

GPC_4T: Validation Plan

GPC_4T VALIDATION PLAN USED IN PROJECT

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

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

Contents

1	Introduction	3
2	GCC RISV-V Compiler & Programs	3
2.1	GCC to generate tests	3
2.2	GPC_4T Feature coverage	4
2.2.1	ISA Coverage – RV32I	4
2.2.2	Alive Test	5
2.2.3	CRs Features & Dedicated tests	6
2.2.4	Multi Thread application	7
2.2.5	Automation & Scripts	8
3	System Verilog Test Bench	8
3.1	Test Bench (gpc_4t_tb.sv)	8
3.2	GPC_4T Log Generator(spc_4t_log_gen.sv)	9
3.3	End Of Test Snapshot Checker	9
3.4	Assertions	10
3.5	RTL complication (list.f)	11
3.6	Load Backdoor I_MEM	11
3.7	Simulation	12
3.8	Automation & Scripts (tb_sim.ps1)	13

Revision History

Rev. No.	Who	Description	Rev. Date
0.1	Adi Levy	Initial GPC_4T Validation plans	06 November 2021

Figures

Figure 1 - Memory definition inside the linker script	4
Figure 2 – Example from Alive Tes	5
Figure 3 - BGE fuction from Alive Test	6
Figure 4 - Branch comperator code typo	6
Figure 5 - gpc_compile_link.sh	8
Figure 6 - A program sv file	8
Figure 7 - tracker files generated after simulation	9
Figure 8 – Memory snapshot Comparison	10
Figure 9 - Assertion	10
Figure 10 - Compilation errors	11
Figure 11 - The Backdoor code	11
Figure 12 - Waveform of a simulation	12
Figure 13 - ALU tracker log file	12
Figure 14- Alive Test debug	12
Figure 15 - tb_sim script running on the PowerShell terminal	13

1 INTRODUCTION

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. For example a very familiar validation is the post-silicon validation which is the performance, functionality, power, voltage and more verifying tests after the CPU is manufactured physically. In GPC_4T project Validation was critical to ensure the correctness of GPC_4T core & Memory functionality, before continuing to future project like FPGA synthesis. Because we don't have a physical GPC_4T unit, we checked GPC_4T functionality with ModelSim simulations on a Testbench.

2 GCC RISV-V COMPILER & PROGRAMS

2.1 GCC TO GENERATE TESTS

After writing a C program that we want to run on the GPC_4T test bench, and after considering the expected results for the program, we use the [RISC-V Toolchain](#) commands on WSL Ubuntu Shell terminal. Before explaining the Toolchain commands, two important files need to be explained:

- **GPC Initialization assembly code file** – a file named `crt0_gpc.S` which contains a code written in RISC-V assembly that supposed to run on the GPC_4T test bench before any C program.

It has 5 main stages:

- **Pipeline cleanup** – 5 "nop" commands entered to GPC_4T in order to clean the pipe from previous or undefined data.
- **Initialize registers** – set all 32 core registers to 0 using "mv" command and register x0 which contains always zero.
- **Stack Initialization** - As explained, each thread has unique stack address. This address is defined on special CR register. At first the CR address is loaded to x5 register, and later the data from CR address loaded to x2 register which is the stack pointer register on the RISC-V.

The stack Initialization commands:

```
li x5,THREAD_STACK_ADDRESS
```

```
lw x2,0(x5)
```

after these are done, each thread running the program will use his own stack pointer address to the data memory.

- **Jump to main** – jump command directly to the C program main function.
- **Terminate Run** – after returning from C program's main, we use EBREAK command to terminate the test bench.

- **Linker script file** – An LD file, which is a script written in the GNU "linker command language".

The Toolchain use the linker to link the crt0 code and the program assembly code to a single assembly program. It also defines the sizes and origin addresses of the instruction & data ram and the stack.

Full disclosure, we had trouble find information about linker scripts, so we used as a

reference on our advisor old linker script which he used on an old project. Due to this, it is possible our linker script contains useless or irrelevant code.

```
MEMORY
{
    instram      : ORIGIN = 0x00000000, LENGTH = 0x1000
    dataram      : ORIGIN = 0x00400800, LENGTH = 0x1000
    stack        : ORIGIN = 0x00400000, LENGTH = 0x0200
}
```

Figure 1 - Memory definition inside the linker script

After opening the Shell and navigate to Toolchain compilers folder, 4 serial Toolchain compiling commands, as seen at [figure FIX](#) are initiated:

1. `gcc -S -ffreestanding -march=rv32i program.c -o program_rv32i.c.s` –
Compile the C program and creates the assembly file from c file.
2. `gcc -O3 -march=rv32i -T./gpc_link.common.ld -nostartfiles -D__riscv__ $1_rv32i.c.s
./crt0_gpc.S -o $1_rv32i.elf` –
link new asm file with gpc initializer using the linker script and creates an "elf" file.
3. `objdump -gd $1_rv32i.elf > $1_rv32i_elf_txt.txt` –
creates a readable elf file.
4. `objcopy --srec-len 1 --output-target=verilog $1_rv32i.elf $1_inst_mem_rv32i.sv` –
creates the sv file, which is the 8 hexadecimal digits instructions dump which is used as the Instruction Memory for the GPC_4T test bench. If Heap memory is used in the c program, then a Heap section will also attach to the file.

after this stage, the C program is coded to a hex code RISC V 32-bit file instructions, which is ready to be used as the instruction memory via Backdoor in the test bench.

2.2 GPC_4T FEATURE COVERAGE

2.2.1 ISA Coverage – RV32I

As mentioned, GPC_4T supports RISC-V 32I alone. The Toolchain GCC compiler creates from a C program – an assembly program, or more particular – RISC-V32I assembly program. This assembly contains only RV32I commands (Not including MUL command for example which is RV32M).

Here is a C program for adding 2 integers:

```
#define SHARED_SPACE ((volatile int *) (0x00400f00))
int main() {
    int a,b,c;
    a = 5;
    b=4;
    c=a+b;
    SHARED_SPACE[0] = c;
}
```

After running the GCC compiler we get this, which is the program equivalent on RV32I assembly:

```
000000a8 <main>:
a8: fe010113      addi    sp, sp, -32
ac: 00812e23      sw     s0, 28(sp)
b0: 02010413      addi    s0, sp, 32
b4: 00500793      li     a5, 5
b8: fef42623      sw     a5, -20(s0)
bc: 00400793      li     a5, 4
c0: fef42423      sw     a5, -24(s0)
c4: fec42703      lw     a4, -20(s0)
c8: fe842783      lw     a5, -24(s0)
cc: 00f707b3      add    a5, a4, a5
d0: fef42223      sw     a5, -28(s0)
d4: 004017b7      lui    a5, 0x401
d8: f0078793      addi    a5, a5, -256 # 400f00 <_bss_end+0x700>
dc: fe442703      lw     a4, -28(s0)
e0: 00e7a023      sw     a4, 0(a5)
e4: 00000013      nop
e8: 00078513      mv     a0, a5
ec: 01c12403      lw     s0, 28(sp)
f0: 02010113      addi    sp, sp, 32
f4: 00008067      ret
```

We can notice that in RV32I assembly the program is a lot longer.

Each line is a RV32I assembly command. On the right is can notice the command name and the commands operands (it can be a hard-coded defines or registers names or offsets and more).

The whole command is encoded to 32bit vector (can be notice on the middle, in hexadecimal) which contains all the data needed for GPC_4T and other RV32I supported devices.

2.2.2 Alive Test

2.2.2.1 General Description

Alive Test is a special C program design to fulfill an important purpose : verify the correctness of most of the RISC-V 32I commands including arithmetic commands, branch operation and memory accesses ,running on the GPC_4T units – mainly the core units such as ALU, Controllers , registers and program counters , on a **single thread**.

This test is the first complete program (C, assembly and 32bit binary code) to run on the GPC_4T test bench.

The main code contains a series of functions, each checks a different RISC V commands (ADD, SLT...) by performing the command on C (which will be translated to the "real" RISC-V assembly with Toolchain) and writing '1' to the memory if the expected result (calculated manually) received.

```
void ORI(int a){
    a=a|26;
    if(a==30)
        D_MEM_SHARED[3] = 1;
}
```

Figure 2 – Example from Alive Test : ORI command called from main with a=12. 12 ORI 26 = 30

This test allowed us to locate and fix core bugs, and after it passed successfully with '1' written to memory on each test and signaling that all GPC_4T units functions correctly , we could continue testing much more complicated programs one the GPC_4T.

2.2.2.2 Bug capture example

An example of a bug we captured thanks to the Alive Test :

```

580:    00f75a63                bge a4,a5,594 <BGE+0x30>
584:    004017b7                lui  a5,0x401
588:    f5478793                addi a5,a5,-172 # 400f54
58c:    00100713                li   a4,1
590:    00e7a023                sw   a4,0(a5)
594:    00000013                nop

```

Figure 3 - BGE fuction from Alive Test translation to assembly

On BGE function that checks BGE RISC command, we expected to see '1' on the memory address with 1 entered to a4 and 7 entered to a5 (with branch **not taken**).

after looked in the generated memory output file we saw that this memory address (400f54) contains '0', which indicated branch is taken and the Program counter branched to 594 instruction address.

Debugging the core signals on the ModelSim waveform we noticed the branch comparator is indeed recognize the BGE command but the input wires were mixed by a typo : both logic wires entered to comparator were actually the same wire!

```

n : branch_comp
condition.
z ({CtrlBranchQ102H , Funct3Q102H})
: BranchCondMetQ102H = (AluIn1Q102H==AluIn2Q102H) ;// BEQ
: BranchCondMetQ102H = ~(AluIn1Q102H==AluIn2Q102H) ;// BNE
: BranchCondMetQ102H = ($signed(AluIn1Q102H)<$signed(AluIn2Q102H)) ;// BLT
: BranchCondMetQ102H = ~($signed(AluIn1Q102H)<$signed(AluIn1Q102H)) ;// BGE
: BranchCondMetQ102H = (AluIn1Q102H<AluIn2Q102H) ;// BLTU
: BranchCondMetQ102H = ~(AluIn1Q102H<AluIn2Q102H) ;// BGEU
: BranchCondMetQ102H = 1'b0 ;

```

Figure 4 - Branch comperator code from GPC_4T core with the typo

as seen in figure FIXME , *AluIn1Q102H* which holds the number 1 is compared with the same *AluIn1Q102H* (not like the other comparations) and the condition excepted was $\overline{(1 < 1)}$ which is logic '1' and thus the branch was taken with *BranchCondMetQ102H* signal assigned to logic '1'.

After fixing the typo to *AluIn2Q102H*, the condition excepted was $\overline{(1 < 7)}$ which is logic '0' and thus the branch is not taken as expected.

2.2.3 CRs Features & Dedicated tests

CRs abilities explained on HAS and MAS. In our tests, we tested some of the features:

- Check current thread ID – on every multi thread test we tested, we used this feature (implementation of this explained on MAS) and a multi thread application will be detailed in the next section.
- Check current core ID.
- Freeze test using CRs – in the application that validates this feature, each thread runs different code section (using switch case). Thread 1,2,3 are writing to the CR address that freezes PC to freeze their own PC. Thread 0 runs some arithmetic calculations and write the result on the shared memory. When Thread0 finished calculating, He unfreeze the other threads and do busy waiting(reading CR called SCRACHPAD values). The unfreezed threads

now runs arithmetic calculations and when they finished they signal thread0 to stop the busy wait by writing to the scratchpad and thread0 will finished running Main program and the test will terminates. On the log file of the shared memory accesses, we can see that thread 1,2,3 write the result long time after thread 0. All threads ran the same calculations with minor changes (should take the same time):

Time	Thread	PC	Address	Read/Write	data
4120.0	00	00000238	00400f00	WRITE	00000064
6350.0	01	00000328	00400f0c	WRITE	00000190
6490.0	02	000002c0	00400f04	WRITE	000000c8
6700.0	03	000002f4	00400f08	WRITE	0000012c

2.2.4 Multi Thread application

For each Multi thread application we tested, we use the same structure:

The core and all 4 threads run the same instructions but each thread runs different section of the code by using a Switch-Case statement that checks the CR holding the current thread ID (explained on MAS).

on each case statement, every thread runs different function.

One example is the matrix multiply test: In this test there are two 4x4 matrixes.

Each thread takes one row(row number according to thread's id) from first matrix and use matrix multiply with the columns of matrix 2 and writing the result (a vector that will be the result matrix row number "thread ID") to the shared space.

Using the multi thread feature in this test we can perform parallel commands.

When we tested a matrix multiplication when all threads run the same code, it took a lot longer.

```
int main() {
    int x = CR_THREAD[0];
    int mat1[4][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,16}};
    int mat2[4][4] = {{13,14,15,16},{9,10,11,12},{5,6,7,8},{1,2,3,4}};
    int row[4];
    switch (x) //the CR Address
    {
        case 0x0 : //expect each thread to get from the MEM_WRAP the co
            for( int i = 0 ; i < 4 ; i++)
                row[i] = mat1[x][i];
            Thread(row, mat2, 0);
            break;
        case 0x1 :
            for( int i = 0 ; i < 4 ; i++)
                row[i] = mat1[x][i];
            Thread(row, mat2, 1);
            break;
        case 0x2 :
            for( int i = 0 ; i < 4 ; i++)
                row[i] = mat1[x][i];
            Thread(row, mat2, 2);
            break;
        case 0x3 :
            for( int i = 0 ; i < 4 ; i++)
                row[i] = mat1[x][i];
            Thread(row, mat2, 3);
            break;
    }
}
```

Figure 5 - Matrix multiplication Multi Thread application code. CR_THREAD defines the address 0x00C00004

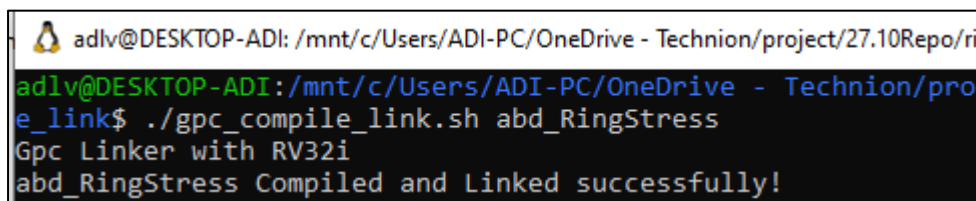
2.2.5 Automation & Scripts

To make the validation stages more efficient we used automation scripts that made all the validation stage easy and swift.

gpc_compile_link.sh

"gpc_compile_link.sh" is a Unix bash script. It's written in bash because the Toolchain works on Unix/Linux environment. The script received a path to c file as argument, wrap and initiate the [4 Toolchain compiling commands](#) and moves all the created files (C , elf, assembly , Verilog) to a newly created folder with C program's name located on the test bench path. The simulation script takes the files from that path.

The name "gpc_compile_link" derived from the wrapped Toolchain command inside : it compiles the C program and links it with the gpc initialization assembly code.



```
adlv@DESKTOP-ADI: /mnt/c/Users/ADI-PC/OneDrive - Technion/project/27.10Repo/ri
adlv@DESKTOP-ADI:/mnt/c/Users/ADI-PC/OneDrive - Technion/pro
e_link$ ./gpc_compile_link.sh abd_RingStress
Gpc Linker with RV32i
abd_RingStress Compiled and Linked successfully!
```

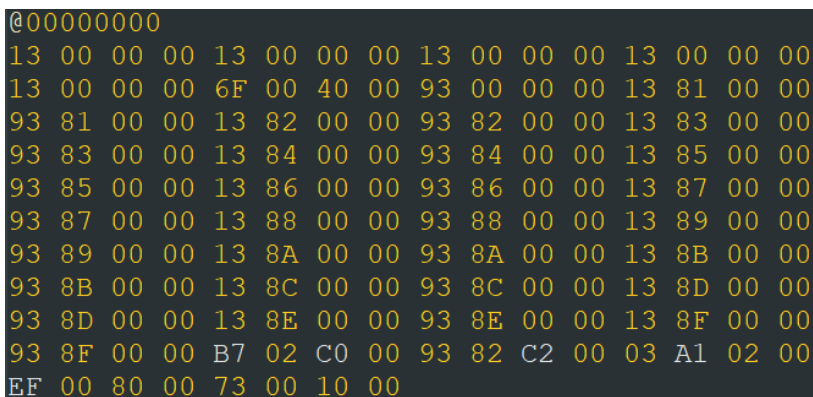
Figure 6 - gpc_compile_link.sh executes on "abd_RingStress.c"

3 SYSTEM VERILOG TEST BENCH

Test Bench is an environment used to verify the correctness or soundness of a design or model. The GPC_4T code is combined from several Verilog code designs : the core, the instruction and data memory, the memory wrapper which instantiates the memory and add wrapping functionality (explained on MAS) and the gpc_4t itself, which instantiates all the designs and connects them to the GPC_4T unit.

3.1 TEST BENCH (GPC_4T_TB.SV)

In this project, we design a simulatable Test bench named "gpc_4t_tb". The TB generate a clock and reset signals, creates an instance of the gpc_4t model and connects the clock and reset to it. For GPC_4T to work and run instructions, the model needs memory. The TB reads an SV file of a specific compiled C program , which is a serial 8 hexadecimal digits instructions dump file and load it to a local array.



```
@00000000
13 00 00 00 13 00 00 00 13 00 00 00 13 00 00 00
13 00 00 00 6F 00 40 00 93 00 00 00 13 81 00 00
93 81 00 00 13 82 00 00 93 82 00 00 13 83 00 00
93 83 00 00 13 84 00 00 93 84 00 00 13 85 00 00
93 85 00 00 13 86 00 00 93 86 00 00 13 87 00 00
93 87 00 00 13 88 00 00 93 88 00 00 13 89 00 00
93 89 00 00 13 8A 00 00 93 8A 00 00 13 8B 00 00
93 8B 00 00 13 8C 00 00 93 8C 00 00 13 8D 00 00
93 8D 00 00 13 8E 00 00 93 8E 00 00 13 8F 00 00
93 8F 00 00 B7 02 C0 00 93 82 C2 00 03 A1 02 00
EF 00 80 00 73 00 10 00
```

Figure 7 - A program sv file. each 8 digits is a RISC V 32-bit instruction that will be decoded by the core

The TB then use this array and performs a **Backdoor** load directly to the GPC_4T instruction memory: it assigns the file loaded array as the instruction memory of the GPC_4T. If the program using a heap memory addressing then the TB also does the same to Backdoor load the heap to the GPC_4T data memory.

After instancing the gpc_4t model and loading an Instruction memory, the Test bench is good to go and the GPC_4T 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 message display that will be printed in simulation program and closing all open newly created log files that created by the log generator.

3.2 GPC_4T LOG GENERATOR(SPC_4T_LOG_GEN.SV)

The "gpc_4t_log_gen" is a system Verilog file design to fulfil a very important task : generate trackers during test bench simulation run. code contains several of file descriptors(integers), each used to open a designated tracker file for the current running program in the simulation. The design using XMR method to sniff a variety of signals and data from the gpc_4t modules and writes the sampled data on the tracker log file. For example file named "trk_d_mem_access" is created and each time the control signal from the core that indicated there is an access to the data memory, an information about that transaction written to the file : the time of the transaction, the thread which accessed, the program counter of the instruction, the address in the data memory that accessed and the data written to that address.

after simulation is finished, the log files are ready to read in a dedicated folder named after the program. these logs are a powerful tool for debug and for verification.

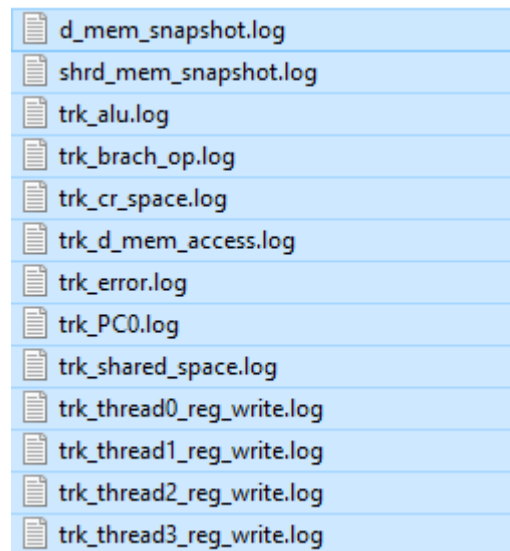


Figure 8 - tracker files generated after simulation

3.3 END OF TEST SNAPSHOT CHECKER

During the work of the project, we tested many C programs such as array sorting, matrix multiplication, prime factoring, multi thread programs (where each thread performs different calculations) and many more. On each of the programs we wrote the result to a data memory area called shared space area (it is an area in the data memory where all thread can access). Before running the program we created a special memory snapshot called "Golden Shared Mem" which is the snapshot of a memory as it should be after simulation.

During simulation the test bench generates a snapshot of the shared space.

In this verification stage, we compared between the Golden snapshot and the generated snapshot. if both are identical then verification completed successfully and the GPC_4T ran the program as expected. If difference found between snapshots, then **debug** is needed and for that we use the waveform or the tracker logs.

shrd_mem_snapshot.log - Notepad	golden_shrd_mem_snapshot.log - Notepad
File Edit Format View Help	File Edit Format View Help
Offset 00400f00 : 00000000	Offset 00400f00 : 00000000
Offset 00400f04 : 00000001	Offset 00400f04 : 00000001
Offset 00400f08 : 00000002	Offset 00400f08 : 00000002
Offset 00400f0c : 00000003	Offset 00400f0c : 00000003
Offset 00400f10 : 00000005	Offset 00400f10 : 00000005
Offset 00400f14 : 00000006	Offset 00400f14 : 00000006
Offset 00400f18 : 00000009	Offset 00400f18 : 00000009
Offset 00400f1c : 00000032	Offset 00400f1c : 00000032

Figure 9 - Comparison between the golden snapshot and the GPC_4T memory snapshot after performing array sort

3.4 ASSERTIONS

One more purpose of the log gen are assertions. Assertions used to "catch" unwanted behavior of the GPC_4T and on some occasions terminate the simulation. For example, an assertion named "BadMem_R&W" triggers if Read and Write control signals are toggled in the same clock cycle. A message will be printed on simulation terminal error information will be written to a file, and simulation will stop. Another assertion will terminate the simulation when EBREAK command is identified (EBREAK is coded at the end of each program because of the [gpc initialization and linker](#))

```
//assertions//
always_comb begin
    if(gpc_4t_tb.gpc_4t.core_4t.AssertBadMemAccessReg)begin
        $fwrite(trk_error,"ERROR : AssertBadMemAccess - Memo
    end
    if(gpc_4t_tb.gpc_4t.core_4t.AssertBadMemAccessCore)begin
        $fwrite(trk_error,"ERROR : AssertBadMemAccess - Memo
    end
    if ( gpc_4t_tb.gpc_4t.core_4t.AssertBadMemR_W)begin
        $fwrite(trk_error, "ERROR : AssertBadMemR_W - RD && W
        end_tb(" Finished with R\W Error");
    end
    if(gpc_4t_tb.gpc_4t.core_4t.AssertIllegalRegister) begin
        $fwrite(trk_error, "ERROR : AssertIllegalRegister - 
    end
    if(AssertEBREAK) begin
        end_tb(" Finished with EBREAK command");
    end
    if(gpc_4t_tb.gpc_4t.core_4t.AssertIllegalPC) begin
        $fwrite(trk_error, "ERROR : AssertIllegalPC",$realtir
        end_tb(" Finished with PC overflow");
    end
    if(AssertIllegalOpCode) begin
        $fwrite(trk_error, "ERROR : AssertIllegalOpCode - IL
    end
end //always_comb
```

Figure 10 - Assertion as seen at log gen code

3.5 RTL COMPLICATION (LIST.F)

Before initiate the simulation, the test bench needs to be compiled to a ModelSim runnable execution file. Because the test bench uses an instance of the complete GPC_4T and all of its modules, all the project source code RTL files need to be compiled as well.

Before compiling, a path to the desired SV instructions file of the program that will be loaded as the instruction memory need to be noticed on the test bench code.

The compilation is perform by the ModelSim program with the ModelSim GUI after loading and defining all of the project source code , or with a PowerShell command that initiate the same as the GUI with the help of an f file called List.f : a file that contains a list of all source code and test bench paths for the ModelSim to use in the compilation.

```
-- Compiling package lotr_pkg
-- Compiling module gpc_4t
-- Importing package lotr_pkg
-- Compiling module core_4t
** Warning: ../source/gpc_4t/rtl/core_4t.sv(52): (vlog-13314) Defaulting port 'CRQnnnH' kind to 'var' rather than 'wire'
due to default compile option setting of -svinutputport=relaxed.
** Error: ../source/gpc_4t/rtl/core_4t.sv(448): (vlog-2730) Undefined variable: 'AluOutQ2H'.
** Error: ** while parsing macro expansion: 'LOTR_MSFF' starting at ../source/gpc_4t/rtl/core_4t.sv(476)
** at ../source/gpc_4t/rtl/core_4t.sv(476): (vlog-2730) Undefined variable: 'PcQ10333H'.
```

Figure 11 - Compilation errors and warnings while compiling the core module

The compilation process also helps to discover code errors as the errors detailed by the ModelSim compiler as seen on the figure above.

3.6 LOAD BACKDOOR I_MEM

As explained [before](#), The Test Bench use an sv file contains text vectors that are the encoded RV32I commands that generated by the Toolchain GCC. It reads this file to an array and performs a **Backdoor** load directly to the GPC_4T instruction memory using XMR: it assigns the file loaded array as the instruction memory module of the GPC_4T.

Because the GPC_4T instance is created, the XMR path to the I_MEM will be :

gpc_4t_tb.gpc_4t.i_mem_wrap.i_mem.mem

using this “path” we push the array of text(sv file) and make the GPC_4T thinks that is a real instruction memory.

If the program using a heap memory addressing then the TB also does the same to Backdoor load the heap to the GPC_4T data memory.

```
initial begin: test_seq
//=====
//load the program to the TB
//=====
$readmemh("../verif/Tests/",hpath,"/",hpath,"_inst_mem_rv32i.sv"), IMemQnnnH);
$readmemh("../verif/Tests/",hpath,"/",hpath,"_data_mem_rv32i.sv"), DMemQnnnH);
// Backdoor load the Instruction memory
gpc_4t_tb.gpc_4t.i_mem_wrap.i_mem.next_mem = IMemQnnnH[I_MEM_OFFSET+SIZE_I_MEM-1:0];
gpc_4t_tb.gpc_4t.i_mem_wrap.i_mem.mem = IMemQnnnH[I_MEM_OFFSET+SIZE_I_MEM-1:0];
// Backdoor load the Instruction memory
gpc_4t_tb.gpc_4t.d_mem_wrap.d_mem.next_mem = DMemQnnnH[D_MEM_OFFSET+SIZE_D_MEM-1:D_MEM_OFFSET];
gpc_4t_tb.gpc_4t.d_mem_wrap.d_mem.mem = DMemQnnnH[D_MEM_OFFSET+SIZE_D_MEM-1:D_MEM_OFFSET];
#200000
end tb(" Finished With time out");
end: test_seq
```

Figure 12 - The Backdoor code of I_MEM and D_MEM with hpath as the path of the sv file to set as instruction memory

3.7 SIMULATION

After compilation finished, the simulation process can be performed. The simulation is the most powerful tool in the validation process . The simulation performed by ModelSim, and the result can be reviewed Using the ModelSim Waveform GUI or with well written log files.

With the simulation we can track the workflow of the GPC_4T pipe stages and can debug issues.

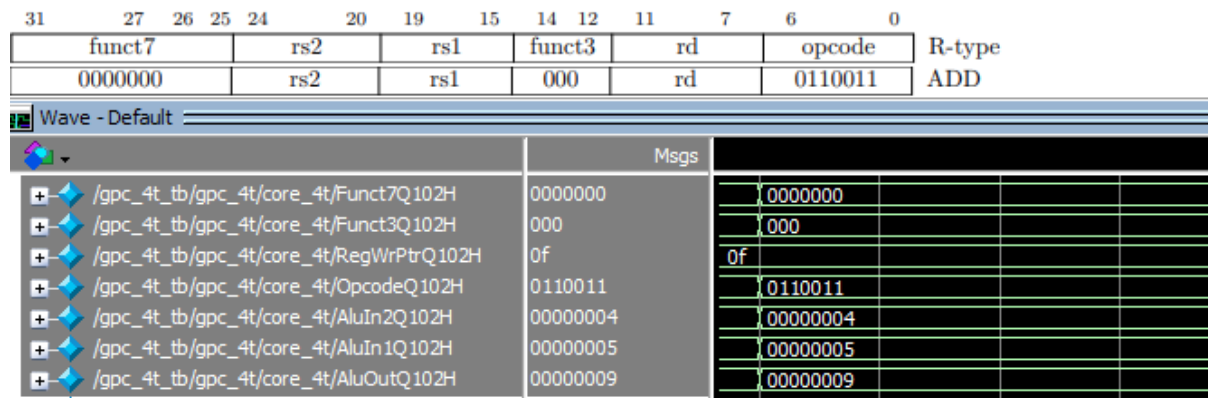


Figure 13 - Waveform of a simulation with ADD command from RISC V spec . with the waveform we can notice the values of the logic signals inside the core. The ALU performs addition between 4 and 5 as ALU inputs and the result on the ALU output. all in execution pipe stage (Q102). you can see funct3 , funct7 and the opcode of ADD command. The result is 9 as expected so we can notice that ADD instruction works as expected.

During the simulation the test bench and the log generator generates the log text files which can also be used in debug and in verification.

Time	PC	Alu Op	AluIn1	AluIn2	AluOut
2050.0	000000cc	OP_OP	00000005	00000004	00000009
2060.0	000000cc	OP_OP	00000005	00000004	00000009
2070.0	000000cc	OP_OP	00000005	00000004	00000009
2080.0	000000cc	OP_OP	00000005	00000004	00000009

Figure 14 - ALU tracker log file generated during simulation. Here we can also indicate that ALU work as expected with the addition of 4 and 5 gave 9 in the ALU out. All 4 threads running the same program so 4 clock cycles look the same.

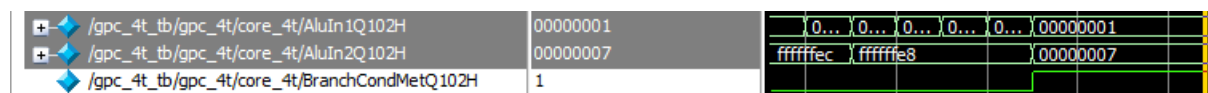


Figure 15- Alive Test debug : BGE a4(=1) a5(=7) shows branch taken although not as expected. More on this bug in previous [Alive Test debugging](#)

3.8 AUTOMATION & SCRIPTS (TB_SIM.PS1)

"tb_sim.ps1" is a Windows PowerShell script. It's written in PowerShell because the ModelSim can be operate (compilation and simulation) with commands from the PowerShell Terminal. This script actually wraps together 3 of the validation stages : (1)RTL compilation & (2)test bench simulation with the ModelSim, with option to simulate with or without GUI interface, and also (3)verification by comparing the memory snapshot output and the golden memory.

The script gets unlimited number of arguments , each is a name of a program dedicated to run on GPC_4T and each has a folder contains the program dedicated files send from "gpc_compile_link" bash script, and perform serial validation operation on each : compile, simulate and verify.

During script is running, information messages prints on the terminal.

The script can also get

"All" as an argument

and with this

argument it will

perform the

operation on each of

the folders located in

the test bench tests

path.

Figure 16 - tb_sim script running on the PowerShell terminal. The script perform validation operations on Binary Search test

```

#####
#####~ Adi & Saar GPC_4T Simulation TestBench ~#####
#####

There are a total of 1 test entered

*****Compiling Test number 0 : Binary_Search test*****

Initiate Compilation...
.
.
.
Model Technology ModelSim - Intel FPGA Edition vlog 2021.1 Compiler 2021.02 Feb  3 2021
Start time: 16:04:03 on Nov 24, 2021
vlog "+define+HPATH=Binary_Search" -f ..\source\gpc_4t\gpc_4t_list.f
-- Compiling package lotr_pkg
-- Compiling module gpc_4t
-- Importing package lotr_pkg
-- Compiling module core_4t
** Warning: ../source/gpc_4t/rtl/core_4t.sv(52): (vlog-13314) Defaulting port 'CRQnnH' kind to 'var'
due to default compile option setting of -svinutputport=relaxed.
-- Compiling module d_mem
-- Compiling module cr_mem
-- Compiling module d_mem_wrap
-- Compiling module i_mem
-- Compiling module i_mem_wrap
-- Compiling module gpc_4t_tb

Top level modules:
  gpc_4t_tb
End time: 16:04:03 on Nov 24, 2021, Elapsed time: 0:00:00
Errors: 0, Warnings: 1
.
.
.
Compilation Ended. Details above

*****Simulating Test: Binary_Search*****

Initiate Simulation...
.
.
.

```