RMC75E FPGA Test Bench System

Project: RMC75E FPGA TEST BENCH
Author: Satchel Hamilton
Company: Delta Motion
Date: 5/11/2023
Last updated: August 14th, 2023

Contents

1. Introduction

This document provides detailed information about the RMC75E Test Bench. It is intended for internal use to aid in development, debugging, and future enhancements.

2. Device Overview

The RMC75E is a high-performance modular motion controller designed for smaller applications. It can handle one or two control axes and offers a variety of communication types, including Ethernet/IP, PROFINET, Modbus/TCP, and more. It's part of Delta's RMC series, all of which use Delta's RMCTools software for programming.

The RMC70, which includes the RMC75E, consists of a base module and up to four expansion modules. The base module is factory-configured and includes the CPU module and the Axis module. Each factory-installed Axis module is available with one or two control axis interfaces.

Includes:

- Position, Pressure, or Force Control
- Position-Pressure and Position-Force Control
- Full PID loop control with velocity and acceleration feed forwards
- S-Curve and Trapezoidal moves

FPGA:

- Formerly Xilinx Spartan
- Now Smartfusion IGLOO2
    - Family: IGLOO2
    - Die: M2GL005
    - Package: 256 VF
    - Speed: STD
    - Core voltage: 1.2
    - Range: COM
    - Default I/O: LVCMOS 3.3
    - PLL supply voltage: 3.3

Other Hardware Components:

- MPC5200 - 32-bit Microcontroller
- A24C01A – 1K Serial EEPROM Device
- ADS8320 - 16-bit Analog-to-Digital Converter
- 74HCT595 – 8-bit Shift Counter

3. Problem Statement and Recommended Solution

Problem Statement:

The RMC75E motion controller system has undergone a rapid design process which has resulted in insufficient testing of the device. Certain states within the device might not be initialized correctly which can lead to incorrect or unpredictable behavior. Furthermore, there is a concern that some signals might not be resolved properly and may "wander off into the weeds," essentially leading to undefined or unpredictable system behavior. This lack of rigorous testing and potential for system instability poses a risk to the functionality and reliability of the RMC75E device in its intended applications.

Recommended Solution:

Implement a comprehensive and systematic testing process, including creating a comprehensive test bench in VHDL, running thorough simulations using ModelSim, debugging and fixing any identified issues, iterative testing, and thorough documentation.

4. Simulating Test Bench with Libero and ModelSim

- In our approach to testing and simulating the RMC75E FPGA Test Bench, we will be employing two highly regarded software tools: Libero SoC and ModelSim.

- Libero SoC, a product of Microsemi and now a part of Microchip, is a comprehensive software suite designed for FPGA design flow. It provides a robust and flexible platform for designing, simulating, and debugging FPGA systems, equipped with a wide array of tools for synthesis, placement, routing, and verification.

- ModelSim, on the other hand, is a multi-language simulation environment that supports VHDL, Verilog, and SystemC. It is renowned for its performance and accuracy and is widely used in both academia and industry for simulating hardware designs prior to fabrication. It enables us to simulate and debug the behavior of our design at both the hardware and software levels, thus allowing us to ensure that our system operates as intended.

- By leveraging the capabilities of both Libero SoC and ModelSim, we can effectively simulate the RMC75E FPGA Test Bench, assess its performance, and carry out necessary debugging and refinements before the final implementation.

5. System Architecture Overview

- The RMC75E motion controller system is built upon a distributed architecture that strategically integrates multiple key components to achieve efficient and precise motion control. This architecture includes the CPU module, Axis modules, and Expansion modules.

- The CPU module serves as the central processing unit of the system, managing communications, executing control algorithms, and interfacing with the RMCTools software.

- The Axis modules, available in several configurations depending on the number of control axes and the type of feedback interface, are responsible for handling the specific control tasks related to each motion axis.

- The Expansion modules further enhance the system's capabilities by providing additional inputs and outputs. They are available in various types, including discrete I/O, analog inputs with pressure control and limit, analog inputs, and quadrature input modules.

- Together, these components form a robust and adaptable system architecture that can cater to a variety of motion control needs.

6. System Architecture Overview

6. Axis Module

There are several types of Axis modules for the RMC70 series:

1. AA1 and AA2: These modules have 1 and 2 axes respectively, with an analog feedback interface (16 bits, ±10V or 4-20mA) and an analog control output (±10V, 16-bit).

2. MA1 and MA2: These modules have 1 and 2 axes respectively, with an MDT (Start/Stop or PWM) and SSI feedback interface, and an analog control output (±10V, 16-bit).

3. QA1 and QA2: These modules have 1 and 2 axes respectively, with a quadrature feedback interface (5 V differential) and an analog control output (±10V, 16-bit).

For providing a current output, Delta's VC2124 voltage-to-current converter can be used with these Axis modules.

7. Expansion Module

1. EXP70-D8 (Discrete I/O): This module has eight discrete I/O that are individually configurable for any combination of inputs or outputs. The inputs and outputs are 12-24 VDC rated, polarity independent, and optically isolated from the controller.

2. EXP70-AP2 (Analog Inputs with Pressure Control and Pressure Limit): This module offers two differentials: ±10 V or 4-20 mA analog inputs for use in position–pressure or position–force control axes. The inputs have a 16-bit resolution and are optically isolated from the controller.

3. EXP70-A2 (Analog Inputs): This module offers two differentials: ±10 V or 4-20 mA analog reference inputs. The inputs have a 16-bit resolution and are optically isolated from the controller.

4. EXP70-Q1 (Quadrature Input): This module adds one 5 V differential quadrature input for position reference feedback and a high-speed input for homing or registration. A maximum of two Q1s can be added per unit.

8. CPU Module

- Ethernet communications for seamless connectivity
- USB port for high-speed connection to RMCTools software
- Supports retentive variables for data persistence across power cycles
- Expanded memory capacity for storing plots, programs, and curves
- Faster loop times for improved control performance and responsiveness
- Advanced 64-bit internal processor architecture
- Accurate and precise mathematical calculations

9. Programming and Usage

The RMC75E motion controller utilizes Delta's RMCTools software for programming. Users have access to a wide range of tools and features within this software suite, which includes a powerful programming environment, graphical system setup, tuning wizards, diagnostics, plot manager, and user programs for advanced control strategies.

RMCTools provides a visual, intuitive way to configure and program the system. Users can leverage the software's drag-and-drop functionality, which makes it easy to set up the system architecture, define motion control parameters, and create complex control strategies.

The RMC75E also supports a variety of communication protocols, including Ethernet/IP, PROFINET, and Modbus/TCP, for seamless integration with a range of other devices and systems.

10. Testing and Debugging

Testing and debugging the RMC75E FPGA Test Bench involves running a comprehensive test bench in VHDL and using ModelSim for simulations. It is recommended to follow a systematic approach in testing, starting with individual components (axis modules, expansion modules, and CPU modules) and gradually moving to the system level.

ModelSim allows the visualization of simulation results, which aids in identifying and resolving potential issues. Any problems that arise during these simulation runs should be thoroughly documented and addressed promptly to ensure the stability and reliability of the RMC75E device.
Debugging involves isolating any problematic behaviors, analyzing the cause, and implementing corrective measures. It's crucial to validate these fixes by rerunning the tests to confirm the issue has been resolved.

11. Future Development and Maintenance

In terms of future development, potential enhancements for the RMC75E might include expanded communication protocol support, new functionalities for the RMCTools software, or more advanced control strategies.

Maintenance of the RMC75E involves regular software updates to ensure the controller's performance and security. Additionally, routine hardware inspection and testing should be conducted to identify and rectify any emerging issues before they affect the device's operation.

Lastly, proper documentation should be maintained for all updates, bug fixes, and modifications made to the RMC75E. This will not only help to keep track of the changes but also provide valuable information for any future development work.

1. Notes

The following are my daily notes documenting my progress on building out the RMC75E Test Bench.

5/18

- Ten days in, mostly been keeping notes in my notepad.
- After a fair amount of fighting with Libero and ModelSim and otherwise learning to swim, I feel like I'm starting to get a handle on things.
- Spent most of yesterday running through ModelSim tutorials, and it seems to really be paying off.
- There does not actually seem to be much need for me to use Libero at the moment, as ModelSim is fully capable of running the test benches with the DUT.
- Using a bottom-up approach and writing test benches for the sub modules with the least dependencies seems to be working well.
- Slowly working my way up the design hierarchy.
- As of today, I have written and ran the following test benches:
    - tb_analog.vhd
    - tb_analog_v2.vhd
    - tb_mdssiroute.vhd
    - tb_ram128x16bits.vhd
    - tb_LatencyCounter.vhd
- I've written quite a few more, but I don't think those will be usable.
- Mostly they were just learning exercises. Currently sitting in my archive directory.
- I have been able to resolve all uninitialized signal states in each of the corresponding DUTs, usually by defining an initial value for the signals within the test bench.
- I assume at some point I will want to move this code into the actual modules, but I am hesitant to modify much of the source code without further approval.
- In terms of lines of code written, I feel like I am keeping a good pace, but will need to sit down with David and Jacob to determine if I'm actually being as productive as I hope I am.
- We all know what a flimsy metric LOC can be.
- Having a lot of fun so far and learning new things, which is great!
- Having some issues with cpuled.vhd:
    - Original source code issue: The DUT RESET and CPULEDWrite signals were of type std_logic. This type mismatched with the expected boolean types in other components, causing the ModelSim error.
    - Either that or MS just really wants this value to be a Boolean.
    - Changes applied:
    - Changed the signal type of RESET and CPULEDWrite in the CPULED entity from std_logic to boolean.
    - In the process block, updated the comparison for RESET and CPULEDWrite to TRUE due to the type change to boolean.
    - This successfully resolved the ModelSim error, allowing the code to compile and run as expected.
    - Recommendation: Always ensure that the data types of the signals match between components to avoid similar issues.

5/19

- Changing the flags from std_logic type to Boolean has led to some minor headaches in debugging the rest of the codebase, as things are wired to expect std_logic signals and not a Boolean.
- I still think making the change was the right thing to do, as representing a True/False condition is exactly what the Boolean type is for, and as far as I can tell converting that back into an std_logic signal is relatively trivial.
- I am still not exactly sure why an std_logic signal was used instead of a Boolean, as this seems to open the program up to undefined behavior—it's not hard to imagine a situation where as an std_logic signal the flag might assume a value other than T/F, which is problematic IMO.
- Counterpoint to that is that the engineers who designed this system know a lot more about how the system is supposed to work than I do.
- Replaced non-standard libraries "std_logic_arith" and "std_logic_unsigned" in CPUConfig.vhd
    - These libraries are not recommended for use due to their non-standard nature.
    - The function has been replaced in more recent VHDL standards by
    - "std_logic_vector" in the standard library "numeric_std".
- Have continued to edit the source code where appropriate, mostly to update the signals that now take a Boolean type rather than std_logic.
- A few signals, such as the ExpansionID0-3 signals inside the DiscoverExpansionID module are uninitialized throughout the test, although they do accept an initial value on startup of the simulation.
- Given that no modules are currently attached, I think this is to be expected.

5/22

- Changing the RESET signal from type std_logic to boolean inside of the cpuled module seems to have had far-reaching effects I did not anticipate.
    - I'm now trying to write test bench for the SSITop module, and at this point, getting things to work with the new boolean logic would require changes to the module that are…extensive.
    - In retrospect, I probably should not have made that change as it is leading me to have to rewrite more and more of the code of the larger modules.
    - I'm going to revert that change and try rebuilding the test benches from that point, now keeping in mind that any "small" edits to the source code will likely lead to the need for larger changes to the code base.
    - I feel such extensive overhauls are outside the scope of my current task, and I should focus on writing test benches for the code base as written.
    - Without a more extensive understanding of the code base, we should not be altering the signals unless absolutely necessary to get the module to compile in MS.
    - Even then, we need to be extremely careful we aren't changing the functionality of the code and should avoid making changes wherever possible.
    - This should help ensure I don't land in this situation again.

- It's now come to my attention that many of the modules simply don't compile without any changes being made to them at all.
    - Given this reality, I am going to continue writing test benches for the modules that are compiled for the time being.
    - I will resolve the other modules when the time comes.
    - Many of the errors do seem to be related to parts of the code expecting a boolean type, so I may not have been entirely off base in making that change in cpuled.

    - Either way, I think writing test benches for the working modules is the best bet until I can figure out how to resolve the extensive compilation issues with the other modules.
  - Did a fair amount of head-banging today.
  - Mostly trying to get a test bench to work with the Quad component.
  - Had to give it up for the time being.
  - There are some things going on in that module that I'm not fully getting.
  - Having other modules that don't compile is also a problem, because that may be causing issues for the module I am trying to test.
  - Was able to write test benches for some of the other major modules though; statemachine, discoverID.
  - And created a new directory for the test benches that run on the unmodified source code, so today wasn't a complete wash at least.
  - Hoping to be more productive tomorrow, I feel like I lost a lot of time fiddling with the Quad component.

5/23

- Making good progress today.
- At this point I have written usable test benches for all but two major components: top.vhd and quad.vhd.
- This does not include components which don't compile in ModelSim, as I am currently unsure of how to test a component that won't compile.
- Now working on the top-level component of the RMC75E modular motion controller.

5/24

- Today is a day for thinking and review.
- I've been going full force on writing out the test benches, and I think it is a good idea to review, reflect, and plan ahead.
- I have test benches written for every module that is able to compile in ModelSim.
- Some are more rudimentary than others, but all of them test the functionality of the DUT to some degree or another.
- I tried my best to write test benches that would provide meaningful information about the DUT.
- Getting feedback from David will be helpful in deciding how to move forward from here.
- We should learn how to generate boiler plate VHDL code, since a lot of the code I have been writing is very repetitive.
- Work on getting the other modules compiled.
- All errors seem to be related to either a type mismatch within the program between an expected Boolean type and the used std_logic type, or a standard mismatch between VHDL standard versions (2002 vs 2008).
- Optimize RMC75E source code?

5/25/2023

- I was able to find the setting in ModelSim that was causing the majority of the compilation issues.
- If you right click on any file in the project tab, navigate to compile -> compile properties, then select the appropriate standard.
- By default ModelSim had been using the 1076-2002 standard, but the source code uses the 1076-2008 standard.
- After resolving that issue, only two of the original source files failed to compile:

    - controloutput.vhd
    - ExpModuleLED.vhd
- For both modules the issue seems to be that not all cases are resolved within the state machine.
- By adding a catch-all statement at the end of the logic block in controlouput, I was able to compile that module successfully.
- Theoretically, the same tactic should work for the other module, but as of yet it still does not want to compile.
- I was able to get it to compile for the time being by suppressing the error warning, but we will want to update the logic inside that module so that the error is no longer present.
- Now that I have all of the source modules compiled, I was able to continue writing test benches for the remaining modules.
- I think I should have test benches written for every module by the end of tomorrow.
- I plan to also set up a private github repo to store the test bench and project files.

5/26/2023

- Test benches have now been assembled for every module in the RMC75E codebase.
- Some are fairly extensive; others are more basic.
- It's likely more test scenarios will be added as my understanding of the correct behavior of the individual components matures.
- ExpModuleLED needs some attention.
- The component does not appear to properly handle all of the states generated by the state machine, and possibly uses some sub-optimal logic to generate the array.
- I will need to investigate this issue to determine how to alter the source code to resolve this error, which is suppressible for the time being.

- On further examination of the state encoding logic inside of the ExpModuleLED, I've determined the logic uses one-hot encoding, where each state is represented by an individual bit within a group. In this case, the "IdleState" is represented by "00", the "ShiftState" by "01", the "ClearState" by "10", and the "EndState" by "11".

- This method is simple and often very efficient, especially in cases where the states have similar behaviors or operations.
- However, it could also lead to potential issues in certain scenarios. For example, error detection and correction can be problematic because any bit flip can lead to a completely different state which could potentially lead to incorrect behavior of the system.
- Best practices might dictate that enumerated types be used instead for state encoding.
- An enumerated type is a user-defined data type that consists of integral constants and each integral constant is given a name. Here's an example of how this would look like in VHDL:

  ```
  type STATE_TYPE is (IdleState, ShiftState, ClearState, EndState);
  signal State: STATE_TYPE;
  ```

- In this way, the states are clearly defined and easier to manage and the VHDL synthesizer will determine the best way to encode these states based on logic.
- However, this approach also has some caveats, particularly when it comes to asynchronous resets or complex state machines. The synthesizer might not handle these situations optimally.

- Looking at the code further, we can see that we have a state machine with states defined as a 2-bit value, so it can take on 4 possible values (00, 01, 10, 11). These are represented by the constants IdleState, ShiftState, ClearState, and EndState.

- The error message suggests that there are 81 possible cases, which is confusing because the STATE_TYPE is a 2-bit signal and should only have 4 possible states:

  ** Error (suppressible):
  C:/Users/SHAMILTON/Desktop/Test_Bench_RMC75E/ExpModuleLED.vhd (234): (vcom-1339) Case statement choices cover only 4 out of 81 cases.

- I suspect that the issue might be related to the use of std_logic in the definition of STATE_TYPE. std_logic can take on 9 different values (not just '0' and '1'), which might be causing ModelSim to think there are 81 possible states (9^2) for a 2-bit std_logic vector.

- Based on my research, it may be better to use std_ulogic for state definitions, which only allows '0' and '1' values, or we could use an **enumerated type**.

- The **Enumeration type** is a type whose values are defined by listing (enumerating) them explicitly.

- We should also ensure that we have a 'when others' case in the state machine to handle any undefined states. If the STATE_TYPE is to be defined as a std_logic vector, the 'when others' case would be necessary to handle all the other possible values outside '0' and '1'.

- Resolved the error by using an enumerated type for the state encoding logic rather than a pseudo-array of constants.

- Added a 'when others' clause to the state machine logic that should handle any unexpected states that might arise.

- Ultimately that should not happen with the improved type logic, but it's always a possibility.

- A base case should always be included.

5/30/2023

- <u>All modules and test benches are now compiling correctly in ModelSim.</u>
- Read up on SSI interface.
- Reviewed test bench.
- Investigated how to improve test bench and ensure the correct clock / reset signals are being propagated through the system.

5/31/2023

- Improving test benches.
- It seems that the smartfusion2 libraries are not present in my ModelSim version, so I installed that library, however the compiler seems to still not recognize that library.

-

6/2/2023

- The clock signals from the clock_gen module do not seem to propagate correctly to the test bench modules.
- I am fairly certain this is the reason certain signals remain undefined after running the simulation.
- This seems to be a naming issue, as the clock_gen module uses different names for certain signals than the rest of the modules.
- Need to figure out how to alias these signals so that are correctly recognized by the test benches.
- So far 4,836 lines of code have been written for the test benches. This includes comments and docstrings, but does not include alterations to the source code, which only accounts for less than 10 lines or so.
- It also does not include code I have written that did not make it into a usable test bench, which may account for several hundred lines of code.

6/6/2023

- Contrary to my initial thought, the signals are aliased correctly inside of the clockcontrol module.

- GL0 => H1_CLK, -- 60 MHz MDT clock
- GL1 => H1_CLK90, -- 60 MHz clock with 90 deg lag
- GL2 => SysClk, -- 30 MHz system clock
- LOCK => DLLQ1_LOCK -- PLL Locked to within xxx ppm

- The question then remains, why the undefined signals?

- Dependency chain: tb_clockcontrol -> clockcontrol -> clock_gen -> clock_gen_clock_gen_0_FCC (highest to lowest dependency)

- I have determined that the clock gen issue is ultimately not stemming from any problem with the modules or the unit tests, but rather with ModelSimPro.
- Despite having installed and mapped the smartfusion2 library into the modelsim_lib directory, MS still does not seem to recognize the packages it contains.
- This also seems to be true of the PolarFire library, which is interesting since that library comes pre-installed.
- Documentation RE installing additional libraries is scant to none, which adds another layer of difficulty.
- This is especially true since MSPro seems to use a different structure for its libraries now that it is under the purview of Microchip rather than Microsemi.
- I have only been able to find user guides from MicroSemi that apply to versions that are several major version numbers behind the up-to-date version.
- Info that is only a year or two old is already out of date and does not seem to apply to the new directory setup and filetypes.

6/7/2023

- Opened up a support ticket with Microchip to see if they can help with getting the smartfusion2 library correctly installed and mapped so that the compiler is able to recognize the contained packages.
- There are multiple pre-compiled libraries for smartfusion2 library based on company:
    - Aldec
    - Cadence
    - Siemens (Formerly Mentor)
    - Synopsys
- While ModelSim recognizes the packages contained in the Siemens specific library, the compiler complains that the compiler version is not compatible with this library. Go figure.
- I am now exploring several other options for instantiating and conditioning the raw clock signals inside of another module that does not rely on the smartfusion2 library.
- Another option would be to explore using a different VHDL simulator, such as QuestaSim.

6/7/2023

- I think I have managed to solve the issue with the missing clock_gen signals.
- After fighting hard to get ModelSim to accept the smartfusion2 libraries, I figured a better option would just be to create a drop-in module that simulates the old modules.
- The older modules are not usable at this time due to the compatibility issue with the ModelSim compiler.
- The new drop-in module seems to be performing admirably, as all signals now appear to instantiate and propagate through the system correctly.
- Although I will have to investigate further to ensure the behavior of the new module emulates the behavior of the old ones correctly for my needs.
- While the new module cannot account for complexities such as clock skew and meta-stability, which require device specific solutions provided by libraries such as the smartfusion2 library, it should be more than suitable for the purposes of testing and debugging within ModelSim.

6/8/2023

- The universe must have a sense of twisted humor. After banging my head against the wall for days trying to get the smartfusion2 library to play nice with ModelSim, and then finally giving up and writing a whole new clock module yesterday, I finally figured out how to correctly map the smartfusion2 library so that the real clock modules will now work just fine. Counterintuitively, it had nothing to do with the library directories in the modelsim directory, rather I had to map the smartfusion2 instance in my MS project to a different subdirectory that was sitting in Libero. I guess the main takeaway is that I got the issue resolved (twice).

- Clock Module Notes:
    1. Ensure that the smartfusion2 library is correctly mapped to the following path in Libero SoC via the vmap command, NOT ModelSimPro:

       vmap smartfusion2
       C:/Microchip/Libero_SoC_v2023.1/Designer/lib/modelsimpro/precompiled/vlog/smartfusion2

2. If the compiler throws an error about a mismatch in the minimum time resolution units, simulate with the following command: vsim -t 1fs tb_clockcontrol

- The major victory of the day has been getting the original (correct) clock modules to work with the rest of the simulation in ModelSim.
- I verified that all modules and test units are compiling correctly and able to simulate.
- I also (crudely) recorded the ModelSim output to help hunt down and rectify rogue signals.
- Did some housekeeping, ensuring everything is up-to-date and that there is sufficient redundancy for the test bench just in case I blow something up.
- Thankfully, this has only happened a couple of times so far.
- Now that the OG clock generator is up and running, this should mean that 100% of the source code is being simulated in ModelSimPro.
- This *should* mean that I can rule out any problems being the result of me improperly emulating a component, or worse yet, not emulating it at all.
- I can now return to my primary objective of ensuring that ALL signals are defined at the end of the simulation, and that nobody is wandering off into the weeds and causing trouble.
- MOST signals are now defined when the simulation is complete, but there are still a few outliers.
- Once that is taken care of, I can move on to ensuring that the signals are not only defined, but that they are all defined *correctly,* and that the test units are able to verify that.
- Some tests are failing, but whether that indicates a problem with the source code or the test itself is yet to be determined.
- According to my betters, all signals should be defined at the end of the simulation.
- Ergo, once the simulation finishes, we should only see happy little green lines on the waveform viewer.
- A golden vector file to test outputs against would be great, so maybe at some point I can start to build that out.
- Additionally, I would like to automate the process of running the test units.
- I have been running the units manually, which is a bit tedious and also a time-sink.

6/9/2023

- I am investigating the remaining undefined signals, mostly just in the information-gathering phase.
- I feel it is better to create a comprehensive strategy based on solid information before I start trying to resolve the remaining undefined signals.
- To that end, I have been adding additional documentation to the source modules to help me better understand the correct behavior of each module.
- I did some prototyping for creating some new toys for the test bench:
    1. A test bench wrapper that can run all of the test units at once.
    2. A tcl script to help automate the testing process.
    3. Some data manipulation files that I intend to use to help define the appropriate behavior of the system by extracting mapped input-ouputs into a golden vector file.

6/12/2023

- Upon reviewing the test bench, I noted that some of the test units don't use the correct period for the clock signals.

- Stepping through them now and ensuring they use the correct periods.
    - signal H1_CLK: std_logic;
    - constant H1_CLK_PERIOD: time:= 16.6667 ns;
    - signal H1_CLK90: std_logic;
    - constant H1_CLK90_PERIOD: time:= 11.1111 ns;
    - signal S30_CLK: std_logic;
    - constant S30_CLK_PERIOD: time:= 33.3333 ns;

6/13/2023

- Added more extensive descriptions to all of the source modules.
- Created some more documentation.
- Revised tb_MDTTopSimp_v4 to emulate the clock and reset signals (v4) more correctly.
- Fixed some signal declaration errors in other modules.
- Stepping through test benches to clean up garbage and fix errors.

6/14/2023

Still working on resolved undefined signals in the test units, notably the M_INT_CLK signal inside of the MDTTopSimp module. This signal requires a fairly lengthy set of conditions to be met for it to be defined properly in the test bench. Emulating this set of conditions is proving to be a bit of a challenge. In addition to other signals and conditions being properly emulated. As far as I can tell, this is the logic that the signal follows in the DUT:

1. The state machine initiates the interrogation signal by setting the StartInterrogation signal high. This happens when the SynchedTick60 is high and RetPulseDelayEnable is low, or when CounterOverFlowRetrigger is high, also given that MDTSelect is high. This indicates the start of an interrogation pulse.

2. The StartInterrogation signal is then used to control the ClearCounter signal in state s0 of the state machine. When StartInterrogation is high, ClearCounter is set high, which will then lead to DelayCountEnable being set to low. This initiates the count sequence.

3. Once the interrogation pulse is being sent (SendInterrogationPulse is high in state s1), the DelayCountEnable is set high. This is the start of the clock cycle.

4. Then, when the interrogation pulse ends (as detected when DelayDone is high), the DelayCountEnable signal is set back to low. This is the end of the clock cycle.

5. The intM_INT_CLK signal takes the value of the DelayCountEnable signal and M_INT_CLK takes the value of intM_INT_CLK, hence it essentially mimics the behavior of DelayCountEnable signal.

6/15/2023

- After a lot of tinkering, I was able to emulate the conditions for the M_INT_CLK signal to capture the correct value and get out of its undefined state.

- Now working on setting up the signals so that actual data from the PositionRead and StatusRead signals is dumped into the data output line, allowing us to read actual meaningful data from the system.

6/15/2023

Upon further investigation of the source code, I have determined the following conditions need to be met to ensure that mdtSimpDataOut receives data from PositionRead:

1. PositionRead signal must be asserted ('1').
2. SSISelect signal must be deasserted ('0').
3. Either StartStopRisingEdge, StartStopFallingEdge, or PWM signals must be asserted ('1').
4. The RetPulseDelayDone signal must be asserted ('1').
5. All other signals (StatusRead, ParamWrite, SysReset, etc.) should be deasserted ('0').
6. The second SynchedTick pulse must have been driven so that a correct data value is latched, and not just the garbage initial value that is captured on the first pulse.

Additionally, the internal state machine in the MDTTopSimp entity controls the sequence of operations. It transitions through different states to generate and handle the MDT counters' count sequence. The state machine should be properly configured to ensure the desired behavior and enable the transfer of data to mdtSimpDataOut.

The RetPulseDelayDone signal indicates that a return pulse has not been received within 50us of the start of the interrogation pulse, which triggers the setting of the NoXducer bit. The combination of these conditions will enable the transfer of data to mdtSimpDataOut.

mdtSimpDataOut can receive data only after the 50us mark has been reached: The RetPulseDelayDone signal, which is set when the Delay counter reaches the value specified by the RetPulseDelay constant (indicating that a return pulse has not been received within 50us), triggers the setting of the NoXducer bit. This condition, in combination with other factors, enables the transfer of data to mdtSimpDataOut. Therefore, mdtSimpDataOut can receive data only when RetPulseDelayDone is asserted, indicating that the return pulse delay has exceeded 50us.

mdtSimpDataOut's data transfer depends on the SynchedTick60 pulse, RetPulseDelay, RetPulseDelayEnable signals, and IntPulseLength: The RetPulseDelay constant specifies the time duration in clock cycles that represents a 50us delay. The RetPulseDelayEnable signal is used to enable the Delay counter, allowing it to count and measure the delay. The IntPulseLength constant represents the duration of the interrogation pulse. The combination of these factors determines the timing and synchronization required for mdtSimpDataOut to receive data correctly.

6/16/2023

We are reading valid data from the mdtSimpDataOut output line! Hurray!

When we send the PositionRead signal, we get this data in the 32-bit register:

- The value "000…11110000000", which in binary corresponds to the number 1984 in base 10.
- The value of 4(480) is 1920. In other words, 1984 is equal to 4 times 480 plus an additional 64.

- This corresponds to the count recorded by the 12-bit synchronous counter.

When we send the StatusRead signal, we get this data in the 32-bit register:

- The value "000…0000000011", which in binary corresponds to the number 3.
- These are the selection bits used to select the correct transducer type, in our case MDT.

Now we can include some more test stimulus to test a few edge cases, such as what happens when SysReset remains inactive.

- When SysReset is continuously high, SSISelect is undefined at the start of the simulation, but becomes defined after about 1 us.
- M_INT_CLK remains undefined indefinitely in this scenario.

6/19/2023

- Hunting down more undefined signals in the test units.

6/20/2023 - 6/21/2023

- RMCTools basic training.

6/22/2023

- Still hunting.

6/23/2023

- Working out some kinks in the SSITop test unit.
- While everything is defined at the end of the simulation, there does not appear to be any data coming out of the data-out line.
- So, same problem as with the MDTTopSimp module really.
- I am also not seeing the nine negative pulses from the SSI_CLK.

6/28/2023

- The SSITop module still plagues me.
- It is possible this module may need to be fed a reset signal that is not present in the DUT architecture.
- I've been trying to trace down where the logic is going wrong and it just keeps leading me down another fruitless rabbit hole.
- Buuut I am learning things in the process and refining my debug / troubleshooting methodology, so always a silver lining I suppose.
- I figured out how to properly construct the 32-bit vectors that need to be passed into intDATA on each paramWrite pulse.

6/30/2023

- Through a lot of trial and error and picking David's brain, I think I finally have the SSITop test bench correctly simulating the module.
- All signals are now defined.
- The ssiDataOut line displays the appropriate data when the read signals are asserted.

- Every loop tick after the initial set-up tick now triggers nine negative clock pulses on the SSI_CLK.
- Given that the read signals are asserted correctly in sync with the loop tick.

7/6/2023

- Now able to synchronize an 8-bit data write from SSI_DATA with SSI_CLK, writing one bit per negative pulse, with the ninth pulse reserved for a special bit.
- On the next read assert, the data shows up correctly on the SSI_DATA_OUT line.
- As long as ShiftOn is initialized to SOME value in the source code, and not left undefined as it was originally, everything should work fine.
- SSITop test bench looks good, though we may want to add to it later, such as doing an additional vector write and ensuring the data on the output line updates correctly.
- Moving on to Quad module.

7/7/2023

- Tinkering with Quad module.

7/10/2023

- Quadxface module now fully and correctly simulated by test bench.

7/12/2023

- Quad / Quadxface testbench now correctly simulates each of the 6 expansion modules in addition to the base module and wrapper code (Quadxface and Quad, respectively).
- Moving on to Analog module

7/12/2023

- After some trial and error I now have all signals defined in the testbench, however the ExpA_CLK signal is still misbehaving and has an irregular pattern.
- The problem seems to be related to the looptime variable and how the clock is syncing with the rest of the source code.
- Looking into the statemachine module which produces the ExpA_CLK signal.

7/17/2023

- After setting SlowEnable to only be active for one clock tick on every 8^{th} tick, rather than low for 8 ticks and high for 8 ticks, the ExpA_CLK signal is now behaving correctly.

7/18/2023

- By uncommenting the following signal value initializations on line 75 of the databuffer module, I am able to write data to the AnalogData output line:

  BankSelect,
  WriteEnable,
  WriteBank0,

WriteBank1: std_logic:= '0';

- I was able to get the ‘o’ signal to properly define by uncommenting the initialization value for the RAM signal on line 89 of the RAM128x16bits module.
- Note that I also commented out the weird signal assignments for ExpA_CS_L and ExpA_CLK on line 121 of analog.vhd that were causing issues with the ExpA_CLK signal.

7/19/2023

- Touched up a lot of test units today, include those for serial memory, I/O control, and discrete I/O.
- I think the test bench is looking pretty darn solid.

7/24/2023

- Control output test unit looking good.
- Note that ControlOutputWriteLatched1 and ControlOutputWriteLatched2 need to be assigned an initial value (1/0) in the source code for the simulation to run correctly.
- Doing some more work on DIO8 module.

7/26/2023

- DIO8 test unit looking good.
- Control IO test unit also looking good.

7/27/2023

- Working on serial memory interface test unit.

7/31/2023

- The state machine logic in the serial memory interface looks like it could benefit from some consolidation.
- There seems to be a lack of logical separation / organization that makes it somewhat hard to follow.
- A lot of logic seems to be repeated.

8/01/2023

- Refined and expanded expansion and control ID discovery modules.
- Added to and consolidated documentation.

8/02/2023

- Made major refinements to the expansion signal routing module.
- Reviewed, tweaked, and refined most of the test units in various ways.

8/03/2023

- Added to the codebase documentation.

8/07/2023

- RE: serial memory module: It seems as though the problem is not in the state machine in so much as the state machine is just waiting for an ACK signal to be sent at the end of every 8-bit word or byte sent. There is room for 3 8-bit words to be sent, therefore there should be a total of 3 ACK signals received by the end of the transmission.

8/11/2023

- By modifying the acknowledgement handling process in the test bench to simulate the correct timing of driving the SDA line low, the simulation seems to be working correctly for the serial memory module.

8/14/2023

- This is my last day with Delta Motion, and it's bittersweet.
- I am proud of the work I put into the test bench system, and I think it should provide a solid foundation for making improvements to the RMC75E unit, along with the hay-load of documentation I produced during the process.
- Over and out.