Arrayed instantiation and `generate` constructs

Opriţoiu Flavius
flavius.opritoiu@cs.upt.ro

November 4, 2024

Introduction

Objectives:

- Build arrayed instances and configure `generate` blocks

For reading:

- "Advanced Module Instantiation", Laboratory notes [AMI**]

Arrayed instantiation shorthand to constructing multiple instances of the same module.

Generate blocks provide a more flexible approach to creating large numbers of instances with a more complex interconnection structure.

Arrayed instantiation

Shortcut to creating several instances of the same module, when all instances are connected to the same signals. Using arrayed instantiation for designs with more complex interconnections can become tedious.

The Verilog implementation bellow constructs a BCD8421 to E3 converter for *k*-digit numbers, using *k* instances of the 4-bit adder, *add4b*.

```
1  module bcde3conv #(
2    parameter k = 4          //number of digits
3  )(
4    input [4*k-1:0] bcd,     //bcd input number
5    output [4*k-1:0] e3      //e3 output number
6  );

8    add4b cnv[k-1:0] (.x(bcd),.y(4'd3),.z(e3));
9  endmodule
```

The arrayed instantiation in line 8 builds *k* instances of *add4b*.

Arrayed instantiation (contd.)

Arrayed instantiation format:

```
module-name instance_name [top-index:bottom-index]
    (.p(s), ...)
```

If the width of the signal *s* equals the number of instances times the width of port *p*, then *s* will be partitioned equally into the number of bits of port *p*, each partition being assigned to one of the instances in the array. For the *bcde3conv* example, input *bcd* is partitioned into 4 bits, each group being assigned to one of the *k* instances (similarly for output *e3*).

If the width of the signal *s* equals the width of port *p*, then *s* is connected to all instances in the array. For *bcde3conv*, the value of 3 on 4 bits, to be added to each BCD8421 digit, has the same width as port *y* of the adder, thus it is connected as it is to all *k* instances.

`generate` blocks

A more versatile mechanism for creating multiple instances of an object within a module. The following object types can be generated:

- one or several modules
- any number of `initial`/`always` blocks
- one or several continuous assignments
- any number of signal declarations

The generated instances are constructed programmatically inside a *generate block*, delimited by the `generate` and `endgenerate` keywords. The *generate block* makes use of the `for` loop for controlling the instances creation. The index control variable used by this loop is of `genvar` type, a special non-negative valued integer.

For finer control over the instantiation, the `if ... else` and `case` instructions can be used inside the *generate block*.

generate blocks (contd.)

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

- uses a `genvar` variable as loop index
- has its content within a *named* `begin ... end` block

The previous BCD8421 to E3 converter is re-written bellow:

```
1  module bcde3conv #(
2    parameter k = 4
3  )(
4    input [4*k-1:0] bcd,
5    output [4*k-1:0] e3
6  );
7    generate
8      genvar i;
9      for (i=0; i < k; i=i+1) begin: vect
10       add4b uconv(
11         .x(bcd[i*4+3:i*4]),.y(4'd3),.z(e3[i*4+3:i*4])
12       );
13     end
14   endgenerate
15 endmodule
```

`generate` blocks (contd.)

The named `begin ... end` block starts in line 9, where `begin` is followed by a valid Verilog identifier (*vect*).

Inside the named block, a single instance is created in each iteration, denoted by a different Verilog identifier (*uconv*).

The port association makes use of loop index variable *i* for grouping together 4 consecutive bits from *bcd* input and 4 from *e3* output. The consecutive bits are selected using the part-select expression $[i * 4 + 3 : i * 4]$ applied to both ports.

The value of 3, represented on 4 bits, is used for the second operand of the *k* instances of *add4b* module.

Solved problem

The input preprocessing unit of a cryptographic application

Exercise: Construct the datapath for the input preprocessing unit (IPU, or simply *the unit*) of a 256-bit Secure Hash Algorithm 2 (SHA-2) unit (see [FIPS15], section 5.1.1).

Solution: The unit receives the message at input, pads it and delivers it at the output. At the input the message is received in packets of 64 bits. At the output the padded message is delivered in blocks of 512-bits.

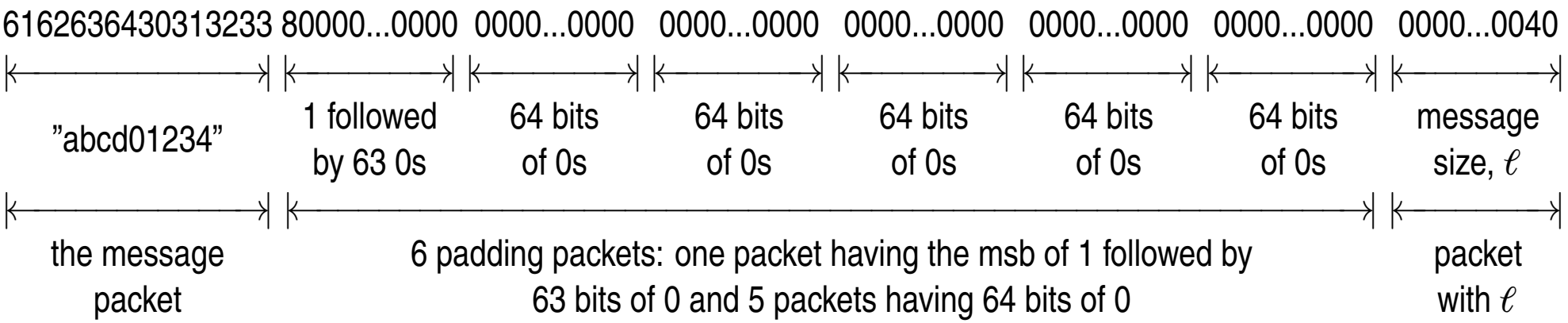
Message padding: Considering a message of ℓ bits (ℓ being a multiple of 64), padding appends, in order:

- one bit of 1
- k bits of 0s, so that $\ell + 1 + k \equiv 448 \pmod{512}$
- the value of ℓ represented on 64 bits

Solved problem (contd.)

Preprocessing phase example

Consider the 8-bit ASCII message "abcd0123" ($\ell = 64$ bits) is sent to the unit. The message is appended one bit of 1 followed by $k = 383$ bits of 0, followed by the value 64 represented on 64 bits. The 512-bit block output of the unit is represented bellow (digits are in hexadecimal):



For the 72 ASCII characters message "Dear All, I am writing to give you an update on your submitted proposal.", the unit would outputs 2 512-bit blocks.

Solved problem (contd.)

Datapath design for the unit

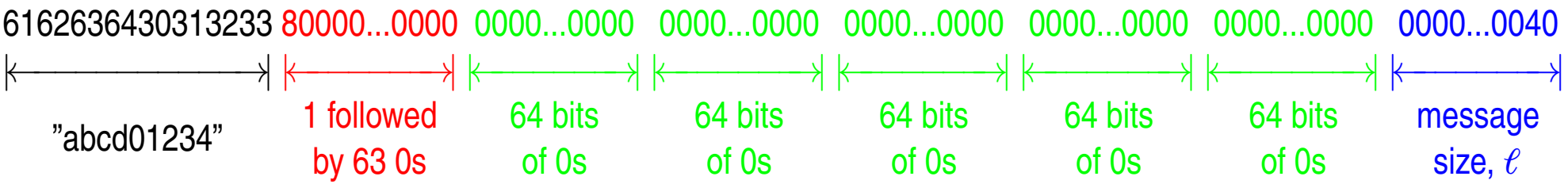
As long as the message is not completely received, in each clock cycle the unit obtains a new 64-bit message packet; $\Rightarrow$ 8 such packets form a 512-bit block. These 8 packets are stored in a register file, *regfl*, with 8 registers, 64-bits each.

Because the padded message's length is multiple of 512 and ℓ is multiple of 64: $\Rightarrow$ the length of the padding data (the bit of 1, k bits of 0s, ℓ on 64 bits) is multiple of 64 bits. It follows that the padding data can be split in 64-bit packets and stored in the register file.

Solved problem (contd.)

Datapath design for the unit

For message "abcd0123", the splitting of the padding data into 64-bit packets is depicted bellow:



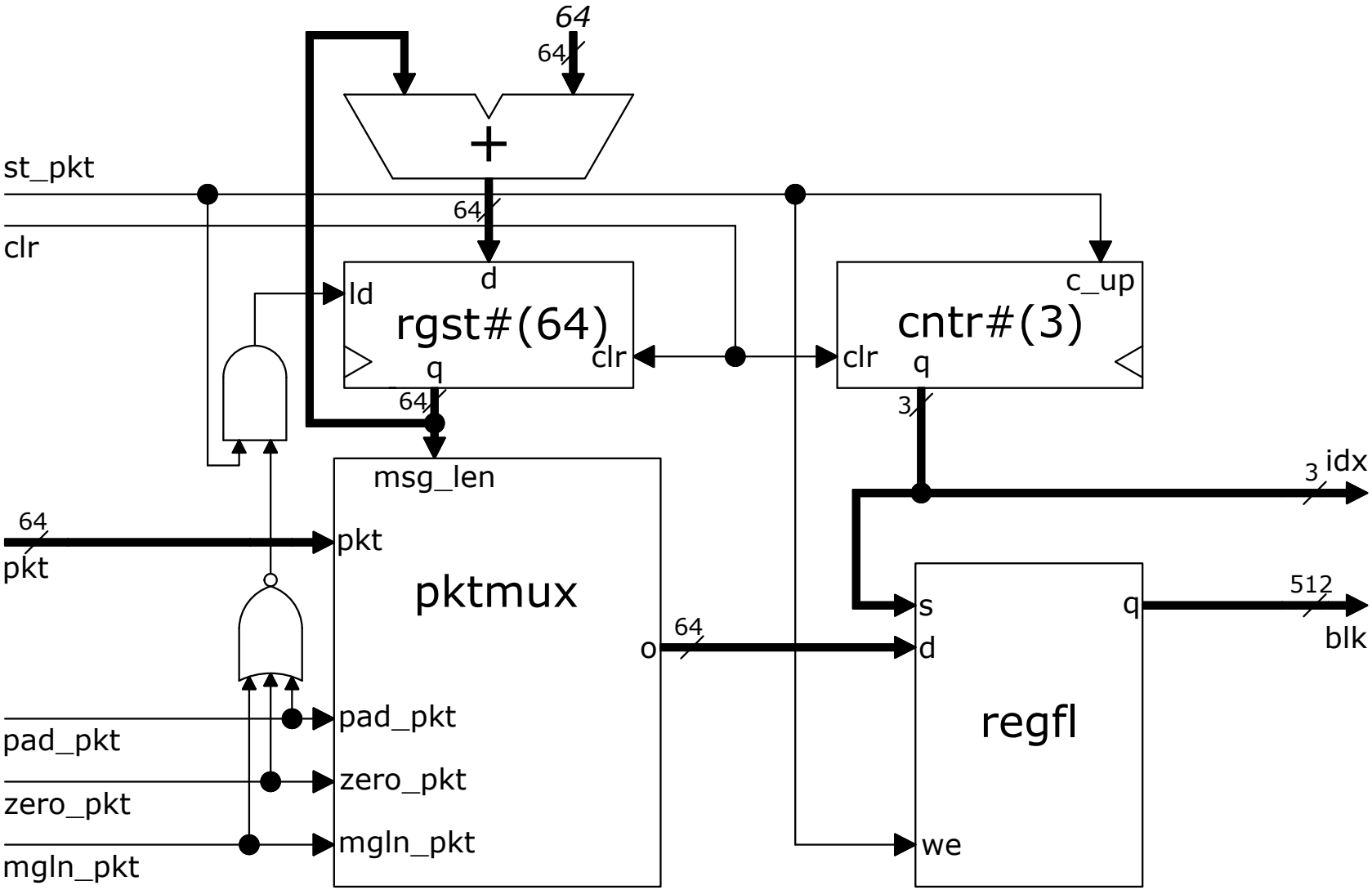
Thus, there are 4 types of packets operated by the unit's datapath:

1. message packet(s)
2. padding packet: containing a bit of 1 followed by 63 of 0s
3. zero packet(s): containing 64 bits of 0s
4. *message length* packet: value of ℓ on 64 bits

The message's length, ℓ, is calculated inside the unit using a 64-bit register, incremented by 64 each time a new packet is stored in *regfl*.

Solved problem (contd.)

Datapath design for the unit



rgst module is available ▸ here while *cntr* module is available ▸ here

Solved problem (contd.)

Datapath design for the unit

The datapath has the following inputs, as seen on the diagram from previous slide:

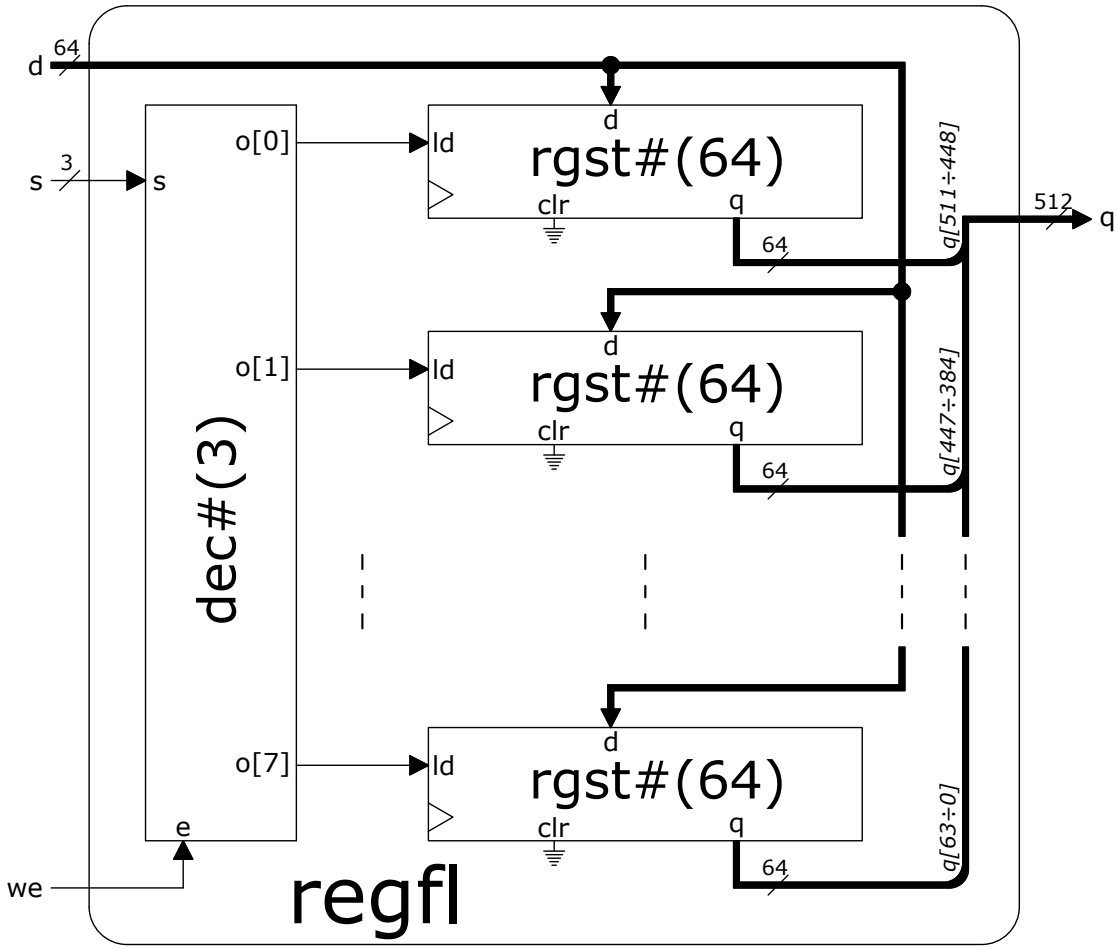
1. *pkt*: on which it receives message packets
2. *st_pkt*: activates storing of current packet
3. *clr*: clears the counter and the 64-bit register storing ℓ
4. *pad_pkt*: active if the current packet is a padding one
5. *zero_pkt*: active if the current packet is a zero packet
6. *mgln_pkt*: active if the current packet is message's length, ℓ

The datapath has the following outputs:

1. *idx*: next available address in *regfl*; it also indicates how many packets were stored so far in the current block
2. *blk*: the output 512-bit block

Solved problem (contd.)

The register file



The *dec#(3)* is an instance of the *dec* module available here, parameterized with the value of 3 for the selection line's width.

Solved problem (contd.)

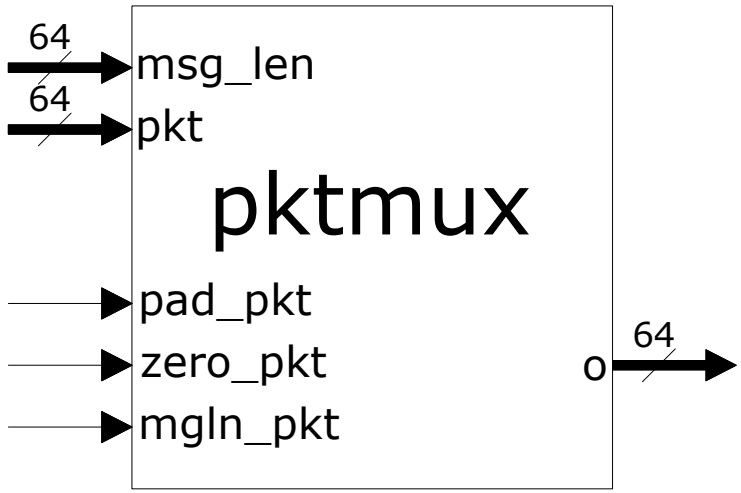
The register file

The register file, *regfl* assembles 8 consecutive packets received on input *d* into a complete blocks. The next available address in the register file is provided by a 3-bit counter to input *s* and storage of the packets is activated by the *we*, enable input.

The register file's output, *q*, is constructed by concatenating the content of al internal registers, with the register at the address 0 providing the most significant bits and the register at address 7 providing the least significant bits of the output.

Solved problem (contd.)

The packet multiplexer



The packet "multiplexer", *pktmux*, provides the current packet to be stored in the register file. Because there are 4 types of packets (see slide 11), *pktmux* has 3, mutually-exclusive control inputs:

1. *pad_pkt*: *pktmux* delivers a padding packet at its output
2. *zero_pkt*: it delivers a zero packet
3. *mgln_pkt*: it delivers the message length packet, provided by the 64-bit register to *pktmux*'s input *ms_len*

If none of the 3 control lines are active, *pktmux* delivers message packets, received on its input *pkt*.

References

[AMI**] Advanced Module Instantiation. [Online]. Available: http://www.eecs.umich.edu/courses/eecs470/OLD/w14/labs/lab6_ex/AMI.pdf (Last accessed 17/04/2016).

[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: http://dx.doi.org/10.6028/NIST.FIPS.180-4 (Last accessed 06/04/2016).