GPC_4T:
Development & Methodologies

Development Environment and Methodologies used during GPC_4T solution

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

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

Contents

Revision History

Rev. No.	Who	Description	Rev. Date
0.1	*Adi Levy*	*Initial GPC_4T Dev & Methodologies*	*06 November 2021*

Figures

1 GIT & GITHUB

1.1 GIT

Git is a well-known software for tracking changes in any set of files. we used git to clone our project files to our personal computers , work on them simultaneously and then upload them to our common cloud stored in github.com. Git also used us for solving "merge conflicts" and track changes.

1.2 GITHUB

GitHub is a free provider of Internet hosting for software development and version control using Git. We used GitHub to store our "repository" which include all the project files, in addition to the parallel Ring Controller project files and the LOTR files developed by our advisor Amichai.

Figure 1 - github logo

2 LOTR REPOSITORY STRUCTURE

The repository was maintained mostly by Amichai and he is the biggest contributor to the hub.
In addition to storing our files and tracking old version using git, the GitHub also include many aspects:

- Code & Code Review : each contributor can upload his code and receive code review before the new uploaded code is merged with the complete repository code .

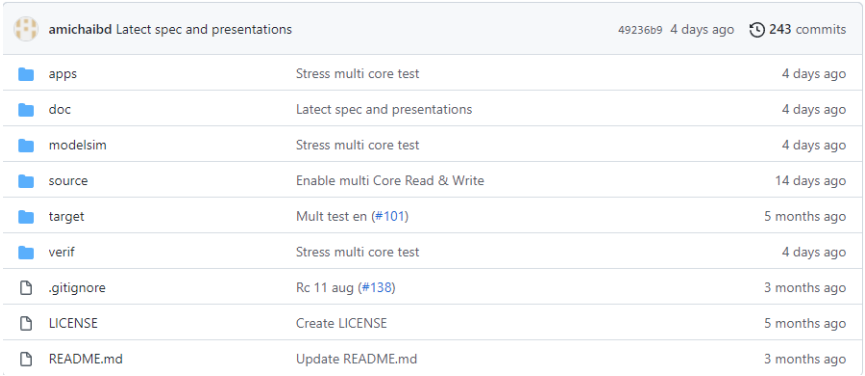

Figure 2 - LOTR complete repository code, including GPC_4T code as seen on GitHub

- Task Management : used by "issues" tab on git lab. Each meeting with the advisor we assign tasks between us and detailed about the task in its page.

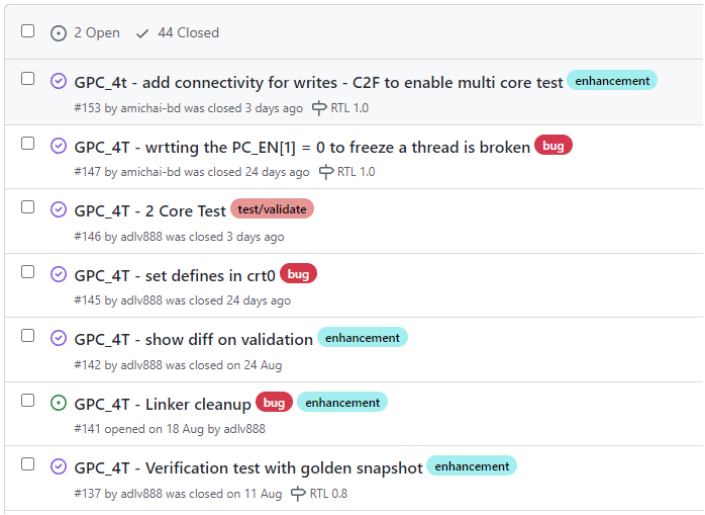

Figure 3 - Project tasks as showen on GitHub

- Discussions and Questions : every time we encountered a problem in our work , we wrote about it in the Discussions tab. one of the many contributors to one of the projects can answer if knows.

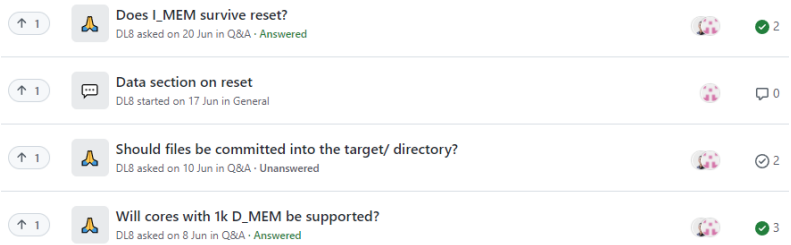

Figure 4 - Questions & Discussions on GitHub repository

- Wikia : An edit free wiki pages were maintained by the GitHub contributors. The idea was to create a data base of knowledge for each piece of the LOTR project, including the GPC_4T. Many of information and figures mentioned in this book was written by us in this wiki during the work on the project.

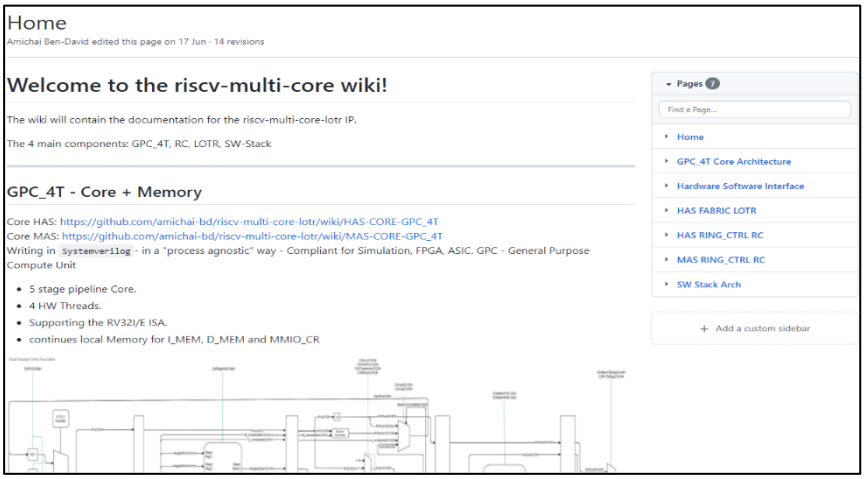

Figure 5 - Project's wiki on GitHub

3 RISC-V GNU COMPILER TOOLCHAIN

The Toolchain is a bundle of software tools cloned from "riscv-gnu-toolchain" Git repositories .It was use to create & verify assembly instructions against the open-source ISA specification for an RV32IM core.

A detailed installation guide can be found in this link on John Winans repository.

3.1 WSL

The Toolchain installed on WSL environment , which is a windows subsystem that behaves much like a Linux kernel, on top of it can be installed real GNU/Linux distribution. In our case Ubuntu OS installed.
There are alternatives by using RISC V compilers on Windows OS environment.

3.2 GCC

In simplicity, the Toolchain contains several of special compilers based on the famous gcc compiler family and we use it to make from a ".c" file written in C language to a RISC-V Assembly Language file (.S) , and from that file, to a binary .sv file that will simulate the instruction memory GPC_4T can read. The toolchain also generates these files as text files that the user can read.

Figure 6 - Toolchain compiling example : clockwise - C program converted to assembly converted to sv binary code

As seen as figure 3, the C program converted to RiscV assembly commands. the assembly code is longer than the C code. each assembly command is coded in a 32bit vector (or 8 hexadecimal digit vector in the text file). all vectors combined to form the simulation instruction memory (bottom

image). The GPC_4T decodes each vector as will described in upcoming sections.
The SV text file will used as an instruction memory to the GPC_4T in the simulations via "Back Door" technique will discussed in Validation section.

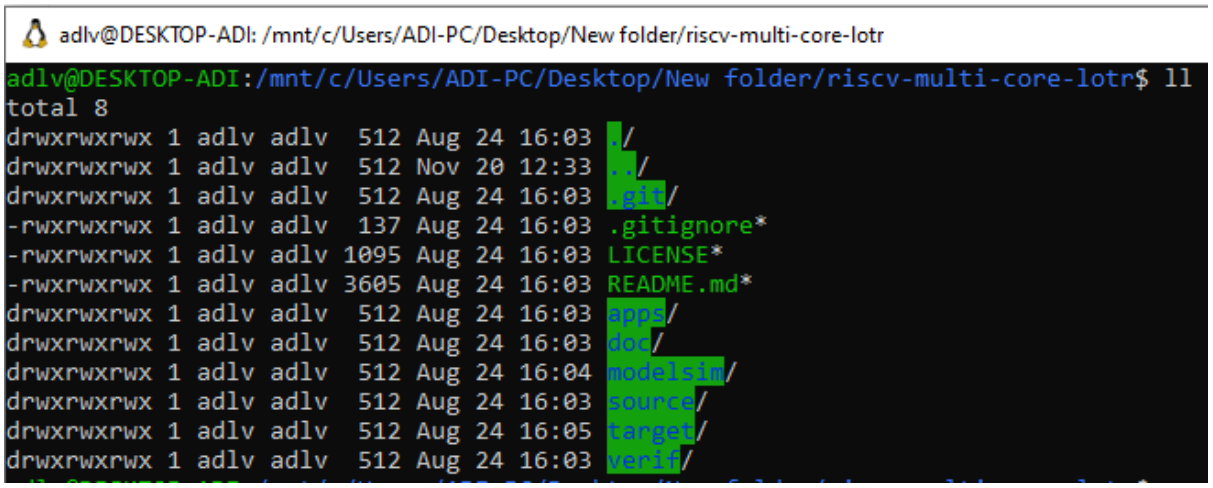

Figure 7 - WSL Ubunto Shell showing project files

after installing Toolchain on the WSL, we used the shell in order to run a series of commands to run the special Toolchain RiscV compilers & Linkers to get all necessary files from C file.

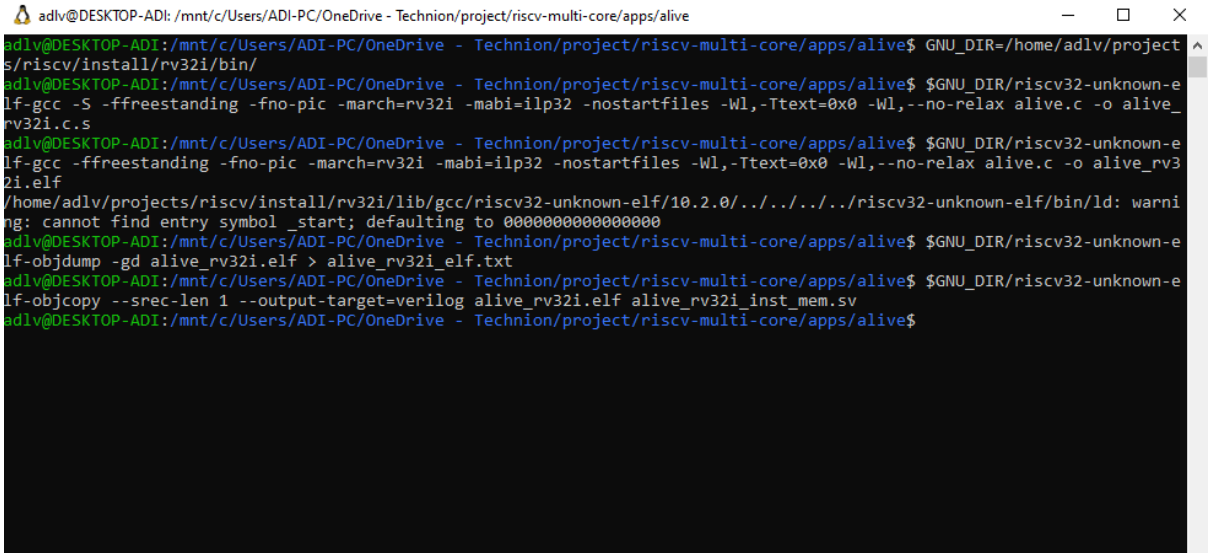

Figure 8 - ToolChain commands use example on alive.c

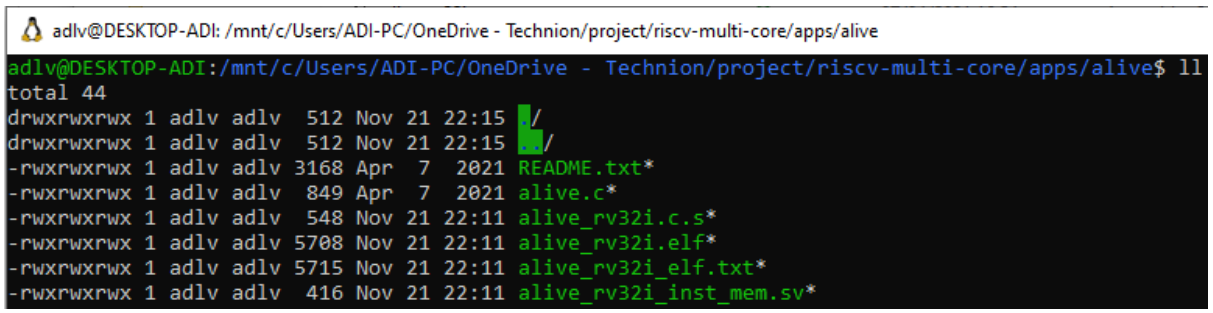

Figure 9 - Created files from alive.c after Figure FIXME commands

4 SYSTEM VERILOG

System Verilog is a hardware description language(HDL) used to model, design, simulate, test and implement electronic systems. SV can used to describe hardware behavior so that it can be converted to digital blocks made up of combinational gates and sequential elements.
the Gpc_4T RTL(Register-Transfer Level) was written in System Verilog.

```
always_comb begin : choose_alu_input
    unique casez ({ CtrlLuiQ102H, CtrlAuiPcQ102H, CtrlITypeImmQ102H, CtrlStoreQ102H })
        4'b1000 : begin // OP_LUI
            AluIn1Q102H = 32'b0;
            AluIn2Q102H = U_ImmediateQ102H;
        end
        4'b0100 : begin // OP_AUIPC
            AluIn1Q102H = PcQ102H;
            AluIn2Q102H = U_ImmediateQ102H;
        end
        4'b0010 : begin // OP_JALR || OP_OPIMM || OP_LOAD
            AluIn1Q102H = RegRdData1Q102H;
            AluIn2Q102H = I_ImmediateQ102H;
```

Figure 10 - System Verilog code example from GPC_4T source code, showed on notepad++ text editor

4.1 PROJECT CODING STYLE

Thru the entire System Verilog programming of the RTL, we used a coding style that was coherence to the entire LOTR projects (GPC_4T, RC and LOTR itself).

- **Macros**: On system Verilog, when you want do design a flip-flop it is known to use this convention:

```
always_ff @(posedge clk)
   if (rst)     q <='0;
   else if(en)  q <= i;
```

In a file named lotr_defines.sv we set our defines for flip-flops. For example to use the flip-flop on the convention above, we use the define :

```
`define  LOTR_EN_RST_MSFF(q,i,clk,en,rst)
```

thus every time we need to use this we write with this define

```
`define  LOTR_EN_MSFF(q,i,clk,en)        \
         always_ff @(posedge clk)        \
            if(en) q<=i;

`define  LOTR_RST_MSFF(q,i,clk,rst)      \
         always_ff @(posedge clk) begin  \
            if (rst) q <='0;             \
            else     q <= i;             \
         end

`define  LOTR_RST_VAL_MSFF(q,i,clk,rst,val)
         always_ff @(posedge clk) begin
            if (rst) q <= val;
            else     q <= i;
         end
```

Figure 11 - lotr defines examples

- **CamelCase** – we used CamelCase coding style convention to name the variables ,logic signals, wires, buffers etc. thru all of the RTL files. On this convention we name the variables like this: SomeNameOfTheLogic with no spaces with each word starts Sometimes a combination of CamelCase and snake_case used when we felt some logic names belong to the same family, for example :
  C2F_RspValidQ502H
  C2F_RspOpcodeQ502H
  C2F_RspThreadIDQ502H
  C2F_RspDataQ502H

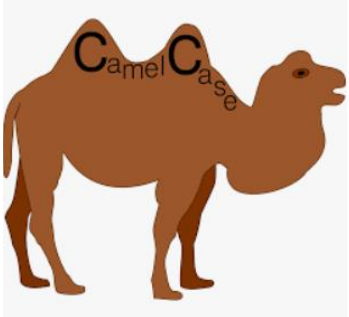

Figure 12 – Double-humped Bactrian camel

- **Lotr_pkg** – all the LOTR project parameters were defined in a file named lotr_pkg.sv. Many parameters such as Op Codes, CR addresses, Memory sizes, offsets, MSB and LSB locations, region(core id) bits location, encoded region(I mem, d mem, cr mem) bits, structs(like core_cr struct) and more were defined there and can be used on all RTL files.

```
parameter OP_LUI      = 7'b0110111;
parameter OP_AUIPC    = 7'b0010111;
parameter OP_JAL      = 7'b1101111;
parameter OP_JALR     = 7'b1100111;
parameter OP_BRANCH   = 7'b1100011;
parameter OP_LOAD     = 7'b0000011;
parameter OP_STORE    = 7'b0100011;
parameter OP_OPIMM    = 7'b0010011;
parameter OP_OP       = 7'b0110011;
parameter OP_FENCE    = 7'b0001111;
parameter OP_SYSTEM   = 7'b1110011;
```

Figure 13 - RV32I commands op code parameters from lotr.pkg

- **Cycle Suffix** – To make development easier, we labeled each signal name with a suffix implying the pipeline stage this signal is used, generated, or sampled. after the signal sampled in flops, its suffix increased by 1. Any signal can be used anywhere on the cycle but that is our convention. For example, AluOut**Q102H** is the ALU output signal on stage Q102H and when it used in stage Q103H it will be sampled on flop and will be named AluOut**Q103H**

```
`LOTR_MSFF   ( PcPlus4Q103H      , PcPlus4Q102H      , QClk)
`LOTR_MSFF   ( AluOutQ103H       , AluOutQ102H       , QClk)
`LOTR_MSFF   ( Funct3Q103H       , Funct3Q102H       , QClk)
`LOTR_MSFF   ( RegWrPtrQ103H     , RegWrPtrQ102H     , QClk)
```

Figure 14 - signals form Q102H stage, sampled before used in Q103H stage

5 MODELSIM

ModelSim is a multi-language environment for simulation of hardware description languages such as VHDL, Verilog and SystemC. Simulation is performed using the graphical user interface (GUI), or automatically using scripts. [Wikipedia]

We first exposed to ModelSim in Digital Systems Course when we used it to program and simulate logic gates.
On the GPC_4T project we used ModelSim in the Validation process to simulate the GPC_4T core and the memories using the System Verilog RTL files. One the first steps of the project, we used the GUI form of ModelSim , especially in the Waveform GUI in order to debug and track signals ,but as the project progressed, we used automation scripting to simulate.

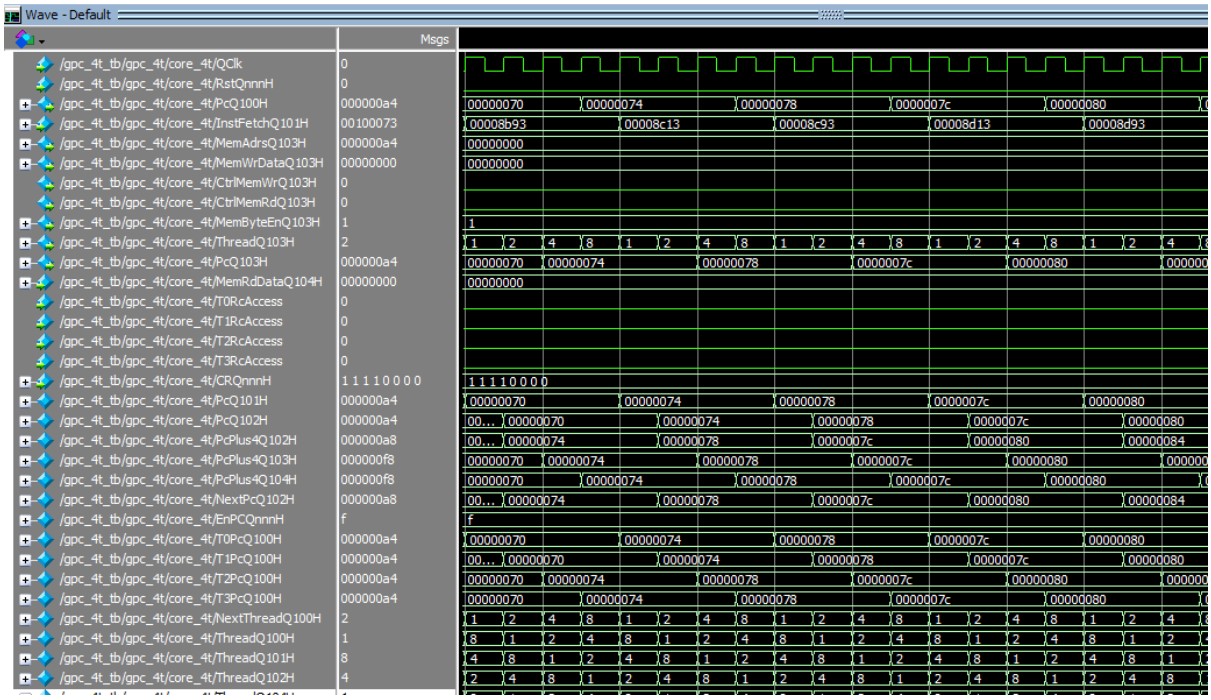

Figure 15- ModelSim Waveform simulation of GPC_4T core

more details on the simulations on Validation chapter.

6 POWERSHELL & BASH

Two famous automation work environments – Bash for Unix/Linux and PowerShell for Windows , used in the project to enhance the validation stage by scripting. Bash scripts used to enhance the Toolchain workflow and PowerShell used for ModelSim simulation workflow. More detailed information on the automation in the Validation Plan chapter.

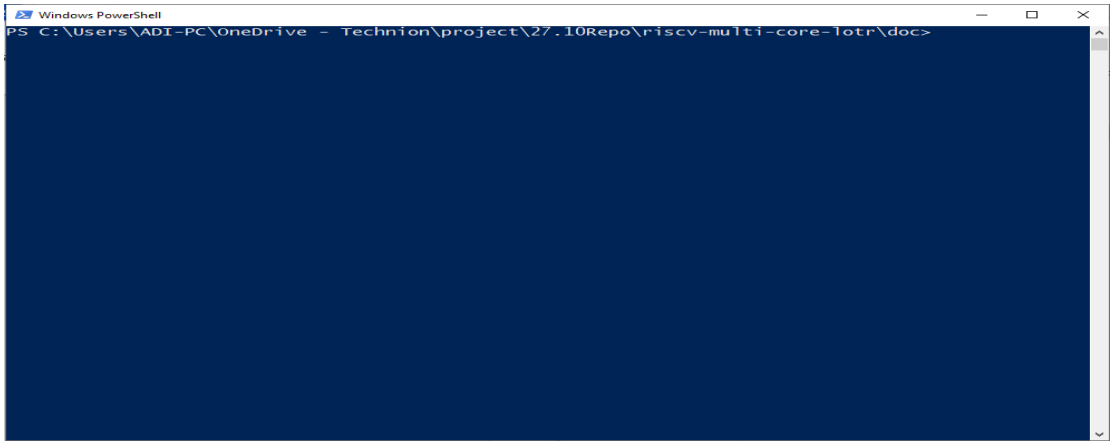

Figure16 - Windows PowerShell