Technion - Israel institute of technology

Electrical & Computer Engineering Faculty

VLSI Labs

LOTR(Lord of the Ring)

RING ARCHITECTURE MULTI CORE FABRIC WITH FPGA SYNTHESIZABILITY AND VGA DISPLAY

Authors:

Adi Levy

Saar Kadosh

Advisor:

Amichai Ben-David

Semester of registration : Winter 2021

Submission date : October 2022

LOTR:

RING ARCHITECTURE MULTI CORE FABRIC WITH FPGA SYNTHESIZABILITY AND VGA DISPLAY

LOTR PROJECT

Adi Levy (adi.levy@campus.technion.ac.il)

Saar Kadosh (skadosh@campus.technion.ac.il)

CONTENTS

Revision History

Rev. No.	Who	Description	Rev. Date
0.5	*Adi Levy*	*Initial sheet*	*02 September, 2022*
0.6	*Adi Levy*	*Ready for submission*	*10 October, 2022*

Figures

Related Documents

Name	Path	Description
riscv_isa_spec.pdf	RISCV_Spec	*The full RISCV Unprivileged Specification file.* *Including the RV32I Baseline ISA*

Glossary

Term	Description
GPC_4T	Hardware embedded 4 thread RISC-V core.
ISA	Instruction Set Architecture. (such as X86, ARM, RISC-V etc.)
IO	Input & output.
IP	Intellectual Property. In this case, RTL building block that can be consumed.
HAS	High Level Architecture Specifications.
MAS	Micro Architecture Specifications.
I_MEM	Instruction memory – where the program is loaded and ready for execution.
D_MEM	Data Memory – where the LOAD & STORE instructions read/write Data.
Pipeline	Common Way to parallel and utilize Hardware https://en.wikipedia.org/wiki/Instruction_pipelining
RISC	Reduce Instruction Set Computer. (Unlike CISC -Complex Instruction Set Computer) https://en.wikipedia.org/wiki/Reduced_instruction_set_computer
Thread	A "hardware thread" is a physical CPU or core that can run a program.
RISC-V	A relatively new open and free ISA. (comparable to intel X86, ARM) https://en.wikipedia.org/wiki/RISC-V
RV32I	“RISC-V 32-bit Integer” The RISC-V baseline compatible ISA (no extensions M/A/F etc.)
Standard interface	Functional characteristics to allow the exchange of information between two systems
Word	32-bits of data - 4 Bytes. The size of an integer in RV32I ISA.
Hazard	Potential source of harm. in this document when reading Outdated data, or wrongly executing Instruction.
Strap	Tie signals to constant value (1’b0 or 1’b1)
MSFF	Main-Secondary Flip Flop. (AKA Master-Slave Flip Flop)
Clock Gating	Logic that allows to condition the MSFF clock. Functionality and power reasons.
Polling	Actively sampling the status of an external device.

1 Introduction

1.1 About The Project

The LOTR project - "Lord of The Ring" also known as "***THE FABRIC***", is a top-level project for a design and architecture of a ring architecture-based multi core Fabric.
the LOTR is based on two other EE faculty students' projects: "**Ring Controller**", by Tzahi Peretz and Shimi Haleluya , and our own **"GPC_4T"** project. The LOTR use the GPC_4T alongside with a Ring Controller to create the Multi-Core Ring Fabric architecture. The design has the ability to be loaded and synthesized on an FPGA kit and to run graphical programs to a VGA based monitor.

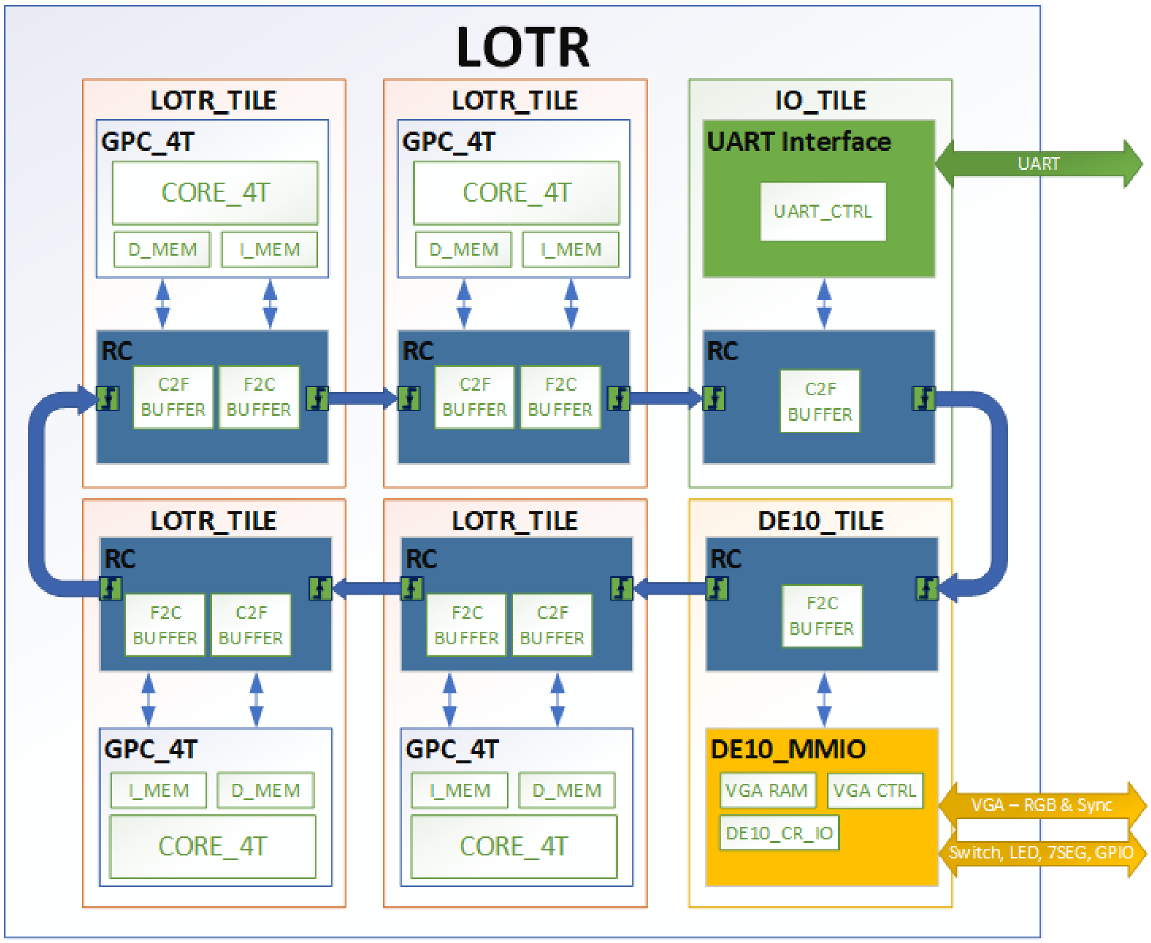


Figure 1 - LOTR block diagram, currently without the UART interface

LOTR Tile – the base unit that will contain the GPC_4T Core , Memory, and an RC unit. Designate to connect many other LOTR Tiles in a ring connection to create the Multi-Core Ring architecture.

The LOTR Fabric contains the LOTR Tiles with cores, each can separately run different multi thread code in parallel, and perform multi core code like reading for another core's memory.

In addition, the entire LOTR project can be loaded to an FPGA kit model: "DE10-Lite" and a New module named "DE10Lite_Tile" was created in order to communicate with the DE10-Lite inputs and outputs, and to communicate with a VGA monitor so that display features are also added to this project.

The **LOTR** Project is a Computer Engineering Students' project created by Adi Levy and Saar Kadosh as a part of BSc degree demands of the Faculty of Electrical and Computer Engineering at the Technion institute and it is the continuation of GPC_4T project.

The motivation of the project was to continue research and develop the abilities of the RISC-V instruction set architecture, which is an open-source ISA for CPUs and a very "hot name" in the industry.
Another motive was to take our design from previous project to the next level by synthesizing it to an FPGA chip and to use VGA display to show the abilities we developed.

The LOTR ring architecture can scale to any number of cores - with the price of latency on non-local real/writes on the fabric . Each core has 4 hardware controlled main threads with 32 registers for each thread

Due to the FPGA constraints, we instantiated Only 2 Cores.

A major change from project A is the memory modules : each core can use 8KB of instruction memory and 8KB of data memory, twice the size of memory on GPC_4T project. In addition, the memory is physically exists on the FPGA kit, unlike the behavioral memory from the old project.

The idea of the project was led by this projects mentor Amichai Ben-David. In the previous project, The LOTR was only a dream idea, hard to implement. But in the end the dream came true and the FABRIC was born.

Like "GPC_4T", The project contains many aspects such as architectural and logic design, HDL programming , low level programming & scripting and validation.

The entire RTL code of the project, documentations, applications and validation & verification tools and guides to operate them are all can be found in the LOTR GitHub repository.

1.2 About The Authors

Adi Levy , BSc Student , Studies Computer & Software Engineering with Advance Programming as areas of expertise at Technion – Israeli Institute of Technology. Currently employed at Intel as a Software Engineer.

Saar Kadosh , BSc Student , Studies Computer & Software Engineering Integrated Circuit and VLSI as areas of expertise at Technion – Israeli Institute of Technology. Currently employed at Apple inc. as an Electronic Engineer.

1.3 About the Advisor/Mentor

Amichai Ben-David , Electrical Engineer. Amichai was a massive contributor to the project, and it couldn't be done without his help. He is currently employed in Nvidia as design engineer and among other things research RISC-V there. Amichai contributes to the project from his free time.

2 General Description

2.1 Project Background

2.1.1 Earlier Projects

To fully understand the LOTR Project, previous knowledge of the two base projects whom the LOTR is based on is required.

2.1.1.1 GPC_4T

implementation of a **RISC-V** 32I base integer **GENERAL PURPOSE COMPUTE UNIT CORE** with 4 hardware Threads (Thus the name GPC_4T) and two **Memory modules** – Data memory and Instruction memory – working with the core.

2.1.1.2 Ring Controller

Also known as RC, is the **arbitration logic for Core & Fabric transactions.** The RC maintains the ordering coherency, starvation & Dead Lock prevention. Contains the **C2F& F2C Buffers** to arbitrate the current Fabric<->Core interface.

Projects documentation can be found here:

https://github.com/amichai-bd/riscv-multi-core-lotr/tree/master/doc

2.1.2 Multi Core Processor

A multi-core processor is a microprocessor on a single integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions.

The instructions are ordinary CPU instructions (such as add, move data, and branch) but the single processor can run instructions on separate cores at the same time, increasing overall speed for programs that support multithreading or other parallel computing techniques.

The Cores in the Multi Core processors can communicate with each other (like accessing memory) by an inter-core communication method based on networking topologies such as bus, **ring**, two-dimensional mesh, and crossbar.

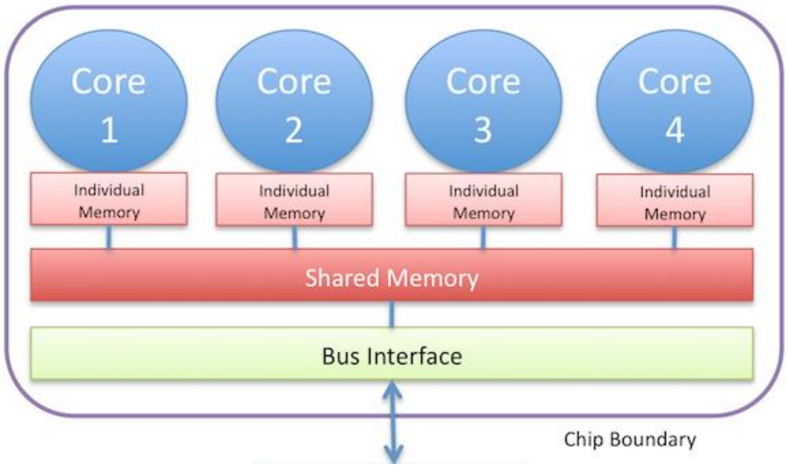


Figure 2 - A basic block diagram of a generic multi-core processor

2.1.3 Ring Architecture

This multi core architecture is based on a Ring Network Topology. Ring network is a network topology in which each node connects to exactly two other nodes, forming a single continuous pathway for signals through each node – a ring. Data travels from node to node, with each node along the way handling every packet.

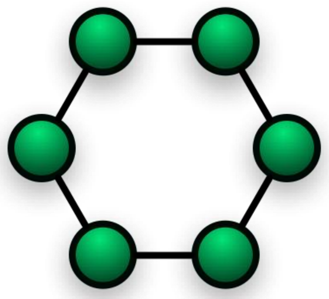


Figure 3 - ring topology

2.1.4 FPGA

FPGA, or a field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence the term field-programmable. The FPGA configuration is generally specified using a hardware description language (HDL), Like System Verilog which we used in this design.

In the project, we used an FPGA kit, **DE10-10Lite** , which is a board contains the FPGA chip and some other I\O units like VGA connector, USB, LEDs, switches, and buttons. More on FPGA integration section.

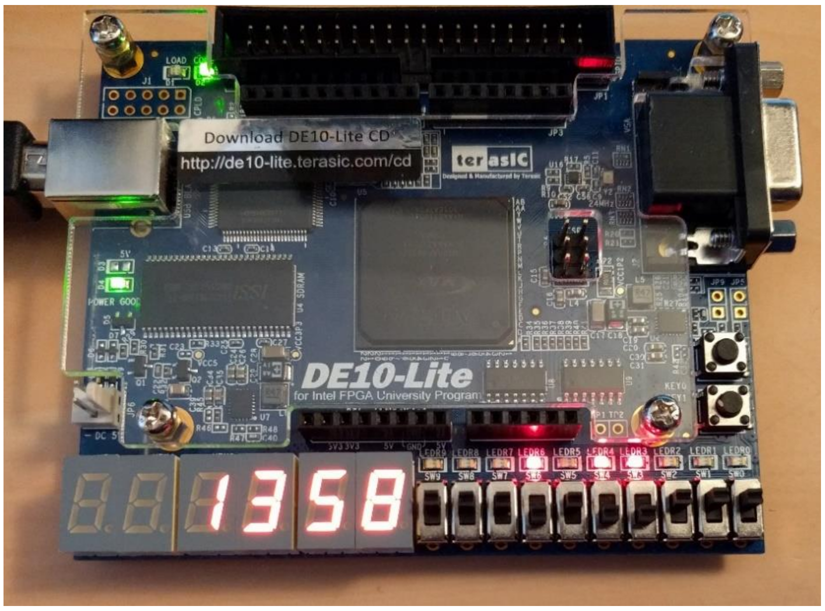


Figure 4 - DE10-Lite kit supplied to us by Avi Salmon

2.2 Top Level Description

LOTR Fabric, which all of this project converse to, is our own implementation of a ring architecture multi General-purpose cores , based on RISC-V32I ISA. Each Core has 4 hardware threads, 16KB of total memory and can ran any program individually.
Each core is located on a LOTR Tile module , alongside the Ring Controller which used to communicate other cores.

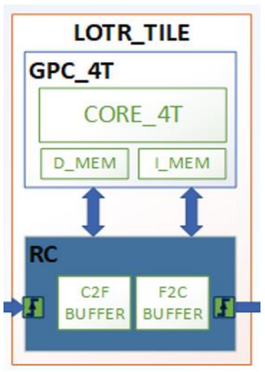


Figure 5 - Lotr Tile

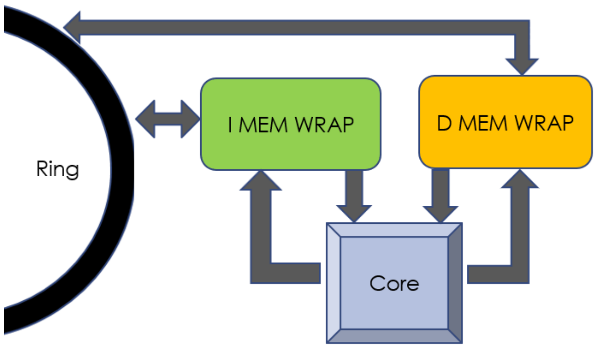


Figure 6 - Very High level GPC_4T block diagram

3 Design Building Blocks

3.1 Tile

The LOTR Tile, also known as the gpc_4t tile is the base building block of the LOTR. It is actually a wrapper for both GPC_4T core and memories, and the Ring controller. Each Tile has its own ID indexed from Tile 1. The tile has the Fabric interface: Ring Request input and output and Ring Response input and output. Both interfaces used to transfer transactions between the outer Rim (Star Wars reference) and the Tile's RC.

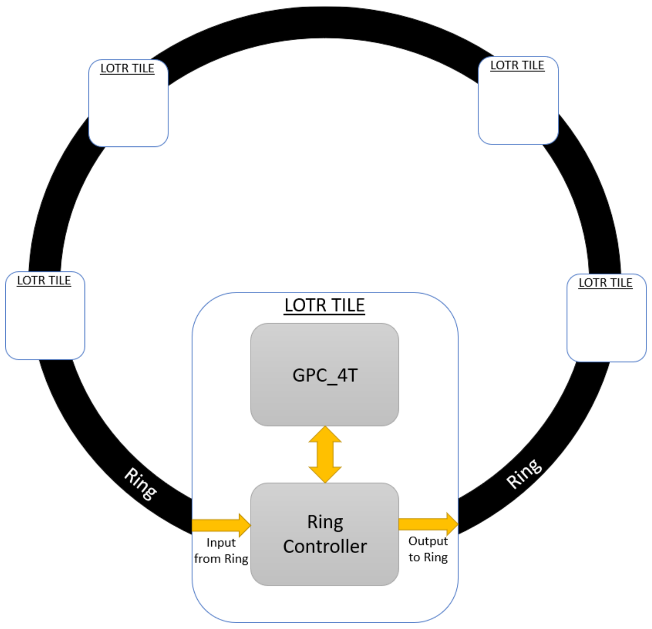


Figure 7 - Lotr Tile in high level

Each transaction coming from the Ring is transferred to the Ring Controller (RC) which determine if the transaction's destination is to this Tile's GPC_4T or not. If yes than the transaction is passed to the Core, and else, the transaction will be passed thru the ring to the next tile, until reaching its destination.

Name	Size	Direction	Description
Fabric(Ring) to Tile			
RingReqInValidQ500H	1	Input	Valid bit for a valid fabric request
RingReqInRequestorQ500H	10	Input	The requestor ID : includes the Core id and the thread id
RingReqInOpcodeQ500H	2	Input	The transaction opcode: read, write, read response
RingReqInAddressQ500H	32	Input	The address of the read\write request, includes the tile id
RingReqInDataQ500H	32	input	The data of the write request
RingRspInValidQ500H	1	Input	Valid bit for a valid fabric response
RingRspInRequestorQ500H	10	Input	The requestor ID : includes the Core id and the thread id
RingRspInOpcodeQ500H	2	Input	The transaction opcode: read response
RingRspInAddressQ500H	32	Input	The address that data response is coming from
RingRspInDataQ500H	32	input	The data of the read response
Tile to Fabric			
RingReqOutValidQ502H	1	output	Valid bit for a valid tile request
RingReqOutRequestorQ502H	10	output	The requestor ID : includes the Core id and the thread id
RingReqOutOpcodeQ502H	2	output	The transaction opcode: read, write, read response
RingReqOutAddressQ502H	32	output	The address of the read\write request, includes the tile id
RingReqOutDataQ502H	1	output	The data of the write request
RingRspOutValidQ502H	1	output	Valid bit for a valid tile response
RingRspOutRequestorQ502H	10	output	The requestor ID : includes the Core id and the thread id
RingRspOutOpcodeQ502H	2	output	Opcode of the Req that is passed to the Fabric: RD_RSP.
RingRspOutAddressQ502H	32	output	The address that data response is coming from
RingRspOutDataQ502H	32	output	Data read for read response.

In our design, due to hardware limitation of the FPGA kit unit, we only used 2 LOTR Tiles.

3.2 RING

The Ring is not an independent module like the Tile, but it's a concept in the LOTR module. The LOTR holds NUM_TILES of the following signals:

Name	Size
RingReqValidQnnnH	1
RingReqRequestorQnnnH	10
RingReqOpcodeQnnnH	2
RingReqAddressQnnnH	32
RingReqDataQnnnH	32
RingRspValidQnnnH	1
RingRspRequestorQnnnH	10
RingRspOpcodeQnnnH	2
RingRspAddressQnnnH	32
RingRspDataQnnnH	32

Each Tile is assigned with all of these set of signals. For example tile 1 will assigned RingReqValidQnnnH[1] and tile 4 will assigned RingReqValidQnnnH[4].
In addition, to complete the Ring formation, the LORT will assign the last ring output to first ring input :

```
assign RingReqValidQnnnH     [1] = RingReqValidQnnnH     [NUM_TILE+1];
assign RingReqRequestorQnnnH[1] = RingReqRequestorQnnnH[NUM_TILE+1];
assign RingReqOpcodeQnnnH    [1] = RingReqOpcodeQnnnH    [NUM_TILE+1];
assign RingReqAddressQnnnH   [1] = RingReqAddressQnnnH   [NUM_TILE+1];
assign RingReqDataQnnnH      [1] = RingReqDataQnnnH      [NUM_TILE+1];
assign RingRspValidQnnnH     [1] = RingRspValidQnnnH     [NUM_TILE+1];
assign RingRspRequestorQnnnH[1] = RingRspRequestorQnnnH[NUM_TILE+1];
assign RingRspOpcodeQnnnH    [1] = RingRspOpcodeQnnnH    [NUM_TILE+1];
assign RingRspAddressQnnnH   [1] = RingRspAddressQnnnH   [NUM_TILE+1];
assign RingRspDataQnnnH      [1] = RingRspDataQnnnH      [NUM_TILE+1];
```

3.3 FPGA INTEGRATION

The implementation of an FPGA SYNTHESIZABILITY was one of the most challenging missions of the project.

3.3.1 FPGA Kit DE10-Lite

The FPGA kit we used is the DE10-Lite that we get from Avi Salmon, Innovation Lead for the Intel Israel Development centers.

At the center of the kit is the FPGA Altera chip.

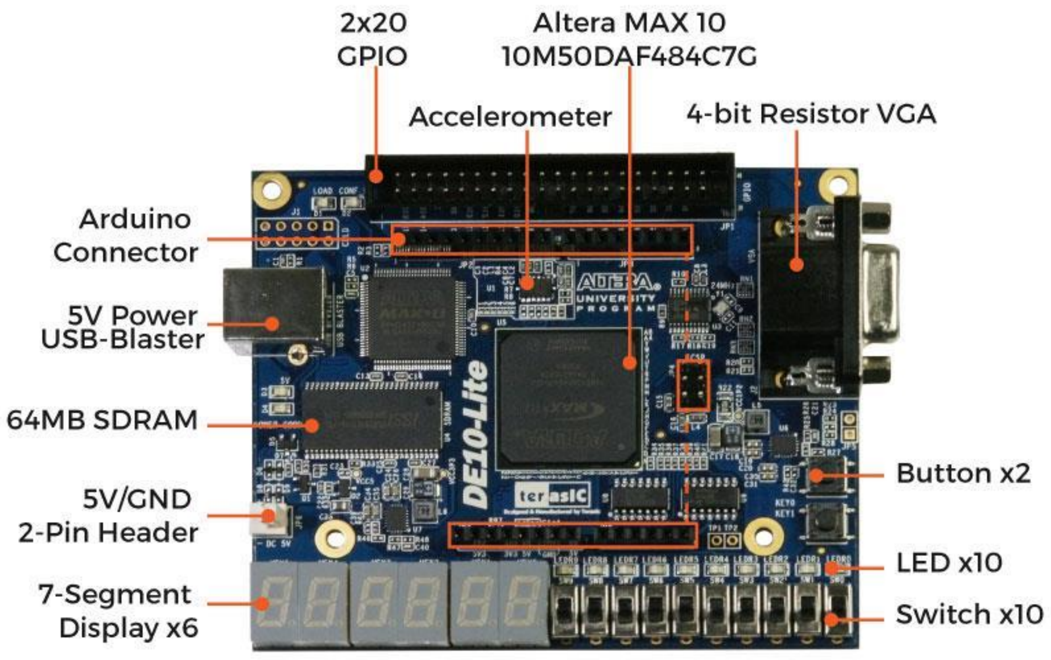


Figure 8 - FPGA kit diagram from www.mouser.co.il

Using Quartus Prime development tool, our entire RTL of the LOTR project from the PC is loaded and synthesized on the FPGA kit thru USB blaster and actually came to life.

3.3.2 FPGA modules hierarchy

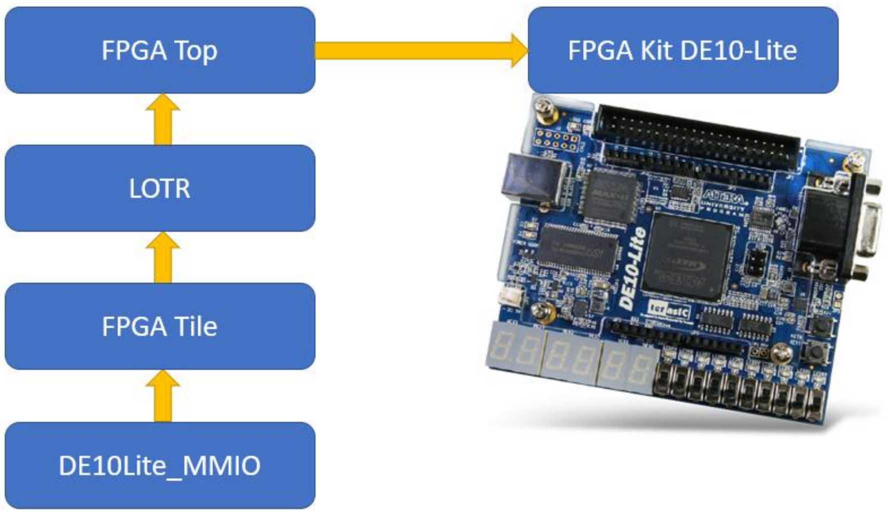


Figure 9 - FPGA tree

3.3.3 DE10Lite_MMIO module

This module functioning as a Core inside a special Tile. Its purpose is to communicate with the FPGA kit I\O.

Beside Clock, reset and core ID signals, the module has the same F2C connections with an RC as a regular GPC_4T core, but no C2F signals as they are redundant .All other inputs and outputs are related to the FPGA kit.

Name	Size	Direction	Description
CLK_50	1	Input	Special clock signal for the VGA memory
QClk	1	Input	The design clock
RstQnnnH	1	Input	Reset signal
CoreID	8	Input	Core id signal coming from the LOTR module
F2C_ReqValidQ502H	1	Input	The data entering the GPC_4T is a Req.
F2C_ReqOpcodeQ502H	1	Input	Opcode of the Req that is passed from the Fabric: RD or WR.
F2C_ReqAddressQ502H	32	Input	The address of the Req in the destination core.
F2C_ReqDataQ502H	32	Input	Data to be written in the destination core (when WR).
F2C_RspValidQ500H	1	output	The data leaving to the Fabric is a Rsp.
F2C_RspOpcodeQ500H	1	output	Opcode of the Rsp that is passed to the Fabric: RD or WR.
F2C_RspAddressQ500H	32	output	The address of the Rsp in the destination core.
F2C_RspDataQ500H	32	output	Data to be written in the destination core (when WR).
Button_0,	1	input	Button 0 from the FPGA kit
Button_1,	1	input	Button 1 from the FPGA kit
Switch,	10	input	Switches from the FPGA kit
Arduino_dg_io	16	input	Arduino inputs from FPGA kit
SEG7_0	8	output	7 segments of the FPGA kit
SEG7_1	8	output	
SEG7_2	8	output	
SEG7_3	8	output	
SEG7_4	8	output	
SEG7_5	8	output	
RED,	4	output	VGA signals
GREEN,	4	output	
BLUE,	4	output	
v_sync	1	output	
h_sync	1	output	
LED	10	output	LED outputs from FPGA kit

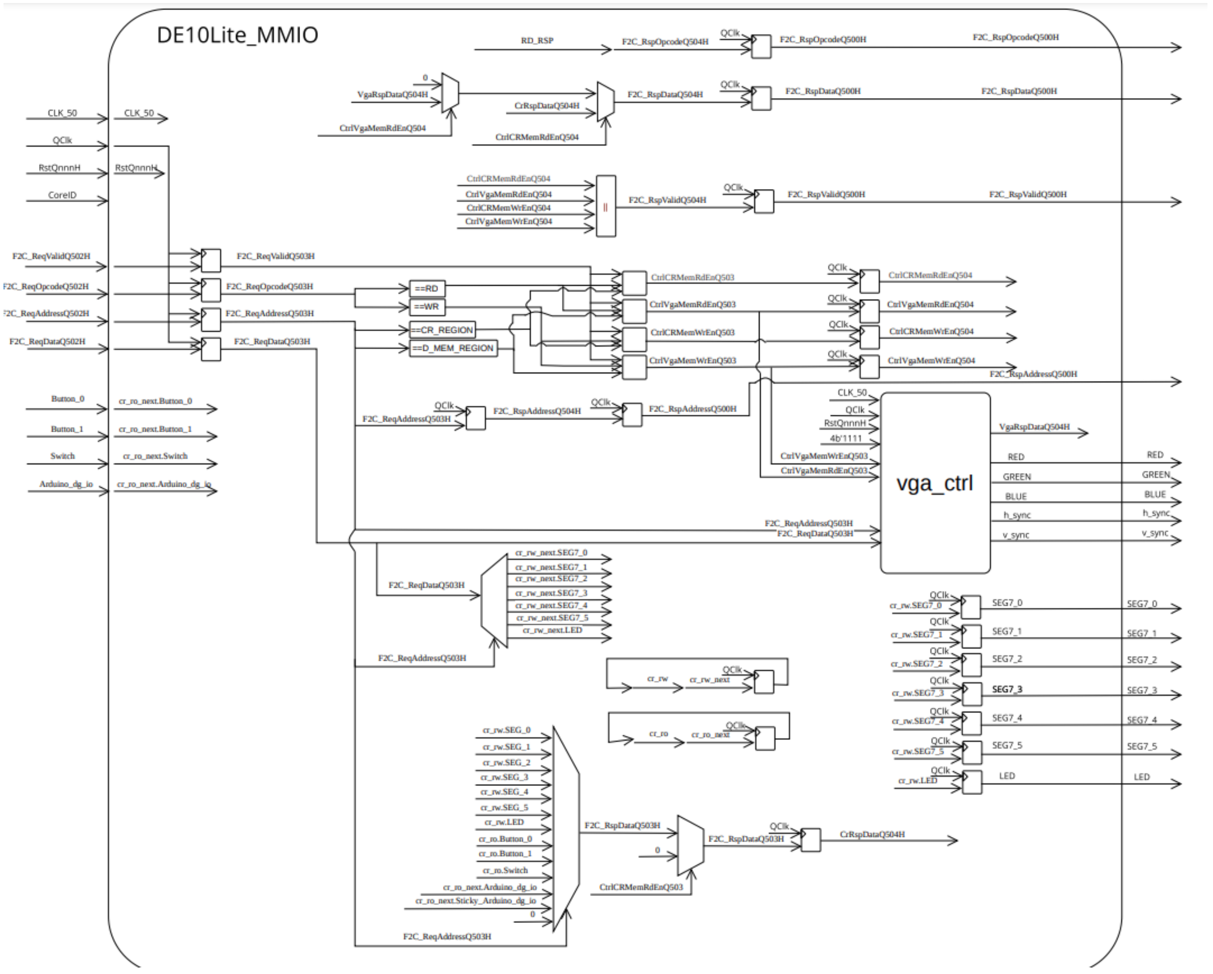


Figure 10 - FPGA MMIO module logic design

Request from regular LOTR Tiles coming to the module thru the RC. All the requests to the FPGA are saved in special Control Registers (CR) on the module, and on the next clock cycle they are all passing to higher level up to the Top level and the FPGA kit.

Easier to understand with an example : lets say Core 1 want to turn on the LEDS of the FPGA kit. It will sent write request (lets say 00000001) to the address : 0x03C02018. Bits 31:24 are for the Tile id. In our design the FPGA tile ID is 3. Bits 23:21 are for the memory region. 0xC is for the CR region. 0x2018 are the offset for the LED CR. The date '0x1' will passed directly to the Top level and the FPGA kit will switch the LED to 1.

The DE10Lite_MMIO also instantiate the VGA modules. More on the VGA in the relevant section.

3.3.4 FPGA Tile

The FPGA tile is a wrapper for the DE10Lite_MMIO module and an RC module and its acting similar to a regular LOTR tile.

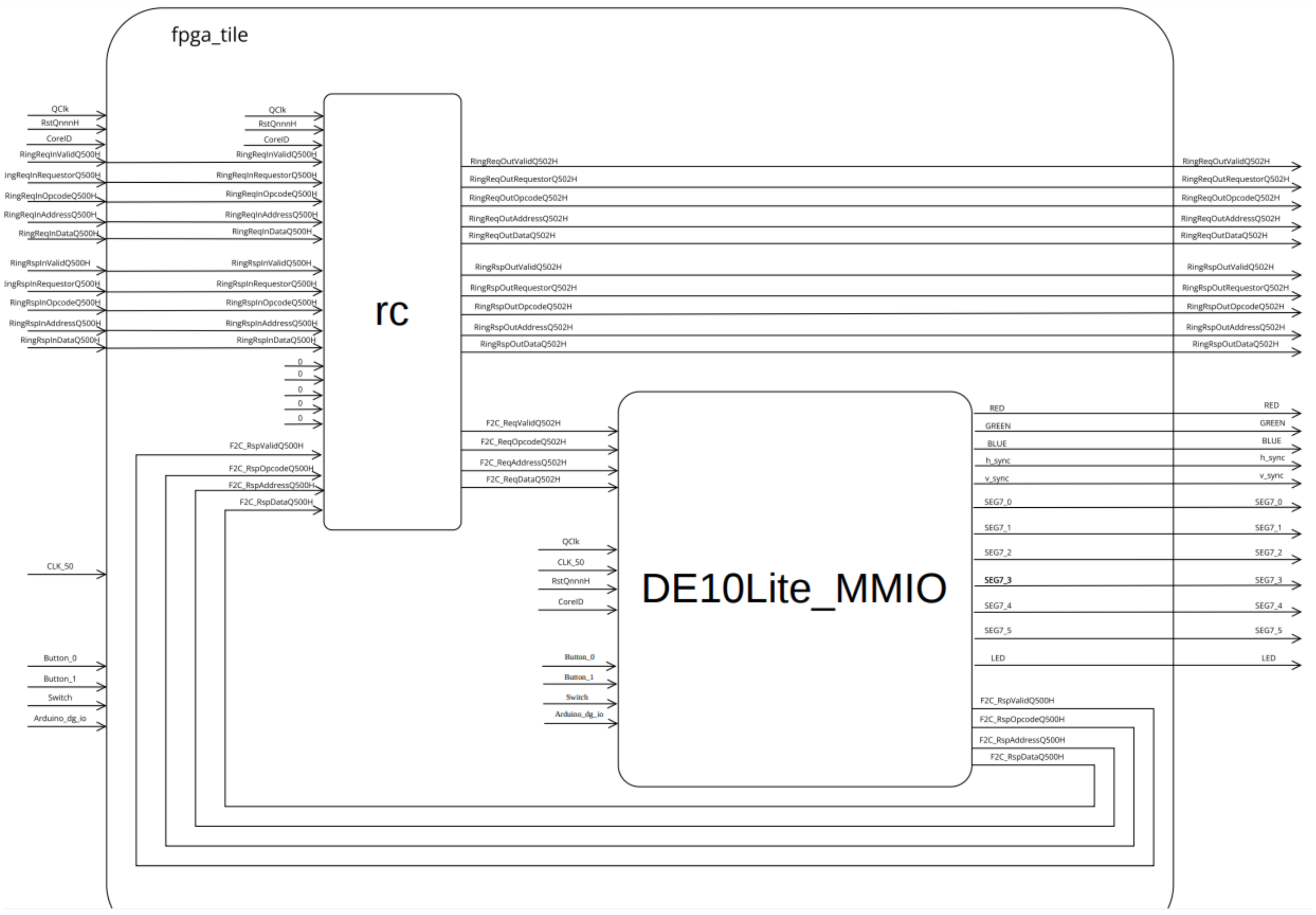


Figure 11 - FPGA tile logic design

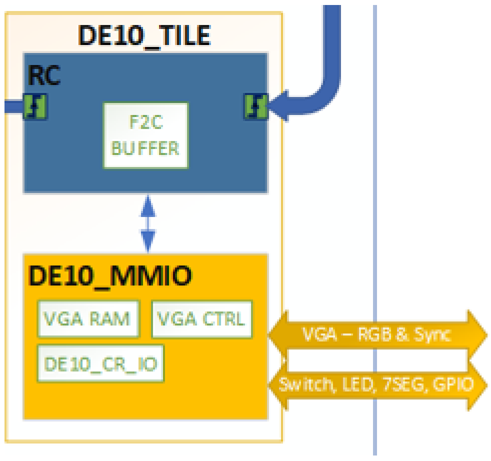


Figure 12 - FPGA tile

3.3.5 FPGA CR's

Offset	Name	Type	Description	Structure
2000	CR_SEG7_0	RW	Hold data pass to 7 segment 0	[7:0]
2004	CR_SEG7_1	RW	Hold data pass to 7 segment 1	[7:0]
2008	CR_SEG7_2	RW	Hold data pass to 7 segment 2	[7:0]
200c	CR_SEG7_3	RW	Hold data pass to 7 segment 3	[7:0]
2010	CR_SEG7_4	RW	Hold data pass to 7 segment 4	[7:0]
2014	CR_SEG7_5	RW	Hold data pass to 7 segment 5	[7:0]
2018	CR_LED	RW	Hold data pass to FPGA kit LEDs	[9:0]
201c	CR_Button_0	RO	Hold value of the kit's button 0	
2020	CR_Button_1	RO	Hold value of the kit's button 1	
2024	CR_Switch	RO	Hold value of the kit's 10 switches	[9:0]
2028	CR_Arduino_dg_io	RW	Arduino interface	[15:0]
202C	CR_Sticky_Arduino_dg_io	RW	Arduino interface	

3.3.6 Top level design

The top-level design is the highest in the hierarchy for the entire synthesizable design. It is the module that "speaks" directly to the FPGA kit. This module contains the LOTR instance, a PLL to transform clocks frequencies and inputs and outputs for the FPGA kit.

3.3.6.1 Memory

Two 8KB SRAM memories are used on the FPGA kit. One for Instruction memory and other for data memory.

The memory configuration is 32 bit words, with byte enable, and using the Little Endian method.

The design is also loading Memory MIF files to be the initial memory. This is the way we load a program (instructions) to the design . each time we want the design to run different code of instructions, we need to switch the MIF file and re Compile the design (analysis, synthesis and elaboration) and it takes a lot of time. In the future the project will have a UART tile that will easily load instruction and data memories, without re-compiling the design.

3.3.6.2 PLL

The design uses PLL - Phase-locked loop to regulate clock signals. the top level design works with 50Mhz clock frequency but in the future we might need a PLL if some modules would need a slower clock, so the PLL generates 5Mhz clock from the 50Mhz clock.

3.4 VGA

In our opinion, the VGA ability was the highlight of the Project. The VGA complete modules was developed by our advisor and some students for Bar Illan's University. The module contains 3 main pieces: VGA controller, VGA Memory and VGA sync generator.

3.4.1 VGA controller

The resolution of our target screen is 640×480. As can be seen from .שגיאה! מקור ההפניה לא נמצא, there are 640 vertical bits lines and 480 horizontal bits lines. Overall, the screen contains 80×480 = 38,400 bytes, and if we take into account that the size of a word is 4 bytes, the screen contains 9600 words. Each pixel on the screen is represented by a single bit, as a result, the total number of pixels is 640×480 = 307,200 pixels. The VGA support 12 bit RGB, but due to constrains, we implemented a monochromatic screen, which each pixel is either on or off.

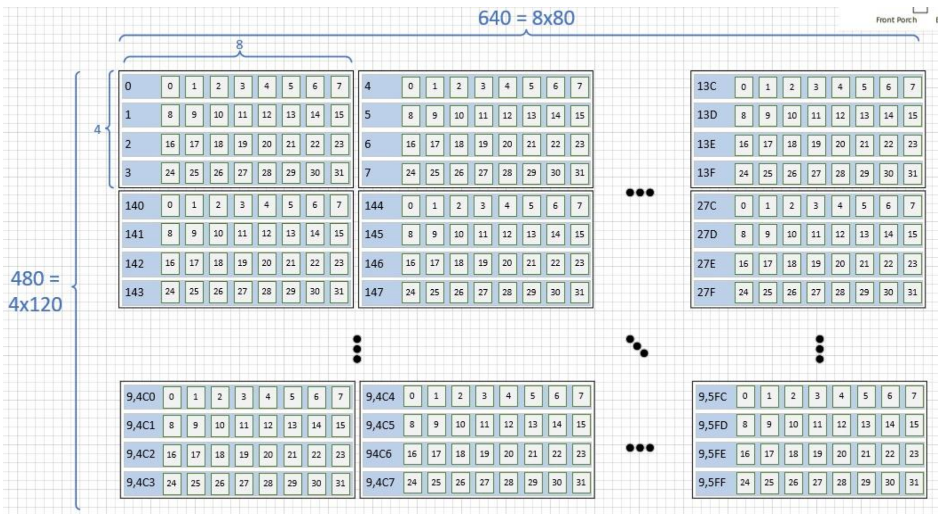


Figure 13 - the monitor

VGA at 640×480 resolution at 60Hz (frames per second) requires a pixel clock of 25.175Mhz, this is the industry standard. The tables below show the necessary timing for the VGA control signals at 640×480 resolution at 60Hz. You can view additional timings for different cases in the following link.

General timing

Parameter	Timing
Screen refresh rate	60 HZ
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Table 1 - VGA general timing

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Table 2 - VGA horizontal timing

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.31777557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.0486593843098
Whole line	525	16.683217477656

Table 3 - VGA vertical timing

To understand how the transmission to the screen is carried out, the first two signals that deserve mention and explanation are the **horizontal and vertical synchronization signals**.

Horizontal Synchronization

Figure below shows the wave diagram of the horizontal synchronization signal. We will look at the hsync (horizontal synchronization) signal and how it relates to how the electron beam of the CRT monitor traverses across the CRT monitor. In Figure below we can see a CRT monitor with a black border, and the reason is because older CRT monitors did have that black border and it is part of the scanning pattern that we do. That is, we do pass the electron beam across the black border too, however we don't show any pixels over there.

The hsync signal is a periodic digital signal that goes between zero and one logic. Based on the value of that signal the CRT monitor generates a sawtooth signal which moves the electron beam across the screen. The period of hsync signal is between 0 all the way to 799, which is 800. Suppose we start at the point marked 0 in the wave diagram of the hsync signal in שגיאה! מקור ההפניה לא נמצא., in principle we can start at any point we want because the signal is periodic. Point 0 is the start of the screen, this point located in the visible portion of the screen, not the border. We set the hsync signal to 1 all the way to 639 which means the electron beam will travers across the CRT monitor from left to right. We can observe that the sawtooth signal in this section is going all the way up, but it didn't finish though. What is happens next is that the electron beam goes through the right border (front porch) for exactly 16 clock cycles (not the system clock cycle). Then, the horizontal line must go backwards for 96 clock cycles, this is what we call the retrace and the sawtooth signal in this section is going all the way down to zero. In the final stage, relative to the point where we started (0) and because the signal is periodic the electron beam goes through the left border (back porch) for exactly 48 clock cycles. Realization of the hsync signal is basically by a simple counter which goes from 0 to 799 (modulo 800).

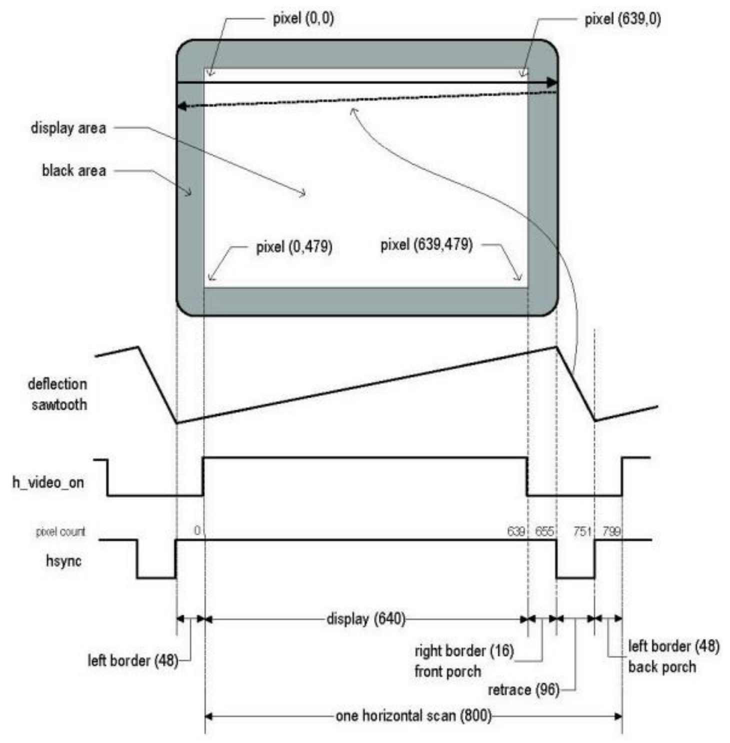


Figure 14 - Horizontal Synchronization

Vertical Synchronization

Fig. below shows the wave diagram of the vertical synchronization signal. Now, we will look at the vsync (vertical synchronization) signal and how it relates to how the electron beam of the CRT monitor traverses across the CRT monitor. The horizontal synchronization signal allowed us to move the electron beam or the scanning of the pixels from left to right, so we need another signal which goes from top to bottom, this is where the vertical synchronization signal comes into play. We want to go from top to bottom in 1/60 seconds because we have a refresh rate of 60Hz.

vsync signal is very similar to hsync signal so the vsync signal is also a periodic digital signal that goes between zero and one logic. Suppose we start at the point marked 0 in the wave diagram of the vsync signal in שגיאה! מקור ההפניה לא נמצא.below , here also we can start at any point we want because the signal is periodic. Point 0 is the start of the screen, this point located in the visible portion of the screen, not the border. We set the vsync signal to 1 all the way to 479 which means the electron beam will travers across the CRT monitor from top to bottom. What is happens next is that the electron beam goes through the bottom border (front porch) for exactly 10 clock cycles (not the system clock cycle). Then, the vertical line must go backwards for 2 clock cycles, this is the retrace. In the final stage, relative to the point where we started (0) and because the signal is periodic the electron beam goes through the top border (back porch) for exactly 33 clock cycles. Realization of the vsync signal is also basically by a simple counter which goes from 0 to 524 (modulo 525).

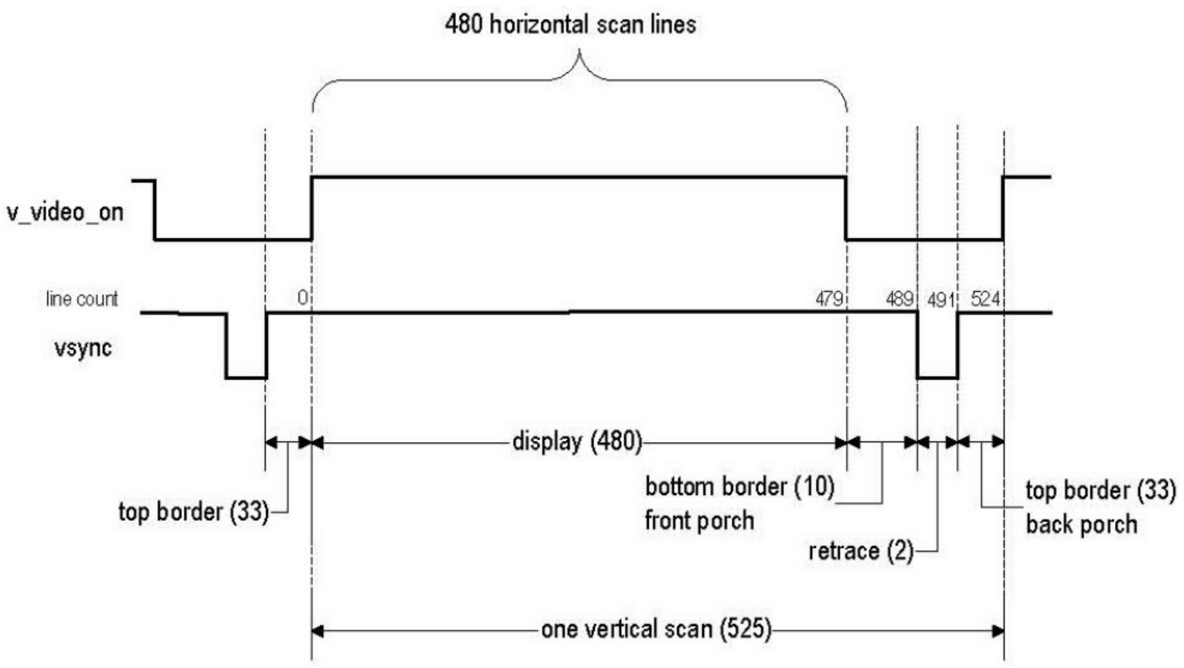


Figure 15 - Vertical Synchronization

Now, let's look at the complete picture. That is, let's consider the horizontal and vertical synchronization signals together. Figure below shows the complete scanning pattern of the screen.

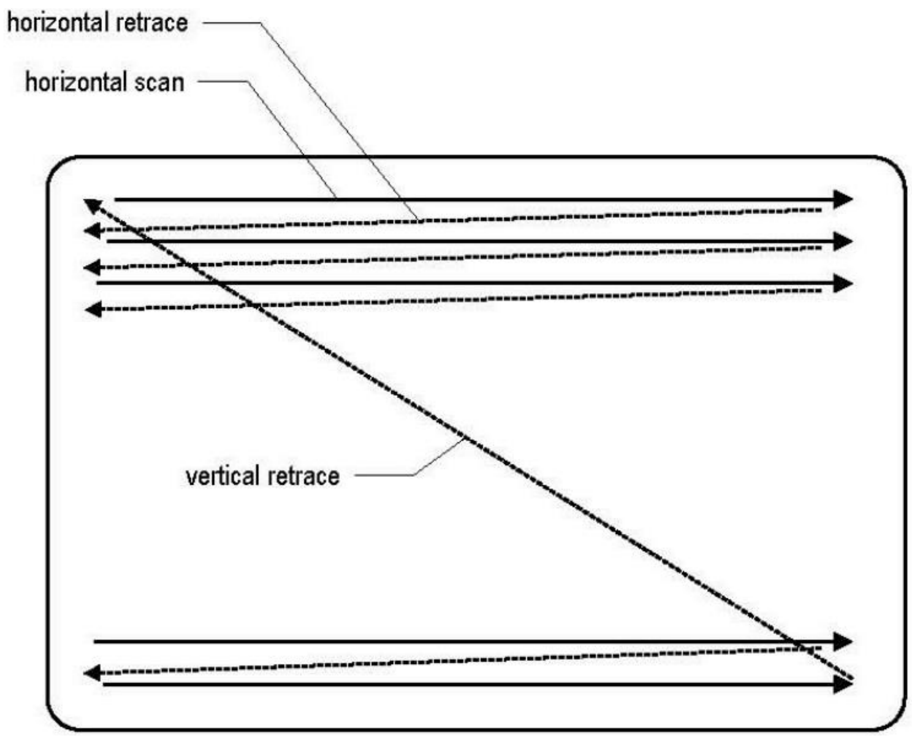


Figure 16 - Screen scanning pattern

3.4.2 VGA Memory

VGA_MEM is the memory region that responsible for communicating with the screen. It is also S-RAM based and its size is 32.5kb. Any writing to a certain area in this memory region will be reflected to the screen and in addition to that the processor can read the status of a certain pixel assuming that the software is interested in it. There is an option for this memory area to be initialized to zero by the software that run on the processor if it so wishes. The memory size is control by the parameter VGA_MEM_MSB and its basic size is 32.5kb. The access to VGA_MEM is a dual-port access because both the processor and the VGA controller can access this memory area together. The processor accesses VGA_MEM in the Memory stage. The driver that pulls the information from this memory region to the screen is the VGA Controller. In the simulation we used behavioral memory, and on the FPGA, we used the on-die FPGA memory (as same as I_MEM and D_MEM).

All the memory regions that have been described in GPC_4T project are wrapped by mem_wrap so that at the level of this component the address is checked, and the reference is made to the relevant memory space. In relation to VGA_MEM, this memory component is wrapped by the VGA Controller and the VGA Controller is a component of mem_wrap. That is, every request to the memory area of the VGA goes through the processor, passes through the mem_wrap to the VGA Controller and then reaches the memory area of the VGA.

3.5 Memory Addressing

The design is using both virtual and physical memory addressing. The physical address of each byte is 32 bits in a designated pattern.
Each memory transaction has a 32-bit address.
The address may be a "Local" address or a "Peer" address.

Core ID	Region	Reserved	Memory Offset
Address[31:24]	Address[23:22]	Address[21:14]	Address[11:0]

In the SoC level, the address must contain a non-zero 8 MS bits, indicating of the Core ID. On the Core level, an address can have zero on bits 31:24, indicating it’s a virtual address for this local memory regions. It means that each byte has a physical address that any core can access to, and a virtual address that only the local core can use.
For example : Core 3 can use address 0x03010001 to access the local data memory region for the first byte, and can use address 0x00010001 to access the same byte!

3.6 Unique ID CR

Each thread of each Core has a special identifier. Using our special CR's, in the pre compilation, this identifier is loaded to a Specific register.

The identifier is a 10 bits combination of the core id [9:2] and thread id [1:0]. For example thread 1 from core 2 has the identifier : ***0b'0000001001.***

Each thread from each core In the code, loads its identifier to CR in offset 0x0.

4 Validation Plan

4.1 Introduction and Background

Validation is one of the most important procedures at product manufacturing and quality assurance. Validation definition is the assurance that a product, System, or service is working as expected.
In the GPC_4T project we detailed a lot about our validation plan. The entire validation plan of GPC_4T was used on this project, so we will only elaborate on new validation features we used on LOTR. It is highly recommended to read GPC_4T validation plan

4.2 Alive Multi Core

The first verification we made after building the basic 2 Core Lotr model is a simple PoC "alive test" c program. In this program we wanted to verify that the ring is functional.

In the program ,thread 0 from Core 1 preforming write transaction of an integer (0x51) to address 0x02400900. This address space belongs to non-local core therefore the transaction will go to the Ring.

```
#define SCRATCHPAD0_CORE_1  ((volatile int *) (0x01400900))
#define SCRATCHPAD0_CORE_2  ((volatile int *) (0x02400900))
#define SCRATCHPAD0_CORE    ((volatile int *) (0x00400900))

int main() {
    int ThreadId = CR_THREAD[0];
    int UniqeId = CR_WHO_AM_I[0];
    int counter = 0 ;
    switch (UniqeId) //the CR Address
    {
        case 0x4 : // thread 0 core 1

                SCRATCHPAD0_CORE_2[0] = 0x51;               //Writing to
Core2

                while(counter++ < 10){};          // busy wait until data
arrived from Core2

                SHARED_SPACE[0] = SCRATCHPAD0_CORE_2[0];    //Reading from
core 2
        break;

        case 0x5 :
            while(1);
        break;

        case 0x6 :
            while(1);
        break;
        case 0x7 :
            while(1);
        break;

        case 0x8 :
            while(1);
        break;

        case 0x9 :
            while(1);
        break;

        case 0xa :
            while(1);
        break;

        case 0xb :
            while(1);
        break;
    }

    return 0;
```

Later, a small delay is happening, to let Core 2 receive the transaction from the Ring.

Finally, a read request for address 0x02400900 to validate that the correct integer has been read. All other 7 threads are doing busy wait.

4.3 Simulation Test Bench

Like GPC_4T, a System Verilog test bench created to simulate the design using ModelSim.
The test bench is quite similar to GPC_4T test bench, with **Back door Memory** loading from a file. In the test bench we generate a clock and reset signals and creates an instance of the LOTR module, with 2 Lotr Tiles, and 1 FPGA tile. Each of the 2 Cores had its own memories with the back door load.

```
    ////TILE 1
        // Backdoor load the Instruction memory
        lotr_tb.lotr.gpc_4t_tile_1.gpc_4t.i_mem_wrap.i_mem.next_mem =
IMemQnnnH[I_MEM_OFFSET+SIZE_I_MEM-1:0];
        lotr_tb.lotr.gpc_4t_tile_1.gpc_4t.i_mem_wrap.i_mem.mem      =
IMemQnnnH[I_MEM_OFFSET+SIZE_I_MEM-1:0];
        // Backdoor load the Inst1uction memory
        lotr_tb.lotr.gpc_4t_tile_1.gpc_4t.d_mem_wrap.d_mem.next_mem =
DMemQnnnH[D_MEM_OFFSET+SIZE_D_MEM-1:D_MEM_OFFSET];
        lotr_tb.lotr.gpc_4t_tile_1.gpc_4t.d_mem_wrap.d_mem.mem      =
DMemQnnnH[D_MEM_OFFSET+SIZE_D_MEM-1:D_MEM_OFFSET];
    ////TILE 2
        // Backdoor load the Instruction memory
        lotr_tb.lotr.gpc_4t_tile_2.gpc_4t.i_mem_wrap.i_mem.next_mem =
IMemQnnnH[I_MEM_OFFSET+SIZE_I_MEM-1:0];
        lotr_tb.lotr.gpc_4t_tile_2.gpc_4t.i_mem_wrap.i_mem.mem      =
IMemQnnnH[I_MEM_OFFSET+SIZE_I_MEM-1:0];
        // Backdoor load the Inst2uction memory
        lotr_tb.lotr.gpc_4t_tile_2.gpc_4t.d_mem_wrap.d_mem.next_mem =
DMemQnnnH[D_MEM_OFFSET+SIZE_D_MEM-1:D_MEM_OFFSET];
        lotr_tb.lotr.gpc_4t_tile_2.gpc_4t.d_mem_wrap.d_mem.mem      =
DMemQnnnH[D_MEM_OFFSET+SIZE_D_MEM-1:D_MEM_OFFSET];
```

After instancing the LOTR model and loading an Instruction memories, the Test bench is good to go and the LOTR can be simulated.

At the end of the TB code there is a task that initiate the ending of the simulation by creating a snapshot of the data memory and the GPC_4T registers to a text file , several of informative massage display that will be printed in simulation program and closing all open newly created log files that created by the log generator.

The simulation initiates using a power shell script performing commands on the ModelSim using lotr_list.f file, the same logic as the GPC_4T and you can read it there.

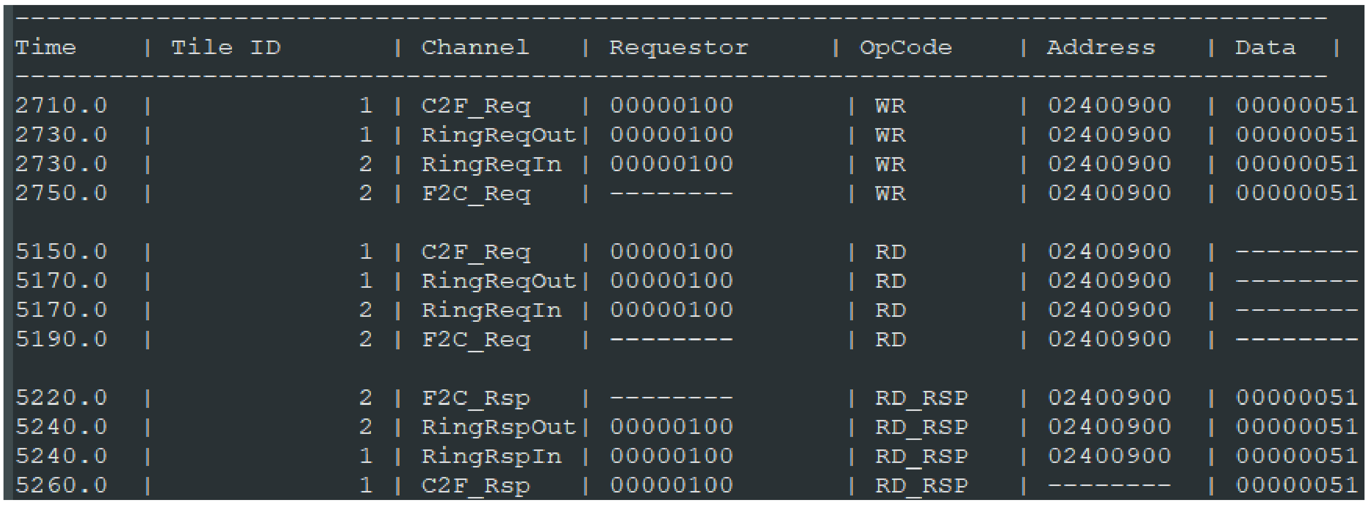
To run the simulation use PowerShell :

.\ lotr_tb_sim.ps1 <TEST_NAME>

4.4 RC LOG

In the GPC_4T test bench there were a lot of logs after a simulation. This the log generates only 2 snapshots of the shared memory for each core.

The unique log for this test bench is the "RC_Tracker" which sniffs all the transactions leaving the Tile for the Ring.



```
-----------------------------------------------------------------------------------------
Time     | Tile ID         | Channel    | Requestor      | OpCode     | Address    | Data   |
-----------------------------------------------------------------------------------------
2710.0   |               1 | C2F_Req    | 00000100        | WR         | 02400900   | 00000051
2730.0   |               1 | RingReqOut| 00000100        | WR         | 02400900   | 00000051
2730.0   |               2 | RingReqIn  | 00000100        | WR         | 02400900   | 00000051
2750.0   |               2 | F2C_Req    | --------        | WR         | 02400900   | 00000051

5150.0   |               1 | C2F_Req    | 00000100        | RD         | 02400900   | --------
5170.0   |               1 | RingReqOut| 00000100        | RD         | 02400900   | --------
5170.0   |               2 | RingReqIn  | 00000100        | RD         | 02400900   | --------
5190.0   |               2 | F2C_Req    | --------        | RD         | 02400900   | --------

5220.0   |               2 | F2C_Rsp    | --------        | RD_RSP     | 02400900   | 00000051
5240.0   |               2 | RingRspOut| 00000100        | RD_RSP     | 02400900   | 00000051
5240.0   |               1 | RingRspIn  | 00000100        | RD_RSP     | 02400900   | 00000051
5260.0   |               1 | C2F_Rsp    | 00000100        | RD_RSP     | --------   | 00000051
```

Figure17 - Ring tracker log

4.5 ALIVE FPGA LED COUNTER

After we finished (with tons of help from our advisor Amichai) the fpga rtl creation and integration, pin planning and memories, we had to create a test that will check PoC for the FPGA kit. A good indication was to turn the FPGA LEDs on and off using the software. The LEDS represented on the FPGA MMIO module as 10 bits (10 LEDS).

In the test, thread 0 of Core 1 will write an increasing number to the CR of the MMIO tile that pass the data to the FPGA kit, in an infinite loop. The numbers should be represented on the LEDS. for example, when the program writes the number "5" to the MMIO FPGA module, the FPGA LEDS will show only the right most LED on and third most right on, and others off

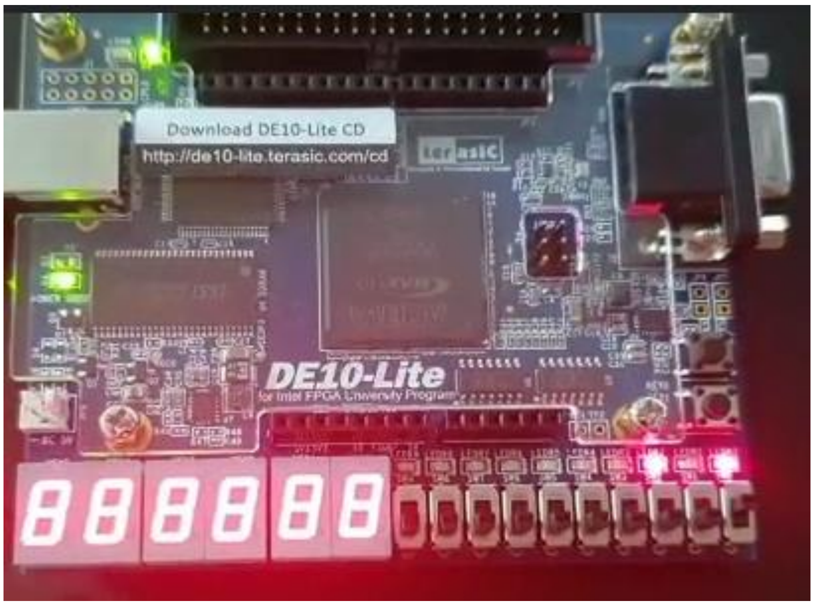

Download DE10-Lite CD
http://de10-lite.terasic.com/cd
terasic
DE10-Lite
for Intel FPGA University Program

Figure 18 - Alive FPGA LED counter showing the number 5

Thus, we have created a LED counter.

When we saw the LED lights moving as a binary counter, we knew that the FPGA is alive.

Figure19 - Dr frankenstein happy to inform

4.6 Alive VGA & VGA log

The final phase of the project was to integrate with a VGA monitor. As the VGA display function in our design as a memory, we simply needed to write 0xffffffff on a diagonal 2 dimensional array. The VGA modules holds the VGA memory and transferred it to the FPGA kit and from there to a monitor. Before connecting a monitor we created another log generator for the display that will generate a snapshot of the VGA memory.

```
#include "LOTR_defines.h"
#define VGA_MEM_OFFSET 3072
```

```
int main() {
    int ThreadId = CR_THREAD[0];
    int UniqeId = CR_WHO_AM_I[0];
    int counter = 0 ;
    int i;
    switch (UniqeId) //the CR Address
    {
        case 0x4 : // parameterize
            for ( i = 0 ; i < 120 ; i++) {
                VGA_FPGA[VGA_MEM_OFFSET + i + 80*i] = 0xffffffff;

            }
        break;
        default :
                while(1);
                break;

    }

    return 0;

}
```

The program flow is simply writing on a diagonal form to the VGA memory.
in the figure is the entire VGA memory snapshot on notepad++ in extra zoom out

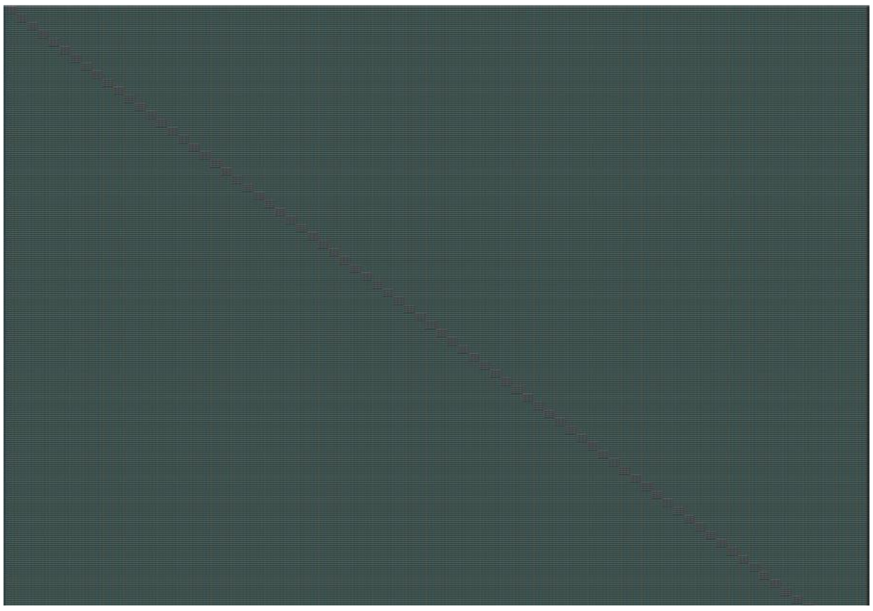


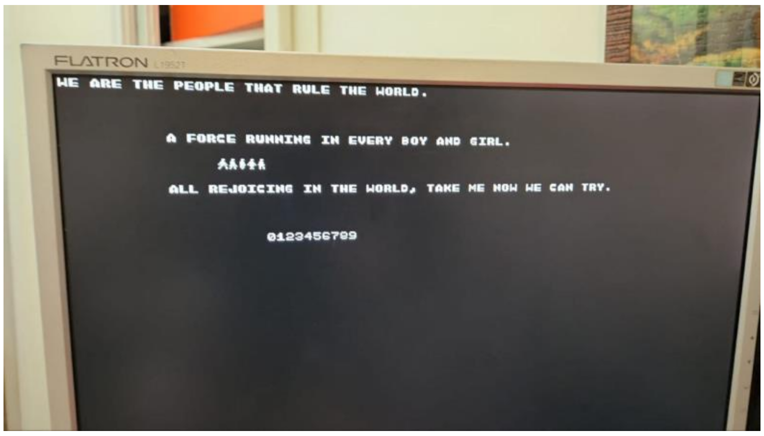
Figure 20 - a diagonal line on the screen log file

After completing the alive VGA, using a library of print functions some students from Bar Ilan university wrote, our advisor helped us to write a full VGA display test what will write a text to the monitor using the print functions that fits the VGA memory logic :

```
#include "graphic.h"

int main()
```

```
{
    int UniqeId  = CR_WHO_AM_I[0];
    switch (UniqeId) //the CR Address
    {
        case 0x4 : // Core 1 Thread 0
        rvc_printf("WE ARE THE PEOPLE THAT RULE THE WORLD.\n");
        draw_symbol(0, 15, 15);
        draw_symbol(1, 15, 16);
        draw_symbol(2, 15, 17);
        draw_symbol(3, 15, 18);
        draw_symbol(4, 15, 19);
        case 0x5 : // Core 1 Thread 1
        set_cursor(10,0);
        rvc_printf("A FORCE RUNNING IN EVERY BOY AND GIRL.\n");
        case 0x6 : // Core 1 Thread 2
        set_cursor(20,0);
        rvc_printf("ALL REJOICING IN THE WORLD, TAKE ME NOW WE CAN TRY.\n");
        case 0x7 : // Core 1 Thread 3
        set_cursor(30,0);
        rvc_printf("0123456789\n");
        case 0x8 : // Core 2 Thread 0
        draw_symbol(0, 10, 15);
        draw_symbol(1, 10, 16);
        draw_symbol(2, 10, 17);
        draw_symbol(3, 10, 18);
        draw_symbol(4, 10, 19);
        default :
            while(1);
        break;
    }// case


    return 0;
}
```

And the result:

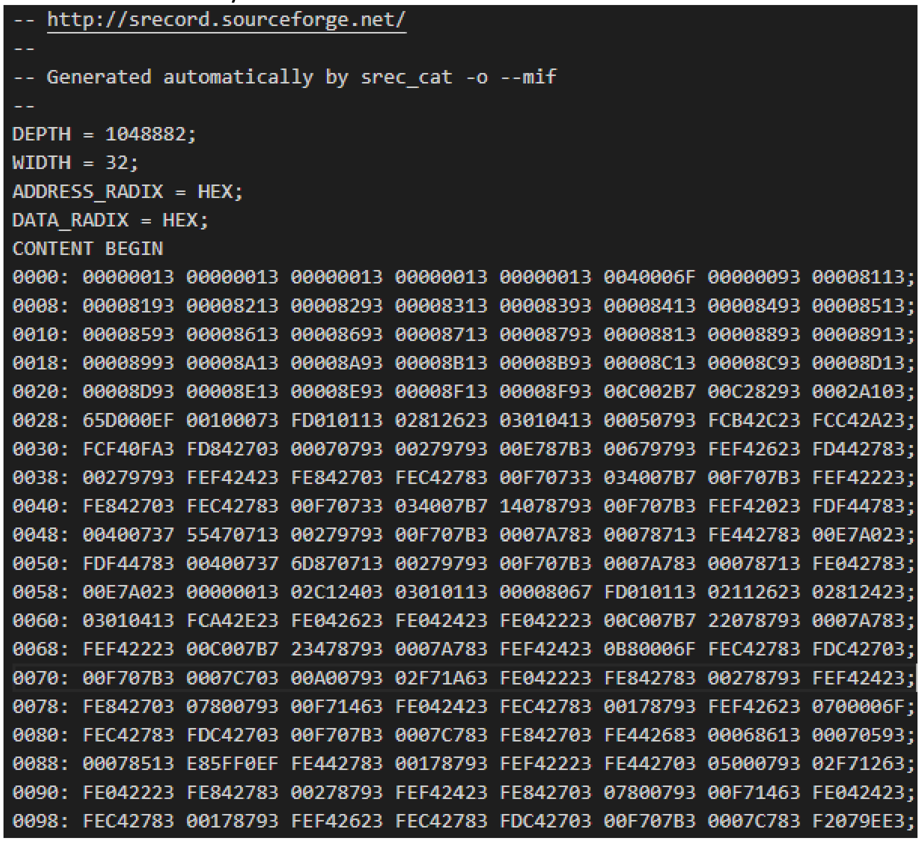


```
WE ARE THE PEOPLE THAT RULE THE WORLD.

A FORCE RUNNING IN EVERY BOY AND GIRL.

ALL REJOICING IN THE WORLD, TAKE ME NOW WE CAN TRY.

0123456789
```

Figure 21 - graphic test on VGA monitor

4.7 Memory Initialization File

The Quartus Prime can use MIF or HEX file to load as an initial memory.
MIF – Memory Initialization File, is An ASCII text file (with the extension .mif) that specifies the initial content of a memory block (RAM or ROM), that is, the initial values for each address. This file is used during project compilation and/or simulation. You can create a Memory Initialization File in the Memory Editor, the In-System Memory Content Editor, or the Quartus® Prime Text Editor.
A Memory Initialization File contains the initial values for each address in the memory. A separate file is required for each memory block.



```
-- http://srecord.sourceforge.net/
--
-- Generated automatically by srec_cat -o --mif
--
DEPTH = 1048882;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT BEGIN
0000: 00000013 00000013 00000013 00000013 00000013 0040006F 00000093 00008113;
0008: 00008193 00008213 00008293 00008313 00008393 00008413 00008493 00008513;
0010: 00008593 00008613 00008693 00008713 00008793 00008813 00008893 00008913;
0018: 00008993 00008A13 00008A93 00008B13 00008B93 00008C13 00008C93 00008D13;
0020: 00008D93 00008E13 00008E93 00008F13 00008F93 00C002B7 00C28293 0002A103;
0028: 65D000EF 00100073 FD010113 02812623 03010413 00050793 FCB42C23 FCC42A23;
0030: FCF40FA3 FD842703 00070793 00279793 00E787B3 00679793 FEF42623 FD442783;
0038: 00279793 FEF42423 FE842703 FEC42783 00F70733 034007B7 00F707B3 FEF42223;
0040: FE842703 FEC42783 00F70733 034007B7 14078793 00F707B3 FEF42023 FDF44783;
0048: 00400737 55470713 00279793 00F707B3 0007A783 00078713 FE442783 00E7A023;
0050: FDF44783 00400737 6D870713 00279793 00F707B3 0007A783 00078713 FE042783;
0058: 00E7A023 00000013 02C12403 03010113 00008067 FD010113 02112623 02812423;
0060: 03010413 FCA42E23 FE042623 FE042423 FE042223 00C007B7 22078793 0007A783;
0068: FEF42223 00C007B7 23478793 0007A783 FEF42423 0B80006F FEC42783 FDC42703;
0070: 00F707B3 0007C703 00A00793 02F71A63 FE042223 FE842783 00278793 FEF42423;
0078: FE842703 07800793 00F71463 FE042423 FEC42783 00178793 FEF42623 0700006F;
0080: FEC42783 FDC42703 00F707B3 0007C783 FE842703 FE442683 00068613 00070593;
0088: 00078513 E85FF0EF FE442783 00178793 FEF42223 FE442703 05000793 02F71263;
0090: FE042223 FE842783 00278793 FEF42423 FE842703 07800793 00F71463 FE042423;
0098: FEC42783 00178793 FEF42623 FEC42783 FDC42703 00F707B3 0007C783 F2079EE3;
```

Figure 22 - MIF file

For generating a MIF file for a program, first we created HEX file from the generated ELF file that the GCC RISC V compiler has created using the command

```
riscv32-unknown-elf-objcopy --srec-len 1 --output-target=ihex file.elf
file.hex
```

From the HEX file, the MIF file creation made with the command:

```
srec_cat file.hex -Intel -o file.mif -Memory_Initialization_File 32
```

after these two commands, 2 MIF file from a elf (risv v decoded C program) is created.

4.7.1 Big & Little Endian

Last thing need to be done prior loading the MIF file : convert the RISC V command from Big Endian method to Little Endian method. The srec_cat generate MIF with Big Endian Method, while the memory that Quartus Prime receive is in Little Endian.

Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first, at the lowest storage address. Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first

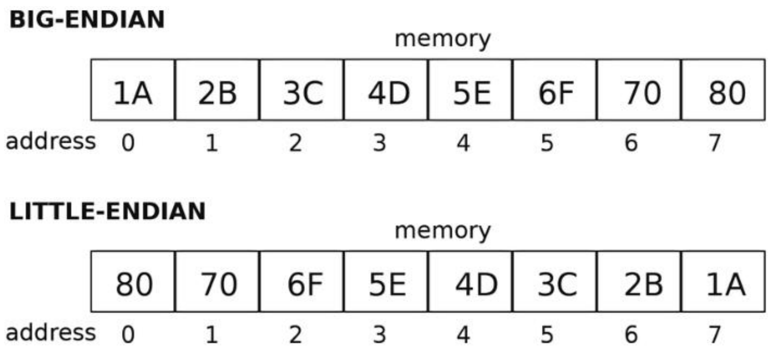


Figure 23 - Representation of 0x1A2B3C4D5E6F7080 in big-endian and little-endian.

Luckily, we have created an automation scrip for this operation.

After the MIF file is ready, you need to split it to Instruction and Data memory and put the files in the designated Quartus directory.

4.8 Load Instruction and Data Memory

Because UART abilities is not enabled yet, if we want to load a different program to the FPGA we need to replace the memory files from the design FPGA folder manually.

- First, we are using the "gpc_compile_link.sh" script to compile the program. The script had been updated with an ability to create a "MIF" file that will be used in the FPGA synthesis to a memory.
- Later we manually copy the content of the generated file "programName.mif" in this form:
    - Up to the address : 0x100000 not include, is the instruction memory. We copy the data to file : "\riscv-multi-core-lotr\FPGA\mem_hex\i_mem.mif" and replace the old data that was there
    - From the address : 0x100000 include, is the data memory. We copy the data to file : "\riscv-multi-core-lotr\FPGA\mem_hex\d_mem.mif" and replace the old data that was there
- We use full compilation from Quartus Prime
- We program the files to the FPGA kit.

4.9 Automation Scripts

All scripts from GPC_4T project were used in this project. 2 worth to mention new scripts:

4.9.1 BEtoLE.sh

Big Endian to Little Endian conversion Bash script made by Adi, that take an auto generated MIF file as an input and convert each 32bits word.
For example, from :

```
0030: A30FF4FC 032784FD 93070700 93972700 B387E700 93976700 2326F4FE 832744FD;
```

To:

```
0030: FCF40FA3 FD842703 00070793 00279793 00E787B3 00679793 FEF42623 FD442783;
```

4.9.2 Lotr_tb_sim.ps1

A PowerShell script, similar to the gpc_4t tb_sim.ps1 script, that wraps all the PowerShell commands that triggers a Simulation run with a ModelSim to a specific or batch of tests. The only difference from old script is that it runs on the LOTR test bent

5 Program Examples

5.1 MC_Multitask - 2 Cores, 8 thread

This massive test checks the limit of the Ring. In this test , 6 threads (3 on each core, the other two just waiting) are doing separate tasks, each task will performed on non-local core:

- thread 0 from core 1 sorting array located on thread 0 core 2.
- thread 1 from core 1 loading an array to be sorted to core 1 shared memory
- thread 2 from core 1 loading 2 matrices to be multiple to core 1 shared memory

- thread 0 from core 2 loading an array to be sorted to core 2 shared memory
- thread 1 from core 2 sorting array located on core 1 shared memory
- thread 2 from core 2 multiplies matrices located on core 1 shared memory

some code parts:

```
        //Core 2 threads
        case 0x8 :
        {
            int j;
            int arr[8] = {6,1,0,3,5,9,50,2};
            for(j = 0 ; j < 8 ; j++){
                SCRATCHPAD0_CORE[j] = arr[j];   //local core write
            }
            CR_ID10_PC_EN[0] = 1;   // signals core 1 to start
sorting
            while(1);
        }
        break;
```

```
        case 0x4 : // parameterize
        {
            CR_ID10_PC_EN[0] = 0;   //freeze itself and waiting for core 2 to
finish loading array
            volatile int* arr = SCRATCHPAD0_CORE_2; //remote core
            bubbleSort(arr, 8); //sorting array on remote core
            while(1);
        }
```

core 2 snapshot, array is sorted:

```
Offset 004008f8 : xxxxxxxx
Offset 004008fc : xxxxxxxx
Offset 00400900 : 00000000
Offset 00400904 : 00000001
Offset 00400908 : 00000002
Offset 0040090c : 00000003
Offset 00400910 : 00000005
Offset 00400914 : 00000006
Offset 00400918 : 00000009
Offset 0040091c : 00000032
Offset 00400920 : xxxxxxxx
Offset 00400924 : xxxxxxxx
Offset 00400928 : xxxxxxxx
```

5.2 LED BLINKY – FPGA KIT LED CONTROLLED BY SWITCHES STATE

In this single thread program, an infinite while loop reads the state of the switches on the FPGA kit. Depends on the switches state, the LEDS will show:

000 : ALL LED BLINK

001 : BINARY COUNTER

010 : ONE LED MOVING LEFT (power of 2 counter)

011 : DECREASE BINARY COUNTER (starts from 2^{10})

100 : ONE LED MOVING RIGHT (power of 2 decreasing counter, from 2^{9})

101 : 7 SEG COUNTER

Some code examples:

```
        else if (*SWITCH_FGPA == 2){
            skipLed = 1;
            delay();
            *LED_FGPA = skipLed;
            skipLed = skipLed * 2 ;
            if (skipLed > 512)
                skipLed = 1;
        }

        else if (*SWITCH_FGPA == 3){
            counter2 = 1023;
            delay();
            *LED_FGPA = counter2;
            counter2 --;
            if (counter2 == 0)
                counter2 = 1023;
        }
```

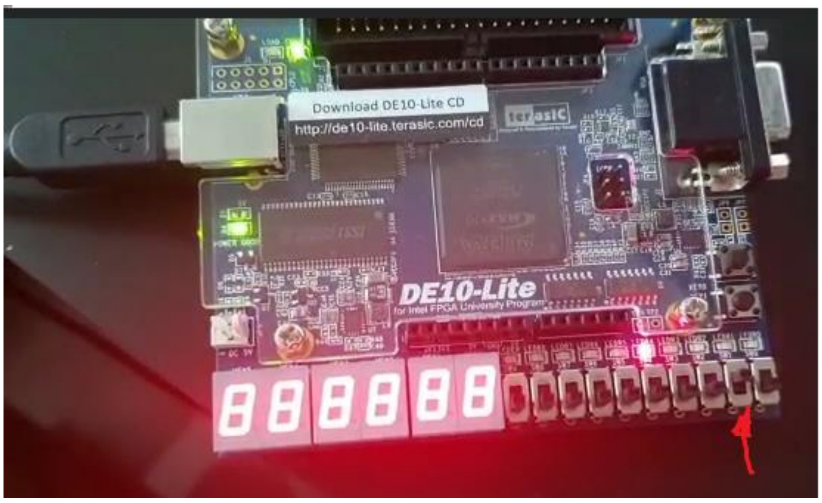

Download DE10-Lite CD
http://de10-lite.terasic.com/cd
DE10-Lite
for Intel FPGA University Program

Figure 24 - state 010: led moving left

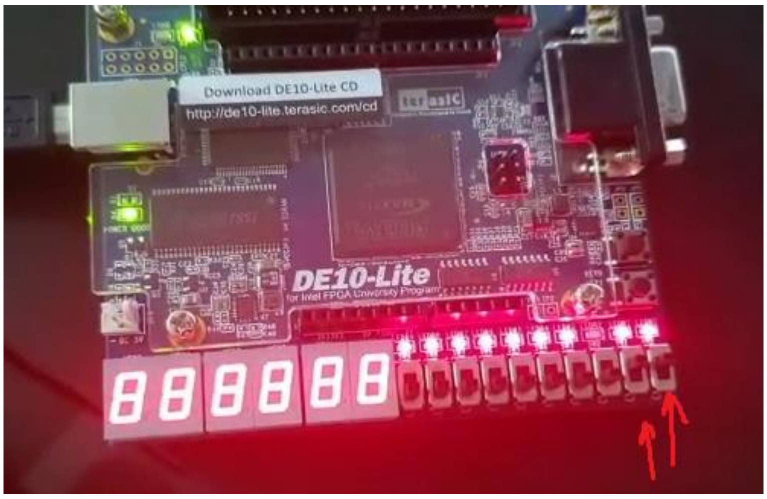

Download DE10-Lite CD
http://de10-lite.terasic.com/cd
DE10-Lite
for Intel FPGA University Program

Figure 25 - state 011: decrease counter

5.3 Snake – interactive game

An interactive snake game, designed by our advisor Amichai. Controlled by an Arduino interface . No need to expand.

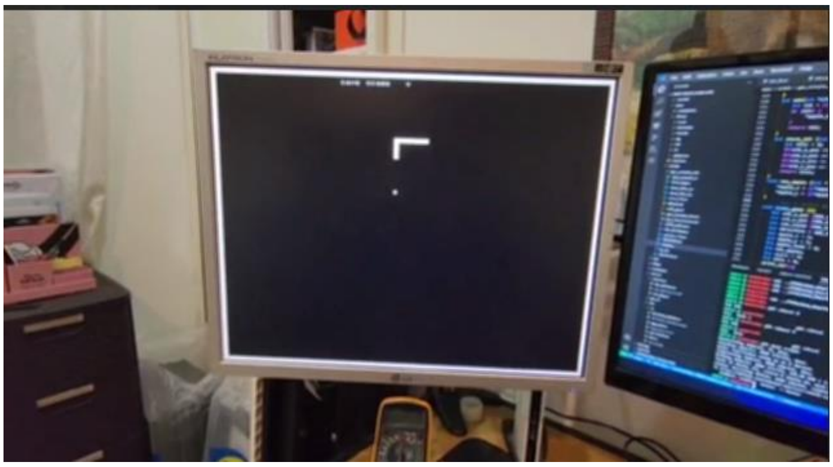


Figure 26 - snake

5.4 SORTING VGA A GRAPHICAL 8 THREAD ARRAY SORTING

In this test, we used the entire threads of the design. Each pair of thread perform a sorting algorithm on the same dataset. The difference between the 2 threads is the sorting algorithm. While one thread use Bubble Sort, the other use Insertion Sort. Using the graphic interface, the screen is divided into 8 parts. On each part, each number from an array is visualized as a vertical rectangle. Each swap the location of each rectangle is cleaned and the new swapped rectangle is drawn.

Using the Bar-Ilan's students' graphic library, we wrote some info on each of the 8 parts.

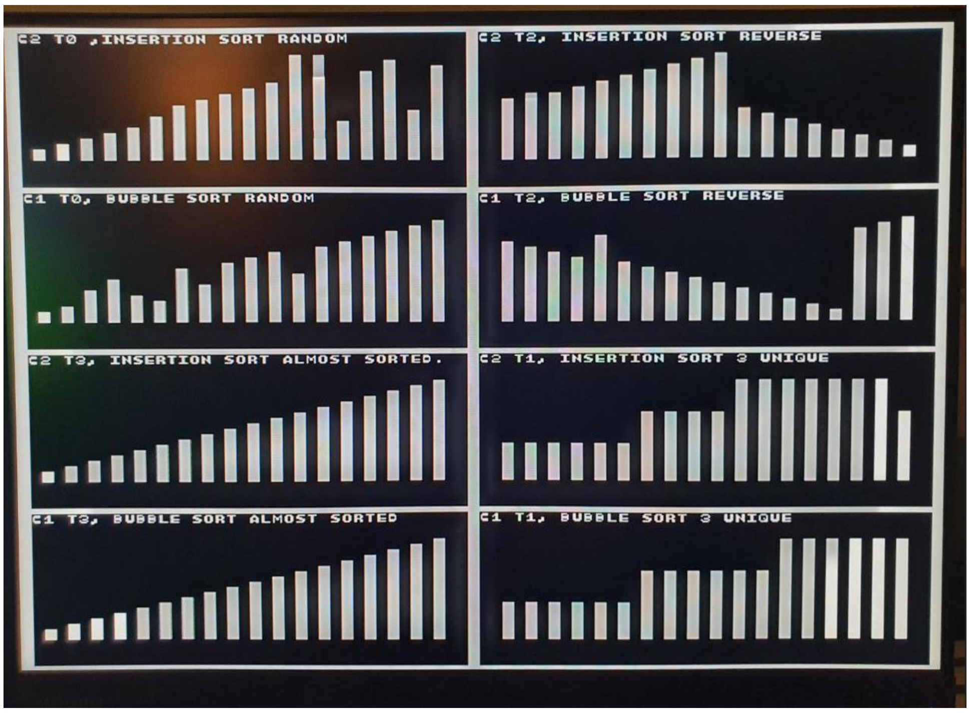

C2 T0 ,INSERTION SORT RANDOM
C2 T2, INSERTION SORT REVERSE
C1 T0, BUBBLE SORT RANDOM
C1 T2, BUBBLE SORT REVERSE
C2 T3, INSERTION SORT ALMOST SORTED.
C2 T1, INSERTION SORT 3 UNIQUE
C1 T3, BUBBLE SORT ALMOST SORTED
C1 T1, BUBBLE SORT 3 UNIQUE

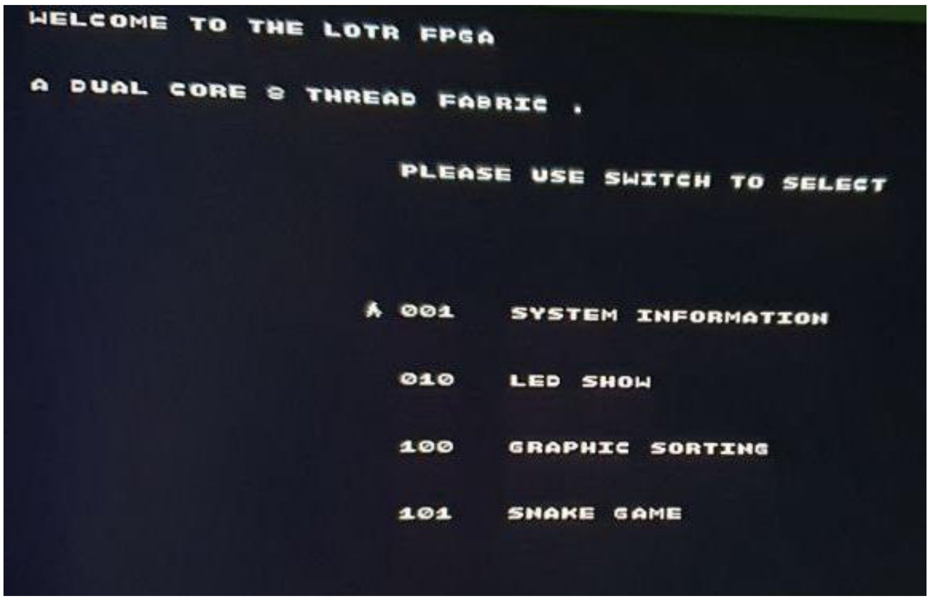
Figure 27 - Sorting VGA

5.5 FPGA MAIN – COMPLETE ALL-AROUND PROGRAM

To make the best impression, we have created a Main program that will include most of the programs detailed above, with an options menu to select the program.

When initiated, the Menu shows options to choose. The selection is made with the switches, and you can navigate between options. To choose the option switch 7 needs to be up.

5.5.1 Main menu

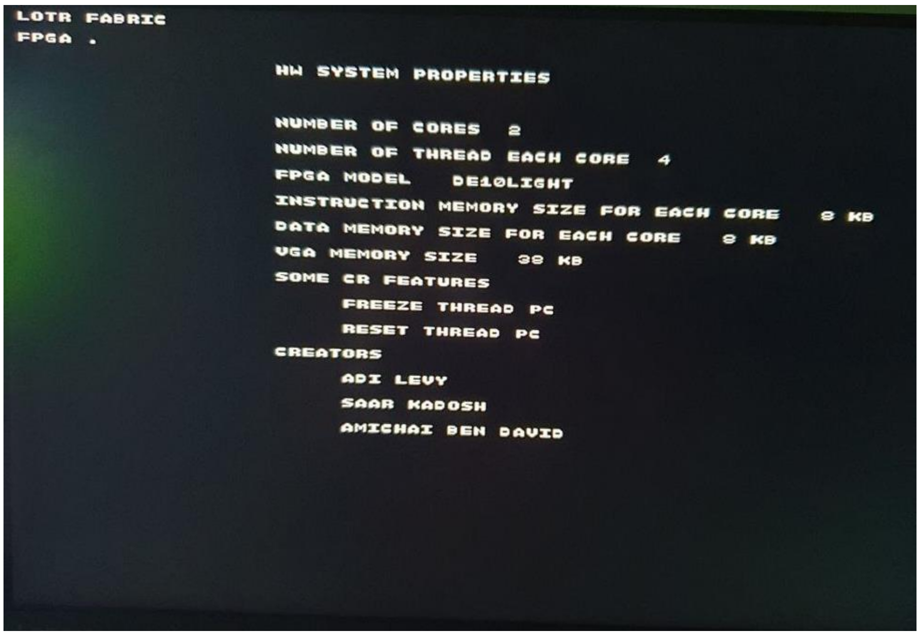


```
WELCOME TO THE LOTR FPGA

A DUAL CORE 8 THREAD FABRIC .

                    PLEASE USE SWITCH TO SELECT


                  ^ 001    SYSTEM INFORMATION

                    010    LED SHOW

                    100    GRAPHIC SORTING

                    101    SNAKE GAME
```

Figure 28 - FPGA main menu

5.5.2 System info

Some details on the design and the project

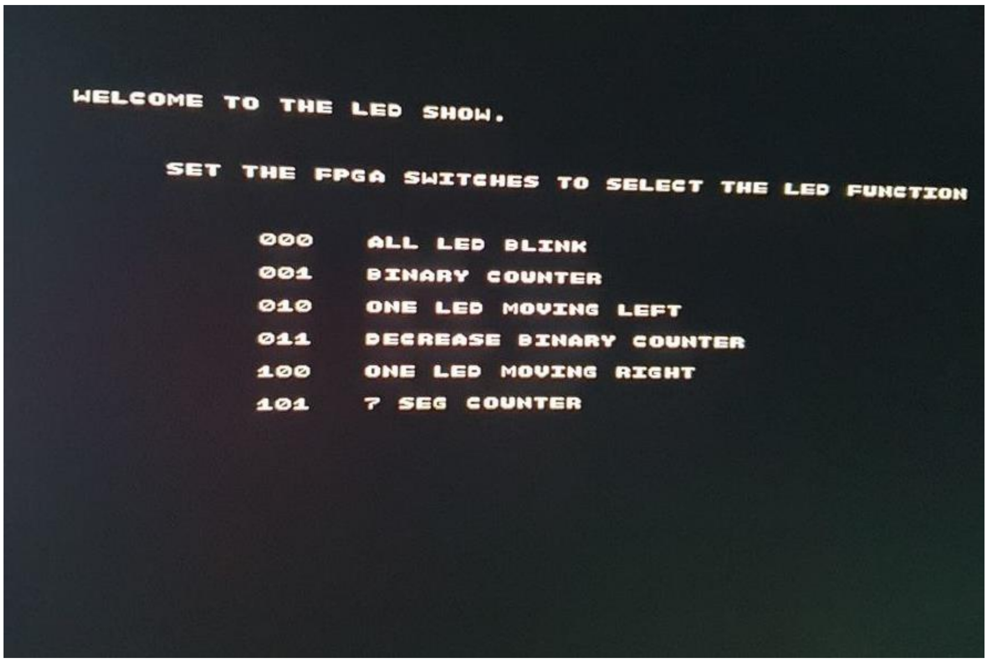


```
LOTR FABRIC
FPGA .
             HW SYSTEM PROPERTIES

             NUMBER OF CORES  2
             NUMBER OF THREAD EACH CORE  4
             FPGA MODEL   DE10LIGHT
             INSTRUCTION MEMORY SIZE FOR EACH CORE   8 KB
             DATA MEMORY SIZE FOR EACH CORE   8 KB
             VGA MEMORY SIZE   38 KB
             SOME CR FEATURES
                  FREEZE THREAD PC
                  RESET THREAD PC
             CREATORS
                  ADI LEVY
                  SAAR KADOSH
                  AMICHAI BEN DAVID
```

Figure 29 - HW information

5.5.3 Led Show

The Led blinky program with an explanation on the features



```
WELCOME TO THE LED SHOW.

        SET THE FPGA SWITCHES TO SELECT THE LED FUNCTION

              000    ALL LED BLINK
              001    BINARY COUNTER
              010    ONE LED MOVING LEFT
              011    DECREASE BINARY COUNTER
              100    ONE LED MOVING RIGHT
              101    7 SEG COUNTER
```

Figure 30 - LED show menu

5.5.4 Graphic sorting

The graphical Sorting algorithm

5.5.5 Snake game

The snake game program

6 DEVELOPMENT & METHODOLOGIES

6.1 INTRODUCTION

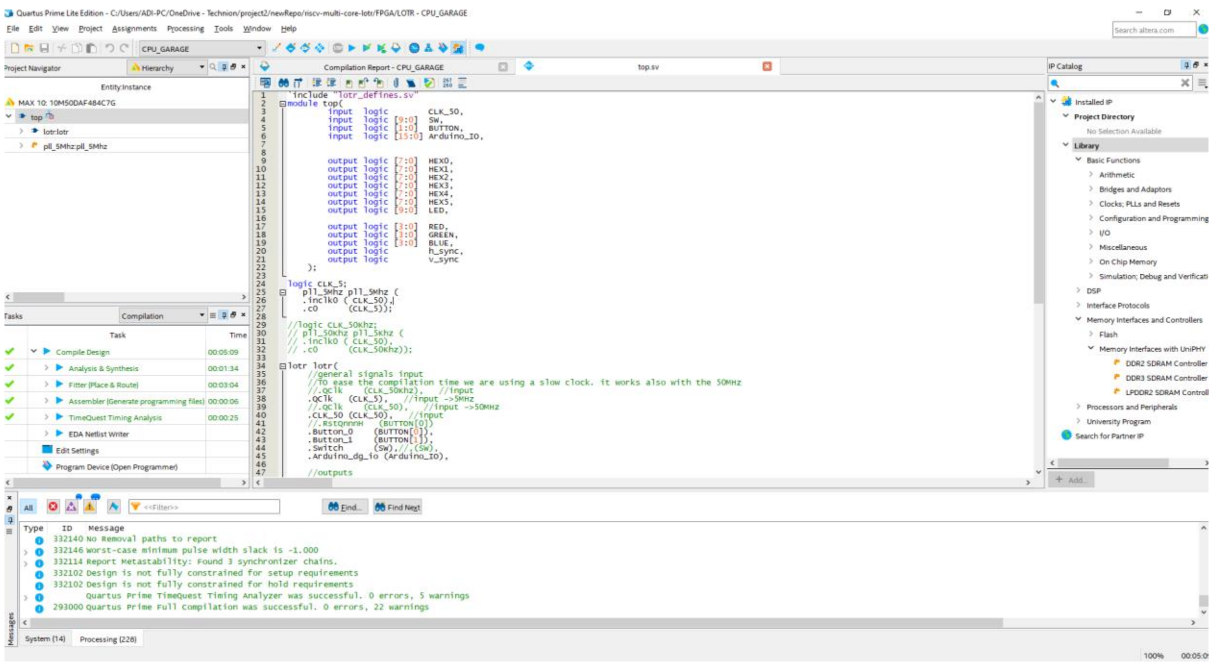
The same development tools from Project GPC_4T was used here, so it is highly recommended that prior read this chapter : GPC_4T Development and Methodologies.

6.2 GIT & REPOSITORY

The same github repository from the two previous projects was used here. Some new contributors were joined, possible to use the repository for future projects.

6.3 INTEL QUARTUS PRIME

The major tool that used for the FPGA integration. Intel Quartus Prime is programmable logic device design software produced by Intel. Quartus Prime enables analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.



```
`include "lotr_defines.sv"
module top(
            input  logic        CLK_50,
            input  logic [9:0]  SW,
            input  logic [1:0]  BUTTON,
            input  logic [15:0] Arduino_IO,


            output logic [7:0]  HEX0,
            output logic [7:0]  HEX1,
            output logic [7:0]  HEX2,
            output logic [7:0]  HEX3,
            output logic [7:0]  HEX4,
            output logic [7:0]  HEX5,
            output logic [9:0]  LED,

            output logic [3:0]  RED,
            output logic [3:0]  GREEN,
            output logic [3:0]  BLUE,
            output logic        h_sync,
            output logic        v_sync
    );

logic CLK_5;
   pll_5Mhz pll_5Mhz (
   .inclk0 ( CLK_50),
   .c0      (CLK_5));

//logic CLK_50Khz;
// pll_50Khz pll_5Khz (
// .inclk0 ( CLK_50),
// .c0      (CLK_50Khz));

lotr lotr(
    //general signals input
    //TO ease the compilation time we are using a slow clock. it works also with the 50MHz
    //.QClk    (CLK_50Khz),   //input
    .QClk   (CLK_5),   //input ->5MHz
    //.QClk    (CLK_50),   //input ->50MHz
    .CLK_50 (CLK_50),   //input
    //.RstQnnnH   (BUTTON[0])
    .Button_0    (BUTTON[0]),
    .Button_1    (BUTTON[1]),
    .Switch      (SW),//.(SW),
    .Arduino_dg_io (Arduino_IO),

    //outputs
```

	Task	Time
✓	Compile Design	00:05:09
✓	Analysis & Synthesis	00:01:34
✓	Fitter (Place & Route)	00:03:04
✓	Assembler (Generate programming files)	00:00:06
✓	TimeQuest Timing Analysis	00:00:25
	EDA Netlist Writer	
	Edit Settings	
	Program Device (Open Programmer)	

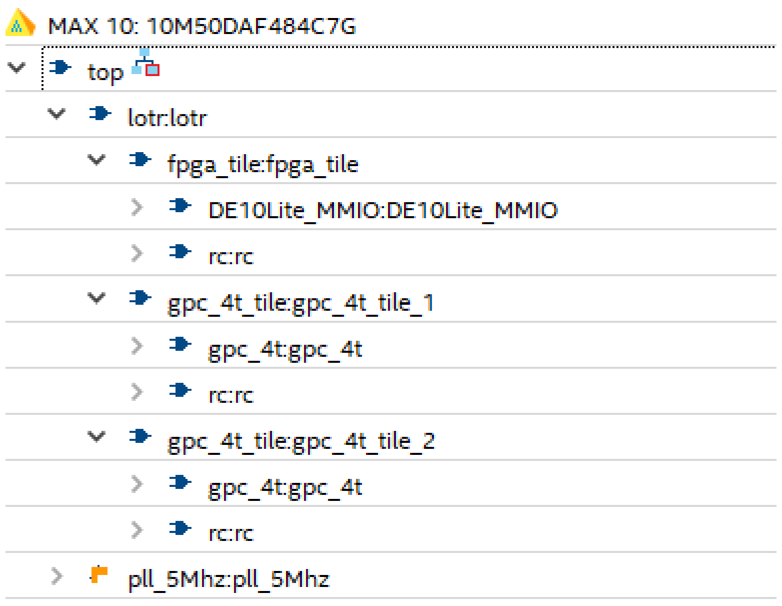
```
Type   ID     Message
       332140 No Removal paths to report
       332146 Worst-case minimum pulse width slack is -1.000
       332114 Report Metastability: Found 3 synchronizer chains.
       332102 Design is not fully constrained for setup requirements
       332102 Design is not fully constrained for hold requirements
              Quartus Prime TimeQuest Timing Analyzer was successful. 0 errors, 5 warnings
       293000 Quartus Prime Full Compilation was successful. 0 errors, 22 warnings
```

Figure 31 - Intel Quartus Prime

Using the Quartus Prime, we have loaded all our LOTR RTL Verilog files and created the Top.sv Verilog file which used as the top of the hierarchy tree of the design.

6.3.1 Top.sv

The Top model is the connection between the FPGA kit I\O's and the RTL.

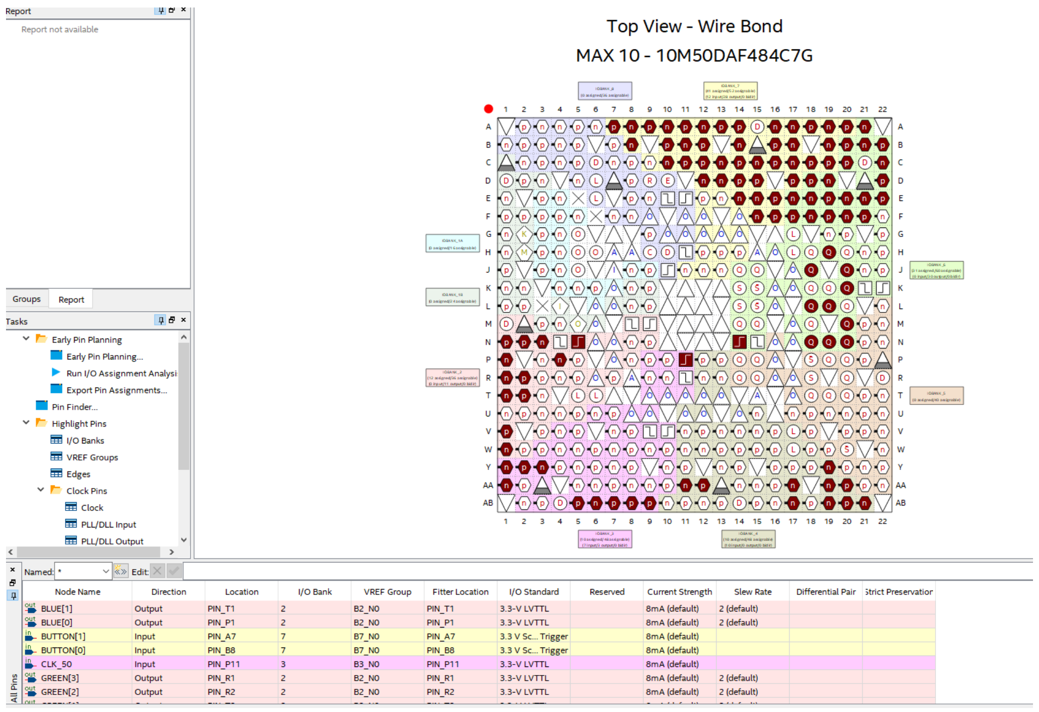


```
MAX 10: 10M50DAF484C7G
  top
    lotr:lotr
      fpga_tile:fpga_tile
        DE10Lite_MMIO:DE10Lite_MMIO
        rc:rc
      gpc_4t_tile:gpc_4t_tile_1
        gpc_4t:gpc_4t
        rc:rc
      gpc_4t_tile:gpc_4t_tile_2
        gpc_4t:gpc_4t
        rc:rc
    pll_5Mhz:pll_5Mhz
```

Figure 32 - Hirachy tree

6.3.2 Pin Planner

Using the Pin Planner feature, we connected and enabled the I\O pins.

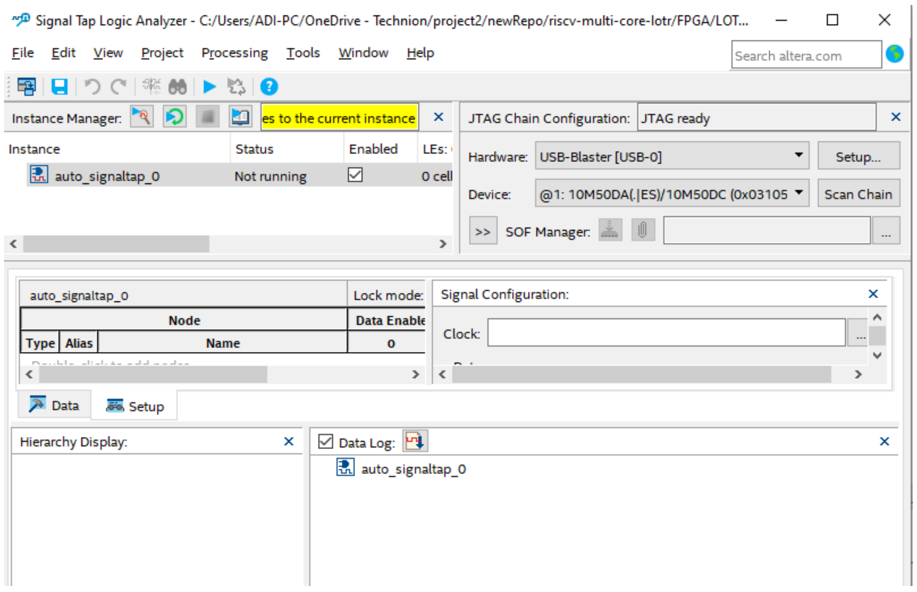


Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	Strict Preservation
BLUE[1]	Output	PIN_T1	2	B2_N0	PIN_T1	3.3-V LVTTL		8mA (default)	2 (default)		
BLUE[0]	Output	PIN_P1	2	B2_N0	PIN_P1	3.3-V LVTTL		8mA (default)	2 (default)		
BUTTON[1]	Input	PIN_A7	7	B7_N0	PIN_A7	3.3 V Sc... Trigger		8mA (default)			
BUTTON[0]	Input	PIN_B8	7	B7_N0	PIN_B8	3.3 V Sc... Trigger		8mA (default)			
CLK_50	Input	PIN_P11	3	B3_N0	PIN_P11	3.3-V LVTTL		8mA (default)			
GREEN[3]	Output	PIN_R1	2	B2_N0	PIN_R1	3.3-V LVTTL		8mA (default)	2 (default)		
GREEN[2]	Output	PIN_R2	2	B2_N0	PIN_R2	3.3-V LVTTL		8mA (default)	2 (default)		

Figure 33 - Pin planner

6.3.3 Signal Tap

We used the signal tap tool to detect some bugs, especially on the enabling phase of the FPGA, to check why LEDS are not toggling .



Signal Tap Logic Analyzer - C:/Users/ADI-PC/OneDrive - Technion/project2/newRepo/riscv-multi-core-lotr/FPGA/LOT...

Instance Manager: es to the current instance

JTAG Chain Configuration: JTAG ready

Instance	Status	Enabled	LEs:
auto_signaltap_0	Not running	☑	0 cell

Hardware: USB-Blaster [USB-0]

Device: @1: 10M50DA(.|ES)/10M50DC (0x03105

SOF Manager:

auto_signaltap_0 Lock mode:

Node			Data Enable
Type	Alias	Name	0

Signal Configuration:

Clock:

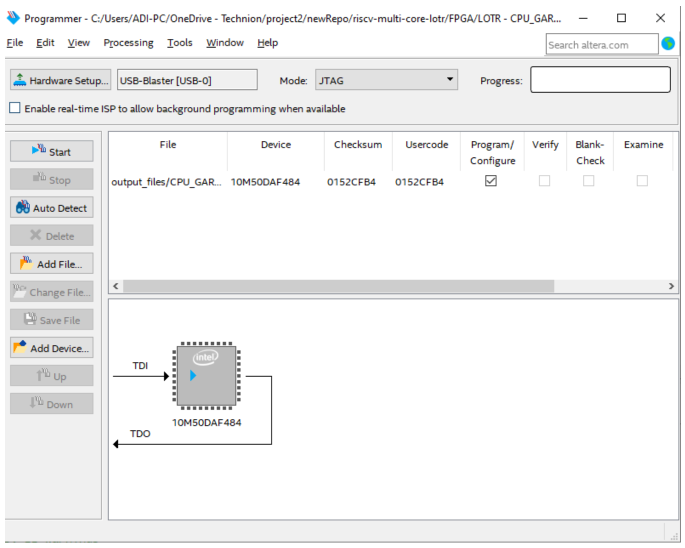
Hierarchy Display:

Data Log: auto_signaltap_0

Figure 34 - signal tap

6.3.4 Programmer

The programmer feature used to program the generated FPGA synthesis files, generated after full compilation, to the FPGA kit using USB Blaster.



Programmer - C:/Users/ADI-PC/OneDrive - Technion/project2/newRepo/riscv-multi-core-lotr/FPGA/LOTR - CPU_GAR...

Hardware Setup... USB-Blaster [USB-0] Mode: JTAG Progress:

☐ Enable real-time ISP to allow background programming when available

File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank- Check	Examine
output_files/CPU_GAR...	10M50DAF484	0152CFB4	0152CFB4	☑	☐	☐	☐

Figure 35 - Quartues programmer

7 FUTURE PLANS

Actually, our Technion Faculty Project is now finished. But The LOTR project has an enormous potential for the Future for any RTL developers' projects.

Some Future plans that Amichai will lead with students in the future:

- ❖ Stall mechanism for a large amount of Ring transactions at once.
- ❖ UART interface
- ❖ Keyboard integration with the FPGA
- ❖ More games – a Console like project
- ❖ Larger FPGA that can support 8 to 16 GPC_4T cores and larger Memory size.

If the Faculty VLSI Lab engineers will be interested and if we could find time:

- ❖ Compete in the Faculty Projects contest and win the 1st place.

8 REFERENCES

- Washington University in St. Louis - Overview of Performance Measurement and Analytical Modeling Techniques for Multi-core Processors
  https://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore/index.html

- intel.com on MIF
  https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_mif.htm

- researchgate.net on Big and Little endian method
  https://www.researchgate.net/figure/Representation-of-0x1A2B3C4D5E6F7080-in-big-endian-and-little-endian_fig1_335670464

- RVC ASAP – Amichai & bar Illans' students project repository
  https://github.com/amichai-bd/rvc_asap

- Wikipedia, The open online encyclopedia.
  https://en.wikipedia.org/wiki

- Website of Digital Systems and Computer structure Course at The Faculty of EE & Computers, Technion.
  https://moodle.technion.ac.il/enrol/index.php?id=6123

- AnandTech, Tech blog, on Ring Architecture.
  https://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed/4

- Official GNU, about linker scripts
  https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html

- RISC-V Toolchain Install guide and user guide repository by John Winans
  https://github.com/johnwinans/riscv-toolchain-install-guide

- GNU GCC Compiler guide
  https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html

- Github – Getting started with Git & Github
  https://docs.github.com/en/get-started/quickstart/hello-world

- Git official, git documentations
  https://git-scm.com/docs/gittutorial

- System Verilog tutorial
  https://verificationguide.com/systemverilog/systemverilog-tutorial/