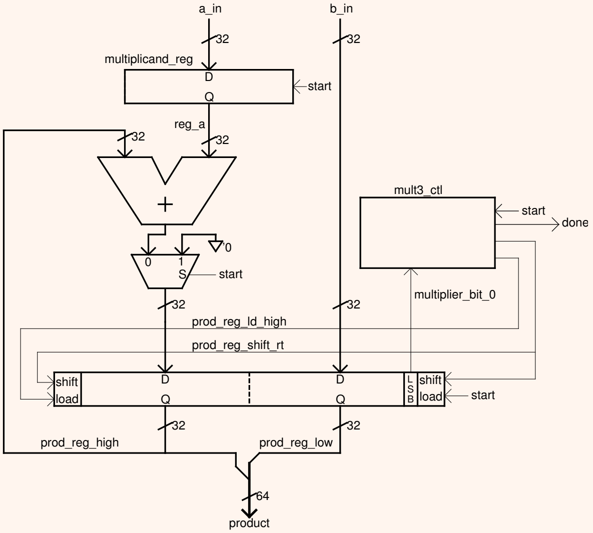
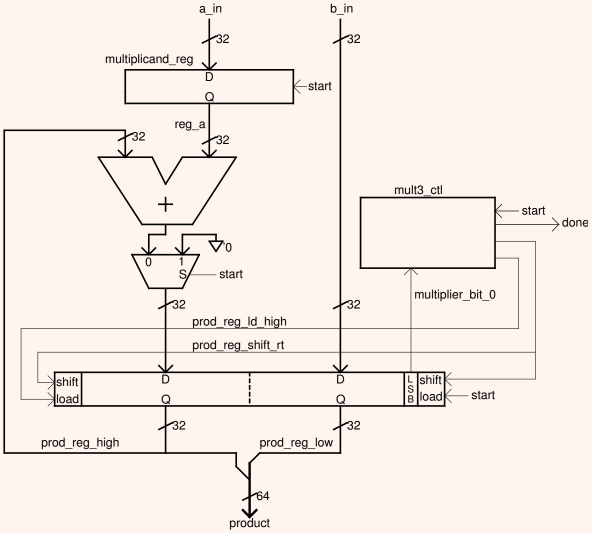
Modules, ports, instantiation

- Development of timing and state diagrams refined the block diagram
- Cleaned up and added better names, added detail



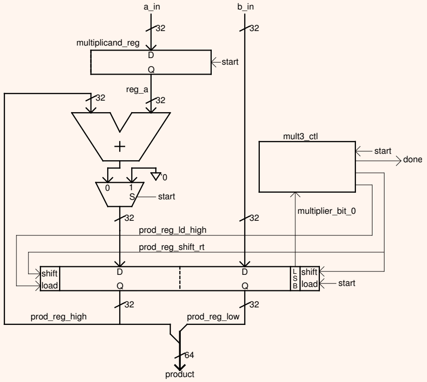
Modules, ports, instantiation

- As its reasonable, we want to maintain our partitioning in SV
- We have two mechanisms to do this: **modules** and **always blocks**



Modules, ports, instantiation

- Simple functionality belongs in an always block.
- If considerable complexity or natural boundary (data vs. control) exists, a module is appropriate.
- Our datapath logic is simple and will be in always blocks
- "mult3_ctl" is a natural partitioning boundary and will be in its own module.



Modules, ports, instantiation

The module is the basic unit of hierarchy in Verilog

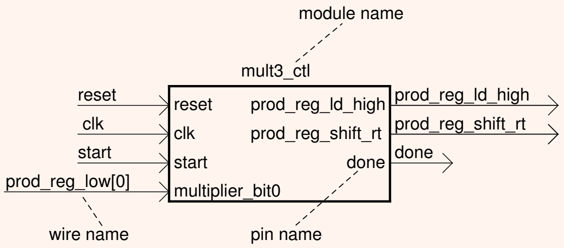
- Modules describe:
  - boundaries [module, endmodule]
  - inputs and outputs [ports]
  - how it works [behavioral or RTL code]
- Can be a single element or collection of lower level modules
- Module can describe a hierarchical design (a module of modules)
- A module should be contained within one file
- Module name should match the file name
- Module mult3_ctl resides in file named mult3_ctl.sv
- Multiple modules can reside within one file (not recommended)

Modules, ports, instantiation

- Modules are declared with their name and ports

```
module mult3_ctl(
  input   reset,
  input   clk,
  input   start,                    //begin multiplication
  input   multiplier_bit0,
  output logic prod_reg_ld_high,    //load high half of register
  output logic prod_reg_shift_rt,   //shift product register right
  output logic done);               //signal completion of mult op
```

- The Verilog declaration of the module corresponds directly to the block diagram. The module mult3_ctl will reside inside mult3_ctl.sv



Modules, ports, instantiation

- Comments
  - Comments are done just like C
  - One line comments begin with //
  - Multi-line comments start with /*, end with: */
- Identifiers
  - Identifiers are names given to objects so that they may be referenced
  - They start with alphabetic chars or underscore
  - They cannot start with a number or dollar sign
  - All identifiers are case sensitive unlike VHDL

Modules, ports, instantiation

- ANSI C Style Ports in System Verilog are declared as:
  - port mode, port type, port width, port name... where:
    - Port mode is *input*, *output* or *inout*
    - Port type defaults to wire, outputs are usually type *logic*
    - Port width is [MSB:LSB]
    - Port name is simply the name of the port

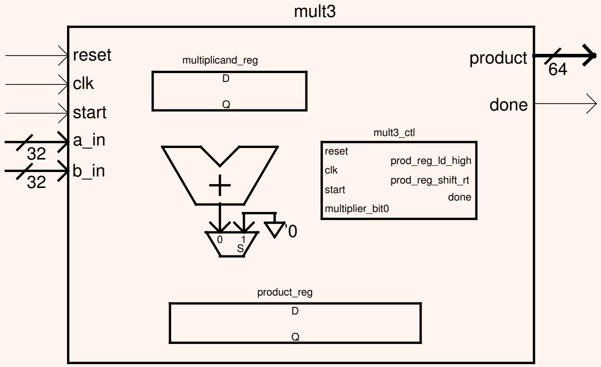
```
  input         [31:0] register_in,    //register input
  output logic  [31:0] register_out,   //register output
```

- Input ports are understood as nets (wires) within the module, and can only be read from.
- Output ports are understood as nets (wires) withing the module and can be read from or written to if they are type logic.
- Inout ports are understood as nets (wires) withing the module and can be read from or written to if they are type logic.
- Use inout ports for tri-state logic only.

Modules, ports, instantiation

Hierarchical Design

- Hierarchical designs have a top level module and lower level modules.
- Lower level modules (like mult3_ctl) are *instantiated* within the higher level module (mult3)



Modules, ports, instantiation

- Module mult3 is just a higher-level module, so we define it as before.
- Module mult3 will reside in another file called mult3.sv.
- The beginning of mult3.sv looks like this:

```
module mult3(
  input  reset,
  input  clk,
  input  [31:0] a_in,
  input  [31:0] b_in,
  input  start,
  output logic [63:0] product,
  output logic done);
```

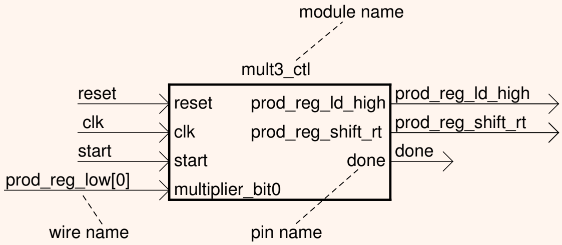
Modules, ports, instantiation

- Module mult3_ctl is instantiated within mult3 as shown and is concluded with the keyword endmodule

```
module mult3(
  input   reset,
  input   clk,
  input   [31:0] a_in,
  input   [31:0] b_in,
  input   start,
  output logic [63:0] product,
  output logic done);
//
//  Other code goes in here
//
//instantiate the control module
mult3_ctl mult3_ctl_0(
  .reset              (reset            ),
  .clk                (clk              ),
  .start              (start            ),
  .multiplier_bit0    (prod_reg_low[0]  ),
  .prod_reg_ld_high   (prod_reg_ld_high ),
  .prod_reg_shift_rt  (prod_reg_shift_rt ),
  .done               (done             ));
endmodule
```

Modules, ports, instantiation

- The instantiation associates a wire in an upper-level module with a pin (or port) on a lower level module. Pins on left, wires on right.



```
//instantiate the control module
mult3_ctl mult3_ctl_0(
  .reset             (reset               ),
  .clk               (clk                 ),
  .start             (start               ),
  .multiplier_bit0   (prod_reg_low[0]     ), //LSB of product reg
  .prod_reg_ld_high  (prod_reg_ld_high    ),
  .prod_reg_shift_rt (prod_reg_shift_rt   ),
  .done              (done                ));
```

Modules, ports, instantiation

Instantiation subtleties

- Ports of the upper-level module are understood to be wires within that module.
- Wires connected to pins can have differing names.
- Wires and pins *should not* have differing widths!
- Followng the module name is an instance name that makes each instantiation of the same module unique.

```
  mult3_ctl mult3_ctl_0(
```

- Comment each line if designer intention is not crystal clear.
- This method of instantiation is called *named association*. Its the preferred way of doing instantiation when pins/names differ.

Modules, ports, instantiation

- Its possible to instantiate using pin/wire position only.
- This is called *positional association*. Its a bad idea.
- For example:

```
//instantiate the control module
mult3_ctl mult3_ctl_0(reset,clk,start,prod_reg_low[0],
                      prod_reg_ld_high, prod_reg_shift_rt,done);
```

- How do you know pins and wires are connected correctly?
- Could you check it quickly if the module had 50 pins on it?
- What if you wanted to add wire in the middle of the list?
- *Do not use Positional Association!*
- It saves time once, and costs you dearly afterwords

Modules, ports, instantiation

Implicit Naming is useful when most pins/wires have the same name.

```
//instantiate the control module
mult3_ctl mult3_ctl_0(
  .reset              (reset             ),
  .clk                (clk               ),
  .start              (start             ),
  .multiplier_bit0    (prod_reg_low[0]   ),  //product reg LSB
  .prod_reg_ld_high   (prod_reg_ld_high  ),
  .prod_reg_shift_rt  (prod_reg_shift_rt ),
  .done               (done              ));
```

```
//instantiate the control module
mult3_ctl mult3_ctl_0(
  .reset                                ,
  .clk                                  ,
  .start                                ,
  .multiplier_bit0    (prod_reg_low[0]),  //named association
  .prod_reg_ld_high                     ,
  .prod_reg_shift_rt                    ,
  .done                                 );
```

Implicit and named association forms can be mixed

Modules, ports, instantiation

If all pins and associated wires have the same name we can do this:

```
//instantiate the control module
mult3_ctl mult3_ctl_0(.*);  //now that's a short cut!
```

- This is the most powerful form of implicit association.
- This method is best reserved for top level testbench/top module instantiation, else, it could hide intent.

Modules, ports, instantiation

Unconnected port pins may be left unconnected like this...

```
//instantiate the control module, don't need connection to done pin
mult3_ctl mult3_ctl_0(
  .reset               (reset                ),
  .clk                 (clk                  ),
  .start               (start                ),
  .multiplier_bit0     (prod_reg_low[0]      ),  //product reg LSB
  .prod_reg_ld_high    (prod_reg_ld_high     ),
  .prod_reg_shift_rt   (prod_reg_shift_rt    ),
  .done                (                     ));
```

- Unused ports *may* simply be left out of the port list...*bad idea!*
- Keep your intent clear!
- Careful formatting of names, parenthesis and commas makes it easy to catch errors.

Modules, ports, instantiation

- How are lower level modules connected if a port definition has not created the wire?
- In the upper-level module, wires are declared; usually near the beginning of the module.

```
module mult3(
  input  reset,
  input  clk,
  input  [31:0] a_in,
  input  [31:0] b_in,
  input  start,
  output logic [63:0] product,
  output logic done);

//declare internal wires
  logic [31:0] reg_a;          //output, multiplicand reg
  logic [31:0] prod_reg_high; //output, upper half of product reg
  logic [31:0] prod_reg_low;  //output, lower half of product reg
  logic prod_reg_ld_high;     //load, upper half of product reg
  logic prod_reg_shift_rt;    //shift product regright

//lots more code goes here

endmodule
```

Modules, ports, instantiation

- Format of wire declaration is: `type width[MSB:LSB] wire_name;`
- Width is optional

```
logic [31:0] prod_reg_low;  //lower half of product register
```

- If no width is given, the wire is assumed to be a single conductor.
- If width is given, a bus is created of that width. Busses are groups of logically related wires.

Modules, ports, instantiation

- Parametrization of modules allows for code reuse
- In the module declaration, the keyword "parameter" is optional

```
//a generic width register declaration
module multi_reg #(parameter width=8) (
  input        reset
  input        clk
  input        enable
  input        data_in  [width-1:0]
  output logic data_out [width-1:0]
  )
  \\
  \\description of register goes here
  \\
endmodule
```

Modules, ports, instantiation

- The parametrized register could be instantiated like this:

```
module top_module;

logic [31:0] data_out;
wire reset, clk, enable, data_in;

//instantiate register and override width to 32 bits
multi_reg #(.width(32)) multi_reg_0(
  .reset      (reset    ),
  .clk        (clk      ),
  .enable     (enable   ),
  .data_in    (data_in  ),
  .data_out   (data_out ));
endmodule
```

Modules, ports, instantiation

- Verilog modules...
  - provide a coarse-grain structuring mechanism
  - contain RTL logic or other modules
  - provide an abstraction mechanism via complexity hiding
  - provide a way for design reuse
- Question: How is a Verilog module different from a C function?