*Design of a General-Purpose Microprocessor Using Software and Hardware Components*

*Phase 1*

*EE577A - Final Project*

Samuel Bruner  
Ming Hsieh Department of Electrical Engineering  
University of Southern California  
sbruner@usc.edu

Yulong Ding  
Ming Hsieh Department of Electrical Engineering  
University of Southern California  
yulongdi@usc.edu

Angsagan Abdigazy  
Ming Hsieh Department of Electrical Engineering  
University of Southern California  
abdigazy@usc.edu

Team 5

Yulong Ding 8456815156 - Front-end / Back-end Python Scripting

Samuel Bruner 3198542984 - Register File / System Integration

Angsagan Abdigazy 2509824628 - ALU

1. Python Scripting and Results

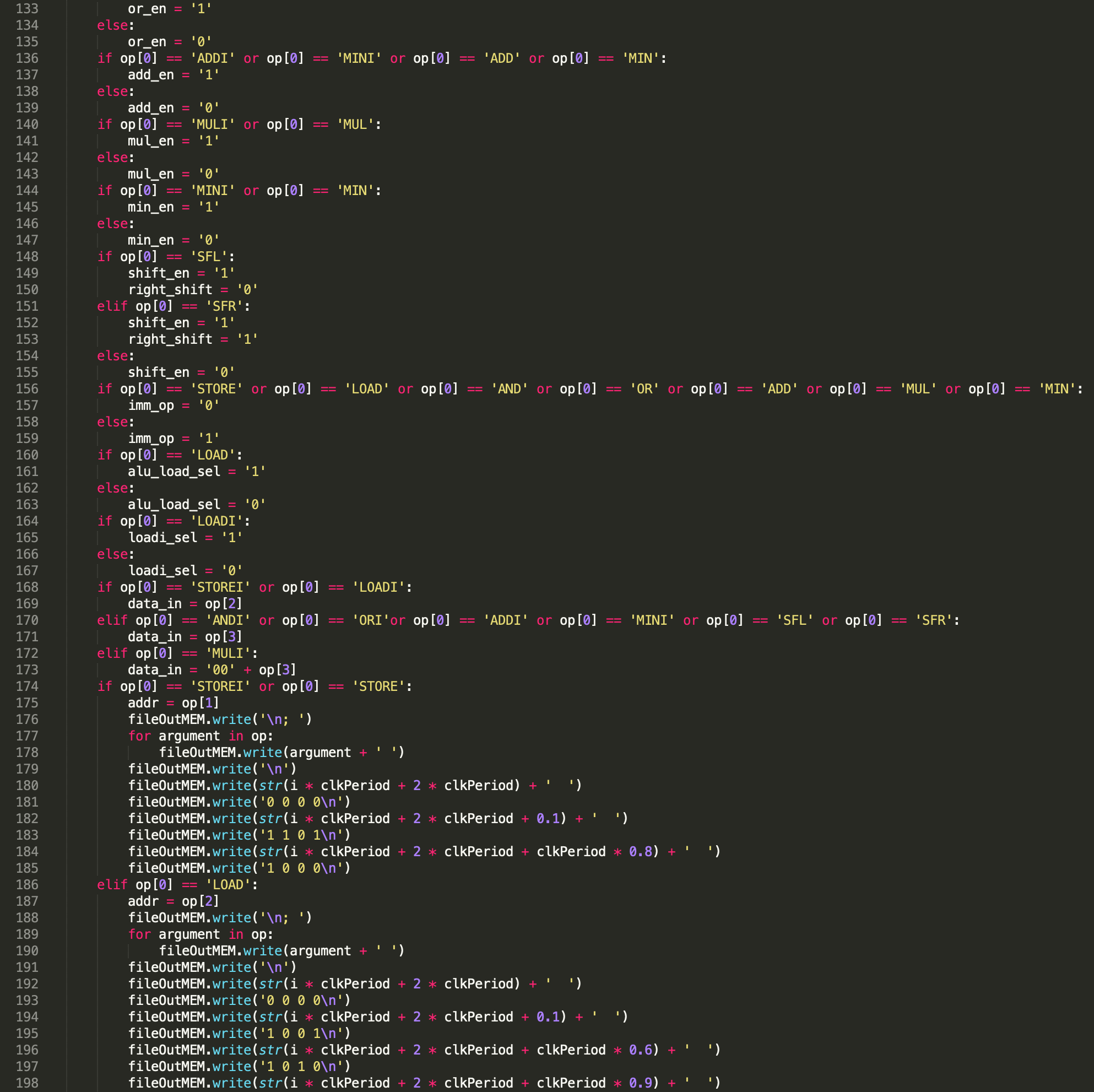
The Python script **project.py** combines the front-end instruction fetching/decoding and the back-end execution result verification in one script.

* **Front-end: Instruction Fetch and Decode**

The front-end of the script takes a text file **sample\_instructions.txt** as input, which contains the instruction sequence to be executed in an assembly-style syntax (as given on the DEN Course Final Project page), and produces vector files that assert appropriate control signals with proper timing for simulating the processor hardware design in Cadence. Read-after-write data dependencies that arise within the given instruction sequence when executed by the 5-stage pipelined processor are detected by the front-end, and appropriate numbers of “no operation” instructions (NOP) are then inserted between dependent instructions to resolve the detected hazard and ensure correct execution results. Because the SRAM memory block of the processor requires asynchronous control signals to perform load and store operations, the front-end also generates a separate vector file **mem.vec** to control precharging, reading from and writing to the SRAM. All other control signals, data and address inputs are generated in the main vector file **cpu.vec**.





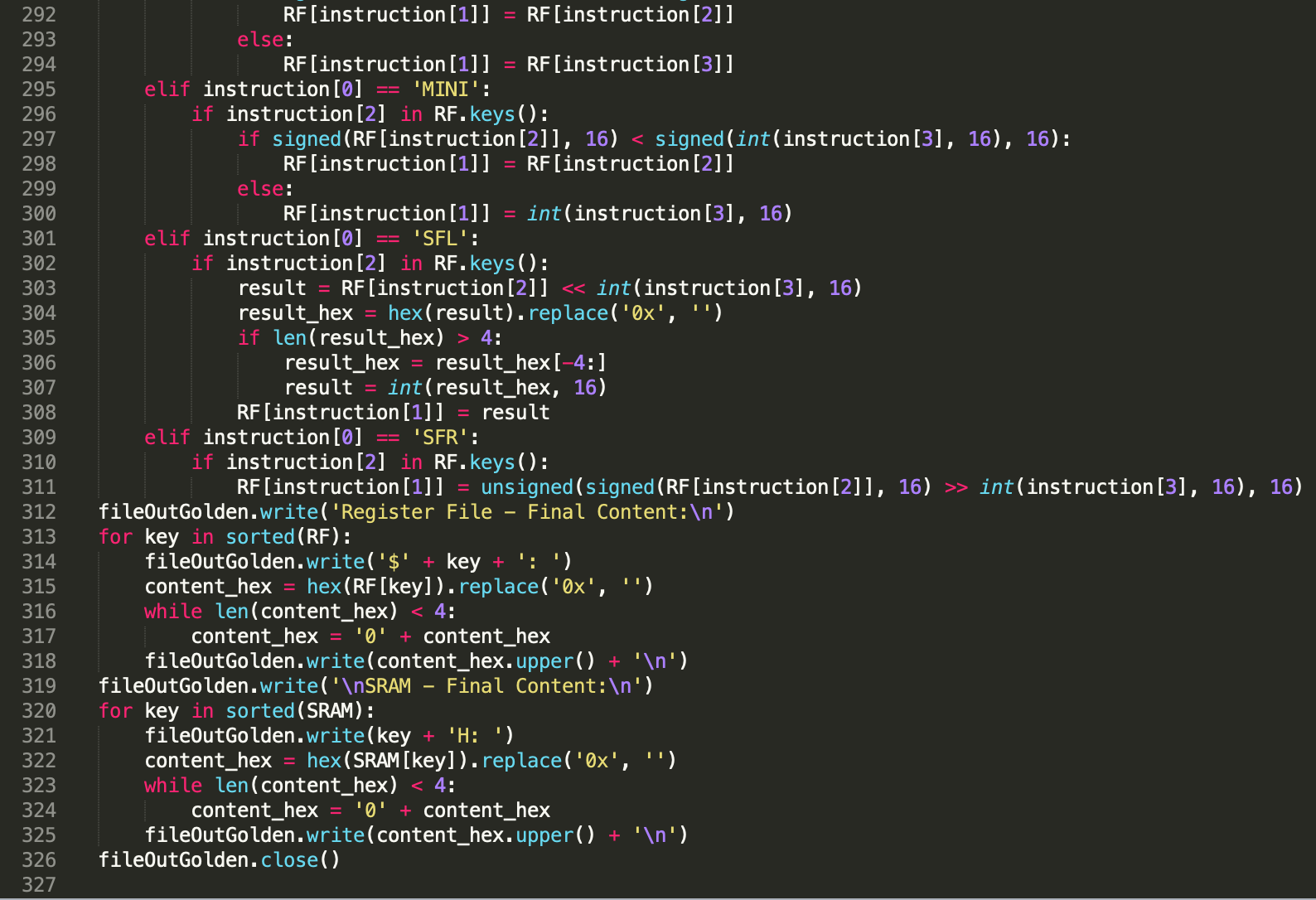




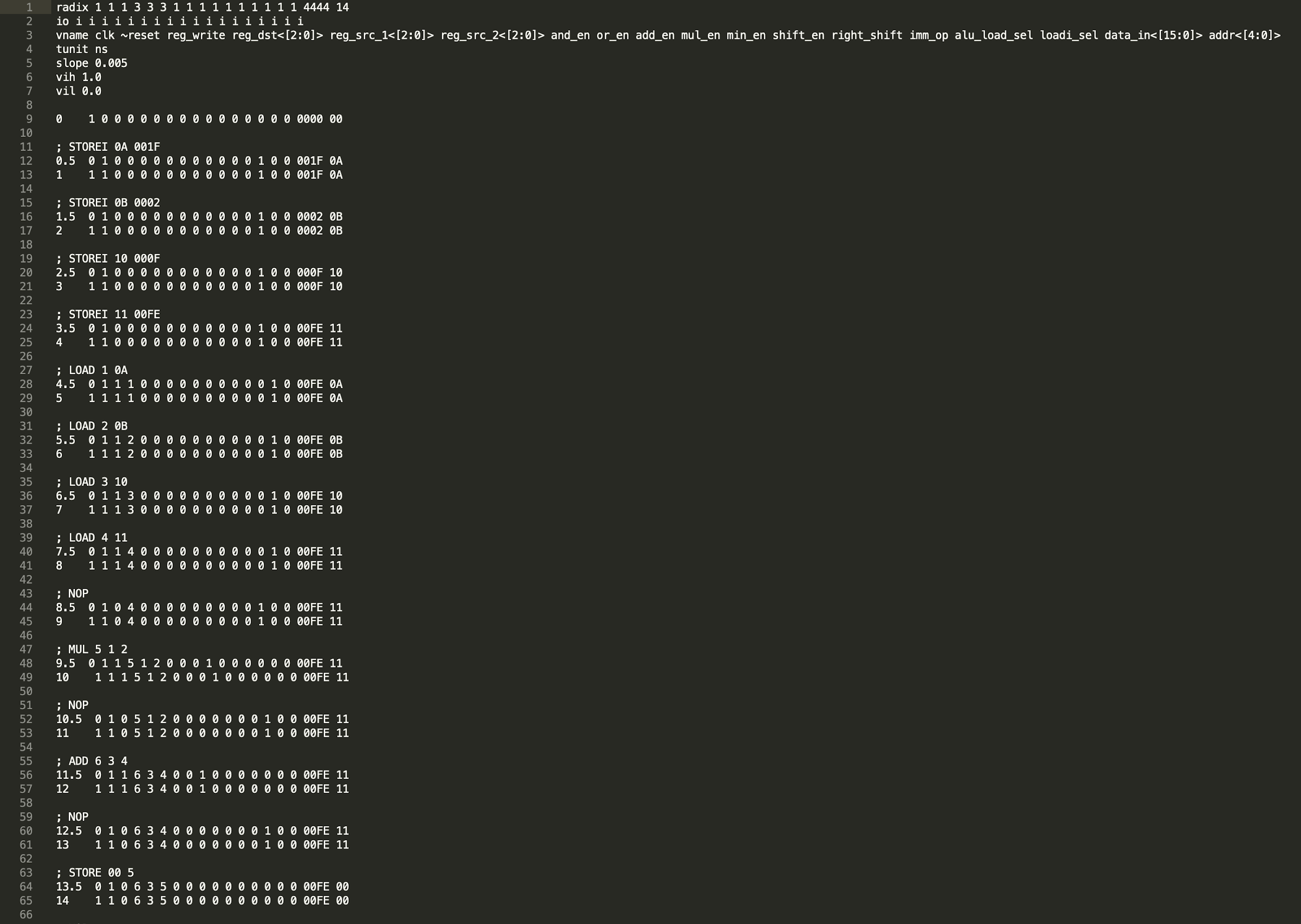
* **Back-end: Automated Result Verification**

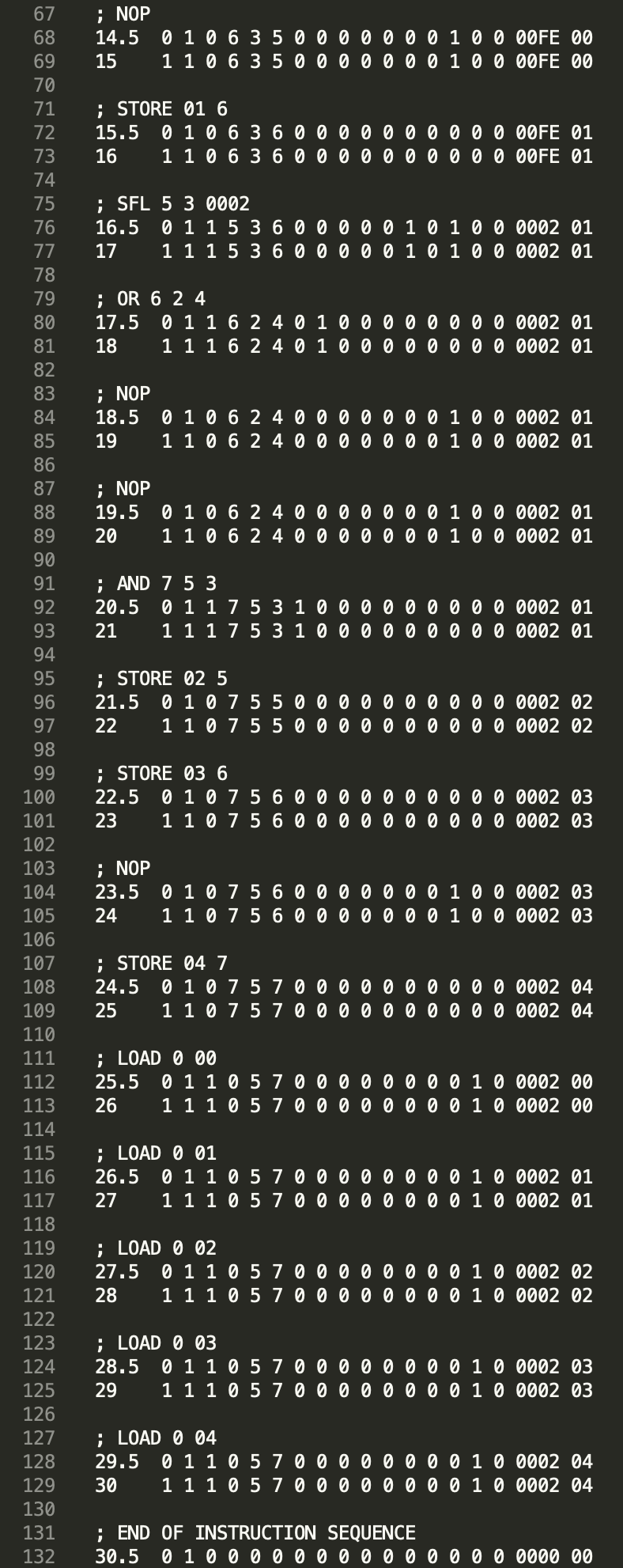
The back-end of the script simulates the execution result of the given sequence by the pipelined processor and produces a text file **golden\_results.txt** to show the final contents of the Register File and the SRAM after executing the entire sequence. By comparing the Cadence simulation results with the golden results generated by the back-end script, the correct functionality of the processor hardware design can be verified. Note that the back-end has been implemented to simulate exactly the behavior of the hardware, such as overflow in 16-bit signed addition, sign-extending the 10-bit product of the lower 5-bit signed multiplication, performing arithmetic (signed) bit shifts, etc.



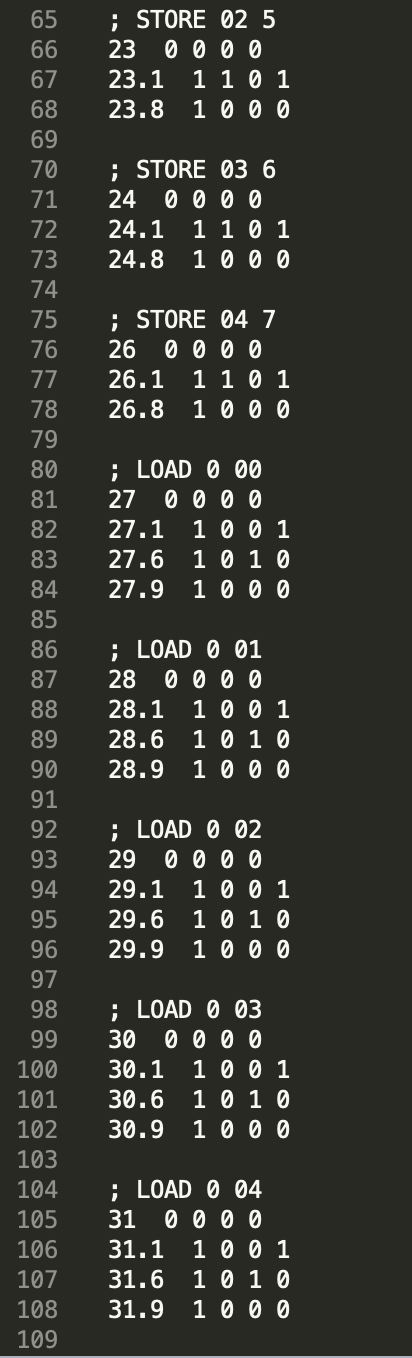
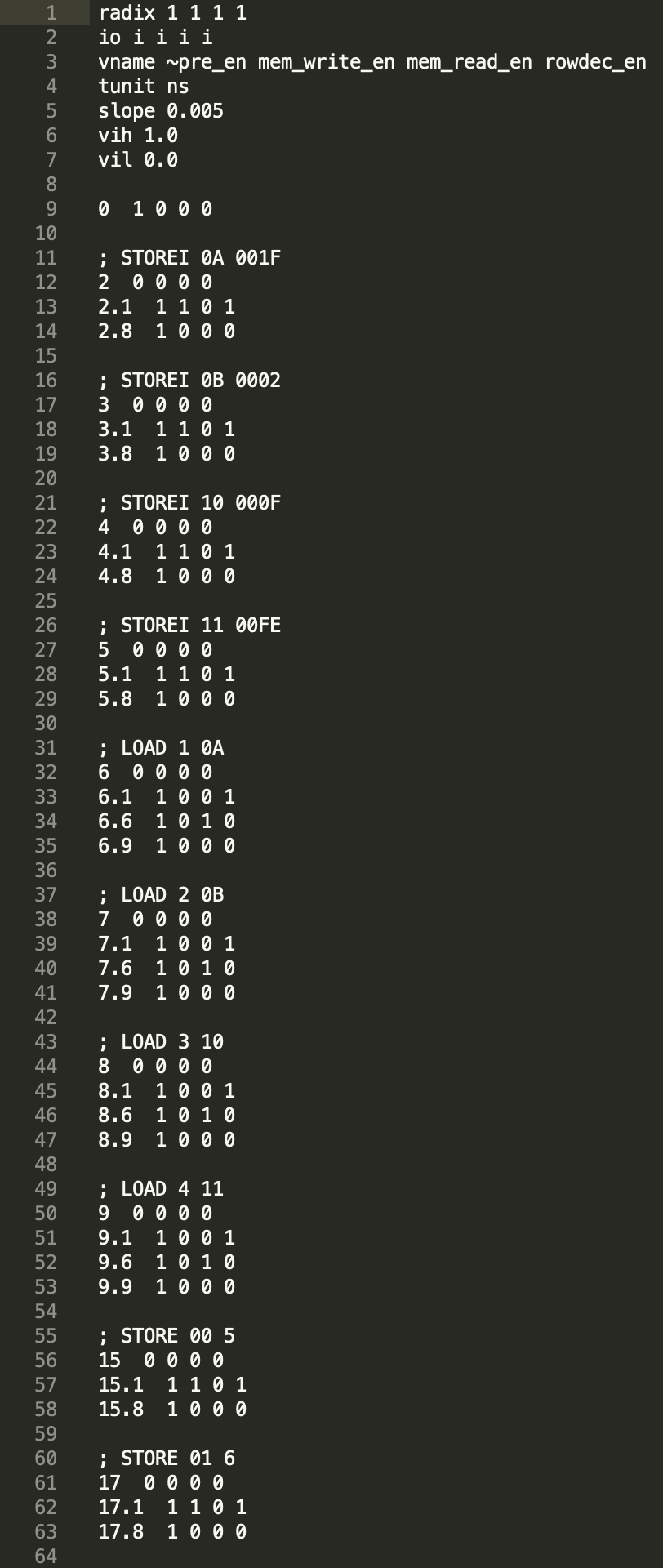


* Front-end generated Cadence vector file: **cpu.vec**

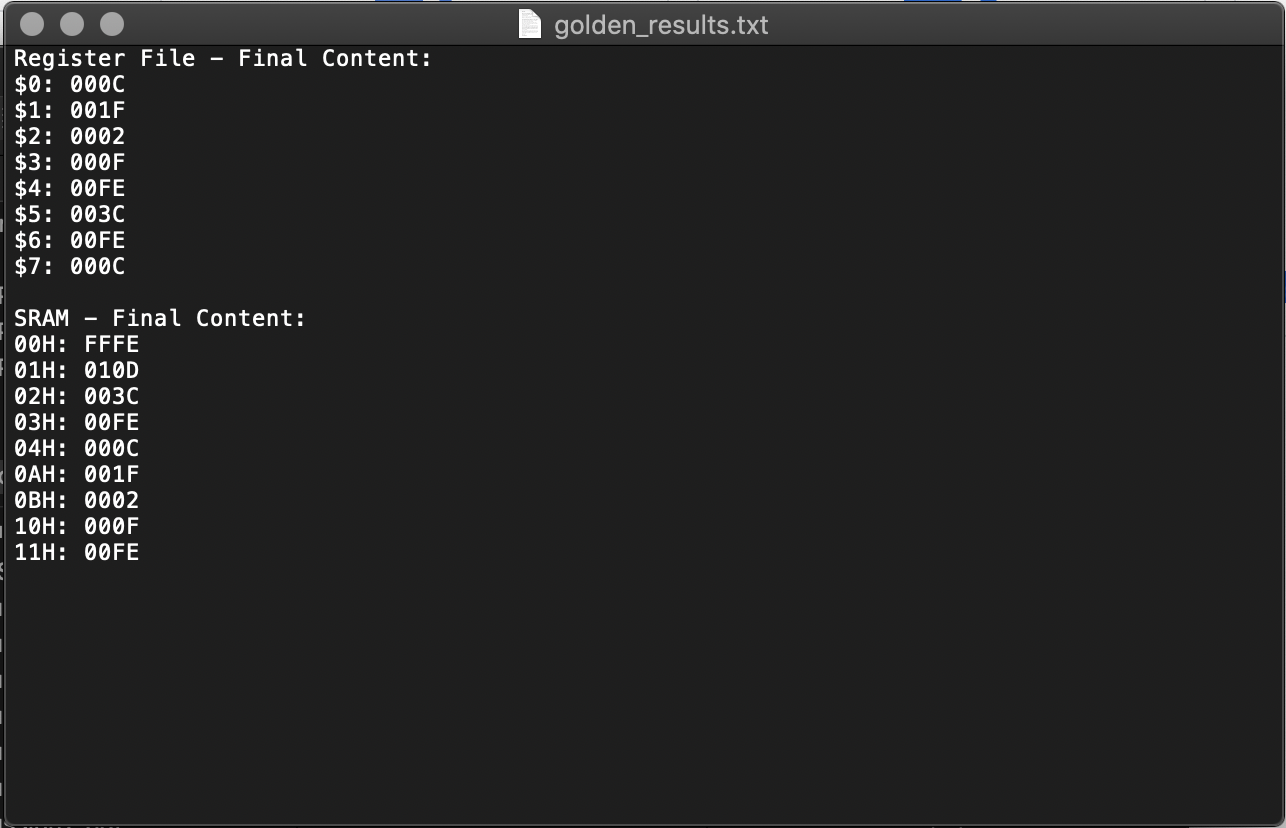




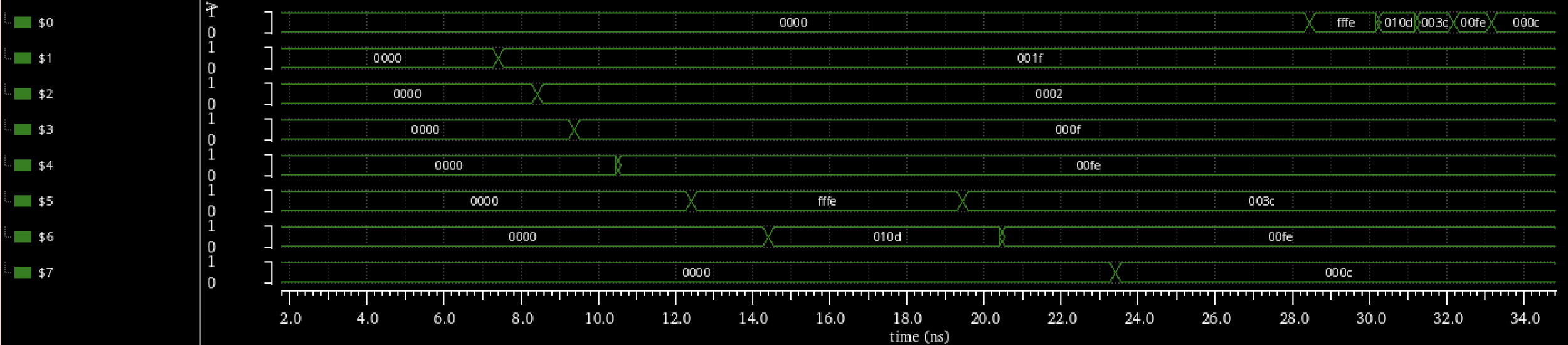
* Front-end generated Cadence vector file: **mem.vec**

****

* Back-end generated golden results: **golden\_results.txt**

****

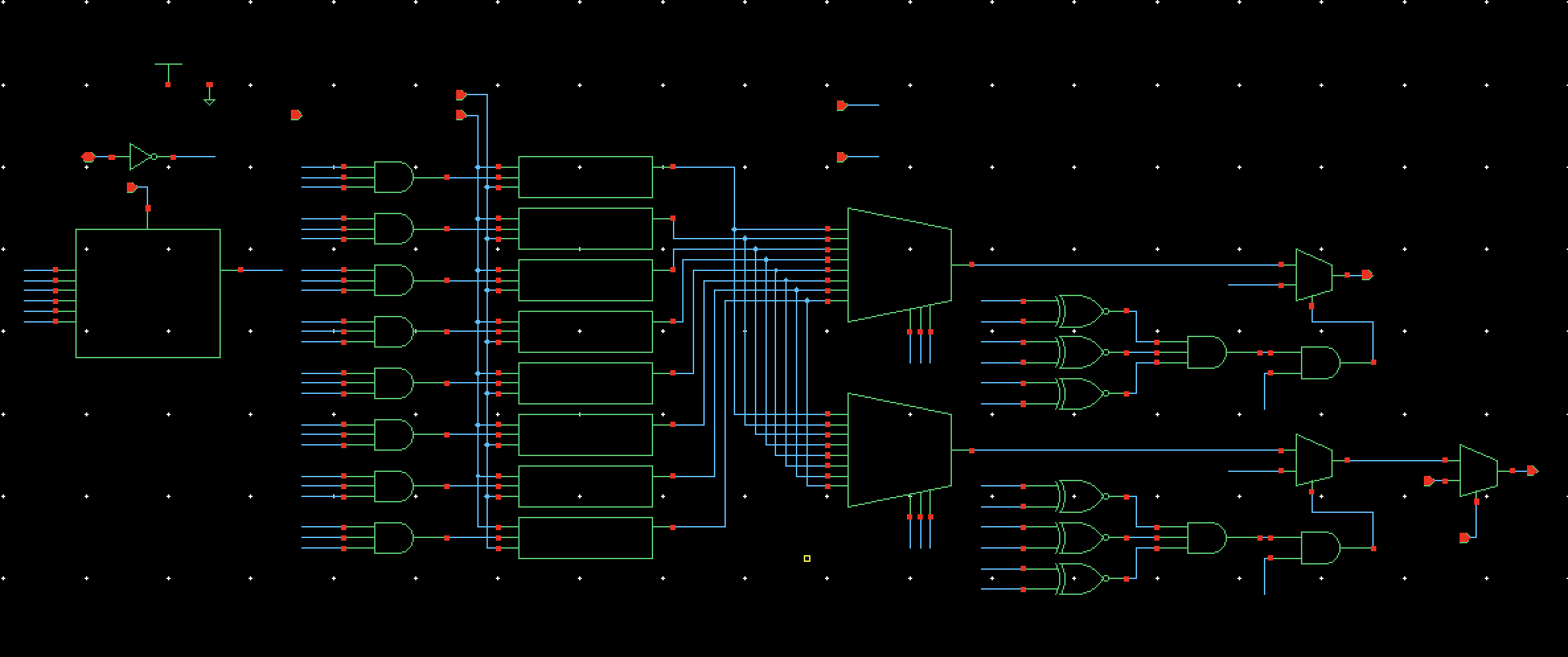
* Final content in Register File after Cadence simulation on hardware top-level schematic

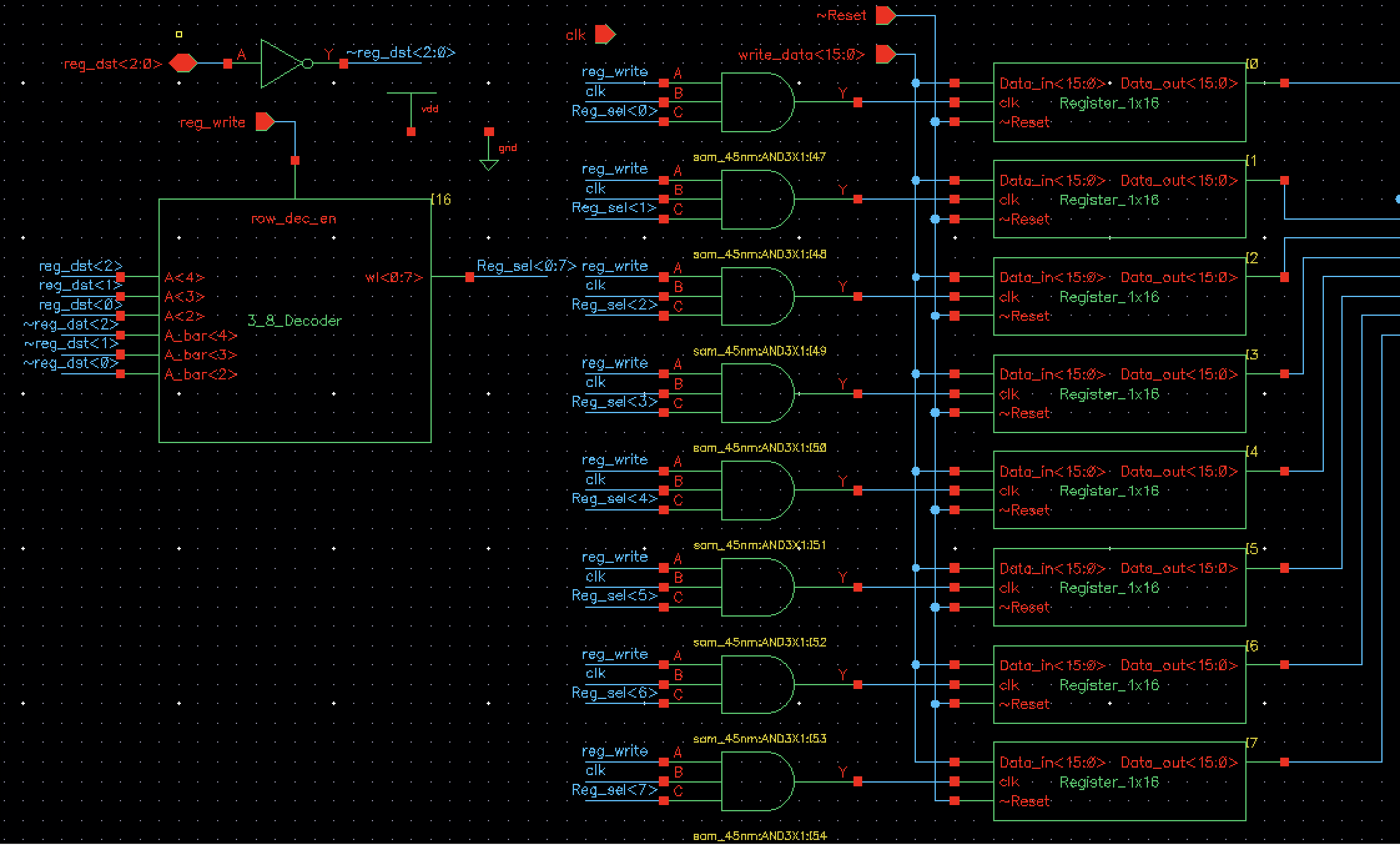


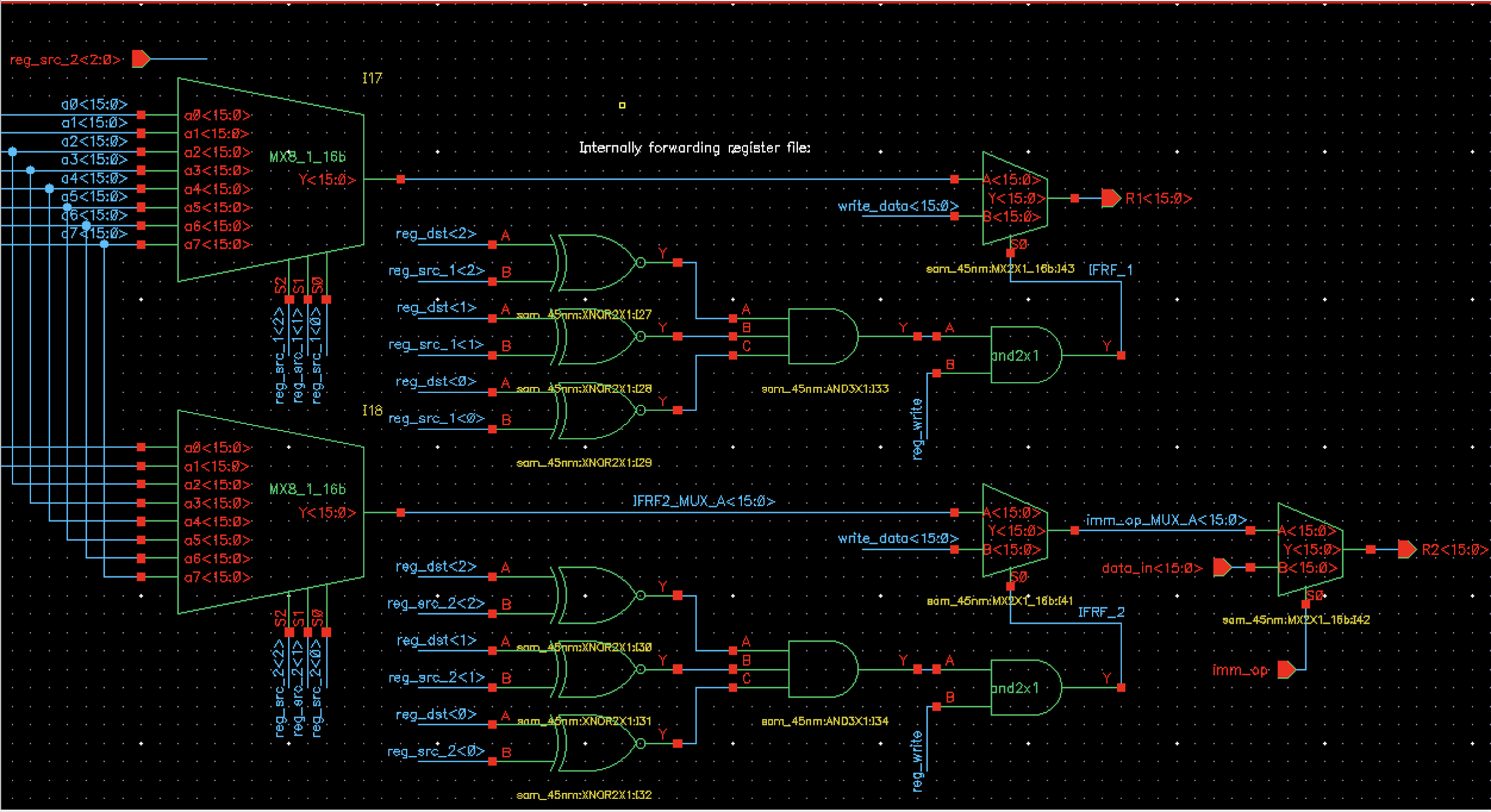
1. ID

* RF

According to the block diagram provided in the project description, the ID stage is divided into two parts. The front end python, as mentioned above, handles the decode portion while the register file is implemented in hardware. The following 3 schematics highlight the register file implementation. The register file memory is implemented using DFFs. There are 8 16 bit registers ($0-$7). The register file supports reading two registers simultaneously by multiplexing the register outputs and selecting a register with the respective source destination. Additionally, the register file supports an internally forwarding register file which is an optimization to save a NOP in phase 2. While this is implemented in HW, our current python script does not support its verification. This functions by comparing the destination register with the respective source register. If these match and the reg\_write signal is asserted, the RF will write this input data to the respective register while also outputting it to the respective RF output line. By default, there also exists a multiplexer on the register source 2 output to select between the register source 2 data or the immediate value for I-type instructions.

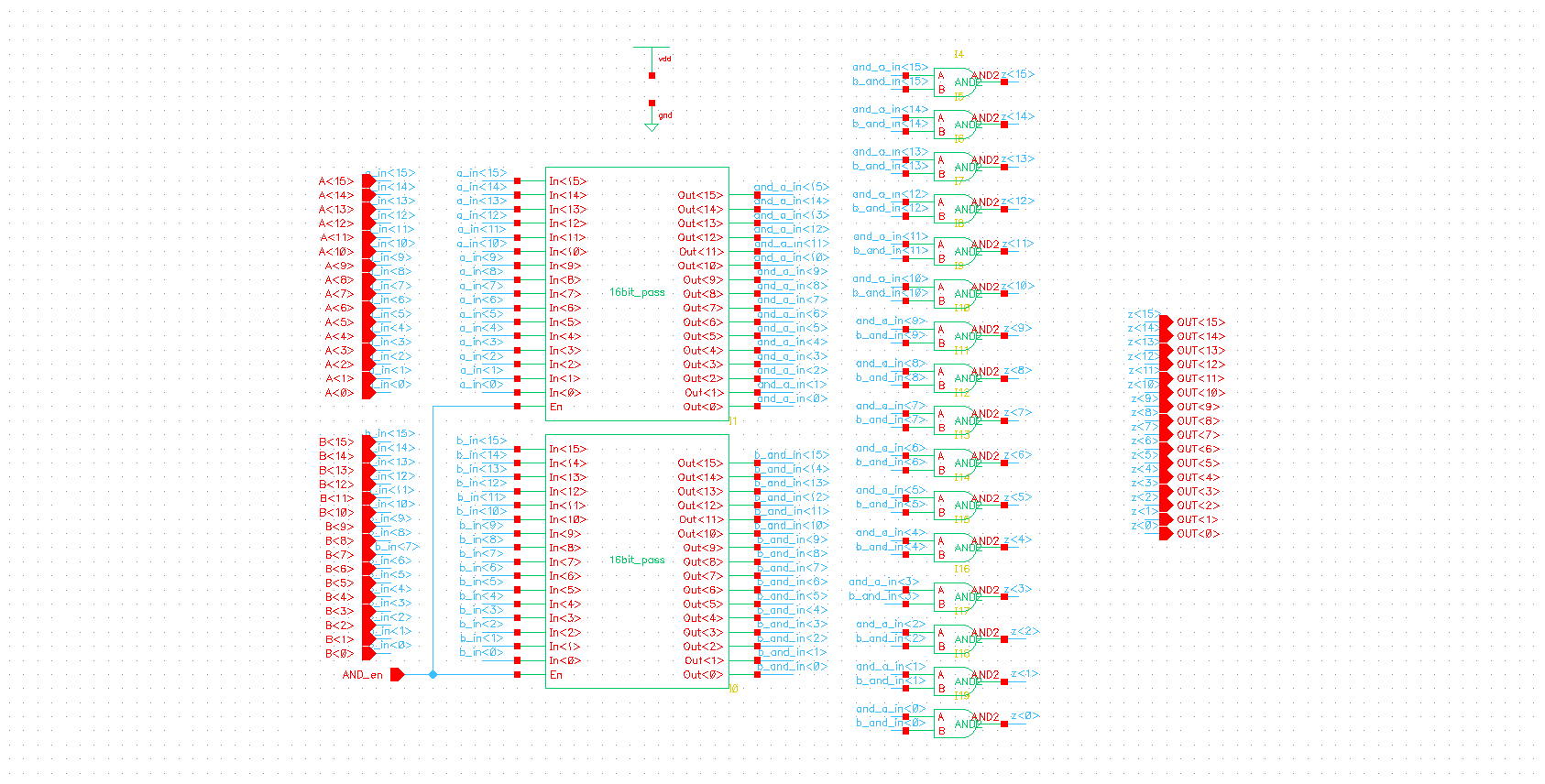




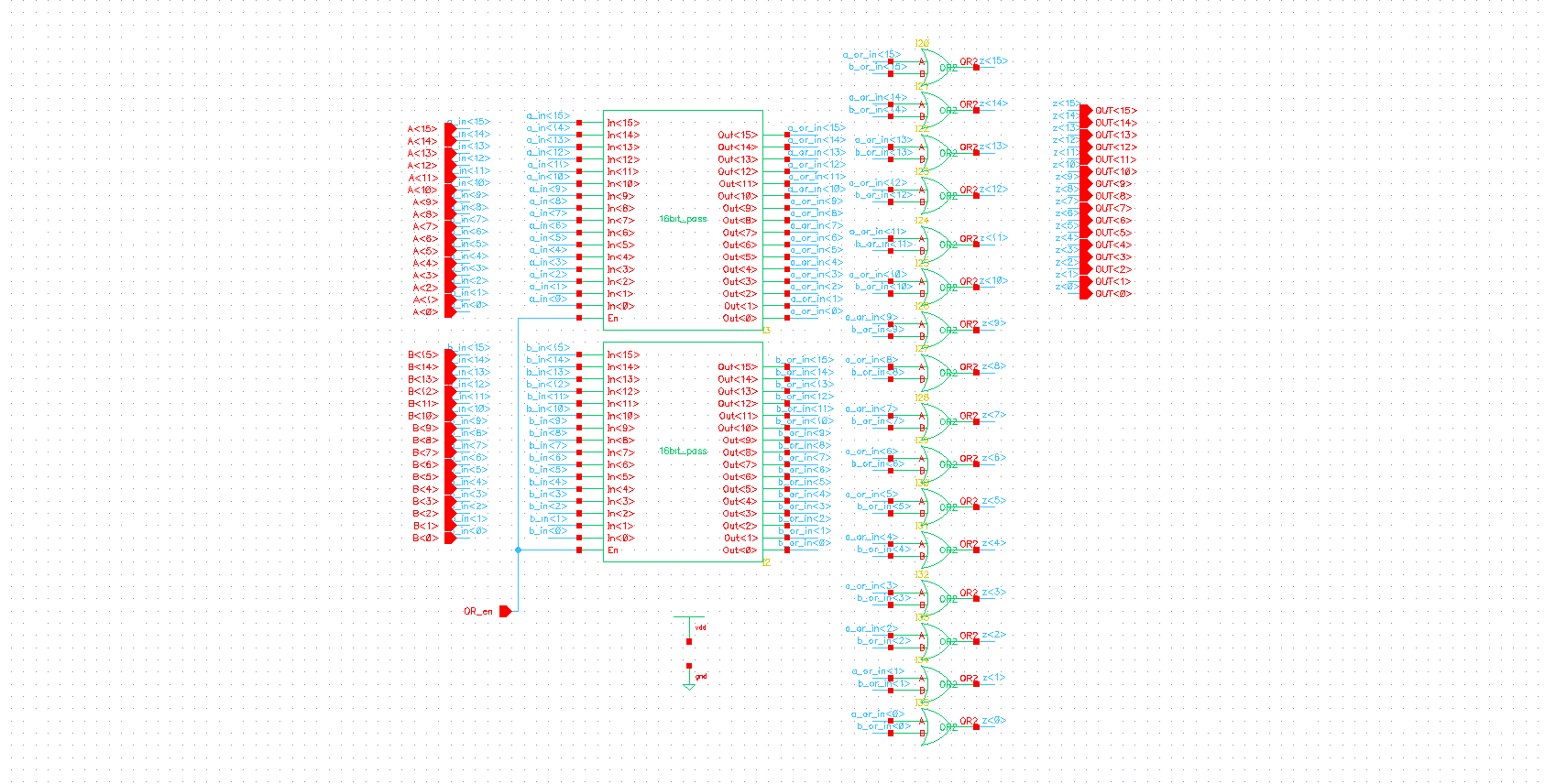


1. EX

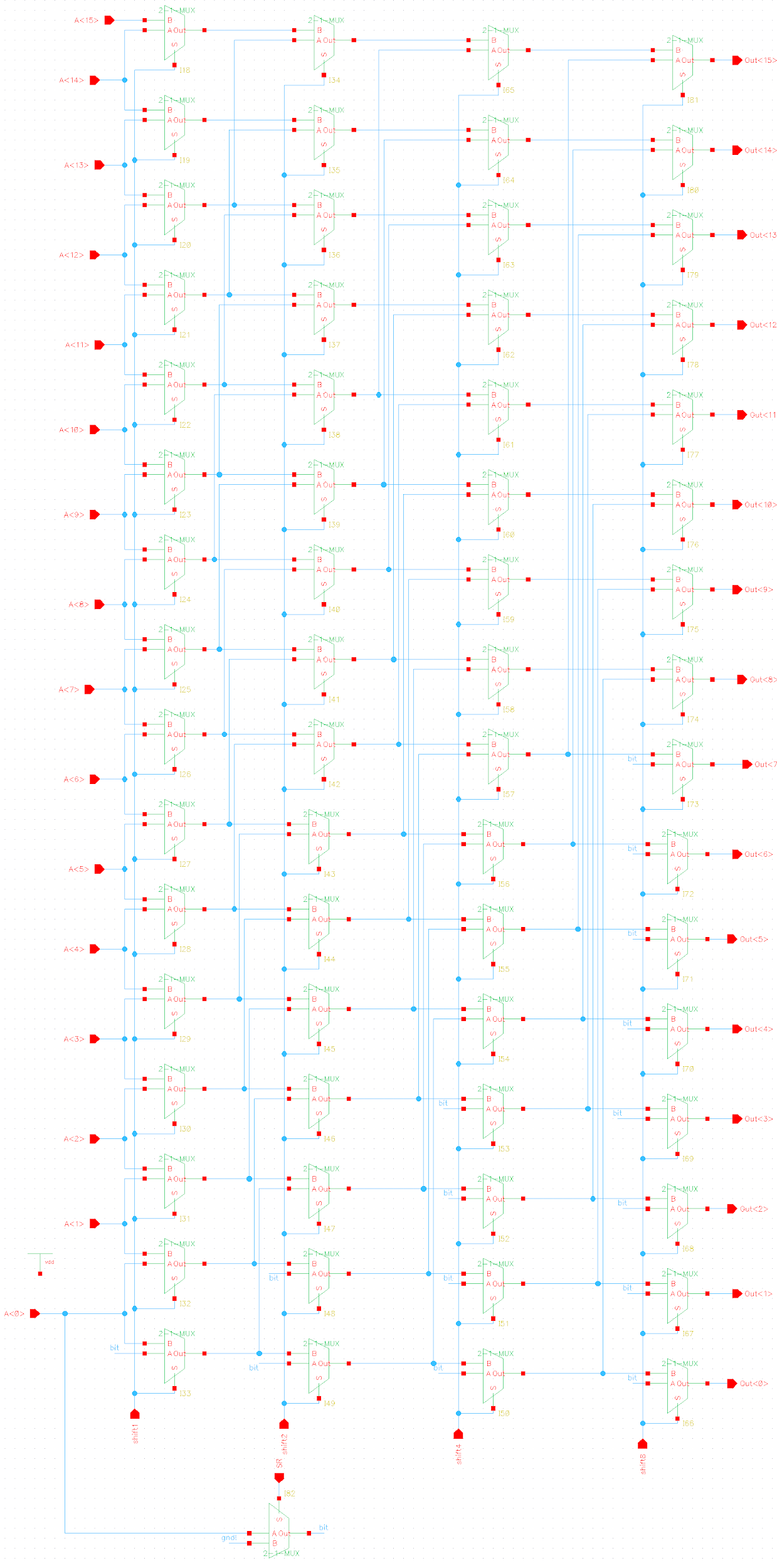
* Bitwise AND

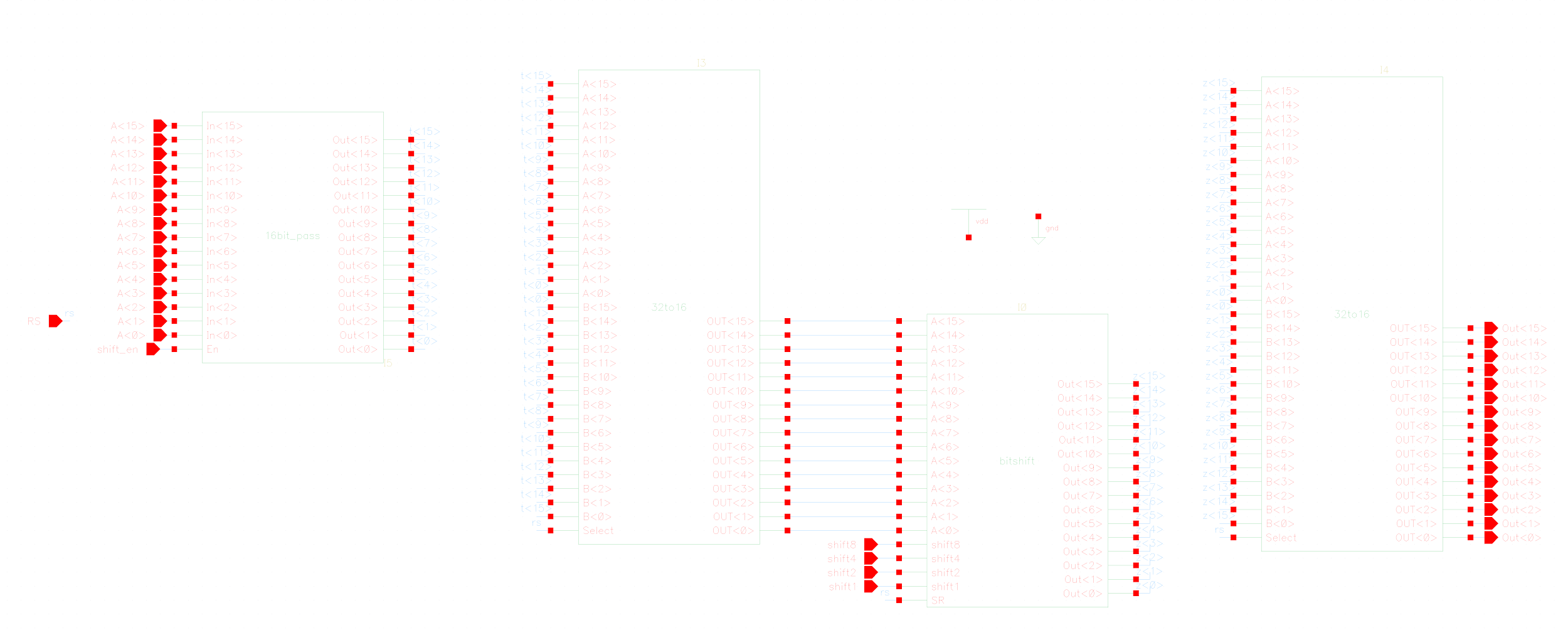


* Bitwise OR



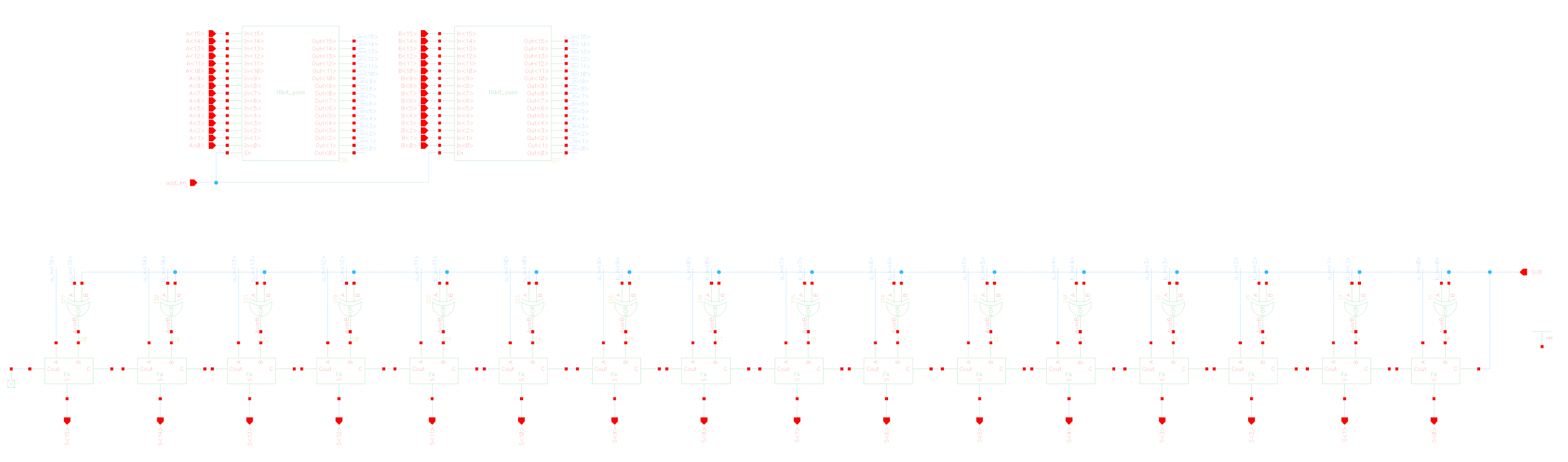
* SFL/SFR





* ADD/MIN

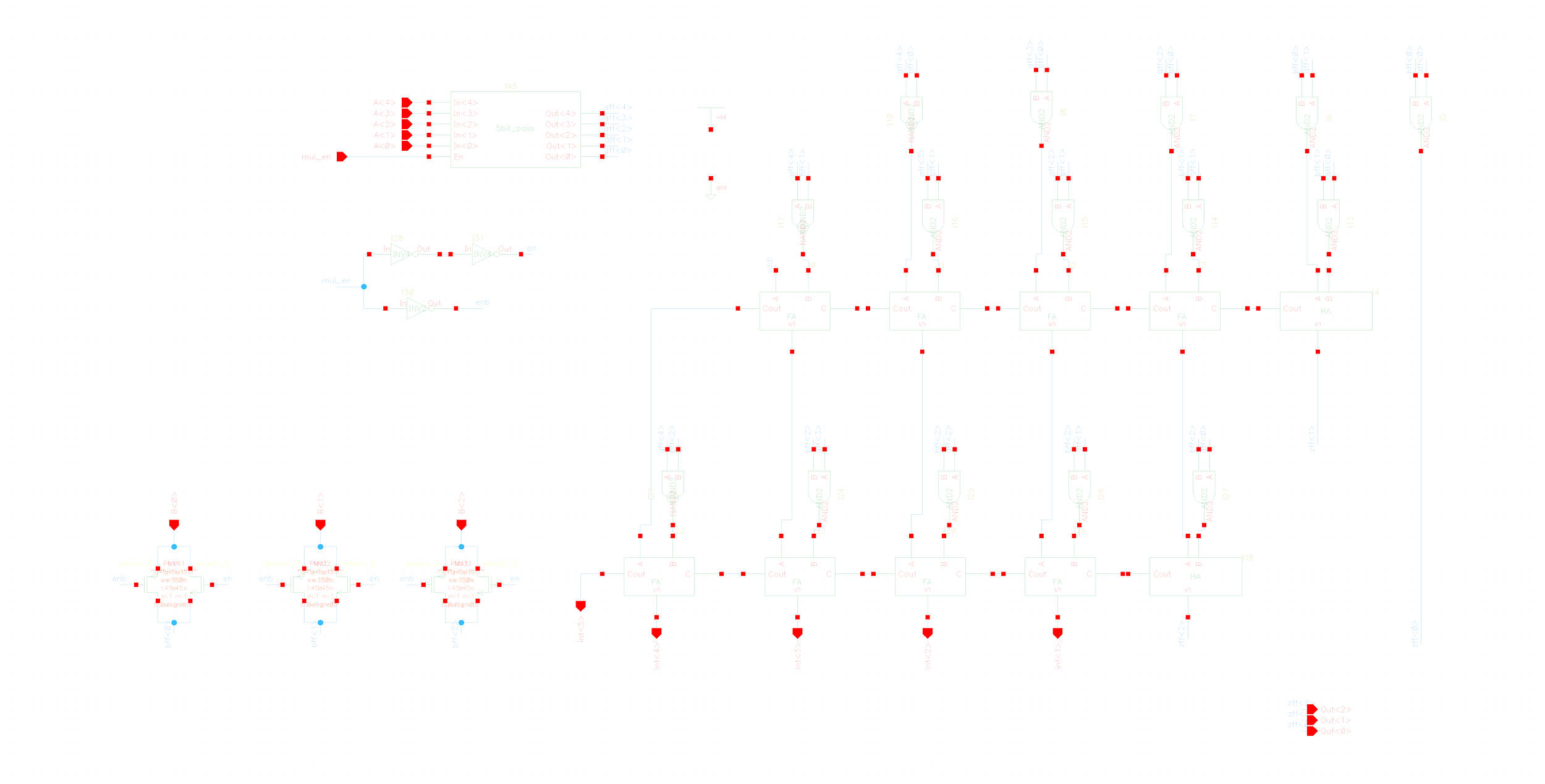
Depending on the value of the input carry, this unit can function as an adder or a subtractor. Subtractor is used to find the minimum value of two inputs. Then, the MSB is used to select that minimum value. Since the multiplier is the bottleneck for the clock frequency, using a Carry Propagate Adder/Subtractor topology does not cause issues with the CPU speed.



* MUL

We split the multiplier into two blocks - one of them resides in EX and another in MEM stage, respectively. This approach allows us to boost the clock frequency significantly because the propagation delay of the single multiplier limits the maximum clock frequency that ensures a proper operation of the CPU. Since 5x5 multiplication gives 10 bits at the output, our configuration takes care of remaining <15:10> bits by replicating the MSB of the original multiplication product.

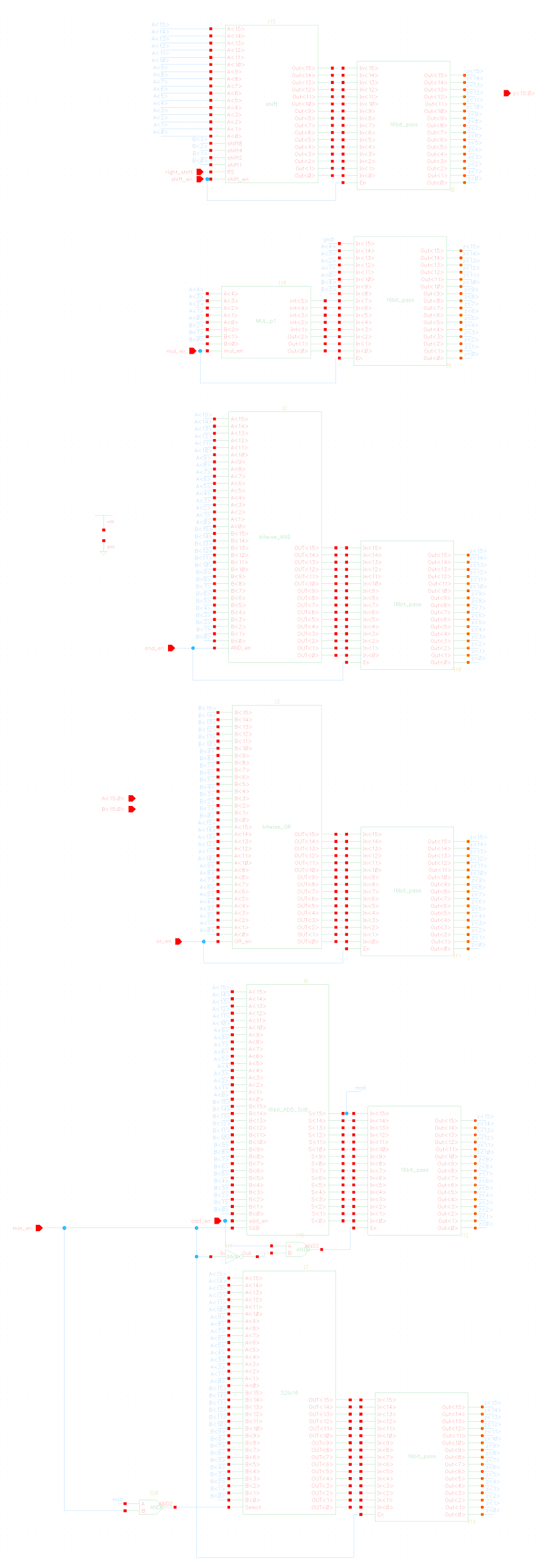
Part 1



Part 2



* Whole EX block

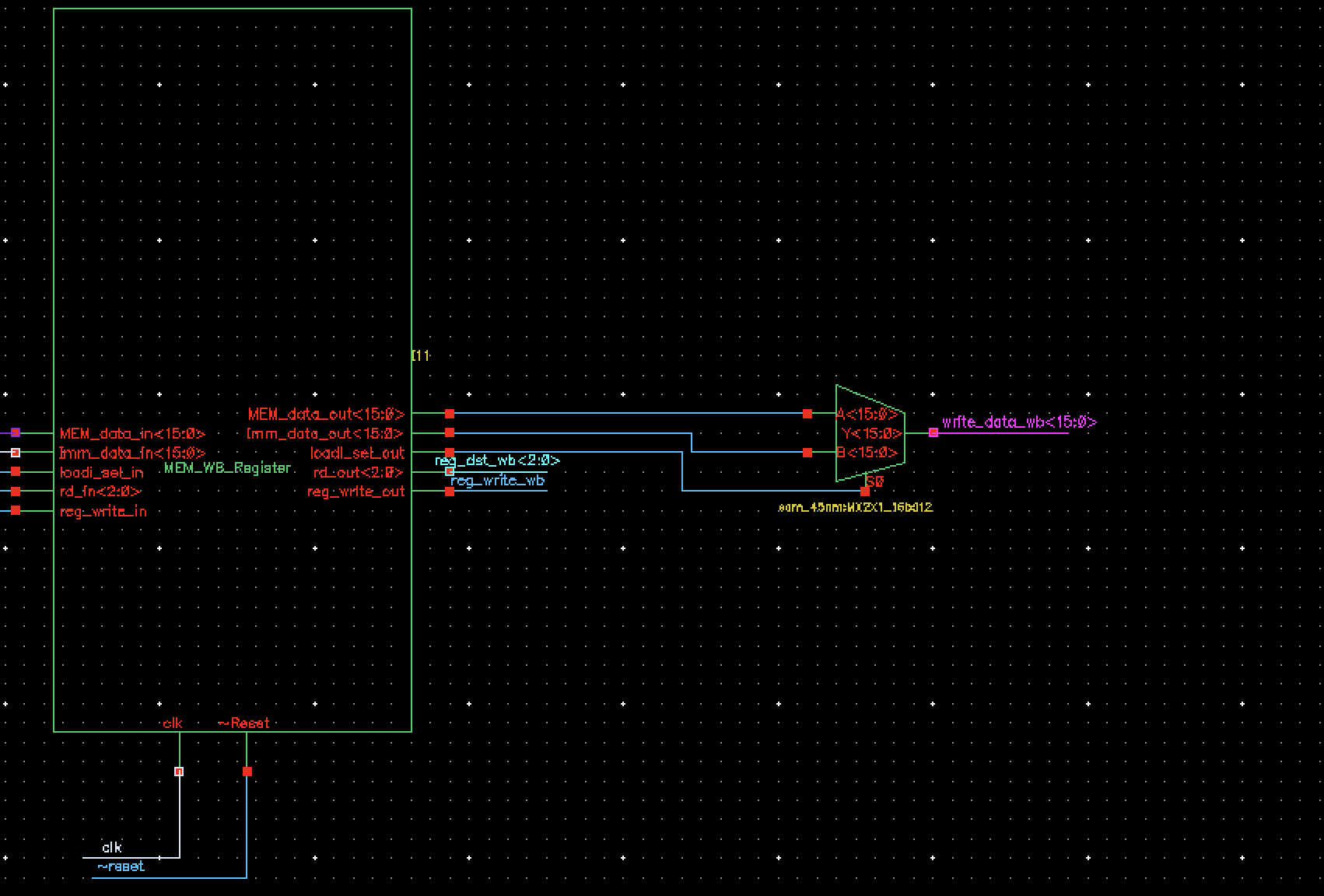


1. MEM

An SRAM was directly adopted from the Lab2 assignment.

1. WB

The write back stage is a simple implementation. It selects between the 16bit data output of the MEM stage (either ALU or SRAM output) or an immediate 16bit value and passes this value to the data input of the register file to be written at reg\_dst\_wb if reg\_write\_wb is high.



1. System Integration

The following screenshots show the high level system integration block with each major module as well as a zoomed in detailed view of each pipeline stage.

