به نام خدا

آزمایشگاه معماری کامپیوتر (CALab)

گزارش جلسه هفتم و هشتم

ایمان رسولی پرتو 810199425 & پارسا حداد منفرد 810198380

SRAM

← مقدمه

در سیستم های بزرگ معمولا حافظه در پردازنده قرار نمیگیره و دارای یک module جداگانه هست. چرا که حافظه های بزرگی در سیستم وجود داره و فقط با پردازنده هم ارتباط نداره (ارتباط با بقیه module ها نظیر accelerator ها) بنابراین یک مکانیزم handshaking بین CPU و Memory خارج از پردازنده وجود داره که این ارتباط رو برقرار کنه.

SRAM Controller ←

با توجه به مقدمه فوق memory موجود در بخش Mem stage حذف میشه و برای ارتباط با SRAM module بورد یک SRAM controller در این stage قرار میگیره. این کنترلر وظیفه گرفتن data از SRAM به هنگام خواندن از حافظه و نوشتن data به هنگام نوشتن در حافظه رو داره؛ با توجه به اینکه در عمل حافظه‌ها کند هستند، این کنترلر وظیفه متوقف کردن pipe تا فراهم شدن data ارسالی توسط SRAM هم داره؛ برای مدل کردن این اتفاق و همچنین مقایسه با حالتی که از cache استفاده میکنیم، فرض میکنیم پس از 6 سیکل SRAM کارش رو (چه نوشتن و چه خوندن) انجام میده. با توجه به اینکه خانه های SRAM 16 بیتی هستند اما داده‌های arm 32 بیتی، از مکانیزم burst برای انتقال داده بین حافظه SRAM و پردازنده استفاده میکنیم.

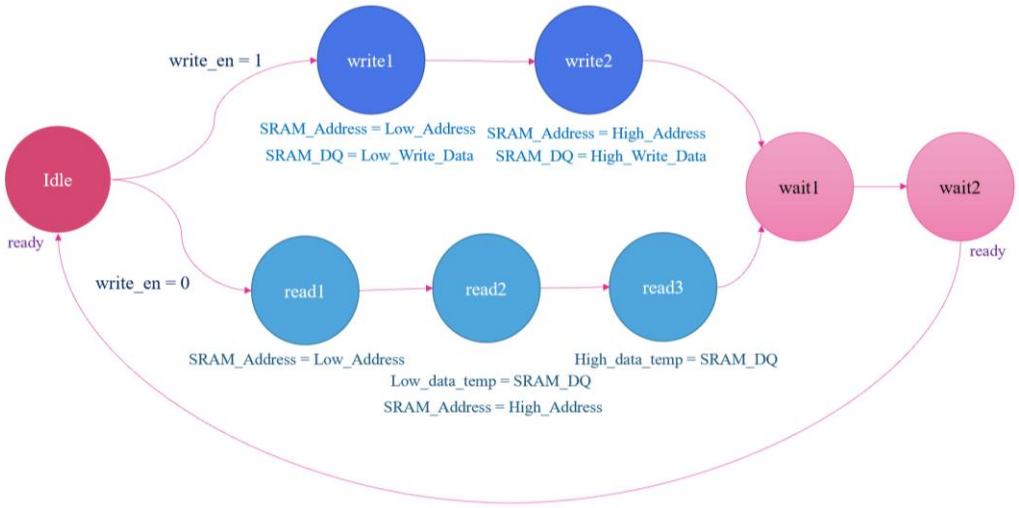

Fig1. SRAM controller state machine

با توجه به شکل بالا، برای نوشتن در حافظه، ابتدا 16 بیت کم ارزش آدرس رو به SRAM میدیم همچنین 16 بیت کم ارزش داده‌ای که قراره توی حافظه نوشته بشه؛ سپس سیگنال "*SRAM_W_EN*" رو 0 میکنیم (active low). در سیکل بعدی همین کار رو برای بیت های پرارزش انجام میدیم. سپس دو سیکل صبر میکنیم، سیگنال ready، active میشه و به Idle state میریم.

برای خواندن از حافظه مشابه نوشتن عمل میکنیم، تفاوت اینه که ابتدا چون SRAM داریم یعنی فقط یک port داریم و از این پورت هم برای خوندن و هم برای نوشتن استفاده میشه (Bidirectional bus) این پورت از طرف کنترلر باید z بشه تا از طرف SRAM روی اون دیتا قرار بگیره؛ تفاوت بعدی اینه که وقتی در read1 state سیگنال read رو فعال کنیم، بخاطر مسائل timing دیتای SRAM در سیکل بعدی آماده خواهد بود. در نتیجه مشابه نوشتن دو سیکل صبر کرده و به Idle state میریم.

```
module SRAAAM_Controller (clk, rst, writeEn, readEn, address, WriteData, ReadData, ready, SRAM_DQ, SRAM_ADDR,
                          SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N);
    input clk, rst;
    input writeEn, readEn;
    input [31:0] address, WriteData;

    output [31:0] ReadData;

    output ready;
    inout [15:0] SRAM_DQ;
    output reg [17:0] SRAM_ADDR;

    output  SRAM_UB_N;          //SRAM high byte Data Mask
    output  SRAM_LB_N;          // SRAM low byte Data Mask
    output reg SRAM_WE_N;          // SRAM write Enable
    output  SRAM_CE_N;          // SRAM chip Enable
    output  SRAM_OE_N;          // SRAM output ENable

    parameter [3:0] idle = 0, write1 = 1, write2 = 2, read1 = 3, read2 = 4, read3 = 5, wait1 = 6, wait2 = 7;

    reg [3:0] ns, ps;
    reg [15:0] readLSB, readMSB;

    assign {SRAM_UB_N, SRAM_LB_N, SRAM_CE_N, SRAM_OE_N} = 4'b0000;

    always @* begin : next_state
        ns = idle;
        case (ps)
            idle: begin
                if (~writeEn & ~readEn)
                    ns = idle;
                else if (writeEn)
                    ns = write1;
                else
                    ns = read1;
            end
            write1: ns = write2;
            write2: ns = wait1;
            read1: ns = read2;
            read2: ns = read3;
            read3: ns = wait1;
            wait1: ns = wait2;
            wait2: ns = idle;
        endcase
    end

    always @(ps) begin : signals
        SRAM_ADDR = 18'b0; SRAM_WE_N = 1'b1;
        case (ps)
            write1: begin
                SRAM_ADDR = {address[18:2], 1'b0};
                SRAM_WE_N = 1'b0;
            end
            write2: begin
                SRAM_ADDR = {address[18:2], 1'b1};
                SRAM_WE_N = 1'b0;
            end
            read1: begin
                SRAM_ADDR = {address[18:2], 1'b0};
            end
            read2: begin
                SRAM_ADDR = {address[18:2], 1'b1};
                readLSB = SRAM_DQ;
            end
            read3: begin
                readMSB = SRAM_DQ;
            end
        endcase
    end

    always @(posedge clk, posedge rst) begin
        if (rst)
            ps <= idle;
        else
            ps <= ns;
    end

    assign SRAM_DQ = (ps == write1) ? WriteData[15:0] :
                     (ps == write2) ? WriteData[31:16] : 16'bz;

    assign ready = (ns == idle) ? 1'b1 : 1'b0;

    assign ReadData = (ps == wait2) ? {readMSB, readLSB}:32'bz;


endmodule
```

Fig2. SRAM Controller Verilog description

با نوشتن یک SRAM برای تست کردن برنامه درحالتی که forwarding فعال باشه نتیجه simulation به شکل زیر خواهد بود:

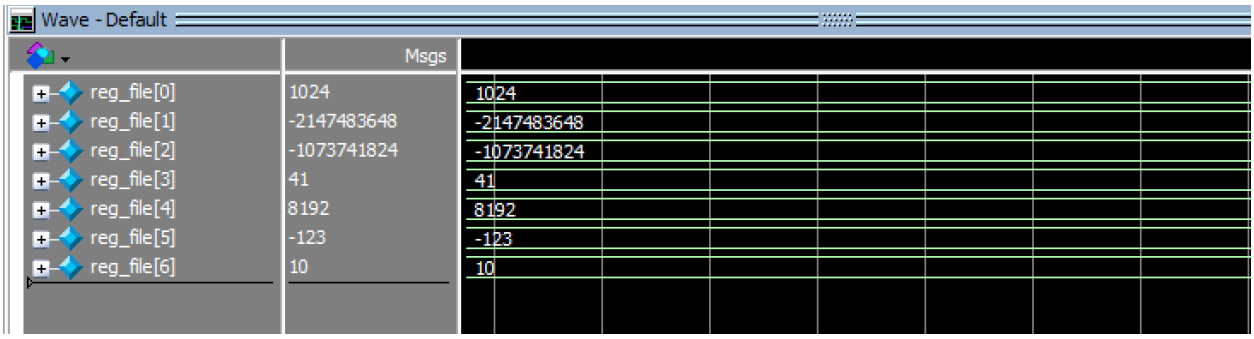

Fig3. Arm output – with SRAM & forwarding

حالا از SRAM خود بورد استفاده میکنیم و نتایج رو در signal tab مشاهده میکنیم:

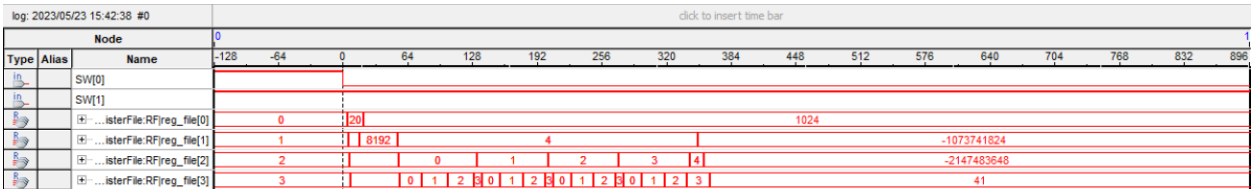

Fig4. Arm output on signal tab

Flow Summary	
Flow Status	Successful - Tue May 23 15:42:28 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	5,193 / 33,216 (16 %)
Total combinational functions	3,441 / 33,216 (10 %)
Dedicated logic registers	3,429 / 33,216 (10 %)
Total registers	3429
Total pins	418 / 475 (88 %)
Total virtual pins	0
Total memory bits	134,144 / 483,840 (28 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Fig5. Compilation report

با توجه به دو شکل بالا، مشاهده میکنیم زمان اجرای برنامه به دلیل استفاده از حافظه خارجی افزایش پیدا کرده، همچنین منابع مصرفی بخاطر اضافه شدن SRAM controller افزایش پیدا کرده. این زمان به نسبت بخش forwarding و بخش بدون forwarding کندتر شده، چرا که کار SRAM 6 سیکل طول میکشه؛ دلیل بعدی وجود مکانیزم burst هست که باعث کند شدن برنامه و handshaking با حافظه میشه.

بنابراین وجود حافظه خارجی میتونه باعث کند شدن روند اجرای برنامه بشه اما به دلایل معقولی در عمل هم همین موضوع وجود داره و حافظه ها خارج از module های سخت افزاری هستند؛ اما راه های افزایش سرعت متنوعی برای غلبه بر این سربار حافظه وجود داره که در بخش بعدی با یکی از اونها یعنی مکانیزم cache آشنا میشیم.