بسمه تعالی

آزمایشگاه سیستم‌های دیجیتال۲

گزارش کار آزمایشگاه سیستم‌های دیجیتال۲

پردیس دانشکده‌های فنی دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

علی ایمانقلی ۸۱۰۱۹۷۶۹۲ – عمید مهرجو ۸۱۰۱۹۸۳۳۳

نیمسال اول ۱۴۰۲-۱۴۰۱

استفاده از $SRAM$ در پردازنده ARM به عنوان حافظه داده:

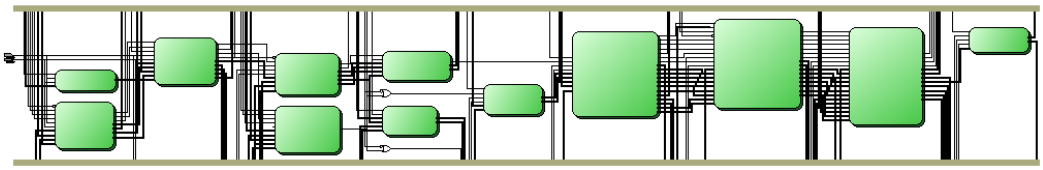
در بخش های قبل حافظه‌ی پردازنده را به صورت ایده‌آل در نظر می گرفتیم، به نحوی که در همان سیکلی که آدرس را بر روی پورت آدرس حافظه قرار می دادیم، عملیات خواند و نوشتن داده انجام می شد. اما در این قسمت می خواهیم حافظه‌ی به کار رفته در پردازنده را به واقعیت نزدیک تر نماییم، فلذا از حافظه‌ی $SRAM$ استفاده می نماییم. همانطور که در صورت گزارش عنوان شده است، برای انجام عملیات های خواندن و نوشتن هنگامی که از حافظه‌ی $SRAM$ استفاده نمی نماییم، نیازمند به ۶ سیکل کلاک می باشیم.

با توجه به اینکه دیگر عملیات دسترسی به حافظه در یک کلام انجام نمی شود و به ۶ کلاک نیاز داریم، باید در هنگام دسترسی به حافظه، پردازنده را $stall$ نماییم تا دستورات بعدی اجاره نشوند و منتظر باشند تا دسترسی به حافظه به صورت کامل انجام شود و سپس دستورات بعدی اجرا شوند.

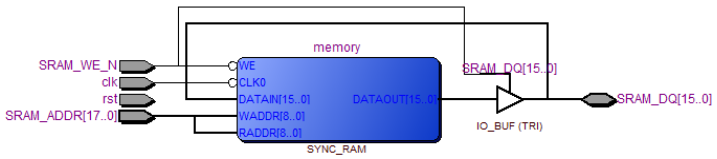
روند دسترسی به حافظه در عملیات های خواندن و نوشتن نیز متفاوت می باشند و به علاوه داده های ذخیره شده در خانه‌های $SRAM$ به صورت ۱۶ بیتی می باشند ولی پردازنده‌ی طراحی شده با داده های ۳۲ بیتی سروکار دارد. این موارد بیانگر این است که نیازمند ماژولی برای مدیریت دسترسی به حافظه می باشیم.

پس علاوه بر اینکه به ماژول حافظه نیاز داریم، به یک ماژول کنترلر نیز برای کنترل کردن دسترسی به حافظه و $stall$ کردن پردازنده در مواقع مورد نظر، نیازمندیم.

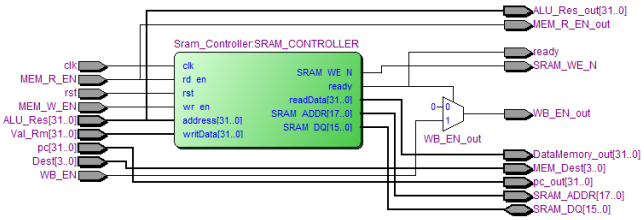
RTL View کلی پردازنده:



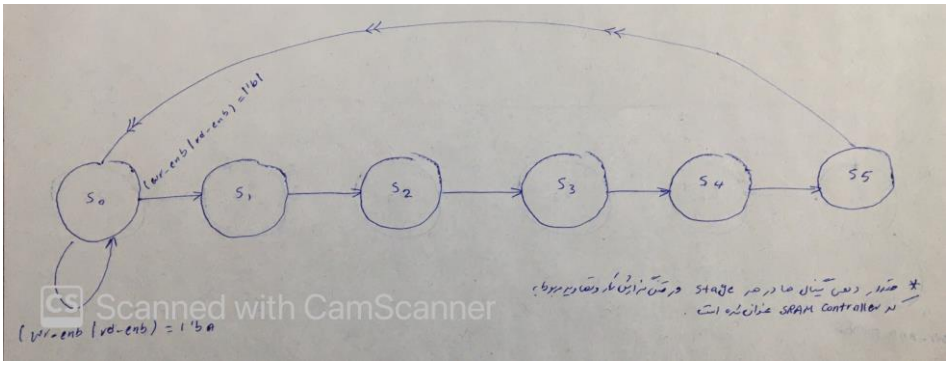
RTL View مربوط به *SRAM*:



RTL View مربوط به *SRAM Controller*:



تصویر state diagram مربوط به SRAM Controller:



اطلاعات مربوط به مقدار دهی سیگنال ها در هر stage قابل بیان در تصویر بالا نمی باشد به همین علت در تصویر بالا نشان داده نشده است، ولیکن در ادامه گزارش کار به تفصیل اطلاعات مربوط توضیح داده شده است.

در ابتدا ماژول کلی حافظه‌ی $SRAM$ را ایجاد می نماییم:

```
`timescale 1ns/1ns
module SRAM(
input clk,
input rst,
input SRAM_WE_N,
input [17:0]SRAM_ADDR,
inout [15:0]SRAM_DQ
);

reg[15:0] memory[0:511];
assign #5 SRAM_DQ=(SRAM_WE_N==1'b1)?memory[SRAM_ADDR]:16'dz;

always@(negedge clk, posedge rst) //write
begin
    if(SRAM_WE_N==1'b0)
        memory[SRAM_ADDR] = SRAM_DQ;
end

endmodule
```

در قطعه کد بالا، در ابتدا ورودی و خروجی ماژول $SRAM$ را تعیین می نماییم. کلاک و ریست از سیگنال های ورودی به منظور همگام کردن این ماژول با باقی ماژول های پردازنده می باشند. علاوه بر کلاک و ریست، ۳ سیگنال دیگر نیز داریم:

$SRAM\_WE\_N$: زمانی که بخواهیم در حافظه عملیات نوشتن را انجام دهیم این سیگنال ۰ می شود. ($active\ low$) و اگر بخواهیم عملیات خواندن را انجام دهیم این سیگنال ۱ می شود.

$SRAM\_ADDR$: این سیگنال آدرس محل خواندن یا نوشتن از حافظه را نشان می دهد.

$SRAM\_DQ$: این سینال هم به صورت ورودی و هم به صورت خروجی می باشد. این ماژول هنگامی مقدار دهی می شود که بخواهیم عملیات خواندن را انجام دهیم، در این صورت ماژول $SRAM$ مقدار این سیگنال را درایو می نمایم. در غیر این صورت ماژول $SRAM$ این سیگنال را $high\ impedance$ یا z می نماید.

در هنگام عملیات نوشتن نیز، ماژول $SRAM$ مقدار داده‌ی موردنظر را از این سگنال می خواند و در خانه‌ی موردنظر از حافظه قرار می دهد.

در ادامه‌ی قطعه کد بالا، حافظه ۵۱۲ خانه‌ی ۱۶ بیتی تعریف می شود که این حافظه ساختار کلی $SRAM$ را تشکیل خواهد داد.

سپس در ادامه‌ی کد عملیات خواندن از حافظه مدریت می شود:

```
assign #5 SRAM_DQ=(SRAM_WE_N==1'b1)?memory[SRAM_ADDR]:16'dz;
```

همانطور که از کد بالا مشخص است، زمانی که سیگنال $SRAM\_WE\_N$ برابر با ۱ باشد، پس از ۵ نانوثانیه مقدار خانه‌ی مشخص شده توسط $SRAM\_ADDR$ بر روی سیگنال $SRAM\_DQ$ قرار می گیرد و به عنوان خروجی $SRAM$ به پردازنده تحویل داده می شود. ولی اگر مقدار $SRAM\_WE\_N$ برابر با ۰ باشد مقدار ۱۶ بیت z بر روی سیگنال خروجی $SRAM\_DQ$ قرار می گیرد.

دو نکته‌ای که باید توجه داشت:

* علت اینکه خواند از حافظه ۵ نانو ثانیه طور می کشد در این است که طبق مطالب عنوان شده در صورت آزمایش، عملیات خواندن از حافظه در سیکل یکسانی صورت نمیگیرد، بدین معنی که در ابتدا آدرس خانه‌ی مورد نظر برروی پورت آدرس حافظه قرار می گیرد و سپس در سیکل بعدی مقدار مورد نظر از پورت $SRAM\_DQ$ خوانده می شود.

* علت اینکه سیگنال $SRAM\_DQ$ را هنگامی $SRAM\_WE\_N$ برابر با ۰ است ۱۶ بیت Z می کنیم در این است که سیگنال $SRAM\_DQ$ به صورت پورت $inout$ تعریف شده است، به نحوی که یا توسط $SRAM$ مقدار دهی می شود و یا توسط $SRAM\ Controller$ مقدار دهی می شود. حال برای اینکه در هنگام مقدار دهی این سیگنال توسط ماژول های $SRAM$ و $SRAM\ Controller$ تعارضی رخ ندهد، باید این سیگنال همواره $high\ impedance$ باشد مگر در صورتی که ماژولی بخواهد آن را مقدار دهی نماید.

در ادامه‌ی کد عملیات نوشتن در حافظه تعریف می گردد:

```
always@(negedge clk, posedge rst) //write
begin
    if(SRAM_WE_N==1'b0)
        memory[SRAM_ADDR] = SRAM_DQ;
end
```

با توجه به کد بالا در لبه‌ی پایین رونده‌ی کلاک، اگر مقدار سیگنال $SRAM_WE_N$برابر با ۰ باشد، مقدار $SRAM\_DQ$ در خانه‌ی موردنظر از حافظه نوشته می شود.

حال در ادامه ماژول $SRAM\ Controller$ را بررسی می نماییم:
کد ماژول موردنظر به صورت زیر می باشد:

```
module Sram_Controller(
input clk,
input rst,
//From Memory Stage
input wr_en,
input rd_en,
input [31:0] address,
input [31:0] writData,
//To Next Stage
output reg [31:0] readData,
//For freeze Other Stage
output reg ready,
inout  [15:0] SRAM_DQ,
output reg [17:0] SRAM_ADDR,
output SRAM_UB_N,
output SRAM_LB_N,
output reg SRAM_WE_N,
output SRAM_CE_N,
output SRAM_OE_N
);

assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'b1111;


// controller stages
reg [2:0] ns, ps;
parameter [2:0] S_0 = 3'b000,
                S_1 = 3'b001,
                S_2 = 3'b010,
                S_3 = 3'b011,
                S_4 = 3'b100,
                S_5 = 3'b101;

always@(posedge clk, posedge rst)
begin
    if(rst==1'b1)
        ps<=S_0;
    else
        ps<=ns;
end

always@(ps, wr_en, rd_en)
begin
    case(ps)
        S_0: if(wr_en | rd_en) ns <= S_1; else ns <= S_0; // if wr_en=1 or rd_en=1 next stage is S_1 otherwise is S_0
        S_1: ns=S_2;
        S_2: ns=S_3;
        S_3: ns=S_4;
        S_4: ns=S_5;
        S_5: ns=S_0;
    endcase
end
```

```
always@(ps, wr_en, rd_en)
begin
    //initial vlaue
    ready=1'b1;
    SRAM_WE_N=1'b1; // deactive high

    case(ps)
        S_0:
        begin
        if(wr_en | rd_en)
            ready=1'b0;
        else
            ready=1'b1;
        end

        S_1:
        begin
            if(wr_en==1'b1)
            begin
                SRAM_ADDR=(address-32'd1024)>>1; //low address
                SRAM_WE_N=1'b0; // active low
                ready=1'b0;
            end
            else // rd_en==1'b1
            begin
                SRAM_ADDR=(address-32'd1024)>>1; //low address
                SRAM_WE_N=1'b1; // deactive high
                ready=1'b0;
            end
        end

        S_2:
        begin
             if(wr_en==1'b1)
             begin
                 SRAM_ADDR=((address-32'd1024)>>1)+1'b1; //high address=low address+1
                 SRAM_WE_N=1'b0; // active low
                 ready=1'b0;
             end
             else // rd_en==1'b1
             begin
                 SRAM_ADDR=((address-32'd1024)>>1)+1'b1; //high address=low address+1
                 SRAM_WE_N=1'b1; // deactive high
                 readData[15:0]=SRAM_DQ;
                 ready=1'b0;
             end
        end

        S_3:
        begin
             if(rd_en==1'b1)
                 readData[31:16]=SRAM_DQ;

             ready=1'b0;
        end

        S_4: ready=1'b0;

        S_5: ready=1'b1;

    endcase
end
assign SRAM_DQ=(SRAM_WE_N==1'b0 && ps==S_1)?writData[15:0]:// write low data
               (SRAM_WE_N==1'b0 && ps==S_2)?writData[31:16]:// write high data
                16'dz;

endmodule
```

حال به تفکیک هر بخش را بررسی می نماییم:

ورودی و خروجی های ماژول:

```
module Sram_Controller(
input clk,
input rst,
//From Memory Stage
input wr_en,
input rd_en,
input [31:0] address,
input [31:0] writData,
//To Next Stage
output reg [31:0] readData,
//For freeze Other Stage
output reg ready,
inout  [15:0] SRAM_DQ,
output reg [17:0] SRAM_ADDR,
output SRAM_UB_N,
output SRAM_LB_N,
output reg SRAM_WE_N,
output SRAM_CE_N,
output SRAM_OE_N
);
```

کلاک و ریست برای همگام سازی ماژول $SRAM\ Controller$ با باقی ماژول های پردازنده.

wr_en و rd_en: این سیگنال های ورودی که توسط دستور موردنظر مشخص می شوند، تعیین می نمایند که باید عملیات خواندن از حافظه انجام شود و یا عملیات نوشتن از حافظه

$address$: سیگنال ورودی ای که آدرس محل نوشتن و یا خواند از حافظه را تعیین می نماید.

$writData$: سیگنال ورودی که مقدار داده ی موردنظر برای نوشتن در حافظه را تعیین می نماید.

$readData$: سیگنال خروجی ای که حاوی مقدار حافظه در خانه‌ی مورد نظرمی باشد.

$ready$: این سیگنال تعیین می نماید که آیا عملیات خواندن و نوشتن از حافظه به صورت کامل صورت گرفته است یا خیر.

توسط این سیگنال می توانیم تا زمانی که عملیات دسترسی به حافظه به صورت کامل صورت نگرفته، پردازنده را $stall$ نماییم.

$SRAM\_DQ$: سیگنالی از جنس $inout$ که انتقال داده میان $SRAM\ controller$ و $SRAM$ را مدیریت می نماید.

$SRAM\_ADDR$: آدرس دسترسی به خانه‌های $SRAM$

$SRAM\_WE\_N$: سیگنالی برای مشخص نمودن عملیات نوشتن و یا خواندن در حافظه (به صورت active low می باشد.)

$SRAM_UB_N$و $SRAM_LB_N$و $SRAM\_CE\_N$ و $SRAM\_OE\_N$: برخی از سیگنال های کنترلی SRAM که در این آزمایش به منظور سادگی، آن ها را فقط مقدار دهی اولیه می نماییم.

در ادامه ۴ پین حافظه را مقدار دهی می نماییم(طبق خواسته‌ی صورت آزمایش):

```
assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'b1111;
```

در ادامه $state$ های کنترلی $SRAM\ Controller$ را تعریف می نماییم:

```
// controller stages
reg [2:0] ns, ps;
parameter [2:0] S_0 = 3'b000,
                S_1 = 3'b001,
                S_2 = 3'b010,
                S_3 = 3'b011,
                S_4 = 3'b100,
                S_5 = 3'b101;
```

ماژول $SRAM\ Controller$ دسترسی به حافظه را در ۶ سیکل مدیریت می نماید(طبق مطالب عنوان شده در صورت آزمایش)

$state$ صفر: تا زمانی که سیگنال های wr_en و یا rd_en فعال نشده اند، در این $state$ باقی می مانیم.

این بدین معنی است که پردازنده قصد دسترسی به حافظه را ندارد. ولیکن به محض فعال شدن هر کدام از این سیگنال ها در لبه‌ی بالا رونده کلاک وارد $state$ بعدی می شویم.

$state$ های ۱ تا ۵: با هربار لبه‌ی بالا رونده‌ی کلاک وارد $state$ بعدی می شویم.

```
always@(posedge clk, posedge rst)
begin
    if(rst==1'b1)
        ps<=S_0;
    else
        ps<=ns;
end

always@(ps, wr_en, rd_en)
begin
    case(ps)
        S_0: if(wr_en | rd_en) ns <= S_1; else ns <= S_0; // if wr_en=1 or rd_en=1 next stage is S_1 otherwise is S_0
        S_1: ns=S_2;
        S_2: ns=S_3;
        S_3: ns=S_4;
        S_4: ns=S_5;
        S_5: ns=S_0;
    endcase
end
```

حال بررسی نماییم که در هر $state$ چه پردازشی صورت می گیرد:

باید توجه داشت که خواندن از $SRAM$ و نوشتن در آن به صورت ۱۶ بیتی انجام می شود و از آنجاییکه پردازنده ی طراحی شده با داده های ۳۲ بیتی سروکار دارد، باید این ۳۲ بیت در چند سیکل منتقل شود.

```
always@(ps, wr_en, rd_en)
begin
    //initial vlaue
    ready=1'b1;
    SRAM_WE_N=1'b1; // deactive high

    case(ps)
        S_0:
        begin
        if(wr_en | rd_en)
            ready=1'b0;
        else
            ready=1'b1;
        end
```

در ابتدا سیگنال های $ready$ و $SRAM\_WE\_N$ را به حالت غیر فعال مقدار دهی اولیه می نماییم. باید توجه داشت که سیگنال های $ready$ و $SRAM\_WE\_N$ به صورت $active\ low$ می باشند.

سپس در $state$ بعدی، $state$ صفر، بررسی می نماییم که اگر سیگنال های wr_en و یا rd_enفعال بودند یعنی می خواهیم دسترسی به حافظه داشته باشیم پس باید سیگنال $ready$ را صفر نماییم تا در پردازنده $stall$ ایجاد نماییم، در غیر این صورت مقدار سیگنال $ready$ را یک می نماییم بدین معنی که دسترسی به حافظه نداریم و نمی خواهیم در پردازنده $stall$ ایجاد نماییم.

```
S_1:
begin
    if(wr_en==1'b1)
    begin
        SRAM_ADDR=(address-32'd1024)>>1; //low address
        SRAM_WE_N=1'b0; // active low
        ready=1'b0;
    end
    else // rd_en==1'b1
    begin
        SRAM_ADDR=(address-32'd1024)>>1; //low address
        SRAM_WE_N=1'b1; // deactive high
        ready=1'b0;
    end
end
```

در $state$ بعد، $state$ یک، مطمئن هستیم که می خواهیم دسترسی به حافظه انجام دهیم، پس تنها بررسی می کنیم که آیا می خواهیم عملیات نوشتن را انجام دهیم و یا عملیات خواندن را.

در عملیات نوشتن، سیگنال $SRAM\_ADDR$ را مقدار دهی می نماییم (بر اساس سیگنال ورودی $address$)، سپس مقدار سیگنال $SRAM\_WE\_N$ را صفر و یا فعال می نماییم. به علاوه سیگنال $ready$ را نیز در مقدار ۰ حفظ می نماییم.

اگر عملیات از نوع خواند بود: سیگنال $SRAM\_ADDR$ را مقدار دهی می نماییم (بر اساس سیگنال ورودی $address$)، سپس مقدار سیگنال $SRAM\_WE\_N$ را یک و یا غیر فعال می نماییم. به علاوه سیگنال $ready$ را نیز در مقدار ۰ حفظ می نماییم.

```
S_2:
begin
    if(wr_en==1'b1)
    begin
        SRAM_ADDR=((address-32'd1024)>>1)+1'b1; //high address=low address+1
        SRAM_WE_N=1'b0; // active low
        ready=1'b0;
    end
    else // rd_en==1'b1
    begin
        SRAM_ADDR=((address-32'd1024)>>1)+1'b1; //high address=low address+1
        SRAM_WE_N=1'b1; // deactive high
        readData[15:0]=SRAM_DQ;
        ready=1'b0;
    end
end

S_3:
begin
    if(rd_en==1'b1)
        readData[31:16]=SRAM_DQ;

    ready=1'b0;
end

S_4: ready=1'b0;

S_5: ready=1'b1;

endcase
end
```

در $state$ بعدی، $state$ دوم، همان عملیات های $state$ یک را انجام می دهیم. تنها تفاوت هایی که وجود دارد در این است که مقدار آدرس را یک واحد زیاد می نماییم تا ۱۶ بیت بعدی را دریافت نماییم.

و اگر عملیات از نوع خواند باشد، مقدار داده‌ی مشخص شده توسط آدرس $state$ قبل را در $readData$ قرار می دهیم(۱۶ بیت LSB)

تذکر: همانطور که می دانیم، در عملیات خواندن از حافظه در ابتدا آدرس مورد نظرتعیین می گردد و در لبه‌ی بالا رونده‌ی کلاک، داده‌ی موردنظر خوانده می شود.

فرایند ذکر شده برای $state$ دوم، در $state$ سوم نیز تکرار می شود.

در $state$ چهارم عملیاتی انجام نمی شود و تنها مقدار سیگنال $ready$ برابر با ۰ تعیین می گردد(حالت غیر فعال)

در state پنجم، عملیات دسترسی به حافظه تکمیل شده است، بنابراین مقدار سیگنال $ready$ را ۱ می نماییم تا پردازنده داده ی موردنظر را دریافت نماید و دچار $stall$ نگردد.

در هنگام نوشتن داده در $SRAM$ باید مقدار $writData$ را به صورت ۱۶ بیت، ۱۶ بیت درچند $state$ برروی پورت $SRAM_DQ$قرار دهیم. پورت $SRAM\_DQ$ از جنس $inout$ می باشد و نمی توان آن را در $alwasys\ statment$ مقدار دهی نمود، پس با استفاده از $assign\ statement$ و تعیین شرط های مناسب مقدار $writData$ را بر روری $SRAM\_DQ$ قرار می دهیم.

قطعه کد زیر نحوه‌ی این فرایند را نشان می دهد:

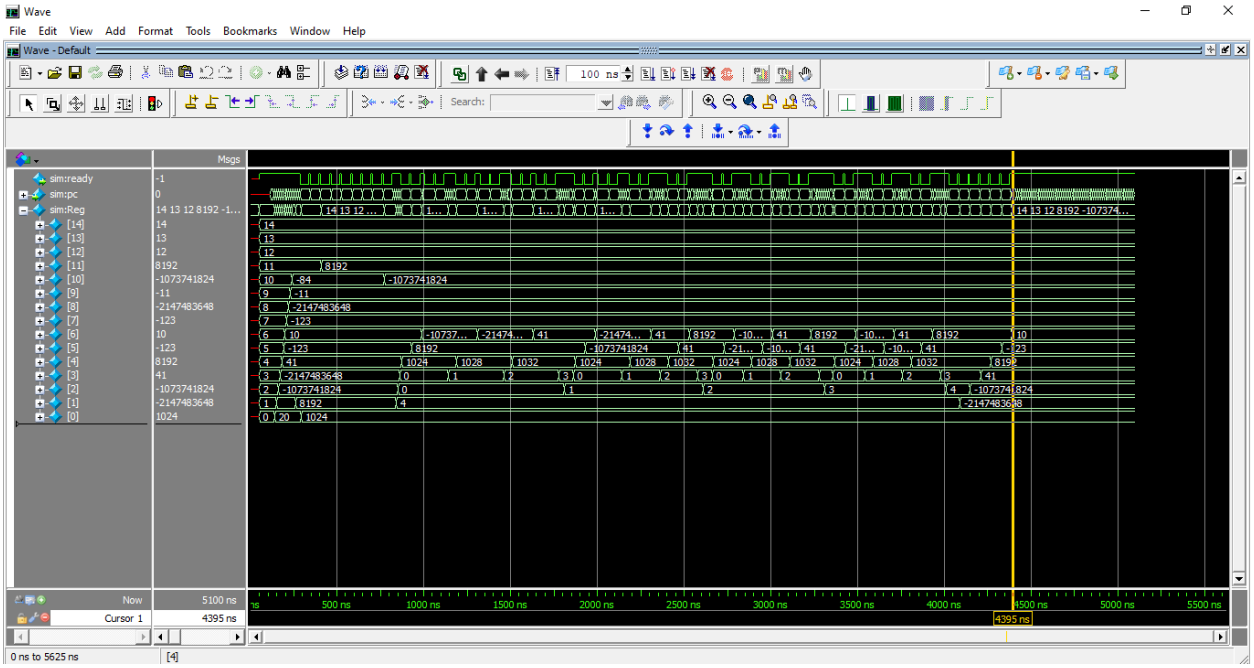
```
assign SRAM_DQ=(SRAM_WE_N==1'b0 && ps==S_1)?writData[15:0]:// write low data
               (SRAM_WE_N==1'b0 && ps==S_2)?writData[31:16]:// write high data
                16'dz;
```

در انتها باید ماژول های $SRAM$ و $SRAM\ Controller$ را به یکدیگر متصل نماییم و سیگنال $ready$ را نیز به $stage$ های دیگر پردازنده ارسال نماییم تا بتواند در مواقع دسترسی به حافظه آن ها را $stall$ نماید.

```
input MEM_W_EN,
input [31:0] ALU_Res,
input [31:0] Val_Rm,
input [3:0] Dest,
input [31:0] pc,
output WB_EN_out,
output MEM_R_EN_out,
output [31:0]ALU_Res_out,
output [31:0]DataMemory_out,
output [3:0] MEM_Dest,
output [31:0] pc_out,
// Sram Controller
output ready,
inout  [15:0] SRAM_DQ,
output [17:0] SRAM_ADDR,
output SRAM_WE_N
);

assign pc_out = pc;
assign WB_EN_out=(ready==1'b1)?WB_EN:1'b0; // freezing WB_EN_out based on ready signal of SRAM controller
assign MEM_R_EN_out = MEM_R_EN;
assign MEM_Dest = Dest;
assign ALU_Res_out = ALU_Res;


Sram_Controller SRAM_CONTROLLER(
.clk(clk),
.rst(rst),
.wr_en(MEM_W_EN),
.rd_en(MEM_R_EN),
.address(ALU_Res),
.writData(Val_Rm),
.readData(DataMemory_out),
.ready(ready),
.SRAM_DQ(SRAM_DQ),
.SRAM_ADDR(SRAM_ADDR),
.SRAM_WE_N(SRAM_WE_N)
);

endmodule
```

پس برای متصل نمودن ماژول‌های $SRAM$ و $SRAM\ Controller$ آن ها را درون $MEM\ stage$ فراخوانی می نماییم و سیگنال های موردنظر را نیز برای آن ها تعریف می نماییم. قطعه کد بالا این فرایند را نشان می دهد.

اجرای برنامه‌ی محک و بررسی صحت عملکرد پردازنده:



همانطور که از تصویر بالا مشخص است، خروجی برنامه محک مطابق با انتظار، صحیح می باشد.

به میزان $4395ns$ طول کشیده است تا تمامی خروجی های برنامه‌ی محک آماده شوند؛ این درحالی است که مرحله‌ی قبل، یعنی پردازنده به همراه حافظه ایده آل ، پس از $2040ns$ خروجی آماده می شود.

این یعنی پردازنده به همراه $SRAM$ به میزان $4395ns - 2040ns = 2355\ ns$ نسبت پردازنده به همراه حافظه ایده آل **کاهش $performance$** داشته است.

نتایج سنتز پردازنده طراحی شده تا بدین مرحله:

Flow Status	Successful - Sat Feb 04 21:10:19 2023
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 II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	0 / 33,216 (0 %)
Total combinational functions	0 / 33,216 (0 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	2 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

همانطور که از نتایج سنتز مشخص است، تعداد المان های به کار رفته در پردازنده طراحی شده (تا بدین جا)

افزایش یافته است، که البته منطقی نیز می باشد، زیرا در این مرحله ۲ ماژول جدید تحت عنوان SRAM و SRAM Controller به پردازنده قبلی اضافه شده است که منجر به پیچیدگی سخت افزاری و افزایش تعداد المان های به کار رفته می شود.

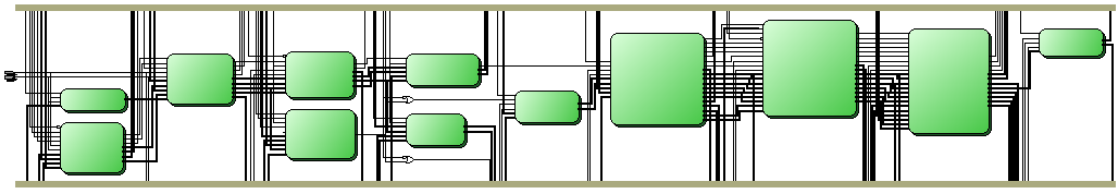
بخش امتیازی:

به منظور افزایش کارایی پردازنده (با وجود $SRAM$ و بدون $Cache$) می توانیم از رویکرد $In\ memory\ computiong$ بهره ببریم. بدین ترتیب که اگر برای انجام عملیاتی نیاز داریم ابتدا داده ی حافظه را بخوانیم و سپس آن را در $Register\ File$ ذخیره نماییم و توسط دستور بعدی بر روی داده ی مورد نظر پردازش انجام دهیم. به جای این کار، یک دستور جدید به پردازنده معرفی نماییم که هنگامی که داده ی موردنظر را از حافظه می خواند پردازش مورد نظر را نیز انجام دهد و بدین ترتیب در زمانی کمتر و تنها توسط یک دستور عملیات موردنظر انجام می شود. البته نیازمند اضافه کردن واحد محاسبات (ALU) به $MEM\ stage$ و تغییر در $SRAM\ Controller$ می باشیم.

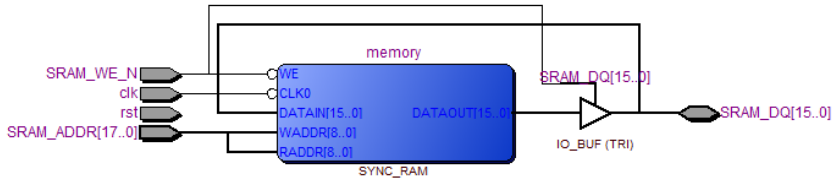
استفاده از حافظه نهان ($Cache$) در پردازنده ARM:

از انجاییکه استفاده از $SRAM$ منجر شد تا کارایی پردازنده طراحی شده کاهش یابد، به عنوان یک راه حل به منظور افزایش کارایی، از $Cache$ به عنوان حافظه‌ی نهان استفاده می نماییم.

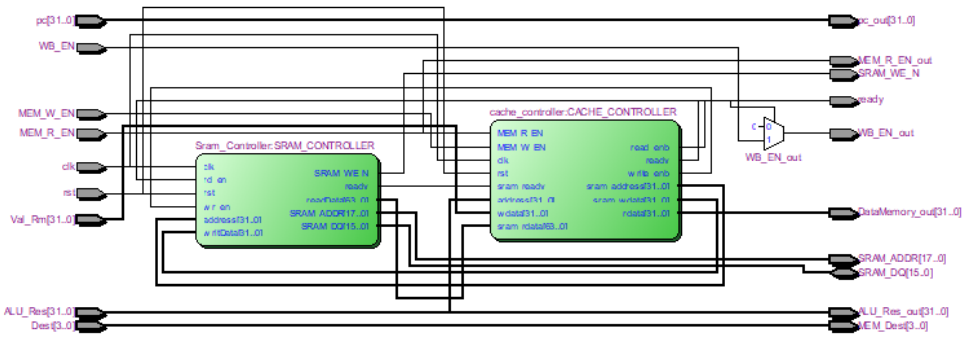
$RTL\ View$ کلی پردازنده:



$RTL\ View$ مربوط به $SRAM$:



$RTL\ View$ مربوط به $SRAM\ Controller$ و $Chace\ controller$:



حال بدین منظور ماژولی تحت عنوان $Cache\ Controlle$r به پردازنده اضافه می نماییم:
قطعه کد زیر ماژول $Cache\ Controller$ نشان می دهد:

```
`define WAY_1_DATA   150:87
`define WAY_1_TAG    86:77
`define WAY_1_VALID  76
`define WAY_1_BLK_0  118:87
`define WAY_1_BLK_1  150:119

`define WAY_0_DATA   75:12
`define WAY_0_TAG    11:2
`define WAY_0_VALID  1
`define WAY_0_BLK_0  43:12
`define WAY_0_BLK_1  75:44

`define LRU          0

module cache_controller
(
input clk,
input rst,
input MEM_R_EN,
input MEM_W_EN,
input [31:0] address,
input [31:0] wdata,
input [63:0]sram_rdata,
input sram_ready,
output [31:0] sram_address,
output [31:0] sram_wdata,
output reg write_enb,
output reg read_enb,
output [31:0] rdata,
output reg ready
);

reg [150:0] cache [0:63]; //[WAY_1_DATA(BLK_0+BLK_1),WAY_1_TAG,WAY_1_VALID]+[WAY_0_DATA(BLK_0+BLK_1),WAY_0_TAG,WAY_0_VALID]+[LRU]

// Extracting Address Elements
reg [16:0]cacheAddress;   //[tag,index,wordOffset]
reg       wordOffset;
reg [5:0] index;
reg [9:0] tag;
assign cacheAddress=((address-32'd1024)>>2);
assign wordOffset=address[0]; //word offset
assign index=address[6:1];    //index
assign tag=address[16:7];     //tag

// Hit/Miss
reg Hit_way_0;
reg Hit_way_1;
reg Miss;
assign Hit_way_0=(cache[index][`WAY_0_VALID]==1'b1)?(tag==cache[index][`WAY_0_TAG])?1'b1:1'b0 :1'b0;
assign Hit_way_1=(cache[index][`WAY_1_VALID]==1'b1)?(tag==cache[index][`WAY_1_TAG])?1'b1:1'b0 :1'b0;
assign Miss= ~Hit_way_0 & ~Hit_way_1;

// Reading From Cache: If the cache has the data, it will send it on rdata, otherwise, rdata will be drived by SRAM
assign rdata=(Hit_way_0==1'b1)?(wordOffset==1'b0)?cache[index][`WAY_0_BLK_0]:cache[index][`WAY_0_BLK_1]:
             (Hit_way_1==1'b1)?(wordOffset==1'b0)?cache[index][`WAY_1_BLK_0]:cache[index][`WAY_1_BLK_1]:
             (wordOffset==1'b0)?sram_rdata[31:0]:sram_rdata[63:32];

// Update LRU based on Last Use
always@(*)
begin
    if(Hit_way_0==1'b1)
        cache[index][`LRU]=1'b0;

    else if(Hit_way_1==1'b1)
        cache[index][`LRU]=1'b1;

    else
        cache[index][`LRU]=cache[index][`LRU];
end
```

```
// Total Ready Signal
always@(*)
begin
    if(MEM_W_EN==1'b1)
        ready=sram_ready;
    else if(MEM_R_EN==1'b1)
    begin
        if(Hit_way_0==1'b1 || Hit_way_1==1'b1)
            ready=1'b1;
        else
            ready=sram_ready;
    end

    else
        ready=1'b1;

end

// Set SRAM Signals
assign sram_address=address;
assign sram_wdata=wdata;

// These signals control "wr_en" and "rd_en" of the SRAM controller
always@(*)
begin
    read_enb=1'b0;
    write_enb=1'b0;

    if(MEM_R_EN==1'b1&&Miss==1'b1)
        read_enb=1'b1;
    else if(MEM_W_EN==1'b1)
        write_enb=1'b1;

end

integer i;
always@(posedge clk, posedge rst)
begin
    if(rst==1'b1)
    begin
        // Reset LRU, WAY_0_VALID, and WAY_1_VALID
        for(i=0; i<64; i=i+1)
        begin
            cache[i][`LRU]  = 1'b0;
            cache[i][`WAY_0_VALID]  = 1'b0;
            cache[i][`WAY_1_VALID] = 1'b0;
        end
    end

    //Update Cache by SRAM because of Miss
    else if(MEM_R_EN==1'b1 && Miss && sram_ready)
    begin
        if(cache[index][`LRU]==1'b0)
        begin
            cache[index][`WAY_1_VALID]=1'b1;
            cache[index][`WAY_1_TAG]=tag;
            cache[index][`WAY_1_DATA]=sram_rdata;
            cache[index][`LRU]=1'b1;
        end

        else //cache[index][LRU]==1'b1
        begin
            cache[index][`WAY_0_VALID]=1'b1;
            cache[index][`WAY_0_TAG]=tag;
            cache[index][`WAY_0_DATA]=sram_rdata;
            cache[index][`LRU]=1'b0;
        end
    end

   // In case of writing data into storge: first of all, data will be written into the SRAM, and meanwhile, if the address is exist in the Cache, it will be invalid
   //-because it won't be point to the new data @ SRAM
   else if(Hit_way_0==1'b1)
   begin
       if(MEM_W_EN==1'b1)
       begin
           cache[index][`WAY_0_VALID]=1'b0;
       end
   end

   else if(Hit_way_1==1'b1)
   begin
       if(MEM_W_EN==1'b1)
       begin
           cache[index][`WAY_1_VALID]=1'b0;
       end
   end

end

endmodule
```

حال به تفصیل هر بخش را توضیح می دهیم:

در ابتدا یکسری پارامتر تعریف می نماییم، که حاوی اعدادی است که به دفعات در کد مورد نظر استفاده می شود.
سپس ورودی ها و خروجی های ماژول تعریف می گردند.
کلاک و ریست برای همگام سازی این ماژول با باقی بخش های پردازنده.
MEM_R_EN و MEM_W_EN: سیگنال های ورودی برای مشخص کردن عملیات خواندن و یا نوشتن
$address$: سیگنال ورودی برای مشخص کردن آدرس محل خواندن یا نوشتن
$wdata$: سیگنال ورودی، مشخص کننده‌ی داده ای که می خواهیم در حافظه نوشته شود.
$sram\_rdata$: مقدار خوانده شده از $SRAM$
باید توجه داشت که مقدار این سیگنال ورودی برابر با ۶۴ بیت است که علت آن در این است که با هربار $miss$ اس که رخ می دهد به جای آن که تنها ۳۲ بیت داده از $SRAM$ بخوانیم، ۶۴ بیت می خوانیم. با اینکار می خواهیم احتمال رخ دادن $miss$ را کاهش دهیم علاوه بر اینکه داده‌ی موردنظر را داریم، داده های کناری آن را نیز داشته باشیم.
$sram\_ready$: سیگنالی که نشان می دهد اگر $miss$ رخ داد یا خواستیم در $SRAM$ بنویسیم، پردازنده تا چه زمانی باید $stall$ باشد.
$sram\_address$: آدرسی که مخواهیم داده را از آن بخوانیم یا در آن بنویسیم.
$sram\_wdata$: داده ای که می خواهیم در $SRAM$ بنویسیم.
$write\_enb$ و $read\_enb$: سیگنال های خروجی عملیات خواندن و نوشتن که میان $ControllerSRAM$ و $Cache\ Controller$ تبادل می شوند.
$rdata$: سیگنال خروجی $Cache\ Controller$ که داده‌ی حاصل از عملیات خواندن را مشخص می نماید.
$ready$: سیگنالی برای مشخص کردن اینکه ایا عملیات $Cache\ Controller$ به صورت کامل انجام شده است یا خیر.

```
`define WAY_1_DATA   150:87
`define WAY_1_TAG    86:77
`define WAY_1_VALID  76
`define WAY_1_BLK_0  118:87
`define WAY_1_BLK_1  150:119

`define WAY_0_DATA   75:12
`define WAY_0_TAG    11:2
`define WAY_0_VALID  1
`define WAY_0_BLK_0  43:12
`define WAY_0_BLK_1  75:44

`define LRU          0

module cache_controller
(
input clk,
input rst,
input MEM_R_EN,
input MEM_W_EN,
input [31:0] address,
input [31:0] wdata,
input [63:0]sram_rdata,
input sram_ready,
output [31:0] sram_address,
output [31:0] sram_wdata,
output reg write_enb,
output reg read_enb,
output [31:0] rdata,
output reg ready
);
```

در ادامه حافظه‌ی کش، به صورت ۱۵۱ خانه‌ی ۶۴ بیتی تعریف می گردد.

```
reg [150:0] cache [0:63]; //[WAY_1_DATA(BLK_0+BLK_1),WAY_1_TAG,WAY_1_VALID]+[WAY_0_DATA(BLK_0+BLK_1),WAY_0_TAG,WAY_0_VALID]+[LRU]
```

سپس $cacheAddress$ و $wordOffset$ و $index$ و tag براساس مطالب عنوان شده در صورت آزمایش تعیین می گردد.

```
// Extracting Address Elements
reg [16:0]cacheAddress;   //[tag,index,wordOffset]
reg       wordOffset;
reg [5:0] index;
reg [9:0] tag;
assign cacheAddress=((address-32'd1024)>>2);
assign wordOffset=address[0]; //word offset
assign index=address[6:1];    //index
assign tag=address[16:7];     //tag
```

درادامه طبق کد زیر، مشخص می شود که آیا hit رخ می دهد یا $miss$ و اگر hit رخ می دهد در کدامین way داده‌ی موردنظر وجود دارد:

```
// Hit/Miss
reg Hit_way_0;
reg Hit_way_1;
reg Miss;
assign Hit_way_0=(cache[index][`WAY_0_VALID]==1'b1)?(tag==cache[index][`WAY_0_TAG])?1'b1:1'b0 :1'b0;
assign Hit_way_1=(cache[index][`WAY_1_VALID]==1'b1)?(tag==cache[index][`WAY_1_TAG])?1'b1:1'b0 :1'b0;
assign Miss= ~Hit_way_0 & ~Hit_way_1;
```

سپس عملیات خواندن از کش مدیریت می شود:

```
// Reading From Cache: If the cache has the data, it will send it on rdata, otherwise, rdata will be drived by SRAM
assign rdata=(Hit_way_0==1'b1)?(wordOffset==1'b0)?cache[index][`WAY_0_BLK_0]:cache[index][`WAY_0_BLK_1]:
             (Hit_way_1==1'b1)?(wordOffset==1'b0)?cache[index][`WAY_1_BLK_0]:cache[index][`WAY_1_BLK_1]:
             (wordOffset==1'b0)?sram_rdata[31:0]:sram_rdata[63:32];
```

اگر داده در کش موجود بوده باشد و یا به عبارتی hit رخ داده باشد، وابسته به اینکه hit در کدام یک از بلاک ها رخ داده و $wordOffset$ چه می باشد، داده ی مورد نظر از کش خوانده می شود.
ولی اگر $miss$ رخ داده باشد داده ی موردنظر از $SRAM$ خوانده می شود(داده $rdata$ توسط سیگنال $sram\_rdata$ درایو می شود.)

با توجه به اینکه اگر $miss$ رخ دهد نیازمند جیاگزینی داده ها می باشیم، با توجه به سیاست جایگرینی LRU عمل می نماییم، وابسته به اینکه آخرین مرتبه در کدام یک از بلاک ها hit رخ داده است، سیگنال LRU آن را فعال و برای دیگری را غیر فعال می نماییم.

```
// Update LRU based on Last Use
always@(*)
begin
    if(Hit_way_0==1'b1)
        cache[index][`LRU]=1'b0;

    else if(Hit_way_1==1'b1)
        cache[index][`LRU]=1'b1;

    else
        cache[index][`LRU]=cache[index][`LRU];
end
```

قطعه کد زیر نحوه‌ی مدیریت سیگنال $ready$ را نشان می دهد:
وابسته نوع دسترسی ای که به حافظه داریم(خواندن یا نوشتن) سیگنال $ready$ را مشخص می نماییم.
اگر می خواهیم در حافظه بنویسیم به ۶ سیکل کلاک نیاز داریم، پس باید ۶ سیکل پردازنده را $stall$ کنیم.
اگر می خواهیم از حافظه بخوانیم، پس باید بررسی شود که آیا hit رخ داده است یا $miss$. اگر $miss$ رخ دهد نیز باید ۶ سیکل پردازنده $stall$ شود تا داده ی مورد نظر از $SRAM$ خوانده شود ولی اگر hit رخ دهد می توانیم داده موردنظر را در همان سیکل بخوانیم و نیازی با $stall$ نباشد.

```
// Total Ready Signal
always@(*)
begin
    if(MEM_W_EN==1'b1)
        ready=sram_ready;
    else if(MEM_R_EN==1'b1)
    begin
        if(Hit_way_0==1'b1 || Hit_way_1==1'b1)
            ready=1'b1;
        else
            ready=sram_ready;
    end

    else
        ready=1'b1;

end
```

در ادامه سیگنال های کنترلی $sram\_address$ و $sram_wdata$و $read_enb$و $write\_enb$ که میان $Cahce\ Controller$ و $SRAM\ Controller$ در زمان هایی که $miss$ رخ می دهد یا می خواهیم در حافظه عملیات نوشتن را انجام دهیم، تبادل می شوند را مدیریت می نماییم.

```
// Set SRAM Signals
assign sram_address=address;
assign sram_wdata=wdata;

// These signals control "wr_en" and "rd_en" of the SRAM controller
always@(*)
begin
    read_enb=1'b0;
    write_enb=1'b0;

    if(MEM_R_EN==1'b1&&Miss==1'b1)
        read_enb=1'b1;
    else if(MEM_W_EN==1'b1)
        write_enb=1'b1;

end
```

درقطعه کد زیر اگر سیگنال ریست آمده باشد، مقادیر موجود در کش را پاک می نماییم (تمامی موارد ۰ می شوند.)
سپس بررسی می شود، اگر $miss$ رخ داده باشد، مقدار موردنظر از $SRAM$ خوانده می شود و سپس کش $update$ می شود.

```
integer i;
always@(posedge clk, posedge rst)
begin
    if(rst==1'b1)
    begin
        // Reset LRU, WAY_0_VALID, and WAY_1_VALID
        for(i=0; i<64; i=i+1)
        begin
            cache[i][`LRU]  = 1'b0;
            cache[i][`WAY_0_VALID]  = 1'b0;
            cache[i][`WAY_1_VALID] = 1'b0;
        end
    end

    //Update Cache by SRAM because of Miss
    else if(MEM_R_EN==1'b1 && Miss && sram_ready)
    begin
        if(cache[index][`LRU]==1'b0)
        begin
            cache[index][`WAY_1_VALID]=1'b1;
            cache[index][`WAY_1_TAG]=tag;
            cache[index][`WAY_1_DATA]=sram_rdata;
            cache[index][`LRU]=1'b1;
        end

        else //cache[index][LRU]==1'b1
        begin
            cache[index][`WAY_0_VALID]=1'b1;
            cache[index][`WAY_0_TAG]=tag;
            cache[index][`WAY_0_DATA]=sram_rdata;
            cache[index][`LRU]=1'b0;
        end
    end
```

در ادامه قطعه کد زیر عملیات نوشتن در حافظه را مدیریت می نماید:
اگر دسترسی به حافظه از نوع نوشتن باشد، ابتدا داده‌ی مورد نظر در $SRAM$ نوشته می شود و سپس اگر آدرس موردنظر درون کش موجود بود، داده‌ی آن پاک می شود زیر دیگر حاوی مقداری درستی نخواهد بود.

```
    // In case of writing data into storge: first of all, data will be written into the SRAM, and meanwhile, if the address is exist in the Cache, it will be invalid
    //-because it won't be point to the new data @ SRAM
    else if(Hit_way_0==1'b1)
    begin
        if(MEM_W_EN==1'b1)
        begin
            cache[index][`WAY_0_VALID]=1'b0;
        end
    end

    else if(Hit_way_1==1'b1)
    begin
        if(MEM_W_EN==1'b1)
        begin
            cache[index][`WAY_1_VALID]=1'b0;
        end
    end

end

endmodule
```

همانطور که پیشتر نیز ذکر گردید، هنگامی که $miss$ رخ می دهد، $Cache\ controller$ ۶۴ بیت داده‌ی از $SRAM$ می خواند، ولیکن با توجه به $SRAM\ Controller$ طراحی شده، تنها می توان ۳۲ بیت داده از $SRAM$ خواند. بنابراین نیازمند آن هستیم که SRAM Controller را تغییر دهیم تا بتوانیم ۶۴ بیت داده از $SRAM$ بخوانیم:

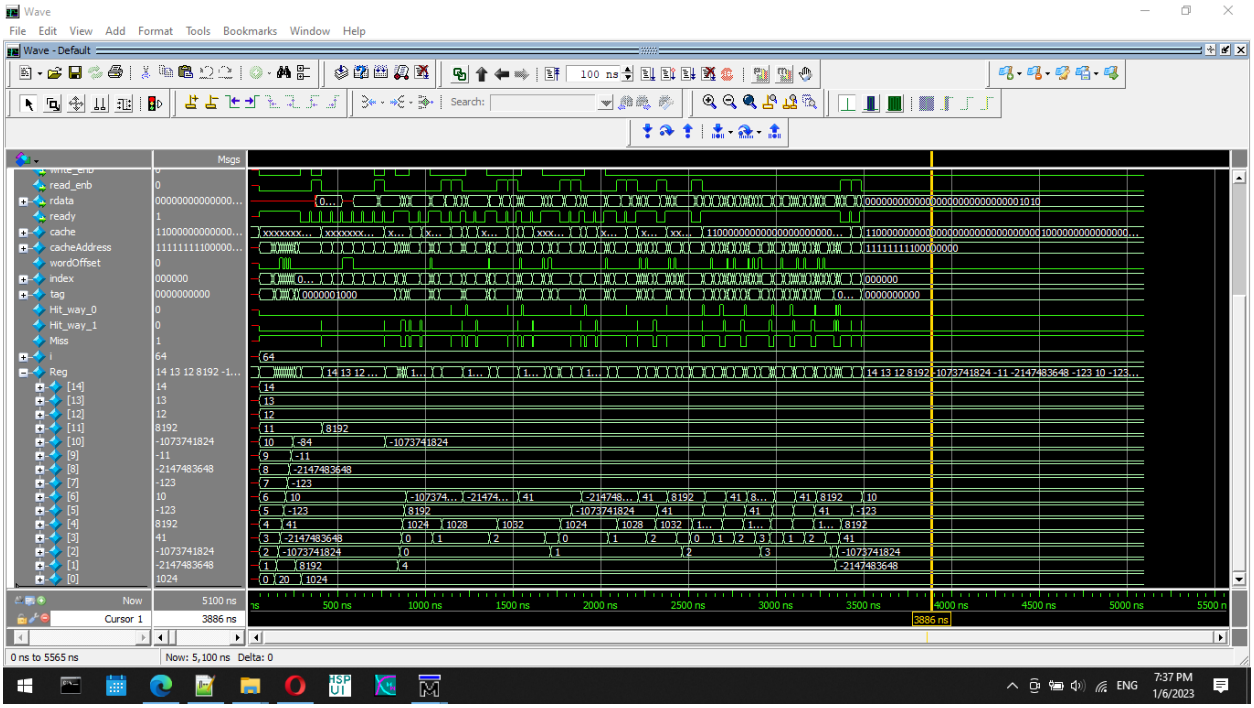
بدین منظور عملیات خواندن را در $stage$ های ۴ و ۵ از $SRAM\ Controller$ ادامه می دهیم. پس در نهایت در $stage$ های ۲ و ۳ و ۴ و ۵ عملیات خواندن انجام می شود و در هر مرتبه ۱۶ بیت خوانده می شود که مجموعا ۶۴ بیت داده از $SRAM$ خواند می شود:

```
always@(ps, wr_en, rd_en)
begin
    //initial vlaue
    ready=1'b0;
    SRAM_WE_N=1'b1; // deactive high

    case(ps)
        S_0:
        begin
        if(wr_en | rd_en)
            ready=1'b0;
        else
            ready=1'b1;
        end

        S_1:
        begin
            if(wr_en==1'b1)
            begin
                SRAM_ADDR=(address-32'd1024)>>1; //low address
                SRAM_WE_N=1'b0; // active low
                ready=1'b0;
            end
            else // rd_en==1'b1
            begin
                SRAM_ADDR=(address-32'd1024)>>1; //low address
                SRAM_WE_N=1'b1; // deactive high
                ready=1'b0;
            end
        end

        S_2:
        begin
            if(wr_en==1'b1)
            begin
                SRAM_ADDR=((address-32'd1024)>>1)+3'd1; //high address=low address+1
                SRAM_WE_N=1'b0; // active low
                ready=1'b0;
            end
            else // rd_en==1'b1
            begin
                SRAM_ADDR=((address-32'd1024)>>1)+3'd1; //high address=low address+1
                SRAM_WE_N=1'b1; // deactive high
                readData[15:0]=SRAM_DQ;
                ready=1'b0;
             end
        end

        S_3:
        begin
            if(rd_en==1'b1)
            begin
                SRAM_ADDR=((address-32'd1024)>>1)+3'd2; //high address=low address+1
                SRAM_WE_N=1'b1; // deactive high
                readData[31:16]=SRAM_DQ;
            end
            ready=1'b0;
        end

        S_4:
        begin
            if(rd_en==1'b1)
            begin
                SRAM_ADDR=((address-32'd1024)>>1)+3'd3; //high address=low address+1
                SRAM_WE_N=1'b1; // deactive high
                  SRAM_WE_N=1'b1; // deactive high
                  readData[47:32]=SRAM_DQ;
              end
              ready=1'b0;
          end

          S_5:
          begin
              if(rd_en==1'b1)
                  readData[63:48]=SRAM_DQ;

              ready=1'b1;
          end

    endcase
end

assign SRAM_DQ=(SRAM_WE_N==1'b0 && ps==S_1)?writData[15:0]:// write low data
               (SRAM_WE_N==1'b0 && ps==S_2)?writData[31:16]:// write high data
                16'dz;

endmodule
```

حال باید ماژول *Chache Controller* و *SRAM Controller* را در *MEM stage* به یکدیگر متصل نماییم:

```
module MEM
(
input clk,
input rst,
input WB_EN,
input MEM_R_EN,
input MEM_W_EN,
input [31:0] ALU_Res,
input [31:0] Val_Rm,
input [3:0] Dest,
input [31:0] pc,
output WB_EN_out,
output MEM_R_EN_out,
output [31:0]ALU_Res_out,
output [31:0]DataMemory_out,
output [3:0] MEM_Dest,
output [31:0] pc_out,
// Sram Controller
output ready,
inout  [15:0] SRAM_DQ,
output [17:0] SRAM_ADDR,
output SRAM_WE_N
);

wire sram_ready;
wire [63:0] sram_rdata;
wire [31:0] sram_address;
wire write_enb;
wire read_enb;
wire [31:0] sram_wdata;

assign pc_out = pc;
assign WB_EN_out=(ready==1'b1)?WB_EN:1'b0; // freesing WB_EN_out based on ready signal of SRAM controller
assign MEM_R_EN_out = MEM_R_EN;
assign MEM_Dest = Dest;
assign ALU_Res_out = ALU_Res;
cache_controller CACHE_CONTROLLER(
.clk(clk),
.rst(rst),
.MEM_R_EN(MEM_R_EN),
.MEM_W_EN(MEM_W_EN),
.address(ALU_Res),
.wdata(Val_Rm),
.sram_rdata(sram_rdata),
.sram_ready(sram_ready),
.sram_address(sram_address),
.sram_wdata(sram_wdata),
.write_enb(write_enb),
.read_enb(read_enb),
.rdata(DataMemory_out),
.ready(ready)
);

Sram_Controller SRAM_CONTROLLER(
.clk(clk),
.rst(rst),
.wr_en(write_enb),
.rd_en(read_enb),
.address(sram_address),
.writData(sram_wdata),
.readData(sram_rdata),
.ready(sram_ready),
.SRAM_DQ(SRAM_DQ),
.SRAM_ADDR(SRAM_ADDR),
.SRAM_WE_N(SRAM_WE_N)
);

endmodule
```

اجرای برنامه‌ی محک و بررسی صحت عملکرد پردازنده:



همانطور که از تصویر بالا مشخص است، خروجی برنامه محک مطابق با انتظار، صحیح می باشد.
به میزان $3500ns$ طول کشیده است تا تمامی خروجی های برنامه‌ی محک آماده شوند؛ این درحالی است که مرحله‌ی قبل، یعنی پردازنده به همراه $SRAM$ ، پس از $4395ns$ خروجی آماده می شود.
این یعنی پردازنده به همراه $cache$ به میزان $4395ns - 3500ns = 895\ ns$ نسبت به پردازنده به همراه $SRAM$ بهبود داشته است.

نتایج سنتز پردازنده طراحی شده تا بدین مرحله:

Flow Status	Successful - Sat Feb 04 21:48:39 2023
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 II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	0 / 33,216 (0 %)
Total combinational functions	0 / 33,216 (0 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	2 / 475 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

همناطور که از نتایج سنتز مشخص است، تعداد المان مورد استفاده توسط پردازنده طراحی شده (تا بدین جا)، افزایش یافته است که البته مطابق با انتظار می باشد. منطقا با اضفه شده $cache$ به پردازنده طراحی شده در مرحله‌ی قبل، تعداد المان های مورد استفاده افزایش می یابد.

بخش امتیازی:

در مکانیزم $write\ through$ هنگامی که داده ای به دفعات تغییر می کند، کارای پردازنده افت کاهش می یابد زیرا زمان زیادی را برای هر تغییر مقدار دورن $SRAM$ را نیز تغییر می دهد. ولی اگر از مکانیزم $write\ back$ استفاده نماییم، حتی اگر داده ای به دفعات نیز تغییر کند این تغییرات فقط در کش ثبت می شود و فقط هنگامی که قرار است از کش حذف شود، $SRAM$ را آپدیت می نماید. در این حالت به علت سرعت بالای کش کارایی پردازنده کاهش کمتری نسبت به حالت $write\ through$ خواهد داشت.

راه حل دیگر به منظور افزایش کارایی پردازنده، استفاده از چندین کش به صورت موازی است، بدین صورت که داده ی مورد نظر در چندین کش جست و جو شود. بدین ترتیب می توانیم داده های زیادی را در کش ذخیره نماییم و از آن جایی که کش ها به صورت موازی عملیات را انجام می دهند، سرعت
جست و جو و احتمال hit شده افزایش می یابد.

راه حل دیگر افزایش ساز کش می باشد، مثلا از کشی با تعداد way بالاتر استفاده نمود.