



Verilog Tutorial, Part Deux

By Sat Garcia



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
 - <http://csg.csail.mit.edu/6.375/>



Designing a GCD Calculator

■ Euclid's Algorithm for GCD (in C):

```
int GCD( int inA, int inB)
{
    int done = 0;
    int A = inA;
    int B = inB;
    while ( !done )
    {
        if ( A < B ) // if A < B, swap values
        {
            swap = A;
            A = B;
            B = swap;
        }
        else if ( B != 0 ) // subtract as long as B isn't 0
            A = A - B;
        else
            done = 1;
    }
    return A;
}
```

How do we implement
this in hardware?

Adapted from Arvind and Asanovic's MIT 6.375 lecture

Take 1: Behavioral Verilog

```
module gcdGCDUnit_behav#( parameter W = 16 ) // parameterize for better reuse
(
  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 )
      swap = A;
      A = B;
      B = swap;
    else if ( B != 0 )
      A = A - B;
    else
      done = 1;
  end

  out = A;
end

endmodule
```

Adapted from Arvind and Asanovic's MIT 6.375 lecture

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 HW structures

```
module gcdGCDUnit_behav#( 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 )
        swap = A;
        A = B;
        B = swap;
      else if ( B != 0 )
        A = A - B;
      else
        done = 1;
    end

    out = A;
  end
endmodule
```

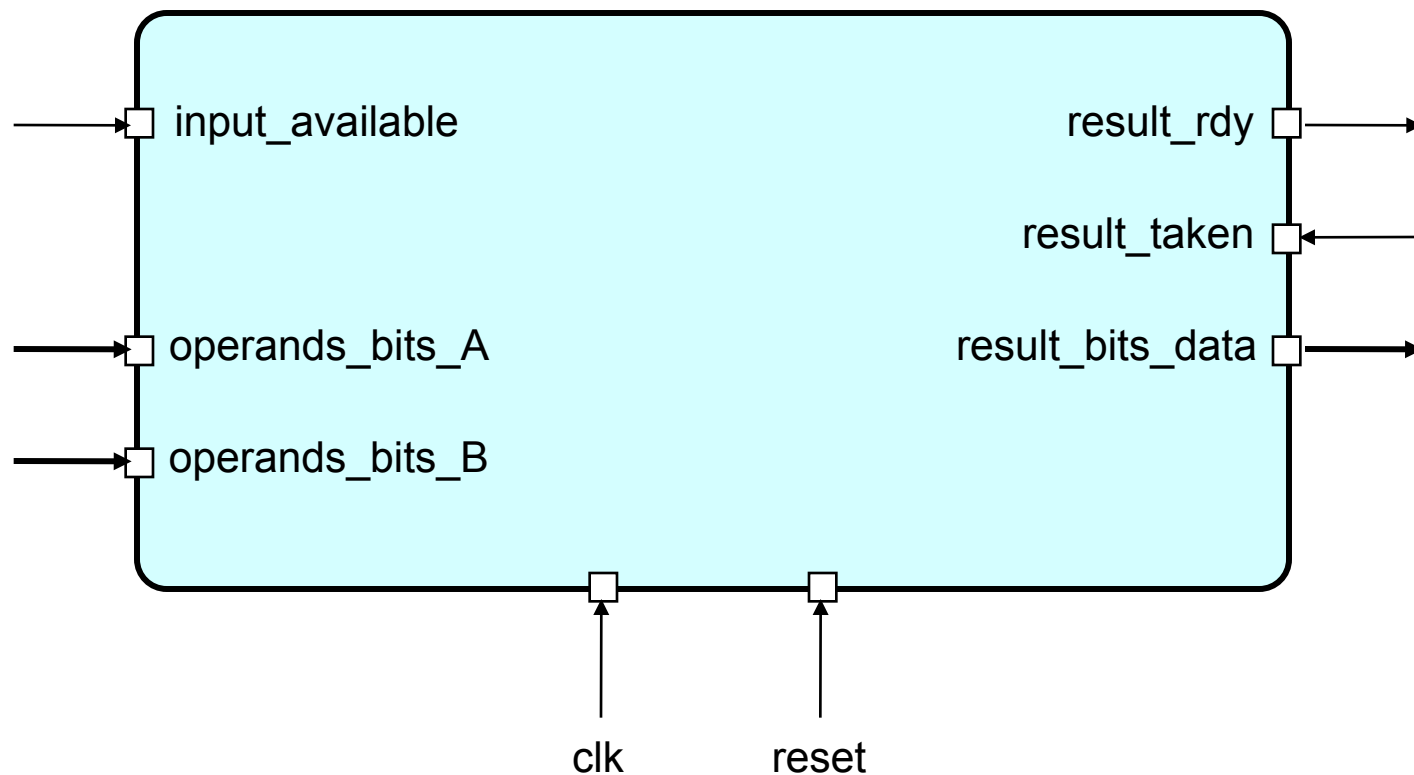
State → Registers

Less than comparator

Equality Comparator

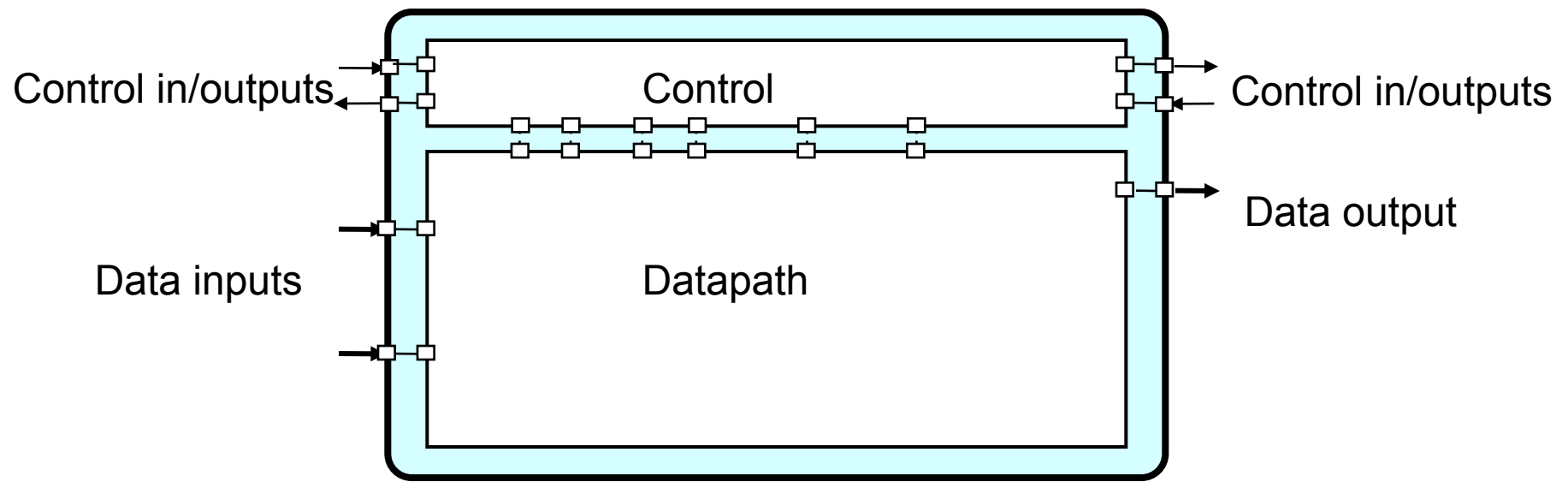
Subtractor

Next step: define module ports

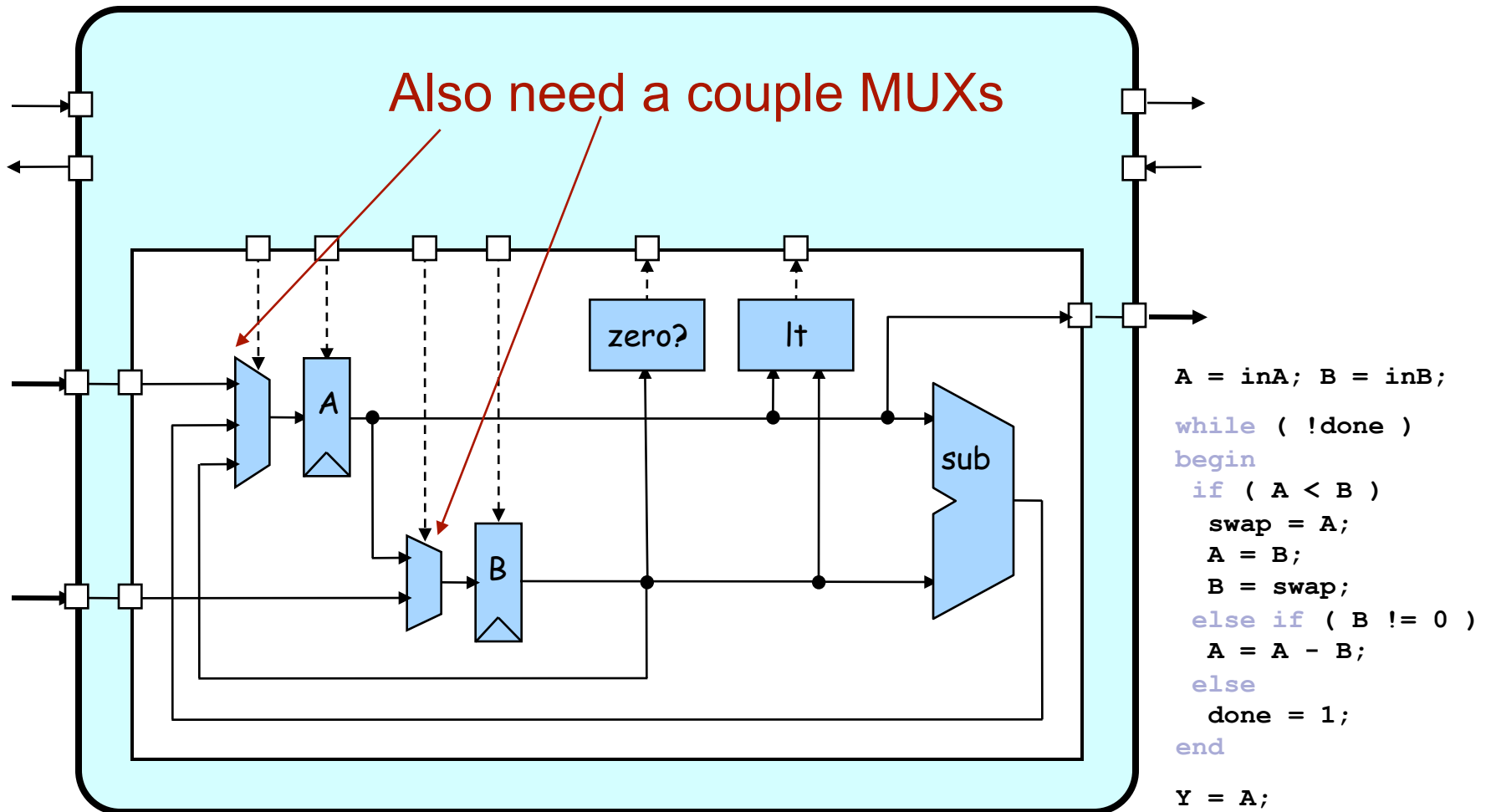


Implementing the modules

- Two step process:
 1. Define datapath
 2. Define control/control path

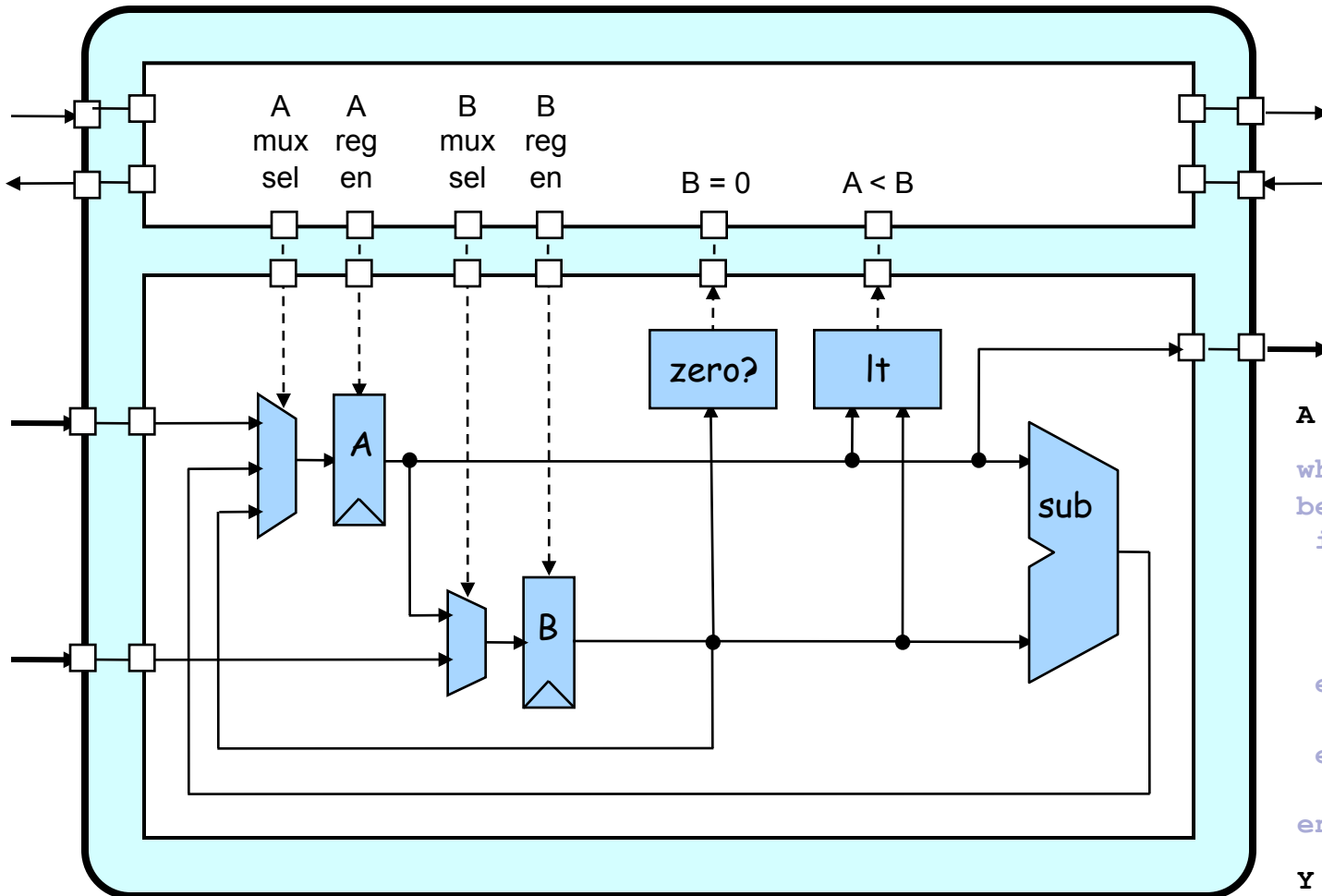


Developing the datapath



Adapted from Arvind and Asanovic's MIT 6.375 lecture

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;

```

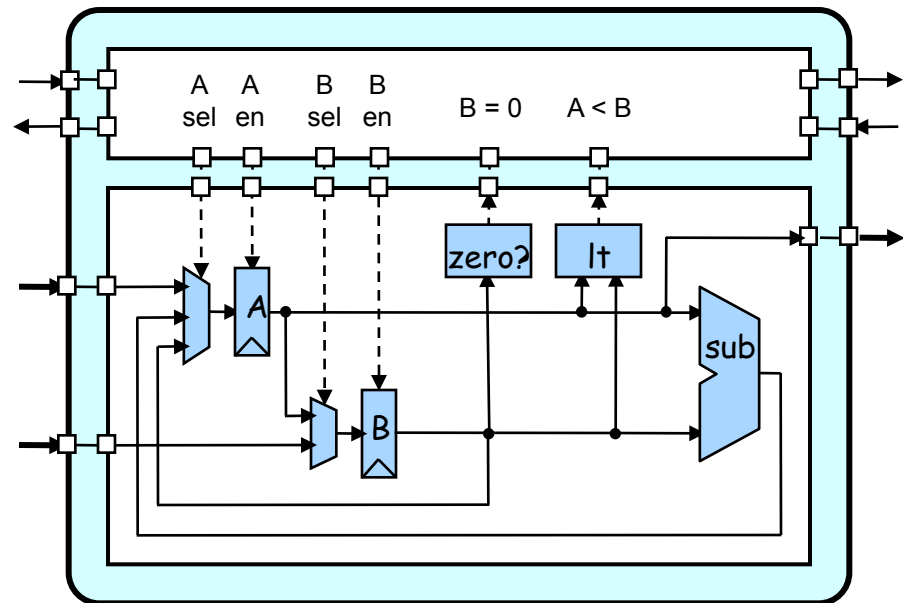
Datapath module

```
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)
);
```

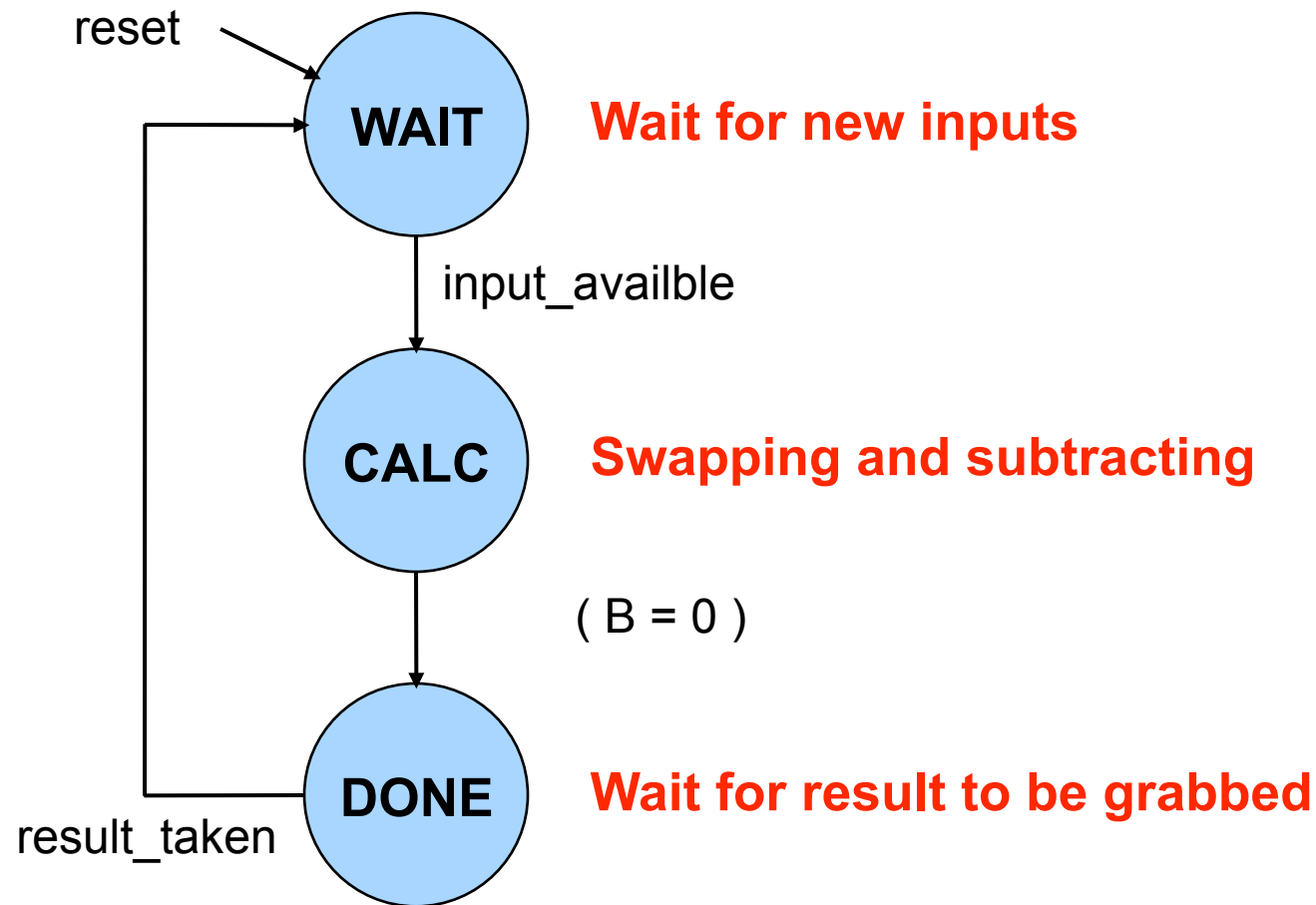
```
wire [W-1:0] B_mux_out;
2inMUX#(W) B_mux
(
    .in0 (operands_bits_B),
    .in1 (A),
    .sel (B_mux_sel),
    .out (B_mux_out)
);

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



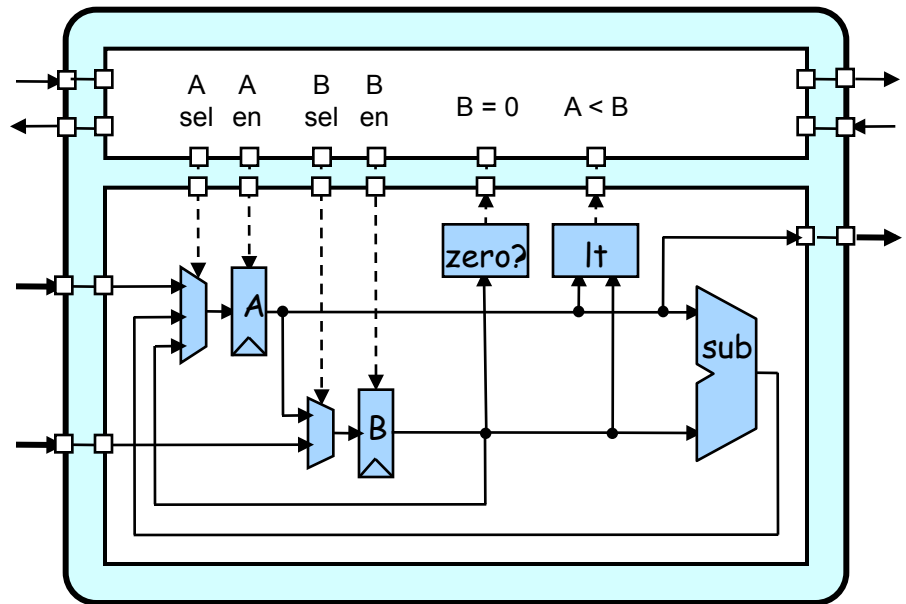
Implementing control module

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

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

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

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



Remember: Keep next state (combin.), state update (seq.), and output logic separated!



State update logic

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

```
localparam WAIT = 2'd0; // local params are scoped constants
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 signals logic

```
reg [6:0] cs;

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;
    input_available = 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;
    input_available = 1'b1;
end

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

DONE :
    result_rdy = 1'b1;
```

Adapted from Arvind and Asanovic's MIT 6.375 lecture

Next state logic

```
always @(*)
begin

    // Default is to stay in the same state
    state_next = state;

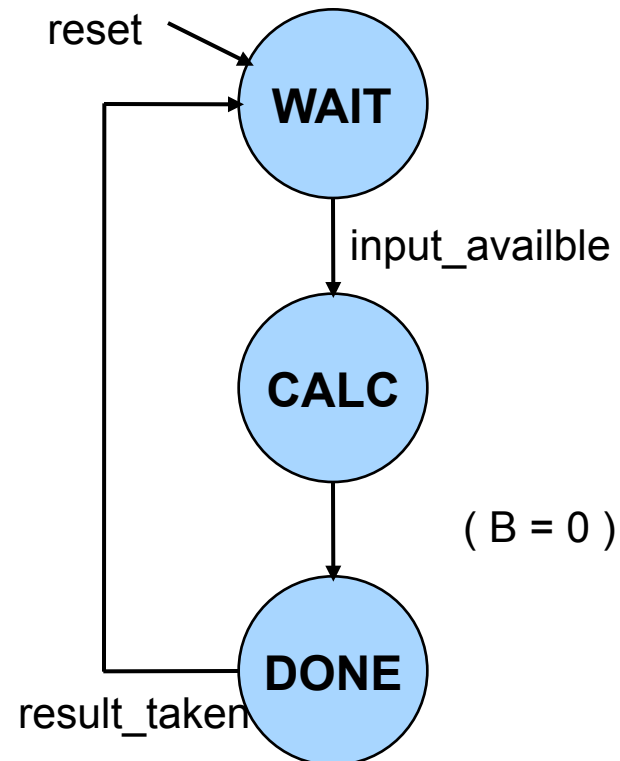
    case ( state )

        WAIT :
            if ( input_available )
                state_next = CALC;

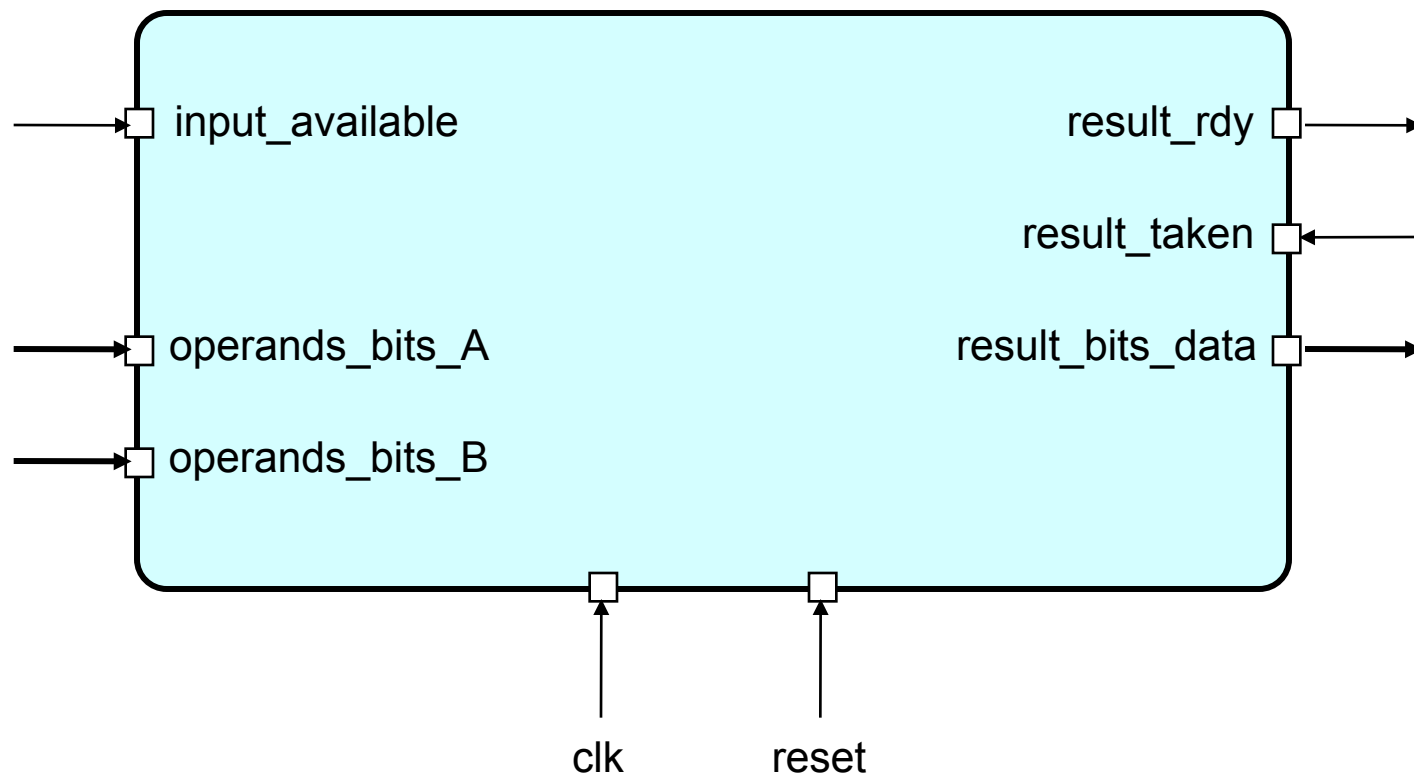
        CALC :
            if ( B_zero )
                state_next = DONE;

        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;
    ...

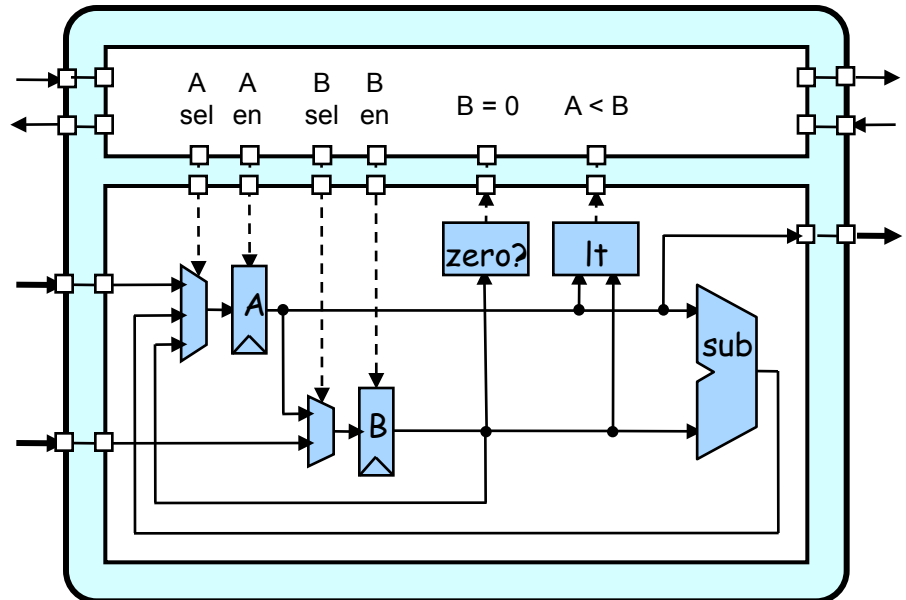
```

```

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

gcdControl#(16) control (
    .A_sel(A_sel),
    ...
)

```





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

);
```



But wait, there's more

- Build test bench a la lab 1
- Test thoroughly, debug
- Measure cycle time, optimize, etc.