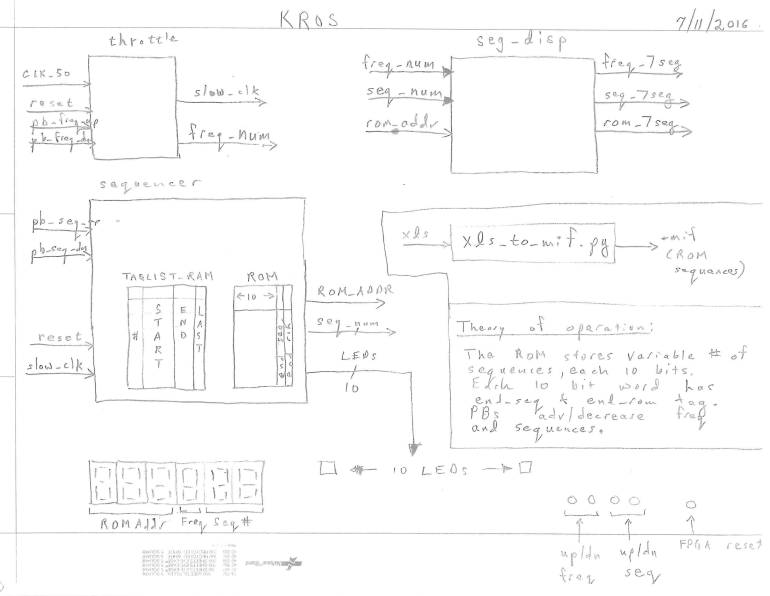
KROS (Knight Rider on Steroids Project)

|  |  |  |
| --- | --- | --- |
| 1/17/2018 | Larry Landis | First rev of write-up |
|  |  |  |

Knight Riders on Steroids (KROS) is a maker space style project that focuses on Verilog RTL design using a Terasic DE10-Lite (or equivalent board).



Features:

1. This project is an LED light sequencer that can light between 1 and 10 LEDs simultaneously. The sequencing is up to the user, and can hold up to 99 sequences. Sequences are stored in ROM.
2. There are 6 7 segment displays HEX0 – HEX 5.
   1. Sequence # (HEX 0 and 1)
   2. Frequency of sequence changing state (HEX 2)
   3. ROM Address (HEX 3-5)
3. There are two push buttons and 10 switches. If SW0 = 0 KEY1 controls sequence up and KEY 0 controls sequence down. If SW0 = 1 KEY1 controls frequency up and KEY 0 controls frequency down.
4. Upon power up the sequencer should display sequence 1 at the lowest frequency.
5. If SW1 = 1 KEY0 should reset the system

Hints on coding up KROS

1. You need a dual port RAM and a dual port ROM. Dual port means two addresses, and an address can have a read, write or both for the RAM and the ROM only has a read associated with each address. The ROM stores the sequences. The RAM stores an index of what is in the ROM.
2. Note that two extra bits are added to the ROM word – end of sequence and last sequence. You need these two extra bits to circulate the address through the sequence and when your address is at the end of sequence you need to start at the beginning of the sequence. The ROM is dual ported, upon power up its read @ 50 MHz. The other port uses the divided clock.
3. The dual port RAM is a pointer to the sequences. Upon power up a high speed clock (50 MHz) reads the ROM and fills up the TAGRAM. See the block diagram on suggestions for what to store in the taglist\_RAM. You can do your own fields, this is just a suggestion.
4. Two state machines are required – the taglist state machine the ROM state machine. Think about how to design these carefully with state diagrams and before coding them up.
5. The push buttons need to be debounced else they can send more than one pulse. Do a search on Verilog debounce circuits you fill find dozens. The debounce circuit filters out unwanted glitches.
6. Use as much prebuilt IP as you can find by search or through the parameterized modules in Quartus IP Catalog. Modules like counters are convenient to use from the library. Try and write as little original code as possible.
7. Go through the warnings in <design>.map.rpt and <design>.fit.rpt and see if you can remove as many as possible. Avoid latches!

Teamwork Considerations

1. This is a team project. Decide who does what and communicate constantly on progress.
2. The ROM state machine sequencer is the most complex module to design. The Taglist state machine is the next most complicated module to design. Simulate them thoroughly and think about corner cases. Make sure your team mates understand what signals they are being sent.
3. You need to store the sequences in a ROM. You might consider making an excel to MIF converter using a Python script. Or just edit the MIF, your choice. The converter script will up your skills in Python coding.
4. Agree on sensible signal names up front. Many teams use \_n at the end for active low signals. Decide in the beginning, agree and stay consistent. Verilog is case sensitive, decide on signal names early – the ones that cross module boundaries and your partners need to route to their modules. Everything you name - modules, wires, IO should be sensible and easy to understand from other team members. Don’t use positional instantiations. Here is a decent write-up on coding style, I suggest using this. <https://github.com/NetFPGA/netfpga/wiki/VerilogCodingGuidelines> . Also agree on formatting of indentation (I don’t like tabs, makes the code run off the page – 2 or 3 spaces is good.
5. It’s a good idea to have a top level integrator / build master who keeps tabs of the overall project state and labelling revisions.
6. Add a ton of comments, don’t be stingy on comments.
7. Perform design reviews on each other’s code. Even though it’s not super complex, it’s a good habit to get into.
8. Consider using git for a depository. Another good habit to get into and will help manage files better. Align on a usage model and stick with it, enforce it. Keep each other accountable for consistent usage. Don’t lose track of checkpointed good revisions by continually overwriting code. Use revision control. I suggest finding some articles on best git usage.

Bonus Projects

1. Connect the output to a VGA monitor. You will need to design or find (preferred) a VGA controller and mimic what you see on the LEDs and send to a VGA screen. The formatting of image will take some time to code.
2. Use system console to create a widget that bolts on to the KROS module and displays the values of the HEX displays and LEDs in a TCL widget.
3. Add audio. Each LED can be associated with a tone. You will need to store a digitized sine wave in memory (or perhaps calculate on the fly) and read out the tone values, send them through a digital to analog conversion and off to a speaker. Or perhaps there are components that do this for you. There are several ways to do this – look at Arduino articles or port the project to a Terasic devkit with an audio jack (DE1-SoC) and look at reference designs on how to get sounds out of the board. I know the fpgauniversity.intel.com site has some projects in this realm.