ECE 4514
Digital Design II
Spring 2008

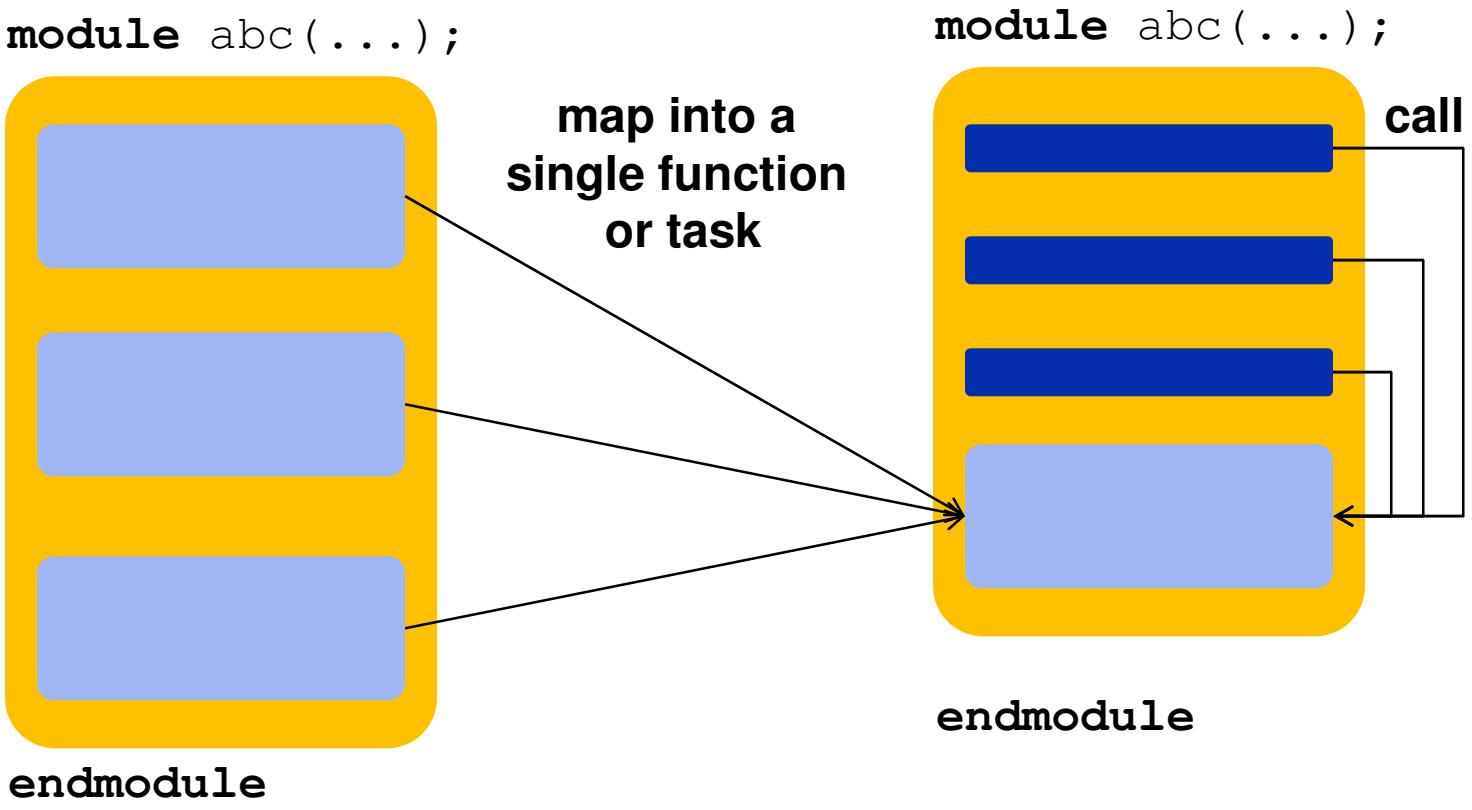
Lecture 21:
Functions and Tasks

A *Language* Lecture

Patrick Schaumont

Tasks and Functions

- Help with the reusability of writing code



Tasks vs Functions

- Tasks, as well as functions, are *behavioral* constructs.
- They support expressions, if-then, case, loop statements
- They are local to a module, and called from within initial and always blocks.

Tasks

- Can contain event control (@, #, wait)
- May execute in non-zero simulation time
- Can have zero or more arguments of type in, out, inout

Functions

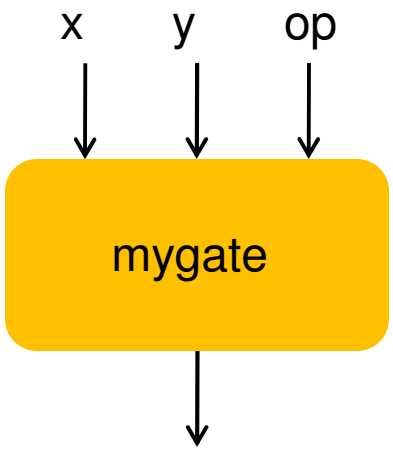
- Cannot contain event control (@, #, wait)
- Must execute in zero simulation time
- Must have at least in input argument, and has a single output argument

Functions

```
module andorgate(q, a, b);
  output [1:0] q;
  input        a, b;

  function mygate;
    input x, y;
    input op;
    case (op)
      1'b0: mygate = x & y;
      1'b1: mygate = x | y;
      default: mygate = 1'b0;
    endcase
  endfunction

  assign q[0] = mygate(a, b, 0); // and gate
  assign q[1] = mygate(a, b, 1); // or gate

endmodule
```

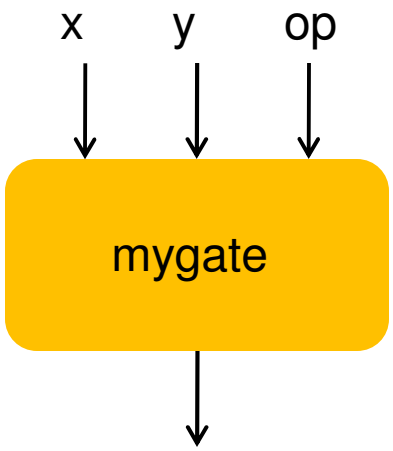
Functions

```
module andorgate(q, a, b);
  output [1:0] q;
  input        a, b;

  function mygate;
    input x, y;
    input op;
    case (op)
      1'b0: mygate = x & y;
      1'b1: mygate = x | y;
      default: mygate = 1'b0;
    endcase
  endfunction

  assign q[0] = mygate(a, b, 0); // and gate
  assign q[1] = mygate(a, b, 1); // or gate

endmodule
```



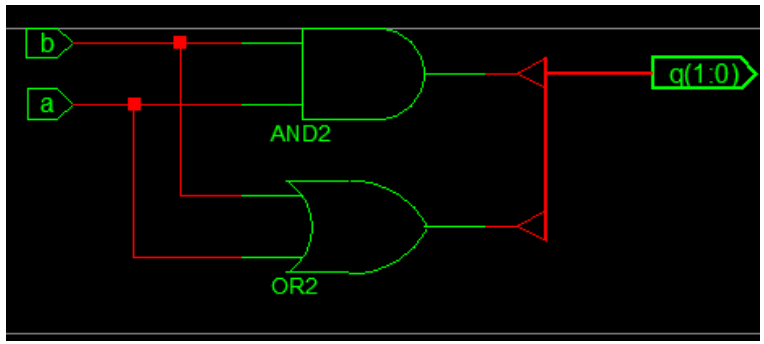
Functions

```
module andorgate(q, a, b);
  output [1:0] q;
  input        a, b;

  function mygate;
    input x, y;
    input op;
    case (op)
      1'b0: mygate = x & y;
      1'b1: mygate = x | y;
      default: mygate = 1'b0;
    endcase
  endfunction

  assign q[0] = mygate(a, b, 0); // and gate
  assign q[1] = mygate(a, b, 1); // or gate
                          x   y   op
endmodule
```



Functions

```
module andorgate(q, a, b);
  output [1:0] q;
  input        a, b;

  function mygate;
    input x, y;
    input op;
    case (op)
      1'b0: mygate = x & y;
      1'b1: mygate = x | y;
      default: mygate = 1'b0;
    endcase
  endfunction

  assign q[0] = mygate(a, b, 0); // and gate
  assign q[1] = mygate(a, b, 1); // or gate

endmodule
```



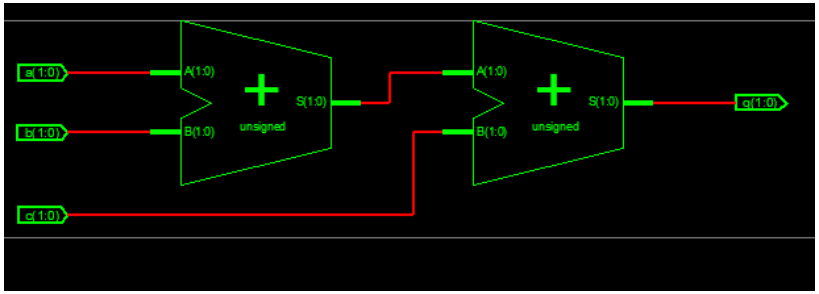
The logic is defined by the function calls, not by the function declaration

Functions can have local variables (reg)

```
module addit(q, a, b, c);
  output [1:0] q;
  input  [1:0] a, b, c;

  function [1:0] myadd;
    input [1:0] x, y, z;
    reg [1:0] w;
    begin
    w     = x + y;
    myadd = w + z;
    end
  endfunction

  assign q = myadd(a, b, c);

endmodule
```

← Two-bit output

← Local variable



Functions can have local variables (reg)

```
module addit(q, a, b, c);
  output [1:0] q;
  input  [1:0] a, b, c;

  function [1:0] myadd;
    input [1:0] x, y, z;
    reg [1:0] w;
    begin
    w     = x + y;
    myadd = w + z;
    end
  endfunction

  assign q = myadd(a, b, c);

endmodule
```

← Two-bit output

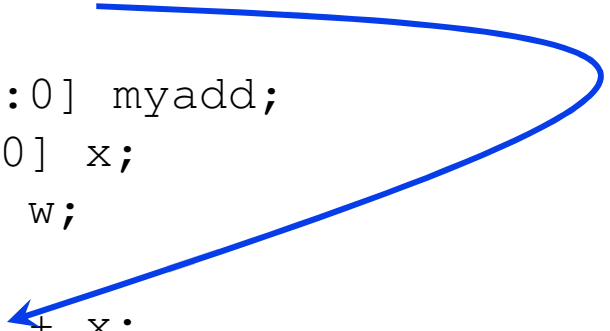
← Local variable

w is shared across *all* invocations of myadd

w is similar to a static variable in C

Functions can access module (global) variables

```
module addit(q, a, b, c);
  output [1:0] q;
  input  [1:0] a, b, c;

  function [1:0] myadd;
    input [1:0] x;
    reg [1:0] w;
    begin
    w     = a + x;
    myadd = w + c;
    end
  endfunction

  assign q = myadd(c);  // will return c + a + c

endmodule
```



Automatic Functions

```
module addit(q1, q2, a, b, c);
  output [1:0] q1, q2;
  input  [1:0] a, b, c;

  function automatic [1:0] myadd;
    input [1:0] x, y, z;
    reg [1:0] w;
    begin
    w     = x + y;
    myadd = w + z;
    end
  endfunction

  assign q1 = myadd(a, b, c);
  assign q2 = myadd(a, a, a);

endmodule
```

automatic ensures that all local variables are truly local

With automatic, each invocation of myadd will use a different w

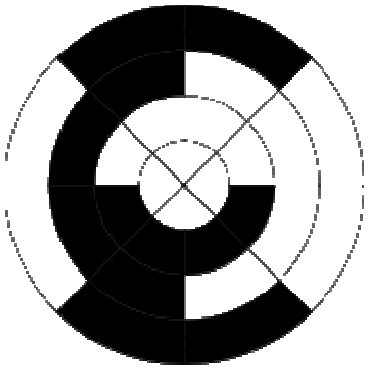
Constant functions

- Don't create logic, but do compile-time calculations

```
module ram(address);
  parameter  ram_depth = 256;
  localparam w = bits(ram_depth);
  input [w-1:0] address;

  function bits;
    input [31:0] v;
    for (bits = 0; v > 1; bits = bits - 1)
      v = v >> 1;
  endfunction

  // ..

endmodule
```

Example: Gray to Bin Conversion Function

- Remember Gray Code ?
  - A gray code is a code where only 1 bit flips per code word



3-bit gray code	**000**
	001
	011
	010
	110
	111
	101
	100

- How do we build a gray-to-bin function?

Example: Gray to Bin Conversion Function

- Gray to Bin should convert this:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

- It can be shown that

bin-bit i = xor-sum(gray-bit i .. gray-bit N)

Example: Gray to Bin Conversion Function

- Gray to Bin should convert this:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

bit3 = gray3 = 1

- It can be shown that

bin-bit i = xor-sum(gray-bit i .. gray-bit N)

Example: Gray to Bin Conversion Function

- Gray to Bin should convert this:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

bit3 = gray3 = 1

bit2 = gray3 xor gray2 = 1 xor 1 = 0

- It can be shown that

bin-bit i = xor-sum(gray-bit i .. gray-bit N)

Example: Gray to Bin Conversion Function

- Gray to Bin should convert this:

IN	OUT
000	000
001	001
011	010
010	011
110	100
111	101
101	110
100	111

bit3 = gray3 = 1

bit2 = gray3 xor gray2 = 1 xor 1 = 0

bit1 = gray3 xor gray2 xor gray1
= 1 xor 1 xor 0 = 0

- It can be shown that

bin-bit i = xor-sum(gray-bit i .. gray-bit N)

Example: Gray to Bin Conversion Function

- Now let's build the gray to bin function

```
function [2:0] gray_to_bin;
  input [2:0] a;
  reg   [2:0] q;
  begin
     q[2] = a[2];
     q[1] = a[1] ^ a[2];
     q[0] = a[0] ^ a[1] ^ a[2];
     gray_to_bin = q;
  end
endfunction
```

Example: Gray to Bin Conversion Function

- Cool. Now let's use a loop to build the gray to bin function

```
function [W:0] gray_to_bin;
  input  [W:0] a;
  reg    [W:0] q;

  begin



  end
endfunction
```

Example: Gray to Bin Conversion Function

- Cool. Now let's use a loop to build the gray to bin function

```
function [W:0] gray_to_bin;
  input  [W:0] a;
  reg    [W:0] q;
  integer i;
  begin
  for (i=0; i<=W; i = i + 1)
    q[i] = ^(a >> i);
  gray_to_bin = q;
  end
endfunction
```

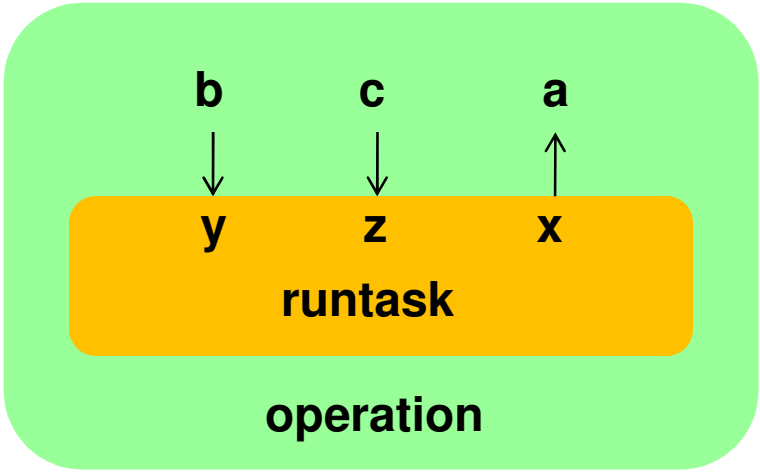
Tasks

```
module operation;
  parameter delay = 5;
  reg [7:0] a, b, c;

  always @(a, b, c)
    runtask(a, b, c);    // same as: #5 a = b + c;

  task runtask;
  output [7:0] x;
  input  [7:0] y, z;
  begin
    #delay x = y + z;
  end
  endtask

  initial begin
    #5  b = 0; c = 2;
    #10 b = 8;
  end
endmodule
```

Tasks

```
module operation;
  parameter delay = 5;
  reg [7:0] a, b, c;

  always @(a, b, c)
    runtask(a, b, c);    // same as: #5 a = b + c;

  task runtask;
  output [7:0] x;
  input  [7:0] y, z;
  begin
    #delay x = y + z;
  end
  endtask

  initial begin
    #5  b = 0; c = 2;
    #10 b = 8;
  end
endmodule
```

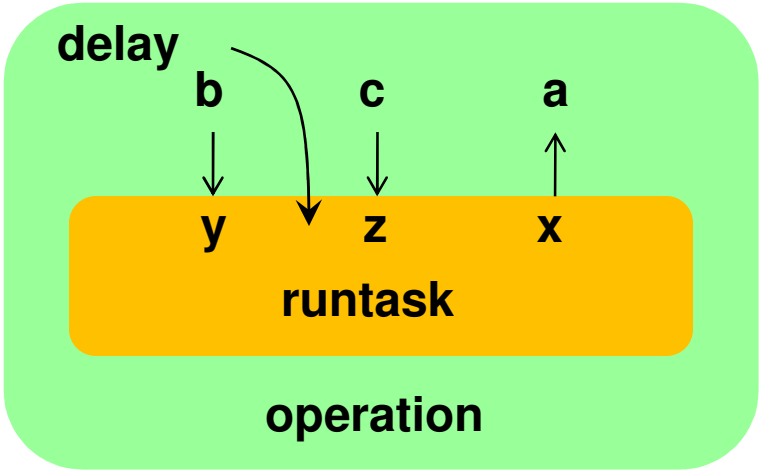
These are not inputs/outputs of the module, but links to the variables of the module



Tasks

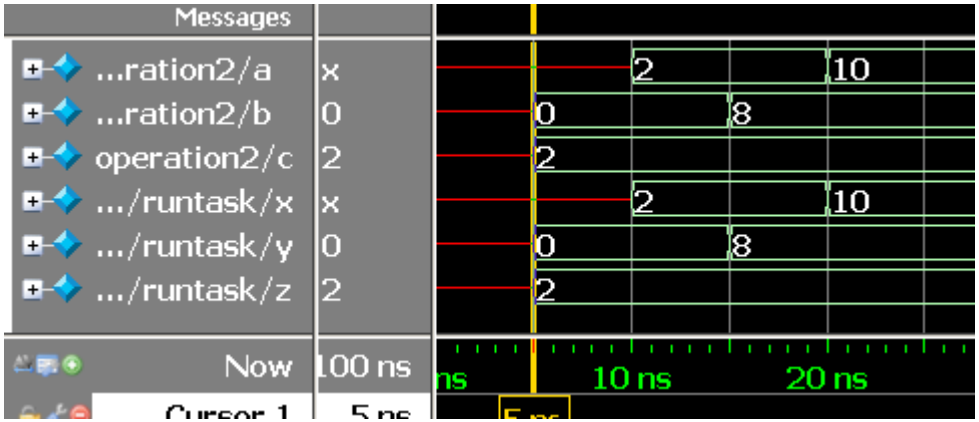
```
module operation;
  parameter delay = 5;
  reg [7:0] a, b, c;

  always @(a, b, c)
    runtask(a, b, c);    // same as: #5 a = b + c;

  task runtask;
  output [7:0] x;
  input  [7:0] y, z;
  begin
    #delay x = y + z;
  end
  endtask

  initial begin
    #5  b = 0; c = 2;
    #10 b = 8;
  end
endmodule
```

Parameters, variables, etc can be passed down from module into a task



Tasks

```
module operation;
  parameter delay = 5;
  reg [7:0] a, b, c;

  always @(a, b, c)
    runtask(a, b, c);    // same as: #5 a = b + c;

  task runtask;
  output [7:0] x;
  input  [7:0] y, z;
  begin
    #delay x = y + z;
  end
  endtask

  initial begin
    #5  b = 0; c = 2;
    #10 b = 8;
  end
endmodule
```



Tasks

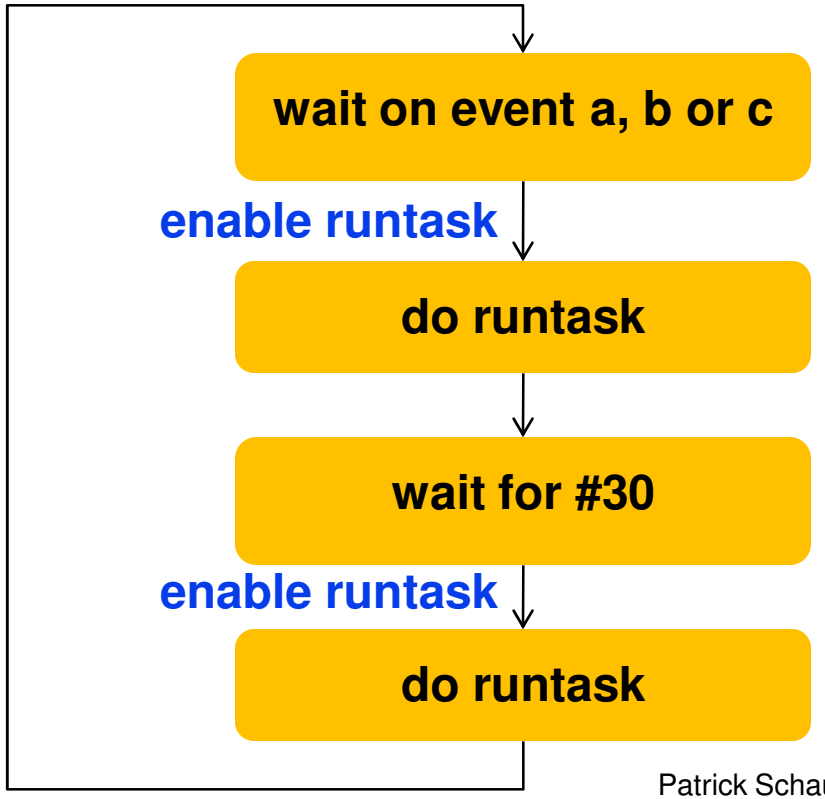
```
module operation;
  parameter delay = 5;
  reg [7:0] a, b, c;

  always @(a, b, c) begin
    runtask(a, b, c);
    #30 runtask(b, c, a);
  end

  task runtask;
  output [7:0] x;
  input  [7:0] y, z;
  begin
    #delay x = y + z;
  end
  endtask

  initial begin
    #5  b = 0; c = 2;
    #10 b = 8;
  end
endmodule
```

The moment when a task is ready to run is called *enabling a task*

Tasks

```
module operation;
  parameter delay = 5;
  reg [7:0] a, b, c;

  always @(a, b, c) begin
    runtask(a, b, c);
    #30 runtask(b, c, a);
  end

  task runtask;
  output [7:0] x;
  input  [7:0] y, z;
  begin
    #delay x = y + z;
  end
  endtask

  initial begin
    #5  b = 0; c = 2;
    #10 b = 8;
  end
endmodule
```

The moment when a task is ready to run is called *enabling a task*



Functions and Tasks Summary

- Functions and Tasks enable shorthand notations within always and initial blocks
- Functions are single-output, and do not support event control
- Tasks may have more then a single output, and may have event-control statements
- In practice, only functions are used in synthesis
  - shorthand for combinational logic
  - calculation of compile-time constants