گزارش پروژه پردازنده ARM آزمایشگاه معماری کامپیوتر

مهیار کریمی (۸۱۰۱۹۷۶۹۰)

دانشور امراللهی (۸۱۰۱۹۷۶۸۵)

بخش اول: پیاده‌سازی پردازنده پایه ARM

در این بخش، پیاده‌سازی عملکرد اولیه ARM با کمک ماژول hazard detection برای رفع مخاطره‌های داده‌ای انجام شده است. ماجول‌های پایه این بخش به‌صورت زیر هستند:

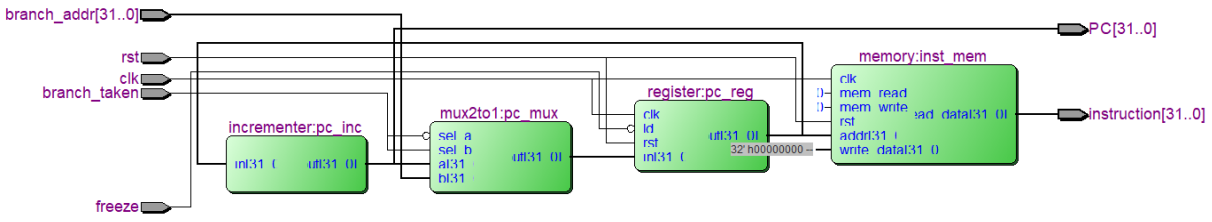
۱. رجیستر با پایه reset و load
۲. مالتی‌پلکسر با یک (و دو) پایه select
۳. ماژول حافظه با پایه‌های read, write؛ در فازهای بعدی، برای حافظه داده از SRAM استفاده می‌شود.

سایر ماژول‌های استفاده شده به‌دلیل طراحی خاص آن‌ها برای همین پردازنده، به‌طور خاص با توجه به طراحی همین پردازنده پیاده‌سازی شده اند.

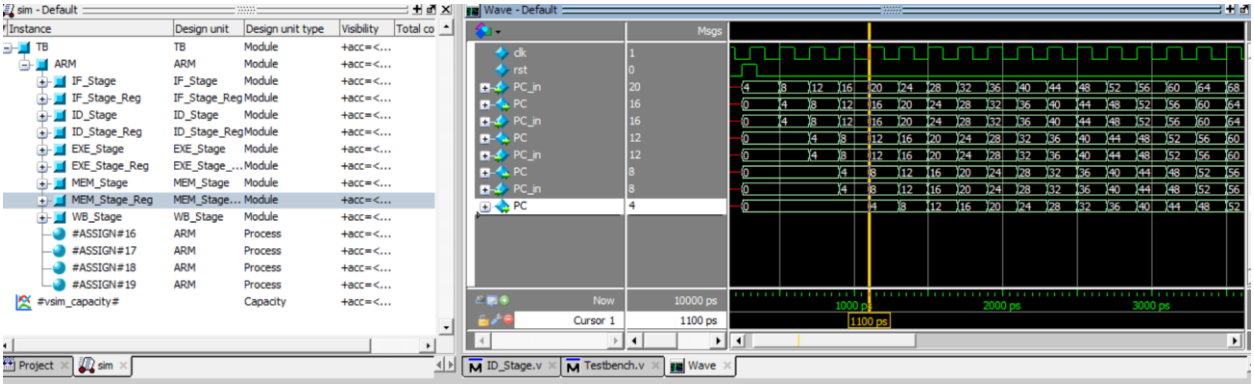
ماژول IF

در این ماژول دستورات از حافظه دستور خوانده می‌شوند و برای decode به ماژول ID فرستاده می‌شوند. همچنین مقدار رجیستر PC نیز با توجه به آخرین دستور خوانده شده (branch بودن یا نبودن آن) یکی از دو مقدار PC + 4 یا مقداری که از branch مشخص شده است را خواهد داشت.

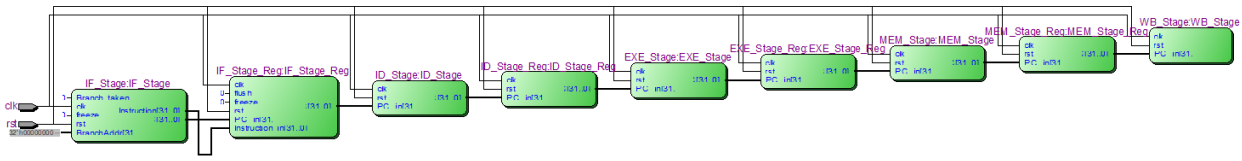
در شکل زیر block diagram مربوط به این بخش را مشاهده می‌کنید (این تصویر پس از کامپایل شدن طراحی در quartus به‌دست آمده‌است.)



در شکل زیر حرکت موج‌گونه PC در مرحله‌های پایپ‌لاین قابل مشاهده است:



شکل زیر وضعیت پردازنده را در حالتی که فقط ماژول IF تکمیل شده است نشان می‌دهد:



در این مرحله مشکلی به‌وجود نخواهد آمد و عملکرد پایپ‌لاین صحیح است.

ماژول ID

این ماژول وظیفه دارد که دستور بدست آمده از مرحله IF را بررسی کند و سیگنال‌های کنترلی متناظر آن را تولید کند. همچنین مقادیر رجیسترهای مورد نیاز و رجیستر مقصد (در مرحله WB) نیز در این بخش تعیین می‌شود.

طبق توضیحات ارائه شده، دستورها در پردازنده ARM به سه دسته قابل تقسیم هستند:

- دسته محاسباتی (arithmetic)
- دسته حافظه (memory)
- دسته branch

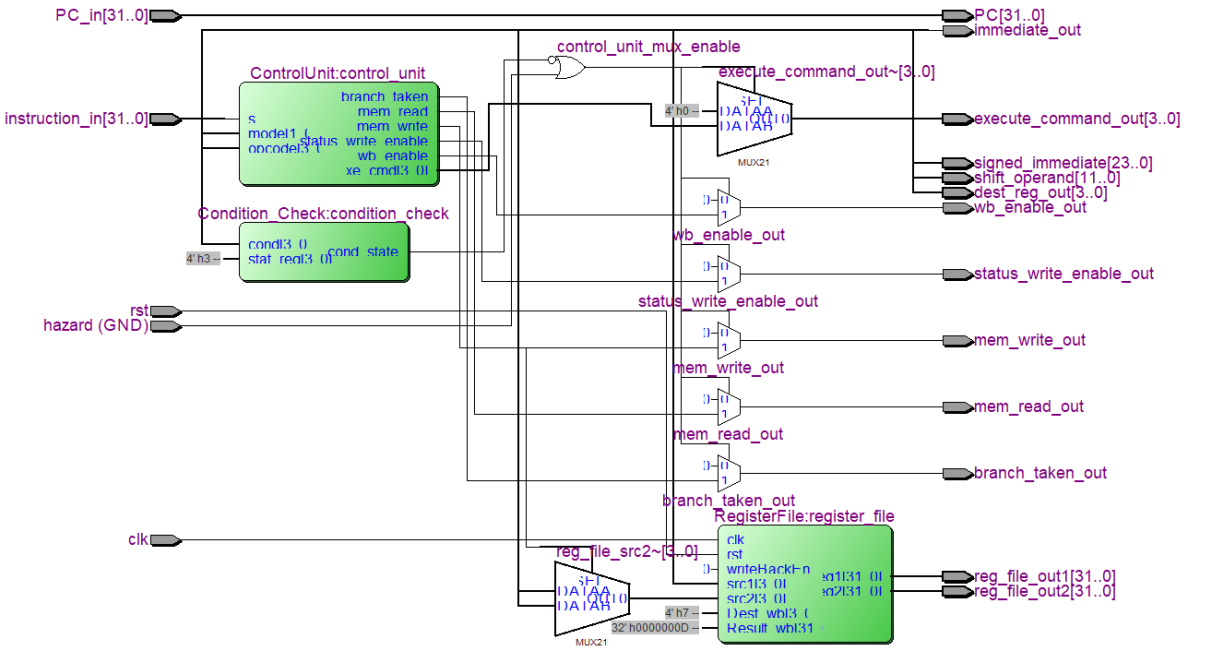
با توجه به دسته‌بندی بالا، بدنه اصلی واحد کنترل به‌صورت زیر طراحی شده است:

```
always @(mode, opcode, s) begin
    exe_cmd = 0; mem_read = 0;
    mem_write = 0; wb_enable = 0;
    branch_taken = 0;
    status_write_enable = 0;

    case (mode)
        `ARITHMETHIC_TYPE : begin···
        end

        `MEMORY_TYPE : begin···
        end

        `BRANCH_TYPE : begin···
        end
    endcase

end
```

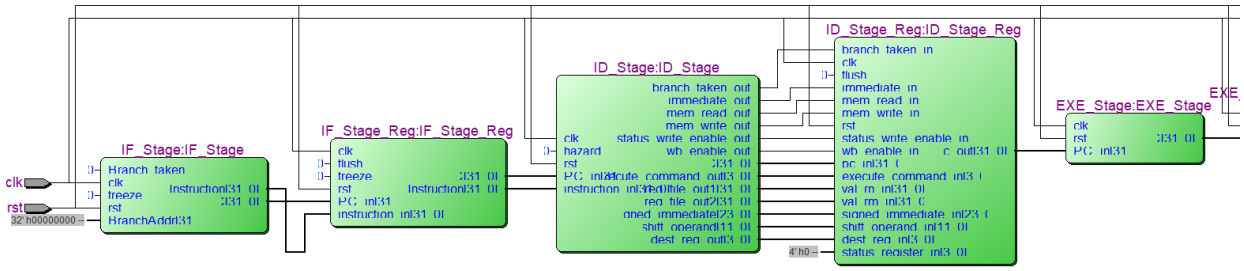
همچنین، در صورتی که شرط اجرای دستور برقرار نباشد، به‌جای همه سیگنال‌های کنترلی، مقدار ثابت صفر می‌گیرد؛ به این ترتیب، اجرای دستور NOP شبیه‌سازی می‌شود.

با توجه به اینکه ماژول hazard برای عملکرد صحیح به پیاده‌سازی کامل هر دو بخش ID و EXE نیازمند است، برای شبیه‌سازی عملکرد صحیح این فاز می‌توانیم فرض کنیم که هیچگاه hazard رخ نمی‌دهد؛ با توجه به اینکه پردازنده در این مرحله عملی انجام نمی‌دهد و تنها دستورهای متناظر اجرای هر دستور را تولید می‌کند، انجام این کار مشکلی در عملکرد پردازنده در این مرحله به‌وجود نمی‌آورد.

شکل زیر block diagram ماژول ID در این مرحله را نشان می‌دهد:



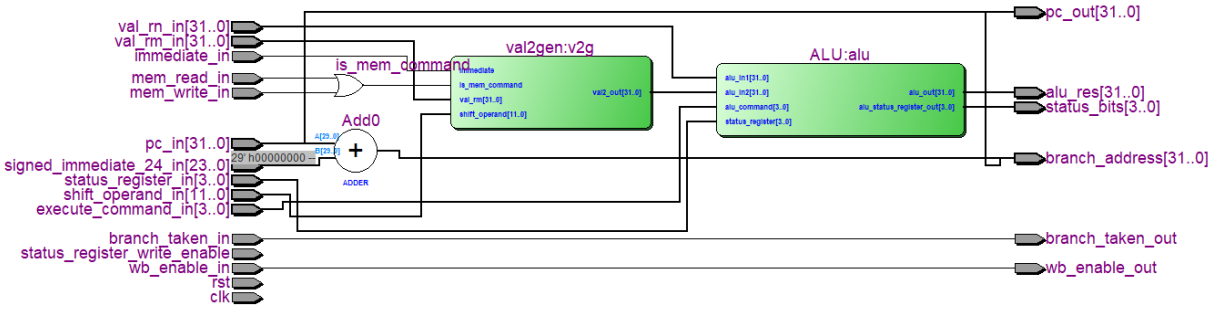
شکل زیر نیز وضعیت پردازنده پس از پیاده‌سازی تقریبا کامل مرحله ID نشان می‌دهد:



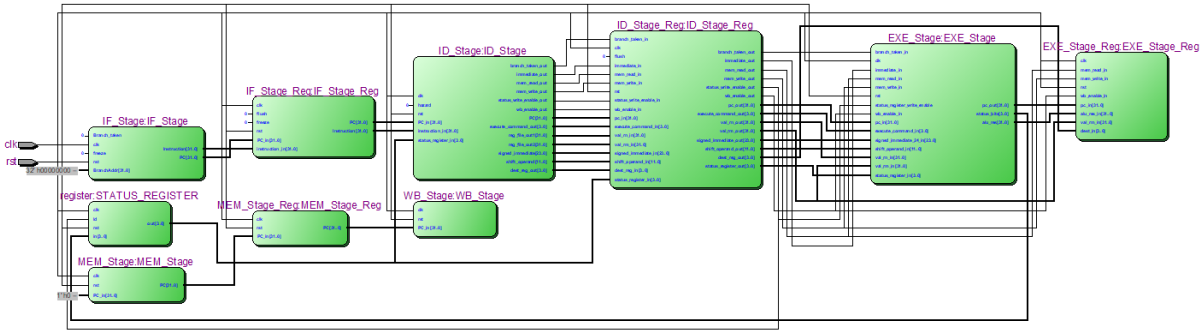
ماژول EXE

عناصر اصلی این ماژول، ALU و Val-2 Generator هستند؛ پیاده‌سازی ماژول ALU پیش‌تر و در پردازنده MIPS نیز انجام شده است بنابراین اینجا از توضیح نحوه پیاده‌سازی این ماژول صرف‌نظر شده است.
ماژول Val-2 Generator برای پیاده‌سازی منطق انتخاب ورودی دوم ماژول ALU می‌باشد. همچنین مقدار بیت‌های رجیستر state نیز توسط ALU تعیین می‌شود.

شکل زیر block diagram ماژول EXE می‌باشد:



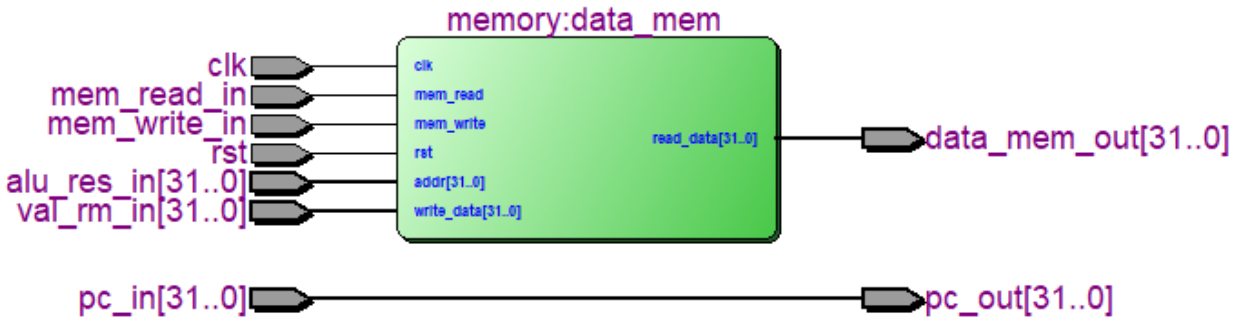
شکل زیر نیز پردازنده را در حالتی که سه بخش ID, IF, EXE پیاده‌سازی شده‌اند نشان می‌دهد:



ماژول MEM

این ماژول عملکرد ساده‌ای دارد و تنها عنصر آن، عنصر حافظه است؛ در ادامه عنصر حافظه در این بخش با عنصر SRAM و سپس SRAM به‌همراه cache جایگزین خواهد شد.

شکل زیر block diagram عنصر حافظه این مرحله را نشان می‌دهد:

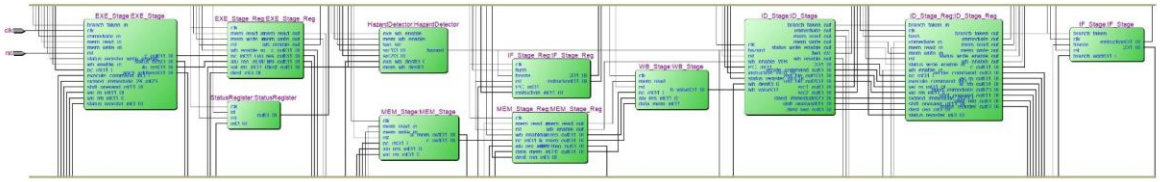


ماژول WB

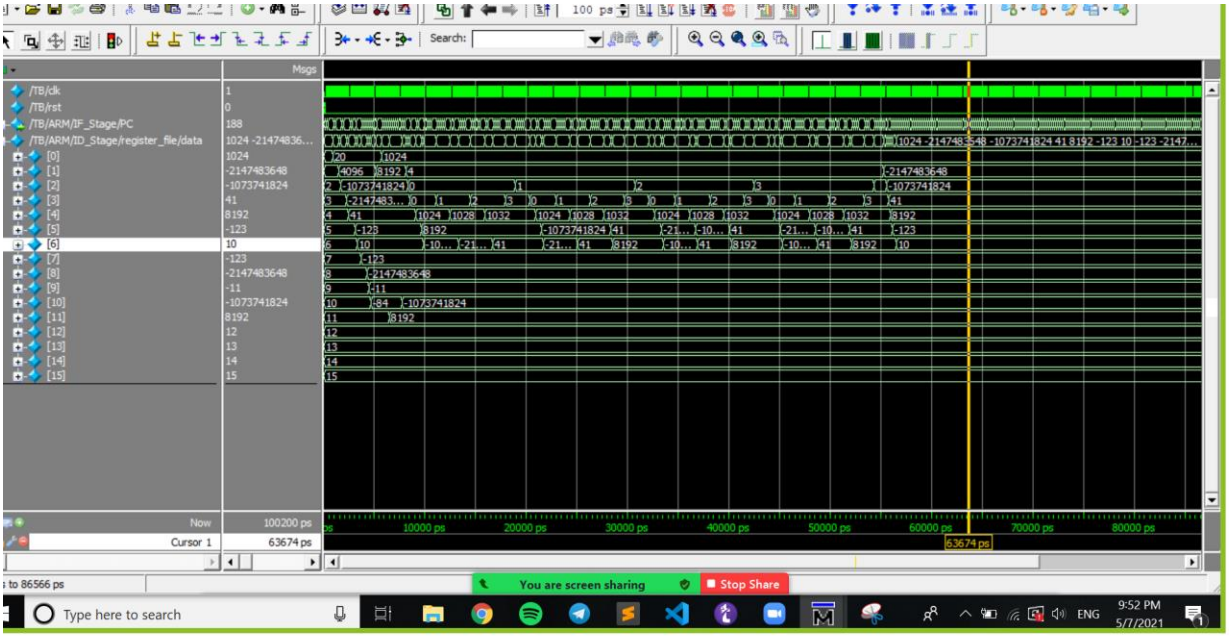
این ماژول عملا یک مالتی‌پلکسراست که مقدار نهایی برای نوشتن در رجیستر فایل را مشخص می‌کند.

آزمون درستی

پس از تکمیل پیاده‌سازی پایه ARM، شکل مسیر داده به‌صورت زیر خواهد بود:



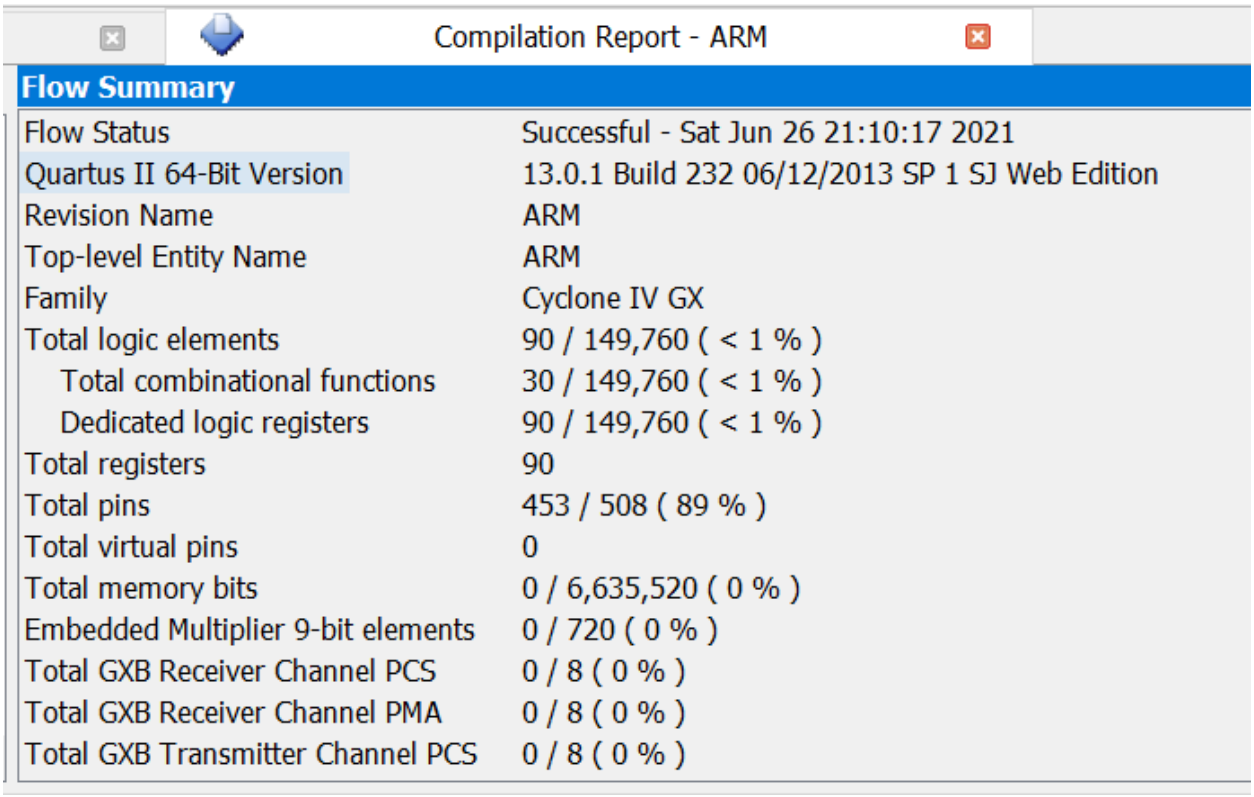
پس از شبیه‌سازی پردازنده در این مرحله، وضعیت نهایی رجیسترها به‌صورت زیر است:



در این مرحله، دوره تناوب clock برابر 200 نانو ثانیه است؛ همچنین، شروع clock از زمان 200 خواهد بود؛ پس تعداد clock-cycle ها طبق رابطه زیر به‌دست می‌آید:

$$(56500 - 200) \div 200 = 281.5$$

نرم‌افزار Quartus تعداد logic elementها را پس از سنتز صفر نشان می‌داد. برای رفع این مشکل، یک خروجی فرضی به طول ۴۵۰ بیت تعریف کردیم و تمام خروجی‌های moduleها را در آن concat کردیم. برای سنتز بخش‌های بعدی هم از همین ایده استفاده کردیم تا نتایج قابل مقایسه باشند.



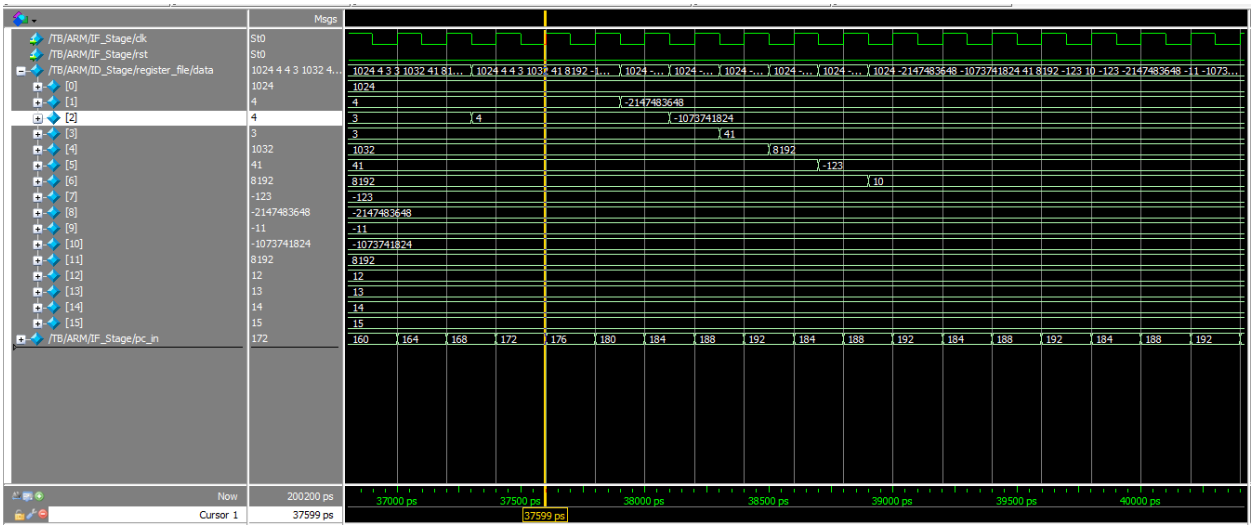
حاصل سنتز ARM بدون فروواردینگ

بخش Forwarding:

```
always@(*) begin
    sel_src1 = 2'b0;
    sel_src2 = 2'b0;
    ignore_hazard = 1'b0;

    if (en_forwarding && WB_wb_en) begin
        if (WB_dst == src1_in) begin
            sel_src1 = `FORW_SEL_FROM_WB;
            ignore_hazard = 1'b1;
        end

        if (WB_dst == src2_in) begin
            sel_src2 = `FORW_SEL_FROM_WB;
            ignore_hazard = 1'b1;
        end
    end

    if (en_forwarding && MEM_wb_en) begin
        if (MEM_dst == src1_in) begin
            sel_src1 = `FORW_SEL_FROM_MEM;
            ignore_hazard = 1'b1;
        end

        if (MEM_dst == src2_in) begin
            sel_src2 = `FORW_SEL_FROM_MEM;
            ignore_hazard = 1'b1;
        end
    end
end
```

شبه‌کد بخش فروواردینگ

در صورتی که رجیستر مقصد نوشتن دستورهایی که در مرحله MEM یا WB هستند قصد نوشتن در رجیستر فایل را داشته باشند، عمل فورواردینگ انجام می‌گیرد تا داده مناسب که هنوز در رجیستر فایل ذخیره نشده، پاس داده شود (سیگنال select مولتی‌پلکسرهایی متناظر به مقدار مناسب تنظیم می‌شود).



در این مرحله، دوره تناوب clock برابر 20 نانو ثانیه است؛ همچنین، شروع clock از زمان 200 خواهد بود؛ پس تعداد clock-cycle ها طبق رابطه زیر به‌دست می‌آید:

$$(4070 - 200) \div 20 = 193.5$$

Compilation Report - ARM

Flow Summary

Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone IV GX
Total logic elements	90 / 149,760 (< 1 %)
Total combinational functions	30 / 149,760 (< 1 %)
Dedicated logic registers	90 / 149,760 (< 1 %)
Total registers	90
Total pins	454 / 508 (89 %)
Total virtual pins	0
Total memory bits	0 / 6,635,520 (0 %)
Embedded Multiplier 9-bit elements	0 / 720 (0 %)
Total GXB Receiver Channel PCS	0 / 8 (0 %)
Total GXB Receiver Channel PMA	0 / 8 (0 %)
Total GXB Transmitter Channel PCS	0 / 8 (0 %)
Total GXB Transmitter Channel PMA	0 / 8 (0 %)
Total PLLs	0 / 8 (0 %)

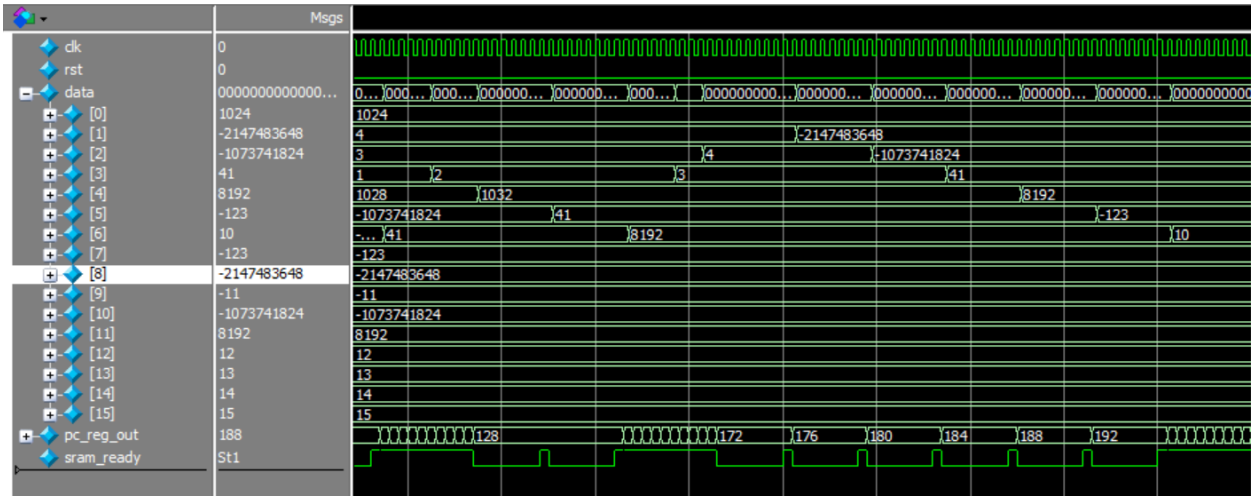
بخش SRAM:

```
always @(*) begin
    case (ps)
        IDLE: begin
            if (read_en)
                ns = READ_WAIT;
            else if (write_en)
                ns = WRITE_WAIT;
            else
                ns = IDLE;
        end

        READ_WAIT: begin
            if (cnt != `SRAM_CNT)
                ns = READ_WAIT;
            else
                ns = IDLE;
        end

        WRITE_WAIT: begin
            if (cnt != `SRAM_CNT)
                ns = WRITE_WAIT;
            else
                ns = IDLE;
        end
    endcase
end
```

واحد SRAM به طور دیفالت در استیت IDLE قرار دارد. با مشاهده سیگنال read_en و یک کلاک، وارد استیت READ_WAIT می‌شود و به تعداد ۴ کلاک صبر می‌کند تا داده خوانده شود. این ۴ کلاک توسط یک counter کمکی شمرده می‌شود. همچنین با مشاهده سیگنال write_en و یک کلاک، وارد استیت WRITE_WAIT می‌شود و مشابه حالت READ_WAIT به تعداد ۴ کلاک صبر می‌کند تا داده نوشته شود.

```
always @(ps) begin
    cnt_enable = 1'b0;
    case (ps)
        IDLE: begin end
        READ_WAIT: cnt_enable = 1'b1;
        WRITE_WAIT: cnt_enable = 1'b1;
    endcase
end

assign ready = ~(read_en || write_en) ? 1'b1 : (cnt == `SRAM_CNT);

always @(posedge clk, posedge rst) begin
    if (rst)
        cnt <= 3'b0;
    else if (cnt_enable) begin
        if (cnt == `SRAM_CNT + 1)
            cnt <= 3'b0;
        else
            cnt <= cnt + 1;
    end
end
```

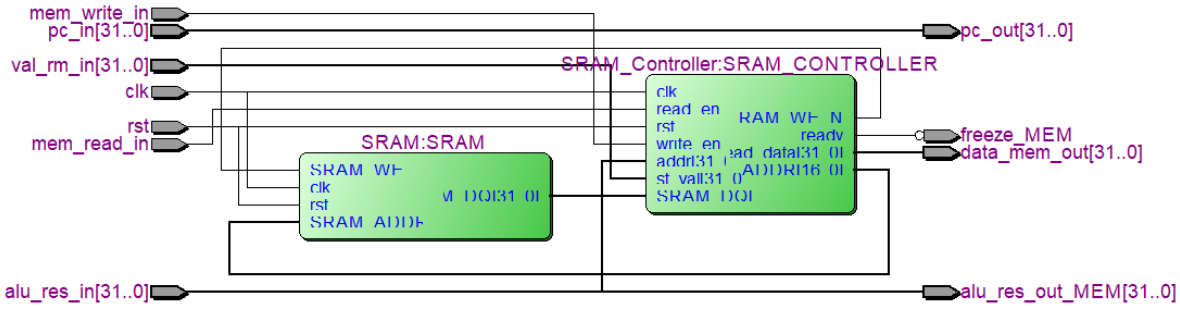
در حالاتی که در استیت‌های READ_WAIT و WRITE_WAIT هستیم منتظر انجام عملیات خواندن/نوشتن هستیم. بنابراین از counterمان می‌خواهیم که با هر کلاک ۱ واحد بشمارد. هرگاه مقدار counter به ۴ برسد، سیگنال ready یک می‌شود. همچنین در حالاتی که عمل خواندن و نوشتن نداشته باشیم هم ready یک می‌شود.



سیگنال sram_ready در حالاتی که دستور load و store دارم را در شکل بالا مشاهده می‌کنید.

در این مرحله، دوره تناوب clock برابر 20 نانو ثانیه است؛ همچنین، شروع clock از زمان 200 خواهد بود؛ پس تعداد clock-cycle ها طبق رابطه زیر به‌دست می‌آید:

$$(9690 - 200) \div 20 = 474.5$$



Compilation Report - ARM

Flow Summary

Flow Status	Successful - Sat Jun 26 20:41:35 2021
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	ARM
Top-level Entity Name	ARM
Family	Cyclone IV GX
Total logic elements	90 / 149,760 (< 1 %)
Total combinational functions	30 / 149,760 (< 1 %)
Dedicated logic registers	90 / 149,760 (< 1 %)
Total registers	90
Total pins	454 / 508 (89 %)
Total virtual pins	0
Total memory bits	0 / 6,635,520 (0 %)
Embedded Multiplier 9-bit elements	0 / 720 (0 %)
Total GXB Receiver Channel PCS	0 / 8 (0 %)
Total GXB Receiver Channel PMA	0 / 8 (0 %)
Total GXB Transmitter Channel PCS	0 / 8 (0 %)

بخش CACHE:

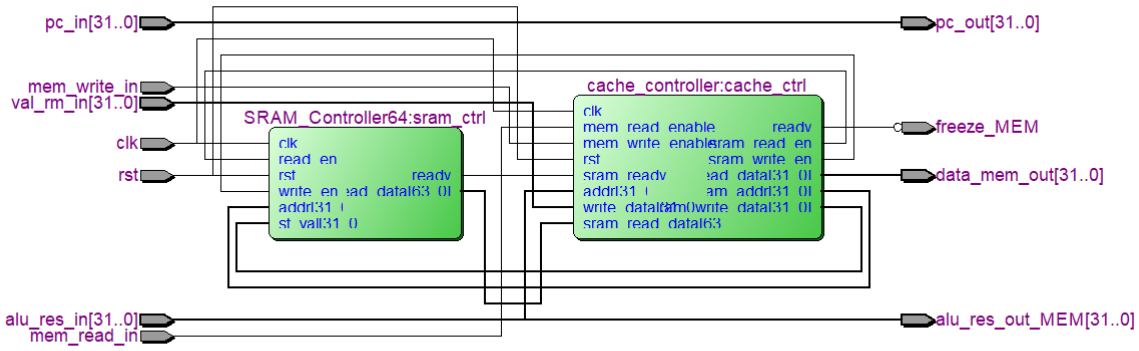
شکل زیر بخشی از کد مربوط به بخش cache را نشان می‌دهد که در آن انتخاب خروجی و همچنین تشخیص بیت hit پیاده‌سازی شده است. همچنین، عمل invalidate کردن آرایه valid در هنگام نوشتن در حافظه اصلی در این شکل نشان داده شده است.

```
wire set_0_hit, set_1_hit;
assign set_0_hit = (tag_0[row] == tag) & valid_0[row];
assign set_1_hit = (tag_1[row] == tag) & valid_1[row];

mux2to1 #(.WORD_LEN(`REGISTER_LEN)) read_data_mux(
    .a(data_0[col][row]),
    .b(data_1[col][row]),
    .sel_a(set_0_hit),
    .sel_b(set_1_hit),
    .out(read_data)
);

assign hit = set_0_hit | set_1_hit;

always @(posedge clk)
if (cache_read_en & hit)
    lru[row] <= set_0_hit ? 1'b0: 1'b1;

always @(posedge clk) begin
    if (invalidate & hit) begin
        if (set_0_hit) begin
            valid_0[row] <= 1'b0;
            lru[row] <= 1'b1;
        end

        else if (set_1_hit) begin
            valid_1[row] <= 1'b0;
            lru[row] <= 1'b0;
        end
    end
end
```

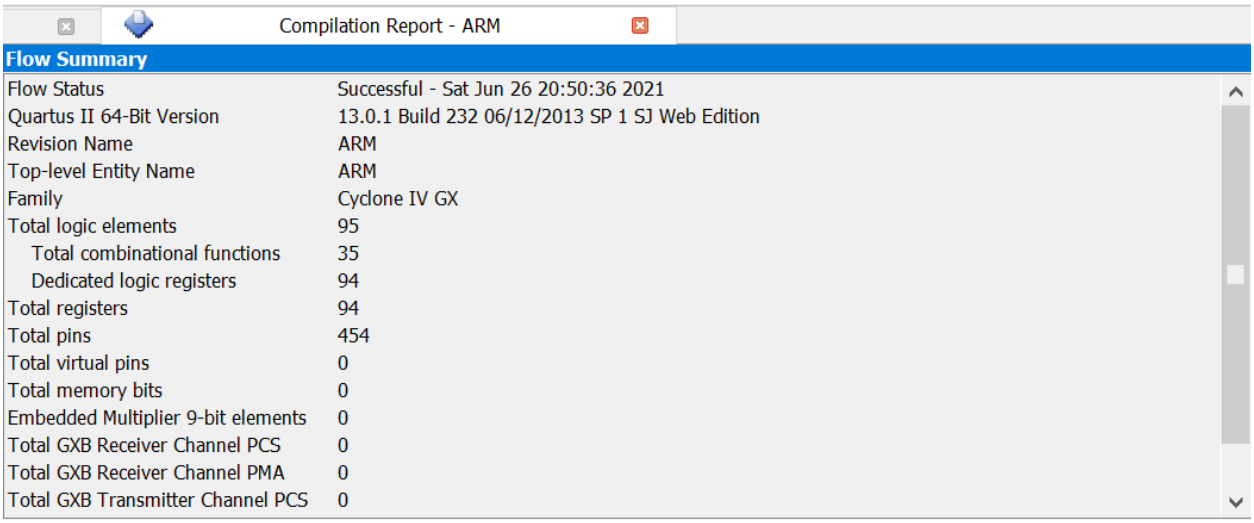
در شکل زیر، پیاده‌سازی نوشتن در cache آمده است؛ طبق توضیحات صورت پروژه، نوشتن در حافظه cache با سیاست LRU انجام می‌شود. با توجه به مقداری که در سطر مورد نظر آدرس در lru نوشته شده است، نوشتن در data_0 یا data_1 انجام می‌شود. جزئیات این کار در زیر آمده است:

```
always @(posedge clk) begin
    if (cache_write_en) begin
        if (~lru[row]) begin
            data_1[1][row] <= write_data[63 : 32];
            data_1[0][row] <= write_data[31 : 0];
            tag_1[row] <= tag;
            valid_1[row] <= 1'b1;
        end

        else if (lru[row]) begin
            data_0[1][row] <= write_data[63 : 32];
            data_0[0][row] <= write_data[31 : 0];
            tag_0[row] <= tag;
            valid_0[row] <= 1'b1;
        end
    end
end
```

شکل زیر block diagram ماژول حافظه پس از اضافه شدن cache_controller است.



برای ساده شدن عمل wiring در ماژول MEM_Stage، واحد cache درون واحد cache_controller می‌باشد. همچنین واحد sram درون واحد sram_controller می‌باشد.



همانطور که انتظار می‌رفت، تعداد logic elementها در مقایسه با حالت قبلی که cache نداشت، بیشتر شد.

در این مرحله، دوره تناوب clock برابر 20 نانو ثانیه است؛ همچنین، شروع clock از زمان 200 خواهد بود؛ پس تعداد clock-cycle ها طبق رابطه زیر به‌دست می‌آید:

$$(7410 - 200) \div 20 = 360.5$$

محاسبه CPI:

در کد testbench، تکه کد زیر را برای شمردن تعداد دستورهای اجرا شده اضافه می‌کنیم:

```
reg [31:0] instr_cnt;
initial instr_cnt = 0;
always @(ARM.IF_Stage.pc_reg_out)
    instr_cnt = instr_cnt + 1;
```

پس از اجرای تست به کمک این کد، مقدار instr_cnt پس از پایان آخرین دستور موثر، مقدار 183 را نشان می‌دهد؛ می‌دانیم این عدد در همه فازها ثابت است زیرا منطق عملکرد پردازنده تغییری نمی‌کند و تنها سرعت آن با روش‌های متفاوت تغییر یافته است. به این ترتیب در رابطه محاسبه CPI داریم:

$$clocks = 183$$

از رابطه زیر نیز برای محاسبه CPI استفاده می‌شود:

$$CPI = \text{Clocks per Instruction} = \frac{clocks}{instruction}$$

بخش پردازنده پایه:

$$CPI = \frac{281.5}{183} = 1.53$$

بخش پردازنده با forwarding:

$$CPI = \frac{193.5}{183} = 1.05$$

بخش پردازنده با sram:

$$CPI = \frac{474.5}{183} = 2.59$$

بخش پردازنده با cache:

$$CPI = \frac{360.5}{183} = 1.96$$

محاسبه speed-up پردازنده با cache به نسبت حالت بدون cache:

$$\text{speed-up} = \frac{2.59}{1.96} = 1.32$$

محاسبه speed-up پردازنده با حالت forwarding به نسبت حالت پایه:

$$\text{speed-up} = \frac{1.53}{1.05} = 1.45$$