دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده مهندسی کامپیوتر

گزارش پایانی پروژه درس VLSI پیشرفته

طراحی و شبیه‌سازی رجیسترفایل ۱۲۸ کلمه‌ای ۳۲ بیتی

نگارش

رضا آدینه پور

استاد درس

جناب آقای دکتر صدیقی

بهمن ۱۴۰۲

چکیده

حافظه‌ها بخش عمده ای از مساحت و ترانزیستورهای مصرفی یک تراشه را تشکیل می‌دهند. حافظه‌ها عمدتا به سه دسته کلی تقسیم می‌شوند. RAM[۱] ها، SAM[۲] و CAM[۳] سه دسته کلی حافظه‌ها هستند. هر کدام از این ۳ دسته نیز به زیربخش های دیگری تقسیم مشوند که در فصل ۱ به آنها خواهیم پرداخت. هدف و تمرکز این پروژه بر روی خانواده RAM ها و به ویژه SRAM[۴] است. در فاز اول پروژه ابتدا مدل RTL[۵] یک فایل حافظه[۶] با ابعاد ۱۲۸ کلمه[۷] ۳۲ بیتی را شبیه سازی کردیم و در فاز دوم همان مدل شبیه‌سازی شده در فاز اول را این بار در سطح ترانزیستور شبیه‌سازی کرده ایم.

کلیدواژه‌ها: حافظه، مدل رفتاری، فایل حافظه

---

[۱] Random Access Memory
[۲] Serial Access Memory
[۳] Content Addressable Memory
[۴] Static RAM
[۵] Register Transfer Level
[۶] Register file
[۷] Word

فهرست مطالب

۱ مقدمه ۱

۱-۱ تعریف مسئله ۱

۱-۲ مراحل انجام پروژه ۲

۱-۲-۱ فاز اول ۲

۱-۲-۲ فاز دوم ۳

۲ مفاهیم اولیه ۴

۲-۱ پورت‌های ورودی/خروجی ۴

۲-۱-۱ SRAM ۴

۲-۱-۲ سلول SRAM ۵

۳ طراحی ارائه شده در فاز اول ۸

۳-۱ مدل Behavioral ۸

۳-۱-۱ رفتار مدل ۹

۳-۱-۲ شبیه‌سازی مدل ۱۰

۳-۲ مدل RTL ۱۳

۳-۲-۱ رفتار مدل ۱۴

۳-۲-۲ شبیه‌سازی مدل ۱۹

۴ طراحی ارائه شده در فاز دوم ۲۱
۴-۱ سلول SRAM ۲۱
۴-۱-۱ حالت خواندن ۲۲
۴-۱-۲ حالت نوشتن ۲۲
۴-۲ سایز ترانزیستورها ۲۳
۴-۳ ساختار طراحی ۲۳
۴-۴ تعریف سلول SRAM در HSPICE ۲۴
۵ مقایسه مدل ها و نتیجه گیری ۲۷
مراجع ۲۹

فهرست جداول

فهرست تصاویر

۱-۱ ساختار کلی حافظه‌ها ۲

۱-۲ معماری یک حافظه ۵

۲-۲ سلول SRAM ۶ ترانزیستوری ۶

۱-۳ بلوک‌دیاگرام مدل رفتاری حافظه ۸

۲-۳ حالت ریست حافظه ۱۱

۳-۳ تست حافظه با مقادیر دلخواه ۱۲

۴-۳ مدار ساخته شده توسط ابزار سنتز ۱۳

۵-۳ خروجی ریپورت ابزار سنتز ۱۳

۶-۳ خروجی ریپورت ابزار سنتز ۱۴

۷-۳ تکنولوژی شماتیک طراحی RTL ۱۸

۸-۳ مدار درونی طراحی شده ۱۸

۹-۳ پارامترهای زمانی گزارش شده برای حالت Balanced ۱۹

۱۰-۳ پارامترهای زمانی گزارش شده برای حالت Timing performance ۱۹

۱۱-۳ خروجی شبیه سازی طراحی RTL ۲۰

۱-۴ مدار ثبات نوع D [لینک] ۲۱

۲-۴ سلول ۶ ترانزیستوری SRAM ۲۲

۳-۴ نمودار حالت خواندن برای سلول ۶ ترانزیستوری SRAM ۲۳

۴-۴ نمودار حالت نوشتن برای سلول ۶ ترانزیستوری SRAM ۲۳
۴-۵ شمای جدید سلول SRAM بر حسب گیت Inverter ۲۴
۴-۶ نمودار پروانه ای ۲۴
۴-۷ شماتیک طراحی انجام شده ۲۵
۴-۸ خروجی شبیه‌سازی ۲۶

۵-۱ پارامترهای زمانی گزارش شده برای حالت Timing performance در طراحی Behavioral ۲۷

فصل ۱

مقدمه

همانطور که در قسمت چکیده گفته شد، حافظه‌ها انواع مختلفی دارند که به صورت کلی به سه دسته RAM و SAM و CAM تقسیم می‌شوند. به دلیل آنکه در این پروژه یکی از زیر مجموعه‌های خانواده RAM طراحی و شبیه سازی شده است، عمده توضیحات ما بر روی این خانواده خواهد بود.

RAM ها به دو دسته حافظه‌های فرار[۱] و غیر فرار[۲] تقسیم می‌شوند. فرار بدین معنی است که با قطع ولتاژ تغذیه حافظه، محتوای آن پاک خواهد شد.

حافظه‌های فرار به دو زیربخش حافظه‌های پویا[۳] و ایستا[۴] تقسیم می‌شوند که در این پروژه حافظه ایستا را شبیه‌سازی کرده ایم.

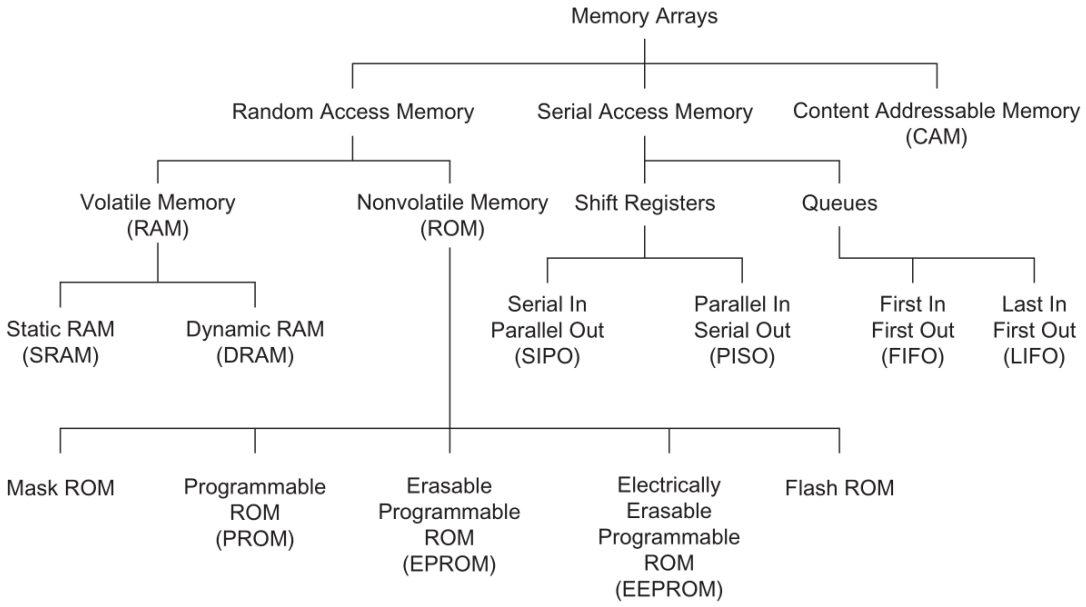
در شکل «۱-۱» ساختار سلسله‌مراتبی حافظه‌ها آورده شده است.

۱-۱ تعریف مسئله

در این پروژه هدف طراحی و شبیه‌سازی Register file ای با اندازه ۱۲۸ کلمه ۳۲ بیت است. هدف از انجام این پروژه آشنایی و انواع حافظه‌ها و نحوه شبیه‌سازی و پیاده‌سازی آنهاست. حافظه‌ها معمولا از دو بخش تشکیل می‌شوند:

---

[۱] Volatile
[۲] Non-volatile
[۳] Dynamic
[۴] Static



شکل ۱-۱: ساختار کلی حافظه‌ها

- بخش حافظه
- بخش سخت‌افزار

بخش حافظه مبتنی‌ست بر تکرار یک طراحی مشخص از یک سلول[۵] حافظه با چینشی مشخص. بخش سخت افزار آن متشکل است از دیکدر[۶] آدرس و مالتی‌پلکسر[۷] داده ها.

۱-۲ مراحل انجام پروژه

برای اطمینان از انجام پروژه و مقایسه بین مدل RTL و مدل سطح ترانزیستوری، این پروژه به دو بخش تقسیم شده است.

۱-۲-۱ فاز اول

در فاز اول با استفاده از زبان توصیف سخت‌افزار[۸] VHDL کد RTL حافظه SRAM نوشته و شبیه‌سازی شده است.

---

[۵] Cell
[۶] Decoder
[۷] Multiplaxer
[۸] Hardware describtion language

برای مقایسه بین دو مدل RTL و رفتاری[۹] حافظه، یک کد مجزا هم برای شبیه‌سازی مدل رفتاری نوشته شده است.

۱-۲-۲ فاز دوم

در فاز دوم پروژه مقیاس طراحی را به سطح ترانزیستور می‌آوریم و به کمک نرم‌افزار HSpice مدل شبیه‌سازی شده در فاز اول را در سطح ترانزیستور شبیه‌سازی می‌کنیم.

برای طراحی این فاز، از نرم‌افزار HSPICE ۲۰۱۰ و سلول ۶ ترانزیستوری SRAM استفاده شده است. که در فصل‌های بعد به بررسی طراحی و خروجی شبیه‌سازی شده و درنهایت به مقایسه مدل طراحی شده در سطح ترانزیستور و مدل RTL فاز یک می‌پردازیم.

[۹] Behavioral

فصل ۲

مفاهیم اولیه

در این فصل به بررسی جزئی تر ساختار حافظه SRAM خواهیم پرداخت و پورت‌های ورودی/خروجی آن و نحوه مشخص کردن بیت‌های آدرس و دیتا را بررسی خواهیم کرد.

۲-۱ پورت‌های ورودی/خروجی

سلول‌های حافظه می‌توانند یک یا چند پورت برای دسترسی داشته باشند. در حافظه‌های خواندنی/نوشتنی هر پورت می‌تواند فقط خواندنی[۱] و یا فقط نوشتنی[۲] و یا دو جهته[۳] باشد.

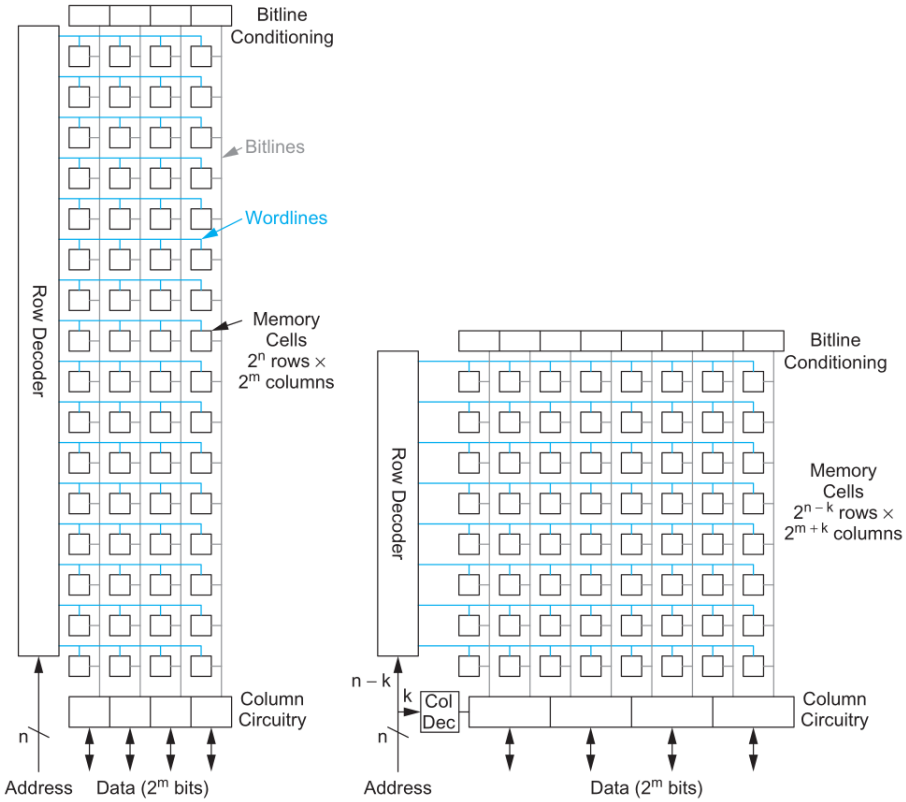
یک حافظه، دارای ۲^n کلمه ۲^m بیتی است که هر بیت در یک سلول حافظه ذخیره می‌شود. شکل «۲-۱» دو نمونه پیاده‌سازی مختلف از یک حافظه ۱۶ کلمه ای ۴ بیتی ($n = ۴, m = ۲$) را نشان می‌دهد.

۲-۱-۱ SRAM

حافظه ایستا[۴] از یک سلول حافظه با با فیدبک[۵] داخلی استفاده می‌کنند که تا زمانی که تغذیه آن متصل است مقدار خود را حفظ می‌کند. SRAM ها دارای خاصیت‌های زیر هستند:

- متراکم تر از فلیپ‌فلاپ ها[۶] هستند

[۱] Read only
[۲] Write only
[۳] Bidirectional
[۴] Static RAM
[۵] Feedback
[۶] Flip flop



شکل ۲-۱: معماری یک حافظه

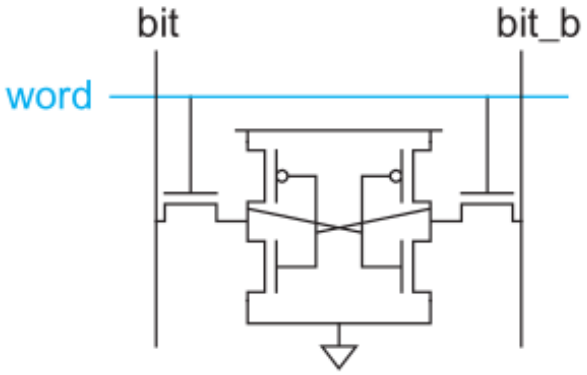
- با تکنولوژی ساخت CMOS استاندارد سازگار هستند
- سریع‌تر از DRAM ها هستند
- استفاده از آنها راحت تر از DRAM هاست

۲-۱-۲ سلول SRAM

یک سلول SRAM باید توان خواندن و نوشتن داده‌ها را داشته باشد و تا زمانی که تغذیه متصل است آنها را بدون تغییر نگه دارد. یک فلیپ‌فلاپ معمولی می‌تواند این نیاز را براورده کند اما اندازه آن بسیار بزرگ است. برای آنکه مشکل بزرگ بودن مساحت فلیپ‌فلاپ برطرف شود، سلول SRAM ای متشکل از ۶ ترانزیستور CMOS به صورت شکل «۲-۲» پیشنهاد شده است.

در مسائل مهندسی همیشه پس آنکه مشکلی را رفع کردیم باید به دنبال آن باشیم که کجا هزینه دادیم. هزینه ای که در کوچک کردن مساحت سلول دادیم آن است که مدارهای جانبی برای نوشتن و خواندن از حافظه را پیچیده کرده ایم. اما در مجموع این یک مبادله[۷] خوب است. چرا که در حافظه هایی با حجم بالا، مساحت

[۷] Trade off



شکل ۲-۲: سلول SRAM ۶ ترانزیستوری

بسیار زیادی از تراشه را، بخش حافظه اشغال می‌کنند و کاهش مساحت حافظه حتی با پذیرفتن پیچیده شدن طراحی مدارهای واسط آن برای ما مطلوب است.

از طرفی هرچقدر اندازه سلول کوچک‌تر باشد، مسیر های[۸] کوتاه تری برای ارتباط با سلول‌های مجاور نیاز است که در نتیجه آن کاهش توان دینامیکی را به دنبال دارد.

سلول ۶ ترانزیستوری SRAM دارای یک جفت معکوس‌کننده[۹] مقابل به هم است که حالت را نگه می‌دارند و یک جفت ترانزیستور که برای مشخص کردن وضعیت خواندن[۱۰] و یا نوشتن[۱۱] است.

فیدبک مثبت در این طراحی با اصلاح اختلالاتی که ناشی از نویز ممکن است ایجاد شود، اثر خود را ایجاد می‌کند.

برای نوشتن داده جدید در این سلول کافیست مقدار داده خود و مکمل[۱۲] آن را بر روی bit و bit_b بگذاریم و خط word را که به گیت ترانزیستور های هدایت کننده متصل است را ۱ کنیم. مقدار جدید جایگزین مقدار قبلی شده و تا زمانی که مجدد آن را تغییر ندهیم بر روی سلول ثبت[۱۳] می‌شود.

برای خواندن از سلول هم دقیقا کاری مشابه با نوشتن را باید انجام دهیم، یعنی نیاز است که یک دفعه خط word را ۱ کنیم. با این تفاوت که مقدار جدیدی را روی خط‌های bit و bit_b نمیگذاریم. و پس از یک کردن word مقدار موجود بر روی حافظه به bit داده می‌شود.

یکی دیگر از چالش‌های طراحی سلول SRAM در کنار کوچک کردن اندازه آن، این است که مداری که حالت را نگه می‌دارد، باید به اندازه کافی ضعیف باشد که در برابر تغییر حالت به هنگام write مشکلی ایجاد

[۸] Interconnect
[۹] Inverter
[۱۰] Read
[۱۱] Write
[۱۲] Complement
[۱۳] Register

نشود. از طرفی باید آنقدر قوی باشد که به هنگام خواندن، مقدار مطلوب[۱۴] به ما بدهد.

در فاز دوم این پروژه با چینش مناسب این سلول در کنار هم برای رجیسترفایلی ۱۲۸ × ۳۲ را طراحی می‌کنیم.

[۱۴] Valid

فصل ۳

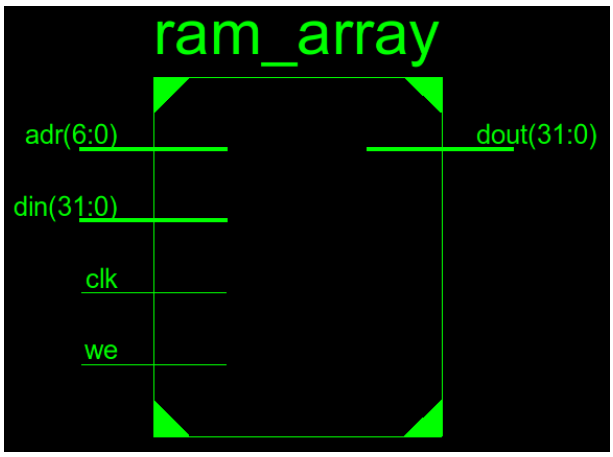
طراحی ارائه شده در فاز اول

در این فصل قدم به قدم به بررسی ۲ طراحی انجام شده، یعنی مدل RTL و Behavioral و بررسی کد های نوشته شده، بررسی عملکرد شبیه‌سازی و مقایه ساختاری هر دو طرح می‌پردازیم.

۳-۱ مدل Behavioral

در شبیه‌سازی مدل Behavioral و RTL حافظه، صرفا نحوه عملکرد حافظه، یعنی نوشتن در حافظه و خواندن از آن مهم است و در این فاز از پروژه SRAM و سایر حافظه‌ها برای ما یکسان هستند و صرفا رفتار آنها مهم است.

برای شبیه‌سازی مدل Behavioral از کد موجود «پیوست آ» کتاب [۱] استفاده کرده ایم. بلوک دیاگرام این حافظه در شکل «۳-۱» آورده شده است.



شکل ۳-۱: بلوک‌دیاگرام مدل رفتاری حافظه

در این مدل پورت we برای مشخص کردن وضعیت Read یا Write تعریف شده است. همچنین برای دیتا، باسی با اندازه ۳۲ بیت و برای آدرس یک باس ۷ بیتی درنظر گرفته شده است. مدل ما یک پورت clk هم دارد که کلاک مدار را تامین می‌کند.

۳-۱-۱ رفتار مدل

رفتار مدار بدین صورت است که با تعریف یک آرایه ۲ بعدی (ماتریس) با ابعاد ۳۲ × $۲^۷$ در کد، عینا رفتار یک حافظه را شبیه‌سازی می‌کنیم.

در هر لبه بالارونده کلاک ابتدا وضعیت we را چک می‌کنیم. اگر مقدار آن برابر با ۱ بود یعنی در فاز write هستیم و مقدار موجود بر روی پورت din را درون adr که پورت آدرس است می‌نویسیم. به بیانی ساده تر با مشخص کردن یک سطر و ستون از ماتریس موجود و انتخاب آن، دیتا مورد نظر را در آن محل می‌نویسیم. در غیر این صورت (اگر we=0) باشد در فاز read قرار داریم و با انتخاب آدرس، محتوای موجود در آن آدرس (محتوای موجود در سطر و ستون متناظر ماتریس) را می‌خوانیم و بر روی پورت dout قرار می‌دهیم.

برای اینکه وضعیت dout در زمان write مشخص باشد، آن را High-impedance (Z) تعریف کرده ایم.

کد توصیف سخت‌افزار این مدل در ادامه آورده شده است.

همچنین فایل کد نیز در مسیر codes/behavioral/ram_array قرار داده شده است.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity ram_array is
6     generic( N: integer := 7; -- 2^7 = 128 word
7              M: integer := 32 ); -- 32 bit data
8
9     port( clk: in std_logic;
10          we: in std_logic; -- write enable
11          adr: in std_logic_vector(N - 1 downto 0); -- address
12          din: in std_logic_vector(M - 1 downto 0); -- data in
13          dout: out std_logic_vector(M - 1 downto 0) ); -- data out
14 end ram_array;
15
16 architecture Behavioral of ram_array is
17    type mem_array is array((2 ** N-1) downto 0) of std_logic_vector(M - 1
         downto 0); --create 2D array (matrix)
18    signal mem: mem_array;
```

```
19
20 begin
21    process(clk)
22    begin
23       if(rising_edge(clk)) then
24          if(we = '1') then
25             mem(conv_integer(adr)) <= din; -- write phase
26             dout <= (others => 'Z');
27          else
28             dout <= mem(conv_integer(adr)); -- read phase
29          end if;
30       end if;
31    end process;
32 end Behavioral;
```

۳-۱-۲ شبیه‌سازی مدل

برای شبیه‌سازی حافظه نوشته شده یک Testbench که فایل آن در مسیر codes/behavioral/tb_ram_mem موجود است نوشته شده است.

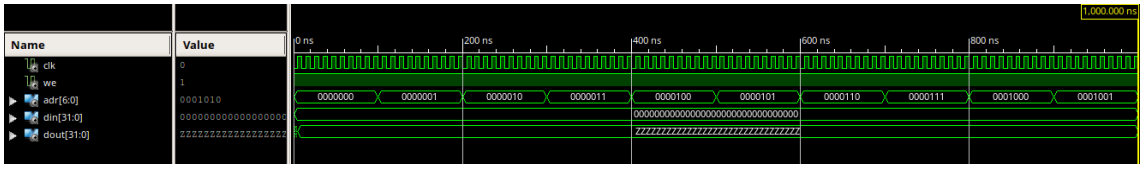
در ابتدای شبیه‌سازی، تمامی خانه‌های حافظه را با صفر پر می‌کنیم. این کار را به این دلیل انجام می‌دهیم که عملکرد ریست خودکار مدار را در ابتدای روشن شدن شبیه‌سازی کنیم.

این قسمت از شبیه سازی با قطعه کد زیر انجام شده است:

```
1  -- Stimulus process
2  stim_proc: process
3  begin
4     for i in 0 to (2 ** adr'length-1) loop
5        we <= '1';
6        adr <= std_logic_vector(to_unsigned(i, adr'length));
7        din <= std_logic_vector(to_unsigned(0, din'length));
8        wait for 100 ns;
9     end loop;
10    wait;
11 end process;
```

خروجی این قسمت در شکل «۳-۲» آورده شده است.

پس از ریست کردن مدار، عملکرد مدار را در دو فاز Write و Read با چند مقدار دلخواه آدرس و دیتا تست می‌کنیم:

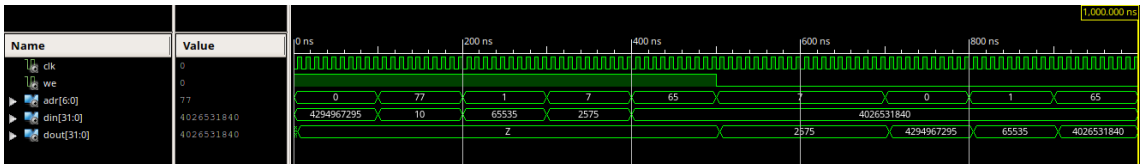


شکل ۳-۲: حالت ریست حافظه

```
-- Stimulus process
stim_proc: process
begin
   -- Write phase
   we <= '1';
   adr <= "0000000";
   din <= X"FFFFFFFF";
   wait for 100 ns;

   adr <= "1001101";
   din <= X"0000000A";
   wait for 100 ns;

   adr <= "0000001";
   din <= X"0000FFFF";
   wait for 100 ns;

   adr <= "0000111";
   din <= X"00000A0F";
   wait for 100 ns;

   adr <= "1000001";
   din <= X"F0000000";
   wait for 100 ns;



   -- Read phase
   we <= '0';

   adr <= "0000111";
   wait for 100 ns;

   adr <= "0000111";
   wait for 100 ns;

   adr <= "0000000";
   wait for 100 ns;

   adr <= "0000001";
```

```
41      wait for 100 ns;
42
43      adr <= "1000001";
44      wait for 100 ns;
45
46      adr <= "1001101";
47      wait for 100 ns;
48  end process;
```

خروجی این قسمت در شکل «۳-۳» آورده شده است. «برای مشاهده بهتر، Radix سیگنال ها از binary به unsigned decimal تغییر داده شده است»



شکل ۳-۳: تست حافظه با مقادیر دلخواه

همانطور که از خروجی سیگنال‌ها مشخص است، زمانی که we=1 است در فاز نوشتن قرار داریم و دیتا در آدرس مورد نظر نوشته می‌شود.

زمانی که we=0 می‌کنیم، مشاهده می‌شود با دادن آدرس هایی که در فاز قبل درون آنها نوشته بودیم، مقادیر آنها بر روی پورت dout قرار داده می‌شود.

ابزار سنتز نرم‌افزار ISE کد نوشته شده را با توجه به خانواده FPGA انتخاب شده به هنگام ساخت پروژه سنتز می‌کند و با دید کامل بر سخت‌افزار و Logic-block های هر FPGA بهینه[۱] ترین حالت ممکن برای پیاده سازی سخت‌افزاری کد را تولید می‌کند.

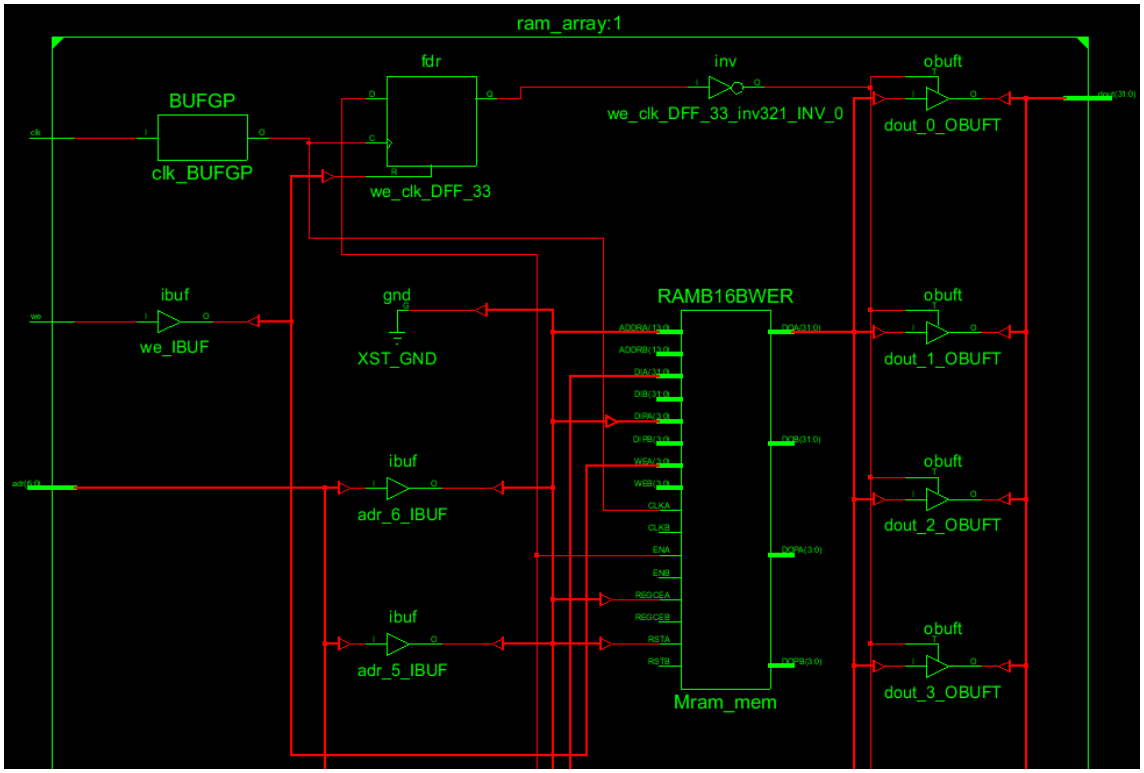
FPGA انتخاب شده در این پروژه از خانواده Xilinx Spartan 6، مدل XC6SLX4 و پکیج TQG144 است.

حال اگر به داخل مدار سنتز شده توسط ابزار سنتز[۲] نگاهی بی‌اندازیم، می‌بینیم که عینا یک بلوک RAM ساخته شده است. شکل «۴-۳»

*** به دلیل بزرگی مدار تولید شده، صرفا در شکل «۴-۳» بخشی از آن آورده شده است. ***

همچنین اگر Report های تولید شده پس از سنتز را مطالعه کنیم، در آن نوشته شده است که این مدار توصیف کننده حافظه ای ۳۲ × ۱۲۸ کلمه ایست. شکل «۶-۳»

[۱] Optimum
[۲] synthesizer tool



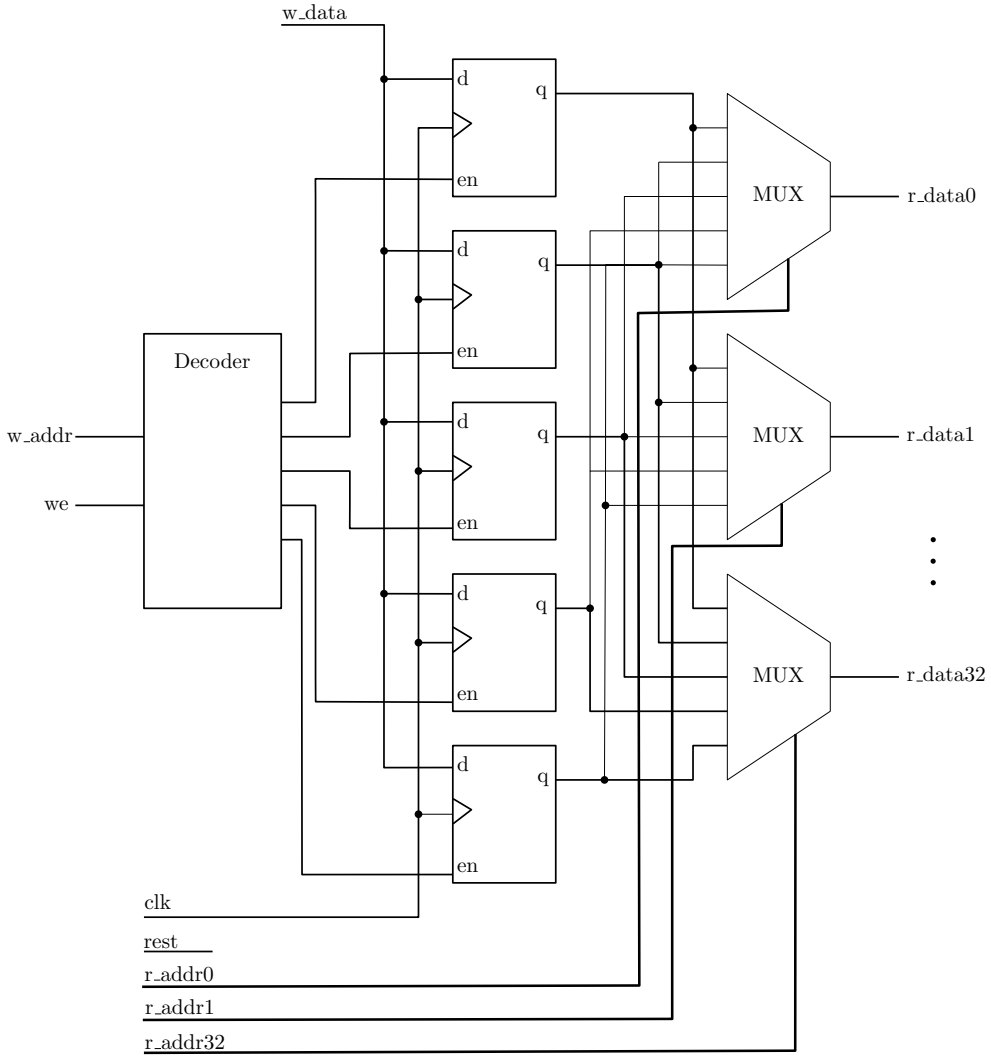
شکل ۳-۴: مدار ساخته شده توسط ابزار سنتز

```
================================================================================
Advanced HDL Synthesis Report

Macro Statistics
# RAMs                                                 : 1
 128x32-bit single-port block RAM                     : 1
# Registers                                            : 1
 Flip-Flops                                           : 1

================================================================================
```

شکل ۳-۵: خروجی ریپورت ابزار سنتز

۳-۲ مدل RTL

در مدل RTL شبیه به مدل Behavioral عمل کردیم یعنی بخش حافظه به صورت رفتاری همانند مدل Behavioral تعریف شده است. اما بخش جانبی (سخت‌افزاری) حافظه که شامل Decoder ها و MUX هاست، به صورت RTL نوشته شده است که در ادامه به توصیف هر بخش از کد می‌پردازیم. بلوک دیاگرام طراحی ما به صورت زیر است:



شکل ۳-۶: خروجی ریپورت ابزار سنتز

۳-۲-۱ رفتار مدل

مشابه به مدل Behavioral در این مدل هم یک پورت برای `clk` و `we` در نظر گرفته شده است که مشخص می‌کند می‌خواهیم در حافظه بنویسیم و یا از آن بخوانیم. یک پورت هم برای `rest` حافظه در نظر گرفته شده است. برای خواندن و نوشتن، ۴ پورت درنظر گرفته شده است. ۲ پورت `w_addr` و `w_data` به ترتیب برای نوشتن آدرس و دیتا و ۲ پورت `r_addr` و `r_data` برای خواندن آدرس و دیتا در نظر گرفته شده است.

** در اینجا می‌بایست `r_addr` و `r_data` به تعداد ۳۲ عدد باشد اما به دلیل شلوغ شدن کد، صرفا یکی از آنها آورده شده است ***

با استفاده از سیگنال های `w_addr` و `we` می‌توان انتخاب کرد که کدام یک از DFF ها فعال شود و پس از انتخاب DFF مورد نظر، می‌توان با استفاده از پورت `w_data` مقدار مورد نظر را در آن نوشت. تمامی عملیات

های گفته شده به صورت همزمان[۳] با هم و در لبه بالا رونده سیگنال clk انجام می‌شود.

در ادامه قسمت entity طراحی آورده شده است.

```
entity reg_file is
  port( clk, rest: in std_logic;
        we: in std_logic;
        w_addr: in std_logic_vector(6 downto 0);
        w_data: in std_logic_vector(31 downto 0);
        r_addr: in std_logic_vector(6 downto 0);
        r_data: out std_logic_vector(31 downto 0) );
end reg_file;
```

مدار خواندن دیتا از حافظه از ۳۲ عدد MUX تشکیل شده است که از r_addr0 تا r_addr32 می‌توان به عنوان سیگنال های کنترلی برای انتخاب خروجی های دلخواه استفاده کرد.

همانند مدل Behavioral ، رجیستر ها به صورت یک آرایه دوبعدی تعریف شده اند و چون در VHDL آرایه دو بعدی وجود ندارد آن را خودمان به عنوان سیگنالی جدید به صورت زیر تعریف می‌کنیم:

```
constant W: natural := 7; -- number of bits in address
constant B: natural := 32; -- number of bits in data

type reg_file_type is array(2 ** W-1 downto 0) of
     std_logic_vector(B-1 downto 0);

signal array_reg: reg_file_type;
signal array_next: reg_file_type;
signal en: std_logic_vector(2 ** W-1 downto 0);
```

کد قسمت رجیستر و نوشتن در حافظه به صورت زیر نوشته شده است.

```
-- register
process(clk, rest)
begin

  if(rest = '1') then
    for i in W+1 downto 0 loop
      array_reg(i) <= (others => '0');
    end loop;
--    array_reg(3) <= (others => '0');
--    array_reg(2) <= (others => '0');
--    array_reg(1) <= (others => '0');
--    array_reg(0) <= (others => '0');

  elsif(rising_edge(clk)) then
```

[۳] Synchronize

```
      for i in W+1 downto 0 loop
         array_reg(i) <= array_next(i);
      end loop;
--       array_reg(3) <= array_next(3);
--       array_reg(2) <= array_next(2);
--       array_reg(1) <= array_next(1);
--       array_reg(0) <= array_next(0);
   end if;
end process;
```

به طور کلی برای قسمت هایی از کد که می‌بایست عمل تکراری را انجام می‌دادیم از دستورات for loop در محیط concurrent و از دستور for generate در محیط parallel استفاده کرده ایم.

در process دوم، enable برای هر رجیستر بررسی می‌شود:

```
-- enable logic for register
process(array_reg, en, w_data)
begin
   for i in W+1 downto 0 loop
        array_next(i) <= array_reg(i);
   end loop;


   for i in W+1 downto 0 loop
      if(en(i) = '1') then
         array_next(i) <= w_data;
      end if;
   end loop;
end process;
```

برای دیکدر آدرس می‌توانستیم به صورت زیر عمل کنیم:

```
-- decoding for write address
process(we, w_addr)
begin
   if(we = '0') then
      en <= (others => '0');
   else

      case w_addr is
         when "00" => en <= "0001";
         when "01" => en <= "0010";
         when "10" => en <= "0100";
         when others => en <= "1000";
      end case;
   end if;
end process;
```

اما به دلیل آنکه تعداد حالات زیادی باید نوشته می‌شد (۱۲۸ حالت)، از دستور for loop استفاده کرده ایم که نمود سخت‌افزاری آن دقیقا معادل است با پیاده سازی یک دیکدر، کد دیکدر آدرس در ادامه آورده شده است:

```
-- decoding for write address
process(we, w_addr)
begin
   if(we = '0') then
      en <= (others => '0');
   else
      for i in 0 to 2**W-1 loop
         if(w_addr = std_logic_vector(to_unsigned(i, w_addr'length))) then
            en(i) <= '1';
         else
            en(i) <= '0';
         end if;
      end loop;
   end if;
end process;
```

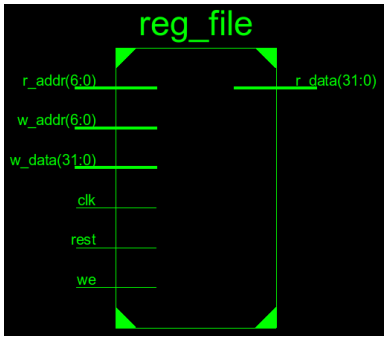
قسمت MUX طراحی نیز به صورت زیر نوشته شده است که به دلیل تعداد حالات بالا، صرفا یکی از خروجی ها و چند حالت آن را در گزارش آورده ایم:

```
-- read multiplexing
with r_addr select
   r_data <= array_reg(0) when "0000000",
             array_reg(1) when "0000001",
             array_reg(2) when "0000010",
             array_reg(3) when "0000011",
             array_reg(4) when "0000100",
             array_reg(5) when "0000101",
             .
             .
             .
             array_reg(127) when others;
```

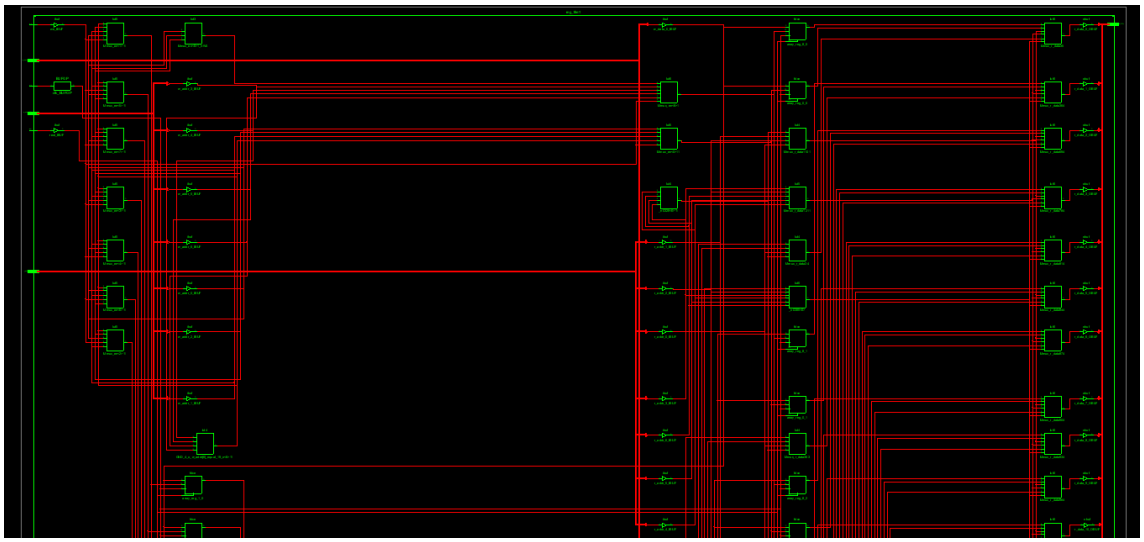
تصویری از Technology schematic طراحی انجام شده در شکل «۳-۷» آورده شده است.

همچنین مدار درونی طراحی شده توسط ابزار سنتز ISE نیز در شکل «۳-۸» آورده شده است. اما به دلیل بزرگی مدار، تنها قسمتی از آن در شکل «۳-۸» آورده شده است.

خروجی شکل «۳-۸» برای حالتی از synthesizer است که Design-goal را بر روی Balanced تنظیم کرده ایم. سایر گزینه های ابزار سنتز به صورت زیر است که بسته به کاربرد می‌توان آن‌ها را انتخاب کرد و ابزار سنتز بر اساس همان، بهینه سازی های طراحی‌مان را انجام دهد.



شکل ۳-۷: تکنولوژی شماتیک طراحی RTL



شکل ۳-۸: مدار درونی طراحی شده

- Area reduction
- Minimum runtime
- Power optimization
- Timing performance

برای مثال زمانی که Design-goal بر روی Balanced تنظیم شده است، پارامترهای زمانی و تاخیر های مدار به صورت زیر در خروجی گزارش می‌شود:

اما اگر حالت Optimization را به Timing performance تغییر دهیم و یک بار دیگر طراحی را سنتز کنیم، مشاهده می‌شود که پارامتر های زمانی همچون تاخیر مسیر[۴] کاهش می‌یابد. شکل «۳-۱۰»

[۴] Path delay

```
Timing Summary:
---------------
Speed Grade: -2

   Minimum period: No path found
   Minimum input arrival time before clock: 5.879ns
   Maximum output required time after clock: 7.669ns
   Maximum combinational path delay: 10.389ns
```

شکل ۳-۹: پارامترهای زمانی گزارش شده برای حالت Balanced

```
Timing Summary:
---------------
Speed Grade: -2

   Minimum period: No path found
   Minimum input arrival time before clock: 5.591ns
   Maximum output required time after clock: 6.542ns
   Maximum combinational path delay: 9.316ns
```

شکل ۳-۱۰: پارامترهای زمانی گزارش شده برای حالت Timing performance

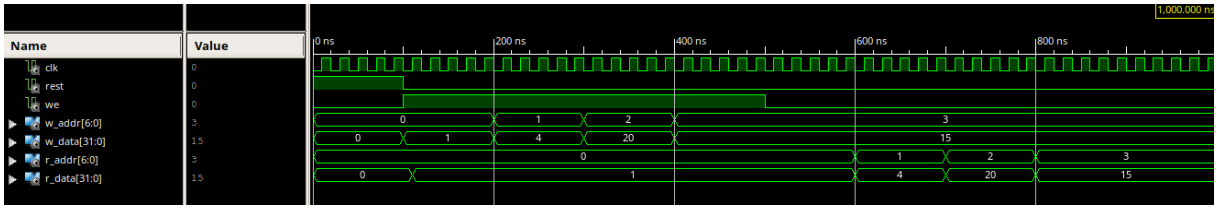
۳-۲-۲ شبیه‌سازی مدل

برای شبیه‌سازی طراحی انجام شده، مشابه با مدل قبل، برنامه Test-bench ای می‌نویسیم که در ادامه حالت های شبیه سازی شده در آن را آورده ایم:

```
-- Stimulus process
stim_proc: process
begin
   rest <= '1';
   wait for 100 ns;

   rest <= '0';
   we <= '1';
   w_addr <= "0000000";
   w_data <= X"00000001";
   wait for 100 ns;

   w_addr <= "0000001";
   w_data <= X"00000004";
   wait for 100 ns;

   w_addr <= "0000010";
   w_data <= X"00000014";
   wait for 100 ns;

```

```
21      w_addr <= "0000011";
22      w_data <= X"0000000F";
23      wait for 100 ns;
24
25      we <= '0';
26      r_addr <= "0000000";
27      wait for 100 ns;
28
29      r_addr <= "0000001";
30      wait for 100 ns;
31
32      r_addr <= "0000010";
33      wait for 100 ns;
34
35      r_addr <= "0000011";
36      wait for 100 ns;
37  wait;
38  end process;
```

خروجی شبیه سازی به صورت زیر گزارش می‌شود:



شکل ۳-۱۱: خروجی شبیه سازی طراحی RTL

همانطور که مشاهده می‌شود، در ابتدای کار مدار که سیگنال rest فعال می‌شود، خروجی حافظه ۰ می‌شود. پس از صفر شدن rest و فعال شدن we وارد فاز نوشتن درون حافظه می‌شویم. و دیتایی که بر روی باس w_data می‌گذاریم را درون آدرس موجود بر روی باس w_addr می‌نویسیم.

زمانی که we=0 می‌شود، وارد فاز Read می‌شویم و دیتاهایی که در فاز Write نوشته ایم را از همان آدرس ها می‌خوانیم.

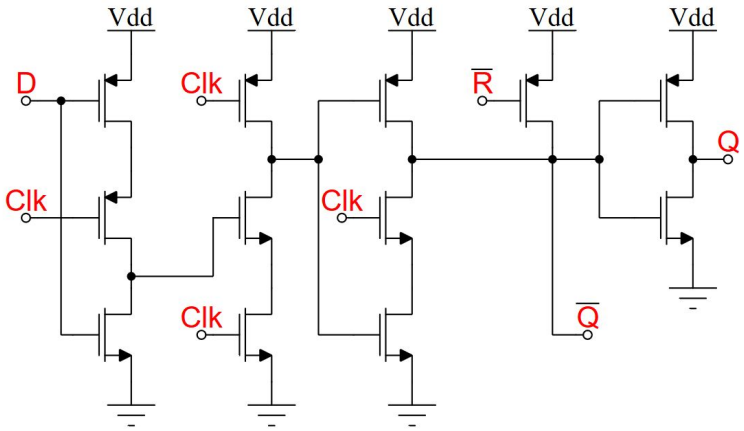
فصل ۴

طراحی ارائه شده در فاز دوم

در این فاز از پروژه به طراحی و شبیه‌سازی در سطح ترانزیستور یک SRAM با اندازه ۱۲۸ کلمه ۳۲ بیتی با استفاده از نرم‌افزار HSPICE می‌پردازیم.

۴-۱ سلول SRAM

یک سلول SRAM باید بتواند دیتا را بخواند و بنویسد و تا زمانی که تغذیه آن قطع نشده است، دیتا را ذخیره[۱] کند. یک ثبات[۲] معمولی می‌تواند این کار را انجام دهد. اما اندازه آن بسیار بزرگ است. در «شکل ۴-۱» نمونه ای از یک ثبات با ترانزیستورهای CMOS آورده شده است.



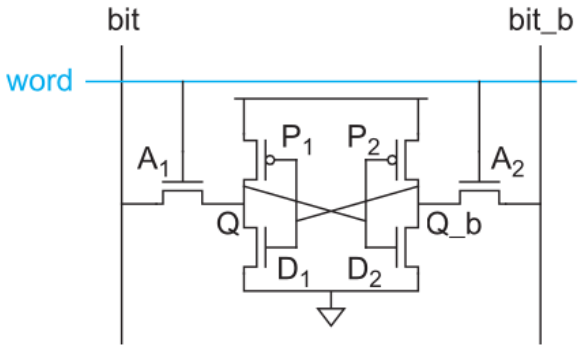
شکل ۴-۱: مدار ثبات نوع D [لینک]

همانطور که مشاهده می‌شود در این طراحی ۱۲ ترانزیستور MOSFET استفاده شده است و اگر بخواهیم

[۱] Register
[۲] Flip Flop

حافظه ای مبتنی بر ثبات‌ها طراحی کنیم، تعداد ترانزیستورهای بسیار زیادی مصرف می‌شود و سطح سلیکون مصرفی برای ساخت تراشه ای از این جنس بسیار زیاد خواهد شد.

بنابراین تمایلی برای ساخت حافظه ای مبتنی بر این مدل نداریم. و به همین دلیل به سراغ مدلی مبتنی بر ۶ ترانزیستور می‌رویم که به مراتب اندازه بسیار کوچک‌تری از فلیپ‌فلاپ دارد. «شکل ۴-۲»



شکل ۴-۲: سلول ۶ ترانزیستوری SRAM

با کوچک شدن ابعاد طراحی طول مسیر سیم‌ها نیز کوتاه‌تر می‌شود پس در نتیجه توان دینامیکی این طراحی نیز به نسبت فلیپ‌فلاپ کمتر است.

نواحی کاری یک سلول حافظه، به صورت زیر تقسیم بندی می‌شوند:

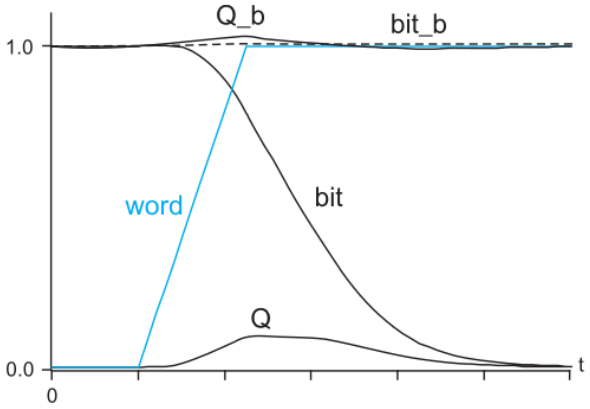
- حالت نوشتن
- حالت خواندن

۴-۱-۱ حالت خواندن

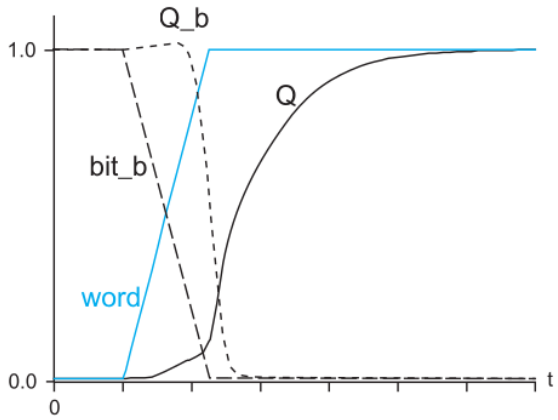
شکل «۴-۳» یک سلول SRAM را در حالت خواندن نشان می‌دهد. در ابتدای کار، بیت‌لاین ها بدون از دست دادن انرژی، شناور هستند. فرض شود Q در ابتدا ۰ است، پس Q_b مقدار ۱ دارد و باید ۱ باقی بماند. وقتی مقدار Word line یک می‌شود، مقدار Bit از طریق ترانزیستور A۱ صفر می‌شود.

۴-۱-۲ حالت نوشتن

شکل «۴-۴» نیز، یک سلول SRAM را برای حالت نوشتن نشان می‌دهد.



شکل ۴-۳: نمودار حالت خواندن برای سلول ۶ ترانزیستوری SRAM



شکل ۴-۴: نمودار حالت نوشتن برای سلول ۶ ترانزیستوری SRAM

۴-۲ سایز ترانزیستورها

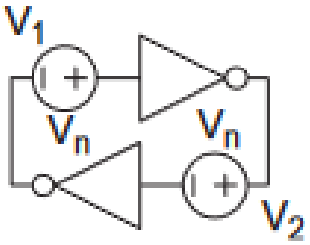
مهم‌ترین بخش این طراحی، تعیین ابعاد[۳] ترانزیستورهاست.

اگر یک سلول SRAM به صورت «شکل ۴-۵» در نظر گرفته شود، با اعمال ولتاژهای V۱ و V۲ نمودار V۲ برحسب V۱ باید به صورت «شکل ۴-۶» شود و الزاما ابعاد ترانزیستور ها به نسبت ۲ به ۱ اینورتر واحد نیست.

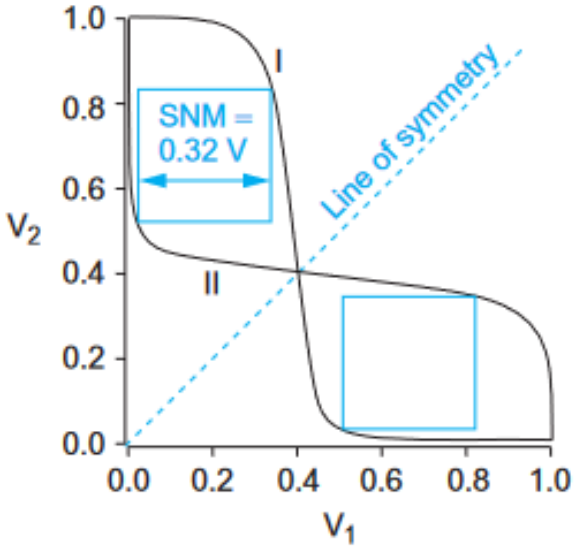
۴-۳ ساختار طراحی

در این پروژه دو طراحی انجام شده است. یک SRAM با ابعاد ۱۲۸ × ۳۲ و SRAM دیگر با ابعاد ۸ × ۴ این کار به این دلیل انجام شده است که، به خاطر تعداد حالات بالا، انتخاب سایز ترانزیستور ها و تست طراحی

[۳] $\frac{W}{L}$



شکل ۴-۵: شمای جدید سلول SRAM بر حسب گیت Inverter



شکل ۴-۶: نمودار پروانه ای

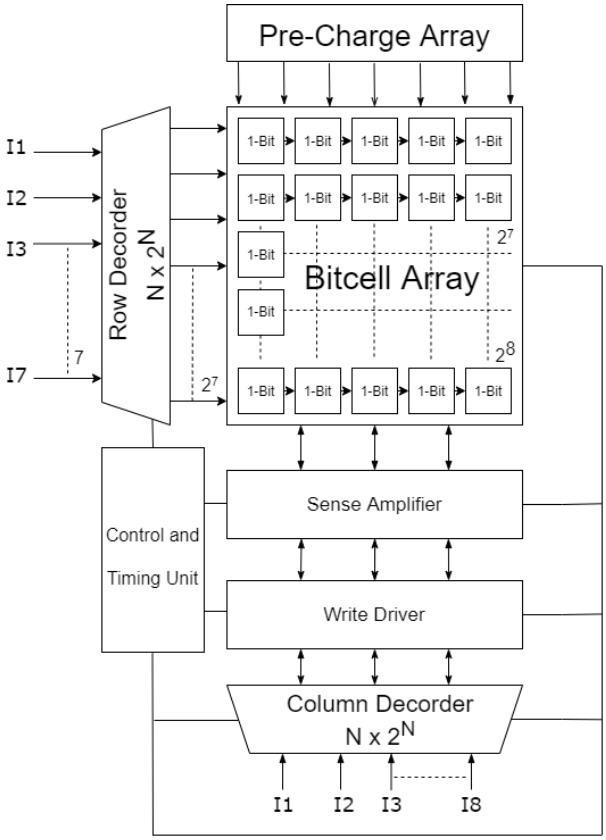
انجام شده دشوار بود. بنابر این در ادامه به بررسی ساختار هر دو کد می‌پردازیم.

طراحی ساختاری که دنبال کردیم، به صورت «شکل ۴-۷» است. که در ادامه به تعریف و بررسی کدهای نوشته شده برای هر بخش می‌پردازیم.

۴-۴ تعریف سلول SRAM در HSPICE

یک سلول SRAM را می‌توان به صورت زیر تعریف کرد:

در این طراحی ابعاد ترانزیستورهای NMOS ، $\frac{W}{L} = \frac{۸}{۴}$ و ابعاد PMOS ها، $\frac{W}{L} = \frac{۱۰}{۴}$ درنظر گرفته شده است. این مقادیر را با سعی و خطا بدست آورده ایم.



شکل ۴-۷: شماتیک طراحی انجام شده

```
.SUBCKT SRAM wl bl blnot q qnot
* D G S
M1   Qnot Q    gnd  gnd   nmos   L= '2*lambda' W= '8*lambda' *==>strong
M2   Qnot Q  Vdd  Vdd pmos     L='2*lambda' W='10*lambda'
M3   Q    Qnot  gnd  gnd nmos      L= '2*lambda' W= '8*lambda' *==>strong
M4   Q    Qnot  Vdd  Vdd  pmos      L='2*lambda' W='10*lambda'
M5   qnot WL   blnot   gnd nmos     L= '2*lambda' W= '4*lambda'
M6   BL     WL  Q    gnd nmos      L= '2*lambda' W= '4*lambda'
.ends

XSRAM WL1 bl1 blnot1 Q1 Qnot1 SRAM
```

برای مدل ترانزیستورها نیز از کتابخانه `mosistsmc180.lib` و فناوری ۱۸۰nm استفاده کرده ایم که می‌توان آن را از مسیر `Code/Sram_Cell/mosistsmc180.lib` در یافت کرد.

در ادامه یک ماژول اینورتر نیز تعریف می‌کنیم که کد آن آورده شده است:

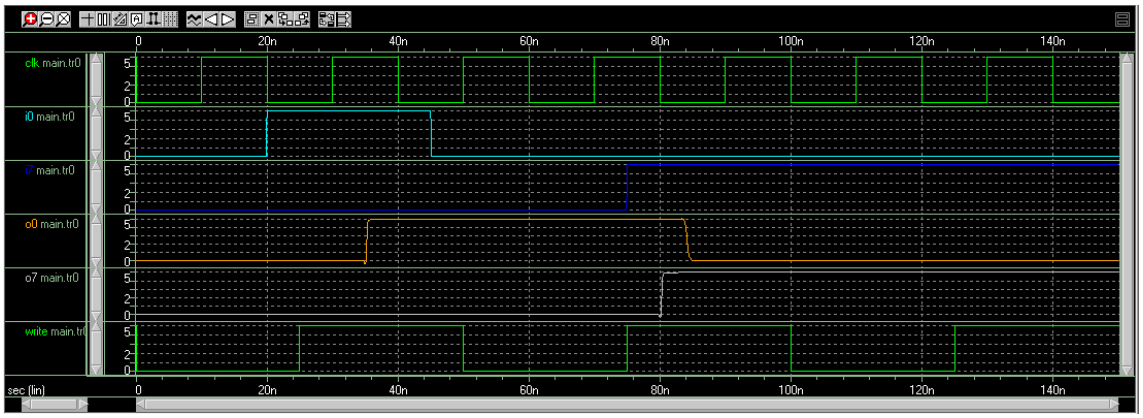
```
*-- Inverter ---
* D G S BODY
.SUBCKT NOT A A!
  MP A! A  VDD VDD  PMOS L='2*lambda' W='10*lambda' *-- inverter ==> 2
```

```
    barabare nmos
5   MN A! A   0    0   NMOS L='2*lambda' W='4*lambda'
6 .ENDS
```

برای مدار جانبی SRAM می‌بایست یک دیکدر نیز تعریف کنیم که به علت طولانی بودن کد در گزارش آن را نیاورده ایم.

پس از تعریف پیش‌نیاز های طراحی، نوبت به آن می‌رسد که اتصالات سلول و مدارات جانبی را به طوری برقرار کنیم که به ابعاد مورد نظر پروژه برسیم. این بخش از کد ها ام به دلیل طولانی بودن در گزارش نیاورده شده است.

پس از آنکه شبیه‌سازی را انجام دادیم، فاز خواندن و نوشتن را برای ۲ بیت انجام می‌دهیم. برای مشاهده شکل موج‌های خروجی، از نرم‌افزار Cscope استفاده کردیم. «شکل ۴-۸»



شکل ۴-۸: خروجی شبیه‌سازی

همانطور که در شکل موج خروجی نیز مشاهده می‌شود، زمانی که مقدار سیگنال Write صفر است، در فاز خواندن قرار داریم و مقدار بیت‌های O۰ و O۷ مقادیری است که در فاز قبلی نوشته شده است.

هنگامی که سیگنال Write یک است، در فاز نوشتن قرار داریم و مقادیر سیگنال‌های i۰ و i۷ را در آدرس مشخص شده «در اینجا آدرس ۰۰۰۰» می‌نویسیم.

فصل ۵

مقایسه مدل ها و نتیجه گیری

به طور عمده تفاوت مدل Behavioral و مدل RTL در نحوه نوشتن کدهای بخش سخت‌افزار (دیکدر های آدرس و مالتی‌پلکسر های دیتاست) به گونه ای که در مدل Behavioral دسترسی به سطر ها و ستون های حافظه به صورت `s[i]` و `s[i][j]` انجام شده است اما در مدل RTL با استفاده از دیکدری که برای دسترسی به هر سطر و ستون نوشته ایم، این دسترسی انجام می‌شود.

همانطور که از شکل‌های «۳-۴» و «۳-۸» مشخص است، تفاوت دیگری که این دو مدل با هم دارند، مربوط به نحوه پیاده‌سازی[۱] آن‌ها توسط ابزار سنتز می‌شود. به طوری که مدار تولید شده در طراحی RTL بسیار از نظر مساحت[۲] بزرگ و ازنظر گیت‌های مصرفی هم بیشتر است.

تفاوت دیگری که این دو مدار دارند، از جهت پارامترهای زمانی سیگنال هاست. اگر حالت Optimization را برای هر دو طراحی بر روی Timing performance قرار دهیم، مشاهده می‌شود که طراحی Behavioral بهتر عمل می‌کند و تاخیرهای آن به نسبت طراحی RTL کمتر است. شکل «۵-۱» و «۳-۱۰»

```
Timing Summary:
---------------
Speed Grade: -2

   Minimum period: No path found
   Minimum input arrival time before clock: 3.373ns
   Maximum output required time after clock: 5.693ns
   Maximum combinational path delay: No path found
```

شکل ۵-۱: پارامترهای زمانی گزارش شده برای حالت Timing performance در طراحی Behavioral

طبق خروجی‌های گرفته شده از شبیه‌سازی سطح بالای هر دو طرح، مشاهده می‌شود که طراحی ای که در

[۱] Implement
[۲] Area

سطح Behavioral انجام شده است، از نظر مساحت، تاخیر، تعداد گیت های مصرفی و توان بهینه تر است. و دلیل این امر، همانطور که قبلا هم مطرح شد، به خاطر آن است که ابزار سنتز ISE طراحی های انجام شده را بر اساس مدل و خانواده FPGA مشخص شده و همچنین Logic-block های اساسی موجود در هر FPGA سنتز و پیاده سازی می‌کند و چون در مدل RTL خودمان قسمت دیکدر و مالتی‌پلکسر را نوشتیم، ابزار سنتز نتوانسته است همانند مدل Behavioral طراحی را به صورت بهینه Implement کند. اما برای اطمینان از صحت عملکرد قسمت حافظه و سخت‌افزار SRAM می‌بایست مدل RTL حافظه را در ابتدا شبیه‌سازی می‌کردیم تا با اطمینان خاطر از عملکرد آن وارد فاز دوم پروژه که پیاده سازی و ترانزیستوری آن در HSpice است، شویم.

در فاز دوم پروژه، دو تحلیل `tran` . که تحلیل زمانی مدار، تحلیل `op` . که تحلیلی‌ست برای پیدا کردن نقطه کار[۳] «ولتاژ و جریان تمام گره‌های مدار» را انجام داده ایم که نتایج این دو تحلیل در فایل `finalsram.lic` موجود است.

همچنین توان مصرفی مدار را هم با استفاده از دستور `.PRINT PTOTAL=PAR('V(VDD)*I(VDD)')` اندازه گیری کرده ایم که در خروجی مقدار ۵۱.۲۷۱۶ میلی وات گزارش شده است.

جریان کل مصرفی SRAM طراحی شده نیز با استفاده از دستور `.PRINT ITOTAL=I(VDD)` محاسبه شده است که مقدار آن ۲۸.۴۸۴۲ میلی‌آمپر گزارش شده است.

---

[۳] Bias Point

Bibliography

[1] D. M. H. Neil H. E. Weste. *CMOS VLSI Design A Circuits and Systems Perspective.* Addison-Wesley, 2011.

[2] V. A. Pedroni. *Circuit Design With VHDL.* Massachusetts Institute of Technology, 2004.

[3] P. P. CHU. *FPGA Prototyping By VHDL Examples.* Wiley-Interscience, 2008.

[4] P. P. CHU. *RTL Hardware Design Using VHDL Coding for Efficiency, Portability, and Scalability.* Wiley-Interscience, 2006.