پرهام گیلانی – 400101859

سوال 1)

الف)

FPGA تراشه ای است که با مجموعه ای از صدها هزار گیت منطقی قابل برنامه ریزی مجدد است که به صورت داخلی به یکدیگر متصل می شوند تا یک مدار دیجیتال پیچیده بسازند. اساساً این یک مدار مجتمع است که می تواند توسط کاربر برنامه ریزی شود تا منطق را ضبط کند.(توان بیشتری میگیرد)

مزایای FPGA:

1. پروتوتایپ های سریع تری دارند.
2. برای حجم تولید پایین به صرفه است.
3. انعطاف دارد.

معایب FPGA:

1. برای حجم های بالا هزینه بر است.
2. محدودیت اندازه دارد.
3. عملکرد محدودی دارد.
4. توان بیشتری میگیرد.

ASIC مخصوصا برای یک برنامه خاص یا هر هدفی ساخته شده است. اگر کسی این را با هر دستگاه دیگری مقایسه کند، سرعت آن بهبود یافته است. اساساً یک مدار مجتمع است که برای یک هدف خاص مشخص شده است. (توان کمتری میگیرد)

مزایای ASIC:

1. سرعتش از هر logic device ای بهتر است.
2. عملکرد خوبی دارد.
3. حجم فضای کمتری اشغال میکند.
4. توان کمتری میگیرد.

معایب ASIC:

1. هزینه اولیه بالایی دارد.
2. روش های تست باید توسعه پیدا کنند که ممکن است هزینه بر باشد و گاها زمان بر است.
3. قابل انعطاف نیست.

Processor برای اهداف کلی کامپیوتر طراحی شده اند تا دستورات را اجرا کنند و قابل برنامه ریزی و انعطاف پذیر هستند اما مانند ASIC و FPGA برای اهداف مشخصی طراحی نشده اند.

مزایای Processors:

1. برای اهداف کلی تری طراحی شده اند.
2. کار کردن با Processor ها از ASIC و FPGA ساده تر است.
3. طیف وسیعی از software ها و operating systemها را ساپورت میکند.

معایب Processors:

1. عملکرد انها از ASIC و FPGA کمتر است.
2. میزان پارالل سازی انها از ASIC و FPGA کمتر است.
3. توان بیشتری از ASIC و FPGA مصرف میکند.

ب) با توجه با نکات گفته شده میتوان گفت که چون تعداد گیرنده های مخابراتی 2 عدد است میتوان از FPGA برای طراحی ان استفاده کرد ولی چون تعداد تراشه دوم 2000 عدد است پس به صرفه تر است از ASIC استفاده شود.

ج) با توجه به نکات گفته شده چون زمان ارائه محصول مهم است پس باید از FPGA استفاده کرد چون طراحی ASIC زمان بر است.

د) FPGA: عموما برای complex digital designs استفاده میشوند و قابلیت انعطاق بالایی دارند و طیف وسیعی از application ها را ساپورت میکند که شامل مدارهای منطقی قابل برنامه ریزی هستند.

CPLD: انها یک Programmable logic device هستند اما تعداد کمتری logic block دارند و interconnect resource ساده تری نسبت به FPGA دارند و گاها برای کار های ساده تر ولی با زمان کمتر طراحی شده اند که به طبع ان توان کمتری مصرف میکند.

Gate Array: انها یک basic logic gate هستند که در یک ارایه در chirp مرتب شده اند و بر خلاف FGPA و CPLD قابل برنامه ریزی نیستند و چیزی در بین انعطاف پذیری FPGA ها و بازده CPLD ها هستند و گاها برای جاهایی که moderate customization مورد نیاز است استفاده میشود و جاهایی که عملکرد و توان مصرفی قابل توجه است.

ه) اجزای تشکیل دهنده FPGA:

1. Logic Blocks: دارای گیت های AND,OR,XOR,flip-flop هستند که برای پیاده سازی توابع منطقی استفاده میشوند.
2. Configurable Interconnects: این سیم ها و سوییچ ها بخش های مختلفی از Logic Block ها را به هم متصل میکنند که اجازه میدهد سیگنال به بخش های مختلف FPGA برود.
3. RAM Blocks: برای ذخیره داده درون FPGA است که یک حافظه نسبتا سریعی است.

4. DSB Slices: بلوک هایی هستند که به طور عمومی برای محاسبات ریاضی و تجذیه و تحلیل سیگنال مورد استفاده قرار میگیرد.
5. I/O Blocks: این بلوک با داستگاه های خارجی ارتباط برقرار میکند که شامل بافر های ورودی و خروجی هستند و clock management دارند.
6. Configuration Memory: این جایی است که Configuration bitstream که رفتار FPGA را تعریف می کند ذخیره می شود. می‌تواند volatile (نیاز به برنامه‌ریزی مجدد با هر بار روشن شدن FPGA) یا non-volatile (حفظ پیکربندی در طول چرخه) باشد.

سوال 2)

الف) در پاراال سازی عملا چند تا برنامه به صورت موازی با هم اجرا میشوند که با هم استقلالی ندارند که باعث سریع تر شدن سیستم میشود و سعی میکند از اکثر هسته ها استفاده کند اما در فرایند همروند هم این فرایند موازی سازی وجود دارد اما در این مورد کد های در حال اجرا (که شبیه به thread هستند) با هم ارتباط و نمیتوان به طور کلی انها را از هم مستقل کرد پس باید نحوه اجرا شدن کد ها نحوه خاصی داشته باید که به طبع خود سختی کار را دارد.

ب)

سطوح پاراال سازی:

1. Hardware Level Parallelism: در پایین ترین سطح، FPGA خود پاراال سازی ذاتی را از طریق معماری خود فراهم می کند. این شامل مسیرهای اجرای موازی در بلوک های منطقی، منابع مسیریابی موازی برای انتقال داده و قابلیت های پردازش موازی بلوک های سخت افزاری تخصصی مانند اسلایس های DSP است.
2. Task Level Parallelism: FPGA می‌تواند از موازی‌سازی در سطح task با تجزیه وظایف محاسباتی به وظایف فرعی مستقل و کوچک‌تر که می‌توانند همزمان اجرا شوند، بهره‌برداری کنند. این وظایف فرعی را می توان به صورت موازی در منابع موجود FPGA پیاده سازی کرد و توان عملیاتی و عملکرد را به حداکثر رساند.
3. Pipeline Parallelism: تکنیکی است که برای افزایش توان عملیاتی از طریق تقسیم یک کار به چند مرحله و اجرای این مراحل به صورت موازی استفاده می شود. FPGAها برای pipeline مناسب هستند و به چندین مرحله از یک محاسبات اجازه می‌دهند به طور همزمان عمل کنند و در نتیجه تاخیر کلی را کاهش دهند.

4. Data Level Parallelism: FPGA ها همچنین می توانند از موازی سازی در سطح داده با پردازش چندین عنصر داده به طور همزمان بهره برداری کنند. این می تواند شامل موازی کردن محاسبات در بردارها یا آرایه های داده، پردازش موازی جریان های داده های متعدد یا اجرای موازی عملیات در مجموعه داده های بزرگ باشد.
5. Instruction Level Parallelism: ILP شامل اجرای چندین دستورالعمل به صورت موازی در یک هسته پردازنده است. در حالی که FPGA ها CPU های سنتی نیستند، می توانند soft processor یا custom instruction set ها را پیاده سازی کنند که در آن تکنیک های ILP می تواند برای موازی سازی اجرای دستورالعمل ها اعمال شود.
6. System Level Parallelism: FPGA ها را می توان در کنار سایر عناصر پردازشی مانند CPU، GPU یا hardware accelerators ادغام کرد. موازی سازی در سطح سیستم شامل توزیع وظایف محاسباتی در بین این عناصر پردازش ناهمگن است تا از قابلیت های پردازش موازی ترکیبی آنها استفاده کند.

ج)

1. Architecture:

GPU: عمدتاً برای اجرای موازی تعداد زیادی عملیات حسابی و منطقی نسبتاً ساده طراحی شده‌اند. آنها از بسیاری از هسته‌های پردازشی کوچک تشکیل شده‌اند که در یک معماری موازی بسیار بهینه شده برای پردازش گرافیکی و کارهای محاسباتی موازی سازماندهی شده‌اند.
FPGA: دستگاه های سخت افزاری قابل تنظیم مجدد هستند که از بلوک های منطقی قابل تنظیم، اتصالات متقابل و بلوک های کاربردی تخصصی مانند اسلایس های DSP و بلوک های حافظه تشکیل شده اند. برخلاف GPU، FPGAها را می‌توان در سطح سخت‌افزار برای پیاده‌سازی مدارها یا الگوریتم‌های دیجیتال خاص طراحی کرد.

2. Programing Model:

GPU : معمولاً از چارچوب های برنامه نویسی سطح بالا مانند CUDA یا OpenCL برای برنامه ریزی محاسبات موازی استفاده می کنند. برنامه نویسان کد را به زبان هایی مانند CUDA C/C++ یا OpenCL C می نویسند که سپس در دستورالعمل های GPU کامپایل میشود.
FPGA : FPGA ها معمولاً با استفاده از زبان‌های توصیف سخت‌افزار (HDL) مانند Verilog یا VHDL برنامه‌ریزی می‌شوند که رفتار مطلوب مدارهای دیجیتالی را که باید پیاده‌سازی شوند، توصیف می‌کنند. از طرف دیگر، ابزارهای سنتز سطح بالا (HLS) به برنامه نویسان اجازه می دهد تا کد را به زبان هایی مانند C یا C++ بنویسند و به طور خودکار توضیحات سخت افزاری را برای پیاده سازی FPGA تولید کنند.

3. Flexibility & Reconfigurability:

GPU: GPU دارای معماری‌های ثابتی هستند که برای انواع خاصی از محاسبات موازی، مانند رندر گرافیکی یا دیپ لرنینگ، بهینه شده‌اند. در حالی که پردازنده‌های گرافیکی توان عملیاتی بالایی را برای وظایف خاص ارائه می‌کنند، اما فاقد انعطاف‌پذیری برای سازگاری با طیف وسیعی از برنامه‌ها بدون تنظیم مجدد هستند.

FPGA: FPGA ها بسیار قابل تنظیم هستند و می توانند برای پیاده سازی طیف گسترده ای از مدارها و الگوریتم های دیجیتال اجرا کنند. این انعطاف‌پذیری به FPGAها اجازه می‌دهد تا در برنامه‌هایی با نیازمندی‌های متغیری دارند یا در جاهایی که سرعت سخت‌افزاری بسیار تخصصی مورد نیاز است، برتری پیدا کنند.

4. Performance & Power Efficiency:

GPU: GPUها در کارهایی که نیاز به موازی سازی عظیم دارند، مانند رندر گرافیکی، عملیات ماتریسی در یادگیری ماشین، و برخی وظایف محاسباتی علمی، برتری دارند. با این حال، ممکن است در مقایسه با FPGA برای برنامه های خاص، انرژی بیشتری مصرف کنند و تاخیر بیشتری داشته باشند.

FPGA: FPGA کارایی عالی دارد که می توانند به شدت موازی شوند یا از سرعت سخت افزاری بهره مند باشند. آنها می توانند با مصرف انرژی کمتر در مقایسه با GPU ها برای کاربردهای خاص، به ویژه آنهایی که محاسبات نامنظم یا وابسته به داده دارند، به تأخیر کم و توان عملیاتی بالا دست پیدا کنند.

5. Time-to-Market & Development Complexity:

GPU: توسعه نرم افزار برای GPU ها معمولا ساده تر و سریعتر از طراحی سخت افزار برای FPGA است. با این حال، بهینه سازی کد GPU برای عملکرد و کارایی همچنان می تواند چالش برانگیز باشد، به خصوص برای الگوریتم های موازی پیچیده.

FPGA: توسعه راه حل های مبتنی بر FPGA به تخصص در طراحی سخت افزار و زبان های برنامه نویسی FPGA نیاز دارد. در حالی که FPGA ها انعطاف پذیری و پتانسیل عملکرد بیشتری را دارند، اغلب در مقایسه با برنامه نویسی GPU به زمان توسعه طولانی تر و مهارت های تخصصی بیشتری نیاز دارند.

سوال 3)

الف) سطوح مختلف زبان سخت افزار:

1. Register-Level Programming:
در این سطح، برنامه نویسان به طور مستقیم رجیسترهای سخت افزاری را برای کنترل رفتار اجزای سخت افزاری خاص مانند ثبات های CPU، I/O با حافظه و رجیسترها دستکاری می کنند. این سطح از کدنویسی به شدت وابسته به سخت افزار است و نیاز به درک عمیق معماری سخت افزاری اساسی دارد.
2. Hardware Description Languages (HDLs):
HDLهایی مانند Verilog و VHDL برای توصیف رفتار مدارها و سیستم‌های دیجیتال در سطح بالاتری استفاده می‌شوند. برنامه نویسان به جای کار با اجزای سخت افزاری مجزا، از HDL برای توصیف اتصالات و عملکرد سیستم های دیجیتال پیچیده مانند CPU، GPU و FPGA استفاده می کنند. HDL ها اغلب در طراحی و تایید سخت افزار دیجیتال استفاده می شوند.
3. High-Level Hardware Description:
این سطح شامل استفاده از زبان های برنامه نویسی سطح بالا، مانند C یا C++، برای نوشتن کدهایی است که از طریق سیستم عامل ها یا کتابخانه های سخت افزار، با سخت افزار تعامل دارد. در حالی که در سطح پایین برنامه نویسی در سطح Register یا HDL نیست، این رویکرد همچنان امکان تعامل مستقیم سخت افزاری، معمولاً از طریق درایورها یا کتابخانه های مخصوص سخت افزار را فراهم می کند.

Register Level :

```
// Example: Configuring GPIO pin on a microcontroller

// Define memory-mapped addresses for GPIO registers

#define GPIO_BASE_ADDRESS 0x40020000

#define GPIO_DIR_OFFSET 0x00

#define GPIO_DATA_OFFSET 0x04

*(volatile uint32_t*)(GPIO_BASE_ADDRESS + GPIO_DIR_OFFSET) |= (1 << 5);

*(volatile uint32_t*)(GPIO_BASE_ADDRESS + GPIO_DATA_OFFSET) |= (1 << 5);
```

HDL Level :

```
module mux2to1 (
   input wire sel,
   input wire in0,
   input wire in1,
   output reg out
);
always @(*) begin
   if (sel == 0)
      out = in0;
   else
      out = in1;
end
```

High Level :

```
#include <stdio.h> #include <stdint.h>

#define GPIO_BASE_ADDRESS 0x40020000

#define GPIO_DATA_OFFSET 0x04

void toggle_led() {

    volatile uint32_t* gpio_data = (volatile uint32_t*)(GPIO_BASE_ADDRESS + GPIO_DATA_OFFSET);

     *gpio_data ^= (1 << 5);

}

int main() {

    while (1) {toggle_led();}

     return 0;

}
```

ب) سطحی که بیشترین تعداد گیت ها در آن کد گذاری می شوند، معمولاً در HDL مانند Verilog یا VHDL است. HDLها برای توصیف رفتار و اتصال مدارهای دیجیتال در سطح پایین استفاده می شوند. در HDL ها، طراحان گیت های منطقی و نحوه اتصال آنها را برای اجرای عملکرد مورد نظر مشخص می کنند. این می تواند شامل توصیف سیستم های دیجیتال پیچیده با هزاران یا حتی میلیون ها gate جداگانه باشد. در حالی که برنامه نویسی در سطح Register و high level نیز شامل تعامل با سخت افزار می‌شود، بیشتر بر کنترل و تعامل با اجزا یا سیستم‌های بزرگ تر تمرکز می‌کنند.

سوال 4)

الف) ابزارهای سنتز در برای ترجمه توضیحات سخت افزاری سطح بالا (اغلب در HDL ها مانند Verilog یا VHDL نوشته می شود) به نمایش سطح پایین تر که می تواند بر روی دستگاه های منطقی قابل برنامه ریزی (مانند FPGA) یا مدارهای مجتمع خاص پیاده سازی شود استفاده می شود. (ASIC). این ابزارها چندین عملکرد کلیدی را انجام می دهند:

1. Translation: ابزارهای سنتز توصیف رفتاری یک مدار دیجیتال را به یک نمایش ساختاری تبدیل می‌کنند. این شامل نگاشت ساختارهای سطح بالا در کد HDL (مانند ماژول ها، توابع و سیگنال ها) به اجزای سطح پایین تر مانند گیت های منطقی، مالتی پلکسرها و فلیپ فلاپ ها است.
2. Optimization: ابزارهای سنتز مدار حاصل را برای برآوردن محدودیت های طراحی خاص مانند مساحت، توان و زمان مورد نیاز بهینه می کنند. هدف این فرآیند بهینه سازی به حداقل رساندن استفاده از منابع، کاهش مصرف انرژی و بهبود عملکرد و در عین حال حفظ عملکرد طراحی اصلی است.
3. Technology Mapping: ابزارهای سنتز اجزای فیزیکی مناسب (مانند جداول جستجو، فلیپ فلاپ‌ها و منابع مسیریابی) را از یک کتابخانه برای پیاده‌سازی انتخاب می‌کنند. این شامل نگاشت نمایش منطقی مدار بر روی منابع موجود دستگاه هدف است.
4. Timing Analysis: ابزارهای سنتز ویژگی های زمان بندی مدار سنتز شده را تجزیه و تحلیل می کنند تا اطمینان حاصل شود که محدودیت های زمان بندی مشخص شده، مانند فرکانس و حداکثر تاخیر را برآورده می کند. تجزیه و تحلیل زمان‌بندی به شناسایی critical paths و بهینه‌سازی آنها برای برآوردن الزامات عملکرد کمک می‌کند.
5. Verification: ابزارهای سنتز ممکن است شامل built-in verification features برای اطمینان از صحت طرح سنتز شده باشند. این می تواند شامل بررسی هم ارزی منطقی بین توصیف رفتاری اصلی و فهرست شبکه ترکیب شده برای تأیید اینکه فرآیند ترکیب هیچ خطایی ایجاد نکرده است باشد.

ب) Place and Route یک گام مهم در جریان طراحی است که از سنتز منطقی پیروی می کند. این شامل دو فرآیند اصلی است: Place و Route.

1. Placement:
در طول فرآیند قرار دادن، عناصر منطقی سنتز شده (مانند جداول جستجو، فلیپ فلاپ ها و سایر منابع) به مکان های فیزیکی خاصی در تراشه FPGA نگاشت می شوند. هدف این است که هر عنصر منطقی را به یک مکان مناسب اختصاص دهیم که تراکم مسیریابی را به حداقل برساند، محدودیت‌های زمان‌بندی را برآورده کند و معیارهای مختلف مانند مساحت و مصرف انرژی را بهینه کند.
2. Routing:
پس از قرار دادن، فرآیند مسیریابی، اتصالات فیزیکی (سیم یا اتصالات متقابل) بین عناصر منطقی قرار داده شده را برای اجرای توابع منطقی مورد نیاز تعیین می کند. الگوریتم مسیریابی باید اطمینان حاصل کند که همه مسیرهای سیگنال الزامات زمان بندی را برآورده می کنند، از درگیری یا تراکم جلوگیری می کنند و استفاده از منابع مسیریابی را بهینه می کنند.

فرآیند Place و Route مستقیماً بر محدودیت های زمان بندی و فرکانس طراحی FPGA به روش های زیر تأثیر می گذارد:

1. Timing Constraints:
محدودیت‌های زمان‌بندی حداکثر تاخیر مجاز برای انتشار سیگنال‌ها در طراحی FPGA را مشخص می‌کنند. این محدودیت ها شامل پارامترهایی مانند حداکثر فرکانس کلاک، زمان راه اندازی، زمان هولد و حداکثر تاخیر مسیر می باشد. در طول فرآیند Place و Route، ابزارها Placement و Routing را بهینه می‌کنند تا اطمینان حاصل شود که این محدودیت‌های زمانی برآورده شده‌اند. نقض محدودیت‌های زمان‌بندی می‌تواند منجر به خطاهای عملکردی یا عملکرد غیرقابل اعتماد طراحی FPGA شود.
2. Frequency Limitations:
حداکثر فرکانس کلاک قابل دستیابی یک طراحی FPGA توسط critical path تعیین می شود که طولانی ترین مسیر از طریق عناصر منطق ترکیبی بین دو فلیپ فلاپ است. در طول مکان و مسیر، ابزارها سعی می‌کنند با بهینه سازی placement و Routing، تاخیر critical path را به حداقل برسانند. با این حال، ویژگی‌های فیزیکی دستگاه FPGA، مانند تاخیرهای Routing و interconnect capacitance، محدودیت‌های عملی را بر فرکانس کلاک قابل دستیابی تحمیل می‌کند. برآوردن محدودیت‌های زمان‌بندی و بهینه‌سازی critical path برای دستیابی به فرکانس عملیاتی مطلوب طراحی FPGA ضروری است.

ج) تفاوت ها:

1. Functionality:

- Hardware Description Languages (HDLs):
HDL ها مانند Verilog و VHDL به طور خاص برای توصیف رفتار و ساختار سخت افزار دیجیتال طراحی شده اند. آنها به طراحان اجازه می دهند تا منطق، اتصالات و زمان بندی مدارهای دیجیتال را مشخص کنند. HDL ها در طراحی، شبیه سازی و سنتز قطعات سخت افزاری دیجیتال مانند مدارهای مجتمع (IC)، FPGA و ASIC استفاده می شوند.
- Other Programming Languages:
زبان های برنامه نویسی همه منظوره مانند C، Python و Java برای نوشتن برنامه های نرم افزاری برای اجرا بر روی CPU یا سایر پلت فرم ها طراحی شده اند. این زبان ها برای توسعه طیف گسترده ای از برنامه های نرم افزاری، از جمله سیستم عامل ها، درایورها، وب و برنامه های تلفن همراه استفاده می شوند.

2. Implementation:

- Hardware Description Languages (HDLs):
کد HDL معمولاً در پیاده‌سازی‌های سخت‌افزاری مانند ASIC یا FPGA سنتز می‌شود. فرآیند سنتز شامل ترجمه کد HDL سطح بالا به یک لیست شبکه ای از گیت ها و اتصالات منطقی است که می تواند سپس به اجزای سخت افزار فیزیکی نگاشت شود.
- Other Programming Languages:
کد نوشته شده در زبان های برنامه نویسی معمولاً توسط ابزارهای توسعه نرم افزار به machine code یا intermediate representations که می توانند توسط CPU یا virtual machines اجرا شوند، کامپایل یا تفسیر می شوند. اجرای برنامه های نرم افزاری شامل اجرای کدهای کامپایل شده بر روی یک پلت فرم محاسباتی مانند کامپیوتر، سرور یا دستگاه تلفن همراه است.

د) در کل 2 معماری برای پیاده سازی توابع و مدارهای منطقی استفاده میشود.

1. LUT-Based Architecture:

- LUT (Lookup Table) : بلوک اصلی ساختارهای FPGA مبتنی بر LUT جدول جستجو است. جدول جستجو یک جزء حافظه است که truth table یک تابع منطقی را ذخیره می کند. به طور معمول، LUT ها در FPGA به عنوان سلول های SRAM (Static Random-Access Memory) پیاده سازی می شوند.
- Functionality: معماری های مبتنی بر LUT توابع منطقی را با ذخیره ورودی های truth table تابع در LUT ها پیاده سازی می کنند. هر LUT معمولاً دارای چندین ورودی و یک خروجی واحد است. هنگامی که یک الگوی ورودی به LUT اعمال می شود، مقدار خروجی مربوطه را از حافظه خود بازیابی می کند.
- Implementation: برای پیاده سازی یک تابع منطقی در یک FPGA مبتنی بر LUT، ابزارهای سنتز تابع منطقی را به تعداد مناسب LUT ها ترسیم می کنند و محتویات LUT را برای مطابقت با truth table مورد نظر پیکربندی می کنند. اتصالات بین LUT و سایر منابع FPGA برای مسیریابی سیگنال ها و اتصالات بر اساس الزامات طراحی پیکربندی شده اند.

2. MUX-Based Architecture:

- MUX (Multiplexer): در معماری های FPGA مبتنی بر MUX، مالتی پلکسرها بلوک های اصلی ساختمان هستند. مالتی پلکسر یک جزء منطقی ترکیبی است که یکی از سیگنال های ورودی متعدد را انتخاب کرده و بر اساس سیگنال انتخابی به خروجی هدایت می کند.
- Functionality: معماری های مبتنی بر MUX توابع منطقی را با استفاده از مالتی پلکسرها برای انتخاب سیگنال های ورودی مناسب بر اساس الگوهای ورودی پیاده سازی می کنند. هر مالتی پلکسر دارای چندین ورودی داده، یک یا چند ورودی انتخابی و یک خروجی واحد است. با کنترل ورودی های انتخاب، سیگنال ورودی مورد نظر به خروجی هدایت می شود.
- Implementation: برای پیاده سازی یک تابع منطقی در یک FPGA مبتنی بر MUX، ابزارهای سنتز تابع منطقی را به تعداد مناسب مالتی پلکسر ترسیم می کنند و سیگنال های انتخاب مالتی پلکسر را برای پیاده سازی رفتار منطقی مورد نظر پیکربندی می کنند. منابع مسیریابی FPGA برای اتصال ورودی و خروجی مالتی پلکسرها با توجه به الزامات طراحی استفاده می شود.

نحوه استفاده در اجرا:

1. LUT-Based Implementation:
   - توابع منطقی مورد نظر را با استفاده از HDL مانند Verilog یا VHDL تعریف کنید.
   - از ابزارهای سنتز برای ترسیم توابع منطقی به LUT ها و ایجاد یک netlist استفاده کنید.
   - Place-Route را برای قرار دادن LUT ها در منابع FPGA و مسیریابی اتصالات انجام دهید.
   - FPGA را با synthesized bitstream برنامه ریزی کنید.

2. MUX-Based Implementation:
   - توابع منطقی مورد نظر را با استفاده از HDL تعریف کنید.
   - از ابزارهای سنتز برای ترسیم توابع منطقی به مالتی پلکسرها و ایجاد یک لیست شبکه استفاده کنید.
   - Place-Route را برای قرار دادن مالتی پلکسرها در منابع FPGA و مسیریابی اتصالات انجام دهید.
   - FPGA را با synthesized bitstream برنامه ریزی کنید.

هر دو معماری مبتنی بر LUT و مبتنی بر MUX به طور گسترده در طراحی های FPGA استفاده می شوند و انتخاب بین آنها به عواملی مانند پیچیدگی توابع منطقی، استفاده از منابع و الزامات عملکرد طراحی بستگی دارد.

سوال 5) ماژولی به اسم count_down تعریف کردم که ورودی های کلاک و ریست و دقیقه و ثانیه را میگیرد و در هر مرحله 1 ثانیه کم میکند تا به صفر برسد و در هر مرحله زمان را نمایش دهد و وقتی که 0 شد خروجی ready را 1 میکند. برای تست بنچ هم ورودی ها را ست کردم و چک کردم اگر ready بود و ریست نبود برنامه را تمام کند.

Code:

```
module count_down (
  input clk, reset,
  input [5:0] seconds, minutes,
  output reg ready,
  output reg [5:0] current_minutes, current_seconds );
  reg [25:0] counter;
  always @(posedge clk) begin
    if(reset == 1'b0) begin
      current_minutes = minutes;
      current_seconds = seconds;
      counter = 0;
      ready = 0;
      $display("reset");
    end if (counter == 10000000) begin
      counter = 0;
      if (current_seconds > 0) begin
        current_seconds <= current_seconds - 1;
        $display(current_minutes," : ",current_seconds);
      end else if (current_minutes > 0) begin
        current_minutes <= current_minutes - 1;
        current_seconds <= 59;
        $display(current_minutes," : ",current_seconds);
      end if(current_seconds == 0 && current_minutes == 0) begin
        $display(current_minutes," : ",current_seconds);
        ready <= 1;
      end
    end
    counter = counter + 1;
  end
  initial begin
    current_minutes <= minutes;
    current_seconds <= seconds;
    counter = 0;
    ready <= 0;
  end
endmodule
```

TestBench:

```
module count_down_tb;
  reg clk ,reset;
  reg [5:0] seconds, minutes;
  wire [5:0] current_minutes, current_seconds;
  integer reset_times;
  count_down ut (
    .minutes(minutes),
    .seconds(seconds),
    .clk(clk),
    .current_minutes(current_minutes),
    .current_seconds(current_seconds),
    .ready(ready),
    .reset(reset) );
  always @(clk) begin
    reset = 1'b1;
    if(ready) begin
      if(reset_times != 0) begin
        reset_times = reset_times - 1;
        reset = 1'b0;
      end else $stop;
    end
    clk <= #10 ~clk;
  end
  initial begin
    minutes = 6'd1; // 1 minutes
    seconds = 6'd7; // 7 seconds
    clk = 1'b1;
    reset = 1'b1;
    reset_times = 1;
    $display("starting time at ",minutes," : ",seconds," with",reset_times,"times of reset");
  end
endmodule
```

سوال 7) در این سوال صرفا یک شیفت رجیستر نوشتم که قابلیت لود ورودی و clear کردن و شیفت به چپ و راست را دارد. برای تست بنچ نیز صرفا داخل initial یک لوپ 10 تایی زدم تا به ازای مقادیر رندم تست کند و نمایش دهد.

Code:

```
`timescale 1ns/1ns
module shift_register(
   input [3:0] input_data,
   input [1:0] control,
   input clk, input_bit, clear,
   output reg [3:0] output_data);


always @(posedge clk) begin
   if(clear) begin
      output_data <= 4'h0;
   end if (control == 2'b00) begin
      output_data <= input_data;
   end else if (control == 2'b01) begin
      output_data <= {input_bit, output_data[3:1]};
   end else if (control == 2'b10) begin
      output_data <= {output_data[2:0],input_bit};
   end
end
endmodule
```

TestBench:

```
`timescale 1ns/1ns
module shift_register_tb();
reg clk = 1'b1;
always @(clk)
   clk <= #10 ~clk;
reg [3:0] input_data;
reg [1:0] control;
reg input_bit,clear;
integer i;
initial begin
   clear = 1;
   control = 2'b11;
   input_data = 4'b0;
   input_bit = 0;
   @(posedge clk);
   @(posedge clk);
   $display("Value in output_data: %b",ut.output_data);
   for (i = 0; i < 10 ; i = i+1) begin
      input_data = $random%20;
      control = $random%2;
      input_bit = $random%2;
      clear = $random%10;
      @(posedge clk);
      @(posedge clk);
      $display("Value in output_data: %b",ut.output_data);
   end
   $stop;
end
shift_register ut (
   .input_data(input_data),
   .control(control),
   .input_bit(input_bit),
   .clk(clk),
   .clear(clear),
   .output_data() );
endmodule
```

سوال 6) در این سوال هم روی op کیس زدم تا اپریشن های مورد نظر را ببینم و برای تقسیم هم یک ماژول divider تعریف کردم که هر بار مقدار ان حساب میشود و اگر تقسیم بود اختصاص میدهم.

Code:

```
module ALU(
  input [3:0] A, B,
  input [1:0] op,
  input clk,
  output reg [7:0] result );
  wire [3:0] Q;
  Divider divi(A, B, clk, Q);
  always @(posedge clk) begin
    case (op)
      2'b00 : result = A+B;
      2'b01 : result = A-B;
      2'b10 : result = A*B;
      2'b11 : result = {4'b0 , Q};
       default: result = 8'b0;  endcase
  end
endmodule
module Divider (
  input [3:0] A, B,
  input clk,
  output reg [3:0] Q );
  reg [3:0] Acopy;
  always @(posedge clk ) begin
    Acopy <= A;
    Q = 4'b0;
     while (Acopy >= B) begin
       Q = Q + 1;
       Acopy = Acopy - B;
     end
  end
endmodule
```

Testbench:

```
module ALU_tb;
reg [3:0] A, B;
reg [1:0] op;
reg clk = 1'b1;
wire [7:0] result;
always @(clk) clk <= #10 ~clk;
ALU ut (
 .A(A),
 .B(B),
 .clk(clk),
 .op(op),
 .result(result) );
initial begin
 A = 4'b1110;
 B = 4'b0110;
 op = 2'b00; // Addition
 #100 $display("%d (%b) + %d (%b) = %d (%b)", A, A, B, B, result, result);
 A = 4'b1110;
 B = 4'b0110;
 op = 2'b01; // Subtraction
 #100 $display("%d (%b) - %d (%b) = %d (%b)", A, A, B, B, result, result);
 A = 4'b1110;
 B = 4'b0110;
 op = 2'b10; // Multiplication
 #100 $display("%d (%b) * %d (%b) = %d (%b)", A, A, B, B, result, result);
 A = 4'b1110;
 B = 4'b0110;
 op = 2'b11; // Division
 #100 $display("%d (%b) / %d (%b) = %d (%b)", A, A, B, B, result, result);
 $stop;
end
endmodule
```