# DIGITAL SYSTEMS 2 – REPORT

Anita Ghandehari 810195533

Soheil Shirvani 810195416

Anita Ghandehari
810195533

Soheil Shirvani
810195416

# فهرست مطالب:

# توضیح آزمایش:

**در این آزمایش ما سعی داریم تا یک پردازنده ی آرم را با کد وریلاک پیاده سازی کنیم و آن را تست و سیمولیت کنیم.**

**این پردازنده یک پایپلاین 5 استیج ای است که انها به ترتیب شامل:**

1) Instruction Fetch (IF)
2) Instruction Decode (ID)
3) Execute Unit (EXE)
4) Memory Unit (MEM)
5) Write Back Unit (WB)

**است که هر کدام از این استیج ها دارای یک قسمت شامل ثبات های بعد از آن نیز هست و دارای 3 ماژول مجزا شامل:**

1) Hazard Unit
2) Status Register Unit
3) Forwarding Unit

**و همین طور دارای یک Controller که در استیج ID قرار دارد و یک Datapath کامل است.**

**در شکل های زیر می توانیم Datapath کلی مدار را نیز مشاهده کنیم:**

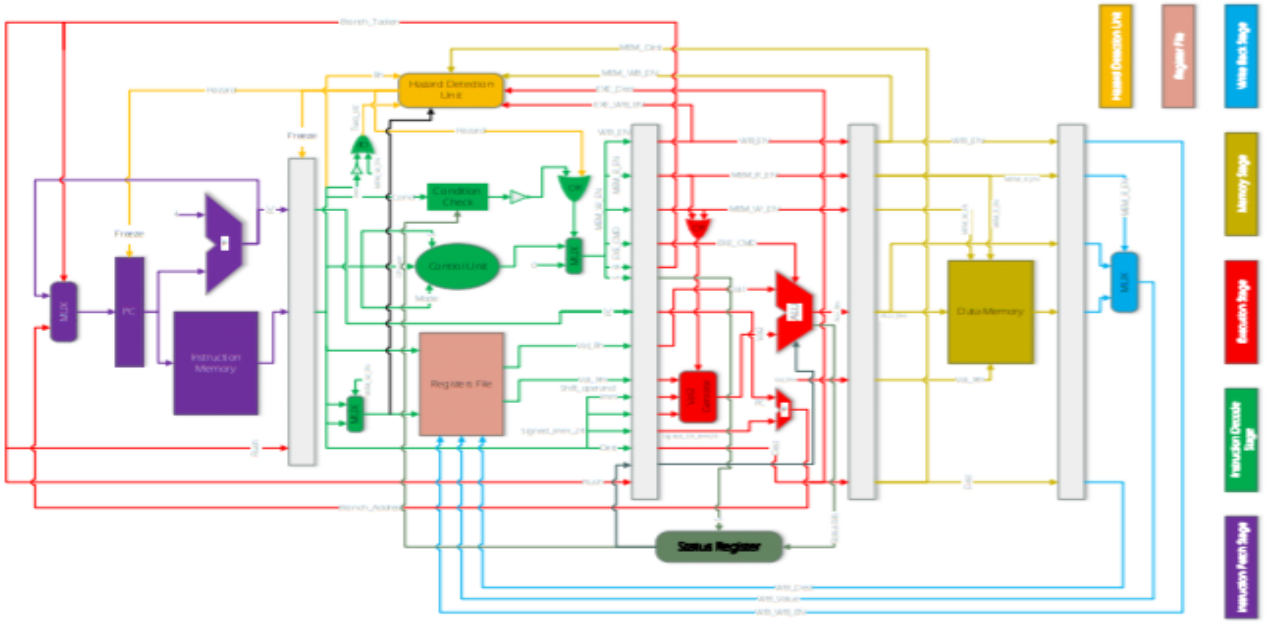



*Figure 1: شکل Datapath مدار بدون forwarding unit*

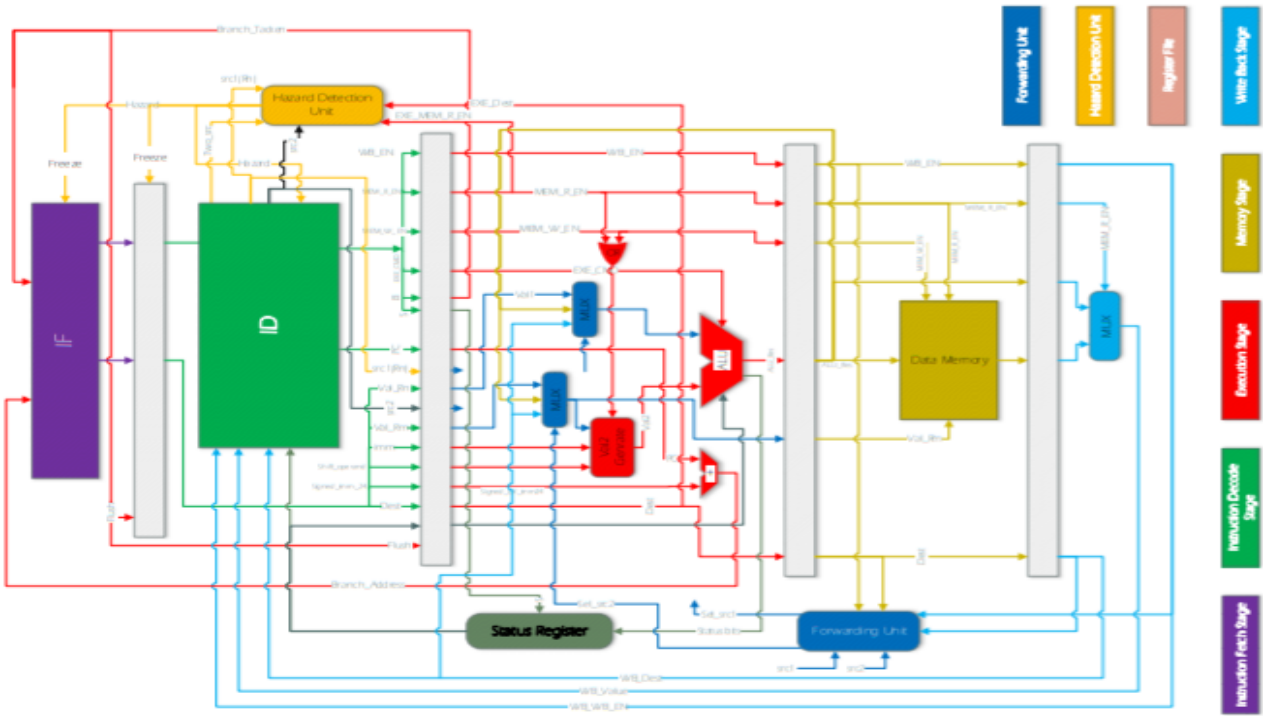



*Figure 2: شکل Datapath مدار با forwarding unit*

**حال برای هر کدام از واحد ها به اختصاص توضیح مربوطه را می دهیم.**

**و از مشخصات دیگر پردازنده میتوان:**

## مشخصات پردازنده

۱- پهنای خط داده: ۳۲ بیت

۲- تعداد مراحل خط لوله: ۵ مرحله‌ای

۳- تعداد دستورات: ۱۳ دستور

۴- میزان تاخیر انشعاب: ۲ مرحله

۵- ۱۶ ثبات همه منظوره ( ثبات ۱۵ به منظور PC استفاده می شود و ثبات ۱۴ نیز به عنوان Link Register)

۶- آدرس‌دهی برحسب بایت و فضای آدرس دستورات (Instructions) و داده (Data) تفکیک شده می‌باشد.
(آدرس ۰ تا ۱۰۲۳ به Program ROM اختصاص دارد و آدرس ۱۰۲۴ به بعد به RAM تعلق دارد.)

۷- تمامی پرش‌ها از نوع محلی تعریف شده است و پس از پرش مقدار رجیستر شمارنده دستور به شکل زیر خواهد بود.

PC=PC+ (signed_immed_24<<2) +4

قابلیت تشخیص و جلوگیری هازاد داده‌ای (Hazard Detection Unit) دارد و واحد ارسال به جلو (Forwarding Unit) ندارد.

## نام برد. و مجموعه دستورات آن:

| R-type Instructions | | Description | Bits | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 31:28 | 27:26 | 25 | 24:21 | 20 | 19:16 | 15:12 | 11:00 |
| | | | Cond. | Mode | I | OP-Code | S | Rn | Rd | shifter operand |
| 0 | NOP[11] | No Operation | 1110 | 00 | 0 | 0000 | 0 | 0000 | 0000 | 00000000000 |
| 1 | MOV | Move | cond | 00 | I | 1101 | S | 0000 | Rd | shifter operand |
| 2 | MVN[12] | Move NOT | cond | 00 | I | 1111 | S | 0000 | Rd | shifter operand |
| 3 | ADD | Add | cond | 00 | I | 0100 | S | Rn | Rd | shifter operand |
| 4 | ADC | Add with Carry | cond | 00 | I | 0101 | S | Rn | Rd | shifter operand |
| 5 | SUB | Subtraction | cond | 00 | I | 0010 | S | Rn | Rd | shifter operand |
| 6 | SBC | Subtract with Carry | cond | 00 | I | 0110 | S | Rn | Rd | shifter operand |
| 7 | AND | And | cond | 00 | I | 0000 | S | Rn | Rd | shifter operand |
| 8 | ORR | Or | cond | 00 | I | 1100 | S | Rn | Rd | shifter operand |
| 9 | EOR | Exclusive OR | cond | 00 | I | 0001 | S | Rn | Rd | shifter operand |
| 10 | CMP | Compare | cond | 00 | I | 1010 | 1 | Rn | 0000 | shifter operand |
| 11 | TST[13] | Test | cond | 00 | I | 1000 | 1 | Rn | 0000 | shifter operand |
| 12 | LDR | Load Register | cond | 01 | 0 | 0100 | 1 | Rn | Rd | offset_12 |
| 13 | STR | Store Register | cond | 01 | 0 | 0100 | 0 | Rn | Rd | offset_12 |
| 14 | B | Branch | cond | 10 | 1 | 0 | signed_immed_24 | | | |

*Figure 3: لیست دستورات*

## که در آن :

**Mode:** دسته دستور را تعیین می کند. تمامی دستورات محاسباتی در دسته 00 قرار می گیرند. دستورات حافظه در دسته‌ی 01 و دستورات پرش در دسته 10 قرار دارند. در این پردازنده ها دستورات ارتباط با پردازنده‌ی کمکی[۱۴] نیز در نظر گرفته شده است که Mode آن برابر 11 است.

**OP-Code:** کد دستورالعمل برای تعیین نوع دستور است. Mode به همراه OP-Code برای تشخیص دستورات در نظر گرفته می شود.

**I:** نشاندهنده فوری بودن عملوند دوم است، در صورت یک بودن داده دوم فوری در نظر گرفته می‌شود.

**S:** در صورت یک بودن S دستورات محاسباطی پس از اجرا ثبات وضعیت (state register) به روز می‌کنند.

**Cond:** در پردازنده‌های ARM تمامی دستورات به صورت شرطی اجرا می‌شوند. در جدول ۳ لیست حالت های اجرای دستورات ذکر شده است. در صورتی که یک دستور به صورت غیرشرطی اجرا شود مقدار بیت‌های شرط برابر 1110 خواهد بود. در صورتی که شرط برقرار نباشد دستور همانند NOP هیچ کاری انجام نخواهد داد. مقدار 1111 نیز در نسل‌های مختلف پردازنده های ARM به صورت متفاوتی اجرا می شود که در پردازنده مورد نظر در آزمایشگاه نیازی به پیاده سازی آن نیست.

| 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* | - |

*Figure 4: کد دستورات شرطی*

که در آن:

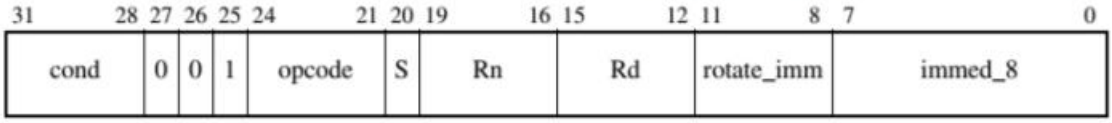
**Rd**: نشاندهنده ادرس ثبات مقصد است. این ادرس در دستور STR به عنوان یکی از مقداری که باید در حافظه ذخیره شود مورد استفاده قرار می‌گیرد.

**Rn**: همواره به عنوان یکی از عملوندهای دستورات مورد استفاده قرار می گیرد.

و برای shifter operand 3 مدل زیر را داریم:

1) 32 بیت شیفت عدد فوری (32-bit immediate):

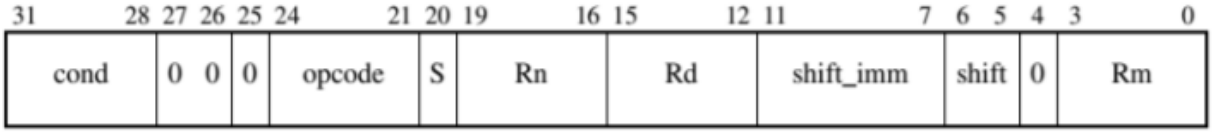
در این حالت مقدار بیت I برابر یک است. عدد ۸ بیتی immed_8 در یک ظرف ۳۲ بیت قرار می‌گیرد سپس به اندازه دو برابر rotate_imm به راست چرخانده می شود (شکل ۱).




2) شیفت فوری (immediate shift):

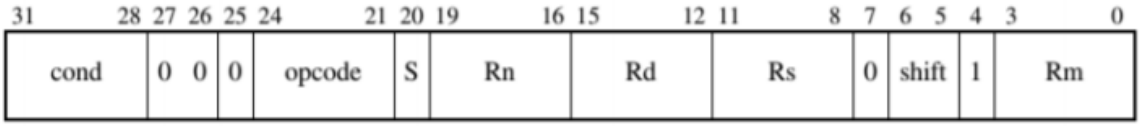
در این حالت بیت I و بیت چهارم دستورالعمل نیز برابر صفر است. عملوند دوم از رجیستر خوانده می شود. سپس عدد خوانده شده براساس حالت شیفت (shift) به مقدار shift_imm شیفت داده می‌شود (شکل ۳). حالت های شیفت در جدول زیر قرار دارد.

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




3) شیفت ثباتی (Register Shift):

در این حالت بیت I برابر صفر است و عملوند دوم از رجیستر خوانده می شود. پس از آن عدد خوانده شده براساس حالت شیفت (shift) به مقدار رجیستر Rs شیفت داده می‌شود (شکل ۴). در پردازنده مورد استفاده در آزمایشگاه نیازی به پیاده سازی این حالت نیست.



## و همین طور پردازنده ی آرم دارای دو نوع ثبات است که شامل:

### 1) ثبات های عمومی:

Register File در پردازنده ARM شامل ۱۶ ثبات ۳۲ بیت می شود که کاربردهای زیر را دارند:

- ثبات ۰ تا ۱۲ ثبات‌های عمومی پردازنده می‌باشند که در همه کاربردها استفاده می‌شوند.
- ثبات ۱۳ به عنوان اشاره‌گر پشته[۱۶] مورد استفاده قرار می گیرند. دستورات پشته مانند push و pop از این ثبات استفاده می کنند.
- ثبات ۱۴ به عنوان آدرس بازگشت پس از دستور BL استفاده می‌شود. دستور BL یا Branch and Link معادل دستور Call در پردازنده‌های دیگر است.
- ثبات ۱۵ به عنوان شمارنده برنامه مورد استفاده قرار می گیرد. در معماری ارائه شده در آزمایشگاه برای سادگی این رجیستر به مرحله واکشی دستور[۱۷] منتقل شده است.

### 2) ثبات وضعیت:

در پردازنده ARM یک ثبات برای نگهداری وضعیت کلی پردازنده در نظر گرفته شده است. این رجیستر Mode اجرای پردازنده و وضعیت اجرای دستورات در پردازنده را بیان می‌کند[۱۹]. بیت‌های N (منفی بودن)، Z (صفر بودن)، C (رقم نقلی) و V (سرریز[۲۰]) برای بررسی شرط مورد استفاده قرار می گیرد. در پیاده سازی پردازنده مورد نظر آزمایشگاه معماری کامپیوتر فقط بیت‌های N، Z، C و V پیاده‌سازی می‌شود. نوشتن در ثبات وضعیت با لبه پایین رونده انجام می‌شود.

| 31 | 30 | 29 | 28 | 27 | 26 25 | 24 | 23 20 | 19 16 | 15 10 | 9 | 8 | 7 | 6 | 5 | 4 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Z | C | V | Q | Res | J | RESERVED | GE[3:0] | RESERVED | E | A | I | F | T | M[4:0] |

**حال برای هر قسمت از مدار توضیحات مربوطه به همراه کد آن داده می شود.**

# توضیح قسمت IF:

این واحد وظیفه ی تولید PC که نشان دهنده ی شماره ی دستور ما و دادن دستور به بعد به مدار را دارد. این واحد دارای یک ثبات برای PC است که بعد از اجرای هر دستور در صورت نبود پرش، PC = PC + 4 می شود تا بتواند دستور بعد را اجرا کند و اگر دستور پرش در حال اجرا باشد مقدار Branch Address درون آن قرار می گیرد تا به دستور مربوطه پرش کند. سپس این مقدار وارد یک حافظه دستور می شود که تمامی دستورات در آن قرار می گیرند. آدرس این حافظه در واقع همان مقدار PC ما است. برای ایجاد PC + 4 نیاز به یک جمع کننده بدون کلاک و برای تعیین پرش بودن یا نبودن ( برای انتخاب مقدار PC برای هر دستور) نیاز به یک مولتی پلکسر داریم که مقدار انتخاب آن از Controller می آید. همچنین ثبات PC ما یک دستور Freeze نیز دارد تا در صورت نیاز بتواند PC را تغییر ندهد و در نتیجه مدار را Stall کند تا دستورات درون پایپ با اتمام برسند و مقادیر مورد نیاز در دستور بعد از دستور قبل محاسبه شوند. در حالت اول می توانیم مقادیر انتخاب مولتی پلکسر (Branch_Taken)، Freeze را صفر قرار دهیم و دستوراتی را از حافظه دستورات خوانده و اجرا کنیم. در این صورت توانسته ایم این واحد را تست کنیم. این قسمت مقدار PC و همین طور 32 بیت دستور خوانده شده از حافظه دستورات را به استیج بعد خروجی می دهد و درحالی که ورودی های آن سیگنال های Branch_Taken، Freeze و Branch_Address است که از مرحله EXE می آیند و بعدا توضیح داده خواهند شد. کد های مربوط به آن:

```
// assign pc_out = PC_middle + 4;
Incrementer #(.WORD_LENGTH(`ADDRESS_LEN)) PC_Incrementer(
    .in(pc_out), .out(PC_middle)
);
```

*Figure 5: کد مربوط به جمع کننده برای PC*

```
Register #(.WORD_LENGTH(`ADDRESS_LEN)) PC_Module(
    .clk(clk), .rst(rst), .ld(~freeze_in),
    .in(pc_in), .out(pc_out)
);
```

*Figure 6: کد مربوط به ثبات PC*

```
// PC+4 Or Branch Address
MUX_2_to_1 #(.WORD_LENGTH(`ADDRESS_LEN)) PC_Mux(
    .first(PC_middle), .second(BranchAddr_in),
    .sel_first(~Branch_taken_in), .sel_second(Branch_taken_in),
    .out(pc_in)
);
```

*Figure 7: کد مربوط به انتخاب PC یا آدرس پرش*

```
// #### Instruction Memory ####
InstructionMemory Instruction_Mem(.clk(clk), .rst(rst), .address(pc_out),
    .WriteData(instruction_write_data), .MemRead(1'b1),
    .MemWrite(1'b0), .ReadData(ReadData)
);
```

*Figure 8: کد مربوط به حافظه دستورات ها*

که این قسمت شامل ثبات های بعد از آن یعنی IF_Reg نیز می شود که در واقع خروجی همین استیج یعنی دستور اولیه Instruction و مقدار PC است (یک سیگنال Flush) نیز از این استیج عبور می کند که در قسمت بعدی توضیح داده خواهید شد که کد آن به شکل زیر می شود:

```
Register_Flush #(.WORD_LENGTH(`ADDRESS_LEN)) reg_PC_in(.clk(clk), .rst(rst), .flush(flush_in),
    .ld(~freeze_in), .in(PC_middle), .out(PC_out));

Register_Flush #(.WORD_LENGTH(`ADDRESS_LEN)) reg_Instruction(.clk(clk), .rst(rst), .flush(flush_in),
    .ld(~freeze_in), .in(ReadData), .out(Instruction_out));
```

*Figure 9: کد مربوط به قسمت IF_REG*

# توضیح قسمت ID:

این واحد وظیفه دریافت دستور از مرحله IF دیکود کردن دستور را دارد. در این مرحله سیگنال های کنترلی تولید و مقادیر رجیستر ها خوانده می شوند. لازم به ذکر است در این مرحله دستور به طور کامل کدگشایی می شود و دیگری نیاز به Op-code نیست.

در بخش اول پیاده سازی این بخش ابتدا یک Register File را پیاده سازی کردیم. در این Register File 32 Register 15 بیتی وجود دارند. این Register File دارای دو پورت خواندن ناهمگام و یک پورت نوشتن همگام با لبه پایین رونده clock می باشد. شیوه پیاده سازی Register File در شکل زیر آورده شده است.

```
module RegisterFile (
  input clk, rst,
  input[`REGISTER_LEN - 1:0] result_wb,
  input [`REG_ADDRESS_LEN - 1:0] src1, src2, dest_wb,
  input writeBackEn,
  output [`REGISTER_LEN - 1:0] reg1, reg2
);
    integer counter = 0;
    reg[`REGISTER_LEN - 1:0] data[0:`REGISTER_MEM_SIZE - 1];

    assign reg1 = data[src1];
    assign reg2 = data[src2];

    always @(negedge clk, posedge rst) begin
    if (rst) begin
      for(counter=0; counter < `REGISTER_MEM_SIZE; counter=counter+1)
        data[counter] <= 0;
    end
    else if (writeBackEn) data[dest_wb] = result_wb;
  end

endmodule
```

*Figure10: کد مربوط به قسمت Register File*

همچنین یک مالتی پلکسر در ورودی Register File وجود دارد که از بین بیت های (0-4) یا (12-15) ورودی با توجه به سیگنال mem_write یکی را به عنوان آدرس دوم به Register File می دهد. کد این بخش در زیر آمده است:

```
MUX_2_to_1 #(.WORD_LENGTH(4)) reg_file_src2_mux(
    .first(Instruction_in[15:12]), .second(Instruction_in[3:0]),
    .sel_first(mem_write), .sel_second(~mem_write),
    .out(reg_file_src2));
```

در بخش بعدی واحد کنترلر را پیاده سازی کردیم. این کنترلر با توجه به Opcode, Mode, S سیگنال های کنترلی مورد نیاز برای همه بخش ها پردازنده را تولید می کند. سیگنال های تولیدی به شرح زیر است:

الف) Execute Command (ریز دستورهای واحد حساب و منطق مطابق جدول ۵ از دستور کار ARM).
ب) سیگنال‌های مرحله حافظه شامل خواندن از حافظه mem_read و نوشتن در حافظه mem_write.
ج) سیگنال مربوط به فعال سازی مرحله باز نشانی WB_Enable.
د) سیگنال مربوط به دستورات Immediate بودن یا نبودن (Imm).
هـ) سیگنال مربوط به دستور پرش B
و) سیگنال مربوط به به روزرسانی ثبات وضعیت. این سیگنال در تمامی دستورات محسابی منطقی(mode=0) برابر ورودی S خواهد بود.

با توجه به موارد بالا، مشخص است که دستور به طور کامل دیکود می شود و در قسمت های بعدی از سیگنال های کنترلی تولید شده استفاده می شود و دستور در بخش های بعدی وارد نشده است.

در شکل زیر بخشی از پیاده سازی Control Unit را مشاهده می کنید:

```
always @(mode, opcode, s) begin

    mem_write_reg = `DISABLE;
    mem_read_reg = `DISABLE;
    wb_enable_reg = `DISABLE;
    branch_taken_reg = `DISABLE;
    ignore_hazard_reg = `DISABLE;

        case (mode)
            `ARITHMETHIC_TYPE : begin

                case(opcode)
                    `MOV : begin
                        wb_enable_reg = `ENABLE;
                        execute_command_reg = `MOV_EXE;
                        ignore_hazard_reg = `ENABLE;
                    end

                    `MOVN : begin
                        wb_enable_reg = `ENABLE;
                        execute_command_reg = `MOVN_EXE;
                        ignore_hazard_reg = `ENABLE;
                    end

                    `ADD : begin
                        wb_enable_reg = `ENABLE;
                        execute_command_reg = `ADD_EXE;
                    end

                    `ADC : begin
                        wb_enable_reg = `ENABLE;
                        execute_command_reg = `ADC_EXE;
                    end

                    `SUB : begin
                        wb_enable_reg = `ENABLE;
                        execute_command_reg = `SUB_EXE;
                    end
```

*Figure11: بخشی از کد مربوط به قسمت Control Unit*

همان طور که در پیاده سازی نیز مشخص است که با توجه به opcode و mode هرکدام از سیگنال های کنترلی مربوط به همان دستور به خصوص تولید می شود.

در بخش بعدی به پیاده سازی بخش Condition check پرداخته شده است. در این بخش ابتدا بیت های بخش cond از دستور ورودی بررسی می شود و با استفاده از Status Register برقراری شرط مورد نظر بررسی می شود. در شکل زیر بخشی از پیاده سازی مربوط به این بخش را مشاهده می کنید.

همچنین یک مالتی پلکسر با توجه به مقدار OR دو سیگنال Hazard و not خروجی condition check یا 10 داده خروجی را به رجیستر پایپ لاین منتقل می کند یا 10 بیت صفر.

```
module ConditionalCheck (
    input [`COND_LEN - 1:0] cond,
    input [3:0] statusRegister,
    output wire condState
);

wire z, c, n, v;
assign {z, c, n, v} = statusRegister;

reg tempCondition;

assign condState = tempCondition;

always @(*) begin
    case(cond)
        `COND_EQ : begin
            tempCondition <= z;
        end

        `COND_NE : begin
            tempCondition <= ~z;
        end

        `COND_CS_HS : begin
            tempCondition <= c;
        end

        `COND_CC_LO : begin
            tempCondition <= ~c;
        end

        `COND_MI : begin
            tempCondition <= n;
        end

        `COND_VS : begin
            tempCondition <= v;
        end

        `COND_VC : begin
            tempCondition <= ~v;
        end

        `COND_HI : begin
            tempCondition <= c & ~z;
        end

        `COND_LS : begin
            tempCondition <= ~c & z;
        end

        `COND_GE : begin
            tempCondition <= (n & v) | (~n & ~v);
        end
        `COND_LT : begin
            tempCondition <= (n & ~v) | (~n & v);
        end

        `COND_GT : begin
            tempCondition <= ~z & ((n & v) | (~n & ~v));
        end

        `COND_LE : begin
            tempCondition <= z & ((n & ~v) | (~n & v));
        end

        `COND_AL : begin
            tempCondition <= 1'b1;
        end
    endcase
end
endmodule
```

*Figure 12: بخشی از کد مربوط به قسمت Condition Check*

تمامی سیگنال ها و داده های خروجی از استیج ID مطابق با Figure1 وارد رجیستر های مربوط به پایپ لاین بین دو استیج ID و EXE می شوند. بخش از کد این بخش در زیر آمده است:

```
Register_Flush #(.WORD_LENGTH(`ADDRESS_LEN)) reg_PC_in(.clk(clk), .rst(rst), .flush(flush),
    .ld(~freeze), .in(PC_in), .out(PC_out));

Register_Flush #(.WORD_LENGTH(1)) reg_mem_read_en_in(.clk(clk), .rst(rst), .flush(flush),
    .ld(~freeze), .in(mem_read_en_mux), .out(mem_read_en_out));

Register_Flush #(.WORD_LENGTH(1)) reg_mem_write_en_in(.clk(clk), .rst(rst), .flush(flush),
    .ld(~freeze), .in(mem_write_en_mux), .out(mem_write_en_out));

Register_Flush #(.WORD_LENGTH(1)) reg_wb_enable_in(.clk(clk), .rst(rst), .flush(flush),
    .ld(~freeze), .in(wb_enable_mux), .out(wb_enable_out));
```

همچنین لازم به ذکر است در این بخش در صورت فعال بودن سیگنال Flush دستورات در رجیستر ورودی به مرحله بعد flush می شوند. این سیگنال توسط کنترلر تولید می شود و برای جلوگیری از اجرای ترتیب نادرست از دستورات با توجه به دستور Branch، رجیستر های خروجی استیج های IFو ID به طور کامل flush می شوند.

## توضیح قسمت EXE:

اجرای تمامی دستورات منطقی در این بخش از پردازنده اجرا می شود. به علاوه آدرس درست پرش، آدرس ذخیره یا خواندن دستورات از حافظه داده نیز در بخش انجام می شود. یکی از مهم ترین بخش ها در این قسمت واحد ALU میباشد. این واحد وظیفه اصلی محاسباتی در این بخش را بر عهده دارد. این ماژول دارای دو ورودی داده و یک خروجی داده می باشد. همچنین 4 بیت(Execute Command) تولید شده در کنترلر در بخش ID برای تعیین کردن عملیات ALU وارد آن می شوند. همچنین بیت C از status register نیز به عنوان ورودی به ALU داده می شود. در شکل زیر بخشی از پیاده سازی این ماژول را میبینید:

```
always @(*) begin
  cout = 1'b0;
  v = 1'b0;

  case(alu_command)
          `MOV_EXE:
              alu_out_temp = alu_in2;
          `MOVN_EXE:
              alu_out_temp = ~alu_in2;
    `ADD_EXE:
              begin
                  {cout, alu_out_temp} = alu_in1 + alu_in2;
                  v = ((alu_in1[`REGISTER_LEN - 1] == alu_in2[`REGISTER_LEN - 1])
                          & (alu_out_temp[`REGISTER_LEN - 1] != alu_in1[`REGISTER_LEN - 1]));
              end

    `ADC_EXE:
              begin
                  {cout, alu_out_temp} = alu_in1 + alu_in2 + cin;
                  v = ((alu_in1[`REGISTER_LEN - 1] == alu_in2[`REGISTER_LEN - 1])
                          & (alu_out_temp[`REGISTER_LEN - 1] != alu_in1[`REGISTER_LEN - 1]));
              end
    `SUB_EXE:
              begin
                  {cout, alu_out_temp} = {alu_in1[31], alu_in1} - {alu_in2[31], alu_in2};
                  v = ((alu_in1[`REGISTER_LEN - 1] == ~alu_in2[`REGISTER_LEN - 1])
                          & (alu_out_temp[`REGISTER_LEN - 1] != alu_in1[`REGISTER_LEN - 1]));
              end

    `SBC_EXE:
              begin
                  {cout, alu_out_temp} = {alu_in1[31], alu_in1} - {alu_in2[31], alu_in2} - 33'd1;
                  v = ((alu_in1[`REGISTER_LEN - 1] == ~alu_in2[`REGISTER_LEN - 1])
                          & (alu_out_temp[`REGISTER_LEN - 1] != alu_in1[`REGISTER_LEN - 1]));
              end
```

*Figure13: بخشی از کد مربوط به قسمت ALU*

علاوه بر ALU نقش ماژول Val2 Generator نیز بسیار حائز اهمیت می باشد. این ماژول وظیفه تولید ورودی دوم ALU را دارد. این ماژول با توجه به سیگنال های تولیدی در استیج ID ورودی دوم مورد نیاز ALU را تولید می کند. در این ماژول در دستورات مختلف عملکردی به شرح زیر دارد:

- در دستورات LDR و STR برابر بت 12 بیت offset می باشد، که از واحد ID وارد واحد EXE می شود
- در دستورات 32 بیت عدد فوری بر اساس براساس shift-operand داده immediate را تولید می کند.
- در دستورات شیفت فوری داده ورودی Rm را شیفت می دهد.

این ماژول 3 ورودی: Rm, shift-operand , immd, is_mem_command را دریافت میکند و با توجه به مقدار is_mem_command که از OR شدن دو سیگنال mem_w _en_in و mem_r _en_in که همان سیگنال های mem_w_en و mem_r_en وارد شده به واحد EXE از واحد ID میباشند، خروجی متناظر را با توجه به دستورات ذکر شده در بالا تولید میکند.

کد این بخش از واحد EXE را در شکل زیر مشاهده می کنید:

```
assign is_mem_command = mem_r_en_in | mem_w_en_in;
Val2Generator val2_generator(.Rm(alu_mux_src2), .shift_operand(shift_operand), .immd(immd),
    .is_mem_command(is_mem_command), .val2_out(val2)
);
```

*Figure14: کد مربوط به قسمت تولید ورودی دوم ALU*

همچنین در شکل زیر کد ماژول Val2 Generator را مشاهده می کنید:

```
assign val2_out = (is_mem_command == `DISABLE) ?
    (
       (immd == 1'b1) ? {24'b0, shift_operand[7:0]} // need for loop here.
       // (immd == 1'b1) ?
       // (
       //     (shift_operand[11:8] == 4'b0) ? {24'b0, shift_operand[7:0]} : imd_shifted
       // )
       : (shift_operand[6:5] == `LSL_SHIFT_STATE) ? Rm << {1'b0, shift_operand[11:7]}
       : (shift_operand[6:5] == `LSR_SHIFT_STATE) ? Rm >> {1'b0, shift_operand[11:7]}
       : (shift_operand[6:5] == `ASR_SHIFT_STATE) ? Rm >>> {1'b0, shift_operand[11:7]}
       : ((shift_operand[6:5] == `ROR_SHIFT_STATE) && (shift_operand[11:8] == 4'b0)) ? Rm
       : rm_rotate
    )
    : {20'b0, shift_operand};


always @(*) begin
    rm_rotate = Rm;
    for (i = 0; i < {1'b0, shift_operand[11:7]}; i = i + 1) begin
        rm_rotate = {rm_rotate[0], rm_rotate[31:1]};
    end
end


always @(*) begin
    imd_shifted = {24'b0, shift_operand[7:0]};
    for (j = 0; j < {1'b0, shift_operand[11:8]}; j = j + 1) begin
        imd_shifted = {imd_shifted[1], imd_shifted[0], imd_shifted[31:2]};
    end
end
```

*Figure15: کد مربوط Val2 Generator*

همچنین در این قسمت یک Adder پیاده سازی شده است که وظیفه آن جمع مقدار PC با مقدار signed_immd_24 برای تولید آدرس پرش در دستورات Branch می باشد.

کد مربوط به این بخش را در شکل زیر مشاهده می کنید:

```
// branch_address = PC_in + signed_immd_24;
Adder #(.WORD_LENGTH(`ADDRESS_LEN)) adder(.in1(PC_in), .in2({signed_immd_24[23],
    signed_immd_24[23], signed_immd_24[23], signed_immd_24[23],
    signed_immd_24[23], signed_immd_24[23],signed_immd_24, 2'b0}),
    .out(branch_address_out)
);
```

*Figure16: کد مربوط به Adder در بخش EXE*

علاوه بر موارد ذکر شده دو مالتی پلکسر برای بخش forwarding نیز پیاده شده است که توضیحات مربوط به آن ها در قسمت forwarding موجود می باشد.

همچنین همانند قسمت های گذشته تمامی خروجی های این قسمت مطابق با Figure 1 در رجیسترهای بین دو استیج دخیره می شوند و با لبه بالا روند کلاک وارد استیج بعدی می شوند.

## توضیح قسمت MEM:

در این قسمت حافظه داده ها در پردازنده پیاده سازی شده است. این حافظه با دریافت سیگنال های mem_r_en_in و mem_w_en_in از حافظه داده می خواند یا می نویسد. این حافظه از آدرس 1024 شروع شده و در هر مرحله 32 بیت داده یا 4 بایت داده خوانده یا نوشته می شود. همچنین در این حافظه امکان دسترسی به تک بایت وجود ندارد. حجم این حافظه 256 بایت و امکان خواندن و نوشتن تنها در آدرس ها با مضارب 4 وجود دارد. در این حافظه 4 ورودی دارد:

alu_res_in: که برای تعیین آدرس خواندن و یا نوشتن می باشد.
Val_Rm: این ورودی مقداری است که باید در حافظه نوشته شود.
mem_r_en_in: سیگنال برای خواندن از حافظه.
mem_w_en_in: سیگنال برای نوشتن در حافظه.
در شکل زیر بخش از کد پیاده سازی حافظه را مشاهده می کنید:

```
module Memory(clk, rst, address, WriteData, MemRead, MemWrite, ReadData);
  input[`INSTRUCTION_LEN - 1:0] WriteData;
  input[`ADDRESS_LEN - 1:0] address;
  input clk, rst, MemRead, MemWrite;
  output wire[`INSTRUCTION_LEN - 1:0] ReadData;

  reg[`DATA_MEMORY_LEN - 1:0] data[0:`DATA_MEMORY_SIZE - 1];

  wire [`ADDRESS_LEN - 1:0] address4k = {address[`ADDRESS_LEN - 1:2], 2'b0} - `ADDRESS_LEN'd1024; //address4k
  wire [`ADDRESS_LEN - 1:0] address4k_p1 = {address4k[`ADDRESS_LEN - 1:2], 2'b01}; // address4k + 1
  wire [`ADDRESS_LEN - 1:0] address4k_p2 = {address4k[`ADDRESS_LEN - 1:2], 2'b10}; // address4k + 2
  wire [`ADDRESS_LEN - 1:0] address4k_p3 = {address4k[`ADDRESS_LEN - 1:2], 2'b11}; // address4k + 3

  assign ReadData = (MemRead == `ENABLE) ?
      {data[address4k], data[address4k_p1], data[address4k_p2], data[address4k_p3]}
      : `INSTRUCTION_LEN'b0;

  integer counter = 0;
  always @(posedge clk, posedge rst) begin
    if (rst) begin
      for(counter=0; counter < `DATA_MEMORY_SIZE; counter=counter+1)
        data[counter] <= 0;
    end
    else if (MemWrite) begin
      data[address4k_p3] = WriteData[`INSTRUCTION_MEM_LEN - 1:0];
      // write to data[address + 1]
      data[address4k_p2] = WriteData[`INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN - 1 : `INSTRUCTION_MEM_LEN];

      // write to data[address + 2]
      data[address4k_p1] = WriteData[`INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN - 1 : `INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN];

      // write to data[address + 3]
      data[address4k] = WriteData[`INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN - 1 : `INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN + `INSTRUCTION_MEM_LEN];
    end
```

*Figure17: کد مربوط به Memory*

همان طور که در شکل 17 مشخص است، پیاده سازی به شکلی می باشد که امکان دسترسی به تک بایت در حافظه وجود ندارد و در هر مرحله 4 بایت در حافظه نوشته یا از آن خوانده می شود.

همچنین خروجی حافظه ، des و مقدار alu_Res وارد رجیستر پایپ لاین در بین دو استیج Memory و WB میشود.

## توضیح قسمت WB:

در این بخش یک مالتی پلکسر پیاده سازی شده است که با توجه به مقدار mem_read_enable انتخاب می کند از بین مقادیر alu_result و data_memory کدام یک به استیج ID منتقل شود. پیاده سازی این بخش در شکل زیر مشاهده می کنید:

```
module WB_Stage(
  input clk,
  input rst,
  input mem_read_enable,
  input wb_enable_in,

  input [`REGISTER_LEN - 1:0] alu_result,
  input [`REGISTER_LEN - 1:0] data_memory,
  input [`REG_ADDRESS_LEN - 1:0] wb_dest_in,

  output wire wb_enable_out,

  output wire [`REG_ADDRESS_LEN - 1:0] wb_dest_out,
  output wire [`REGISTER_LEN - 1:0] wb_value
);

  MUX_2_to_1 #(.WORD_LENGTH(`REGISTER_LEN)) wb_stage_mux(
    .first(data_memory), .second(alu_result),
    .sel_first(mem_read_enable), .sel_second(~mem_read_enable),
    .out(wb_value));

  assign wb_dest_out = wb_dest_in;
  assign wb_enable_out = wb_enable_in;

endmodule
```

*Figure 18: کد مربوط به WB*

# توضیح ماژول Status Register:

**این ماژول مسئول مگه داری Status های ما است. به دلیل اینکه تمام دستورات نمی توانند Status را تغییر دهند یک سیگنال لود برای آن قرار داده شده است که هر وقت S و یا همان STATUS در دستور فعال بود بتوان مقادیر**

Status جدید که از مرحله EXE به آن می رسد را تغییر دهد. تمامی حالات Status در توضیح آزمایش و دستورات گفته شده است. در این آزمایش ما فقط از Status 4 استفاده می کنیم. این موارد در دستورات شرطی که انجام شدن یا نشدن انها با توجه به مقدار یکی از این Status ها است نیاز است و تغییر آن فقط در زمان هایی که دستور های خاصی که S ان ها 1 است انجام می پذیرد. بقیه مدار مانند یک ثبات عادی کار می کند که در صورت وجود لود مقدار ورودی را در خروجی می برد.

کر مربوط به این قسمت در شکل زیر آمده است که چون کد ساده ای است توضیحات بالا برای آن امده است.

```
1  `include "Defines.v"
2
3  module Status_Register (
4      input clk, rst,
5      input ld,
6      input [3:0] data_in,
7
8      output reg [3:0] data_out
9  );
10
11
12   always@(negedge clk, posedge rst)
13   begin
14     if (rst) data_out <= 0;
15     else if (ld) data_out <= data_in;
16   end
17
18 endmodule
```

# توضیح ماژول Hazard:

ما در پردازنده ها در کل 3 نوع مخاطره (Hazard) داریم.

**الف) مخاطره ساختاری:** این مخاطره در بطن ساختار خط لوله وجود دارد به همین دلیل ساختاری نا گرفته است. این مخاطره بین مرحله WB و ID به دلیل همزمانی خواندن و نوشتن از ثبات های عمومی ناشی می شود. برای رفع این مخاطره نوشتن در ثبات‌های عمومی را به لبه پایین رونده منتقل کردیم. بنابراین این مخاطره در پردازنده رفع شده است.

**ب) مخاطره کنترلی:** این مخاطره ناشی از دستورات پرش است. در صورتی که دستور پرش وارد خط لوله شود به دلیل تأخیر در تشخیص و محاسبه آدرس پرش دو دستور به اشتباه وارد خط لوله می‌شود. برای رفع این مشکل سیگنال های Flush به پردازنده افزوده شد. بنابراین این محاطره نیز رفع شده است.

**ج) مخاطره داده ای:** مخاطره داده‌ای به صورت زیر دسته‌بندی می گردد:

۱- **خواندن پس از نوشتن (RAW):** مخاطره خواندن بعد از نوشتن در حالتی رخ می‌دهد که یک دستور، رجیستری که هنوز محاسبه یا ذخیره نشده است را فراخوانی می‌نماید. در این حالت می‌بایست دستورات جدید تا محاسبه یا ذخیره شدن آن رجیستر متوقف گردند. در مثال زیر دستور ۲ نیاز به رجیستر R2 دارد که دستور ۱ آنرا محاسبه می نماید، بنابراین تا ذخیره یا محاسبه شدن مقدار R2 دستور ۲ باید متوقف گردد.

1. SUB R2,R0,R1
2. AND R3,R2,R1

۲- **نوشتن پس از خواندن (WAR):** در مثال زیر رجیستر R1 در حالاتی ممکن است قبل از خوانده شدن توسط دستور ۱ مقدار آن به وسیله دستور ۲ تغییر کند. به این رخداد هازارد داده‌ای از نوع WAR گفته می‌شود.

1. SUB R2,R0,R1
2. AND R1,R3,R4

۳- **نوشتن پس از نوشتن(WAW):** هازارد داده‌ای نوشتن پس از نوشتن نیز همچون هازرد WAR در پردازنده‌های In-Order رخ نمی‌دهد. این نوع مخاطره ممکن است در پردزنده های Out-of-order رخ دهد و ترتیب نوشتن در رجیستر مقصد تغییر کند.

برای رفع هازارد RAW ماژول Hazard Detection Unit همانند شکل زیر به پردازنده ARM اضافه نمایید. در این واحد، منابع Src1 و Src2 در مرحله ID با مقصدهای مراحل EXE و MEM به صورت مجزا مقایسه می‌شود و در صورت برابر بودن یکی از منابع با مقصدها، سیگنال کنترلی Hazard_Detected_Signal را برابر ۱ قرار می‌دهد. این سیگنال باید دستورات درون IF و رجیسترهای پس از آن را متوقف نماید و حبابی را به خط لوله تزریق نماید. برای ایجاد حباب کافی است تمامی سیگنال های حیاتی پردازنده صفر گردد. برای پیاده سازی این مرحله به ترتیب زیر عمل کنید:

**حالت هایی که هازارد RAW رخ می دهد به شرح زیر است:**

- برابری src1 با مقصد EXE در صورت یک بودن WB_EN در مرحله اجرا
- برابری src1 با مقصد MEM در صورت یک بودن WB_EN در مرحله حافظه
- برابری src2 با مقصد EXE در صورت یک بودن WB_EN در مرحله اجرا و دو منبعی بودن دستور
- برابری src2 با مقصد MEM در صورت یک بودن WB_EN در مرحله حافظه و دو منبعی بودن دستور

**برای ایجاد این ماژول همانند بالا دستورات را کنترل می کنیم.**

**ورودی های این ماژول ثبات مقصد در استیج EXE، سیگنال نوشتن در استیج EXE، ثبات مقصد ار استیج MEM و سیگنال نوشتن در استیج MEM است و همین طور منبع های ثبات های ما که همان scr1 و src2 هستند.**

همان طور که گفته شده ما می خواهیم RAW را بر طرف می کنیم، می دانیم این hazard زمانی رخ می دهد که دستور ما Store باشد و یا بیت Immediate دستور ما 1 باشد (شامل دستوراتی که تنها دارای یک منبع هستند). برای زمان store بیت نوشتن در مموری یا MEM_W و برای زمان Immediate بیت I 0 است که نات آن را استفاده می کنیم. با OR کردن این 2 بیت می فهمیم دستور ما احتمال hazard دارد و آن را با حالت های بالا چک می کنیم. کد مربوط به آن:

```
assign internal_hazard_with_forwarding = ((src1_address == exe_wb_dest) && (exe_wb_en == 1'b1)) ? 1'b1
        : ((src2_address == exe_wb_dest) && (exe_wb_en == 1'b1) && (have_two_src == 1'b1)) ? 1'b1
        : 1'b0;

assign hazard_detected = (ignore_hazard == 1'b1) ? 1'b0
    : (with_forwarding == 1'b1) ? internal_hazard_with_forwarding & EXE_mem_read_en
```

است که تمامی حالت های hazard را با هم چک می کند.

در این حالت، زمانی که هر کدام از اتفاقات گفته شده رخ دهد این واحد می تواند پایپ کلی پردازنده را متوقف کند. این کار توسط خروجی همین واحد که در واقع همان سیگنال Freeze که در استیج IF و ID گفته شد است. در صورت یک بودن این سیگنال ثبات ها تغییر نمی کنند و کنترلر تمامی سیگنال های کنترلی را 0 خروجی می دهد.

پردازنده می تواند در حالت Hazard که همین حالت است و یا حالت Forwarding که در قسمت بعد توضیح داده می شود کار کند. در حالت Hazard سیستم کند تر کار می کند چرا که در صورت وقوع RAW، پایپ ها می ایستند و منتظر می مانند تا محاسبات قبلی انجام شوند ولی در حالت Forwarding این اتفاق نمی افتد که در حالت بعد توضیح داده می شود.

# توضیح ماژول Forwarding:

در این قسمت می خواهیم همان hazard هایی که در قسمت قبل با Hazard Detection بر طرق کردیم با واحد Forwarding انجام دهیم. فرق این قسمت با قبلی آن است که در حالت قبل ما در صورت بروز Hazard ما تمام پایپ را متوقف می کنیم تا محاسبات انجام شوند و منبع های مورد نیاز نوشته شوند ولی در این قسمت می خواهیم بدون توقف پایپ ها منبع مورد نیاز را از استیج بعدی به استیج های قبلی انتقال دهیم و بتوانیم سریع تر به ان ها دسترسی پیدا کنیم.

برای این قسمت ما از داده های رو به جلو برای دو قسمت می توانیم استفاده کنیم، یکی زمانی که store اتفاق می افتد ما به جای توقف پایپ با استفاده از Hazard Detection، می توانیم از نتیجه واحد ALU در استیج ممووری و ثبات مقصد در استیج باز نشانی استفاده کنیم به این صورت که یک مولتی پلکسر به استیج EXE اضافه می کنیم و سیگنال انتخاب آن را با استفاده از همین واحد درست می کنیم. این مولتی پلکسر آدرس حافظه در استیج MEM را می سازد.

برای قسمت بعد زمانی که حالت Immediate داریم و RAW رخ داده است می توانیم به جای توقف پایپ، یک مولتی پلکسر دیگر به استیج EXE اضافه می کنیم تا بتوانیم داده ی آن را از استیج بعد به آن منتقل کنیم. در این مولتی پلکسر داده یا از نتیجه واحد ALU در استیج ممووری یا ثبات مقصد در استیج باز نشانی و یا در صورت عادی و همان منبع 2 دستور است. که انتخاب آن نیز از همین واحد ساخته می شود. این مولتی پلکسر منبع دوم واحد ALU را می سازد و در صورت برابر بودن با ثبات مقصد در استیج Write Back باید ثبات مقصد Write Back انتخاب شود

در این قسمت در صورتی که در استیج ممووری سیگنال نوشتن فعال باشد (MEM_WB_EN) و ثبات مقصد در استیج ممووری با ثبات منبع 1 که همان منبع 1 خروجی از فایل ثبات ها در استیج EXE است برابر باشد باید خروجی واحد ALU را که از استیج MEM آمده است بر داریم. در این جالت سیگنال انتخاب مولتی پلکسر اول را انتخاب می کنیم.

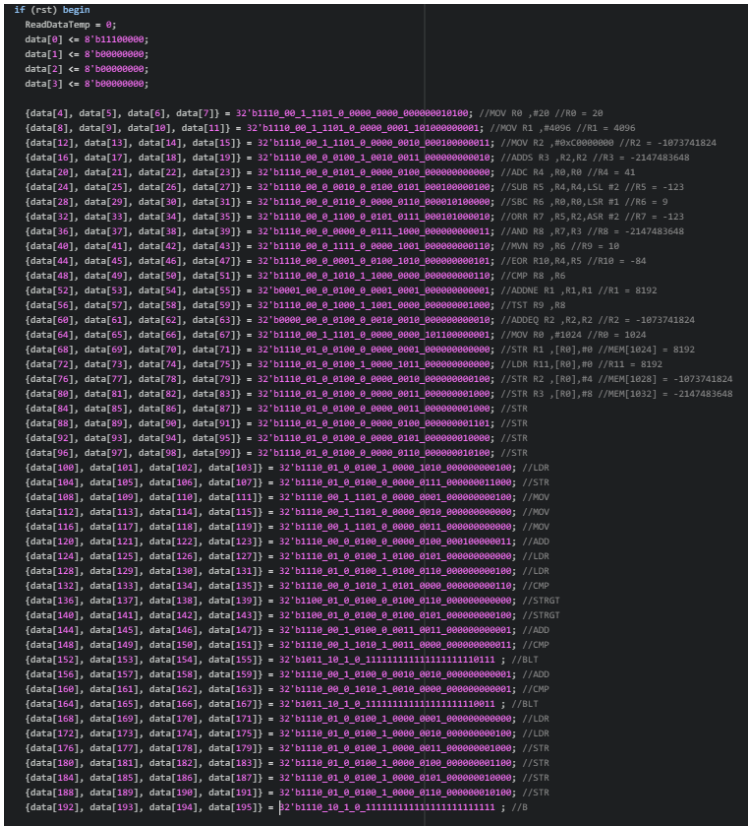
برای مولتی پلکسر دوم، همانند قبل است و زمانی که شروط بالا برابر با منبع 2 از ثبات فایل در استیج EXE به جای منبع 1 بود باید داده واحد ALU از ممووری برداشته شود و در صورت برابر بودن با ثبات مقصد در استیج Write Back باید ثبات مقصد Write Back انتخاب شود. کد مربوط به این قسمت:

```
assign sel_src1 = (en_forwarding == 1'b1) ?
    (
        (MEM_wb_en && (MEM_dst == ID_src1)) ? `FORW_SEL_FROM_MEM
        : (WB_wb_en && (WB_dst == ID_src1)) ? `FORW_SEL_FROM_WB
        : 2'b0
    )
    : 2'b0;

assign sel_src2 = (en_forwarding == 1'b1) ?
    (
        (MEM_wb_en && (MEM_dst == ID_src2)) ? `FORW_SEL_FROM_MEM
        : (WB_wb_en && (WB_dst == ID_src2)) ? `FORW_SEL_FROM_WB
        : 2'b0
    )
    : 2'b0;
```

# نتایج آزمایش:

در این قسمت می خواهیم به نتایج این آزمایش بپردازیم.

برای این کار یک تست بنچ می نویسیم و ورودی های آن ریست مدار و سیگنال داشتن forwarding و یا hazard است. دستورات تست را درون حافظه دستورات در مرحله واکشی می نویسیم. دستورات مانند زیر می شوند:



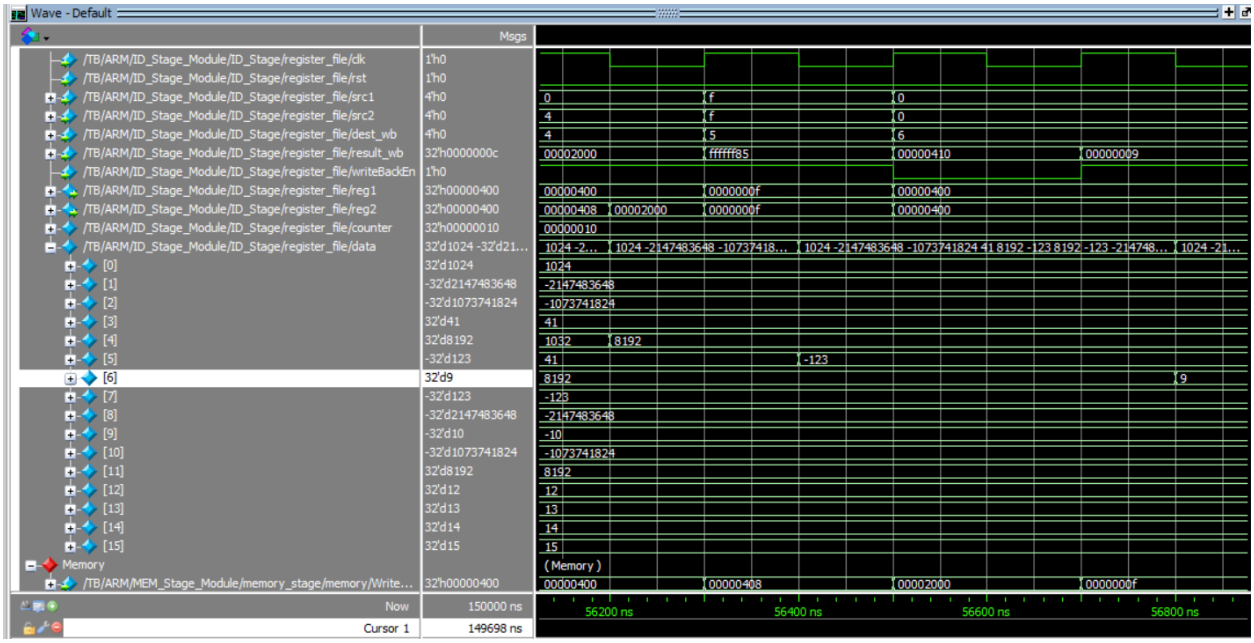
```
if (rst) begin
  ReadDataTemp = 0;
  data[0] <= 8'b11100000;
  data[1] <= 8'b00000000;
  data[2] <= 8'b00000000;
  data[3] <= 8'b00000000;

  {data[4], data[5], data[6], data[7]} = 32'b1110_00_1_1101_0_0000_0000_000000010100; //MOV R0 ,#20 //R0 = 20
  {data[8], data[9], data[10], data[11]} = 32'b1110_00_1_1101_0_0000_0001_10100000001; //MOV R1 ,#4096 //R1 = 4096
  {data[12], data[13], data[14], data[15]} = 32'b1110_00_1_1101_0_0000_0010_000100000011; //MOV R2 ,#0xC0000000 //R2 = -1073741824
  {data[16], data[17], data[18], data[19]} = 32'b1110_00_0_0100_1_0010_0011_000000000010; //ADDS R3 ,R2,R2 //R3 = -2147483648
  {data[20], data[21], data[22], data[23]} = 32'b1110_00_0_0101_0_0000_0100_000000000000; //ADC R4 ,R0,R0 //R4 = 41
  {data[24], data[25], data[26], data[27]} = 32'b1110_00_0_0010_0_0100_0101_000100000100; //SUB R5 ,R4,R4,LSL #2 //R5 = -123
  {data[28], data[29], data[30], data[31]} = 32'b1110_00_0_0110_0_0000_0110_000010100000; //SBC R6 ,R0,R0,LSR #1 //R6 = 9
  {data[32], data[33], data[34], data[35]} = 32'b1110_00_0_1100_0_0101_0111_000101000010; //ORR R7 ,R5,R2,ASR #2 //R7 = -123
  {data[36], data[37], data[38], data[39]} = 32'b1110_00_0_0000_0_0111_1000_000000000011; //AND R8 ,R7,R3 //R8 = -2147483648
  {data[40], data[41], data[42], data[43]} = 32'b1110_00_0_1111_0_0000_1001_000000000110; //MVN R9 ,R6 //R9 = 10
  {data[44], data[45], data[46], data[47]} = 32'b1110_00_0_0001_0_0100_1010_000000000101; //EOR R10,R4,R5 //R10 = -84
  {data[48], data[49], data[50], data[51]} = 32'b1110_00_0_1010_1_1000_0000_000000000110; //CMP R8 ,R6
  {data[52], data[53], data[54], data[55]} = 32'b0001_00_0_0100_0_0001_0001_000000000001; //ADDNE R1 ,R1,R1 //R1 = 8192
  {data[56], data[57], data[58], data[59]} = 32'b1110_00_0_1000_1_1001_0000_000000001000; //TST R9 ,R8
  {data[60], data[61], data[62], data[63]} = 32'b0000_00_0_0100_0_0010_0010_000000000010; //ADDEQ R2 ,R2,R2 //R2 = -1073741824
  {data[64], data[65], data[66], data[67]} = 32'b1110_00_1_1101_0_0000_0000_101100000001; //MOV R0 ,#1024 //R0 = 1024
  {data[68], data[69], data[70], data[71]} = 32'b1110_01_0_0100_0_0000_0001_000000000000; //STR R1 ,[R0],#0 //MEM[1024] = 8192
  {data[72], data[73], data[74], data[75]} = 32'b1110_01_0_0100_1_0000_1011_000000000000; //LDR R11,[R0],#0 //R11 = 8192
  {data[76], data[77], data[78], data[79]} = 32'b1110_01_0_0100_0_0000_0010_000000000100; //STR R2 ,[R0],#4 //MEM[1028] = -1073741824
  {data[80], data[81], data[82], data[83]} = 32'b1110_01_0_0100_0_0000_0011_000000001000; //STR R3 ,[R0],#8 //MEM[1032] = -2147483648
  {data[84], data[85], data[86], data[87]} = 32'b1110_01_0_0100_0_0000_0011_000000001000; //STR
  {data[88], data[89], data[90], data[91]} = 32'b1110_01_0_0100_0_0000_0100_000000001101; //STR
  {data[92], data[93], data[94], data[95]} = 32'b1110_01_0_0100_0_0000_0101_000000010000; //STR
  {data[96], data[97], data[98], data[99]} = 32'b1110_01_0_0100_0_0000_0110_000000010100; //STR
  {data[100], data[101], data[102], data[103]} = 32'b1110_01_0_0100_1_0000_1010_000000000100; //LDR
  {data[104], data[105], data[106], data[107]} = 32'b1110_01_0_0100_0_0000_0111_000000011000; //STR
  {data[108], data[109], data[110], data[111]} = 32'b1110_00_1_1101_0_0000_0001_000000000100; //MOV
  {data[112], data[113], data[114], data[115]} = 32'b1110_00_1_1101_0_0000_0010_000000000000; //MOV
  {data[116], data[117], data[118], data[119]} = 32'b1110_00_1_1101_0_0000_0011_000000000000; //MOV
  {data[120], data[121], data[122], data[123]} = 32'b1110_00_0_0100_0_0000_0100_000100000011; //ADD
  {data[124], data[125], data[126], data[127]} = 32'b1110_01_0_0100_1_0100_0101_000000000000; //LDR
  {data[128], data[129], data[130], data[131]} = 32'b1110_01_0_0100_1_0100_0110_000000000100; //LDR
  {data[132], data[133], data[134], data[135]} = 32'b1110_00_0_1010_1_0101_0000_000000000110; //CMP
  {data[136], data[137], data[138], data[139]} = 32'b1100_01_0_0100_0_0100_0110_000000000000; //STRGT
  {data[140], data[141], data[142], data[143]} = 32'b1100_01_0_0100_0_0100_0101_000000000100; //STRGT
  {data[144], data[145], data[146], data[147]} = 32'b1110_00_1_0100_0_0011_0011_000000000001; //ADD
  {data[148], data[149], data[150], data[151]} = 32'b1110_00_1_1010_1_0011_0000_000000000011; //CMP
  {data[152], data[153], data[154], data[155]} = 32'b1011_10_1_0_11111111111111111110111 ; //BLT
  {data[156], data[157], data[158], data[159]} = 32'b1110_00_1_0100_0_0010_0010_000000000001; //ADD
  {data[160], data[161], data[162], data[163]} = 32'b1110_00_0_1010_1_0010_0000_000000000001; //CMP
  {data[164], data[165], data[166], data[167]} = 32'b1011_10_1_0_11111111111111110011 ; //BLT
  {data[168], data[169], data[170], data[171]} = 32'b1110_01_0_0100_1_0000_0001_000000000000; //LDR
  {data[172], data[173], data[174], data[175]} = 32'b1110_01_0_0100_1_0000_0010_000000000100; //LDR
  {data[176], data[177], data[178], data[179]} = 32'b1110_01_0_0100_1_0000_0011_000000001000; //STR
  {data[180], data[181], data[182], data[183]} = 32'b1110_01_0_0100_1_0000_0100_000000001100; //STR
  {data[184], data[185], data[186], data[187]} = 32'b1110_01_0_0100_1_0000_0101_000000010000; //STR
  {data[188], data[189], data[190], data[191]} = 32'b1110_01_0_0100_1_0000_0110_000000010100; //STR
  {data[192], data[193], data[194], data[195]} = 32'b1110_10_1_0_11111111111111111111111 ; //B
```

**حال این مدار را یک بار در Modelsim و یک بار در Quartus اجرا می کنیم و نتایج را مشاهده می کنیم. برای مشاهده نتایج در Modelsim نتیجه آخری در ثبات ها و حافظه را نشان می دهیم و تعداد clock هایی که طول کشیده تا دستورات کامل شوند را برای هر دو حالت نمایش می دهیم. و در Quartus، مدار RTL تولید شده به همراه تعداد المان های منطقی و resource استفاده شده توسط مدار را نشان می دهیم.**

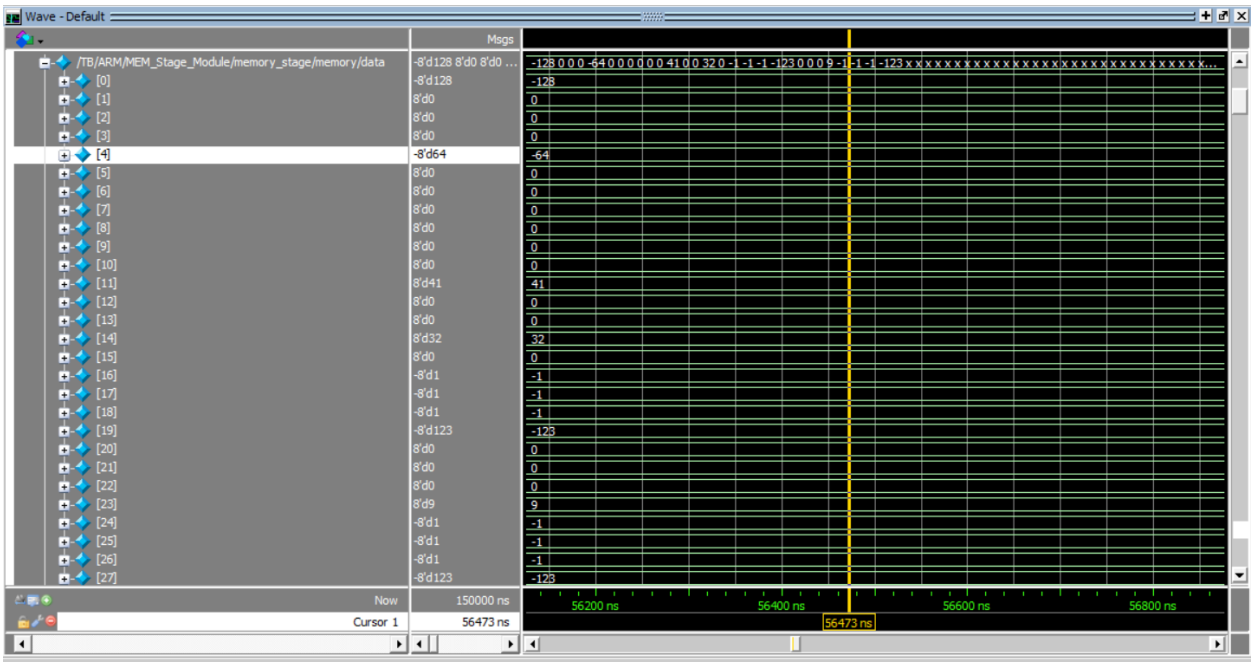
- **نتایج شبیه سازی آزمایش در نرم افزار Modelsim: همان طور که میدانیم باید جواب هر دو حالت با هم یکی باشد که در عکس ها هم آمده و درست است.**

**در حالت داشتن Hazard Detection Unit:**

**مقادیر ثبات ها در انتها و تغییرات آخر انها به شکل:**




**مقادیر نهایی در حافظه:**

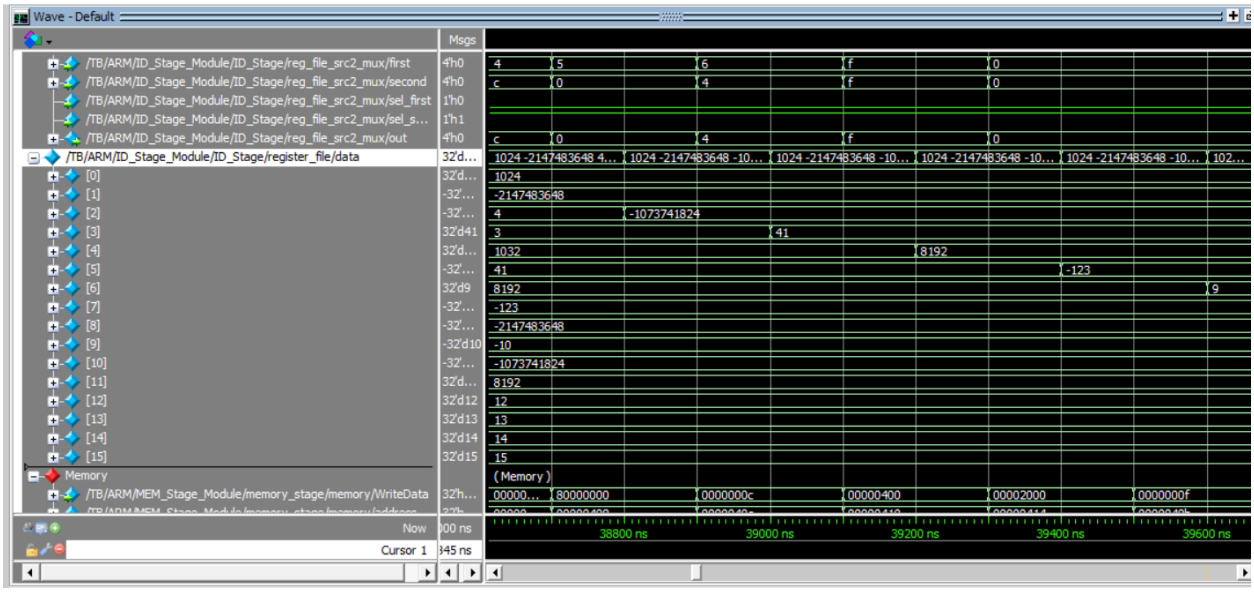



**همان طور که در شکل هم دیده می شود مدار بعد از 56800 ns تمام دستورات را به اتمام می رساند.**

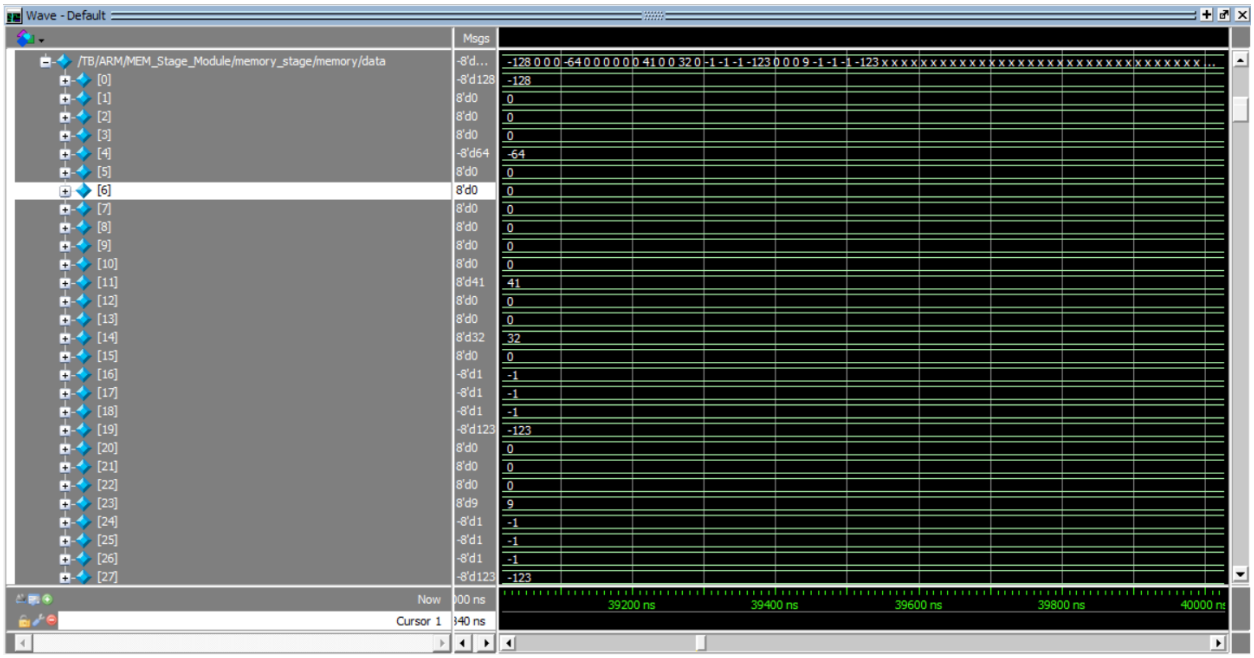
**زمان کلاک ما در این مدار 100*2 بوده است که در نتیجه: $\frac{56800}{2*100} = 284\ Clock$ زمان می برد تا کار مدار تمام شود. این زمان رای حالت Hazard Detection است.**

**در حالت داشتن Forwarding Unit:**

**مقادیر ثبات ها در انتها و تغییرات آخر انها به شکل:**




**و مقادیر ذخیره شده در حافظه به شکل:**

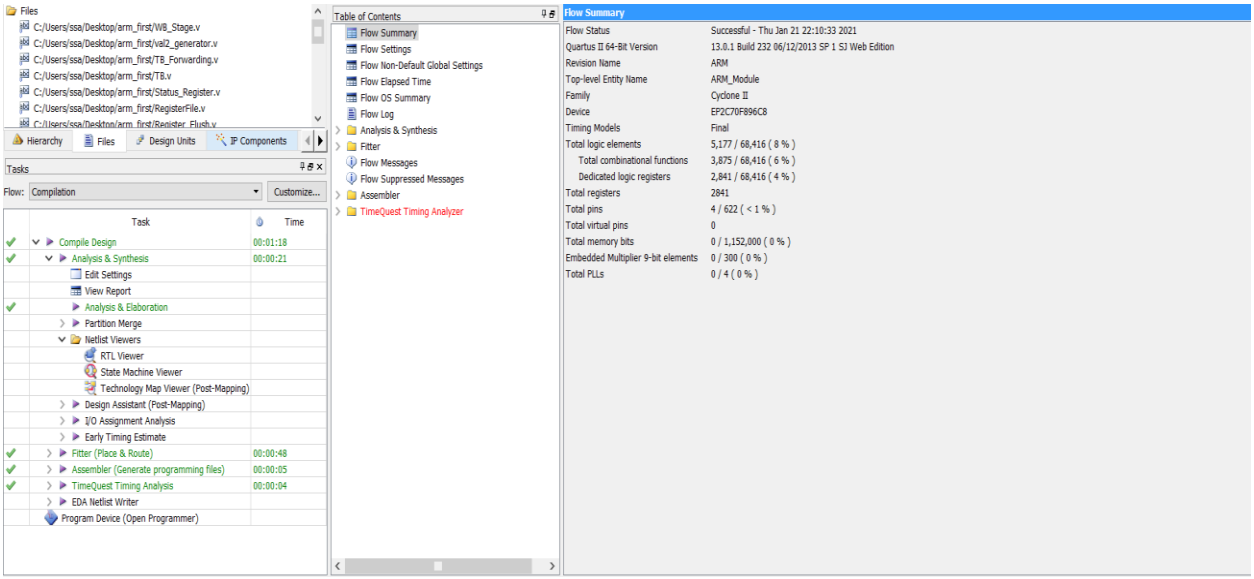



**همان طور که در شکل هم دیده می شود مدار بعد از 39600 ns تمام دستورات را به اتمام می رساند. زمان کلاک ما در این مدار 100*2 بوده است که در نتیجه: $\frac{39600}{2*100} = 198\ Clock$ زمان می برد تا کار مدار تمام شود. این زمان رای حالت forwarding است.**

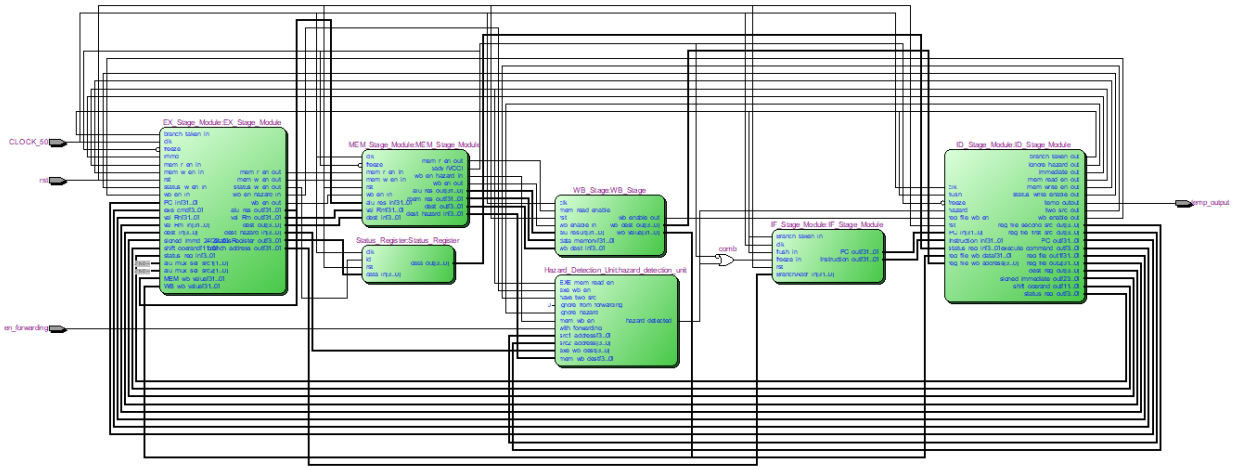
- **حال همین مدار ها را در Quartus شبیه سازی می کنیم و داریم:**

**در حالت اول برای مدار با *Hazard Detection* داریم:**

**نتایج بعد از اجرای کد درون نرم افزار:**



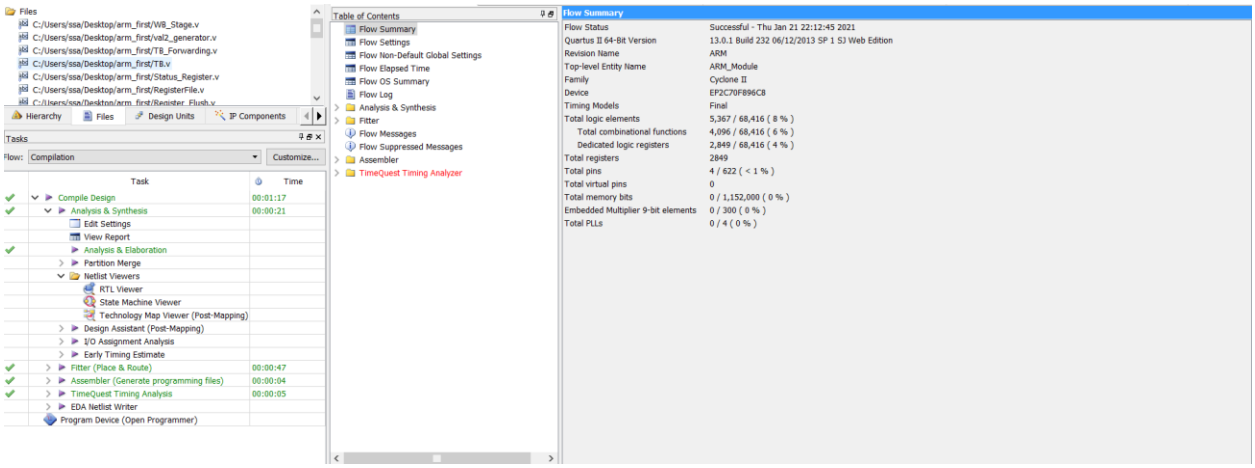

**مدار دیده شده توسط نرم افزار:**



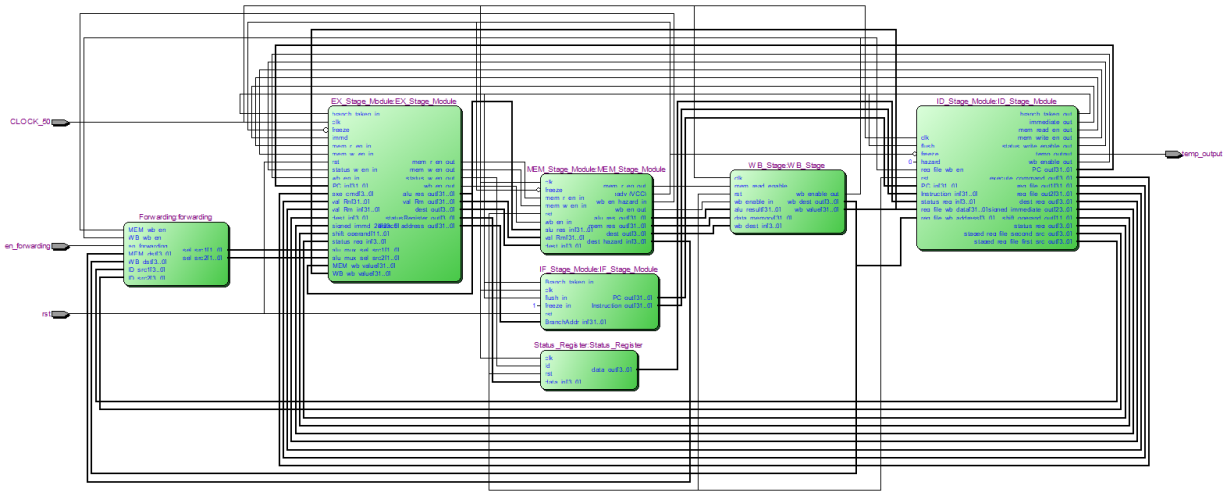

**به علت اینکه خروجی برای مدار نهایی از مرحله *ID* گذاشتیم مدار *ID* جلو تر کشیده شده است. ولی همان طور که می دانستیم اول *IF* و *ID* و بقیه مدار بود که درست است. علاوه بر آن همان طور که می دانستیم *Hazard Unit* خروجی ای به *IF* و *ID* می دهد که درست است. بقیه مدار و سیگنال ها نیز با مقایسه با حالت اولیه می بینیم که همگی همان و درست است.**

**و در حالت فورواردینگ داریم:**

**نتایج بعد از اجرای کد درون نرم افزار:**




**مدار دیده شده توسط نرم افزار:**




**به علت اینکه خروجی برای مدار نهایی از مرحله *ID* گذاشتیم مدار *ID* جلو تر کشیده شده است. ولی همان طور که می دانستیم اول *IF* و *ID* و بقیه مدار بود که درست است. علاوه بر آن همان طور که می دانستیم *Forwarding* خروجی های به *EXE* می دهد که در واقع انتخاب مولتی پلکسر ها است که درست است. ورودی های این واحد نیز از قسمت های *MEM, EXE* آمده است که درست است. بقیه مدار و سیگنال ها نیز با مقایسه با حالت اولیه می بینیم که همگی همان و درست است.**

**برای مقایسه دو حالت جدول زیر را داریم:**

| | ***Total Clock*** | ***Total Logic Element*** | ***Total Register*** |
|---|---|---|---|
| | | | |

| | | | |
|---|---|---|---|
| *Hazard Detection Unit* | *284* | *5177 (8%)* | ***2841*** |
| *Forwarding Unit* | *194* | *5367 (8%)* | ***2649*** |

$$Speed\ Up = \frac{284}{194} = 1.46$$

**همان طور که انتظار داشتیم در حالت دو به جلو سرعت مدار تقریبا 1.5 برابر حالت هازارد بود. ولی این سرعت باعث گرفتن مقدار بیشتری از المان های منطقی نیز بوده است چرا که همان طور که دیده می شود مقدار المان های منطقی در حالت رو به جلو بیشتر است که نشان از بیشتر گرفتن ریسورس کل مدار می شود. در اینجا ما مقداری ریسورس بیشتر رو فدای سرعت بالاتر کرده ایم که می زان تسریع نشان می دهد این *trade-off* به خوبی انجام شده است چرا که سرعت مناسبی دریافت کرده ایم.**

# مشکلات پیش آمده:

**برخی از مشکلات پیش آمده به شرح زیر است:**

1. **زمانی که می خواستیم قسمت *ID* را به قسمت قبل وصل کنیم و تست کنیم با مشکلات زیادی رو به رو شدیم. این قسمت برای گروه ما بیشترین ایراد را داشت که بیشترین قسمت آن قسمت *Controller* آن بوده است. چرا که در حالت اول برخی از دستورات را کاملا آشنا نبودیم و کمی طول کشید تا سیگنال های کنترلی همه را بدون ایراد درست کنیم.**
2. **مشکل دیگر زمانی بود که می خواستیم مدار نهایی را تست کنیم. این مدار که در هر مرحله تست شده بود زمانی که با کل دستورات تست شد غلط های زیادی از جمله رجیستر نشده بودن 1 سیگنال کنترلی، تغییر نکردن درست آدرس و ... بود. بیشترین زمان در انجام پروژه برای این قسمت بود چرا که باید مرحله به مرحله کا کد را چک می کردیم و پیدا می کردیم هر کدام از دستورات در کدام مرحله دچار مشکل می شوند. همین طور برخی از سیگنال ها در مرحله آخر مقدار *X* می گرفتند که به معنی وصل نبودن بودند. زمان زیادی در این قسمت صرف شد تا مدار بتواند برای بار اول کار کند.**
3. **مشکل دیگر زمانی بود که می خواستیم بعد از نوشتن مدار رو به جلو آن را با کل مدار تست کنیم. در ابتدا فکر کردیم شاید مشکل از قسمت قبل هم بوده باشد در حالی که با تست کردن دوباره مرحله به مرحله برای پیدا کردن مشکل فهمیدیم که مولتی پلکسر هایی که در این مرحله به مدار اضافه شده بودند مشکل داشتند. زمانی نسبتا خوبی برای درست کردن این مشکل گرفت که در آخر موفق به درست کردن آن شدیم.**
4. **یکی از مشکلات جالب زمان *Compile* با *Quartus* داشتیم. در این زمان تمام *resource* های ما 0 گزارش می شدند و نمی دانستیم دلیل آن چیست. بعد از تحقیق متوجه شدیم به این دلیل است که نرم افزار خود *Optimization* هایی روی مدار انجام می دهد و چون مدار ما هیچ خروجی ای از مدار نداشت مدار را درست نمی کرد. به همین دلیل، یک خروجی از رجیستر فایل برای مدار نهایی در نظر گرفتیم که آن خروجی *SRC2* برای این ماژول بود. بعد از وصل کردن آن به یکی از پین های *FPGA* مدار ساخته شد و توانستیم خروجی را مقایسه کنیم.**
5. **مشکلات دیگری نیز در مراحل وجود داشت ولی سخت ترین آنها با شرح بالاست**