DAILY ASSESSMENT FORMAT

|  |  |  |  |
| --- | --- | --- | --- |
| Date: | 1/06/2020 | Name: | Akshay |
| Course: | DIGITAL DESIGN USING HDL | USN: | 4AL17EC008 |
| Topic: | FPGA and ASIC | Semester & Section: | 6TH & A |
| Github Repository: | Akshay-Online-Course |  |  |

|  |
| --- |
| FORENOON SESSION DETAILS |
|  |
| Report:  What is an FPGA?  An FPGA is a (mostly) digital, (re-)configurable ASIC.  I say mostly because there are analog and  mixed-signal aspects to modern FPGAs.  For example, some have A/D converters and PLLs.  I  put re- in parenthesis because there are actually one-time-programmable FPGAs, where once you  configure them, that’s it, never again.  However, most FPGAs you’ll come across are going to be re-  configurable.  So what do I mean by digitally configurable ASIC?  I mean that at the core of it, you’re designing a digital logic circuit, as in AND, OR, NOT, flip-flops,  etc.  Of course that’s not entirely accurate and there’s much more to it than that, but that is the gist at  its core.  1. Parallel processes – if you need to process several input channels of information (e.g. many  simultaneous A/D channels) or control several channels at once (e.g. several PID loops).  2. High data-to-clock-rate-ratio – if you’ve got lots of calculations that need to be executed  over and over and over again, essentially continuously. The advantage is that you’re not tying  up a centralized processor. Each function can operate on its own.  3. Large quantities of deterministic I/O – the amount of determinism that you can achieve  with an FPGA will usually far surpass that of a typical sequential processor. If there are too  many operations within your required loop rate on a sequential processor, you may not even  have enough time to close the loop to update all of the I/O within the allotted time.  4. Signal processing – includes algorithms such as digital filtering, demodulation, detection  algorithms, frequency domain processing, image processing, or control algorithms.  5. Complex calculations infrequently – If the majority of your algorithms only need to make a  computation less than 1% of the time, you’ve generally still allocated those logic resources  for a particular function (there are exceptions to this), so they’re still sitting there on your  FPGA, not doing anything useful for a significant amount of time.  6. Sorting/searching – this really falls into the category of a sequential process. There are  algorithms that attempt to reduce the number of computations involved, but in general, this is  a sequential process that doesn’t easily lend itself to efficient use of parallel logical resources.  Check out the sorting section here and check out this article here for some more info.  7. Floating point arithmetic – historically, the basic arithmetic elements within an FPGA have  been fixed-point binary elements at their core. In some cases, floating point math can be  achieved (see Xilinx FP Operator and Altera FP White Paper ), but it will chew up a lot of  logical resources. Be mindful of single-precision vs double-precision, as well as deviations  from standards. However, this FPGA weakness appears to be starting to fade, as hardened  floating-point DSP blocks are starting to be embedded within some FPGAs (see Altera Arria  10 Hard Floating Point DSP Block).  8. Very low power – Some FPGAs have low power modes (hibernate and/or suspend) to help  reduce current consumption, and some may require external mode control ICs to get the most  out of this. Check out an example low power mode FPGA here. There are both static and  dynamic aspects to power consumption. Check out these power estimation spreadsheets to  start to get a sense of power utilization under various conditions. However, if low power is  critical, you can generally do better power-wise with low-power architected microprocessors  or microcontrollers.  9. Very low cost – while FPGA costs have come down drastically over the last decade or so,  they are still generally more expensive than sequential processors.  What’s inside – Core components (or at least what everyone likes to think about):  LUT (Look-Up Table) –  The name LUT in the context of FPGAs is actually misleading, as it doesn’t convey the full power of  this logical resource.  The obvious use of a LUT is as a logic lookup table (see  examples here and here), generally with 4 to 6 inputs and 1 to 2 outputs to specify any logical  operation that fits within those bounds.  There are however two other common uses for a LUT:  1. LUT as a shift register – shift registers are very useful for things like delaying the timing of an  operation to align the outputs of one algorithm with another. Size varies based on FPGA.  2. LUT as a small memory – you can configure the LUT logic as a VERY small volatile random-access  memory block. Size varies based on FPGA  FF (Flip-flop) –  Flip-flops store the output of a combinational logic calculation.  This is a critical element in FPGA  design because you can only allow so much asynchronous logic and routing to occur before it is  registered by a synchronous resource (the flip-flop), otherwise the FPGA won’t make timing.  It’s the  core of how an FPGA works.  Flip-flops can be used to register data every clock cycle, latch data, gate off data, or enable signals.  Block Memory –  It’s important to note that there are generally several types of memory in an FPGA.  We mentioned  the configuration of a LUT resource.  Another is essentially program memory, which is intended to  store the compiled version of the FPGA program itself (this may be part of the FPGA chip or as a  separate non-volatile memory chip).  What we’re referring to here though, is neither of those types of  memory.  Here we’re referring to dedicated blocks of volatile user memory within the FPGA.  This  memory block is generally on the order of thousands of bits of memory, is configurable in width and  depth, and multiple blocks of memory can be chained together to create larger memory elements.  They can generally be configured as either single-port or dual-port random access, or as a FIFO.  There will generally be many block memory elements within an FPGA.  Multipliers or DSP blocks –  Have you ever seen the number of digital logic resources that it takes to create a 16-bit by 16-bit  multiplier?  It’s pretty crazy, and would chew through your logical and routing resources pretty  quickly.  Check out the 2-bit by 2-bit example here:   FPGA vendors solve this problem with  dedicated silicon to lay down something on the order of 18-bit multiplier blocks.  Some architectures  have recognized the utility of digital signal processing taking place, and have taken it a step further  with dedicated DSP (Digital Signal Processing) blocks, which can not only multiply, but add and  accumulate as well.  I/O (Input/Output) –  If you’re going to do something useful with an FPGA, you generally have to get data from and/or  provide data outside the FPGA.  To facilitate this, FPGAs will include I/O blocks that allow for  various voltage standards (e.g. LVCMOS, LVDS) as well as timing delay elements to help align  multiple signals with one another (e.g. for a parallel bus to an external RAM chip).  Clocking and routing –  This is really a more advanced topic, but critical enough to at least introduce.  You’ll likely use an  external oscillator and feed it into clocking resources that can multiply, divide, and provide phase-  shifted versions of your clock to various parts of the FPGA.  Routing resources not only route your clock to various parts of the FPGA, but also your data.  Routing resources within an FPGA are one of the most underappreciated elements, but so critical.  Check out this sea of madness:  a verilog code to implement NAND gate in all different styles  Gate Level modeling  module NAND\_2(output Y, input A, B);  wire Yd;  and(Yd, A, B);  not(Y, Yd);  endmodule;  Data flow modeling  module NAND\_2\_data\_flow (output Y, input A, B);  assign Y = ~(A &amp; B);  endmodule  Behavioral Modeling  module NAND\_2\_behavioral (output reg Y, input A, B);  always @ (A or B) begin  if (A == 1&#39;b1 &amp; B == 1&#39;b1) begin  Y = 1&#39;b0;  end  else  Y = 1&#39;b1;  end  endmodule |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date:1/6/2020 |  | Name: Akshay |  | |
| Course:PYTHON |  | USN:4AL17EC008 |  | |
|  |  |  |  | |
| Topic: Web Scraping |  | Semester & Section:6TH A SEC |  | |
| AFTERNOON SESSION DETAILS | | | |
| Image of session | | | |
| Report – Report can be typed or hand written for up to two pages.  Web scraping is an automated method used to extract large amounts of data from websites. The data on the websites are unstructured. Web scraping helps collect these unstructured data and store it in a structured form. There are different ways to scrape websites such as online Services, APIs or writing your own code. In this article, we’ll see how to implement web scraping with python.   * **Ease of Use:** Python is simple to code. You do not have to add semi-colons “;” or curly-braces “{}” anywhere. This makes it less messy and easy to use. * **Large Collection of Libraries:** Python has a huge collection of libraries such as [Numpy](https://www.edureka.co/blog/python-numpy-tutorial/" \t "_blank), [Matlplotlib](https://www.edureka.co/blog/python-matplotlib-tutorial/" \t "_blank), [Pandas](https://www.edureka.co/blog/python-pandas-tutorial/) etc., which provides methods and services for various purposes. Hence, it is suitable for web scraping and for further manipulation of extracted data. * **Dynamically typed:** In Python, you don’t have to define datatypes for variables, you can directly use the variables wherever required. This saves time and makes your job faster. * **Easily Understandable Syntax:** Python syntax is easily understandable mainly because reading a Python code is very similar to reading a statement in English. It is expressive and easily readable, and the indentation used in Python also helps the user to differentiate between different scope/blocks in the code. * **Small code, large task:** Web scraping is used to save time. But what’s the use if you spend more time writing the code? Well, you don’t have to. In Python, you can write small codes to do large tasks. Hence, you save time even while writing the code. * **Community:** What if you get stuck while writing the code? You don’t have to worry. Python community has one of the biggest and most active communities, where you can seek help from.   When you run the code for web scraping, a request is sent to the URL that you have mentioned. As a response to the request, the server sends the data and allows you to read the HTML or XML page. The code then, parses the HTML or XML page, finds the data and extracts it.  To extract data using web scraping with python, you need to follow these basic steps:   1. Find the URL that you want to scrape 2. Inspecting the Page 3. Find the data you want to extract 4. Write the code 5. Run the code and extract the data   As we know, Python is used for various applications and there are different libraries for different purposes. In our further demonstration, we will be using the following libraries:   * **Selenium**:  Selenium is a web testing library. It is used to automate browser activities. * **BeautifulSoup**: Beautiful Soup is a Python package for parsing HTML and XML documents. It creates parse trees that is helpful to extract the data easily. * **Pandas**: Pandas is a library used for data manipulation and analysis. It is used to extract the data and store it in the desired format. | | | |