم به درستی محاسبه شود، دو عملوند را یک بیت sign extend می کنیم. بیت 33ام حاصل از جمع دو عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت نباشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.
- کامند TST: در این حالت برای سنجش تساوی دو عدد ورودی ALU، نتیجه bitwise and ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم.
- کامند LDR: در این حالت برای محاسبه آدرس حافظه، نتیجه حاصل جمع ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم. بیت 33ام حاصل از جمع دو عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت باشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.
- کامند STR: در این حالت برای محاسبه آدرس حافظه، نتیجه حاصل جمع ورودی اول و ورودی دوم ALU را در خروجی ALU قرار می دهیم. بیت 33ام حاصل از جمع دو عدد را در صورتی که وجود داشته باشد، در C_out قرار می دهیم. در صورتی که دو عدد ورودی اول و دوم هم علامت باشند و حاصل جمع آن دو عدد با آن دو عدد هم علامت نباشند، آنگاه سریزر با overflow رخ داده است و بیت V را برابر یک می کنیم.

- ماژول **Val2Generator**:

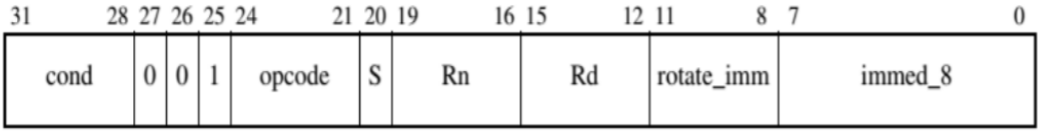
```
Val2Generator.v
1    `timescale 1ns/1ns
2
3    module Val2Generator(
4        input imm, is_MEM_command,
5        input [11:0] Shifte_operand,
6        input [31:0] Val_Rm,
7        output reg [31:0] Val2_result
8    );
9
```

در پردازنده ARM پیاده سازی شده 32 برای بدست آوردن عدد ورودی دوم ALU از این ماژول استفاده می کنیم. در این پردازنده برای مقدار Shift_operand دو حالت 32 بیت فوری و شیفت فوری پیاده سازی شده است. از ورودی ها، بیت MEM_R_EN و MEM_W_EN از ControlUnit مرحله Decode به دست آمده است. بیت imm مشخص کننده این است که به 32 بیت عدد فوری یا 32 بیت عدد شیفت فوری داده شده نیاز داریم. مقدار Shifte_operand برابر 12 بیت کم ارزش instruction است. سیگنال is_MEM_command در صورتی که برابر یک باشد یعنی command با مموری و عمل خواندن یا نوشتن کار داشته است. مقدار Val_Rm برابر با مقدار ذخیره شده در رجیستر Rm است. در نهایت Val2_result همان خروجی این ماژول است.

```
15       integer i;
16
17       always @(*) begin
18           if (is_MEM_command == 1'b1)
19               Val2_result <= {20'b0, Shifte_operand};
20
21           else if (imm == 1'b1) begin
22                   Val2_result = {24'b0, Shifte_operand[7:0]};
23                   for (i = 0; i < {1'b0, Shifte_operand[11:8]}; i = i + 1) begin
24                       Val2_result = {Val2_result[1], Val2_result[0], Val2_result[31:2]};
25                   end
26           end
27           else begin
28               case (Shifte_operand[6:5])
29                   LSL: Val2_result = Val_Rm << {1'b0, Shifte_operand[11:7]};
30                   LSR: Val2_result = Val_Rm >> {1'b0, Shifte_operand[11:7]};
31                   ASR: Val2_result = Val_Rm >>> {1'b0, Shifte_operand[11:7]};
32                   ROR: begin
33                       Val2_result <= Val_Rm;
34                       for (i = 0; i < {1'b0, Shifte_operand[11:7]}; i = i + 1) begin
35                           Val2_result = {Val2_result[0], Val2_result[31:1]};
36                       end
37                   end
38                   default: Val2_result = Val_Rm;
39               endcase
40           end
41       end
42
43   endmodule
```

در ادامه حالات مختلف پیاده سازی شده در always را بررسی می کنیم.

- **دستور حافظه:** اگر سیگنال is_MEM_command برابر یک بود و با حافظه کار داشتیم، مقدار Shifte_operand را تا 32 بیت sign extend می کنیم و در خروجی Val2_result قرار می دهیم.
- **32 بیت عدد فوری:** اگر بیت imm برابر یک بود به این حالت بر می خوریم. در ابتدا 8 بیت کم ارزش Shifte_operand که همان immed_8 است را تا 32 بیت sign extend می کنیم و سپس به اندازه 2 برابر 4 بیت پر ارزش Shifte_operand که همان مقدار rotate_imm است به راست شیفت می دهیم. برای پیاده سازی، حلقه را به تعداد 4 بیت پر ارزش Shifte_operand اجرا می کنیم ولی هر بار مقدار را دو بیت به راست شیفت می دهیم. در نهایت اجرای حلقه مقدار Val2_result به دست می آید.

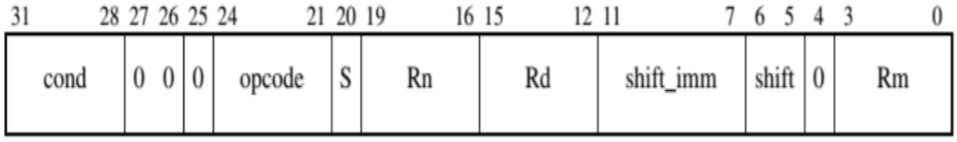


شکل ۱: دستورالعمل از نوع ۳۲ بیت عدد فوری

- **شیفت فوری:** اگر بیت imm برابر صفر بود به این حالت بر می خوریم. در اینجا عدد خوانده شده از رجیستر Rm را بر اساس حالت شیفت که در جدول زیر آمده است، به مقدار shift_imm شیفت می دهیم. بر اساس بیت 5 و 6 shift_operand، چهار حالت برای شیفت فوری خواهیم داشت.
  - حالت LSL: در این حالت مقدار Val_Rm را به اندازه پنج بیت پر ارزش shift_operand به چپ شیفت می دهیم.
  - حالت LSR: در این حالت مقدار Val_Rm را به اندازه پنج بیت پر ارزش shift_operand به راست شیفت می دهیم.
  - حالت ASR: در این حالت مقدار Val_Rm را به اندازه پنج بیت پر ارزش shift_operand به صورت علامت دار به چپ شیفت می دهیم.
  - حالت ROR: در این حالت مقدار Val_Rm را به اندازه چهار بیت پر ارزش shift_operand به راست rotate می کنیم.

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

جدول ۴- وضعیت شیفت در دستورات شیفت فوری



شکل ۳: دستورالعمل از نوع شیفت فوری

- **ماژول EXE_reg:**

از این ماژول برای ذخیره سازی مقدار هایی که قرار است از مرحله Execution به مرحله Memory برود، استفاده می شود.

در این ماژول از رجیستر های 1 بیت و 4 بیت و 32 بیت که در فایل رجیستر ها تعریف شده اند، استفاده شده است.

این رجیستر ها مقدار سیگنال freeze را به عنوان پایه load خود دریافت می کنند.

هر کدام از این رجیستر ها با یک always تعریف شده اند و در صورت یک بودن سیگنال rst مقدار اولیه صفر می گیرند.

و در صورتی که مقدار پایه load آن ها برابر با یک بود، مقدار ورودی در خروجی آن ها ریخته می شود.

EXE_reg.v

```
1   `timescale 1ns/1ns
2
3   module EXE_reg(
4       input clk, rst, freeze,
5       input WB_en_in, MEM_R_EN_in, MEM_W_EN_in,
6       input [3:0] dst_in,
7       input [31:0] ALU_result_in, ST_Val_in,
8       output WB_en_out, MEM_R_EN_out, MEM_W_EN_out,
9       output [3:0] dst_out,
10      output [31:0] ALU_result_out, ST_Val_out
11  );
12
13      Reg_1bit WB_en_reg(clk, rst, freeze, WB_en_in, WB_en_out);
14      Reg_1bit MEM_R_EN_reg(clk, rst, freeze, MEM_R_EN_in, MEM_R_EN_out);
15      Reg_1bit MEM_W_EN_reg(clk, rst, freeze, MEM_W_EN_in, MEM_W_EN_out);
16      Reg_4bit dst_reg(clk, rst, freeze, dst_in, dst_out);
17      Reg_32bit ALU_result_reg(clk, rst, freeze, ALU_result_in, ALU_result_out);
18      Reg_32bit ST_Val_reg(clk, rst, freeze, ST_Val_in, ST_Val_out);
19
20  endmodule
```

مرحله حافظه (Memory)

در این مرحله داده ها از مموری خوانده یا در آن نوشته می شوند. این مرحله دارای زیر ماژول های زیر است:

- **ماژول Memory:**

در این ماژول یک حافظه شبیه سازی شده است که داده ها در آن نوشته یا از آن خوانده می شوند. سیگنال های ورودی و خروجی آن به شرح زیر هستند:

1. ورودی clk
2. ورودی MEMread که تعیین می کند آیا مقداری از حافظه خوانده شود یا خیر.
3. ورودی MEMwrite که تعیین می کند آیا مقداری در حافظه نوشته شود یا خیر.
4. ورودی address که آدرس خانه ای از حافظه که برای نوشتن یا خواندن به آن دسترسی پیدا شده است را تعیین می کند.
5. ورودی data که مقداری را که باید در آدرس ورودی نوشته شود تعیین می کند.
6. خروجی MEM_result که مقداری است که در خانه address موجود است.

```
module Memory(
    input clk, MEMread, MEMwrite,
    input[31:0] address, data,
    output[31:0] MEM_result
);
```

پیاده سازی این ماژول به شکل زیر است:

```
    reg [31:0] MemByte[0:255];
    wire[31:0] adr = address - 32'd1024;
```

ابتدا یک مموری 256 تایی تعریف می شود تا دادهها در آن نوشته و یا از آن خوانده شود. سپس آدرس ورودی 1024 واحد کم می شود تا بتوان در این مموری از آدرس استفاده کرد. در اصل index حافظه تعریف شده می شود.

```
assign MEM_result = MEMread ? MemByte[{2'b0, adr[31:2]}] : 32'bz;
```

سپس با استفاده از یک assign و با توجه به سیگنال MEMread خروجی MEM_result تعیین می شود. توجه کنید آدرس دو واحد به چپ شیفت داده شده است یعنی در اصل تقسیم بر چهار شده است زیرا آدرس ها همگی مضرب چهار هستند. در صورتی که سیگنال خواندن فعال نبود روی خروجی چیزی قرار داده نمی شود. همچنین توجه کنید این قسمت به صورت ترکیبی پیاده سازی شده است.

```
always @(posedge clk) begin
  if (MEMwrite)
    MemByte[{2'b0, adr[31:2]}] <= data;
end
```

قسمت نوشتن در حافظه با استفاده از یک always پیاده سازی شده است که در صورتی که سیگنال نوشتن فعال باشد داده ورودی بر روی آدرس مورد نظر قرار داده می شود. این آدرس نیز با استدلال قبلی دو واحد شیفت داده شده است. توجه کنید این always به لبه بالا رونده کلاک حساس است.

- **ماژول MEM_Stage:**

این ماژول اصلی این قسمت است که حافظه اصلی در آن قرار دارد. سیگنال های ورودی و خروجی آن به شرح زیر هستند:

1. ورودی های clk و rst
2. ورودی WB_EN_IN که نشان می دهد آیا دستور نیاز به نوشتن در رجیستر فایل دارد یا خیر. این ورودی برای استفاده در قسمت بازنویسی است.
3. ورودی MEM_R_EN_IN که سیگنال فعال سازی خواندن از حافظه است که از قسمت اجرا وارد می شود.
4. ورودی MEM_W_EN_IN که سیگنال فعال سازی نوشتن در حافظه است که از قسمت اجرا وارد می شود.

5. ورودی ALU_RES_IN که خروجی ALU در قسمت قبل است که ممکن است حاوی آدرس حافظه ای که باید با آن تعامل کرد باشد.
6. ورودی Val_Rm که مقدار اپرند دوم رجیستر فایل است که ممکن است حاوی مقداری باشد که باید در حافظه نوشت.
7. ورودی Dest_IN که شماره رجیستری است که ممکن است بخواهیم نتیجه را در آن بنویسیم. (بسته به نوع دستور)
8. خروجیهای WB_EN و MEM_R_EN که ورودی متناظر را خروجی می دهند.
9. خروجی MEM_OUT که مقدار خارج شده از حافظه اصلی است.
10. خروجی های ALU_RES و Dest که ورودی متناظر را خروجی می دهند.

```
module MEM_Stage(
    input clk, rst,
    input WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,
    input [31:0] ALU_RES_IN, Val_Rm,
    input [3:0] Dest_IN,
    output WB_EN, MEM_R_EN,
    output [31:0] MEM_OUT, ALU_RES,
    output [3:0] Dest
);
```

پیاده سازی این ماژول به شکل زیر است:

```
    Memory mem(clk, MEM_R_EN_IN, MEM_W_EN_IN, ALU_RES_IN, Val_Rm, MEM_OUT);
    assign WB_EN = WB_EN_IN;
    assign ALU_RES = ALU_RES_IN;
    assign Dest = Dest_IN;
    assign MEM_R_EN = MEM_R_EN_IN;
```

همانطور که مشاهده می شود از حافظه اصلی یک نمونه گرفته می شود و ورودی ها و خروجی های مربوط به آن داده می شود. همانطور که ذکر شد خروجی ALU شامل آدرس حافظه و Val_Rm شامل داده ای که باید در حافظه نوشته شود است. در نهایت نیز سیگنال هایی که باید از این ماژول گذر می کردند با استفاده از assign مقداردهی شدند.

- **ماژول های MemRegisterXbit:**

این ماژول ها رجیستر هستند، که در میان مرحله حافظه و مرحله بازنویسی استفاده می شوند. تفاوت آنها در تعداد بیت است که به ازای X های 1 و 4 و 32 تعریف شده هستند. ورودی و خروجی این ماژول ها به شرح زیر هستند:

1. ورودی clk که کلاک پردازنده است.
2. ورودی rst که سیگنال reset پردازنده است.
3. ورودی load که تعیین میکند آیا ورودی روی خروجی قرار گیرد یا خیر.
4. ورودی in که ورودی رجیستر است.
5. خروجی out که خروجی رجیستر است.

```
module MemRegister1bit(
    input clk, rst, load,
    input in,
    output reg out
);
    always@(posedge clk, posedge rst) begin
        if (rst) out <= 0;
        else if (load) out <= in;
    end
endmodule
```

به عنوان نمونه رجیستر یک بیتی آورده شده است. همانطور که مشاهده می شود این ماژول با استفاده از یک always که به لبه بالا رونده سیگنال های clk و rst حساس است نوشته شده است. اگر سیگنال rst یک بود خروجی برابر صفر قرار داده می شود و رجیستر reset می شود. همچنین اگر سیگنال load یک بود و سیگنال rst صفر بود خروجی برابر ورودی قرار داده می شود. با این نوع پیاده سازی ابتدا به rst و سپس به load اولویت داده شده است.

- **ماژول MEM_Stage_Reg:**

این ماژول واسط بین مراحل حافظه و بازنویسی است. ورودی ها و خروجی های این ماژول به شرح زیر هستند:

1. ورودی های clk و rst
2. ورودی flush که در صورت یک بودن خروجی این رجیسترهای میانی را صفر میکند.
3. ورودی freeze که در صورت یک بودن باعث فریز شدن این رجیسترها می شود و باعث می شود دستور به جلو حرکت نکند.
4. سایر ورودی ها و خروجی ها همان سیگنال های تولید شده در ماژول ID_Stage هستند که صرفا در رجیستر قرار گرفته اند. یعنی هر ورودی وارد یک رجیستر شده است و خروجی ماژول خروجی متناظر با آن ورودی است که از رجیستر خارج می شود.

```
module MEM_Reg (
    input clk, rst, freeze,
    input WB_en_in, MEM_R_en_in,
    input [31:0] ALU_result_in, Mem_read_value_in,
    input [3:0] Dest_in,
    output WB_en, MEM_R_en,
    output [31:0] ALU_result, Mem_read_value,
    output [3:0] Dest
);
```

پیاده سازی این ماژول به شکل زیر است:

```
MemRegister1bit R1(clk, rst, freeze, WB_en_in, WB_en);
MemRegister1bit R2(clk, rst, freeze, MEM_R_en_in, MEM_R_en);
MemRegister32bit R3(clk, rst, freeze, ALU_result_in, ALU_result);
MemRegister32bit R4(clk, rst, freeze, Mem_read_value_in, Mem_read_value);
MemRegister4bit R5(clk, rst, freeze, Dest_in, Dest);
```

همانطور که مشاهده می شود هر ورودی وارد یک رجیستر شده است و خروجی متناظر از آن از رجیستر خارج شده است. همچنین هر استفاده از هر رجیستر با توجه به تعداد بیت های آن انجام شده است. از سیگنال freeze نیز برای load رجیستر استفاده شده است تا فریز شدن پردازنده کنترل شود.

- **ماژول MEM:**

در این ماژول اتصال بین ماژول های MEM_Stage و MEM_Stage_Reg انجام می شود و نکته خاصی در این ماژول وجود ندارد. پیاده سازی آن به شکل زیر است:

```
module MEM(
    input clk, rst,
    input WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,
    input [31:0] ALU_RES_IN, Val_Rm,
    input [3:0] Dest_IN,
    output WB_en, MEM_R_en,
    output[3:0] Dest,
    output[31:0] ALU_result, Mem_read_value
);
    wire[31:0] mem_out_temp, alu_res_temp;
```

```
    wire[3:0] dest_temp;
    wire WB_EN_temp, MEM_R_EN_temp;
    MEM_Stage memory_access(
        clk, rst,
        WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,
        ALU_RES_IN, Val_Rm,
        Dest_IN,
        WB_EN_temp, MEM_R_EN_temp,
        mem_out_temp, alu_res_temp,
        dest_temp
    );

    MEM_Reg memory_access_registers(
        clk, rst, WB_EN_temp, MEM_R_EN_temp,
        alu_res_temp, mem_out_temp,
        dest_temp,
        WB_en, MEM_R_en,
        ALU_result, Mem_read_value,
        Dest
    );

endmodule
```

همانطور که مشاهده می شود صرفا از ماژول های MEM_Stage و MEM_Stage_Reg نمونه برداری شده است و سیگنال های مورد نیاز این دو ماژول به عنوان ورودی و سیگنال های مورد نیاز در مرحله بازنویسی به عنوان خروجی در نظر گرفته شده است که در

اصل ورودی ها و خروجی های همین دو ماژول است. نقش تمام این سیگنال های ورودی و خروجی در قسمت توضیحات ماژول های MEM_Stage و MEM_Stage_Reg مفصلا شرح داده شده است.

مرحله بازنویسی (Write Back)

در این مرحله سیگنالهای مورد نیاز برای نوشتن در رجیستر فایل تنظیم می شوند. به بررسی ماژول استفاده شده در این مرحله می پردازیم:

- **ماژول WB_Stage:**

سیگنال های ورودی و خروجی این ماژول به شرح زیر هستند:

1. ورودی های clk و rst.
2. ورودی ALU_result که خروجی محاسبه شده توسط ALU است و از مرحله حافظه به این مرحله رسیده است.
3. ورودی MEM_result که بیانگر داده ای است که در مرحله قبل خوانده شده است.
4. ورودی Dest_in که بیانگر شماره رجیستری است که قرار است داده در آن نوشته شود.
5. ورودی MEM_R_en که بیانگر این است چه داده ای باید در رجیستر فایل نوشته شود.
6. ورودی WB_EN_in که بیانگر این است آیا داده باید در رجیستر فایل نوشته شود یا خیر.
7. خروجی out که بیانگر داده ای است که در رجیستر فایل نوشته می شود.
8. خروجی Dest که بیانگر شماره رجیستری است که داده در آن نوشته میشود.
9. خروجی WB_EN که بیانگر این است آیا داده ای در رجیستر فایل نوشته می شود یا خیر.

```
module WB(
    input clk, rst,
    input [31:0] ALU_result, MEM_result,
```

```
    input [3:0] Dest_in,
    input MEM_R_en, WB_EN_in,
    output [31:0] out,
    output [3:0] Dest,
    output WB_EN
);
```

پیاده سازی این ماژول به شکل زیر است:

```
    assign WB_EN = WB_EN_in;
    assign Dest = Dest_in;
    MUX2_32 mux(MEM_result, ALU_result, MEM_R_en, out);
```

توسط دو assign مقادیر WB_EN و Dest تعیین می شود. نکته ای که مهم است این است که کدام داده در رجیستر مورد نظر نوشته شود (در صورت نوشته شدن) که توسط یک مالتی پلکسر این تصمیم گرفته می شود. اگر دستور از نوع حافظه بود یعنی MEM_R_en آن یک بود باید خروجی حافظه یعنی MEM_result در رجیستر نوشته شود. اما اگر اینطور نبود یعنی دستور محاسباتی بوده و خروجی ALU یعنی ALU_result باید در رجیستر فایل نوشته شود.

بخش ثبات وضعیت (Status Register)

این ماژول، 4 بیت zero و carry و negative و overflow را در خود ذخیره می کند و در صورت نیاز به ماژول مربوطه خروجی می دهد.
ورودی اول clk و ورودی دوم rst که برای ریستارت پردازنده استفاده می شود.
ورودی بیت S همان پایه load رجیستر است که در صورتی که فعال باشد مقدار ورودی 4 بیتی SR_in در خروجی 4 بیتی SR_out قرار می گیرد.

StatusReg.v

```v
module Status_Reg(
    input clk, rst, S,
    input [3:0] SR_in,
    output reg [3:0] SR_out
);

    always @(negedge clk, posedge rst) begin
        if (rst) begin
            SR_out <= 4'b0;
        end
        else if (S == 1'b1) begin
            SR_out <= SR_in;
        end
    end

endmodule
```

عملکرد این رجیستر با یک always پیاده سازی شده است؛ با لبه پایین رونده کلاک می خورد و با لبه بالا رونده و در صورتی که مقدار سیگنال rst برابر یک بود ری ست و درون رجیستر مقدار صفر ریخته می شود.

با هر بار کلاک خوردن و در صورتی که مقدار سیگنال rst برابر صفر نباشد، بررسی می کنیم که اگر مقدار سیگنال S برابر یک بود، آنگاه مقدار SR_in را در SR_out که خروجی رجیستر است قرار می دهیم. در غیر این صورت اگر مقدار سیگنال S برابر صفر بود، رجیستر مقدار قبلی خود را حفظ می کند.

بخش تشخیص مخاطره داده ای (Hazard Detection Unit)

HazardDetectionUnit.v

```v
module Hazard_Detection_Unit(
    input EXE_WB_EN, MEM_WB_EN, Two_src, Forward_en, EXE_MEM_R_EN, hasSrc1,
    input [3:0] Src1, Src2, EXE_dst, MEM_dst,
    output reg Hazard
);

```

این ماژول برای جلوگیری از وقوع مخاطره داده ای از نوع read after write استفاده می شود.

سیگنال Forward_en بیانگر فعال یا غیر فعال بودن امکان فورواردینگ در پردازنده ARM است.

سیگنال های MEM_WB_EN و EXE_WB_EN که به ترتیب از بخش های Memory و Execution آمده است، بیانگر امکان نوشتن و write back در رجیستر فایل هستند. سیگنال Two_src در صورت یک بودن، بیانگر این است که دستور ورودی دوم را دارد و در صورت صفر بودن این سیگنال، دستور یک ورودی دارد یا ورودی ندارد. سیگنال hasSrc1 که به صورت امتیازی پیاده سازی شده است، بیانگر این است که اگر صفر باشد، دستور ورودی ندارد و اگر یک باشد، دستور یک ورودی را دارد. سیگنال EXE_MEM_R_EN که از مرحله Execution آمده است، بیانگر این است که دستور در بخش اجرا، امکان خواندن از حافظه را دارد یا ندارد. مقدار های MEM_dst و EXE_dst بیانگر شماره رجیستر مقصد دستور های مراحل به ترتیب Memory و Execution است. مقدار های Src1 و Src2 بیانگر شماره رجیستر های اول و دوم مبدا دستور در همین بخش Decode است.

سیگنال خروجی این ماژول بیانگر تشخیص دادن یا ندادن هازارد است.

```
6
7       always @(*) begin
8           Hazard = 1'b0;
9           if (Forward_en == 1'b0) begin
10              if ((EXE_WB_EN == 1'b1) && (Src1 == EXE_dst) && (hasSrc1 == 1'b1))
11                  Hazard = 1'b1;
12              else if ((EXE_WB_EN == 1'b1) && (Src2 == EXE_dst) && (Two_src == 1'b1))
13                  Hazard = 1'b1;
14              else if ((MEM_WB_EN == 1'b1) && (Src1 == MEM_dst) && (hasSrc1 == 1'b1))
15                  Hazard = 1'b1;
16              else if ((MEM_WB_EN == 1'b1) && (Src2 == MEM_dst) && (Two_src == 1'b1))
17                  Hazard = 1'b1;
18          end
19          else begin
20              if ((EXE_MEM_R_EN == 1'b1) && (EXE_WB_EN == 1'b1) && (Src1 == EXE_dst) && (hasSrc1 == 1'b1))
21                  Hazard = 1'b1;
22              else if ((EXE_MEM_R_EN == 1'b1) && (EXE_WB_EN == 1'b1) && (Src2 == EXE_dst) && (Two_src == 1'b1))
23                  Hazard = 1'b1;
24          end
25      end
26
27  endmodule
28
```

در یک always ابتدا مقدار اولیه سیگنال هازارد را برابر صفر قرار می دهیم. سپس بررسی می کنیم که اگر امکان فورواردینگ فعال بود، از بخش دوم و در صورت غیرفعال بودن این امکان از بخش اول برای تشخیص هازارد استفاده شود.

در صورتی که هازارد تشخیص داده شود، تمام سیگنال های خروجی از ماژول ControlUnit را بی اثر می کنیم و مرحله IF را freeze می کنیم.

در بخش اول چهار حالت برای تشخیص مخاطره داده ای read after write داریم:

- اگر دستور در مرحله Decode، دارای Src1 بود و دستور در مرحله Execution در همین رجیستر Src1 می نوشت.
- اگر دستور در مرحله Decode، دارای Src2 بود و دستور در مرحله Execution در همین رجیستر Src2 می نوشت.
- اگر دستور در مرحله Decode، دارای Src1 بود و دستور در مرحله Memory در همین رجیستر Src1 می نوشت.
- اگر دستور در مرحله Decode، دارای Src2 بود و دستور در مرحله Memory در همین رجیستر Src2 می نوشت.

در بخش دوم دو حالت برای تشخیص مخاطره داده ای read after write داریم:

- اگر دستور در مرحله Decode، دارای Src1 بود و دستور در مرحله Execution از نوع دستور های مرتبط با حافظه باشد و در همین رجیستر Src1 می نوشت.
- اگر دستور در مرحله Decode، دارای Src2 بود و دستور در مرحله Execution از نوع دستور های مرتبط با حافظه باشد و در همین رجیستر Src2 می نوشت.

مرحله فوروارد کردن داده (Forwarding Unit)

ForwardingUnit.v

```
1  `timescale 1ns/1ns
2
3  module ForwardingUnit(
4      input MEM_wb_en, WB_wb_en, Forward_en,
5      input [3:0] src1, src2, MEM_dst, WB_dst,
6      output reg [1:0] Sel_src1, Sel_src2
7  );
```

امکان دارد که حالتی رخ دهد و دو دستور متوالی پشت سر هم به هازاد بخورند یا یک دستور و دو دستور بعدی به هازاد بخورند.

یعنی:

- دستوری که در مرحله Execution و دستوری که در مرحله Memory است با هم مخاطره داده ای داشته باشند.
- دستوری که در مرحله Execution و دستوری که در مرحله Write Back است با هم مخاطره داده ای داشته باشند.

در این صورت آن تعداد stall که در بخش تشخیص هازارد برای به روز کردن مقدار داده رجیستر می دادیم دیگر لازم نیست؛ و می توان از مقدار به دست آمده برای آن رجیستر در مراحل Memory و Write Back استفاده کرد. در ادامه این حالات توضیح داده خواهند شد.

سیگنال ورودی Forward_en بیانگر فعال بودن یا نبودن امکان فورواردینگ در پردازنده ARM است. سیگنال های ورودی MEM_wb_en و WB_wb_en بیانگر فعال بودن یا نبودن امکان نوشتن در رجیستر فایل برای دستور های مراحل به ترتیب Memory و Write Back است.

مقدار های src1 و src2 بیانگر شماره رجیستر های ورودی است و از بخش Decode به دست آمده است و در مرحله Execution استفاده می شود. مقدار های MEM_dst و WB_dst بیانگر شماره رجیستر های مقصد دستور های مراحل به ترتیب Memory و Write Back است. و در نهایت Sel_src1 و Sel_src2 مقدار های خروجی این ماژول، بیانگر selector های مالتی پلکسر های پشت دو ورودی اول و دوم ALU است.

```
8
9       always @(*) begin
10          {Sel_src1, Sel_src2} = 4'b0;
11
12          if (Forward_en == 1'b1) begin
13              if ((MEM_wb_en == 1'b1) && (src1 == MEM_dst))
14                  Sel_src1 = 2'b01;
15              else if ((WB_wb_en == 1'b1) && (src1 == WB_dst))
16                  Sel_src1 = 2'b10;
17
18              if ((MEM_wb_en == 1'b1) && (src2 == MEM_dst))
19                  Sel_src2 = 2'b01;
20              else if ((WB_wb_en == 1'b1) && (src2 == WB_dst))
21                  Sel_src2 = 2'b10;
22          end
23      end
24
25  endmodule
```

در ابتدای always مقدار های این دو selector را برابر مقدار پیش فرض صفر یا بدون فورواردینگ قرار می دهیم. و در ادامه در صورتی که امکان فورواردینگ در پردازنده وجود داشت، حالات زیر را بررسی می کنیم:

- اگر مقدار src1 با شماره رجیستر مقصد دستور در مرحله Memory برابر باشد و امکان نوشتن در رجیستر و به روز شدن آن برای دستور در مرحله Memory فعال باشد، مقدار selector اول برابر 01 خواهد شد. (دو دستور متوالی با یک فاصله)
- با اولویت کمتر به نسبت به حالت قبل، اگر مقدار src1 با شماره رجیستر مقصد دستور در مرحله Write Back برابر باشد و امکان نوشتن در رجیستر و به روز شدن آن برای دستور در مرحله Write Back فعال باشد، مقدار selector اول برابر 10 خواهد شد. (دو دستور متوالی با دو فاصله)
- اگر مقدار src2 با شماره رجیستر مقصد دستور در مرحله Memory برابر باشد و امکان نوشتن در رجیستر و به روز شدن آن برای دستور در مرحله Memory فعال باشد، مقدار selector دوم برابر 01 خواهد شد. (دو دستور متوالی با یک فاصله)
- با اولویت کمتر به نسبت به حالت قبل، اگر مقدار src2 با شماره رجیستر مقصد دستور در مرحله Write Back برابر باشد و امکان نوشتن در رجیستر و به روز شدن آن برای دستور در مرحله Write Back فعال باشد، مقدار selector دوم برابر 10 خواهد شد. (دو دستور متوالی با دو فاصله)

بخش حافظه خارجی (SRAM)

حافظه اصلی همواره مقدار قابل توجهی delay دارد و این گونه نیست که بتواند در یک کلاک پاسخ مورد نظر ما را فراهم کند. یکی از ماژول های مورد استفاده برای حافظه، SRAM است که در این بخش این ماژول را پیاده سازی کردیم. در ابتدا به پیاده سازی این ماژول می پردازیم. ورودی ها و خروجی های این ماژول به شرح زیر است:

1. ورودی های clk و rst
2. ورودی SRAM_WE_N که تعیین می کند آیا باید در sram داده نوشته شود یا خیر. توجه کنید این سیگنال با مقدار صفر فعال است.
3. ورودی SRAM_ADDR که آدرس حافظه مورد دسترسی است.
4. ورودی-خروجی SRAM_DQ که باس خروجی sram است. هنگامی که قرار است در حافظه مقداری نوشته شود آن مقدار روی این باس قرار داده می شود و وقتی از sram چیزی خوانده می شود خروجی روی این باس قرار می گیرد.

```
module SRAM (
    input clk,
    input rst,
    input SRAM_WE_N,
    input[16:0] SRAM_ADDR,
    inout[31:0] SRAM_DQ
);
```

پیاده سازی این ماژول به شرح زیر است:

```
    reg[31:0] memory[0:511];
    assign #30 SRAM_DQ = SRAM_WE_N ? memory[SRAM_ADDR] : 32'bz;


    always @(posedge clk) begin
        if (~SRAM_WE_N) begin
            memory[SRAM_ADDR] <= SRAM_DQ;
        end
    end
```

در صورتی که دستور از نوع نوشتن نباشد مقدار آدرس با یک تاخیر روی خروجی قرار داده می شود. این اتفاق با یک assign پیاده سازی شده است و به صورت ترکیبی اتفاق می افتد. برای نوشتن در حافظه نیز از یک always استفاده شده است که با لبه بالا رونده کلاک کار می کند و در صورتی که سیگنال نوشتن در حافظه فعال باشد مقدار روی باس را در حافظه می نویسد. این ماژول می تواند رفتار یک حافظه واقعی را تا حدودی برای ما شبیه سازی کند. توجه کنید فرکانس کلاک حافظه نصف کلاک پردازنده است.

توجه کنید که ما ماژول حافظه را از پردازنده خارج کرده ایم. برای مدیریت دسترسی به حافظه یک ماژول با نام SRAMController تعریف می کنیم که وظیفه کار کردن با حافظه را دارد. ورودی ها و خروجی های این ماژول به شرح زیر هستند:

1. ورودی های clk و rst
2. ورودی write_en که نشان می دهد آیا داده باید در sram نوشته شود یا خیر.
3. ورودی read_en که نشان می دهد آیا داده باید از sram خوانده شود یا خیر.
4. ورودی address که آدرس مورد دسترسی است.
5. ورودی writeData که داده ای است که باید در حافظه نوشته شود.
6. خروجی readData که نتیجه ای است که از حافظه خوانده می شود.
7. ورودی-خروجی SRAM_DQ که باس حافظه است و ممکن است داده روی آن برای نوشتن قرار داده شود یا داده از روی آن خوانده شود.
8. خروجی SRAM_address که آدرس برای دسترسی به sram است. توجه کنید که ما sram را از پردازنده خارج کرده ایم.
9. خروجی SRAM_WE_EN که سیگنال نوشتن در حافظه است.
10. سایر سیگنال ها در این پروژه استفاده نمی شوند.

```
module SRAM_Controller(
    input clk,
    input rst,

    input write_en,
    input read_en,
    input[31:0] address,
    input[31:0] writeData,

    output[31:0] readData,
    output ready,

    input[31:0] SRAM_DQ,
```

```
    output[16:0] SRAM_ADDR,
    output SRAM_UB_N,
    output SRAM_LB_N,
    output SRAM_WE_N,
    output SRAM_CE_N,
    output SRAM_OE_N
);
```

حال به پیاده سازی این ماژول می پردازیم:

```
    parameter [1:0] IDLE = 2'b00;
    parameter [1:0] READING = 2'b01;
    parameter [1:0] WRITING = 2'b11;

    wire[1:0] ns, ps;
    wire[2:0] counter, counter_out;
    wire[31:0] dq;
```

ابتدا پارامترها برای state machine مورد استفاده در این ماژول را تعریف می کنیم. سپس سیگنال های مورد نیاز در ادامه را تعریف می کنیم.

```
    assign SRAM_UB_N = 1'b0;
    assign SRAM_LB_N = 1'b0;
    assign SRAM_CE_N = 1'b0;
    assign SRAM_OE_N = 1'b0;
```

سیگنال هایی که استفاده نمی شوند را برابر صفر قرار می دهیم.

```
Reg_2bit state_register(clk, rst, ns, ps);
Reg_3bit counter_register(clk, rst, counter, counter_out);
Reg_32bit dq_reg(clk, rst, 1'b1, SRAM_DQ, dq);
```

سپس سه رجیستر برای ذخیره سازی اطلاعات نمونه گیری می کنیم. یکی برای نگهداری state. دیگری برای اینکه بدانیم چند کلاک از شروع نوشتن یا خواندن می گذرد. توجه کنید که اعمال نوشتن یا خواندن از sram هر کدام باید شش کلاک به طول بینجامد و این شمارنده برای شمردن این شش کلاک استفاده می شود. رجیستر آخر نیز برای ریختن خروجی SRAM در یک مقدار درونی است تا هنگامی که دستور از نوع خواندن است از آن استفاده شود. توجه کنید دو رجیستر اول سیگنال load ندارند و load رجیستر سوم یک در نظر گرفته شده است یعنی همواره load می شود.

```
wire [31:0] adr = address - 32'd1024;
assign SRAM_ADDR = adr[18:2];
```

سپس آدرس ورودی منهای 1024 میشود و فقط بیتهای 2 تا 18 در نظر گرفته میشود تا به آدرس دهی استاندارد پردازنده برسیم.

```
assign SRAM_DQ = (ns == WRITING && counter < 3'b101) ? writeData : 32'bz;
```

با استفاده از یک assign سیگنال SRAM_DQ تعیین می شود. اگر در state نوشتن بودیم و مقدار شمارنده کمتر از پنج باشد مقداری که باید در حافظه نوشته شود را روی SRAM_DQ قرار میدهیم در غیر این صورت روی آن چیزی قرار نمی دهیم تا بتوانیم مقدار آن را بخوانیم.

```
assign SRAM_WE_N = (ns == WRITING && counter < 3'b101) ? 1'b0 : 1'b1;
```

سپس سیگنال فعال سازی نوشتن در حافظه با یک assign تعیین می شود. اگر در state نوشتن بودیم و مقدار شمارنده کمتر از پنج باشد این سیگنال را فعال و در غیر این صورت آن را غیر فعال می کنیم.

```
    assign readData = read_en ? dq : 32'bz;
```

سپس اگر سیگنال خواندن فعال باشد، مقدار روی باس حافظه را روی readData می ریزیم.

```
assign ready = (ns == READING && counter < 3'b101) ? 1'b0 :
        (ns == WRITING && counter < 3'b101) ? 1'b0 : 1'b1;
```

بعد از آن سیگنال ready را تعیین می کنیم. اگر در state خواندن یا نوشتن باشیم و شمارنده کمتر از پنج باشد این سیگنال را صفر میکنیم زیرا خروجی آماده نیست. در غیر این صورت این سیگنال را یک می کنیم.

```
assign counter = rst ? 3'b0 : (ps == READING && counter_out != 3'b101) ?  (counter_out + 1'b1) :
    (ps == WRITING && counter_out != 3'b101) ? (counter_out + 1'b1) : 3'b0;
```

سپس مقدار شمارنده را بروزرسانی می کنیم. اگر در state نوشتن یا خواندن باشیم و هنوز به پنج نرسیده باشیم شمارنده را یکی زیاد می کنیم. در غیر این صورت شمردن ری ست می شود. همچنین اگر سیگنال ورودی ری ست یک باشد نیز شمارنده ری ست می شود.

```
assign ns = rst ? IDLE :
(
    (ps == IDLE && read_en) ? READING :
    (ps == IDLE && write_en) ? WRITING :
    (ps == READING && counter != 3'b101) ? READING :
    (ps == WRITING && counter != 3'b101) ? WRITING : 2'b00
);
```

سپس state machine را بروزرسانی می کنیم. اگر در شروع کار باشیم و سیگنال خواندن فعال شود وارد state خواندن می شویم. همچنین اگر در شروع کار باشیم و سیگنال نوشتن فعال شود وارد state نوشتن می شویم. اگر در هر کدام از این state ها هستیم تا وقتی شمارنده به پنج نرسد در همان state می مانیم و اگر شمارنده به پنج رسید یعنی کار ما تمام است و بعد از پنج کلاک به حالت اولیه باز می گردیم و منتظر ورودی های بعدی می شویم.

با تغییرات داده شده بعضی ماژول های دیگر دستخوش تغییر می شوند. ماژول MEM به صورت زیر تغییر می کند:

```
module MEM_Stage(
```

```
    input clk, rst,
    input WB_EN_IN, MEM_R_EN_IN, MEM_W_EN_IN,
    input [31:0] ALU_RES_IN, Val_Rm,
    input [3:0] Dest_IN,
    output WB_EN, MEM_R_EN,
    output [31:0] MEM_OUT, ALU_RES,
    output [3:0] Dest,
    output sram_ready,
    inout[31:0] SRAM_DQ,
    output[16:0] SRAM_ADDR,
    output SRAM_WE_N
);

    wire SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N;

    SRAM_Controller mc(clk, rst, MEM_W_EN_IN, MEM_R_EN_IN,
        ALU_RES_IN, Val_Rm, MEM_OUT,
        sram_ready,
        SRAM_DQ,
        SRAM_ADDR,
        SRAM_UB_N,
        SRAM_LB_N,
        SRAM_WE_N,
        SRAM_CE_N,
        SRAM_OE_N
```

```
  );
  assign WB_EN = WB_EN_IN;
  assign ALU_RES = ALU_RES_IN;
  assign Dest = Dest_IN;
  assign MEM_R_EN = MEM_R_EN_IN;
endmodule
```

همانطور که مشاهده می شود به جای حافظه قسمت قبل از sram استفاده شده و سیگنال های خروجی آن نیز به عنوان خروجی این ماژول در نظر گرفته شده است. این خروجی ها به عنوان خروجی کل پردازنده نیز در نظر گرفته شده است تا به sram که کنار پردازنده است وارد شود.

جدول ۱ – بیت‌های کنترلی SRAM

SRAM_WE_N	PIN_AE10	SRAM Write Enable
SRAM_OE_N	PIN_AD10	SRAM Output Enable
SRAM_UB_N	PIN_AF9	SRAM High-byte Data Mask
SRAM_LB_N	PIN_AE9	SRAM Low-byte Data Mask
SRAM_CE_N	PIN_AC11	SRAM Chip Enable

مرحله حافظه نهان (Cache)

ما در این پردازنده، بین پردازنده ARM و حافظه خارجی SRAM از یک حافظه نهان Cache استفاده کرده ایم. از آن جایی که زمان خواندن و نوشتن در حافظه خارجی چند کلاک طول می کشد و به هنگام دسترسی به SRAM برای خواندن یا نوشتن نیاز است که پردازنده freeze شود پس دسترسی به حافظه خارجی به عنوان یک گلوگاه در کارایی پردازنده مطرح می شود. برای حل این مشکل و

بهتر کردن کارایی پردازنده از حافظه بیار سریع نهان استفاده می کنیم که قابلیت دسترسی در یک کلاک را دارد و هنگام دسترسی به حافظه نهان نیاز به freeze کردن پردازنده نیست.

ویژگی های حافظه نهان پیاده سازی شده:

- دارای معماری Two-way Set Associates
- اندازه set ها 64 تا
- اندازه هر بلوک حافظه 64 بیت
- اندازه هر کلمه 32 بیت
- اندازه حافظه نهان 1 کیلوبایت برای دیتا
- گذرگاه حافظه 19 بیت
- تعداد بیت tag ها 10 تا
- تعداد بیت index ها 6 بیت
- دارای 1 بیت valid برای هر بلوک حافظه
- دارای سیاست جایگزینی حافظه LRU و یک بیت used برای این هدف (قدیمی ترین داده استفاده شده باید به روز شود.)
- دارای سیاست بازنویسی داخلی رویه نویسی کامل (Write Through)

ما برای پیاده سازی، ماژول CacheController را بین حافظه خارجی SRAM و پردازنده ARM قرار می دهیم. پردازنده برای دسترسی به حافظه ابتدا به حافظه نهان رجوع می کند. در صورتی که داده را پیدا کند به آن خواسته پاسخ می دهد ولی در صورتی که داده را نیافت به حافظه اصلی رجوع می کند. اگر درخواست خواندن از حافظه باشد، ابتدا یک بلوک حافظه از حافظه SRAM به حافظه Cache منتقل می شود و سپس به درخواست حافظه پردازنده پاسخ داده می شود. اما در صورتی که درخواست نوشتن در حافظه باشد، تنها در حافظه SRAM نوشته می شود و به داده به حافظه نهان وارد نمی شود.

CacheController.v

```
1   `timescale 1ns/1ns
2
3   module CacheController(
4       input clk,
5       input rst,
6
7       // memory stage unit
8       input [31:0] address,
9       input [31:0] wdata,
10      input MEM_R_EN,
11      input MEM_W_EN,
12      output [31:0] rdata,
13      output ready,
14
15      //SRAM controller
16      input [63:0] sram_rdata,
17      input sram_ready,
18      output reg [31:0] sram_address,
19      output reg [31:0] sram_wdata,
20      output sram_read_en,
21      output sram_write_en
22  );
```

مقدار ورودی address بیانگر مقدار آدرس حافظه برای دسترسی است. مقدار سیگنال های MEM_R_EN و MEM_W_EN بیانگر فعال بودن یا نبودن به ترتیب خواندن از حافظه و نوشتن در حافظه هستند. مقدار wdata بیانگر مقداری است که در حافظه Sram باید نوشته شود.

مقدار sram_rdata بیانگر مقداری است که از حافظه Sram خوانده شده است. و سیگنال sram_ready بیانگر آماده بودن یا نبودن حافظه Sram است.

مقدار خروجی ready در صورت آماده بودن CacheController صادر می شود. مقدار rdata بیانگر مقداری است که از حافظه Cache یا Sram خوانده شده است. مقدار sram_address بیانگر خانه حافظه متناظر در sram برای دسترسی است. مقدار sram_wdata بیانگر مقداری است که در حافظه اصلی Sram باید نوشته شود. دو سیگنال خروجی sram_read_en و sram_write_en بیانگر فعال بودن امکان به ترتیب خواندن از حافظه و نوشتن در حافظه هستند.

```
32
33      assign adr = address - 1024;
34      assign cache_address = adr[17:2];
35      assign cache_read_en = (ps == IDLE);
36      assign is_store = (ps == IDLE && ns == WRITE);
37      assign cache_write_en = (ps == READ && sram_ready);
38      assign ready = (ns == IDLE);
39      assign sram_write_en = (ps == WRITE) ? 1'b1 : 1'b0;
40      assign sram_read_en = (ps == READ) ? 1'b1 : 1'b0;
41      assign rdata = (ps == IDLE && hit) ?
42          cache_read_data : (ps == READ && sram_ready) ? (cache_address[0] ? sram_rdata[63:32] : sram_rdata[31:0]): 32'bz;
43
44      always @(posedge clk, posedge rst) begin
45          if (rst) ps <= IDLE;
46          else ps <= ns;
47      end
48
```

در always عکس بالا، بر اساس لبه بالا رونده کلاک، state های state machine را جلو می بریم. و در صورت فعال بودن سیگنال rst به IDLE state بر می گردیم.

در دو تا assign اول، آدرس متناظر خانه حافظه در Cache را با منهای 1024 کردن و سپس دو بیت به چپ شیفت دادن بدست آوردیم.

اگر در IDLE state باشیم، مقدار سیگنال cache_read_en را برابر یک می کنیم تا بتوانیم از حافظه نهان بخوانیم. اگر در READ state باشیم و مقدار سیگنال sram_ready فعال باشد، یعنی از حافظه اصلی، داده را خوانده ایم و Sram آماده است، پس سیگنال cache_write_en را فعال می کنیم تا داده را در حافظه نهان بنویسیم.

سیگنال ready را در صورتی فعال می کنیم که state بعدی IDLE باشد و از state های دیگر به این state برگردیم. چون در صورتی که از state های دیگر به این state برگردیم، یعنی کار با حافظه نهان به پایان رسیده است. سیگنال is_store در صورتی فعال می شود که دستور نوشتن در حافظه را داشته باشیم، یعنی state فعلی برابر IDLE باشد و state بعدی برابر WRITE باشد.

سیگنال های sram_write_en و sram_read_en در صورتی فعال می شوند که به ترتیب در state های WRITE و READ باشیم.

مقدار rdata بدین صورت به دست می آید که:

- در IDLE state هستیم و hit رخ داده است؛ پس این مقدار برابر با مقدار خوانده شده از حافظه نهان یا cache_read_data است.

- حالت قبل رخ نداده است؛ پس اگر در READ state هستیم و مقدار سیگنال sram_ready برابر یک است، یعنی دیتا خوانده شده از حافظه اصلی آماده است. حال اگر کم ارزش ترین بیت cache_address برابر یک بود، بلوک چپ و در غیر این صورت بلوک راست دیتای 64 بیتی خوانده شده را در خروجی rdata قرار می دهیم.

در این ماژول، از ماژول Cache یک instance گرفته شده است که در ادامه توضیح داده خواهد شد.

```
48
49      always @(ps, MEM_R_EN, hit, MEM_W_EN, sram_ready) begin
50          case (ps)
51              IDLE: begin
52                  ns = ps;
53                  if (MEM_R_EN && ~hit) ns = READ;
54                  else if (MEM_W_EN) ns = WRITE;
55              end
56
57              READ: begin
58                  if (sram_ready) ns = IDLE;
59              end
60
61              WRITE:  begin
62                  if (sram_ready) ns = IDLE;
63              end
64
65              default : ns <= IDLE;
66          endcase
67      end
68
69      always @(ps) begin
70          {sram_address, sram_wdata} = 64'bz;
71          if (ps == READ || ps == WRITE) sram_address = address;
72          if (ps == WRITE) sram_wdata = wdata;
73      end
74
```

بر اساس always ابتدایی که برای جا به جایی state های state machine به کار رفته، داریم:

- اگر در IDLE state باشیم، اگر سیگنال MEM_R_EN فعال باشد و hit رخ نداده باشد، به READ state می رویم. در غیر این صورت اگر سیگنال MEM_W_EN فعال باشد به WRITE state می رویم. و در غیر این صورت در این state می مانیم.
- اگر در READ state باشیم، اگر سیگنال sram_ready فعال باشد، به IDLE state برمی گردیم.
- اگر در WRITE state باشیم، اگر سیگنال sram_ready فعال باشد، به IDLE state برمی گردیم.

بر اساس always دوم داریم که اگر در state های READ یا WRITE باشیم، یعنی با حافظه Sram کار داریم؛ پس مقدار address را در sram_address قرار می دهیم. و نیز اگر در WRITE state باشیم، یعنی باید دیتایی را در حافظه اصلی Sram بنویسیم؛ پس مقدار wdata را در sram_wdata قرار می دهیم.

- **ماژول Cache:**

Cache.v

```
1   `timescale 1ns/1ns
2
3   module Cache(
4       input clk, rst,
5       input read_en, write_en, is_store,
6       input [16:0] address,
7       input [63:0] SRAM_data,
8       output reg [31:0] read_data,
9       output hit
10  );
```

این ماژول در قسمت CacheController به کار رفته است.

سیگنال های ورودی read_en و write_en بیانگر فعال بودن امکان خواندن از حافظه Cache و نوشتن در حافظه Cache است.

سیگنال is_store بیانگر قرار است دیتایی در حافظه اصلی Sram نوشته شود. مقدار Sram_data بیانگر مقدار بلوک 64 بیتی از حافظه است که باید در بلوک حافظه hit نشده Cache نوشته شود. مقدار address بیانگر خانه متناظر حافظه در Cache است.

سیگنال خروجی hit مشخص کننده پیدا شدن یا نشدن دیتا در حافظه نهان است تا در صورت پیدا نشدن، به حافظه اصلی Sram رجوع شود. مقدار خروجی read_data بیانگر دیتای خوانده شده (در صورت رخ داد hit) از حافظه Cache است.

```
12      reg [31:0] datas_left [0:1][0:63];
13      reg [31:0] datas_right [0:1][0:63];
14
15      reg [9:0] tags_left [0:63];
16      reg [9:0] tags_right [0:63];
17
18      reg valids_left [0:63];
19      reg valids_right [0:63];
20
21      reg used [0:63];
22
23      wire [9:0] tag = address[16:7];
24      wire [5:0] index = address[6:1];
25      wire offset = address[0];
```

برای دو بخش 32 بیتی چپ و راست هر بلوک 64 بیتی حافظه نهان، 10 بیت tag، یک بیت valid، یک بیت used برای سیاست جایگزینی تعریف شده است. که از 17 بیت آدرس این بخش، 10 بیت پر ارزش همان tag آدرس، کم ارزش ترین بیت همان offset آدرس، و بقیه بیت ها همان index آدرس هستند.

```
26
27      integer i;
28      initial begin
29          for (i = 0; i <= 63; i = i + 1) begin
30              used[i] = 1'b0;
31              tags_left[i] = 10'b0;
32              tags_right[i] = 10'b0;
33              valids_left[i] = 1'b0;
34              valids_right[i] = 1'b0;
35          end
36      end
37
```

در این بخش مقدار اولیه صفر در فیلد های حافظه نهان ریخته شده است. در ادامه در always با هر بار rst شدن پردازنده، این فیلد ها مقدار اولیه صفر را دریافت می کنند.

```
37
38      wire hit1, hit2;
39      assign hit1 = (tags_left[index] == tag) && valids_left[index];
40      assign hit2 = (tags_right[index] == tag) && valids_right[index];
41      assign hit = hit1 || hit2;
42
```

در این بخش بر اساس index بدست آمده از آدرس، tag آدرس را با tag متناظر index به دست آمده از آدرس در حافظه نهان مقایسه می کنیم. در صورتی که برابر بودند، یک بودن بیت valid متناظر index به دست آمده از آدرس را بررسی می کنیم. در صورت برقرار بودن این دو شرط برای هر کدام از بلوک های چپ یا راست یک بلوک حافظه، hit چپ یا راست رخ داده است. در صورت فعال شدن یکی از بیت های hit چپ یا راست، در حافظه Cache عملیات hit رخ داده است.

```
56          else begin
57
58              if (read_en) begin
59
60                  if (hit) begin
61                      if (hit1) begin
62                          read_data = datas_left[offset][index];
63                          used[index] = 1'b1;
64                      end
65                      else if (hit2) begin
66                          read_data = datas_right[offset][index];
67                          used[index] = 1'b0;
68                      end
69                  end
70              end
71
```

در این بخش در صورت فعال بودن امکان خواندن از حافظه Cache، اگر hit رخ داد باشد، بر اساس hit در بلوک چپ یا راست، مقدار دیتا را از حافظه Cache روی خروجی read_data قرار می دهیم. و بیت used را به بلوک مقابل به روز رسانی می کنیم تا در دسترسی بعدی برای نوشتن در حافظه سیاست جایگزینی را رعایت کرده باشیم.

Cache.v

```
72          if (is_store && hit) begin
73              if (hit1) begin
74                  valids_left[index] = 1'b0;
75                  used[index] = 1'b0;
76              end
77              else if(hit2) begin
78                  valids_right[index] = 1'b0;
79                  used[index] = 1'b1;
80              end
81          end
```

در این بخش، اگر hit رخ داده باشد و سیگنال is_store یک باشد و لازم باشد در حافظه اصلی، دیتایی را بنویسیم، در این جا بلوک متناظرش را در حافظه نهان invalid می کنیم و بیت used متناظر بلوک را برای سیاست جایگزینی به روز می کنیم.

```
83          if (write_en) begin
84
85              if (used[index] == 1'b1) begin
86                  valids_right[index] = 1'b1;
87                  used[index] = 1'b0;
88
89                  datas_right[0][index] = SRAM_data[31:0];
90                  datas_right[1][index] = SRAM_data[63:32];
91
92                  tags_right[index] = tag;
93              end
94
95              else if (used[index] == 1'b0) begin
96                  valids_left[index] = 1'b1;
97                  used[index] = 1'b1;
98
99                  datas_left[0][index] = SRAM_data[31:0];
100                 datas_left[1][index] = SRAM_data[63:32];
101
102                 tags_left[index] = tag;
103             end
```

در این بخش در صورت فعال بودن سیگنال write_en یعنی باید دیتایی را در حافظه نهان بنویسیم. بر اساس آخرین دسترسی به این حافظه نهان و بیت used بلوک های چپ و راست، در بلوک مقابلش که قدیمی تر است می نویسیم. سپس 10 بیت tag را در محل مربوطه می نویسیم. بیت used را به روز رسانی می کنیم. و بیت valid بلوک را فعال می کنیم.

مرحله Top Level (پردازنده ARM)

در این مرحله همه ماژول های مورد نیاز برای پیاده سازی پردازنده ARM، در کنار هم قرار می گیرند و wiring مورد نظر بین آن ها انجام می شود. پردازنده پیاده سازی شده دارای بخش های زیر می باشد.

- دارای پنج stage:
  - مرحله Instruction Fetch
  - مرحله Instruction Decode
  - مرحله Execution
  - مرحله Memory
  - مرحله Write Back
- بخش تشخیص مخاطره داده ای
- بخش فوروارد کردن داده
- بخش حافظه خارجی Sram واقع در خارج از پردازنده
- بخش حافظه نهان Cache واقع در بخش Memory پردازنده

در ادامه به نحوه پیاده سازی این پردازنده می پردازیم.

ورودی ها و خروجی های پردازنده:

ARM.v

```
1   `timescale 1ns/1ns
2
3   module ARM(
4       input clk, rst, Forward_en,
5       inout[63:0] SRAM_DQ,
6       output[16:0] SRAM_ADDR,
7       output SRAM_WE_N
8   );
```

بخش Instruction Fetch:

```
    or fr(freeze, ~sram_ready, hazard);
    IF instruction_fetch(
        clk, rst, freeze, Branch_taken, Branch_taken,
        Branch_Address,
        PC_if, Instruction
    );
```

بخش Instruction Decode:

```
    ID instruction_decode(
        clk, rst, Branch_taken, sram_ready,
        // From IF
        Instruction, PC_if,
        // From WB
        WB_EN,
        wb_dest,
        wb_out,
        // From hazard
        hazard,
        // From Status Register
        SR,
        src1_id, src2_id, src1_id_reg, src2_id_reg,
        Two_src_id,
        WB_EN_id, MEM_R_EN_id, MEM_W_EN_id, Branch_taken, S_id, hasSrc1_id,
        exe_cmd_id,
        PCid,
        Val_Rn_id, Val_Rm_id,
        imm_id,
        Shift_operand_id,
        Signed_imm_24_id,
        dest_id
    );
```

بخش Execution:

```
    EXE execute(
        clk, rst, sram_ready,
        imm_id, MEM_R_EN_id, MEM_W_EN_id, WB_EN_id,
        Sel_src1, Sel_src2,
        exe_cmd_id, dest_id, SR,
        Shift_operand_id,
        Signed_imm_24_id,
        PCid, Val_Rn_id, Val_Rm_id, ALU_result_exe, wb_out,
        MEM_R_EN_exe, MEM_W_EN_exe, WB_EN_exe,
        Dest_exe, SR_exe,
        ALU_result_exe, Branch_Address, Val_rm_exe
    );
```

بخش Memory:

```
76      MEM mem(
77          clk, rst, sram_ready,
78          WB_EN_exe, MEM_R_EN_exe, MEM_W_EN_exe,
79          ALU_result_exe, Val_rm_exe,
80          Dest_exe,
81          WB_EN_mem, MEM_R_EN_mem,
82          Dest_mem,
83          ALU_result_mem, Mem_read_value_mem,
84          sram_ready,
85          SRAM_DQ,
86          SRAM_ADDR,
87          SRAM_WE_N
88      );
```

بخش Hazard Detection:

```
89
90      Hazard_Detection_Unit hazard_decetion(
91          WB_EN_id, WB_EN_exe, Two_src_id, Forward_en, MEM_R_EN_id, hasSrc1_id,
92          src1_id, src2_id, dest_id, Dest_exe,
93          hazard
94      );
95
```

بخش Status Register:

```
95
96      Status_Reg status_register(
97          clk, rst, S_id,
98          SR_exe,
99          SR
100     );
101
```

بخش Write Back:

```
102     WB write_back(
103         clk, rst,
104         ALU_result_mem, Mem_read_value_mem,
105         Dest_mem,
106         MEM_R_EN_mem, WB_EN_mem,
107         wb_out,
108         wb_dest,
109         WB_EN
110     );
```

بخش Forwarding:

```
111
112     ForwardingUnit forwarding(
113         WB_EN_exe, WB_EN_mem, Forward_en,
114         src1_id_reg, src2_id_reg, Dest_exe, wb_dest,
115         Sel_src1, Sel_src2
116 );
```

که clk , rst را از ماژول Test Bench می گیرد.

سیگنال Forward_en بیانگر فعال بودن یا نبودن امکان فوروارد کردن داده در پردازنده است.

مقدار SRAM_DQ بیانگر دو حالت است، یا یک بلوک بیتی 64 حافظه برای هنگام hit نشدن حافظه نهان است که باید در حافظه نهان نوشته شود؛ یا اینکه یک بلوک 64 بیتی داده است که باید در Sram نوشته شود.

مقدار SRAM_ADDR بیانگر آدرس خانه ای از حافظه خارجی Sram است.

مقدار سیگنال SRAM_WE_N بیانگر فعال بودن یا نبودن امکان نوشتن در حافظه Sram است؛ که این سیگنال در صورت صفر بودن فعال می شود.

مرحله Test Bench

در این بخش یک instance از پردازنده و حافظه خارجی می گیریم و آن ها را تست می کنیم.

TestBench.v

```
3   module TB();
4
5       wire[63:0] SRAM_DQ;
6       wire[16:0] SRAM_ADDR;
7       wire SRAM_WE_N;
8       reg clk, rst, Forward_en, long_clk;
9       ARM arm(clk, rst, Forward_en, SRAM_DQ, SRAM_ADDR, SRAM_WE_N);
10      SRAM ram(long_clk, rst, SRAM_WE_N, SRAM_ADDR, SRAM_DQ);
11
```

```
12      initial begin
13          long_clk = 1'b0;
14          repeat (500) begin
15              #20 long_clk = ~long_clk;
16          end
17      end
18
19      initial begin
20          clk = 1'b0;
21          rst = 1'b0;
22          Forward_en = 1'b1;
23
24          #10 clk = 1'b1;
25          #10 rst = 1'b1;
26          #10 clk = 1'b0;
27          #10 rst = 1'b0;
28          #10 clk = 1'b1;
29
30          repeat (1000) begin
31              #10 clk = ~clk;
32          end
33      end
```

مقدار سیگنال clk برای پردازنده ARM و مقدار سیگنال long_clk برای حافظه خارجی Sram استفاده می شود.

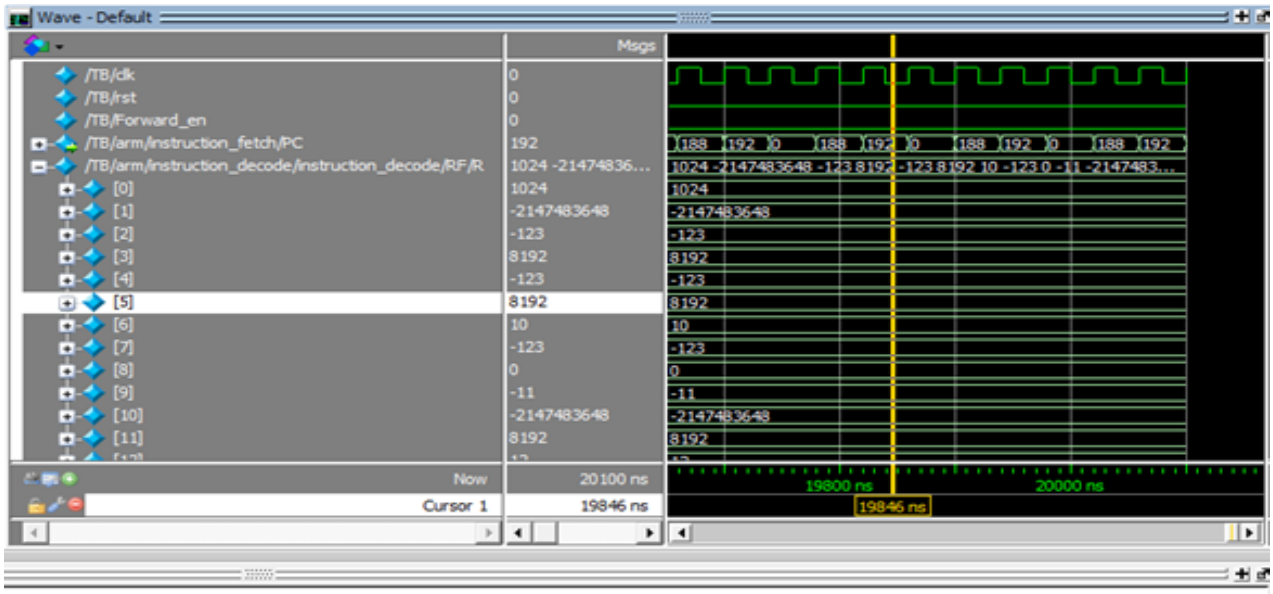
حالت فوروارد کردن دیتا هم فعال است.

مقدار سیگنال rst در ابتدا فعال و پس از یک clk غیر فعال می شود.

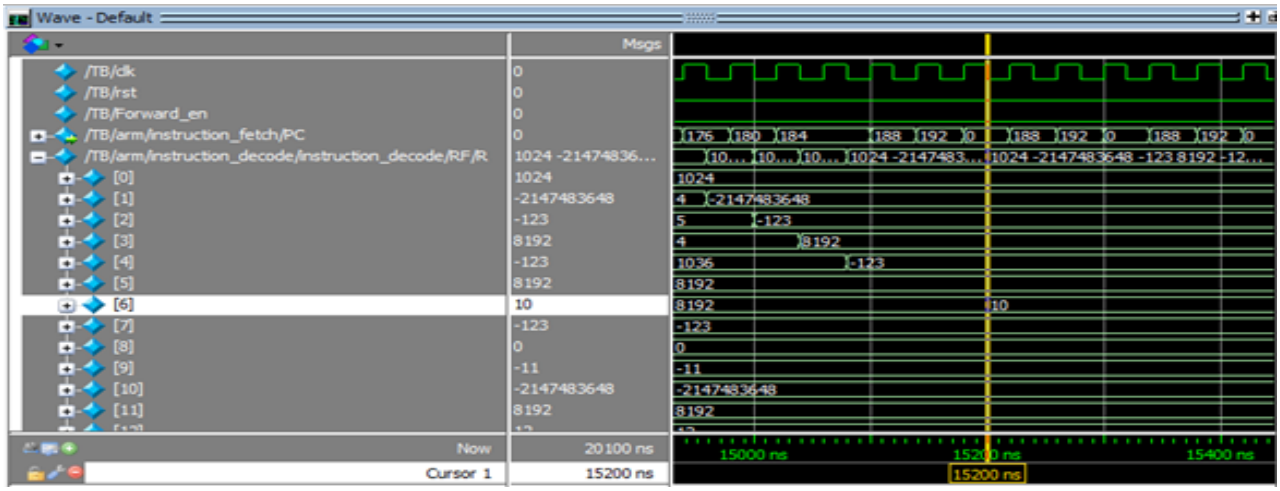
نتایج شبیه سازی (تعداد سیکل و کارایی)

- **نتایج پردازنده ARM بدون فورواردینگ و SRAM:**

ابتدا صحت خروجی را بررسی می کنیم:



مشاهده می شود در انتهای شبیه سازی مقادیر رجیسترها مرتب شده هستند و مقدارهای مورد انتظار ما را دارند. حال باید تعداد کلاک ها را بیابیم. برای اینکار ابتدا باید ببینیم محاسبه نتیجه چه مقدار طول کشیده است. برای این کار لحظه ای که مقدار رجیستر R6 برابر ده شده است را می یابیم:

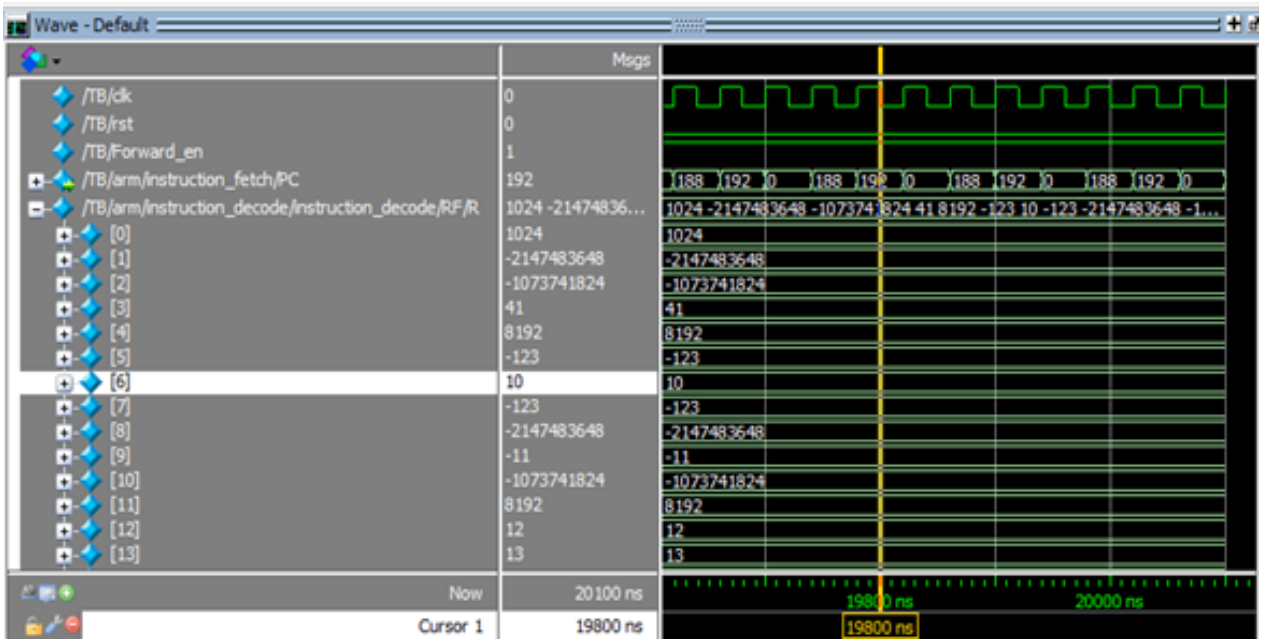


همانطور که مشاهده می شود در لحظه 15200 نانو ثانیه این اتفاق می افتد. با توجه به testbench نوشته شده، صد نانو ثانیه در ابتدای تست برای ریست کردن سیستم بوده است. همچنین طول هر کلاک 40 نانو ثانیه است. در نتیجه تعداد کلاک هایی که طول می کشد تا برنامه به پایان برسد برابر 378 کلاک است. توجه کنید در زمان 15200 نانو ثانیه که مقدار در رجیستر نوشته می شود شاهد لبه پایین رونده کلاک هستیم. $CPI = 378 / 47 = 8.04$

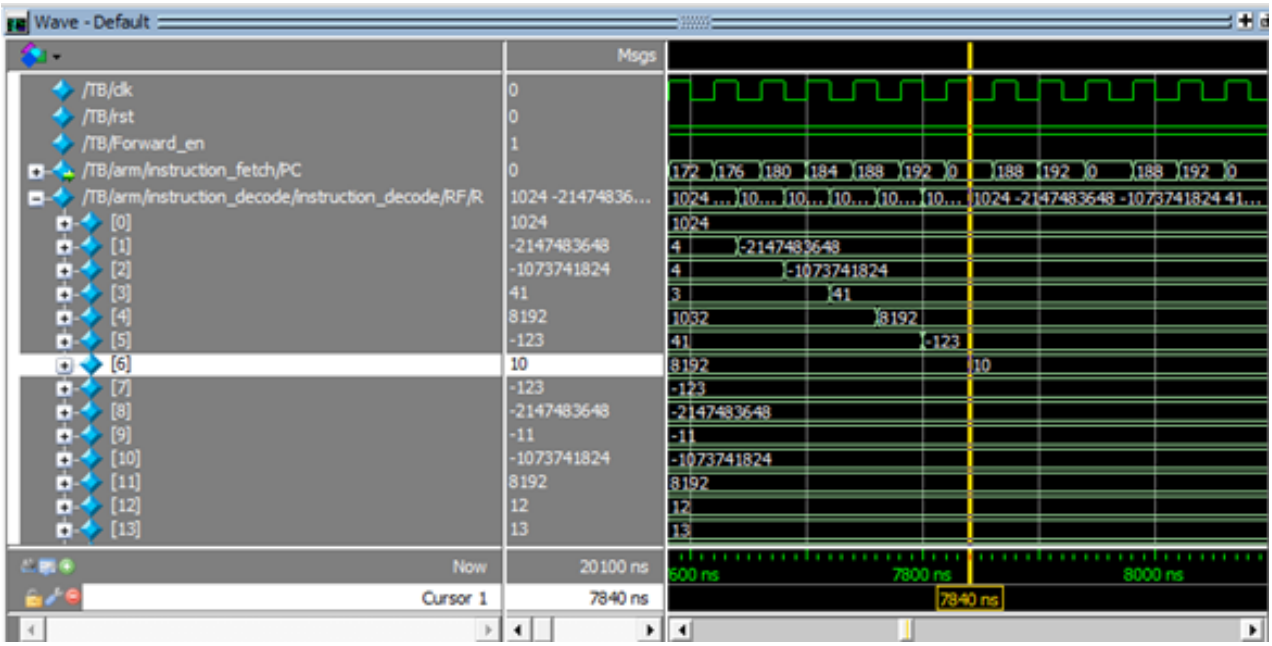
تعداد کلاک بدون فورواردینگ : 378

- **نتایج پردازنده ARM همراه با Forwarding:**

ابتدا صحت خروجی را بررسی میکنیم:



مشاهده می شود در انتهای شبیه سازی مقادیر رجیسترها مرتب شده هستند و مقدارهای مورد انتظار ما را دارند. حال باید تعداد کلاک ها را بیابیم. برای اینکار ابتدا باید ببینیم محاسبه نتیجه چه مقدار طول کشیده است. برای این کار لحظه ای که مقدار رجیستر R6 برابر ده شده است را می یابیم:



همانطور که مشاهده می شود در لحظه 7840 نانو ثانیه این اتفاق می افتد. با توجه به testbench نوشته شده، صد نانوثانیه در ابتدای تست برای ریست کردن سیستم بوده است. همچنین طول هر کلاک 40 نانو ثانیه است. در نتیجه تعداد کلاک هایی که طول می کشد تا برنامه به پایان برسد برابر 172 کلاک است. توجه کنید در زمان 7840 نانو ثانیه که مقدار در رجیستر نوشته می شود شاهد لبه پایین رونده کلاک هستیم.

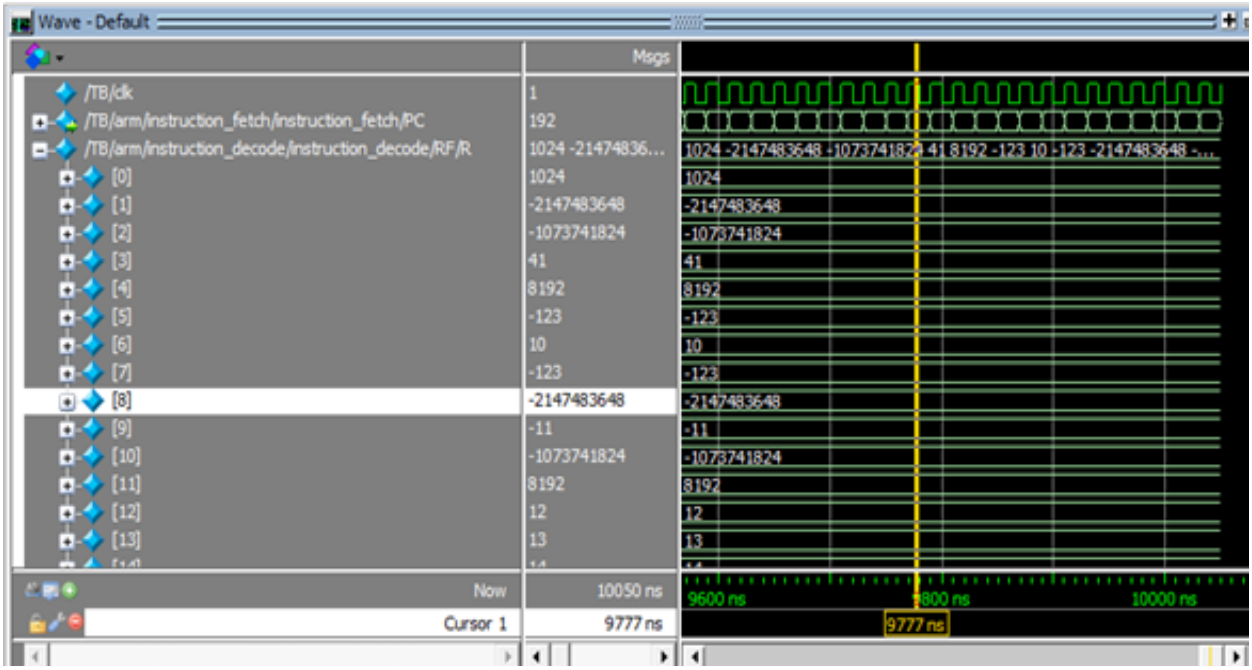
$$CPI = 172 / 47 = 3.65$$

تعداد کلاک همراه فورواردینگ : 172

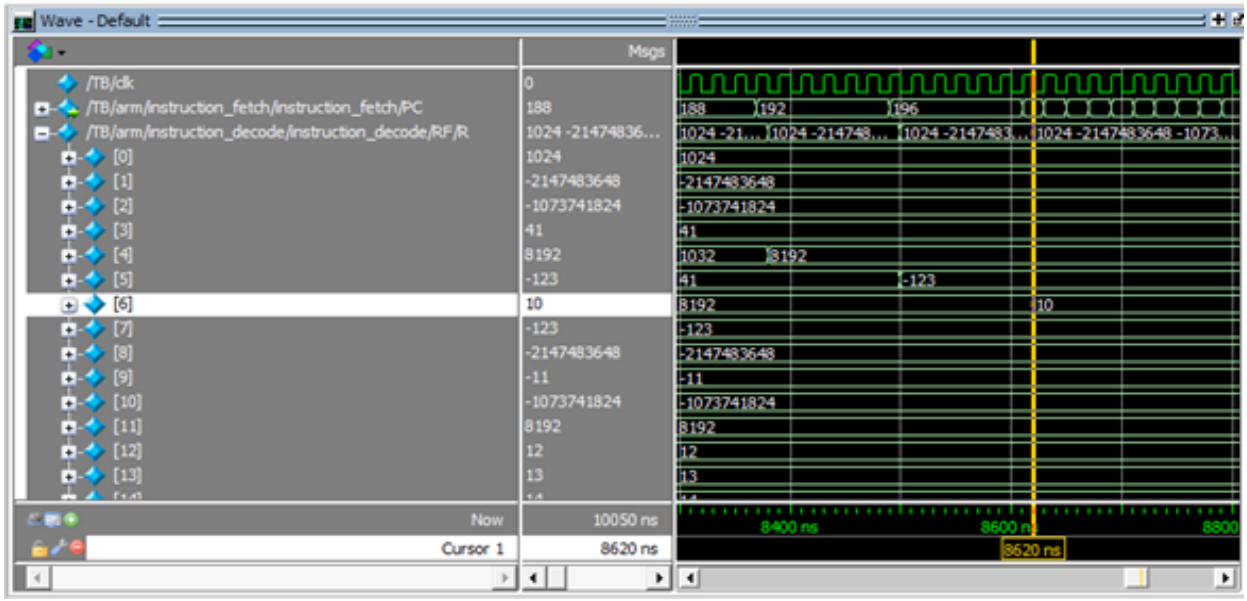
یعنی کارایی پردازنده حدود 94 درصد بهبود یافته است.

- **نتایج پردازنده ARM همراه با Forwarding و SRAM:**

ابتدا صحت خروجی را بررسی میکنیم:



مشاهده می شود در انتهای شبیه سازی مقادیر رجیسترها مرتب شده هستند و مقدارهای مورد انتظار ما را دارند. حال باید تعداد کلاک ها را بیابیم. برای اینکار ابتدا باید ببینیم محاسبه نتیجه چه مقدار طول کشیده است. برای این کار لحظه ای که مقدار رجیستر R6 برابر ده شده است را می یابیم:



همانطور که مشاهده می شود در لحظه 8620 نانو ثانیه این اتفاق می افتد. با توجه به testbench نوشته شده، صد نانو ثانیه در ابتدای تست برای ریست کردن سیستم بوده است. همچنین طول هر کلاک 20 نانو ثانیه است. در نتیجه تعداد کلاک هایی که طول میکشد تا برنامه به پایان برسد برابر 426 کلاک است. توجه کنید در زمان 8620 نانو ثانیه که مقدار در رجیستر نوشته می شود شاهد لبه پایین رونده کلاک هستیم.

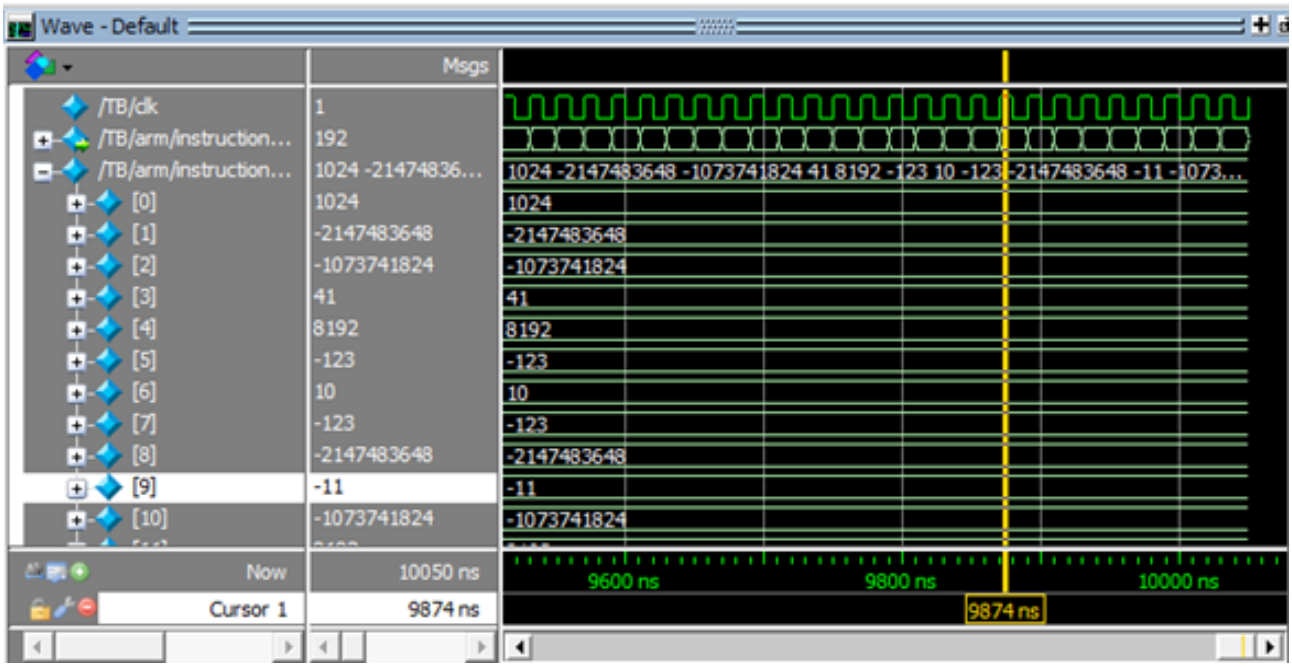
$$CPI = 426 / 47 = 9.06$$

تعداد کلاک همراه SRAM : 426

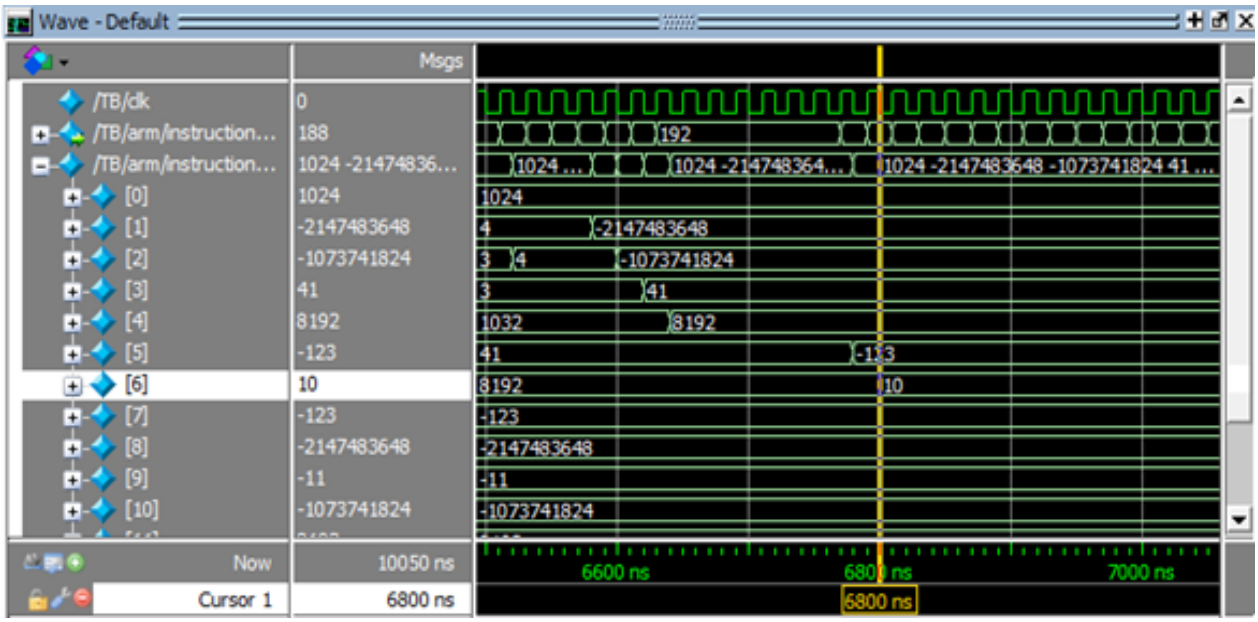
همانطور که انتظار داشتیم نتیجه نسبت به دو حالت قبل بدتر شده است. نسبت به حالت اول 15 درصد و نسبت به حالت دوم شصت درصد افت کارایی داشتیم. برای حل مشکل کارایی حافظه نهان را به سیستم اضافه خواهیم کرد.

نتایج پردازنده ARM همراه با Forwarding و SRAM و حافظه نهان:

ابتدا صحت خروجی را بررسی میکنیم:



مشاهده می شود در انتهای شبیه سازی مقادیر رجیسترها مرتب شده هستند و مقدارهای مورد انتظار ما را دارند. حال باید تعداد کلاک ها را بیابیم. برای اینکار ابتدا باید ببینیم محاسبه نتیجه چه مقدار طول کشیده است. برای این کار لحظه ای که مقدار رجیستر R6 برابر ده شده است را می یابیم:



همانطور که مشاهده می شود در لحظه 6800 نانو ثانیه این اتفاق می افتد. با توجه به testbench نوشته شده، صد نانو ثانیه در ابتدای تست برای ریست کردن سیستم بوده است. همچنین طول هر کلاک 20 نانو ثانیه است. در نتیجه تعداد کلاک هایی که طول میکشد تا برنامه به پایان برسد برابر 335 کلاک است. توجه کنید در زمان 6800 نانو ثانیه که مقدار در رجیستر نوشته می شود شاهد لبه پایین رونده کلاک هستیم.

$$CPI = 335/\ 47 = 7.12$$

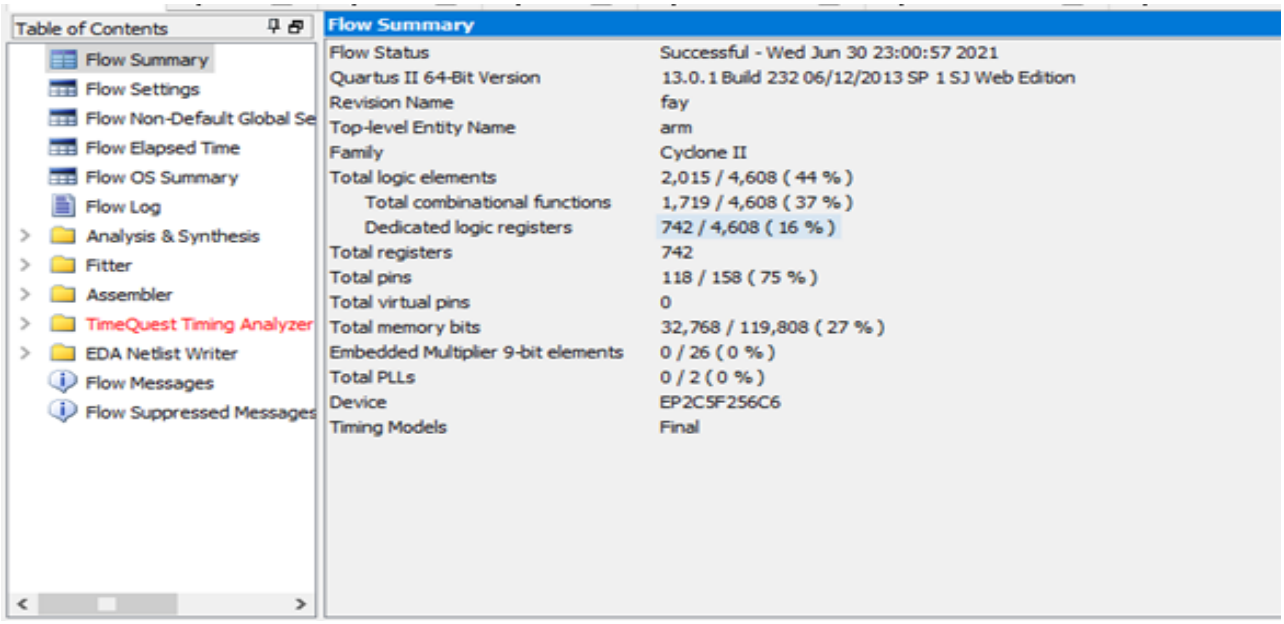
تعداد کلاک همراه 335 : SRAM

پس نسبت به حالتی که از حافظه نهان استفاده نمی کنیم، تقریبا 28 درصد بهبود کارایی داریم.

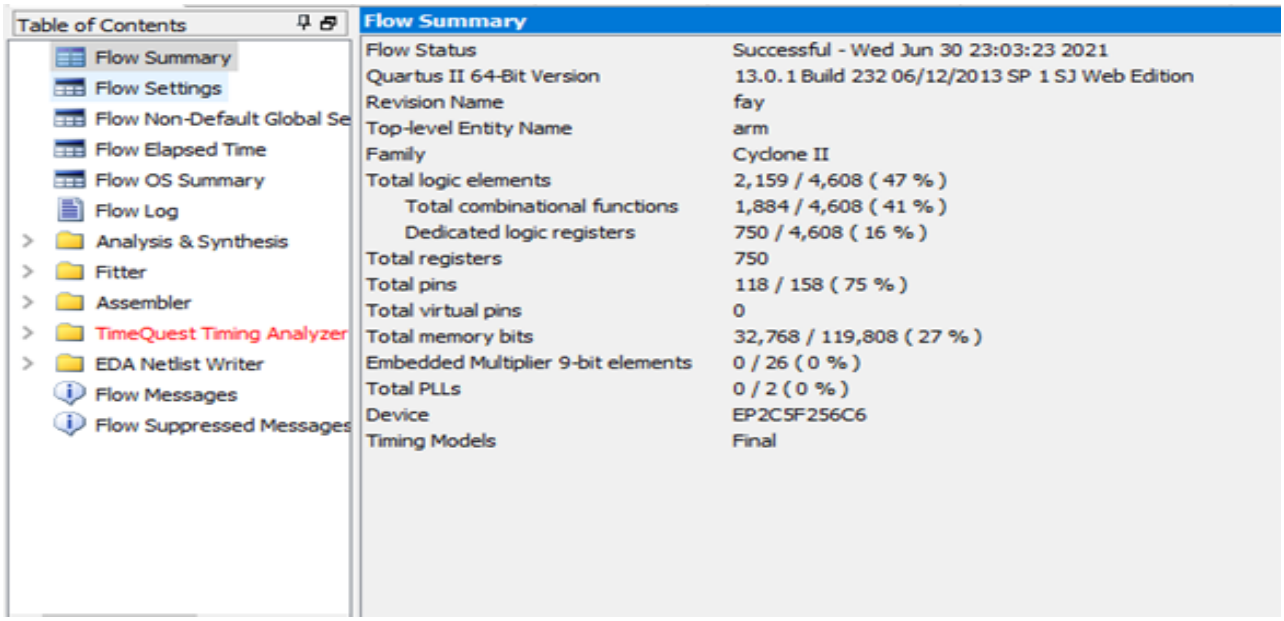
نتایج سنتز

نتایج سنتز هر فاز در ادامه آورده شده است. تعداد بلاک های منطقی استفاده شده در هر فاز مشخص است.

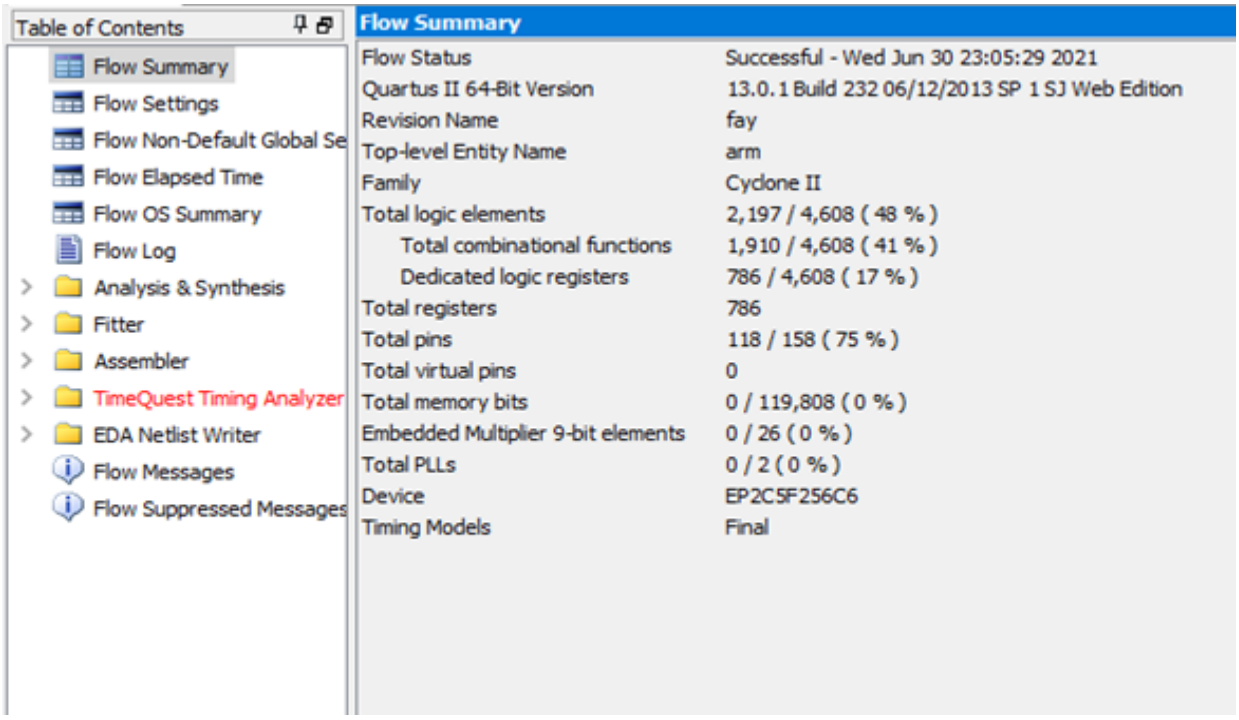
فاز اول:



فاز دوم:



فاز سوم:



فاز چهارم: