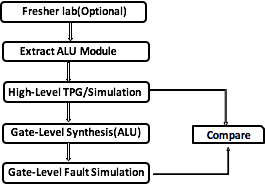
****

**Fresher Lab(Optional)** - Refreshes students’ knowledge of VHDL language and Modelsim simulator.

In this lab, the [Guide for model sim](http://priit.ati.ttu.ee/?page_id=2033) will help you get familiar with Modelsim or Questasim tool. You can skip the guide if you feel comfortable with modelsim or Questasim already

A full adder is a digital circuit that adds two binary numbers including values carried in (carry-bit) from the previous less significant stage[3]. The inputs of a full adder are usually represented as A, B and C-in. While the outputs are sum and carry\_out. Fig 1 and table 1 show the schematic and truth table of a 1 bit adder

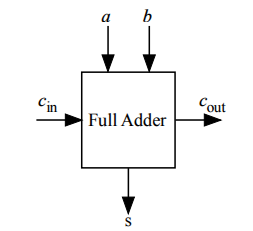
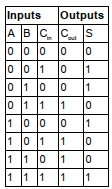
 

Fig 1. Schematic of 1-bit full adder Table 1. Truth table of 1-bit full adder

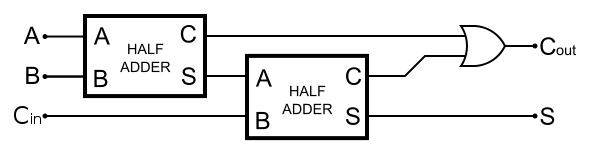


Fig 2: 1 bit full adder block diagram

In this task, a 2-bit adder Implemented using 1-bit full adder as component is given. The 1-bit full adder was built using 2 half adders as shown in the fig 2.

VHDL files 1\_bit\_adder, 2\_bits\_[adder](https://drive.google.com/open?id=0Bw0HjDYLy-4_WkpBUE5iRWp4WjA) and [adder](https://drive.google.com/open?id=0Bw0HjDYLy-4_OFp4VndDSTJxQ2M)\_tb for a 1-bit adder, 2-bits adder and testbench are provided respectively.

**Task:**

1. Using the guide above, simulate the 2-bit adder, if there are error, you are required to correct them. Once you are able to correct the errors, check that your simulation result looks like figure 2

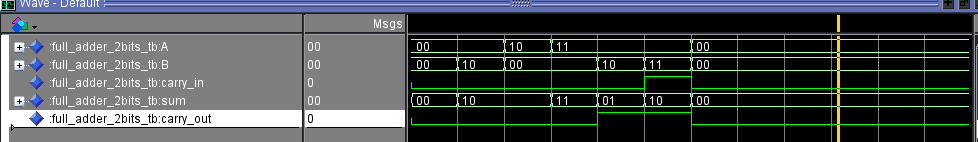


Figure 3: Simulation result of a 2-bit adder

1. Modify the testbench given to add all possible combinations of inputs A and B. Write the output of the simulation to a file named adder\_output.txt. The output should be in the form: A | B | SUM | Carry

**Questions**

-What type of adder is implemented in this exercise?

- Can you combine 1-bit adder and 2-bit adder into a single vhd file?

- What is the difference between ripple\_carry adder and carry look ahead adder

**Extracting ALU of processor and Simulate Behaviour**

The miniMIPS is a 32 bits core and has a Von Neumann architecture. It isa 5-stage pipeline processor and the stages are as follows:

- Instruction extraction

- Instruction decoding

- Execution

- Memory access

- Update registers

The processor contains 32 registers and the schematic of the processor is given below.

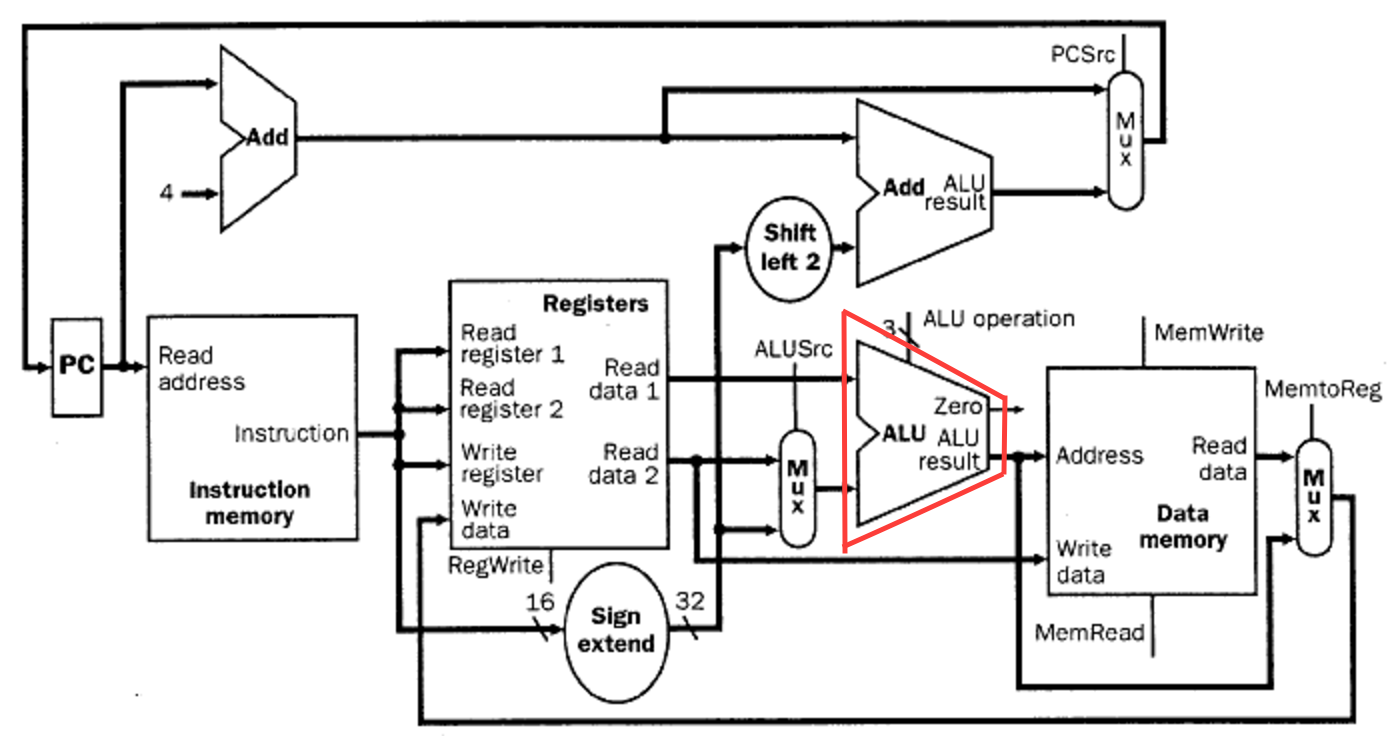


Fig 4. Schematic of miniMIPS processor

Our target is to use the angel's approach where ***we verify the correctness of sub-circuits***. We can exploit the knowledge of already tested sub-circuit to extend the system step by step until we have completely tested the whole system. For the purpose of this lab, we will extract the ALU unit and test it. Extending the system is beyond the scope of this lab.

For this experiment, we have provided a miniMIPS processor which has several modules including the ALU. You are required to extract the ALU module(The red part of the schematics). A testbench is provided for the ALU. You will need to simulate all the ALU instructions as given by the instruction set architecture and write the output to a file called output.txt. This functional result will be used in subsequent labs

**Task:**

1. Create a directory, name of directory could be your choice. This directory will contain the extracted ALU module, testbench for simulation, input test pattern file for the ALU and simulation script in python. Some of the files are provided in lab2.
2. Copy the ALU and pack\_mips from the miniMIPS directory ***(Lab2/minimips/trunk/miniMIPs/src)*** into the file created. Copy also the testbench.vhd, my\_package.vhd, input.txt, merge.py, randomGenerate.py, simulate.do and simulate.py files as provided. ***Merge.py***, ***RandomGenerate.py****,* ***Simulate.do and simulate.py can be located in the Lab 3 folder***. The testbench takes input for each data operand for the ALU from the input file and simulate all instructions of the ALU, functional result of simulation is written to a file out.txt
3. ***Open simulate.do and remove the src directory from the vcom command directory navigation***.
4. Open a terminal window and navigate to the folder which contains simulation files. Run simulate.py or ***python simulate.py***. This will invoke modelsim. Ensure that you have chosen the configuration for the environment if running from the lab class. To do this enter the **cad** command twice and chose **option 2**(Mentor Graphics 2017 EDA version) to be in ModelSim environment.

***Enter test length [1500] and bit length [32].***

NOTE: when asked to exit modelsim, click no if you would like to add waves and visually verify the simulation.

1. **If QuestaSim/ModelSim opens without running simulate.py or does not display the waveform or any evidence that the command worked, manually enter the commands in *Simulate.do* step by step.**
2. Observe the output file. For every pattern pair, there is a functional output per instruction set
3. **Optionally**, you could add waves for each signal to modelsim and restart the simulation for visual verification. For help, consult the guide in the previous lab

**Generating high level test pattern using**

1. Random Approach
2. Random Greedy Approach
3. Compare 1 and 2 in terms of test length and time of generation (can you explain why?)

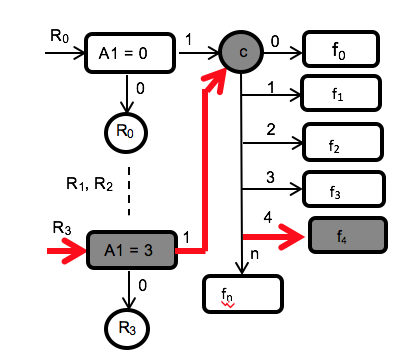
In this lab, we focus on testing of the ALU module of processor cores. We consider a typical subset of instructions of the ALU of a miniMIPS processor in Table 1. The operations are represented by operation codes and the related formulas *fi* for calculation of the output values of ALU (This can be obtained from simulation of the module). The high-level structure of an ALU is depicted in Fig 1. The control variable *c* can have values from the domain {0,1,2,3…,n}. We denote by *Ii* the instruction (with opcode *ci*) which performs in ALU the function *fi*.

***Table 1.*** *Instruction subset for an ALU of a microprocessor*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mnemonic | *fi* | *Opcode c* | Mnemonic | *fi* | *Opcode c* |
| ADD | *f*0 |  | SLT | *f*8 |  |
| ADDU | *f*1 |  | SLTU | *f*9 |  |
| SUB | *f*2 |  | BEQ | *f*10 |  |
| SUBU | *f*3 |  | BNE | *f*11 |  |
| AND | *f*4 |  | BLTZ | *f*12 |  |
| OR | *f*5 |  | BGTZ | *f*13 |  |
| XOR | *f*6 |  | BLEZ | *f*14 |  |
| NOR | *f*7 |  | BGEZ | *f*15 |  |

We can represent the instruction set in Table 1 by the High-Level Decision Diagram (HLDD) in fig 1. The HLDD has a single decision node labeled by the control variable c and 16 terminal nodes labeled by the functions *fi* implemented in the ALU data path and selected by instruction *fi* respectively. The node c represents the whole control part of the ALU. In general case, if the system is described by more control variables (representing control fields of the instruction word, register addresses, flags, conditions etc.), the internal structure of the HLDD will be as well more complex. In this case, we will assume only one control variable.

Each instruction in Table 1 can be modelled by the related path in the HLDD model. When simulating an instruction, its related path in the HLDD is activated. For example, when simulating the instruction f4, the following paths *l* in Fig.1 are activated: *l*(*A*1, *c*, *f4*). The paths are highlighted by bold edges and grey coloured nodes in Fig.1.



***Figure 1.*** *HLDDs for the processor given by instructions in Table 1*

we would be generating test patterns for testing the control path c using two different methods.

The first approach is purely random. Here we have a huge search space of random data and iterate through each data while checking its contribution to the fault coverage of the test for the control path. Data that contributes to the coverage of the test is added to the list if patterns needed until we reach 100% fault coverage or detect redundant faults if present. The second approach is called a greedy random method. Here we search through the entire search space and select only patterns with the best contribution to the fault coverage until we achieve a 100%.

**Task:**

*You have been provided with two tools for generating high level test patterns using purely random and greedy random algorithms respectively.* You have `also been provided with a tool that generate any number of random data depending on the search space desired and the word length of the unit to be tested. In [lab3](https://drive.google.com/open?id=0Bw0HjDYLy-4_UE9BdEFrdGdLb2s) you have a directory called High-level-TG-Lab which contains two subdirectories (src and patterns) and some set of scripts.

The src subdirectory contains the following file:

* alu.vhd : ALU sub-circuit of the processor
* pack\_mips.vhd: the package file for the processor.
* my\_package.vhd: Package file for testbench
* testbench.vhd: Testbench for ALU simulation

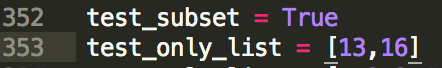
Other files in the High-level-TG-Lab are:

* randomGenerate.py : generates random set of data operands depending on the search space and word length specified.
* merge.py:
* Simulate.py: invokes modelsim simulator
* simulate.do:

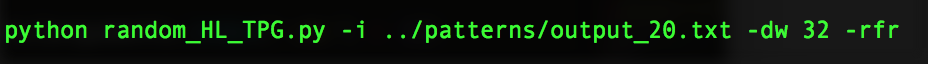
1. Create a directory and copy all the content of High\_Level\_TP\_Generation in lab3 into it.
2. From terminal, change directory to your directory and enter the **cad** command twice and chose **option 2**(Mentor Graphics 2016 EDA version) to be in ModelSim environment.
3. Run simulate.py using python command. Specify the length of search space and data word. e.g(1500, 32). This will functionally simulate the ALU and exercise all the instruction set as described in the testbench. The functional data (output.txt) generated from this will be used for generating test data by the high-level test data generation tools. (you can locate this file in **generated\_file** directory)



1. Edit package.py file and check that test\_subset = True. This enables option to generate test patterns for only selected instructions. If set to false, all instructions in the ALU will be exercised. In the test\_only\_list list, modify to contain only instruction set you have been given. The list of instructions and numeric arrangement is given in …….





1. Change directory to src, execute **random\_HL\_TPG.py**. The options are presented. You example: 



**NB:**

**In step 4:**

Under the *test\_only\_list,* in accordance variants below, F1 represents instruction ADDU *(according to table 1)*. The instruction set are under “pre\_determinde\_patterns” in the package.py file. You need to comment the previous “test\_only\_list” and replace with the new instruction set, according to the variants.

**In step 5:**

In my case, ../patterns/output\_20.txt file was somehow missing after it generated. After several trials, the file was not gotten. So, I copied the “output.txt” file in *“my\_folder/sim\_generated\_file”* into *“my\_folder/patterns”*. I then used it to process the command in step 5.

**Output:**

I found an interesting file, titled table.txt. The file contains the instruction set entered in the package.py file and some generated text bits.

**Variants:**

1. Search space: 250, 500, 750, 1500: Instructions ADD, SUB, OR, MULT
2. Search space: 200, 400, 800, 1600: Instructions ADDU, SUB, XOR, NOR
3. Search space: 100, 300, 650, 1450: Instructions SUB, SUBU, AND, SLT

500

Look at the instruction set and understand how it works.

*Graph not necessary for us right now*

Template for the

Instu sect can be rep by such a graph.

We call it HL DD.

INSTR belongs to the same grp and beleong to the ssame template

Extraction, takin out the ALU from the MP.

In mimiIPs, extraction f ALU ia a lil easier, cz you don’t have to veerif.

Find out the feasibility of the rxtration fror abither orofessor,

TetraMax is usefu for faults.

If we donot’’ want expier self Sim---UnderGifPatterns, Tetramax..patterns are inputs for ALU.

Program is running and weach pattern coming here is sored. All it takes is to

Control part, we mess with the HL patters.

Control test, we are satisfied.

1 bit – 8 combo, 2 – wh

Whjen generating test data, always look out for human fvidfs to test data is reured.