****

**CSCE330402 - Digital Design II**

**Project 2 Report**

**first deliverable**

**Fall 2020**

Submitted By:   Nouran Abdelaziz   900170863

                          Ashrakat Elkhalifa    900170051

Amr Abdelazeem 900172775

Hussam Gawish 900171054

1. **Project Description:**

This project is to design Ibex based Soc chip using open sources.

1. **Tools and Sources Used:**

* IBEX CPU source code:
* This is a complete open source code implementation of the IBEX core which we use in our SoC.
* Caravel chip source code:
* This is the source code for the layout of the chip in which we will use the mega project area to put or IBEX based SoC.
* SocGen:
* This is an open source tool which takes the jason description of our system and automatically generates the verilog files out of it. It also has built in IPs and masters libraries.
* OpenLane:
* Finally, this is an open source tool which will help us to go through the ASIC flow without human intervention by taking the verilog implementation of our chip and producing a design clean GDS2 files.

1. **IBEX Hardening:**

In the hardening process, we cloned a copy of the ibex files from the repo <https://github.com/lowRISC/ibex>.

We used the makefile and the available scripts to generate one of the possible configurations, in the meanwhile as we do more research we decided to use the “small” configuration with flipflop register files (with an area of 26.06kGE according to yosys), at this moment it is not the smallest but it has the best verification status.

Afterwards we used the provided sv2v scripts to convert the System Verilog files implementing the ibex core to older Verilog files (which is necessary to get them to run through yosys, and thus necessary for the yosys and abc component of openlane).

We then started to test hardening the ibex core using openlane with different configurations to try and reach the best performance/size. We identified two approaches to this, a) we harden the ibex core separately and then simply link it to the SoC generated from socgen as a hard macro, then rerun this through openlane, or b) we flatten out both parts, by linking the ibex core to the SoC then doing the hardening step once. The second approach should be more efficient in terms of sizing, but could face more congestion problems and would be heavily intensive in terms of computation power, meaning that using available university resources would be essential. We have generated all the three stages and their verilog codes of the IBEX Core.

1. **SocGen Test:**

In the SocGen test:

We got a copy of the project up and running on our local machine for development and testing purposes using: git clone https://github.com/habibagamal/SoC\_automation.

Then,when we tried to run the script compile.sh in SoCGen test, we got the following two errors: one in the build.sh, where a path to the riscv64-gnu-compiler and the other error was a syntax error in the real\_dump.vvp. Therefore, we downloaded the riscv64-gnu-compiler using: git clone --recursive <https://github.com/riscv/riscv-gnu-toolchain>. Then we put the path of the generated riscv gnu compiler in the build.sh to be able to build test cases from c code that can run on the core.We will be using the N5 cpu demo to be able to test the SoCGen. We generated the main.hex file.Base\_addr.H also was generated to test the C-files. Starting addresses with 40 defines the APB subsystem,as the addresses of the subsystem starts with 40.

The test passed but the I2C test was not passed.

1. **AHB Master Interface for IBEX:**

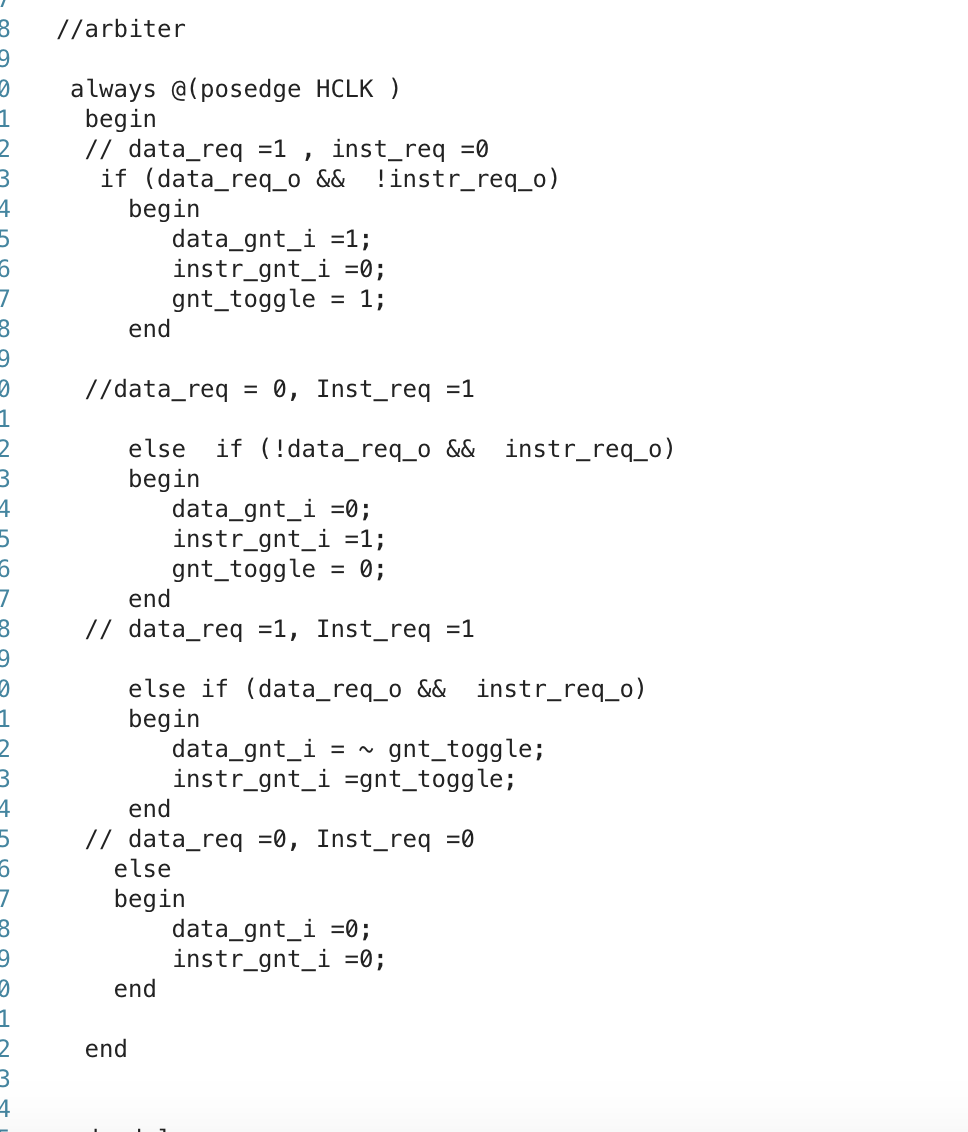
* Problem Statement:
* SocGen currently supports only masters that are compatible with AHB while IBEX core is not an AHB master, so we need to make the IBEX core compatible with AHB.
* The Ibex core has two interfaces, data interface and instruction interface, so we will need an arbitrary to regulate the transmission between two interfaces
* Solving the problem:
* The IBEX\_core module is the top module of the IBEX open source code, and we want to convert this module to an AHB master to be able to generate the SoC through SocGen. That is why we created an IBEX module that acts as an AHB master which takes the exact inputs and outputs of the original IBEX\_core but it manipulates the inputs and outputs signals that are related to extraction of instructions in the fetch stage or data in the load-store stage.
* The IBEX core which acts as the AHB master is fed to the SocGen which handles the generating of the verilog code of the AHB bus and other IPs in our system.
* IBEX wrapper inputs and outputs explanation:

Inputs:

* HCLK: It is connected directly to the ibex core’s clock (clk\_i).
* HRESETn : It is connected directly to the ibex core’s reset (rstn\_i).
* HRDATA: It is the read data coming from AHB bus and it is connected to both the read data for the ibex core’s instruction and data interfaces. (instr\_rdata\_i and data\_rdata\_i)
* HREADY: It is connected to data\_rvalid\_i and instr\_rvalid\_i which indicates how many clock cycles the data phase will take.

Outputs:

* HSIZE: It is set according to ibex cores byte\_en\_o signal ( explained more below )
* HADDR: It is not aligned but the address coming from ibex core is word aligned that is why we set it also according to byte\_en\_o.
* HTRANS: It is assumed that the trans will always be non-sequential as we will not have any incremental or wrapped bursts for now.
* HWDATA: It comes from the ibex core’s data\_wdata\_o which is the data that the core wants to write.
* HWRITE: It comes from the ibex core’s data\_we\_o signal which tells whether it is write or read operation.
* HSIZE and HADDR from byte enable Explanation:
* As we know, Hsize is the size of the bytes that will be read or write, it can be byte, half world or a world. We deduce HSIZE from the byte\_enable signal which is an output from the ibex\_core module.
* If the byte enable contains only one of ‘1’ , then the HSIZE will be [0,0,0] which is equivalent to 1 byte.
* If the byte enable contains 2 of 1’s on its signal then it indicates that 2 bytes (half word) are required to be read/write, the the HSIZE will be [001] which is equivalent to half word
* If the byte enable contains3 of 1’s ts signal then it indicates 3 bytes. However, a word or half word can be read/write so we assign HSIZE [010] that indicates that a word is read/written.
* The address offset is also set according to each case then the 2 least significant bits are replaced by the address offset.
* IBEX Wrapper arbiter logic:
* There are two ibex core output signals which indicate when the instruction interface or data interface wants to perform a transaction which are data\_req and istr\_req.
* There are another two ibex core input signals which indicates which of those two is allowed to perform the transaction in this clock cycle so there will be no new address produced from an interface unless its relenvent grant is equal to one.
* The way we assign instr\_gnt and data\_gnt is round robin which means that if both interfaces are requesting then we will grant the one which did not perform the last transaction and if only one of them is requesting then this interface will be granted the transaction.



1. **TestBench:**

* We decided to use a system generated by socGen and testbench generated to test our wrapper.
* We copied the files generated in the N5\_demo and copied to the directory where our wrapper.v and ibex module files reside, added the files to our list and replaced instances of calling N\_five modules by our wrapper.
* However, it needs further investigation as it seems that instructions were not read from memory.