W11. Iterative architectures in Verilog language

Message scheduler for a cryptographic application

1. Laboratory objective

Implementing an iterative design for a cryptographic application

2. Theoretical Background

2.1 Message scheduling for Secure Hash Algorithm 2 architecture

- Presented in [FIPS15]
- Expands a 512-bit block delivered by the preprocessing step to a 2048-bit data packet

2.2.1 Message scheduling method

The 2048 bits are organized in 64 words of 32 bits each: $W_0, W_1, W_2, \dots, W_{62}, W_{63}$.

The first 16 words, $W_0, W_1, W_2, \dots, W_{14}, W_{15}$ make up the initial 512-bit block (W_0 being the most significant word).

The remaining 48 words are calculated with the following iterative formula:

$$W_t = \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16}$$

$$16 \le t < 64$$

The sum formulas on the previous page are done in the module $\mathbf{2^{32}}$ which means that any carry-out bit is ignored. The functions ***sigma0*** și ***sigma1*** retrieve a 32-bit word at the input and generate a new 32-bit word with the following relations:

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}$$

where by the ***ROTR*** function is meant right-hand bit rotation with ***n*** positions (***n*** - the power of the function), SHR meaning the movement of the bits with ***m*** positions (***m*** - the power of the function) and $\oplus$ signifies the EXOR operator.

2.2.2 Arrayed instantiation in Verilog language

Allows multiple instantiations using an arrayed instantiation **[AMI**]**. Consider, for example, the construction of 32 registers of 8 bits each. This aspect can be implemented using the code fragment below:

```
wire [255:0] reg_in ;

wire [255:0] reg_out ;

// ...

rgst # (

       .w(8)

) i_rgs [31:0] (

.clk(clk),

.rst_b(rst_b),

.d(reg_in),

.q(reg_out)

 // ...

);
```

If a signal ***i*** with a width of ***i*** bits is connected to a port with a width of ***i*** bits then the signal will be connected to the respective ports of all instances in the created vector (the 1-bit ***clk*** signal will be connected to the ***clk*** port, also on 1 bit).

If a signal with a width of ***k x i*** bits is connected to a port declared on ***i*** bits, the signal will be partitioned into sub-vectors of how many bits that will be connected to the respective ports of all ***k*** instances in the vector (reg_in [255: 248] is connected to the port of the most significant instance, i_rgs [31], reg_in [247: 240] is connected to the port of the most significant instance, i_rgs [30], etc).

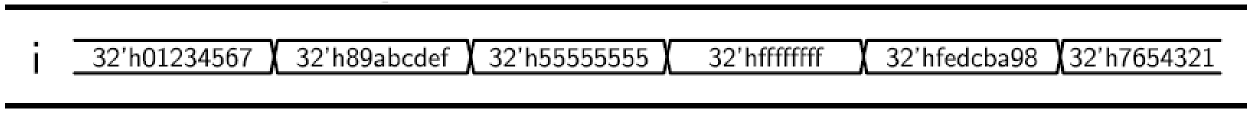
3. Proposed Exercises

E1. Build, using Verilog language, the operator $\boldsymbol{\sigma}_{\mathbf{0}}^{\{256\}}(*)$, defined in [FIPS15] (page 10 of the document).

Use the following interface:

```
module ssgm_0 (

input [31:0] i ; // initial word

output [31:0] o // generated word

);

assign o = // complete with code
```

Verify the ***ssgm_0*** operator with a testbench that generates the inputs as shown in the diagram below:

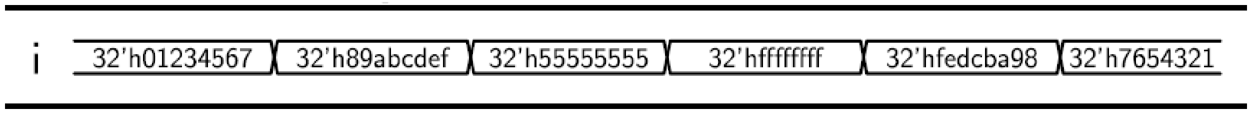


E2. Build, using the Verilog language, the operator $\boldsymbol{\sigma}_{\mathbf{1}}^{\{256\}}(*)$, defined in [FIPS15] (page 10 of the document).

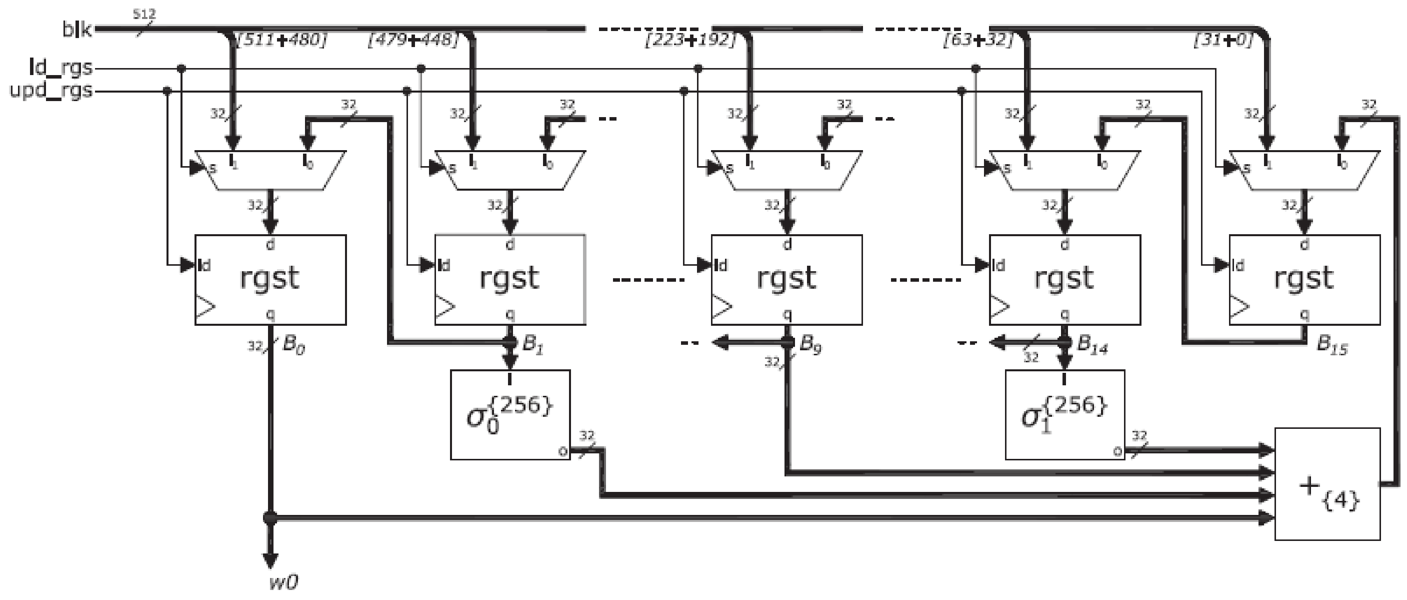
Use the interface on the next page:

```
module ssgm_1 (

input [31:0] i ; // initial word

output [31:0] o // generated word

);

assign o = // complete with code
```

Verify the ***ssgm_1*** operator with a testbench that generates the inputs as shown in the diagram below:



E3. Construct, using Verilog language, the iterative architecture that generates the expanded message scheduler block described below:



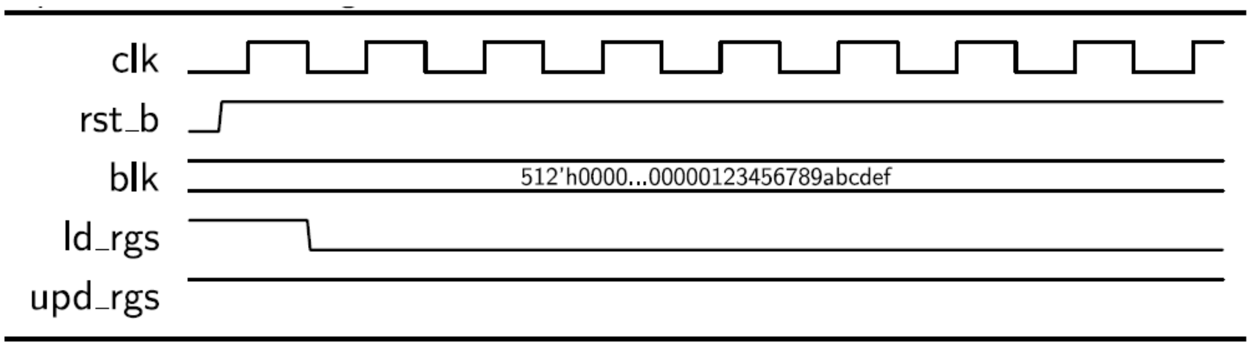
The ***ld_rgs*** input selects whenever the contents of the 16 registers are loaded with the information provided by the ***blk*** input at the beginning of message scheduling, or is generated according to the message scheduler update relation.

The ***upd_rgs*** input activates the updating of the contents of the registers.

In order to implement the design, use the following interface:

```
module msg_sch (

input clk ,

input rst_b ,

input [511:0] blk ,

input ld_rgs ,

input upd_rgs ,

output [31:0] w0

);
```

Verify the ***msg_sch*** architecture with a testbench that generates the inputs as shown in the diagram below:



4. References

[AMI]** Advanced Module Instantiation. [Online]. Available: http://www.eecs.umich.edu/courses/eecs470/OLD/w14/labs/lab6_ex/AMI.pdf

[FIPS15] National Institute of Standards and Technology, “FIPS PUB 180-4: Secure Hash Standard,” Gaithersburg, MD 20899-8900, USA, Tech. Rep., Aug. 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf