W10. Using generate blocks for multiple instances

Construction of multiple instantiations in Verilog

1. Laboratory objective

Implementing generate constructs using Verilog HDL

2. Theoretical background

2.1 Arrayed instantiation and generate constructs

Arrayed instantiation allows programmers to quickly write multiple instances of the same module.

Generate blocks offer a flexible way to create a large number of instances, with complex interconnections.

2.2.1 Arrayed instantiation

Makes it easy to quickly create multiple instances of the same module when all instances are connected to the same signals.

The Verilog implementation on the next page builds a BCD8421 to E3 converter for ***k***-digit numbers, using ***k*** instances of a 4-bit adder, called ***adder_4b***.

In general, the format of instantiation vectors adopts the following rule:

```
module-name instance_name [top-index: bottom index]

       ( .p(s), ... )
```

The Verilog code for the BCD8421 to E3 converter is shown below:

```
module bcde3_conv #(
parameter k = 4 ; // number of digits
)(
 input  [4*k-1:0] bcd, // input number bcd
 output [4*k-1:0] e3 // output number e3
);


   adder_4b cnv[k-1:0] (
        .x(bcd),
        .y(4’d3),
        .z(e3)
);
endmodule
```

Regarding the general format of instance vectors, if the signal width ***s*** is equal to the number of instances multiplied by the width of port ***p***, then s is equally partitioned in the number of bits of port ***p***, each partition being linked to one of the created instances. For the ***bcde3_conv*** example, the ***bcd*** input will be partitioned into 4-bit groups, each group is linked to one of the ***k*** instances (similar to the ***e3*** output).

If the width of signal ***s*** is equal to the width of port ***p***, then the entire signal ***s*** is bound to all instances. For the example ***bcde3_conv***, the value 3, represented by 4 bits, which will be added to each digit ***BCD8421***, has the same width as the ***y*** port of the adder and, therefore, will be connected to all ***k*** instances.

2.2.2 Generate Blocks

The ***for loop*** inside the generate block:

- Uses a ***genvar*** variable as an index
- Has the content contained in a ***begin*** … ***end*** block with a dedicated name

The previous converter from BCD8421 to E3 is rewritten as below:

```
module bcde3_conv #(

parameter k = 4 ; // number of digits

)(

 input  [4*k-1:0] bcd, // input number bcd

 output [4*k-1:0] e3 // output number e3

);

  generate

       genvar i;

              for (i = 0 ; i < k ; i = i + 1) begin: vect

               adder_4b uconv ( .x(bcd[i*4+3:i*4]),

                                .y(4’d3),

                                .z(e3[i*4+3:i*4])

);

              end

  endgenerate

endmodule
```

The ***begin ... end*** block starts with the line where the for cycle was defined. The keyword ***begin*** is followed by an identifier (in this case ***vect***, but any valid Verilog identifier can be used).

In the named block, only one instance, called uconv, is built in each iteration.

The port association uses the ***i*** index to group 4 consecutive bits in the bcd input and 4 bits in the e3 output. 4-bit groups are made using the part-select expression [i * 4 + 3: i * 4].

The value 3, represented by 4 bits, is connected to the ***y*** port of the adders.

2.2 Laboratory Task

The preprocessing unit of a cryptographic application

Exercise: Build the data path for the input preprocessing unit of a 256-bit Secure Hash Algorithm 2 (SHA-2) design.

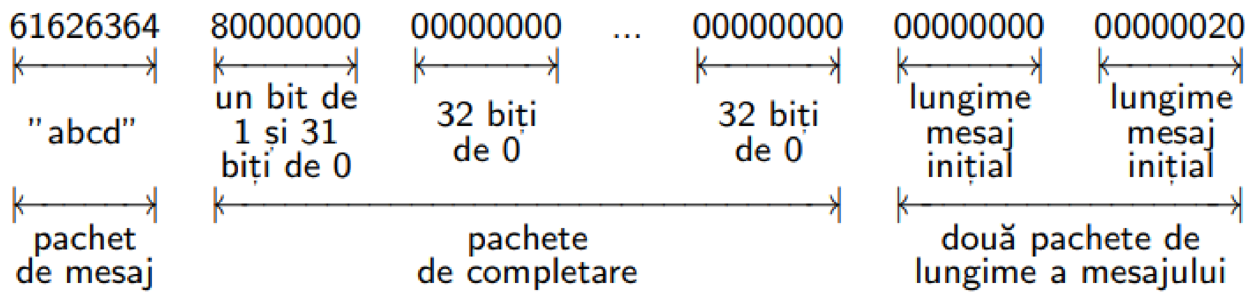
Solution: The 256-bit SHA-2 input preprocessing unit implements the preprocessing phase, described in [FIPS15] (section 5.1.1) consisting of completing and dividing the message.

Completing the message: For an initial message of length l, the completion adds a bit of 1 and k bits of 0, so that l + 1 + k = 448 (mode 512). At the end, the length of the initial message is attached as a 64-bit unsigned number.

Dividing the message: The initial message with the completed bits above is divided into 512-bit blocks, which are delivered to the unit's output. For brevity, the unit receives the initial message in 32-bit packets and the initial message has a length l, multiple of 32.

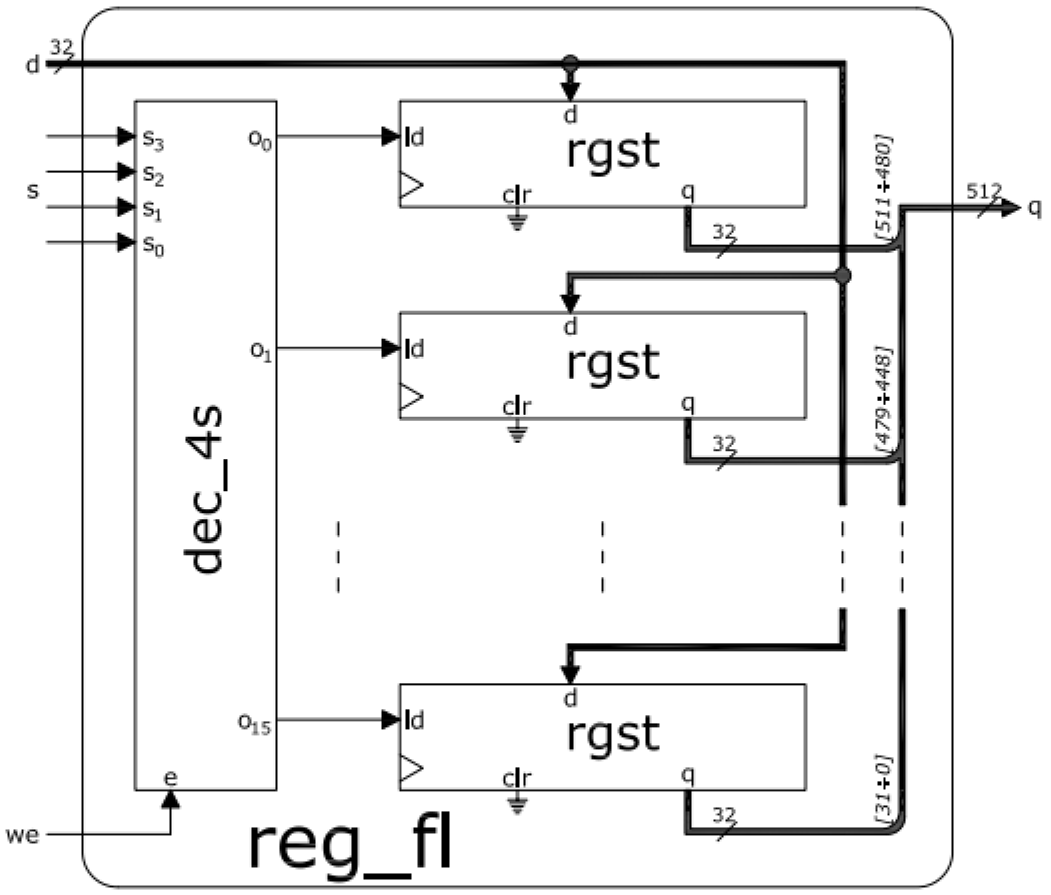
The message "abcd", represented in 8-bit ASCII code, is considered to be processed. The length of the message is ***l*** = 32 bits and it will be completed with 1 bit of 1 followed by ***k*** = 415 bits of 0 (k is obtained from the equation ***l + 1 + k*** = 448 (mode 512)). Then attach the initial length, in bits, of the message, represented as a 64-bit unsigned number.

The 512-bit block generated by the input module for this example is detailed below (fields are represented in hexadecimal):



The processing unit operates with 5 types of packages:

- ***message***: packets in which the initial message is divided and which are delivered at the entrance of the unit, one or more such packets may be received
- ***padding***: a packet with the most significant bit of 1 and 31 bits of 0, groups the first 32 bits attached in the completion phase, a single packet of this type is generated
- ***zero***: 32-bit packet of 0, group bits successively, attached in the completion phase, 0, 1 or more such packets can be generated
- ***the superior half of the message***: a packet containing the most significant 32 bits of the initial length of the message, a single packet of this type is generated
- ***the inferior half of the message***: a packet containing the least significant 32 bits of the initial length of the message, a single packet of this type is generated.



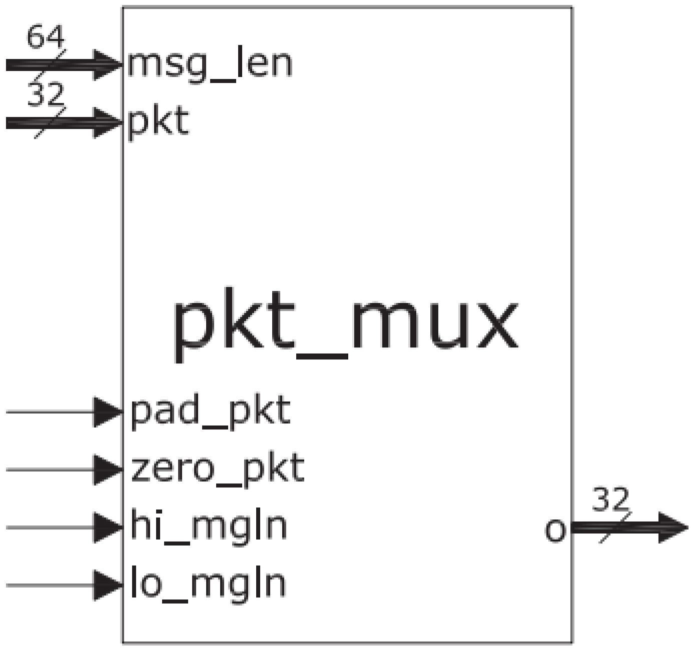
The processing unit will divide the entire binary sequence (initial message + completion bits + length of the initial message) into 512-bit blocks. In each clock cycle, the input processing unit receives a new packet from the initial message and after receiving 16 consecutive packets a new 512-bit block will be delivered to the output.

A dedicated register file, ***reg_fl***, is used to store 16 32-bit packets, its concatenated contents forming the block. After assembling a block, it is delivered to the output and the next 16 message packets are resumed. When the entire message has been received, the other 4 types of packets are generated, also stored in ***reg_fl***, assembled in 512-bit blocks.

A 4-bit counter keeps track of the write address in ***reg_fl***, for storing the next packet. A 64-bit register is used to count the length of the initial message, which will be incremented by an adder for each received message packet.

The register file receives packets at input d, the write address is provided at input s and we enable writing. At the output, q is concatenated the content of all internal registers, the register at address 0 having the most significant content.

The ***dec_4s*** module in the Register file figure is a parameterization of the available module **here.**



The dedicated ***pkt_mux*** multiplexer provides the following register file package. Its data entries are ***pkt***, for message packets and ***msg_len***, for the length of the original message.

Output ***o*** is controlled by the following selection lines:

- ***pad_pkt*** – provides a padding package
- ***zero_pkt*** – provides a zero package
- ***hi_msgln*** – provides the most significant half of the message length (received at msg_len entry)
- ***lo_msgln*** – provides the least significant half of the message length

Two or more selection entries cannot be active at the same time. If none of the 4 selection inputs is active, the current message packet received at the ***pkt*** input is provided at the output.

3. Referințe bibliografice

[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