Digital Design with Verilog

Verilog

Lecture 16: Data path and Control path Design

Complete the quote

"Good artists ___copy___.
Great artists ___steal___ ."

-Pablo Picasso

- The following slides are only slightly modified from those in the MIT 6.375 course, Prof. Arvind http://csg.csail.mit.edu/6.375/
- Verilog Tutorial, by Dr. Sat Garcia University of San Diego.
- Datapath Design, Coding Standards by Michael B.Taylor, UCSD.

Learning Objectives

- Data path & Control path
- Separating control from data
- Designing Data path
- Designing Control path
- Integrating Data path & Control path

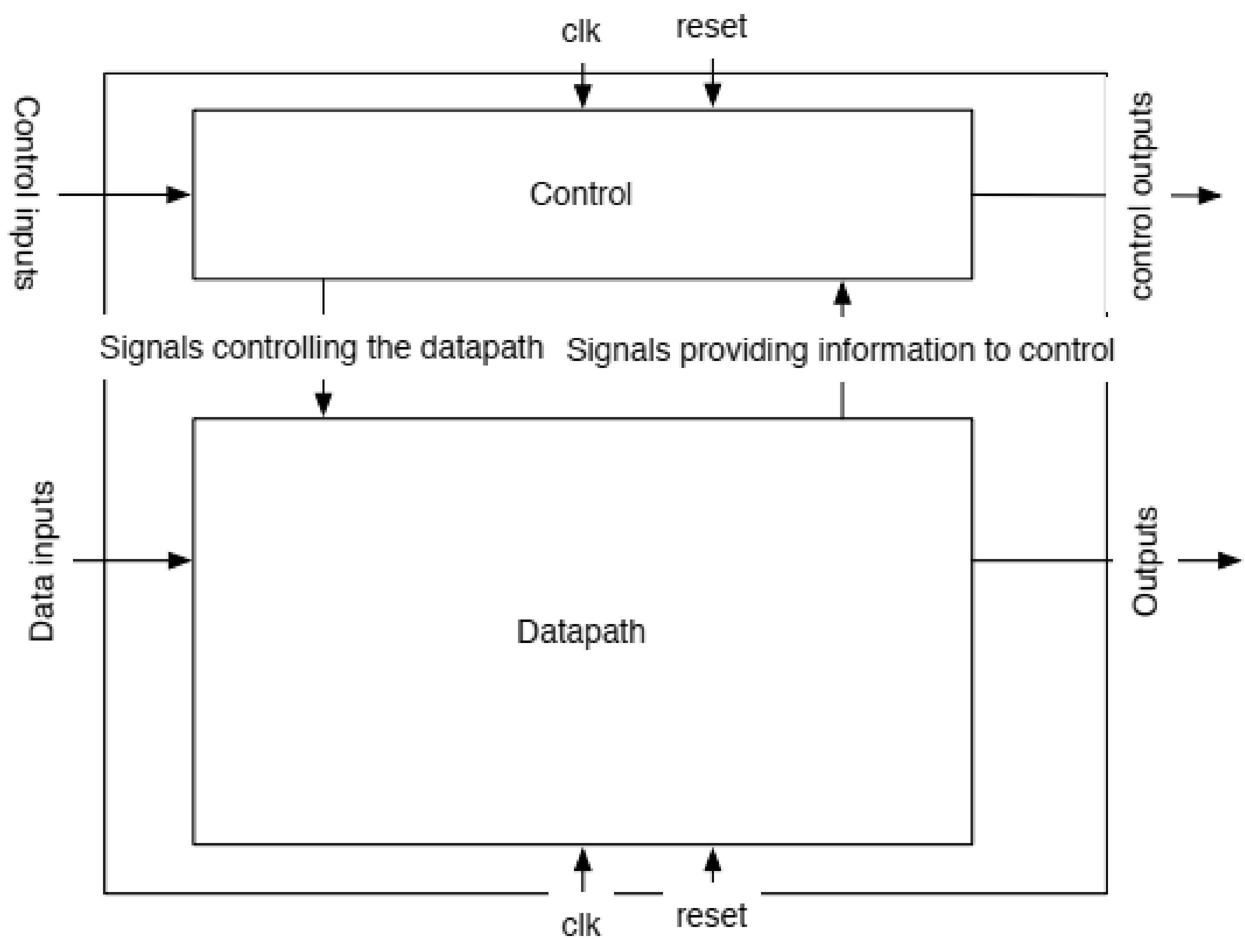
Data Path

- The datapath is where data moves from place to place.
  - Computation happens in the datapath
  - No decisions are made here.
  - Things you should find in a datapath
    - Muxes
    - Registers
    - ALUs
  - Mostly about wiring things up

Control Path

- Control is where decisions are made
- Things you will find there are
  - State machines
  - Random lots of complex logic
  - Little state (maybe just a single register)
- There are best practices from people who build real chips---Following them will save you lots of pain.

Basic Design



Designing a GCD Calculator

Euclid's algorithm for computing the Greatest Common Divisor (GCD):

15		6	
9		6	*subtract*
3		6	*subtract*
6		3	*swap*
3		3	*subtract*
0	*answer:*	3	*subtract*

Euclid’s algorithm in C

```
int GCD( int inA, int inB) {
    int done = 0;
    int A = inA;
    int B = inB;
    while ( !done ) {
        if ( A < B ) {
            swap = A;
            A = B;
            B = swap;
        }
        else if ( B != 0 )
            A = A - B;
        else
            done = 1;
        }
    return A;
}
```

How do we implement this in hardware?

Greatest Common Divisor

```
module gcd_beh #( parameter W = 16 ) (
        input [W-1:0] inA, inB,
        output [W-1:0] out);

        reg [W-1:0] A, B, out, swap;
        integer done;

        always @(*)
        begin
                done = 0;
                A = inA;
                B = inB;
                while (!done)
                begin
                        if ( A < B )
                        begin
                                swap = A;
                                A = B;
                                B = swap;
                        end
                        else if ( B!=0 )
                                A=A-B;
                        else
                                done = 1;
                        end
                                out = A;
                end
endmodule
```

Greatest Common Divisor

```
module gcd_beh_tb;
        reg [15:0] inA, inB;
        wire [15:0] out;
        gcd_beh#(16) gcd_unit( .inA(inA), .inB(inB), .out(out) );
        initial
        begin
                // 3 = GCD( 27, 15 )
                inA = 27;
                inB = 15;
                #10;
                if ( out == 3 )
                        $display( "Test ( gcd(27,15) ) succeeded, [ %x == %x ]", out, 3 );
                else
                        $display( "Test ( gcd(27,15) ) failed, [ %x != %x ]", out, 3 );
                $finish;
        end
endmodule
```

Greatest Common Divisor

```
module gcd_beh #( parameter W = 16 ) (
        input [W-1:0] inA, inB,
        output [W-1:0] out);

        reg [W-1:0] A, B, out, swap;
        integer done;

        always @(*)
        begin
                done = 0;
                A = inA;
                B = inB;
                while (!done)
                begin
                        if ( A < B )
                        begin
                                swap = A;
                                A = B;
                                B = swap;
                        end
                        else if ( B!=0 )
                                A=A-B;
                        else
                                done = 1;
                        end
                                out = A;
                end
endmodule
```

What's wrong with this approach?

Doesn't synthesize! (notice that data dependent loop?)

Making the code synthesizable

- Start with behavioral and find out what hardware constructs you'll need
  - Registers (for state)
  - Functional units
    - Adders / Subtractors
    - Comparators
    - ALU's

Identify the Hardware Structures

```
module gcd_beh #( parameter W = 16 ) (
        input [W-1:0] inA, inB,
        output [W-1:0] out);

        reg [W-1:0] A, B, out, swap;
        integer done;

        always @(*)
        begin
                done = 0;
                A = inA;
                B = inB;
                while (!done)
                begin
                        if ( A < B )
                        begin
                                swap = A;
                                A = B;
                                B = swap;
                        end
                        else if ( B!=0 )
                                A=A-B;
                        else
                                done = 1;
                        end
                                out = A;
                end
endmodule
```

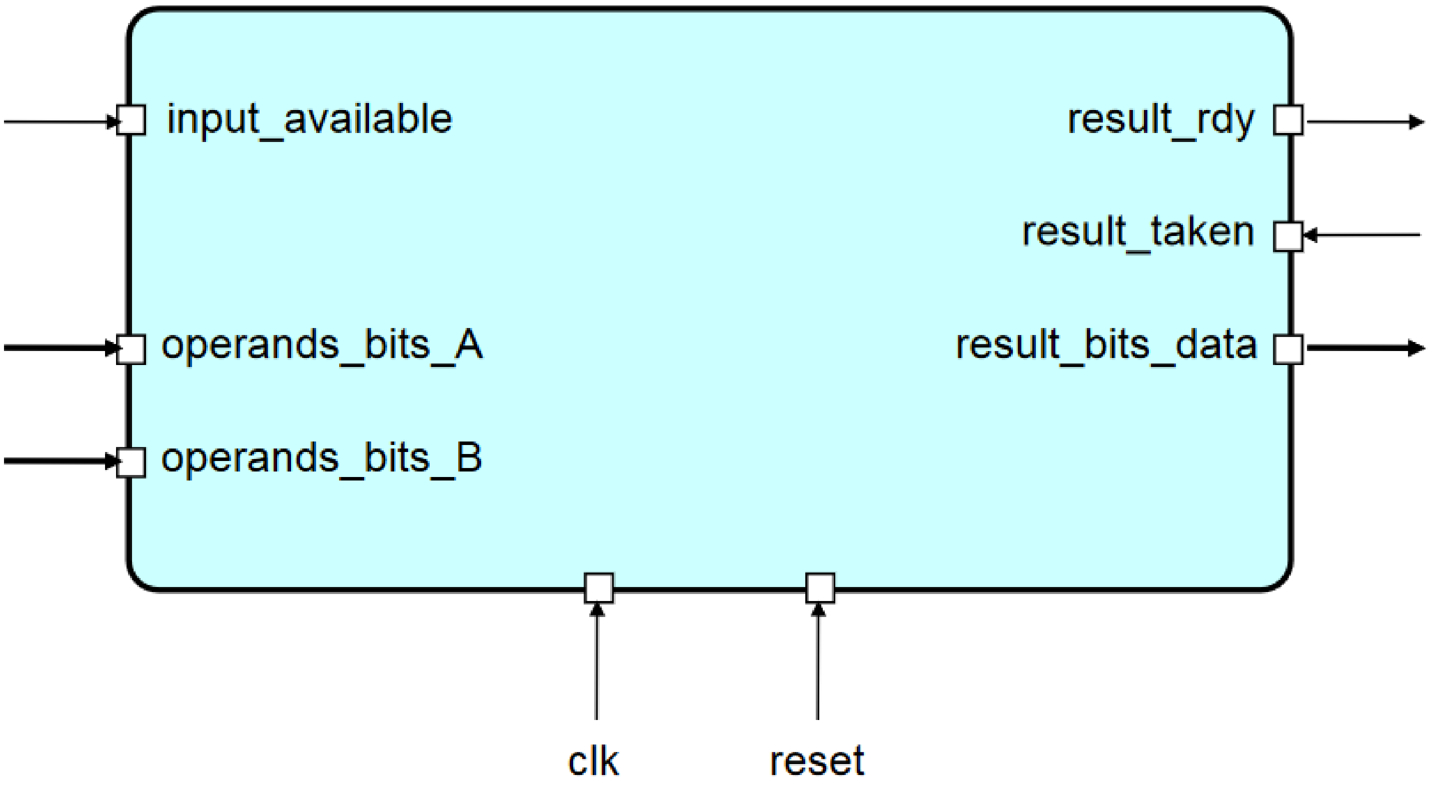
State → Registers

Less than comparator

Equality comparator

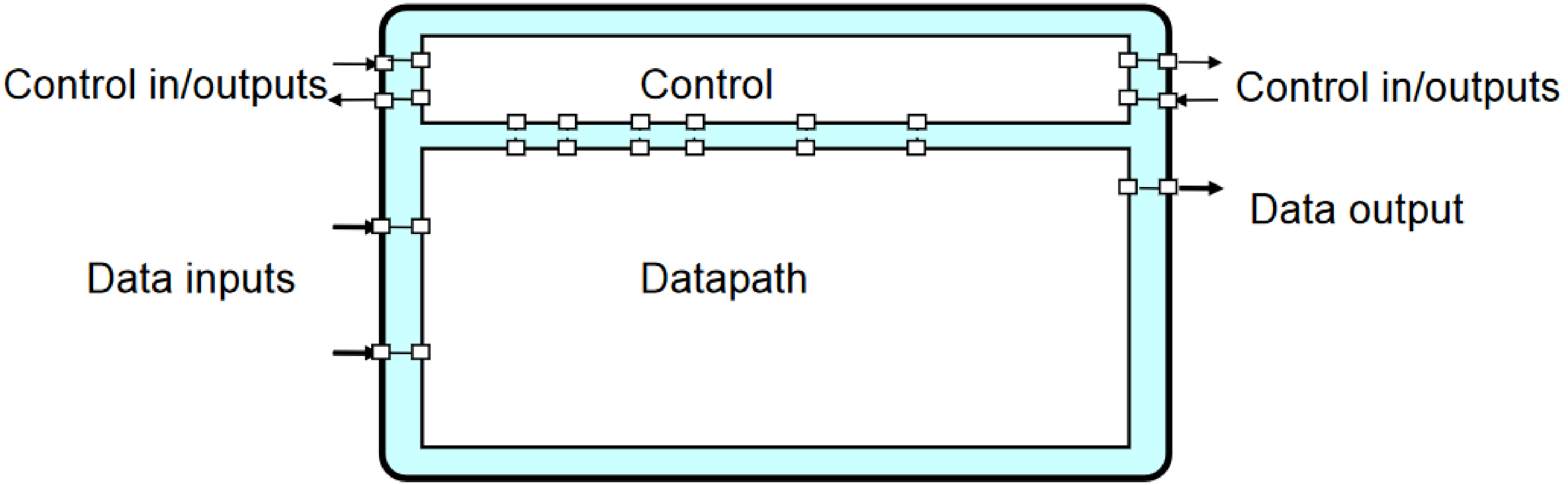
Subtractor

Next step: Define module ports

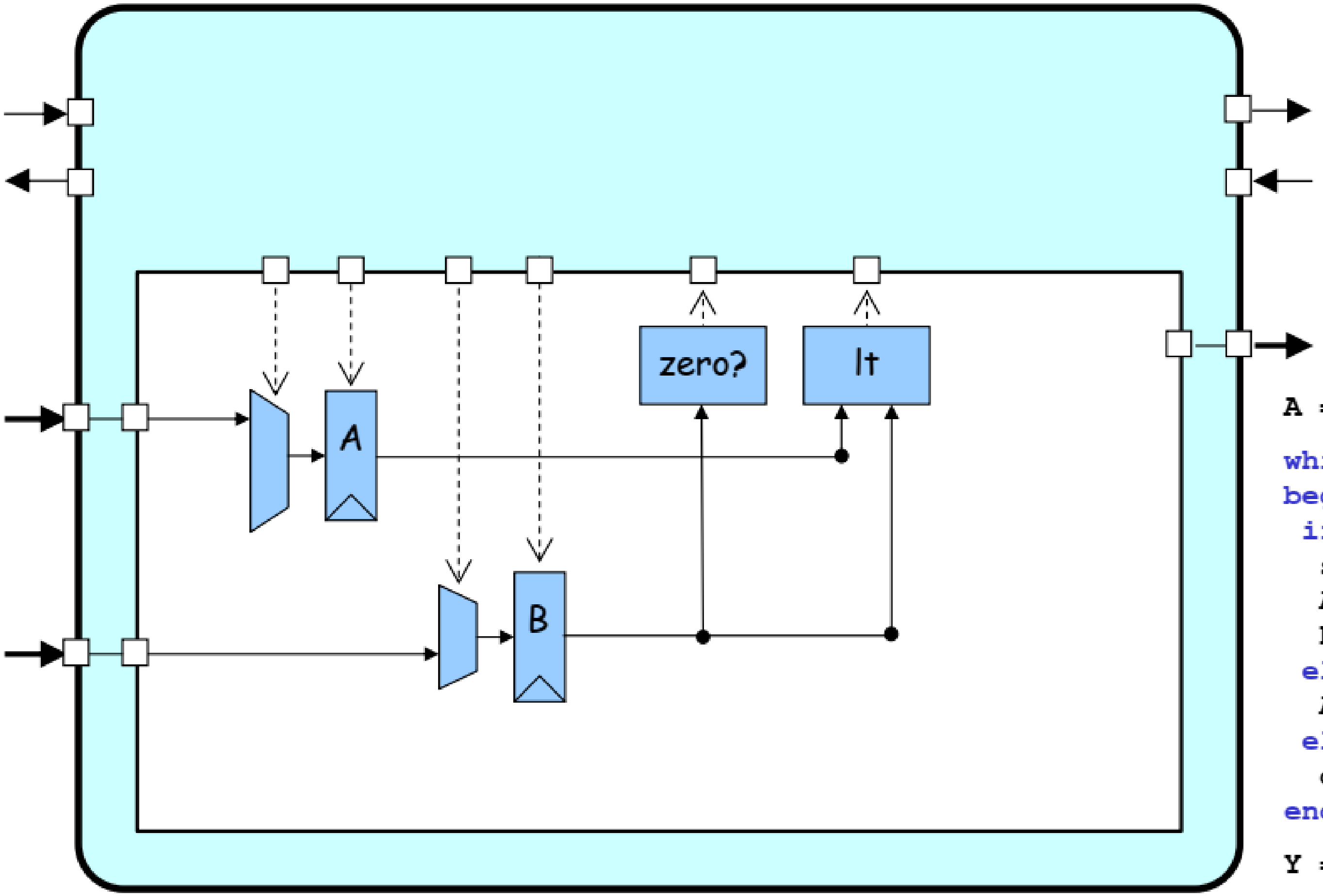


Implementing the modules

- Two step process:
  - Define datapath
  - Define controlpath

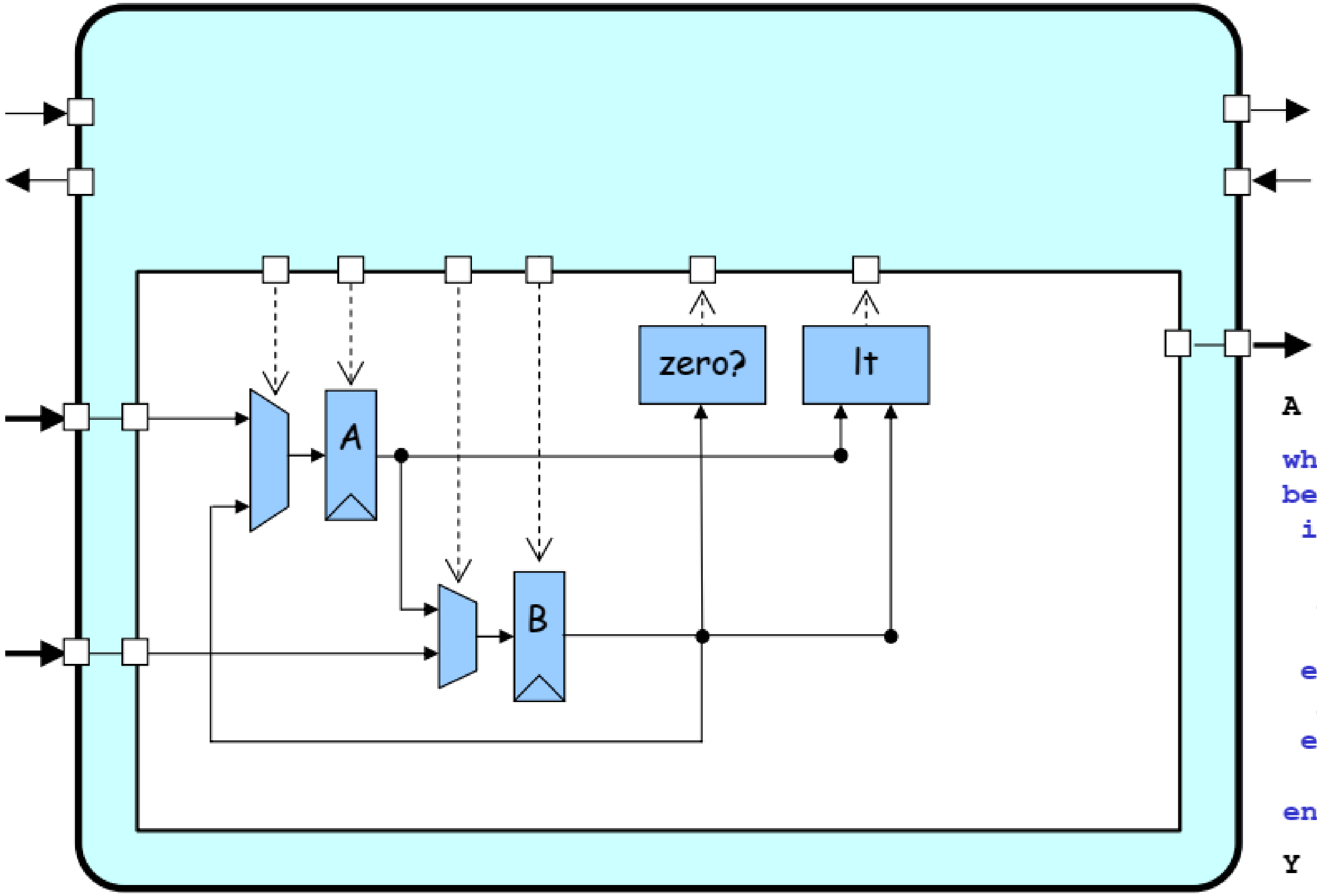


Developing the Datapath



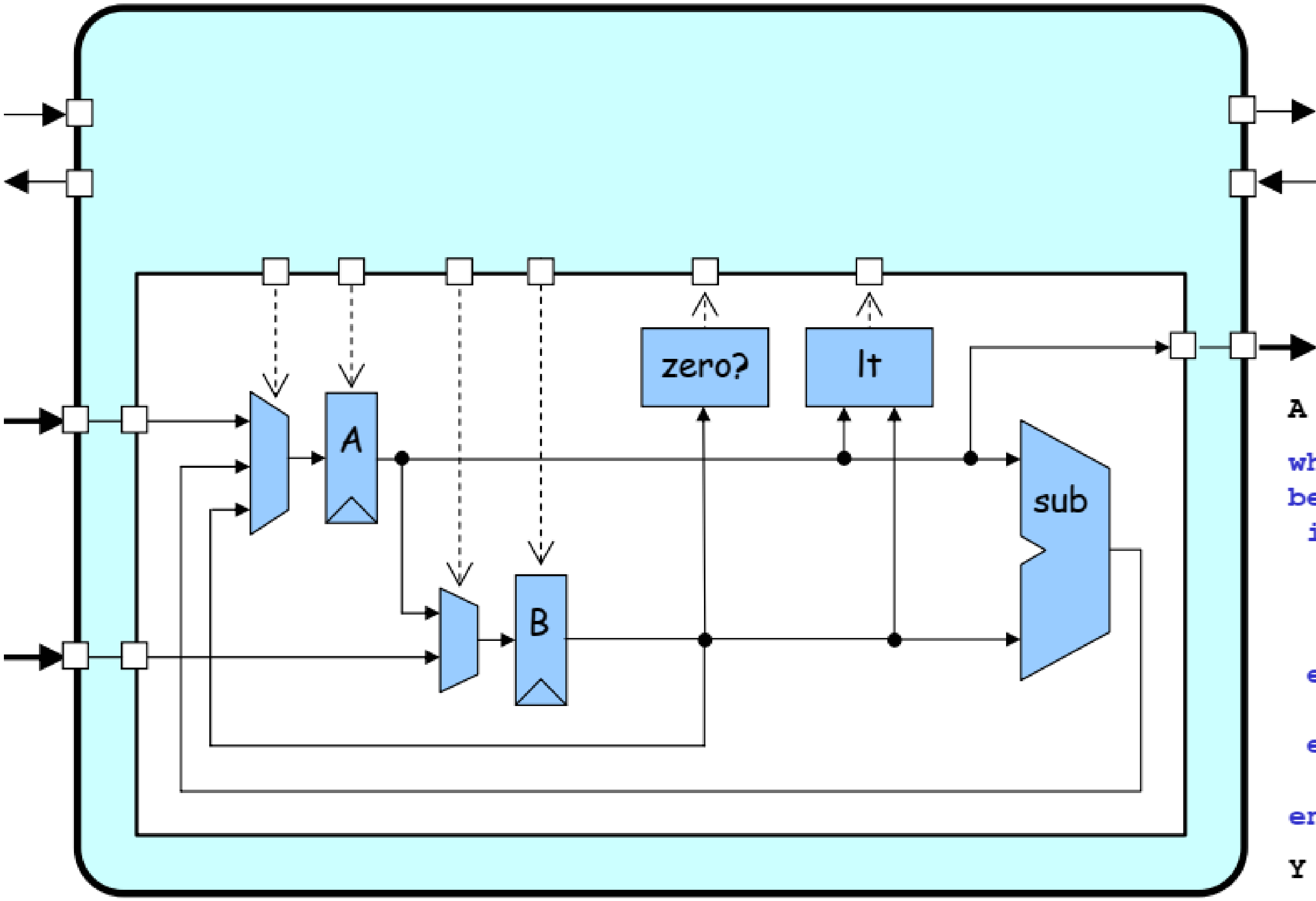

```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if ( B != 0 )
  A = A - B;
 else
  done = 1;
end

Y = A;
```

Developing the Datapath



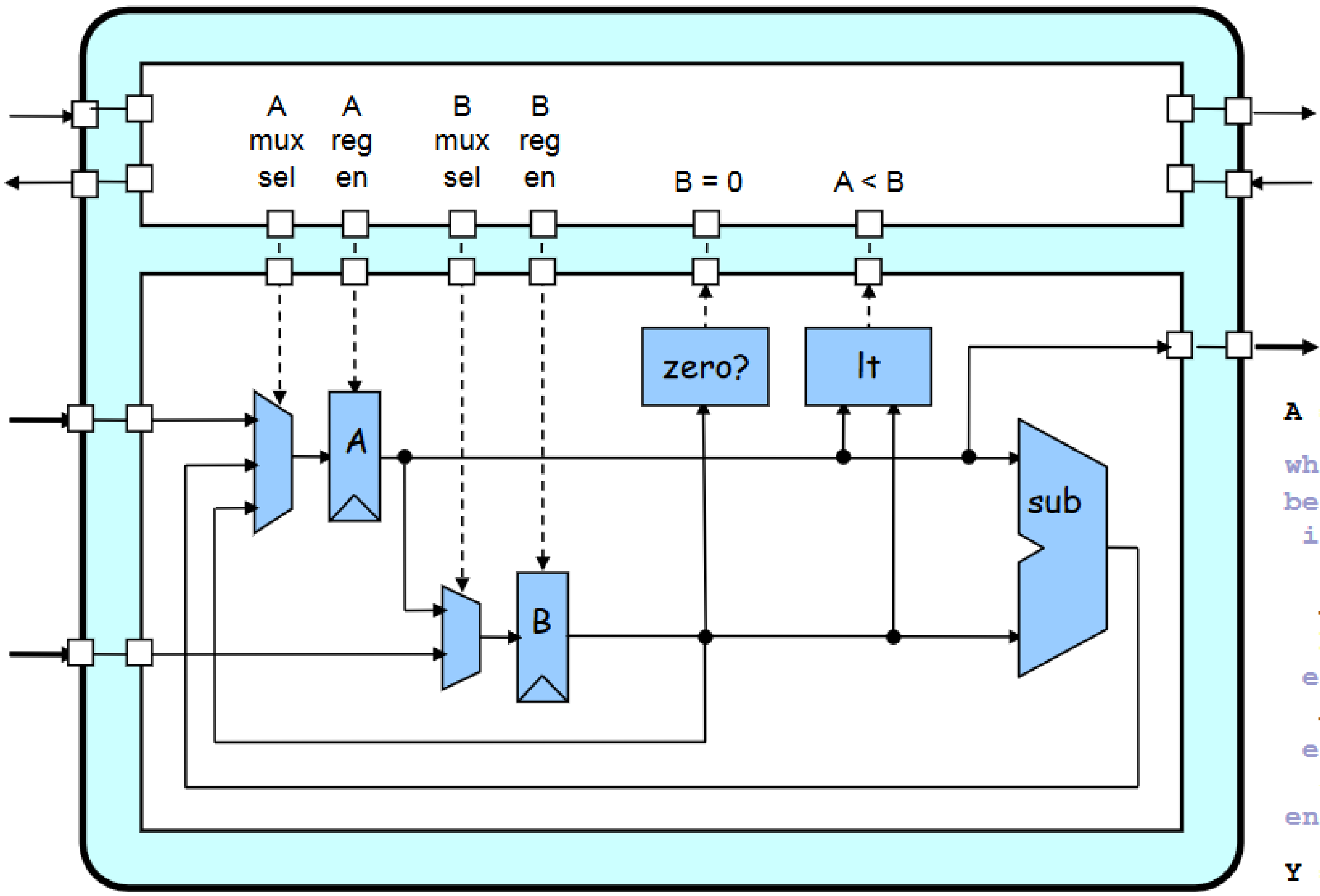

```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if ( B != 0 )
  A = A - B;
 else
  done = 1;
end

Y = A;
```

Developing the Datapath




```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if ( B != 0 )
  A = A - B;
 else
  done = 1;
end

Y = A;
```

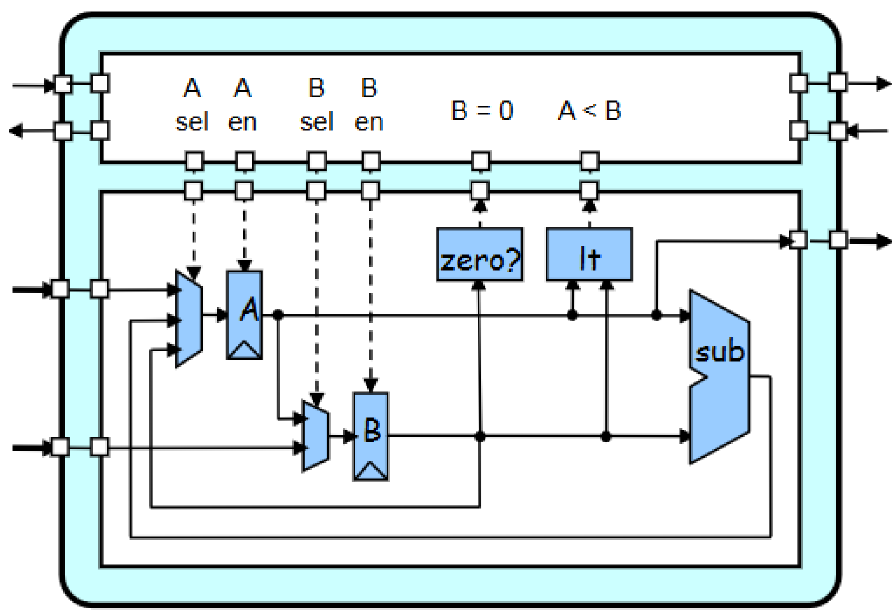
Adding Control




```
A = inA; B = inB;

while ( !done )
begin
 if ( A < B )
  swap = A;
  A = B;
  B = swap;
 else if ( B != 0 )
  A = A - B;
 else
  done = 1;
end

Y = A;
```

Datapath Module

```
module gcdDatapath#( parameter W = 16 )
(
  input       clk,
  // Data signals
  input   [W-1:0] operands_bits_A,
  input   [W-1:0] operands_bits_B,
  output  [W-1:0] result_bits_data,
  // Control signals (ctrl->dpath)
  input            A_en,
  input            B_en,
  input   [1:0] A_mux_sel,
  input            B_mux_sel,
  // Control signals (dpath->ctrl)
  output           B_zero,
  output           A_lt_B
);
```



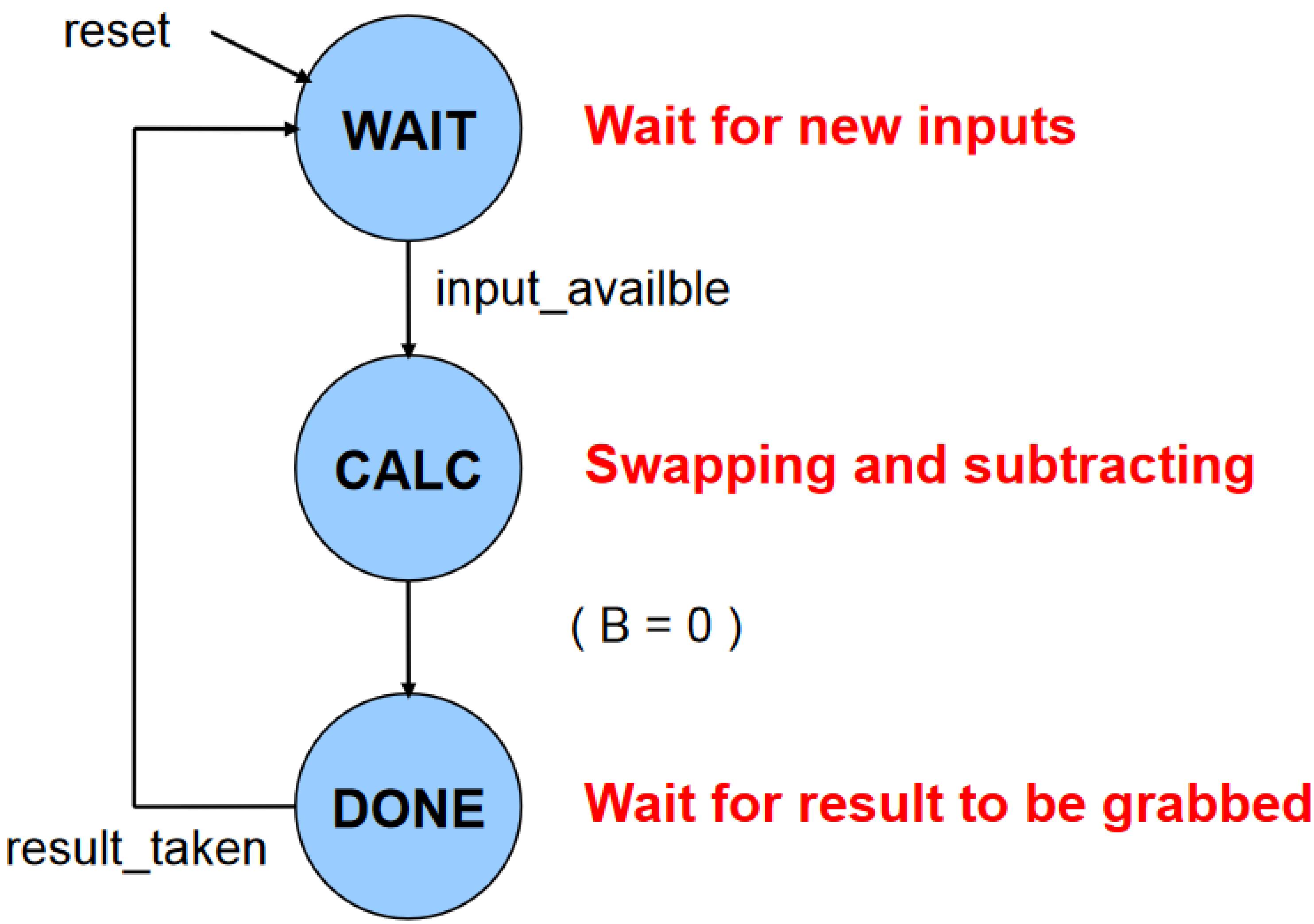
Implementing Datapath Module

```
wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_mux_out;
3inMUX#(W) A_mux
(   .in0 (operands_bits_A),
    .in1 (B),
    .in2 (sub_out),
    .sel (A_mux_sel),
    .out (A_mux_out) );

wire [W-1:0] A;

ED_FF#(W) A_ff // D flip flop
(              //  with enable
    .clk  (clk),
    .en_p (A_en),
    .d_p  (A_mux_out),
    .q_np (A) );
```

```
wire [W-1:0] B_mux_out;
2inMUX#(W) B_mux (
        .in0 (operands_bits_B),
        .in1 (A),
        .sel (B_mux_sel),
        .out (B_mux_out) );
ED_FF#(W) B_ff (
        .clk  (clk),
        .en_p (B_en),
        .d_p  (B_mux_out),
        .q_np (B) );
2inEQ#(W) B_EQ_0 (
        .in0(B),
        .in1(W'd0),
        .out(B_zero) );
LessThan#(W) lt (
        .in0(A),
        .in0(B),
        .out(A_lt_B) );
Subtractor#(W) sub (
        .in0(A),
        in1(B),
        .out(sub_out) );
assign result_bits_data = A;
```

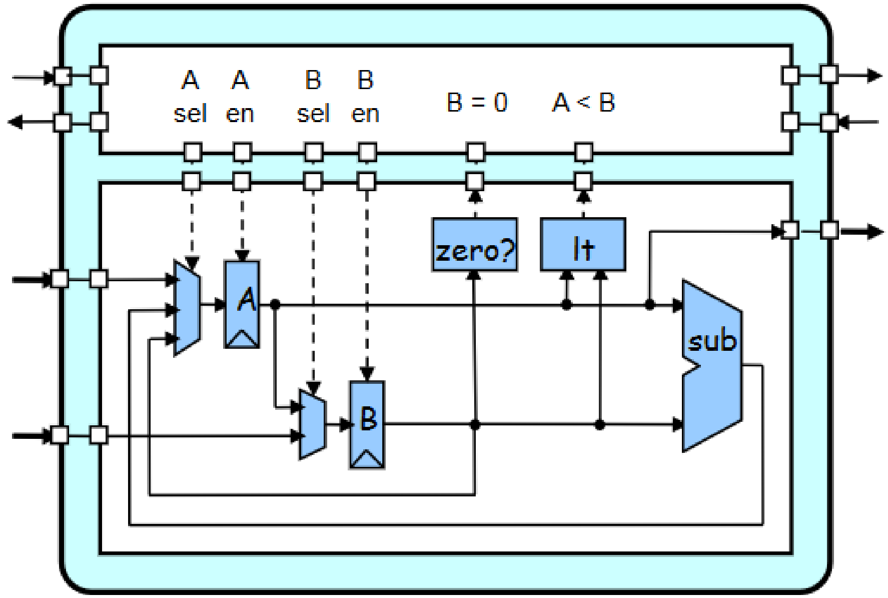
Remember: Functionality only in “leaf” modules!

State Machine for Control



Implementing control module

```
module gcdControlUnit (
   input       clk,
   input      reset,


   // Data signals
   input      input_available,
   output  reg     result_rdy,
   input     result_taken,


   // Control signals (ctrl->dpath)
   output  reg         A_en,
   output  reg        B_en,
   output  reg [1:0] A_mux_sel,
   output  reg         B_mux_sel,


   // Control signals (dpath->ctrl)
   input          B_zero,
   input          A_lt_B
);
```



State Update Logic

- Remember: keep state update, next state calculation, and output logic separated

```
// local params are scoped constants
localparam WAIT = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;

reg  [1:0] state_next;
wire [1:0] state;

RD_FF state_ff ( // flip flop with reset
    .clk (clk),
    .reset_p (reset),
    .d_p (state_next),
    .q_np (state) );
```

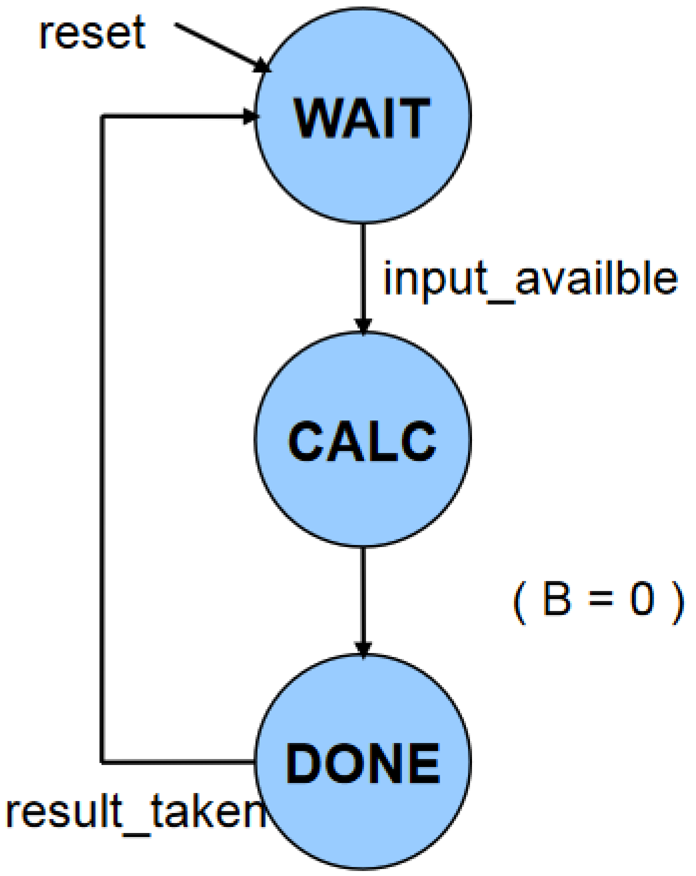
Output Signal Logic

```
always@(*)
begin // Default control signals
  A_mux_sel    = A_MUX_SEL_X;
  A_en         = 1'b0;
  B_mux_sel    = B_MUX_SEL_X;
  B_en         = 1'b0;

  result_rdy   = 1'b0;
  case ( state )
     WAIT :
      ...
     CALC :
      ...
     DONE :
      ...
  endcase
end
```

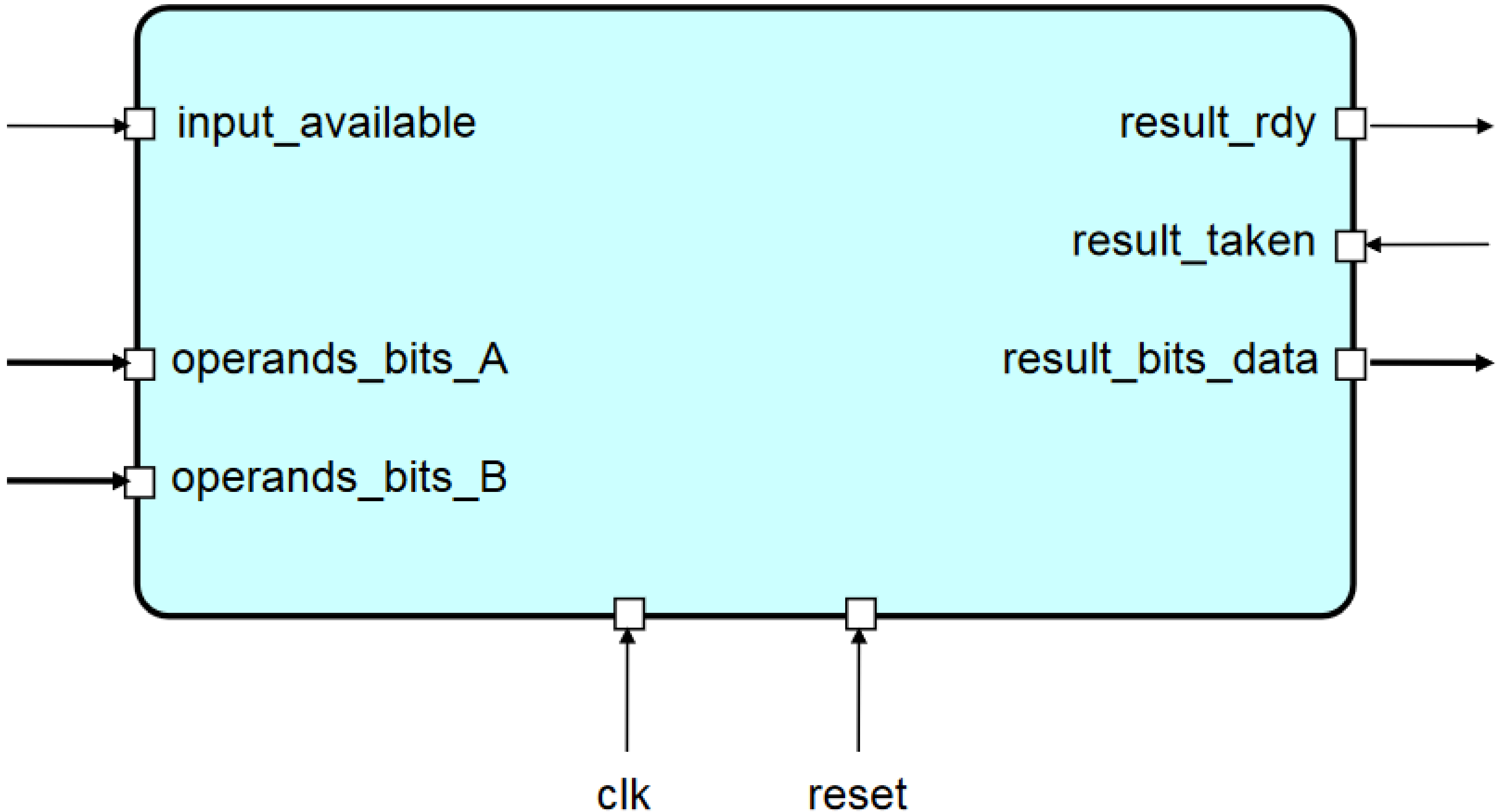
```
WAIT :
 begin
   A_mux_sel     = A_MUX_SEL_IN;
   A_en          = 1'b1;
   B_mux_sel     = B_MUX_SEL_IN;
   B_en          = 1'b1;

 end
CALC :
 if ( A_lt_B )
   begin
          A_mux_sel = A_MUX_SEL_B;
          A_en      = 1'b1;
          B_mux_sel = B_MUX_SEL_A;
          B_en      = 1'b1;
   end
  else if ( !B_zero )
   begin
         A_mux_sel = A_MUX_SEL_SUB;
         A_en      = 1'b1;
   end

DONE :
  result_rdy = 1'b1;
```

Next State Logic

```
always @(*)
begin
   // Default is to stay in
   // the same state
   state_next = state;
   case ( state )
     WAIT :
       if ( input_available )
          state_next = CALC;
     CALC : if ( B_zero )
          state_next = DONE;
     DONE : if ( result_taken )
          state_next = WAIT;
  endcase
end
```

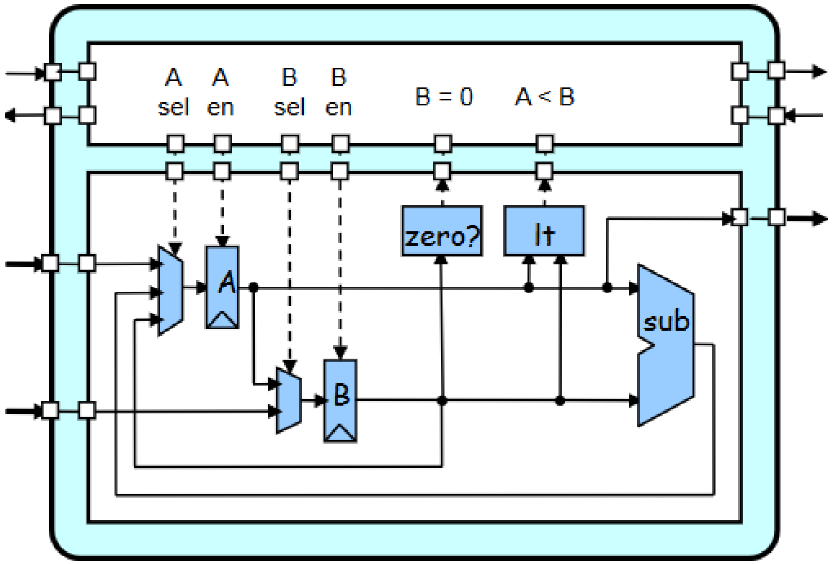


Next Step: Define Module Ports



Wire them together

```
module gcd#( parameter W = 16 )
(
   input      clk,
   // Data signals
   input  [W-1:0] operands_bits_A,
   input  [W-1:0] operands_bits_B,
   output [W-1:0] result_bits_data,
   // Control signals
   input input_available,
   input reset,
   output result_rdy,
   input result_taken
);

wire[1:0] A_sel;
wire A_en;
...
...
```




```
gcdDatapath#(16)  datapath (
   .operand_bits_A(operands_bits_A),
   ...
   .A_mux_sel(A_sel),
   ...
)
gcdControl#(16) control (
   .A_sel(A_sel),
   ...
)
```

References

- Mentioned in the first slide ☺

Thank you