**ELC 5313: ADVANCED Computer Architecture**

**Fall 2025**

**Lab 2: Adder with Reservation Stations**

**Introduction**

In this lab, you will implement a small portion of Tomasulo’s algorithm: an adder with reservation stations (RS). Reservation stations allow instructions to wait until all operands are available before executing, supporting out-of-order execution.

Unlike a simple ALU that immediately processes two inputs, this design:

* Supports multiple instructions waiting in reservation stations.
* Handles invalid operands (representing future results from other RS).
* Communicates results via a Common Data Bus (CDB).

Key concepts:

* Reservation Stations (RS): Hold operations until operands are ready.
* CDB (Common Data Bus): Transmits results from execution units to waiting instructions.
* Valid/Invalid Operands: Operands can be immediately available or pending from another RS.
* Tri-state buses: Allow multiple sources to safely share a common bus.

You will work with 32-bit signed operands and three reservation stations for the adder unit. The ALU supports addition, subtraction, and basic logic operations (AND, OR, XOR, NOT).

**Scope**

This lab introduces:

* Implementing reservation station logic in SystemVerilog.
* Handling instruction issue, execution, and CDB write-back.
* Using a provided testbench to simulate multiple instructions.
* Observing timing diagrams in Vivado to confirm correct behavior.

You will:

1. Complete the adder.sv module with reservation station logic.
2. Run simulations using the provided testbench.
3. Observe instruction flow: issue → execute → write-back.

**Prerequisites**

* mod3counter.sv must already exist in your project.
* Both adder.sv and mod3counter.sv must be included in the same project to handle station cycling.

**Provided Files**

* adder.sv – template for the adders module.
* mod3counter.sv – used to cycle RS in a round-robin manner.
* adders\_test.sv – testbench to verify your implementation. Students must use this testbench.

**Module: adders.sv**

This module should implement the following:

1. Accept arithmetic operation requests:  
   alu\_add, alu\_sub, alu\_and, alu\_or, alu\_not, alu\_xor.
2. Assign each operation to a reservation station (RS) if one is available.
3. Track which RSs are busy and which are waiting for operands.
4. Execute operations once operands are ready.
5. Communicate results back via the CDB.

**Reservation Station (RS) Logic**

**1. RS Slots**

* 3 stations: adder\_1, adder\_2, adder\_3.
* Each RS tracks:
  + Vj and Vk: operand values (if ready).
  + Qj and Qk: producing RS of operands (if not ready).
  + Busy: whether the station currently holds an operation.

**2. Issuing Operations**

On issue signal:

1. Check for the first available RS (Busy = 0).
2. Store operation and operands in the selected RS:
   * If operand is valid (A\_invalid = 0 or B\_invalid = 0), store in Vj/Vk and mark Qj/Qk as valid.
   * If operand is not valid (A\_invalid = 1 or B\_invalid = 1), store the producing RS in Qj/Qk.
3. Set issued to indicate which RS was used.

If all RSs are busy, set error = all\_rs\_busy.

**3. Execution**

* Each clock cycle, check if a RS has valid operands (Qj = Qk = valid) and if the adder unit is free.
* If ready, execute the operation (alu\_add, alu\_sub, etc.) and mark the unit as busy.
* Use #delay to simulate computation time.
* Output the result on CDB\_data and indicate the RS that produced it (CDB\_source).

**4. CDB Update**

* On the negative edge of the clock, update all RSs waiting for a value:
  + If CDB\_source matches Qj or Qk, copy CDB\_data to Vj or Vk and mark the operand as valid.
* This allows stations to dynamically receive operands as other operations complete.

**5. Priority & Starvation Prevention**

* Cycle RSs using mod3counter in a round-robin order:  
  Priority\_Station → Second\_Station → Last\_Station.
* Execute operations in this order to ensure fairness.

Start from this file adder.sv located in ELC5313-Lab:

module adder(

input wire clock,

input wire issue,

input wire signed [31:0] A, B,

input wire A\_invalid, B\_invalid,

input wire [5:0] opcode,

input wire CDB\_xmit,

inout wire signed [31:0] CDB\_data,

inout wire [5:0] CDB\_source,

inout wire CDB\_write,

output reg CDB\_rts,

output wire available,

output wire [5:0] RS\_available,

output reg [5:0] issued,

output wire [5:0] RS\_executing,

output reg error

);

// Constants

// Disconnected buses have high impedance

parameter disconnected = 32'bz;

// Calculation delay

parameter delay = 20;

// Updating delay

parameter update\_delay = 5;

// No unit is calculating

parameter none = 2'b00;

// Clear Qj/Qk, thus Vj/Vk valid

parameter valid = 6'b000000;

// Error code

parameter no\_error = 1'b0;

parameter all\_rs\_busy = 1'b1;

// Command not issued

parameter not\_issued = 6'b000000;

// Status of reservation station or ALU

parameter not\_busy = 1'b0;

parameter in\_use = 1'b1;

// Is the value ready to send or not

parameter ready = 1'b1;

parameter not\_ready = 1'b0;

// Used to clear a data value

parameter clear = 0;

// Reservation station

parameter no\_rs = 6'b000000;

parameter adder\_1 = 6'b000001;

parameter adder\_2 = 6'b000010;

parameter adder\_3 = 6'b000011;

// Operation codes used by ALU

parameter alu\_add = 3'b000;

parameter alu\_sub = 3'b001;

parameter alu\_or = 3'b100;

parameter alu\_and = 3'b101;

parameter alu\_not = 3'b110;

parameter alu\_xor = 3'b111;

// Internal registers

reg signed [31:0] CDB\_data\_out;

reg [5:0] CDB\_source\_out;

reg CDB\_write\_out;

reg [5:0] operation [2:0];

reg [5:0] Qj [2:0], Qk [2:0];

reg signed [31:0] Vj [2:0], Vk [2:0];

reg Busy [2:0];

reg Unit\_Busy;

reg [1:0] adder\_calculating;

reg [5:0] RS\_num\_of [2:0];

reg [1:0] Priority\_Station;

wire [1:0] Second\_Station;

wire [1:0] Last\_Station;

wire [5:0] RS\_availability\_of\_Second\_or\_Last, RS\_availability\_of\_Last;

// Tri-state assignments for CDB connections

assign CDB\_data = CDB\_xmit ? CDB\_data\_out : disconnected;

assign CDB\_source = CDB\_xmit ? CDB\_source\_out : disconnected;

assign CDB\_write = CDB\_xmit ? CDB\_write\_out : disconnected;

// Logic to indicate that at least one RS is available

assign available = ~(Busy[0] & Busy[1] & Busy[2]);

// assign RS\_availability\_of\_Last =

// finish this line

// assign RS\_availability\_of\_Second\_or\_Last =

// finish this line

assign RS\_available = ~Busy[Priority\_Station] ? RS\_num\_of[Priority\_Station] : RS\_availability\_of\_Second\_or\_Last;

// RS executing, to track the unit that is currently in use

// assign RS\_executing = // finish this line

// Cycle through the order of the stations to prevent starvation

mod3counter mod3count1(.num(Priority\_Station), .mod3num(Second\_Station));

// calculate the last station

// Initialize signals

initial begin

CDB\_rts = not\_ready;

CDB\_data\_out = clear;

CDB\_source\_out = no\_rs;

CDB\_write\_out = not\_ready;

Priority\_Station = no\_rs;

Unit\_Busy = not\_busy;

Busy[0] = not\_busy;

Busy[1] = not\_busy;

Busy[2] = not\_busy;

issued = not\_issued;

error = no\_error;

RS\_num\_of[0] = adder\_1;

RS\_num\_of[1] = adder\_2;

RS\_num\_of[2] = adder\_3;

adder\_calculating = none;

end

// Reset signals on each clock cycle

always @(posedge clock) begin

issued = not\_issued;

error = no\_error;

end

// Clean up when CDB transmission finishes

always @(negedge CDB\_xmit) begin

CDB\_rts = not\_ready;

// CDB\_write\_out = finish this line

// Unit\_Busy = finish this line

// Busy[adder\_calculating] = finish this line

// adder\_calculating = finish this line

end

// Handle execution each cycle

always @(posedge clock) begin

begin

// Each cycle handle execution

end

// Handle updates on the negative edge of clock

always @(negedge clock) begin

//handle updates

end

// Handle issue logic

always @(negedge clock) begin

//handle issue

end

endmodule

**Implementation Notes**

* Integration: Ensure adder.sv and mod3counter.sv are in the same project.
* Tri-state CDB: Only one RS can write at a time. Use conditional assignments for CDB\_data, CDB\_source, and CDB\_write.
* Error Handling: Reset issued and error at each clock cycle.
* Simulation Delays: Use #delay for adder computation time. Optionally, use #update\_delay for updating RS signals.

**Testing**

* Use the provided testbench (adder\_tb.sv) in ELC5313-Lab

module adder\_tb;

// Constants

parameter clock\_period = 10;

// ALU operation codes

parameter alu\_add = 3'b000;

parameter alu\_sub = 3'b001;

parameter alu\_or = 3'b100;

parameter alu\_and = 3'b101;

parameter alu\_not = 3'b110;

parameter alu\_xor = 3'b111;

// Inputs

reg clk;

reg issue\_command;

reg [5:0] command;

reg signed [31:0] A, B;

reg A\_invalid, B\_invalid;

reg CDB\_xmit;

// I/O for outputs

wire signed [31:0] data\_out;

wire [5:0] data\_from\_rs\_num;

wire data\_valid;

// Outputs

wire CDB\_rts;

wire available;

wire [5:0] issued\_to\_rs\_num, RS\_available, RS\_executing;

wire error;

// Clock generation and stimulus

initial begin

clk = 0;

issue\_command = 0;

command = 0;

A = 0;

B = 0;

A\_invalid = 0;

B\_invalid = 0;

CDB\_xmit = 0;

// Initial clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// First issue command (ALU add)

issue\_command = 1;

command = alu\_add;

A = 5;

B = 5;

A\_invalid = 0;

B\_invalid = 0;

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Second issue command (ALU sub)

issue\_command = 1;

command = alu\_sub;

A = 3;

B = 13;

A\_invalid = 1;

B\_invalid = 0;

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Third issue command (ALU and)

issue\_command = 1;

command = alu\_and;

A = 15;

B = 60;

A\_invalid = 0;

B\_invalid = 0;

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Fourth issue command (ALU add)

issue\_command = 1;

command = alu\_add;

A = 25;

B = 35;

A\_invalid = 0;

B\_invalid = 0;

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// No more issues

issue\_command = 0;

command = 0;

A = 0;

B = 0;

A\_invalid = 0;

B\_invalid = 0;

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Transmit CDB data

CDB\_xmit = 1;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Stop transmitting CDB data

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Transmit CDB data again

CDB\_xmit = 1;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Stop transmitting CDB data

CDB\_xmit = 0;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Transmit CDB data once more

CDB\_xmit = 1;

// Clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

// Stop transmitting CDB data

CDB\_xmit = 0;

// Final clock cycles

#clock\_period clk = ~clk;

#clock\_period clk = ~clk;

end

// Instantiate the adders module

adder my\_adders (

.clock(clk),

.issue(issue\_command),

.A(A),

.B(B),

.A\_invalid(A\_invalid),

.B\_invalid(B\_invalid),

.opcode(command),

.CDB\_xmit(CDB\_xmit),

.CDB\_data(data\_out),

.CDB\_source(data\_from\_rs\_num),

.CDB\_write(data\_valid),

.CDB\_rts(CDB\_rts),

.available(available),

.RS\_available(RS\_available),

.issued(issued\_to\_rs\_num),

.RS\_executing(RS\_executing),

.error(error)

);

endmodule

* Verify that:
  1. Multiple operations are issued correctly.
  2. Operations execute only when operands are ready.
  3. CDB outputs the correct results.
  4. RS allocation and priority cycling work correctly.
  5. The error flag is set if all RSs are busy.

**Deliverables**

1. adder.sv – completed module.
2. adders\_test.sv – testbench.
3. Screenshot of waveform simulation showing correct RS and CDB behavior.

**Grading Rubric**

* **Adder + RS implementation** (60 pts)
  + Correct handling of issue, execution, and CDB write-back.
* **Testbench correctness** (10 pts)
  + Easy points. Proper simulation of multiple instructions and CDB timing.
* **Successful simulation & waveform** (20 pts)
  + Timing diagram shows correct instruction progression and results. An example of what your waveform should look like is below.

A screenshot of a computer

AI-generated content may be incorrect.

* **Code clarity & comments** (10 pts)
  + Descriptive names and comments explaining design choices.

**Due Date**

**9/24/2025** (Submit individually via Canvas).