*Digital System Design Lab*

# CEL-442



*Class: BCE-6(A).*

### Enrollment no: 01-132182-024

**Lab 14**

**Background:**

This type of technique is used to increase the throughput of the computer system. An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously.

**Introduction:**

**Pipelining** organizes the execution of the **multiple instructions simultaneously**. Pipelining improves the **throughput** of the system. In pipelining the instruction is divided into the subtasks. Each subtask performs the dedicated task.

The output of the first pipeline becomes the input for the next pipeline. It is like a set of data processing unit connected in series to utilize processor up to its maximum.

**Procedure:**

1. In low level programming, we mostly manipulate data that is present in registers.
2. Registers are units of memory which are closely coupled with the processor core (ALU) and in most architectures the ALU operations are carried out on these registers and the results obtained are also stored in them.
3. Registers are a limited resource and efficient management of registers is a key issue in low level programming.
4. Processors must carry out a large number of functions utilizing the limited number of registers available.
5. Pipelining is a technique for overlapping operations during execution. Today this is a key feature that makes fast CPUs. Different types of pipeline are instruction pipeline, operation pipeline, and multi-issue pipelines. In this paper we are going to simulate the pipelining stages such as fetch, decode, execute, store separately.

**Task1: 4-bit Ripple carry adder using Pipelining:**

module ripple\_carry\_4\_bit(a, b, cin, sum, cout);

input [3:0] a,b;

input cin;

wire c1,c2,c3;

output [3:0] sum;

output cout;

full\_adder fa0(.a(a[0]), .b(b[0]),.cin(cin), .sum(sum[0]),.cout(c1));

full\_adder fa1(.a(a[1]), .b(b[1]), .cin(c1), .sum(sum[1]),.cout(c2));

full\_adder fa2(.a(a[2]), .b(b[2]), .cin(c2), .sum(sum[2]),.cout(c3));

full\_adder fa3(.a(a[3]), .b(b[3]), .cin(c3), .sum(sum[3]),.cout(cout));

endmodule

**Task2: 16-bit adder using Pipelining:**

module pipeline\_adder\_16bit(clk,reset,a, b, cin, sum, cout);

input clk,reset;

input [15:0] a,b;

input cin;

output [15:0] sum;

output cout;

wire [2:0] c;

wire [15:0] s;

reg [3:0] r1,r2,r3;

reg[3:0] s1,s2,s3,s4;

reg[3:0] t1,t2,t3,t4,t5;

reg[3:0] u1,u2,u3,u4,u5,u6;

reg[3:0]t1,t2,t3,t4,t5;

reg a1,a2,a3;

carry\_select\_adder\_4bit\_slice CLA1(

.a(a[3:0]),

.b(b[3:0]),

.cin(cin),

.sum(s[3:0]),

.cout(c[0]));

carry\_select\_adder\_4bit\_slice CLA2(

.a(s1),

.b(s2),

.cin(a1),

.sum(s[7:4]),

.cout(c[1]));

carry\_select\_adder\_4bit\_slice CLA3(

.a(t3),

.b(t4),

.cin(a2),

.sum(s[11:8]),

.cout(c[2]));

carry\_select\_adder\_4bit\_slice CLA4(

.a(u5),

.b(u6),

.cin(a3),

.sum(sum[15:12]),

.cout(cout));

always @(posedge clk or posedge reset)

begin

if (reset)

begin

r1<=0;r2<=0;r3<=0;

a1<=0;a2<=0;a3<=0;

s1<=0;s2<=0;s3<=0;s4<=0;

t1<=0;t2<=0;t3<=0;t4<=0;t5<=0;

u1<=0;u2<=0;u3<=0;u4<=0;u5<=0;u6<=0;

end

else

begin

//1st level

r1<=s[3:0];

r2<=r1;

r3<=r1;

//2nd Level

s1<=a[7:4];

s2<=b[7:4];

a1<=c[0];

s3<=s[7:4];

s4<=s3;

//3rd Level

t1<=a[11:8];

t2<=b[11:8];

t3<=t1;

t4<=t2;

a2<=c[1];

t5<=s[11:8];

//4th Level

u1<=a[15:12];

u2<=b[15:12];

u3<=u1;

u4<=u2;

u5<=u3;

u6<=u4;

a3<=c[2];

end

end

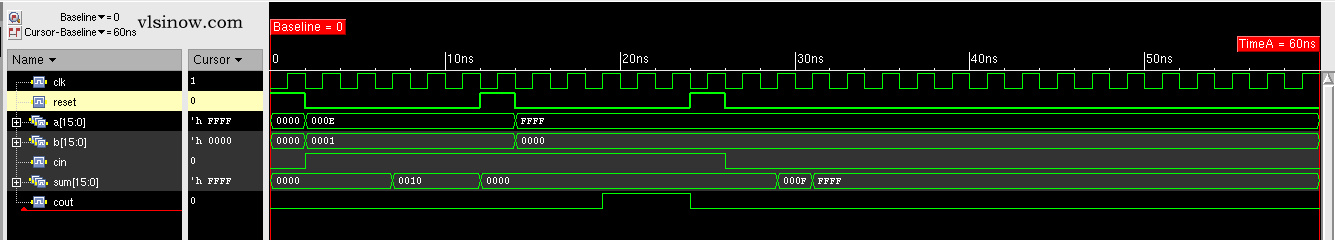
assign sum[3:0]=r3;

assign sum[7:4]=s4;

assign sum[11:8]=t5;

endmodule

**OUTPUT:**

****

**Conclusion:**

By doing this practical I learn about PI and its working and how to implement its Verilog code. And doing its Verilog code and verify it.and also observed how circuitry becomes more efficient using Pipelining.