Verilog: Function, Task

• Verilog: Functions
  • A function call is an operand in an expression.
  • It is called from within the expression and returns a value used in the expression.
  • Functions may be called from within procedural and continuous assignment statement.

• Verilog: Tasks
  • A task call is a separate procedural statement. It can not be called from a continuous assignment statement.

module processor1; //using function multiply
reg [15:0] m [0:8191]; // 8192 x 16 bit memory
reg [15:0] pc; // 16 bit program counter
reg [15:0] acc; // 16 bit accumulator
reg ck; // a clock signal
reg [15:0] ir; // 16 bit instruction register
always begin: executeInstructions
  @(posedge ck)
  ir = m [pc];

  @(posedge ck)
  case (ir [15:13])
    ................
    3'b111: acc = multiply(acc, m [ir[12:0]]);
    endcase
  pc = pc + 1;
end

function [15:0] multiply;
  input [15:0] a;
  input [15:0] b;
begin: serialMult
  reg [7:0] mcnd, mpy;
  mpy = b[7:0];
  mcnd = a[7:0];
  multiply = 0;
  repeat (8)
    begin
      if (mpy[0]) multiply = multiply + {mcnd, 8'b00000000};
      multiply = multiply >> 1;
      mpy = mpy >> 1;
    end
  end
endfunction
endmodule

Multiply

• Binary multiplication is just a bunch of right shifts and adds

\[
\begin{array}{c}
\text{multiplicand} \\
\text{multiplier} \\
\{ \text{partial product array} \}
\end{array}
\]

\[
\begin{array}{c}
\text{double precision product} \\
2n
\end{array}
\]
module processor2; //using task multiply
reg [15:0] m [0:8191]; // 8192 x 16 bit memory
reg [15:0] pc; // 16 bit program counter
reg [15:0] acc; // 16 bit accumulator
reg ck; // a clock signal
reg [15:0] ir; // 16 bit instruction register
always begin: executeInstructions
  @(posedge ck)
  ir = m [pc];
  @(posedge ck)
  case (ir [15:13]) //case expressions
    3'b111: begin
      wait (~done)
      mcnd = m [ir[12:0]];
      go = 1;
      wait (done);
      acc = prod;
      end
    endcase
  pc = pc + 1;
end
endmodule

module multiply (prod, mpy, mcnd, go, done);
output [15:0] prod;
input [7:0] mpy, mcnd;
input go;
output done;
reg [15:0] prod;
reg [7:0] myMpy;
reg done;
always begin
  done = 0;
  wait (go);
  myMpy = mpy;
  prod = 0;
  repeat (8)
    begin
      if (myMpy[0])
        prod = prod + {mcnd, 6'b000000};
      prod = prod >> 1;
      myMpy = myMpy >> 1;
      end
    end
  done = 1;
  wait (~go);
end
endmodule

module processor3; //instantiate multiply module
reg [15:0] m [0:8191]; // 8192 x 16 bit memory
reg [15:0] pc; // 16 bit program counter
reg [15:0] acc; // 16 bit accumulator
reg ck; // a clock signal
reg [15:0] mcnd;
reg go;
wire [15:0] prod;
wire done;
multiply mul (prod, acc, mcnd, go, done);
always begin
  @(posedge ck) //other case expressions
    3'b111: begin
      wait (~done)
      mcnd = m [ir[12:0]];
      go = 1;
      wait (done);
      acc = prod;
      end
    endcase
  pc = pc + 1;
end
endmodule

task multiply;
inout [15:0] a;
input [15:0] b;
begin: serialMult
  reg [7:0] mcnd, mpy; //multiplicand and multiplier
  reg [15:0] prod; //product
  mpy = b[7:0];
  mcnd = a[7:0];
  prod = 0;
  repeat (8)
    begin
      if (mpy[0])
        prod = prod + {mcnd, 6'b000000};
      prod = prod >> 1;
      end
    end
  a = prod;
endtask
endmodule
### Booth’s Algorithm

1. Scan the multiplier from right (l.s.) to left (m.s.) and do the followings:
   a. **00**: Middle of a string of 0s, so no arithmetic operations.
   b. **10**: Beginning of a string of 1s, so subtract the multiplicand from the left half of the product.
   c. **01**: End of a string of 1s, so add the multiplicand to the left half of the product.
   d. **11**: Middle of a string of 1s, so no arithmetic operation.

2. Shift the Product register right (arithmetic) 1 bit.

---

### Booth’s Algorithm

### Example: $2 \times -4 = 8$

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial</td>
<td>0000 1100 0</td>
</tr>
<tr>
<td>1</td>
<td>00 → No operation</td>
<td>0000 1100 0</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>0000 0110 0</td>
</tr>
<tr>
<td>2</td>
<td>00 → No operation</td>
<td>0000 0110 0</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>0000 0011 0</td>
</tr>
<tr>
<td>3</td>
<td>10 → Prod − Mcaand</td>
<td>1110 1011 0</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td>4</td>
<td>11 → No operation</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>1111 1010 1</td>
</tr>
</tbody>
</table>

---

### Example: $2 \times -3 = -6$

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Step</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial</td>
<td>0000 1110 0</td>
</tr>
<tr>
<td>1</td>
<td>10 → Prod − Mcaand</td>
<td>1110 1101 0</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>1111 0110 1</td>
</tr>
<tr>
<td>2</td>
<td>01 → Prod + Mcaand</td>
<td>0001 0110 1</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>0000 1010 0</td>
</tr>
<tr>
<td>3</td>
<td>10 → Prod − Mcaand</td>
<td>1110 1011 0</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td>4</td>
<td>11 → No operation</td>
<td>1111 0101 1</td>
</tr>
<tr>
<td></td>
<td>arithmetic right shift</td>
<td>1111 1010 1</td>
</tr>
</tbody>
</table>