طراحی سیستم های مبتنی بر ASIC/FPGA

تمرین سری ۲

علی یداللهی

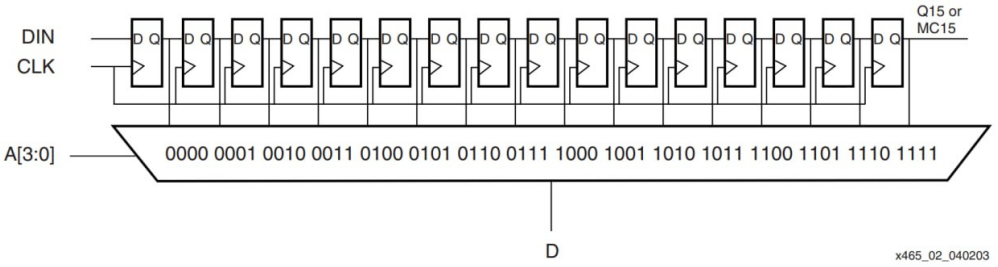
شماره دانشجویی : ۴۰۰۱۰۲۲۳۳

۱

الف

چند روش مختلف برای پیاده سازی این شیف رجیستر وجود دارد:

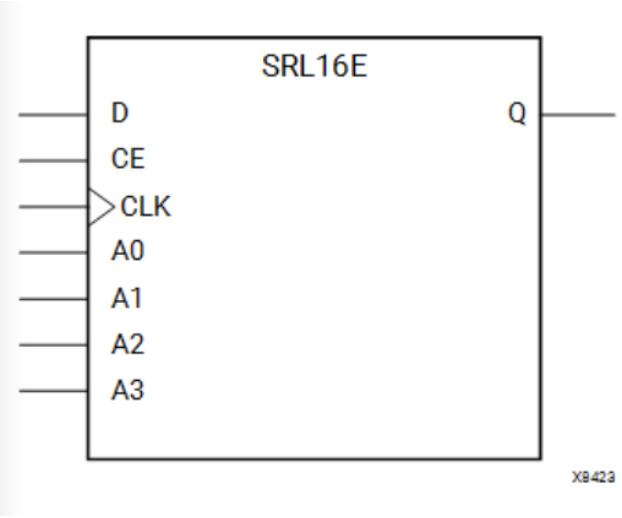
- استفاده از فلیپ فلاپ: یک روش استفاده از فلیپ فلاپ های موجود در FPGA برای ایجاد یک شیفت رجیستر است. با اتصال چند فلیپ فلاپ در یک زنجیره و کنترل سیگنال های کلاک، می توان داده ها را از طریق فلیپ فلاپ ها جابجا کرد.در شکل زیر ساختار لازم برای یک بیت نمایش داده شده است. در مثال ما طول ساختار برابر با ۶۴ خواهد بود و با قرار دادن ۶ تا از این ساختارها در کنار هم به عرض ۶ بیت می رسیم.



- استفاده از Block RAM : بلوک RAM در یک FPGA از چندین سلول حافظه تشکیل شده است که به صورت ردیف‌ها و ستون‌ها سازماندهی شده‌اند. هر سلول حافظه می‌تواند مقدار خاصی از داده را ذخیره کند.
در ابتدا، باید داده اولیه به مکان‌های بلوک RAM که متناظر با مرحله اول شیفت رجیستر هستند، لود شوند. این داده به با اعمال کلاک وارد شیفت رجیستر شده و در طول آن حرکت می کند.
برای شیفت داده، داده باید از یک مکان در بلوک RAM خوانده شود و در کلاک بعدی به مکان مجاور نوشته شود. این فرآیند برای هر چرخه کلاک ادامه پیدا می‌کند و به طور موثر داده را از طریق شیفت رجیستر حرکت می‌دهد.
برای کنترل خواندن و نوشتن به RAM و فرآیند شیفت نیاز به پیاده سازی یک منطق کنترلی داریم.
عرض و عمق بلوک RAM تعیین‌کننده ظرفیت و اندازه شیفت رجیستر است که قابل پیاده سازی است ما می‌توانیم بلوک RAM را به صورت چند ردیف و ستون پیکربندی کنیم تا یک شیفت زجیستر ۶۴ تایی با عرض بیت ۶ را درست کنیم.
- استفاده از LUT :برای پیاده‌سازی یک شیفت رجیستر با استفاده از LUT ها می‌توان از LUTها برای ایجاد یک زنجیره از فلیپ فلاپ‌ها استفاده کرد به این صورت که خروجی یک LUT به ورودی LUT بعدی به صورت پشت سر هم متصل شود.
با استفاده از۶۴ LUT می توان داده را تا ۶۴ بار شیفت داد. با قرار ۶ ساختار ۶۴ تایی در کنار هم به عرض ۶ بیت می رسیم.

ب

شکل کلی یک Shift Register LookUp Table به صورت زیر است:



ورودی های A_0 ، A_1 ، A_2 و A_3 عرض شیفت رجیستر را تایین می کنند.عرض شیفت رجیستر می تواند از ۱ تا ۱۶ بیت متغیر باشد و با فرمول زیر تایین می شود:

$$Depth = (8 \times A_3) + (4 \times A_2) + (2 \times A_1) + A_0 + 1$$

به عنوان مثال اگر ورودی A برابر (0000) باشد عرض شیفت رجیستر برابر ۱ بیت و اگر برابر با (1111) باشد عرض آن برابر با ۱۶ بیت خواهد بود.

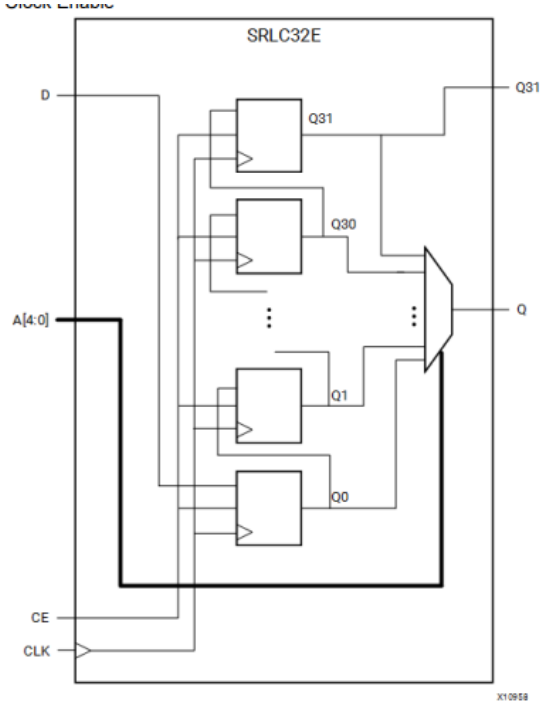
معمولا یک مقدار اولیه INIT برای شیفت رجیستر در نظر گرفته می شود. اگر مقداری برای INIT مشخص نشود مقدار اولیه شیفت رجیستر به صورت پیش فرض برابر با ۰ درنظر گرفته می شود.

هنگامی که CE برابر با ۱ است؛ داده D در لبه کلاک به داخل شیفت رجیستر لود می شود. اگر CE همچنان برابر یک باشد داده قبلی در لبه بعدی کلاک یک واحد به جلو حرکت کرده و داده جدید وارد شیفت رجیستر می شود.هنگامی که داده مورد نظر عرض شیفت رجیستر را طی کند در خروجی Q ظاهر خواهد شد.

اگر ورودی CE برابر ۰ باشد شیفت رجیستر محتوای خود را حفظ می کند و داده جدیدی به داخل رجیستر لود نمی شود.

نوع دیگری از SRL ها SRLC32E است که ساختار آن با حالت قبل تقریبا یکسان است با این تفاوت که ورودی A ۵ بیتی است و همچنین با اتصال خروجی Q31 به ورودی SRL بعدی می توان آنها را موازی کرد و SRL هایی با طول بیشتر داشت.

با سری کردن دو SRL با هم میتوانیم یک شیفت رجیستر ۶۴ تایی بسازیم. با کنار هم قرار دادن ۶ تا از این ساختارها می توان به عرض ۶ بیت دست یافت.



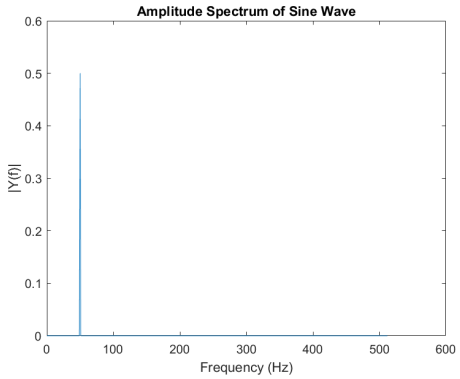
۲

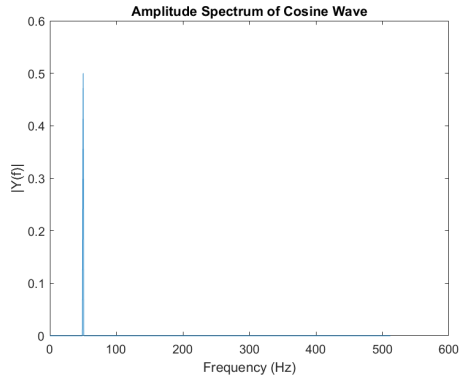
الف_ب

در این سوال ابتدا دو سیگنال سینوسی و کسینوسی به طول ۱۰۲۴ تولید کردم،سپس دو سیگنال را به فرمت fixed point i؛ که قسمت طبیعی عدد ۲ بیت و قسمت اعشاری ۱۴ بیت دارد تبدیل کردم.سپس این دو سیگنال را در دو فایل جداگانه ذخیره کردم. سپس این دو فایل را توسط وریلاگ خوانده و با توجه به ورودی op فرکانس خروجی را تایین کرده و دو سیگنال سینوسی و کسینوسی جدید را در فایل های جدیدی ذخیره کردم. نحوه تولید سیگنال ها با فرکانس بالاتر به این صورت است که برای فرکانس دوبرابر داده های اولیه به صورت یکی درمیان نوشته می شوند و برای فرکانس ۴ برابر ۴ تا در میان و به همین ترتیب.

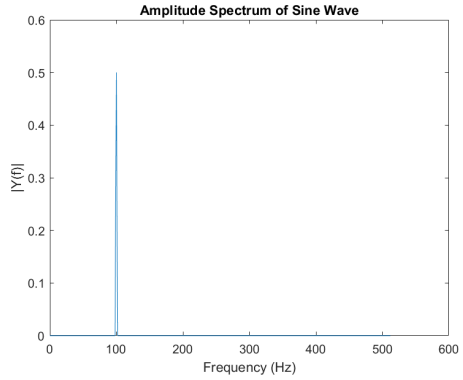
سپس داده ها را در دو فایل جدید(یکی برای سینوس و یکی کسینوس) ذخیره می کنیم و فایل های جدید را توسط متلب خوانده و دو تابع جدید را مقدار دهی می کنیم. سپس با استفاده از دستور fft متلب طیف فرکانسی آنها را رسم می کنیم.در ادامه خروجی نهایی را برای هر چهار حالت فرکانس آورده شده است.(فرکانس سیگنال ابتدایی برابر ۵۰ هرتز است.):

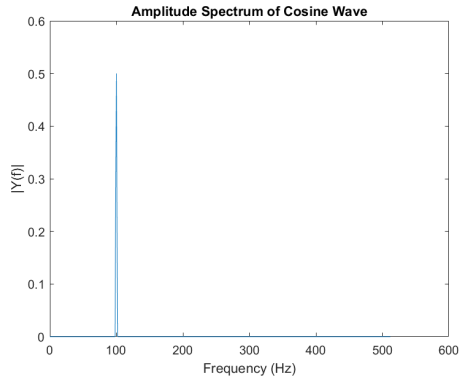
- $op = 2'b00 (f_{new} = 1 \times f_{old})$



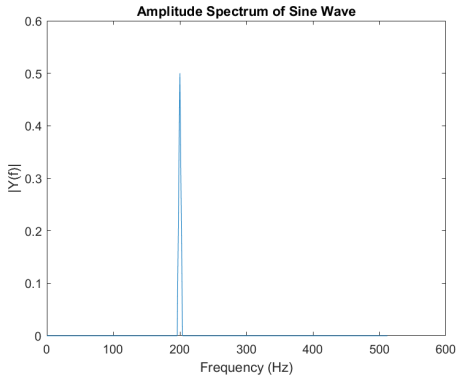


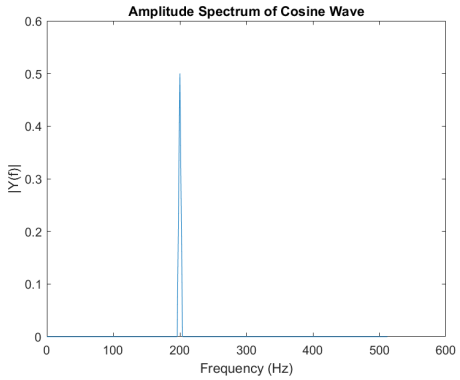
- $op = 2'b01 (f_{new} = 2 \times f_{old})$



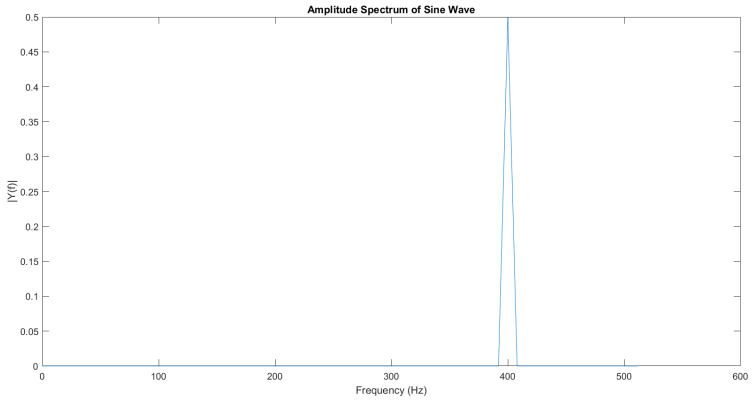


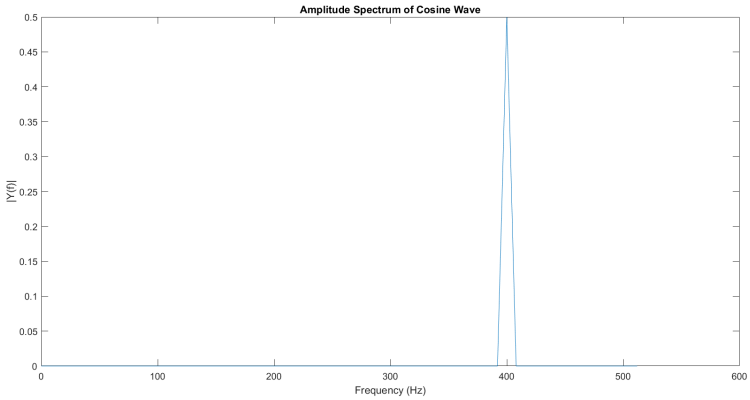
- $op = 2'b10 (f_{new} = 4 \times f_{old})$





- $op = 2'b11 (f_{new} = 8 \times f_{old})$





ج

- هسته‌های نرم‌افزاری:

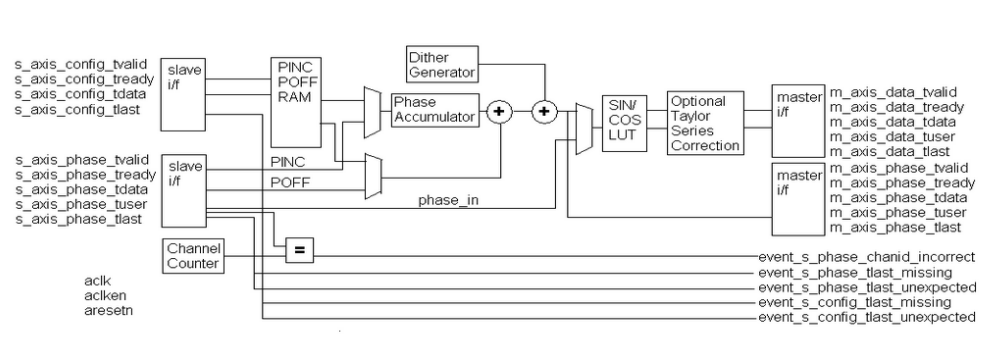
هسته‌های نرم‌افزاری در یک FPGA معمولا به قابلیت‌هایی اشاره دارند که با استفاده از یک هسته پردازنده مانند پردازنده MicroBlaze یا Nios II پیاده‌سازی می‌شوند. این پردازنده‌های نرم بر روی ساختار FPGA سنتز شده و می‌توانند با استفاده از ابزارهای توسعه نرم‌افزار مانند Xilinx SDK یا Intel Quartus Prime برنامه‌نویسی شوند. نرم‌افزاری که بر روی این پردازنده‌ها اجرا می‌شود، رفتار FPGA را کنترل می‌کند و می‌تواند با سایر مؤلفه‌های سخت‌افزاری در FPGA تعامل داشته باشد.

- هسته‌های سخت‌افزاری:

هسته‌های سخت‌افزاری در یک FPGA به قابلیت‌های اشاره دارند که با استفاده از منابع سخت‌افزاری اختصاصی بر روی ساختار FPGA پیاده‌سازی می‌شوند، مانند دروازه‌های منطقی، فلیپ فلاپ‌ها، بلوک‌های DSP و بلوک‌های حافظه. هسته‌های سخت‌افزاری با استفاده از زبان‌های توصیف سخت‌افزار مانند Verilog یا VHDL طراحی می‌شوند و به بایت استریم پیکربندی FPGA سنتز می‌شوند. هسته‌های سخت‌افزاری عملکرد سریعتر و قطعی تر را نسبت به هسته‌های نرم‌افزاری فراهم می‌کنند، زیرا به طور مستقیم از منابع سخت‌افزار FPGA بهره می‌برند.

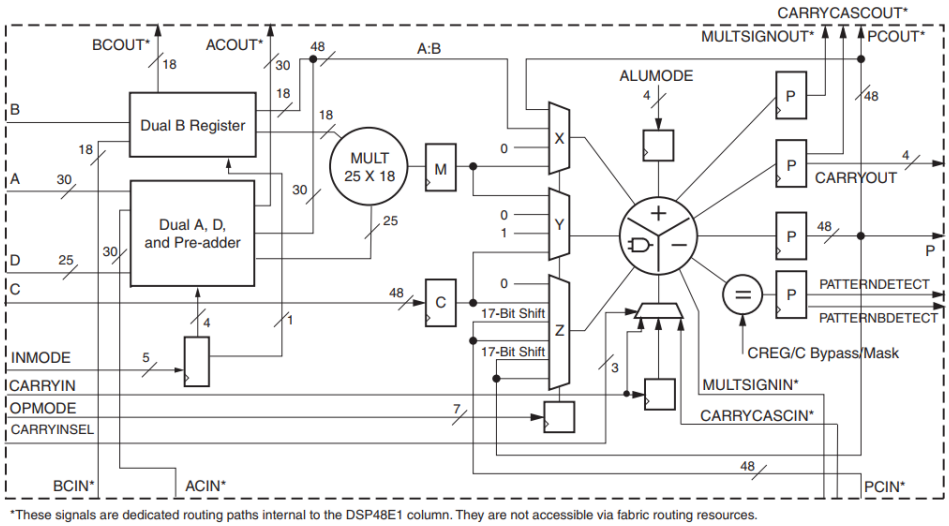
DDS منبع تولید کننده سگنال های سینوسی برای استفاده در بسیاری از کاربردها است.
برای درک کامپایلر DDS، لازم است بدانیم که چگونه بلوک در سخت افزار FPGA پیاده سازی می شود. شکل زیر بلوک دیاگرام هسته کامپایلر DDS است. هسته از دو بخش اصلی تشکیل شده است، یک بخش ژنراتور فاز و یک بخش LUT(SIN/COS).

- Phase Generator: ژنراتور فاز از یک accumulator و به دنبال آن یک جمع کننده برای ایجاد آفست فاز تشکیل شده است.افزایش و آفست فاز می تواند ثابت،قابل برنامه ریزی(programmable) یا پویا (dynamic) باشد.
- هنگامی که پیاده سازی فقط با استفاده از SINE/COS LUT انجام می شود از ژنراتور فاز استفاده ای نمی شود و با استفاده از یک Look Up table سیگنال های سینوسی و کسینوسی تولید می شوند.



۳

ساختار کلی اسلایس dsp48 به صورت زیر است:



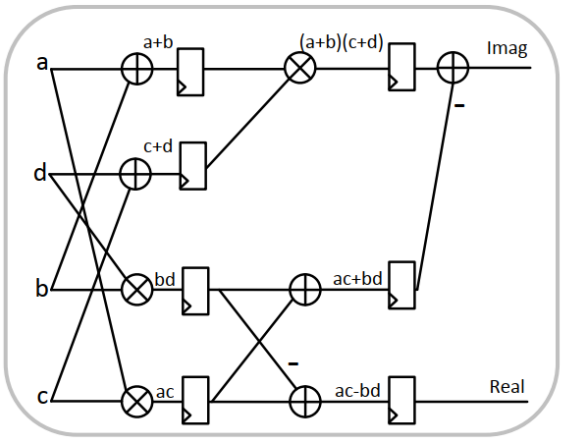
اسلایس dsp شامل یک ضرب‌کننده و سپس یک جمع‌کننده است.برای عملکرد بهینه، هر دو عملیات ضرب و ضرب–جمع نیاز به حداقل سه رجیستر به صورت پایپ لاین دارند.

سیگنال های کنترلی (OPMODE, ALUMODE, CARRYINSEL) به مولتی پلکسرها قابلیت انعطاف پذیری می دهند. در ادامه اجزای تشکیل دهنده و خلاصه ای از جزئیات آنها آورده شده است.

- 25-bit pre-adder with D register to enhance the capabilities of the A path
- INMODE control supports balanced pipelining when dynamically switching between multiply (A*B) and add operations (A:B)
- 25 x 18 multiplier
- 30-bit A input of which the lower 25 bits feed the A input of the multiplier, and the entire 30-bit input forms the upper 30 bits of the 48-bit A:B concatenate internal bus.
- Cascading A and B input
- Semi-independently selectable pipelining between direct and cascade paths
- Separate clock enables two-deep A and B set of input registers

• Independent C input and C register with independent reset and clock enable.
• CARRYCASCIN and CARRYCASCOUT internal cascade signals to support 96-bit accumulators/adders/subtracters in two DSP48E1 slices
• MULTSIGNIN and MULTSIGNOUT internal cascade signals with special OPMODE setting to support a 96-bit MACC extension
• Single Instruction Multiple Data (SIMD) Mode for three-input adder/subtracter which precludes use of multiplier in first stage
• Dual 24-bit SIMD adder/subtracter/accumulator with two separate CARRYOUT signals
• Quad 12-bit SIMD adder/subtracter/accumulator with four separate CARRYOUT signals
• 48-bit logic unit
• Bitwise logic operations – two-input AND, OR, NOT, NAND, NOR, XOR, and XNOR
• Logic unit mode dynamically selectable via ALUMODE
• Pattern detector
• Overflow/underflow support
• Convergent rounding support
• Terminal count detection support and auto resetting
• Cascading 48-bit P bus supports internal low-power adder cascade
• The 48-bit P bus allows for 12-bit/QUAD or 24-bit/DUAL SIMD adder cascade support
• Optional 17-bit right shift to enable wider multiplier implementation
• Dynamic user-controlled operating modes
• 7-bit OPMODE control bus provides X, Y, and Z multiplexer select signals
• Carry in for the second stage adder
• Support for rounding
• Support for wider add/subtracts
• 3-bit CARRYINSEL multiplexer
• Carry out for the second stage adder
• Support for wider add/subtracts
• Available for each SIMD adder (up to four)

در این سوال از ساختار معرفی شده در اسلایدهای درس که یک ساختار پایپ لاین است استفاده می کنیم. شکل ساختار مورد نظر در ادامه آورده شده است.



به ازای چند ورودی خروجی مدار در ادامه آورده شده است:

```
VSIM 26> run -all
# (      1+j      1)(      1+j      1)=
#              0+j              2
# (      2+j      3)(      4+j      1)=
#              5+j             14
# (     16+j     31)(      1+j     32)=
#            -976+j            543
# (      1+j     -2)(      1+j     -1)=
#             -1+j             -3
# ** Note: $stop    : D:/Term 6/fpga/HW/HW02/Complex_mult_T.v(48)
#    Time: 400 ns  Iteration: 0  Instance: /Complex_mult_T
# Break in Module Complex_mult_T at D:/Term 6/fpga/HW/HW02/Complex_mult_T.v line 48

VSIM 27>
```

حال کد را سنتز می کنیم تا ببینیم در پیاده سازی آن از چند dsp استفاده می شود

	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18	Pins
1	> |Complex_mult	191 (171)	298 (298)	0	11	1	5	149

می بینیم که تعداد dsp استفاده شده برابر ۱۱ است.

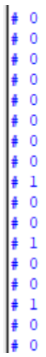
حال اندازه ورودی را افزایش می دهیم و برابر ۲۴ بیت قرار می دهیم می توان مشاهده کرد که تعداد dsp های استفاده شده تقریبا دوبرابر شده و به ۲۱ عدد می رسد.چون ضرب کننده درونی dsp ۲۵ در ۱۸ است و با افزایش سایز ورودی به بیشتر از ۱۸ بیت تعداد بلوک های مورد نیاز تقرسبا دو برابر می شود.

	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18
1	> |Complex_mult	400 (303)	407 (407)	0	21	3	9

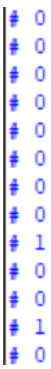
۴

ماشین مور مدنظر ما از ۹ state تشکیل می شود که از ۰ تا ۸ شماره گذاری شده اند و خروجی که نشان دهنده تشخیص دادن الگوی مورد نظر است فقط در state شماره ۸ (I) برابر با یک می شود و در بقیه حالات صفر است.
به عنوان مثال در این جا رشته 10110110110 به عنوان ورودی داده شده است. خروجی حاصل به این شکل است:

که با توجه به مثال آورده شده در صورت تمرین خروجی به دست آمده درست است.
برای ورودی 10110110110110100 خروجی در پایین آورده شده. میبینیم که الگوی مورد نظر ۳ بار تشخیص داده شده است.



در ماشین میلی می توانیم تعداد state ها را یک واحد کمتر کرده و ۸ حالت داشته باشیم که از ۰ تا ۷ شماره گذاری می شوند.در ماشین میلی خروجی هم به حالت فعلی و هم ورودی مرتبط است. در ماشین حالت طراحی شده وقتی در state شماره ۷ (H) ورودی برابر ۰ باشد خروجی ۱ می شود و الگو تشخیص داده می شود. به عنوان مثال در این جا رشته 101101101100 به عنوان ورودی داده شده است. خروجی حاصل به این شکل است:



که با توجه به مثال آورده شده در صورت تمرین خروجی به دست آمده درست است و الگو دو بار تشخیص داده می شود

```
# 0
# 0
# 0
# 0
# 0
# 0
# 0
# 0
# 1
# 0
# 0
# 1
# 0
```

.

برای ورودی 1011011011011010 خروجی در پایین آورده شده. میبینیم که الگوی مورد نظر ۳ بار تشخیص داده شده است.

```
# 0
# 0
# 0
# 0
# 0
# 0
# 0
# 0
# 0
# 1
# 0
# 0
# 1
# 0
# 0
# 1
# 0
```

منابع مصرفی در ماشین مور:

	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18	Pins
1	|Q4	11 (11)	9 (9)	0	0	0	0	4

منابع مصرفی در ماشین میلی:

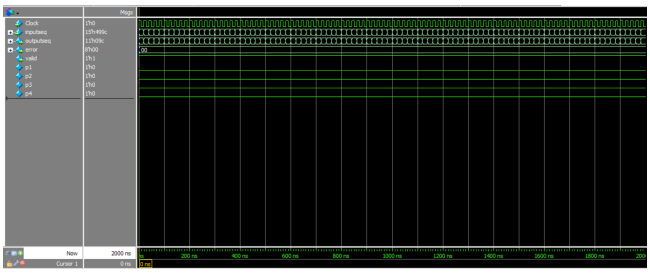
	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits	DSP Elements	DSP 9x9	DSP 18x18	Pins
1	|Q4	9 (9)	7 (7)	0	0	0	0	4

۵

با استفاده از نرم افزار متلب داده های یازده بیتی به صورت رندوم تولید شد و سپس با استفاده از کدگذاری همینگ (Ham-ming Code) آن را کدگذاری کرده و داده های ۱۵ بیتی حاصل را در فایلی به اسم hammingcode.txt ذخیره شد. داده های اولیه نیز در فایلی به اسم eleven.txt ذخیره شده اند.

در ادامه توسط وریلاگ فایل hammingcode را می خوانیم و صحت اطلاعات را چک می کنیم.و در صورت درستی اطلاعات داده مورد نظر و در صورت رخ دادن خطا داده تمام صفر را در خروجی قرار داده و در فایلی ذخیره می کنیم.

در حالت اول که هیچ داده ای تغییر داده نشده خروجی ها به این صورت هستند:



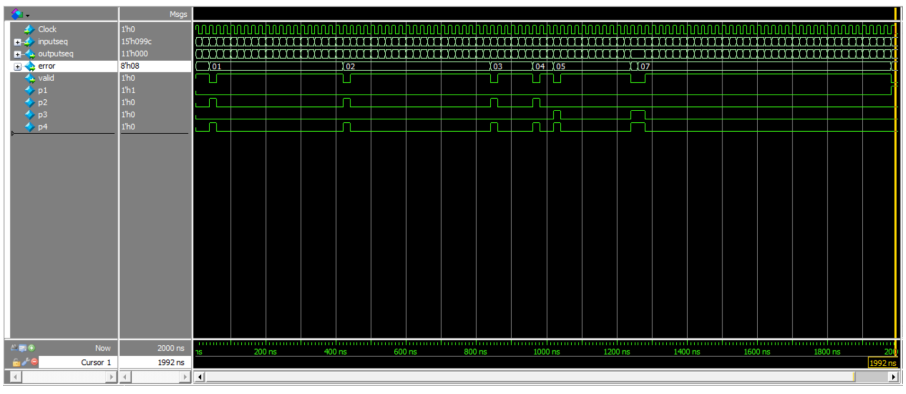
می بینیم که سیگنال valid همواره برابر ۱ است و مقدار سیگنال error هم همواره برابر صفر است. بنابراین داده ها به درستی منتقل شده اند و خطایی نداریم.

حال با استفاده از ماژول دیگری در وریلاگ به اسم receiverTester داده های خروجی را با داده های اولیه مقایسه می کنیم.می بینیم که صحت عملکرد مدار تایید می شود.

```
VSIM 3> run -all
#   0 errors detected
# ** Note: $stop    : D:/Term 6/fpga/HW/HW02/receiver_Tester.v(22)
#    Time: 0 ns  Iteration: 0  Instance: /receiver_Tester
# Break in Module receiver_Tester at D:/Term 6/fpga/HW/HW02/receiver_Tester.v line 22

VSIM 4>
```

حال به صورت دستی ۸ تا از داده ها را در فایل hamming code تغییر می دهیم.



می بینیم که سیگنال valid بعضی اوقات برابر ۰ شده و در نهایت مقدار سیگنال error برابر ۸ شده است.

```
VSIM 17> run -all
#    8 errors detected
# ** Note: $stop    : D:/Term 6/fpga/HW/HW02/receiver_Tester.v(22)
#    Time: 0 ns  Iteration: 0  Instance: /receiver_Tester
# Break in Module receiver_Tester at D:/Term 6/fpga/HW/HW02/receiver_Tester.v line 22

VSIM 18>
```

در نهایت می بینیم که تعداد اورورها ۸ تا تشخیص داده می شود.