RISC-V Soft Core Implementation for Intel's DE-10 Lite FPGA

Terence Williams, Nicholas Goralka, Rylan Moore

***Abstract*—The goal of our research aims at creating a soft-core processor on top the existing FPGA hardware in the DE-10 Lite board. The FPGA itself uses a MAX-10 chip made by Altera, featuring many on board components, such as 7 segment displays, an accelerometer, buttons, and switches. The soft core could then be implemented in future classes at the University of Colorado Boulder, such as Programming Digital Systems (PDS). The base core is A L30 RISC-V processor, provided by Codasip. Full functionality of the processor will be achieved by implementing memory over the AHB-Lite bus in BRAM on the FPGA fabric, making I/O accessible through writing to specific memory locations, and by routing JTAG signals to I/O for debug of code on the core. C code to run on the core will be compiled using the open source RISC V toolchain, converted to a 32-bit hex format, and added to the core at compile time of the Verilog implementation of the core.**

I. Introduction & Objectives

Starting off, what exactly is a soft core CPU? A soft core CPU is a hardware description language implementation of a microprocessor architecture. This will be done through the use of Verilog, a hardware description language. The end goal of our soft core processor would allow us to load a simple "Hello World" C program onto the DE-10 Lite board.

Motivation behind the implementation of a soft core lies in the fact that making the DE-10 capable of running code from the open source RISC-V architecture will extend the functionality of the device and allow for the device to be potentially used by other courses. Developing a way to pass GPIO and other peripheral devices into the soft-core was a beneficial learning experience and provided insight to the different work that goes into making hardware resources available in a higher level program such as C code. This project allowed the team to work with a variety of aspects of computer architecture including architecture on the low level, to C code on the high level, and the different parts that allow the two to work together in between.

The plan for this project was to reach different goals in phases. The first step was to port the RV32I ISA to the FPGA. The implementation was done in Verilog but another high level synthesis language that will enable us to generate RTL code could also have been used. The FPGA will physically implement the logic required for a RISC-V CPU in a 3 stage pipeline. Next means for the FPGA flash to accept uploaded binaries compiled with a RV32I cross-compiler will be written. Once a working implementation of a CPU is verified, work will be completed to add some of the functionality of peripherals found on the DE-10 Lite Board. This includes the variety of switches, LEDs, and digital seven segment displays on the board. Finally, the team attempted to get programming working. For this a user will need to be capable of flashing C code to the device and have it execute on the board with access to all the available peripherals. The Codasip platform allows the user to automatically generate a compiler. The compiler generation tool learns the programmers intent of each instruction from a "semantics" file. The goal was to use this generated compiler but as discussed later in the findings we needed another method of compiling code for our RISC V soft core implementation. Testing on this portion of the project involves writing test code and getting a "Hello world" program working that can test functionality of the IO that has been previously configured. Software support for the peripherals must be written. Simple drivers in order to drive the LEDs, seven-segment display, and other peripherals must be written in order to provide a level of abstraction for the end user. The user should be able to call Arduino like functions in order to drive peripherals rather than worrying about individual bits within registers. The process for synthesizing and uploading the RTL code as well as having the ability to compile and flash C code to the soft-core CPU must be repeatable. In order to accomplish this, comprehensive documentation must be written so that this process can be repeated and built upon in the future.

The results are evaluated based on the timeline of completeness. We had a few milestones in order to track team progress. Working alongside Keith Graham over at Codasip, we were provided with a L30 core. This core is a 3 stage RISC-V processor which we will be expanding on for the DE-10 Lite board. Codasip has a tool that allowed us to work in RTL then convert that to Verliog which can be loaded onto the FPGA. Our initial goal is to familiarize ourselves with RTL (Verilog) and get the base processor loaded onto the DE-10. From there we will begin routing the peripherals such as button/switch I/O as well as the 7 segment displays. As a stretch goal we can then alter the ISA using the Codasip tool, however we don't plan on doing this until a working implementation is running. With the plan being well defined above, once we start implementation, we will then be able to set minimum and stretch goals in order to make sure all of the requirements are met in a timely manner.

II. Related Works

Many trends in computer architecture which have held true for the last 60 years have come to an end or are in the process of ending. Computer architects used to be able to count on the fact that processors would increase in speed and

efficiency while also decreasing in size through Dennard Scaling and Moore's Law. Additionally, Amdhal's Law limits the performance which can be increased through parallelization. As is the opinion of the engineers at Codasip, the future of computer architecture is designing for specialized applications. Historically there has been a rift between designing general purpose hardware (ie processor) and the designing for flexible custom hardware solutions (ie custom FPGA designs). This distinction between the general purpose standardization you get with a general purpose processor versus the efficiency achieved with a fully custom hardware solution tends to fluctuate about every ten years. This trend was first noticed by the CTO of Sony as thus it is commonly referred to as Makimoto's Wave [4].

However, utilizing the open source RISC-V ISA and the high level synthesis tools which were generously provided by Codasip. It is feasible to extend the functionality of an existing general purpose processor and build domain specific accelerators. As the RISC-V ISA is a open source architecture one could simply build the entire microarchitecture for scratch using a traditional HDL, however utilizing an architectural high level synthesis language like Codal has advantages in that there is a complete processor description. Such support makes verification much easier. Semiconductor manufacturers also offer processors which have a FPGA on die to support customization.

In order for our research project to be successful, we took a look at an article called "Implementation of Soft-Core Processors in FPGAs"[3]. This research paper provides a base knowledge on the concept of soft-core processors, highlighting key theoretical and practical concepts. The authors compare the advantages of soft-core processors over the traditional "hard-core" counterparts. The study concludes that the soft-core processors are a flexible as well as efficient solution for implementing various applications in FPGAs. The authors present a case study where they implemented a soft-core processor in an FPGA for the use of digital signal processing, then evaluated its performance. In our research project, we won't need to use the DE-10 for such advanced techniques such as signal processing, however it is beneficial to be aware of the capabilities of our FPGA. Another important mention in the paper was the section on debug tools and options. The researchers emphasized the importance of having effective tools in regards to identifying a problem or error. Some of the ones recommended were embedded logic and bus analyzers, software simulators and trace capability.

In 2022 a team from Sun Yat-sen University took a look at implementing a RISC V softcore on an FPGA for the purposes of creating a low-cost Edge-AI accelerator [2]. Their research focused on the base RISC-V core with an added convolution accelerator and peripherals mapped through the APB3 bus. Their core design was implemented on both a Pango and Xilinx board and each had different performance. The research found that timing convergence could only be achieved at 100 MHz on the Pango while the Xilinx FPGA could reach 250 MHz. This difference in clock speed is a result of the variance in architecture of each FPGA platform. This research intends to evaluate a RISC-V core on a DE10 Lite board which will likely have other differences in performance. Implementation of peripherals on the DE10 Lite should be similar to the boards used in this research and be implemented on the APB3 bus. In this research we extend the work of implementing a RISC-V soft core by implementing peripheral functionality for the RISC-V softcore on the DE10 Lite. This work will focus on the ability to access these external devices (switches, displays, buttons) to expand the use cases of the soft core and make it more appealing for implementation in an engineering course. Along with specific accelerators the soft core will have a variety of potential applications that are of value to a university program that benefits from reuse of a board students already need to have purchased.

III. Overview and Implementation

The base L30 core was provided by Codasip for further development on the DE-10 Lite FPGA board. The steps in this work will be to implement memory and the bus interface between core and memory. The core provided does not implement the memory interface. Memory will communicate with the core over the AHB-lite bus. Additionally, some of the other peripheral signals need to be routed from the core including a JTAG bus in order to facilitate inline debugging as well as GPIO output. A simplified high level block diagram is shown below.

Codasip studio outputs the RISC V core in system verilog and is then imported into Quartus for flashing. The core module is then instantiated in our system top file, and all signals are routed in and out of the core from other units. 29 of the 37 signals into the top core module have already been routed. The following signals that the core module requests have not been routed from the rest of the system yet. These signals may need to be routed to I/O or will not be needed. These remaining signals include:

- p-meip
- p-msip
- p-mtip
- p-nmi
- dmactive
- ndmreset
- p-wfi
- p-boot-adr

Since implementing an entire AHBlite memory interface from scratch was a daunting task at the start of the project, we initially attempted to utilize an open source IP core that we can use per the license we can use for non-commercial purposes. The IP core implementation by ROA Logic is a fully parameterized soft IP implementing memory on the FPGA's fabric that can be accessed by a AHB-Lite master. However, it should be noted that the IP core could not originally be compiled in the student version of Quartus Prime without slight modifications. The IP core is implemented entirely in SystemVerilog. SystemVerilog is supported by Quartus Prime but not the most recent versions of the language. As such in

order to get the core to compile correctly, there were a few syntax tricks that had to be slightly modified in order to get the core to compile, however, now the project is compiling.

The memory interface IP core that we instantiate synthesizes RAM blocks in a generic way. That is the precise RAM technology that gets synthesized is not set until compile time. Right now from the memory hierarchy it appears the technology that gets synthesized is the altsyncram which should work for our purposes. However, if we need to change to BRAM we have the ability to do so and can modify the memory IP core. C code can then be compiled using a RISC V cross compiler and uploaded to the instruction memory at FPGA compile time utilizing the "$readmemh(INITFILE, mem-array);" command in SystemVerilog. As long as the boot address is correct the processor should begin executing the code. All the code we send to the core will have the program counter start at 0 to hopefully keep the entire setup process simple.

After attempting to use the AHB IP core to implement BRAM we realized that the memory interface found would not be entirely sufficient for our uses. The memory interface needed to be pretty much implemented from scratch. The code consisted of three modules.

The first was the AHBLite-BRAM module - this operated a wrapper which translated particular signals within the AHBlite bus into signals which could be understood by the technology specific implementation of the BRAM on the FPGA fabric. The address of the signal in the AHBlite bus is 32 bits wide, however, the size of the memories is not 4Gb as it would take up the entire FPGA fabric. Additionally, this wrapper module performs arbitration between all the reads and writes to the BRAM.

The next module was a single port read/write BRAM implementation that acted as the instruction memory to the system. This module was implemented as a seperate module in order to maintain clean coding practices and make the system easier to test and debug. Finally, the load/store memory implementation was created as a single port read/write BRAM and also mapped certain memory addresses to I/O registers.

- $HADDR[31:0]$ The address to be accessed in memory.
- $HTRANS[1:0]$ Indicates the type of transfer to occur. $[00] = IDLE, [01] = BUSY, [10] = Non-sequential, [11] = Sequential.$
- $HWDATA[31:0]$ Transfers data from core to memory in a write.
- $HRDATA[31:0]$ Transfers data from memory to core in a read.
- $HWRITE$ Indicates transfer direction Read or Write.
- $HREADY$ Memory pulls high to indicate that previous transfer has completed on the entire system
- $HSIZE$ Size of the memory transfer in bytes
- $HREADYOUT$ Indicate that the particular transaction for the slave device has completed.
- $HRESP$ Error Code
- $HSEL$ Slave Select Line
- $HBURST$ Indicates a burst transaction
- $HPROT$ Protection Signal Encoding

In order to verify that our implementation of the memory interface was correct, an automated test bench for the system was created and simulated in the ModelSim extension provided by Quartus. In this testbench, the state of the BRAM is initialized by a hex file of known values. The program iterates of the addresses reading the values in such a way that the AHB-lite bus is forced to pipeline certain operations and service multiple transactions in parallel. Values are both read and written to in order to verify that the memory address are holding the expected values. Results from these testbenches will be discussed later.

Memory mapped registers will serve to allow I/O operation. These registers will allow us to route LEDs, Buttons, and displays to memory addresses. Then we can toggle LEDs and other devices with a Store operation. Part of the reason we implemented our own AHB memory system was to allow for routing of these I/O signals out of the memory unit to the I/O to physically control functions on the board. In order to debug our C code, we will be using OpenOCD (Open On-Chip Debugger), and will require an additional hardware device, which uses JTAG. The PLL and clock configuration is described in the documentation provided by Codasip. Memory changes should be fine since we're building memory ourselves and are not using their existing implementation.

Our next steps were to clean up the Github repository. At the time it was tracking all of the compiled files, and a simple fix on our end will be to add gitignore which will allow us to specify files for github to ignore. Our next step following a quick cleanup will be to add testbenches in order to verify our logic design in hardware. This will save us some time debugging in the future as we can create functional unit tests for our design. Finally, our last step is to configure the top level file, and compile C code to upload to the memory on the FPGA in order to test full system operation.

IV. Evaluation and Results

Results will be validated based on a few different methods. First, a series of verilog testbenches will be used to verify that the memory system is working properly for loads and stores. Next, LED and 7-segment displays will be used to show test outputs of program execution in the core. Finally, JTAG will allow analysis of code execution and allow for standard debugging.

Configuration for testing will be relatively simple. The core and memory system which exists in verilog will be flashed to the DE10 Lite board with the program loaded into memory before runtime. The first instruction in memory will have its address coded into the core so that execution will start on bootup. Once execution begins the team will have a couple of ways to validate that the core is working properly. The first is to simply write programs which access lights and other I/O on the FPGA which will at runtime either

function as intended or not. In the case that there are issues with accessing I/O or running programs there is also a JTAG interface to the core. There are 10 switches, 10 LED's, and 6 seven segment displays which will be accessible from the core.

JTAG is routed to some of the digital pins on the FPGA and can be picked off and analyzed through a debugger. A J-Link EDU along with Open OCD will allow analysis of the data collected over JTAG. Through the use of JTAG the team hopes to verify that instructions are executing through the pipeline and that all of the various interfaces are available to the core. Specifically loads and stores will need to access the memory which works through the AHB3 Lite bus. This bus implementation is new to the Codasip Softcore and likely will need some tweaking to get working properly. JTAG should yield some information on the status of attempting to load data from and store data to the instruction and data memories in the system. Other I/O will not work until the memories are working as the communication between the softcore and the FPGA to control onboard I/O will occur through loads and stores to predetermined memory addresses, which the verilog system will use to determine if the core is driving each I/O element low or high.

Codasip studio has provided us with an online C compiler for their L-30 core. The compiler is consistent with the ISA, therefore it should not be necessary for us to match an open source RISC-V compiler with our existing core architecture. The codasip provided compiler will be used to create the binary output of our "Hello World" C-Program which is then used to replace the default init file in Quartus. After loading the firmware onto the FPGA, the goal will be to see "Hello World!" rotate across the 7 segment displays. During our work the team discovered that the built in compiler would not be sufficient for our work. Codasip's process relies on Xilinx software elf2mem or other similar tools which convert the executable output by Codasip to something that can be loaded onto the FPGA. We instead used the opensource RISC-V toolchain to compile and link our programs. With flags set for 32-bit output, no startup files, and reduced program size we successfully built a binary which begins execution at Program Counter 0. From here we needed a hex file which could be loaded onto the DE-10. To accomplish this without Xilinx software bin2mem, we found and modified a short python script that uses the library itertools to convert the binary to a hex file that has 32 bits on each line, generating an output file compatable with the $readmemh$ command in Verilog [7].

Regarding benchmarks and testing, we are assuming that the core provided functions as expected and therefore won't be configuring any test-benches to verify it's operation. However, we are creating test benches for the on board memory system. This process includes uploading the memory system and a corresponding test-bench to the FPGA. The goal of this test-bench is simply to verify that we can load and store data from the memory which is being implemented in the FPGA rather than in the DRAM.

Overall this research aims at creating functional interfaces for the DE-10 Lite which will allow for C code to be run on the L-30 core. Through the use of LED and 7-segment displays, the team can visually verify the functionality of the soft core. The configuration for testing is straightforward, with the core and memory system flashed onto the board and the program loaded into memory before run time. The mapping of the JTAG peripheral is highly important as it will then be used to confirm that the instructions are executed as expected and provide a finer look at success of the soft-core when compared to simply viewing the various LED's on the board.

A. Testbench Output

A testbench verified our implementation of the AHB-Lite memory system. The testbenches automated reading and writing to addresses in memory as well as the pipelined communication on the bus. Outputs of these tests are shown the following images.

The first tested was a write transaction. We wrote $0xfeed$ to address 0 and looked at the expected output and what we saw. Results shown in figure 2.

Next we tested a read transaction by reading $0x0ff0313$ from address 0. Results shown in figure 4.

Finally we performed a larger test to verify the AHB transaction pipeline. The results of this are shown in figure 6.

Based on these three test benches we are confident that our memory implementation will be functional when connected to the core, and that a store to a I/O mapped memory location will result in the proper signal being set as output from the memory module, allowing the core to access I/O.

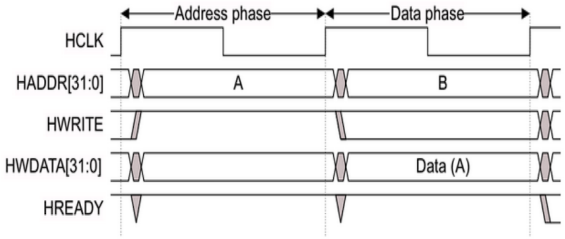

Fig. 1. Expected Write Transaction signals

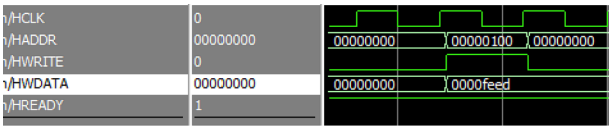

Fig. 2. Measured Write Transaction signals

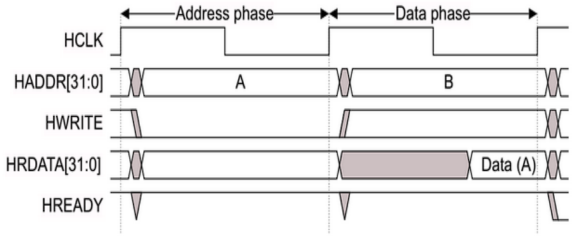

Fig. 3. Expected Read Transaction signals

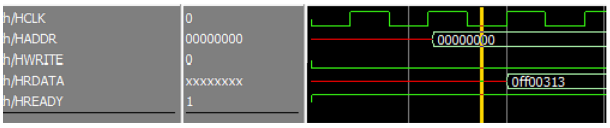

Fig. 4. Measured Read Transaction signals

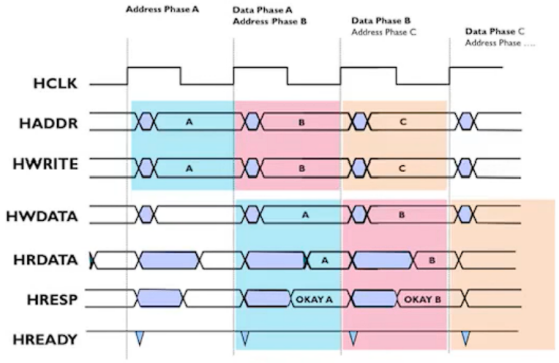

Fig. 5. Expected Pipelined Transaction Signals

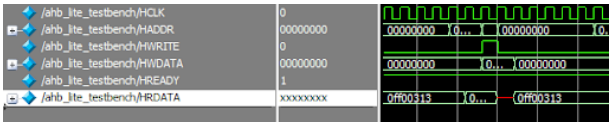

Fig. 6. Measured Signals of AHB pipelined transaction

B. C Code Implementation

The other major part of this research was finding a reliable way to get code written in C uploaded to the soft core for execution. With the Codasip being not replicable in our timeframe and with the resources available we needed a new way to compile and prepare code. Using the open source RISC V toolchain we compiled code with the following flags:

- -Os (For no optimization).
- mabi=ilp32 (standard integer naming scheme in gcc).
- -march=rv32i (For 32 bit compilation).

We also used the following linker flags to get a binary:

- -Tlink.ld (call the linker used for bare metal).
- -fno-builtin-printf (printf not available without additional work in the soft core).
- –specs=nano.specs (For bare metal compatibility).
- -nostartfiles (Program is simple with PC=0 at start).

After generating a binary file we need a hex conversion in order to upload this code to the memory in the FPGA. We used a base python script from SiFive to do this. It uses the Itertools library in Python3 to convert a RiscV binary to a hex file with a variable line size. We call the python script with the following:

```
python3 bin2hex_python.py -w 32 code.bin 32bit.hex
```

This generates a hex file that contains one instruction per line and no other characters. This file does not throw an error when added to Quartus and called in a $readmemh$ command. Some flags may need to be changed in this process but the base process should be successful at generating a program from C that will work on the L30 core. As discussed in the conclusion we ran into issues with code beginning execution on the core. More time would allow us to test interface with the core over JTAG, allowing C code to really be tested.

V. Conclusion

At the end, we were unsuccessful in creating a fully functioning soft-core on the DE-10 lite. Using the Signal Tap tool, which comes with Quartus, we synthesized a logic analyzer directly into the FPGA fabric in order to view the use of resources in our design. The processor is currently utilizing 70% of BRAM, with 21% being logical elements. In addition, we used one of 4 PLLs on board the DE-10 Lite. These results are shown in figure 10. Some of the issues we ran into during the end of our research was the programming itself. We needed to initialize the state of the BRAM at synthesis using a command called $readmemh. This command only accepts hex files with no non-hexadecimal characters. In addition, the RISC-V compiler we chose outputs a binary, and we needed all hex characters. As a team we were able to use a built in python library within an adapted SiFive script to convert the binary to hex with 32-bit lines. During this time frame, we looked into other options and noticed that Xilinx, who is another FPGA company, provides a Data2Mem function which would have been nice to have. However, since Altera only supports Intel based architectures, we moved away from the idea. After scaling back from our initial plan to scroll ”hello world” across the 7-segment displays, we decided on writing a simple program to blink a few LEDs. The team took advantage of JTAG and OpenOCD in order to look at the actual execution on the core. We planned on being able to see instructions executing in intended program order, memory access should maintain integrity, and the core should be able to interact with the ”outside world” via I/O peripherals. After setting up a logic analyzer via the signal tap, we noticed that the program counter was not incrementing, meaning that no other instructions are being fetched. Some possible causes of this error was the Reset signal was routed to the incorrect logic, and was always being held high, leaving the core in a state or reset.

The memory interface that was developed for this project has been thoroughly tested and should function as expected

when issues with core reset have been fixed. Along with this the method for compiling code for the core can be modified without any major hassle if other flags in the compiler or linker are needed for the core to operate in the future. The L30 core should be verified and tested extensively in Codasip Studio to ensure that the core works as expected before it is translated to Verilog. With some more time there is a clear path to getting basic functionality and execution on the core operational.

VI. Future Works

Future works of this research will include diving into the root cause for the continuous reset of the soft-core. In addition, the other peripherals of the AHB-Lite Interconnect need to be routed, these including the UART interface, I2C buses, SPI, and VGA. Shown in figure 11. Also, software support on Xilinx chips provides a ready to go IP blocks for the AMBA interfaces. Xilinx provides software tools for formatting data into appropriate hex formats. The research from this article can be taken and adapted to a Xilinx FPGA and many of the problems that the team ran into can be avoided. After avoiding these problems the team faced, the progress can be used to make a fully functioning soft-core for an FPGA. The code provided by Codasip should be fully tested in their studio before export to Verilog as well.

Intel recently launched their own NIOS-V processor for Altera FPGA boards, which implements a RISC-V style processor in the FPGA fabric. This core is similar to the goals of this research and should be analyzed for its' potential use in courses at the Univeristy.

Acknowledgements

The research presented in this article would not have been possible without the contributions from Keith Graham, Tadej Murovic and Codasip Studio.

References

1. Baklouti, Mouna, and Mohamed Abid. "Multi-softcore architecture on FPGA." International Journal of Reconfigurable Computing 2014 (2014): 14-14.
2. Wu, Hailong, et al. "Implementation of CNN Heterogeneous Scheme Based on Domestic FPGA with RISC-V Soft Core CPU." 2022 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA), Integrated Circuits, Technologies and Applications (ICTA), 2022 IEEE International Conference On, Oct. 2022, pp. 158–59. EBSCOhost, https://doi.org/10.1109/ICTA56932.2022.9963056.
3. Minev, Petar Borisov, and Valentina Stoianova Kukenska. "Implementation of soft-core processors in FPGAs." UNITECH'07 International Scientific Conference. 2007.
4. Urquhart, Roddy. "Scaling Is Failing." Codasip, 2 Mar. 2022.
5. Prikryl, Zdenek. "Creating Domain-Specific Processors Using Custom RISC-V ISA Instructions." Codasip, 23 Sept. 2020.
6. https://github.com/RoaLogic/ahb3lite_memory
7. https://hackmd.io/@vuJ_c2nYTM2v6lZwDtjajg/HJikviz3Y

Diagrams

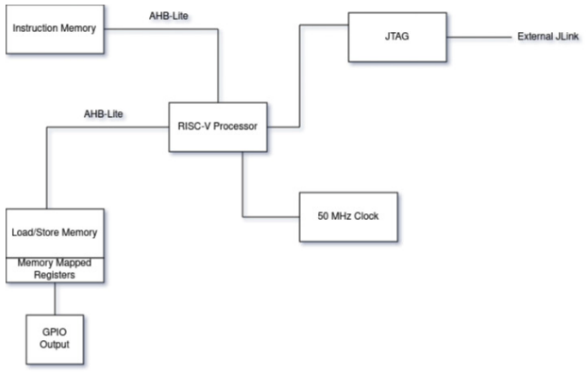

Fig. 7. Starting Block Diagram

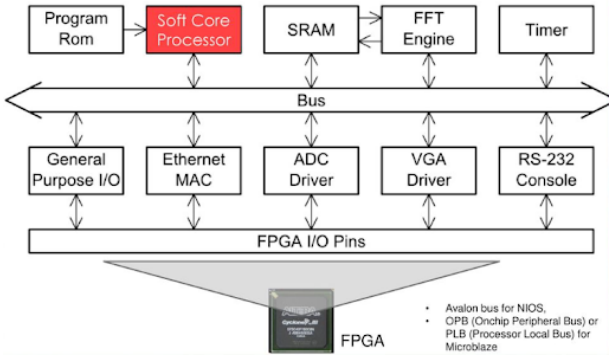

Fig. 8. Soft-Core Visual

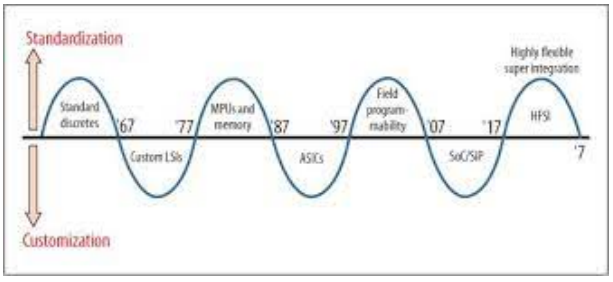

Fig. 9. Makimoto's Wave

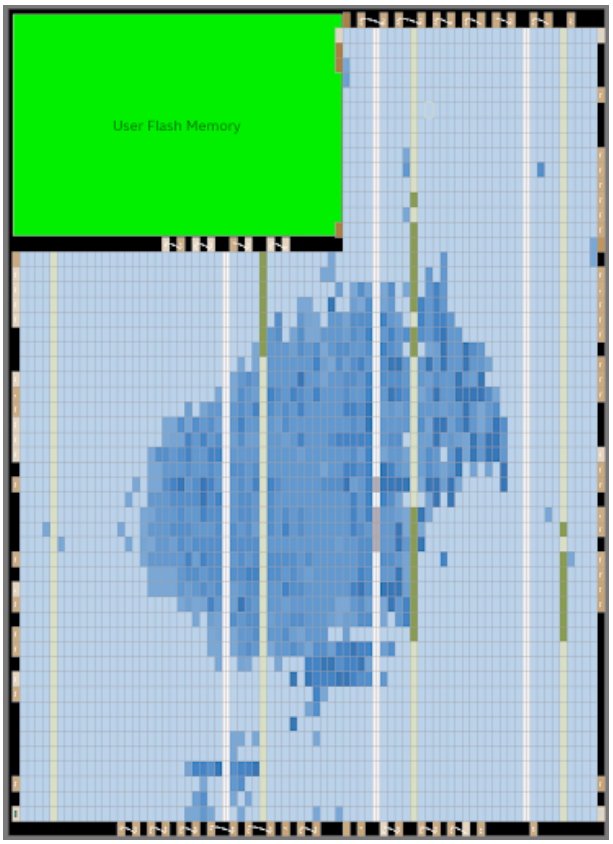

Fig. 10. Total Design Resources within the DE-10 Fabric

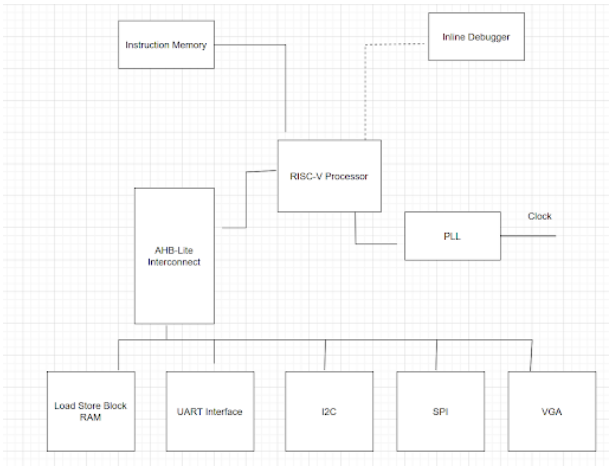

Fig. 11. Future Expansion of interface with the core