SHA 256 VHDL

Alex Pegoraro

9 September 2024

Objective of the Project

The project consist in the realization of a VHDL device able to perform the mining operation of a bitcoin transaction, with a brute-force search. Its main component is "Block Routine", the unit responsible of the `SHA256` hash computation. The structure of the code is highly hierarchical, and it includes combinatorial components as well as sequential components and finite state machines.

Combinatorial Components

The backbone of the project is represented by the combinatorial components, that I implemented with the data flow paradigm. In primis I realized the bit-wise operations defined as follow in the `SHA256` specification:

- $Ch(x, y, z) = xy \oplus \overline{x}z$
- $Maj(x, y, z) = xy \oplus xz \oplus yz$
- $\Sigma_0(x) = RotR(x, 2) \oplus RotR(x, 13) \oplus RotR(x, 22)$
- $\Sigma_1(x) = RotR(x, 6) \oplus RotR(x, 11) \oplus RotR(x, 25)$
- $\sigma_0(x) = RotR(x, 7) \oplus RotR(x, 18) \oplus ShR(x, 3)$
- $\sigma_0(x) = RotR(x, 17) \oplus RotR(x, 19) \oplus ShR(x, 10)$

Then I moved to the arithmetic components, that have been made generic to handle input of different sizes:

- Ripple Carry Adder
- Hierarchical Comparator
- Barrel Shiftier

Since the circuit has may addition with 3 or more items, the adder is also equipped with a 3-to-2 compressor to speedup this possible bottleneck.

Sequential Components

The other face of the work regards the sequential components, implemented with processes. They include:

- Register File: It is the main memory source of the project. It is generic in the number of words as well as in the word size.
- Shift Register: to handle the constants that cyclically return each iteration I designed a register able to shift its content at word level. Apart from the serial input and output used in the shift, it also support parallel load and store.
- Counter: In the design of finite state machines there's often the need to count a specific number of iterations, for this reason I included a ripple counter.
- Debouncer: this component was not necessary to the miner itself, but since it is designed to be run on an FPGA a debouncer is required to interact with its noisy buttons.

Complex Components

In this section I will briefly discuss the elements that don't have a naive explanation, highlighting their role inside the device workflow.

Next Block Calculator

The `SHA256` algorithm operates on 512-bit blocks. Each block undertakes a routine of 64 cycles that doesn't work on the raw block, but on 64 words extracted from it. This component, has the role to compute them.

Following the documentation, the first 16 words are simply the split of the block into 16 32-bit words. Then, the remaining 48 words are obtained with the following formula:

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}$$

To realize this behaviour, the presence of a shift register initially load with the input block has been crucial. At each cycle, the first word of the register is popped out to be used by the datapath, while some specific words are extracted from the parallel out and sum to compute the next word to push in the register. This implementation avoid the necessity to precompute the words but generates them on the flight at each round, in parallel with the main routine, sparing a lot of time.

Register Update

This is the core of the algorithm. `SHA256` requires eight registers (named with the letters from a to h) to be updated each iteration. This combinatorial component wants as an input the value of the registers and two parameters, w_i and k_i, to return the value of the registers for the next cycle. The operation it performs are the following:

$$\begin{array}{llll} T_1 = h + \Sigma_1(e) + Ch(e,f,g) + k_i + w_i & & T_2 = \Sigma_0(a) + Maj(a,b,c) & \\ a = T_1 + T_2 & b = a & c = b & d = c \\ e = d + T_1 & f = e & g = f & h = g \end{array}$$

Since there are many additions, the compressors are put in a hierarchy.

Difficulty Decoder

Difficulty is the threshold below which the transaction hash must stay to be considered valid. This value is encoded in the transaction itself in a particular format that needs to be decompressed. This part of the code takes as input a 32-bit value, known as "Bits", and converts it into a 256-bit value that can be fed into the comparator.

The encoding is a sort of simplified floating point, where the first 8 bits of the "Bits" are the exponent, and the other 24 bits are padded into a 256-bit value. Then, the following operation occurs:

$$\text{difficulty} = ShL(\text{mantissa}, ShL(\text{exponent} - 3, 3))$$

Project Level

The previous components are arranged to form different datapaths, each one controlled by a Finite State Machine. The most important are "Block Routine", that execute the `SHA256` function on a single block, and "Bitcoin Miner", that effectuate the nonce research to validate the transaction. I also provided a top level to execute it on an FPGA.

Block Routine

This device works on `H-input`, the set of registers, and `m-block`, the input block. Its output `H-output` denotes the hash of the block given the starting registers. Figure 1 presents the datapath required to do so. In the plot the signals have different colors to highlight their role: black lines are internal signals, green ones are constants, blue represents input/output and red denotes control signals directed to the FSM.

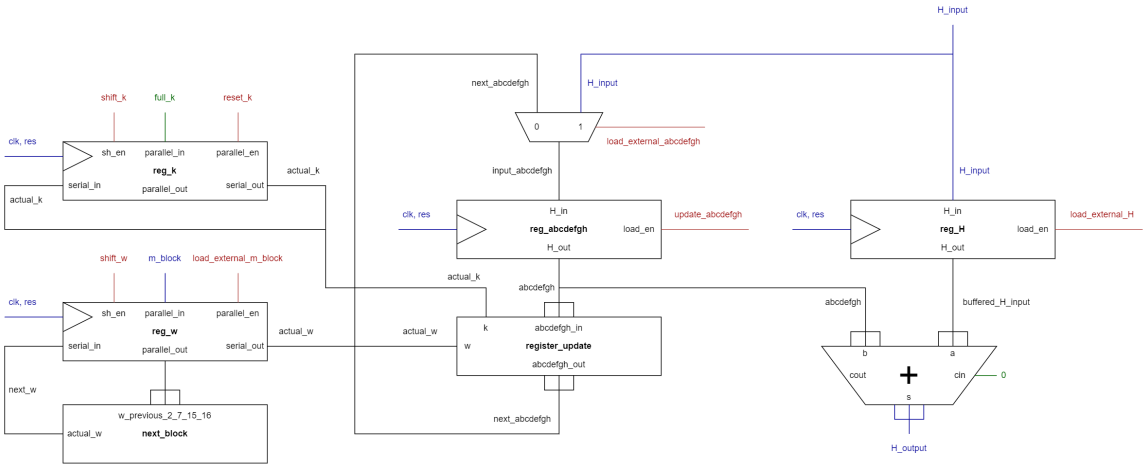

Figure 1: Block Routine Datapath

`m-block` is the slice of the input we are currently analyzing, and is considered a set of 16 words of 32 bits, for a total of 512 bits. `H-input` is composed by 8 words, that lead to 256 bits, and according to the specifications is initialized with a constant for the first block, and with the previous hash for

each subsequent iteration. The unit is agnostic to this and accepts any set of registers as input, it is a responsibility of the caller to remember the proper initialization and previous hashes.

The main routine is generated by the loop that connects `reg-abcdefgh` to `register-update`, causing the modification of `abcdefgh` registers. This process is enforced with the `next-block` unit, described in a previous section, responsible to extract the proper w_i for each iteration from m_{block}, and `reg-k`, a shift register that does a similar job looping throw a set of constants to provide k_i.

The flow is interrupted by the possibility to load `H-input` thanks to a multiplexer. It is also necessary to buffer this value into `reg-H`, since the final output will be the sum of `abcdefgh` with it. The finite state machine in Figure 2 controls this datapath through control signals, ensuring the proper execution of the algorithm.

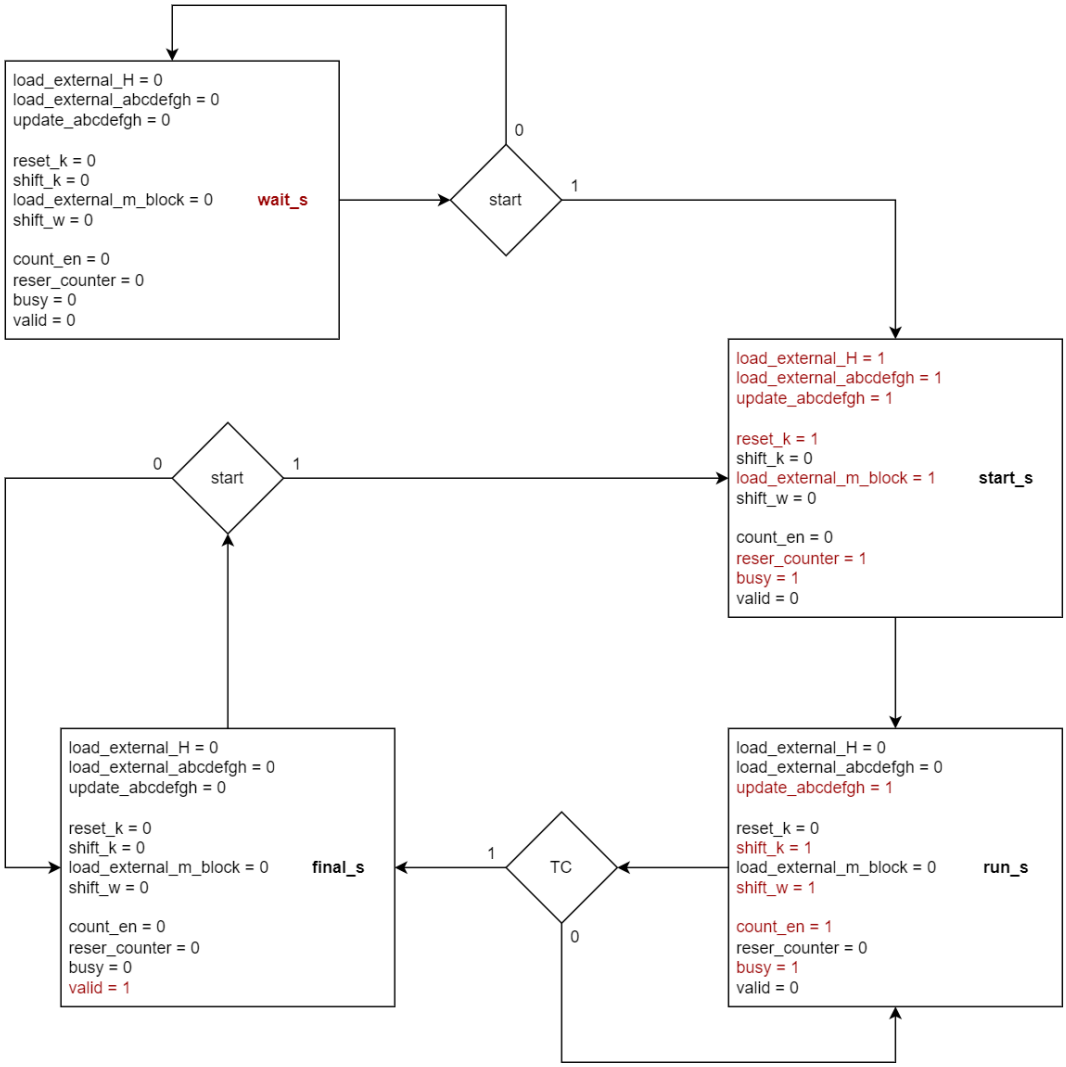

Figure 2: Block Routine FSM

It also has three input from the environment, namely a `start` signal, the `cancel` signal, and the `Terminal-Count` of an internal counter. It is not indicated in the figure, but `cancel` acts as a synchronous reset, when detected to one at any moment the FSM returns in `wait-state`. The only outputs directed to the caller rather than the datapath are `busy` and `valid`: the former indicates whether the machine is computing or not, the latter indicates if `H-output` is valid or garbage.

In the `wait-state`, all the registers are prevent form update to preserve the state. When `start` is set to one, an initialization phase occurs: `H-input` and `m-block` are load in the respective registers, `reg-k` is filled with the constants and the counter is reset.

Next, the code automatically move to the `run-state`, responsible of the iteration rounds. In this phase, `reg-abcdefgh` is repeatedly filled with the output of `register-update`, while w_i and k_i shift at every cycle and the counter is incremented.

When Terminal Count is reached, the finite state machine enters `final-state`, a frozen register situation equivalent to `wait-state`, with the notable exception that `valid` is put to one to indicate that the hash was computed successfully. When start will be one again, the device will move to `start-state` directly.

Bitcoin Miner

The full Bitcoin Miner Datapath is represented in Figure 3, with the same color encoding as before.

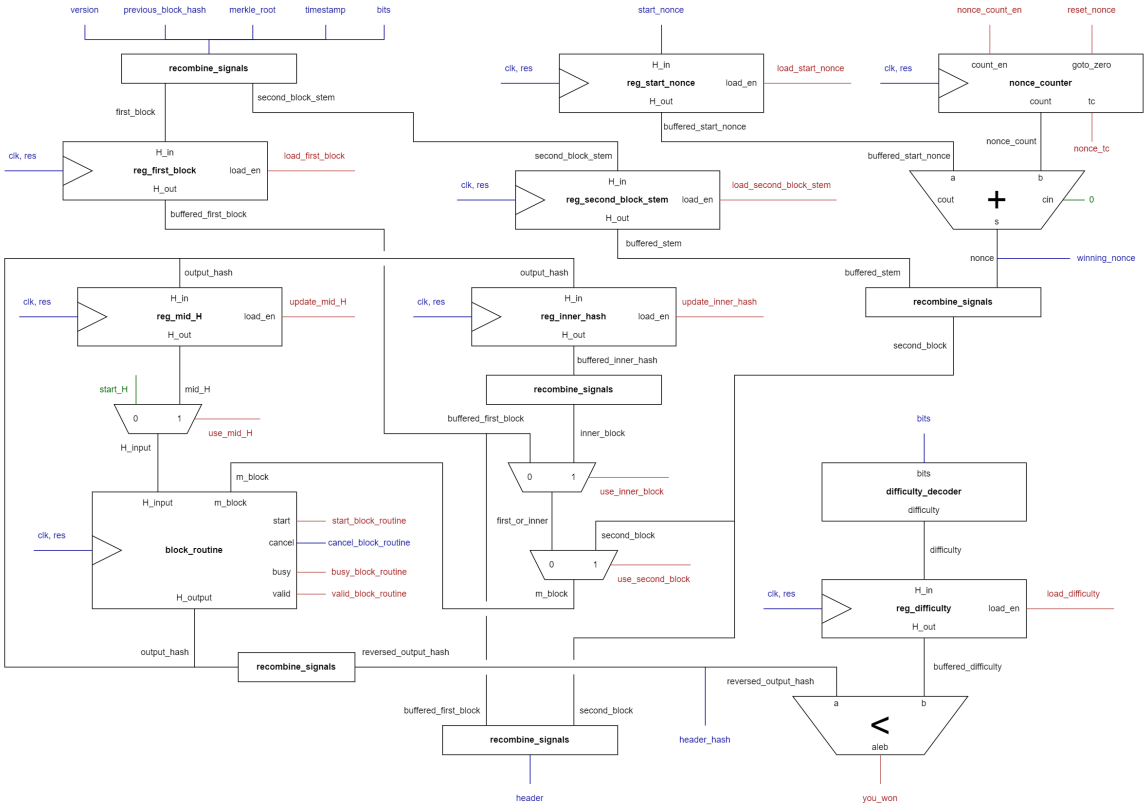

Figure 3: Bitcoin Miner Datapath

According to the Bitcoin specification, there are many elements that compose a transaction:

- Version: Identifies the version of the protocol.
- Previous Block Hash: The `SHA256` hash of the precious block in the blockchain.
- Merkle Root: The `SHA256` hash of the transaction items.
- Timestamp: A number that represents the time at which the transaction was generated.
- Bits: A pseudo floating point representation of the threshold the transaction hash must stay below.
- Nonce: An arbitrary number used to enlow the hash.

The Bitcoin miner aims to find a Nonce that makes the hash of the transaction lower than the threshold. This is done with a brute force search, but I gave the possibility to indicate a start nonce different from zero, to allow the users to perform a guess or to work with mining pools.

In my implementation these values are pure binary big endian numbers, but the header construction requires some signal recombination such as the reverse of the endiannity and the juxtaposition of signals. Table 1 present the proper header structure.

Field	Size	Endiannity
Version	32 bits	Little Endian
Previous Block Hash	256 bits	Little Endian
Merkle Root	256 bits	Little Endian
Timestamp	32 bits	Little Endian
Bits	32 bits	Little Endian
Nonce	32 bits	Little Endian

Table 1: Transaction Header structure

Furthermore, the `SHA256` algorithm requires to split the header into 512-bit blocks. This mean the first block will include Version, Previous Hash and a part of Merkle Root, while the second will contain remaining Merkle Root, Timestamp, Bits, Nonce, a trailing 1, a zero padding, the size of the header (640 bits, represented in 16 bits big endian).

These two blocks are stored in `register-first-block` and `register-second-block-stem`. The starting nonce is stored into a register as well, since it will be sum with the value of a counter to obtain the current nonce. At the same time, the Bits are expanded to the difficulty, that is stored into `register-difficulty`.

The bitcoin protocol does not operate on naive `SHA256` but on `double SHA256`, meaning the 256-bit output hash is expanded to a 256-bit block, appending a 1, some zeroes and its size as 16-bit integer, and this new block is processed as well. This is the reason why the inner hash is stored into `register-inner-hash`. To select the proper block used in `block-routine`, a cascade of multiplexers decides between the first block, the second block (with the starting nonce substituted with the current one) and the inner block.

Regarding the selection of `H-input`, the device can decide between a constant defined by the protocol, used for the first block, and the output of the previous block, buffered into `register-middle-H`. The output hash can finally be compared with the difficulty, but its endiannity needs to be reversed first. The result of this comparison is send to the FSM through the `you-won` signal. Regarding the

output of this component, the most important is `winning-nonce`, the value that made the hash low enough. I also decided to return the header itself and its hash.

On the other hand, Figure 4 and Figure 5 represent the finite state machine architecture and the value of control signal for each state respectively.

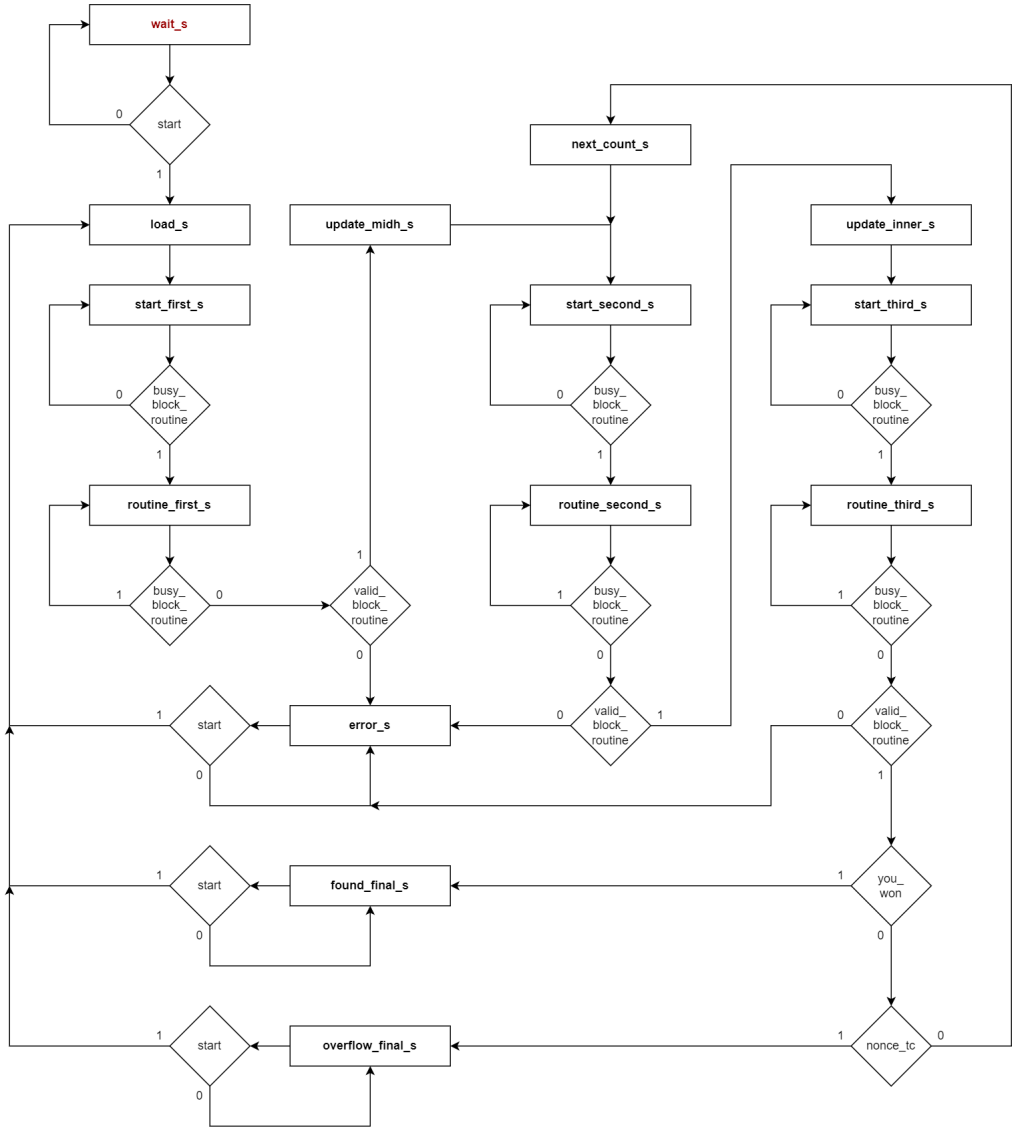

Figure 4: Bitcoin Miner FSM

Again, the situation begins with a `wait-state` that freezes all the registers, until `start` signal is set to one. This event triggers the passage to `load-state`, where all the inputs are rearranged and loaded into their respective registers. Immediately after this, `start-first-state` communicates `block-routine` to start the hash computation, indicating the constant as `H-input` and the first

block as `m-block`. It also checks the correct reception of the message controlling the `busy` signal. `routine-first-state` does nothing apart waiting the completion of hash computation.

After the production of a valid hash, `update-midh-state` stores that value into its register, and resets the nonce counter. `start-second-state` and `routine-second-state` are symmetrical with their "first" equivalents, with the exception that the second block is chosen for `m-block` and the previous hash for `H-input`.

`update-inner-state` stores the output into `register-inner-hash`, then `start-second-state` and `routine-second-state` execute `SHA256` another time, using the inner block and the constant registers.

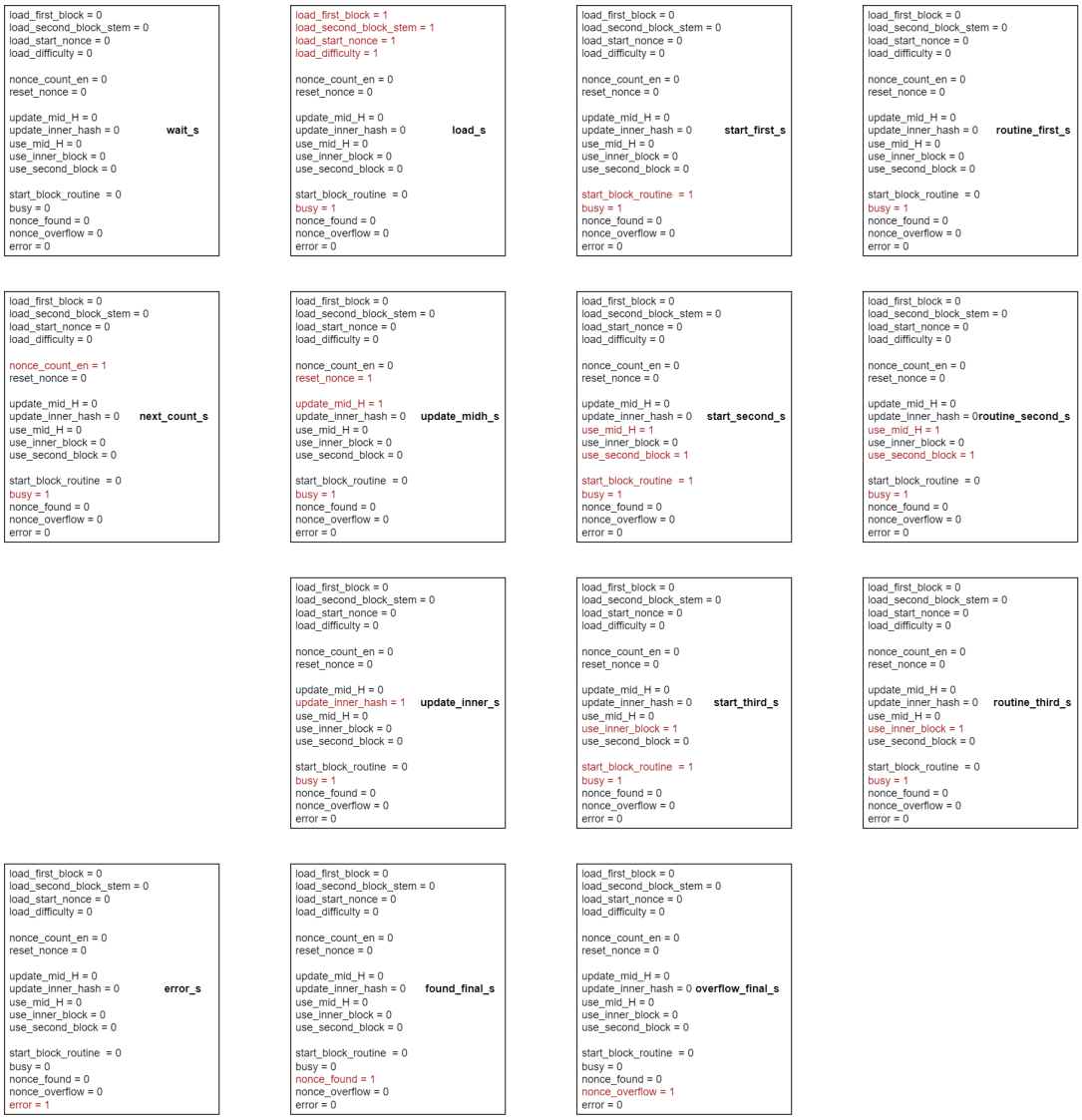

Figure 5: Control signals of Bitcoin Miner FSM

This time is not enough to control the output to be valid, but two control signals from the datapath are important: if `you-won` is one, the nonce have been found meaning the research can be stopped, and if `nonce-tc` is detected all the nonces have been tested, indicating again to stop the resource. These situation are accomplished with the transaction to states that communicate it to the external caller with output signals, and act as well as waiting states monitoring `start`.

If neither of the signal is set, another state is reached, namely `next-count-state`. Its only job is to increment the `nonce-counter` and go back to `start-second-state`. The reason why the hash of the first block is not recalculated for every nonce is that this block has nothing to do with it, since it is introduced with the second block. A last remark is that in all the three routine states, if the output is detected to be invalid the flow is deviated to `error-state`, another wait-like state that communicate this situation through output signals.

Top Level

It is quite difficult to provide the a 256-bit input to a feasible FPGA by hand, and display the resulting hash on the board is hard as well. Since I didn't have time to implement a communication interface to send and receive this data from a computer, I decided to hardcode some notable headers and compare their result with the expected one. In this component lies the necessity to include a debouncer and a 7 segment driver in the project. I didn't program these two units myself, but I used an implementation given me by prof. Roberto Passerone.

When the user presses a button on the FPGA, it triggers the load of a predefined header into the Bitcoin Miner, whose status can be observet through the lights. Led 16 encodes the output status: Red means the FSM reached the `error-state`, Blue indicates `overflow-final-state`, Green `found-final-state` and turned off signifies it is still computing. Led 17 is on the other hand connected with the `busy` signal, as it is Blue when it's one and Green when it's zero.

The output hash can be visualized with the switches and the 7-segments display. Since the board can plot up to 8 hexadecimal digits and the result is 256-bit, this means we need to split it into 8 parts. Switches from 0 to 7 allows the user to decide which part of the has to visualize. At the same time, the activation of switch 15 indicates the board to display the winning nonce, that is a 32-bit value visualizable as a whole.

Since the test-cases are static, the user knows their behaviour already, and it can compare the resulted computed by the machine with the expected winning nonces I reported on Table 2. To give an example, the test-case of the Up Button is reported in Table 3.

Case	Winning Nonce
Up Button	`0x33087548`
Right Button	`0x64089FFE`
Down Button	`0x9962E301`
Left Button	`0x97DC5290`

Table 2: Winning nonces of the test-cases

Credits

The code used for the 7-segment display driver and the button denouncer have been taken by the files *prof. Roberto Passerone* shared with his students in the *Logic Network* course.

Field	Hex Value
Version	0x00000002
Previous Block Hash	0x00000000 00000001 17c80378 b8da0e33 559b5997 f2ad55e2 f7d18ec1 975b9717
Merkle Root	0x871714dc bae6c819 3a2bb9b2 a69fe1c0 440399f3 8d94b3a0 f1b44727 5a29978a
Timestamp	0x53058b35
Bits	0x19015f53
Start Nonce	0x33080000
Winning Nonce	0x33087548

Table 3: Test-cases of Up Button