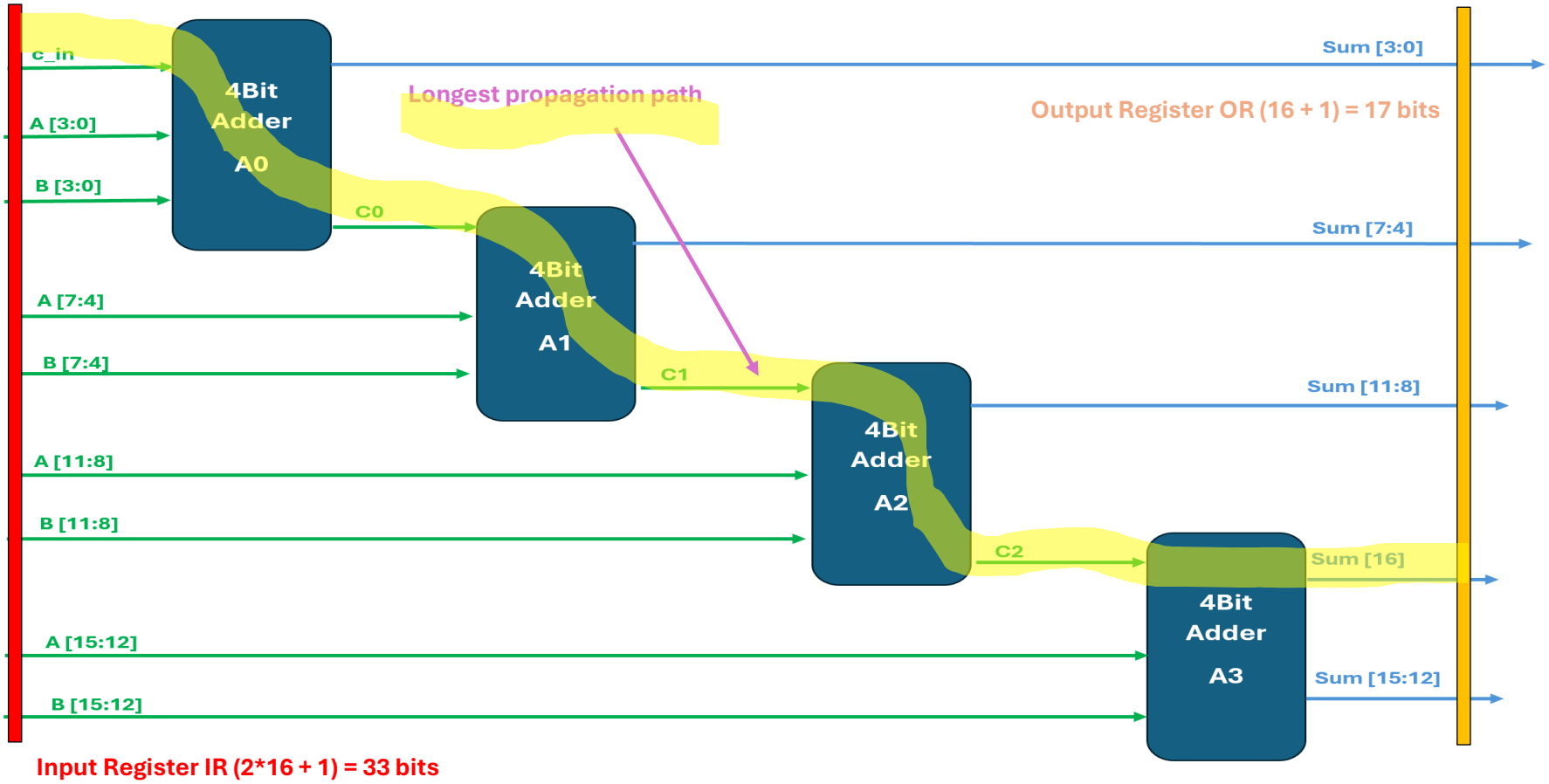
Pipelined Adder Explanation

In the earlier lessons for EE 417, we talked about the combinational design of adders. We started with the hardware design of a simple full adder that added single bits (a, b, and c_in) and the output were two single bits (sum and carry_out) which could be concatenated into sum [1:0]. This building block of a full adder was instantiated 4 times and cascaded to give a 4-bit adder. The 4bit adder can be again used as a building block to create adder with higher bits. (8bit, 16bit, 32bit adders … etc).

Now, we are using these adders in a clocked module, where the inputs A, B, and c_in are varying at a specific rate, and the calculations need to be completed fast enough to to have the output ready before the input data changes. Measures of the longest propagation delay need to be completed. In the 16bit adder given below for example, the output that takes the longest time to calculate is Sum [16] or the final output carry. The contribution or effect of A[0], b[0], and c_in may propagate all the way to the final calculation of Sum [16]. The clock period may be long enough to accommodate the propagation delay through the longest path. The input register guarantees that all the input signals are updated at the same time at the positive edge of the synchronizing clock.

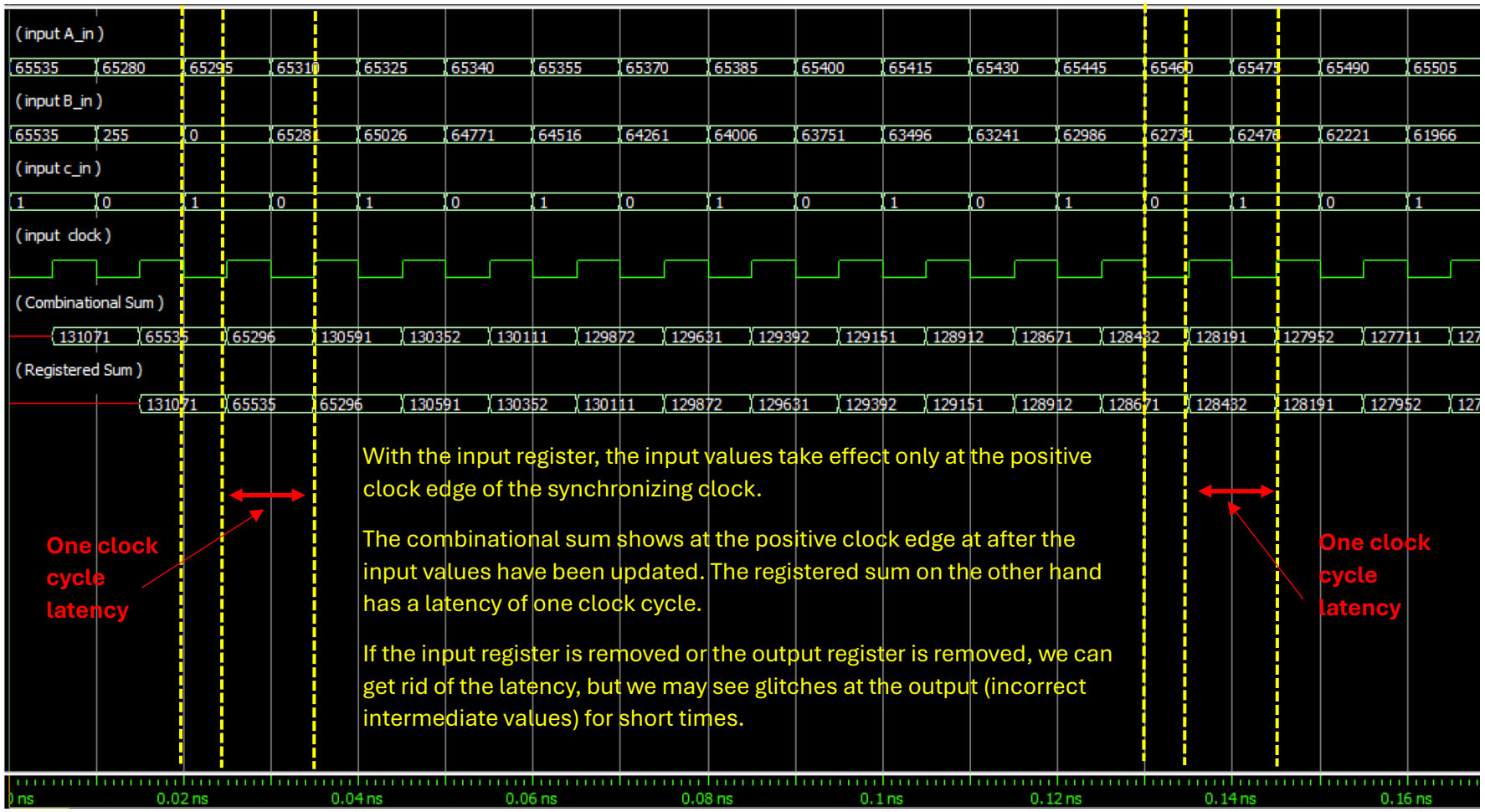


```
module NoPipeline_adder ( A_in, B_in, c_in, clk, sum_out);

input      [15:0]    A_in, B_in;    // 16 bit input words
input                c_in;          // carry in
input                clk;
output reg [16:0]    sum_out;       // including the carry_out accounting for maximum A_in and B_in.

wire       [16:0]    comb_add;
reg        [32:0]    IR;            // input register

assign  comb_add = IR[32:17] + IR[16:1] + IR[0];   // A short way to describe the addition

always @ (posedge clk)
begin
IR       <=  {A_in[15:0],B_in[15:0],c_in};
sum_out  <=  comb_add;                          // Registered output sum
end

endmodule
```

Combinational logic

Input and output registers updated only at positive clock edges

```
module Pipeline_tb ();

reg  [15:0]    A_in, B_in;      // 16 bit input words
reg            c_in;            // carry in
reg            clk;
wire [16:0]    Sum_out;            // No intermediate registers

NoPipeline_adder UUT ( A_in, B_in, c_in, clk, Sum_out);

wire [16:0]    Combinational_SUM;
assign         Combinational_SUM = UUT.M1.comb_add;

initial
begin
clk = 0;
forever  begin
#5 clk = ~clk;    end
end

initial
begin
     A_in = 16'hFF_FF;    B_in = 16'hFF_FF;    c_in = 1'b1;                    // Maximum possible sum
#10  A_in = 16'hFF_00;    B_in = 16'h00_FF;    c_in = 1'b0;
forever  begin
#10  A_in = A_in + 16'h000F;  B_in = B_in - 16'h00FF;     c_in = ~ c_in;      // random input values
end
end

endmodule
```

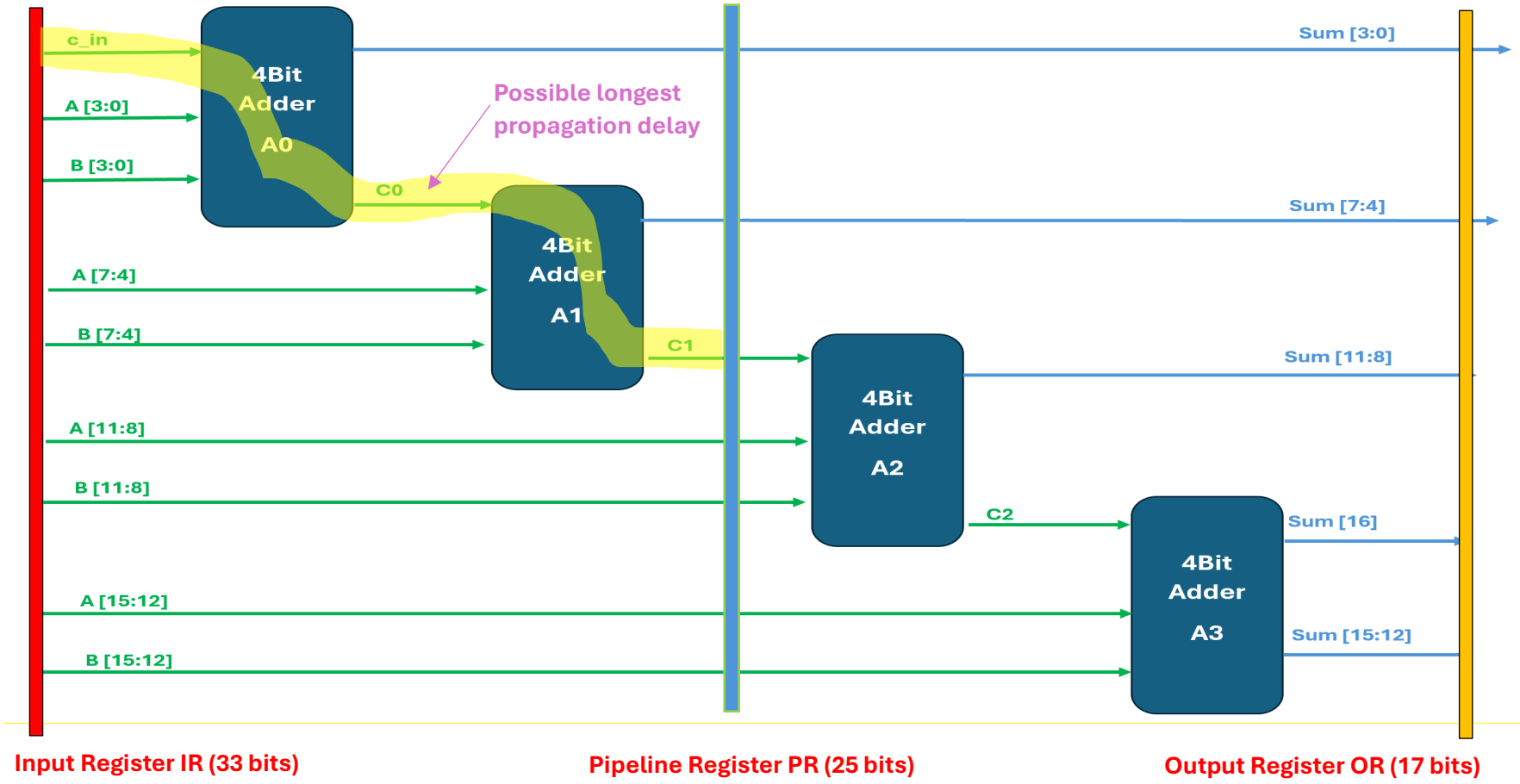


Simulation results given in unsigned decimal values. The sum has been verified for correct adder functionality. Instead of cascading 4Bit adders, we used the addition using the assign operator for the combinational logic. The registered output shows a latency of one clock cycle for all the inputs. The throughput is at every clock cycle with the change in the input values.

Adding One Intermediate Pipelining Register (cut-set):

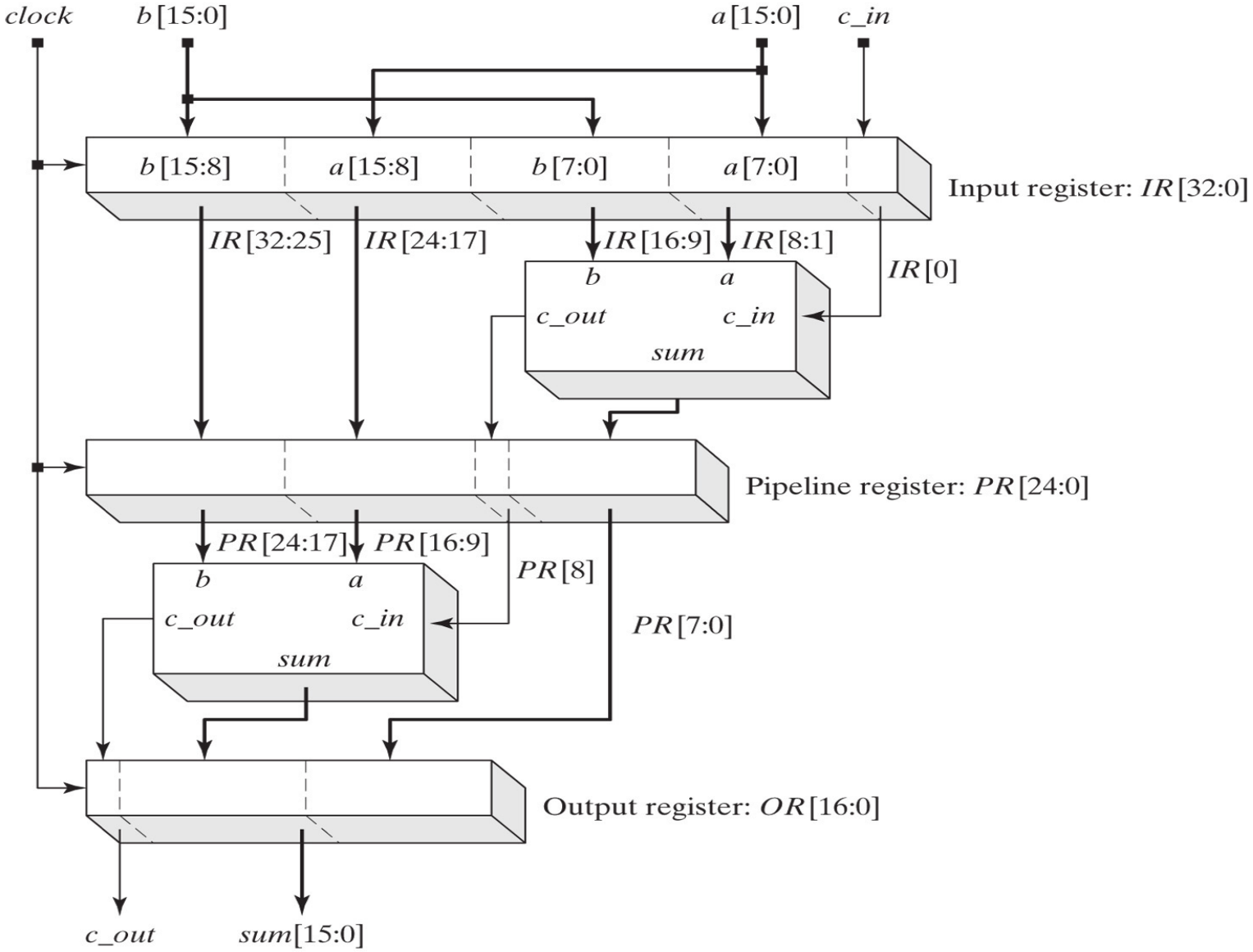
Adding one intermediate cut-set with the Pipeline Register shown below eases the restriction on the clock frequency. Now, the clock period needs to accommodate a shorter propagation delay path. The longest propagation delay is either between IR and PR or between PR and OR.

Notice where the cut-set (PR register) is positioned. The cut-set does not allow any direct path of signals between IR and OR. Any signal has to path through the added intermediate Pipeline register. This ensures data coherence (The aligned A, B, and c inputs are still the ones added together, but just over two clock cycles instead of one). This setup calculates the addition over the 8 LSBits first, and then over the 8MSBits carrying over the carry c1, and passing the sum [7:0] for data coherency.



```
module Pipeline_adder_8 ( A_in, B_in, c_in, clk, sum_out);

input  [15:0]    A_in, B_in;      // 16 bit input words
input            c_in;            // carry in
input            clk;
output [16:0]    sum_out;         // including the carry_out accounting for maximum A_in and B_in.

// Internal Registers:

reg [32:0]  IR;    // Input Register    = input words (2*16 = 32) and carry_in (1)
reg [24:0]  PR;    // Pipeline Register = the sum of LSBs & carry_in (8) + 2 MSBs (16) + carry_LSB (1)
reg [16:0]  OR;    // output register   = the total sum (16) + carry_out (1)

assign sum_out = OR;

always @ (posedge clk)
begin

// Load Input Register with new data words:
IR[0]      <=  c_in;           // carry in
IR[ 8: 1]  <=  A_in [ 7:0];    // Least significant Byte of A_in;
IR[16: 9]  <=  B_in [ 7:0];    // Least significant Byte of B_in;
IR[24:17]  <=  A_in [15:8];    // Most  significant Byte of A_in;
IR[32:25]  <=  B_in [15:8];    // Most  significant Byte of B_in;

// Load Pipeline Register:
PR[ 8: 0]  <= IR[16:9] + IR[8:1] + IR[0];    // Adding LSBs and the carry_in of previous entries
PR[24: 9]  <= IR[32:17];                     // passing the MSBs for data coherency

// loaad the Output Register:
OR[ 7: 0]  <= PR[ 7: 0];                     // passing the LSBs sum from PR as is
OR[16: 8]  <= PR[24:17] + PR[16:9] + PR[8];  // Adding MSBs and the carry_out of the LSB addition

end
```

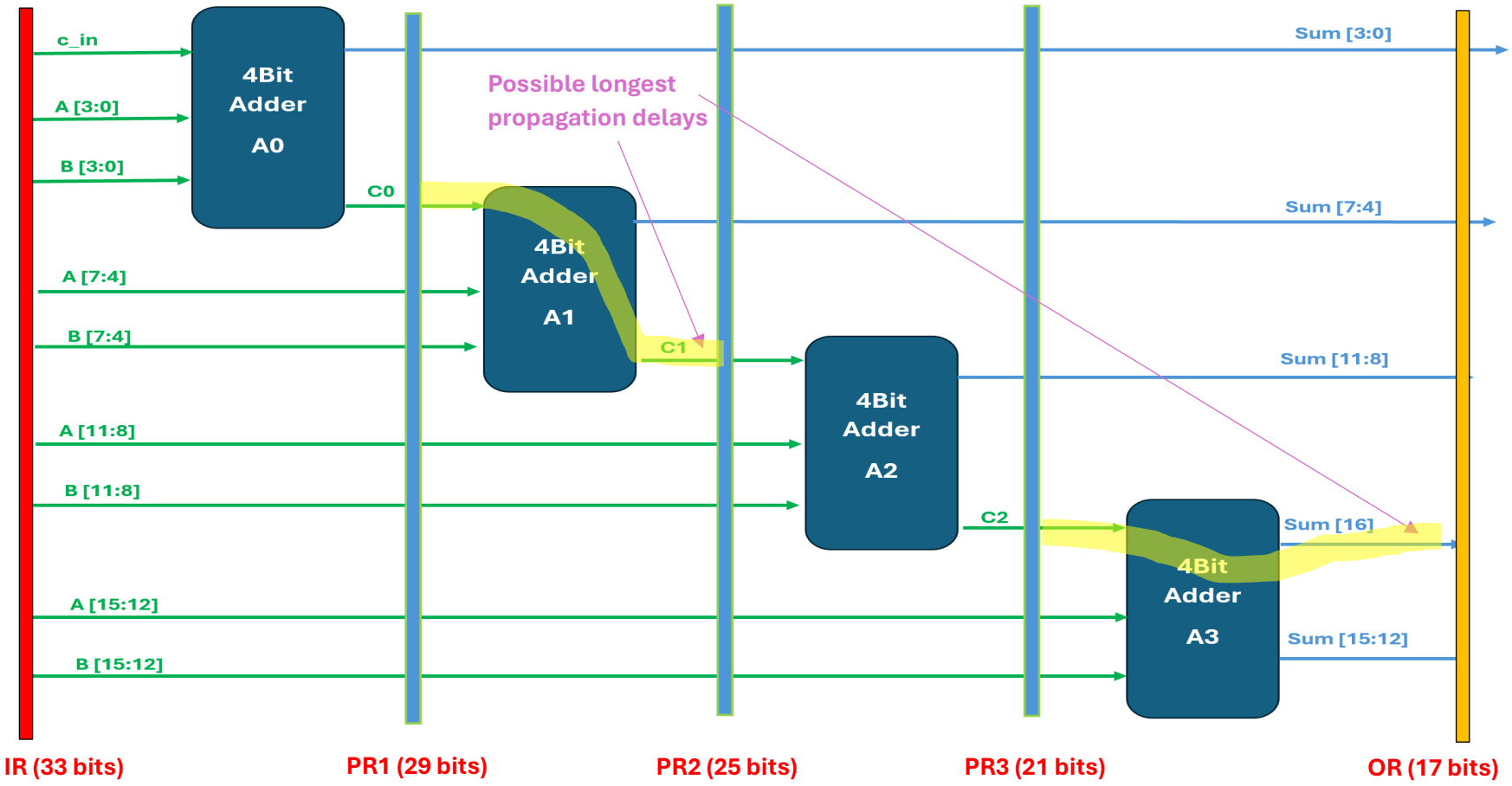
This code can be visualized using the register block diagram from the textbook (showing below)


clock
b[15:0]
a[15:0]
c_in
b[15:8]
a[15:8]
b[7:0]
a[7:0]
Input register: IR[32:0]
IR[32:25]
IR[24:17]
IR[16:9]
IR[8:1]
IR[0]
b
a
c_out
c_in
sum
Pipeline register: PR[24:0]
PR[24:17]
PR[16:9]
PR[8]
PR[7:0]
b
a
c_out
c_in
sum
Output register: OR[16:0]
c_out
sum[15:0]

Adding Three Intermediate Pipelining Registers (cut-sets):

Adding three intermediate cut-sets with the Pipeline Registers shown below eases the restriction on the clock frequency even more. Now, the clock period needs to accommodate much shorter propagation delay paths. The longest propagation delay is either between IR and PR1, between PR1 and PR2, between PR2 and PR3, or between PR3 and OR. The clock cycle can be much shorter, allowing the clock frequency to be maximized.

Notice where the cut-sets (PR registers) are positioned. Each cut-set block any direct path of signals between IR and OR. Any signal has to path through all the intermediate Pipeline registers. This ensures data coherency. This setup calculates the addition over 4 bits at a time carrying over the carry bits from the lower 4 bits to the next higher 4 bits. Notice how the propagation delays between consecutive registers are shorter than the previous cases.



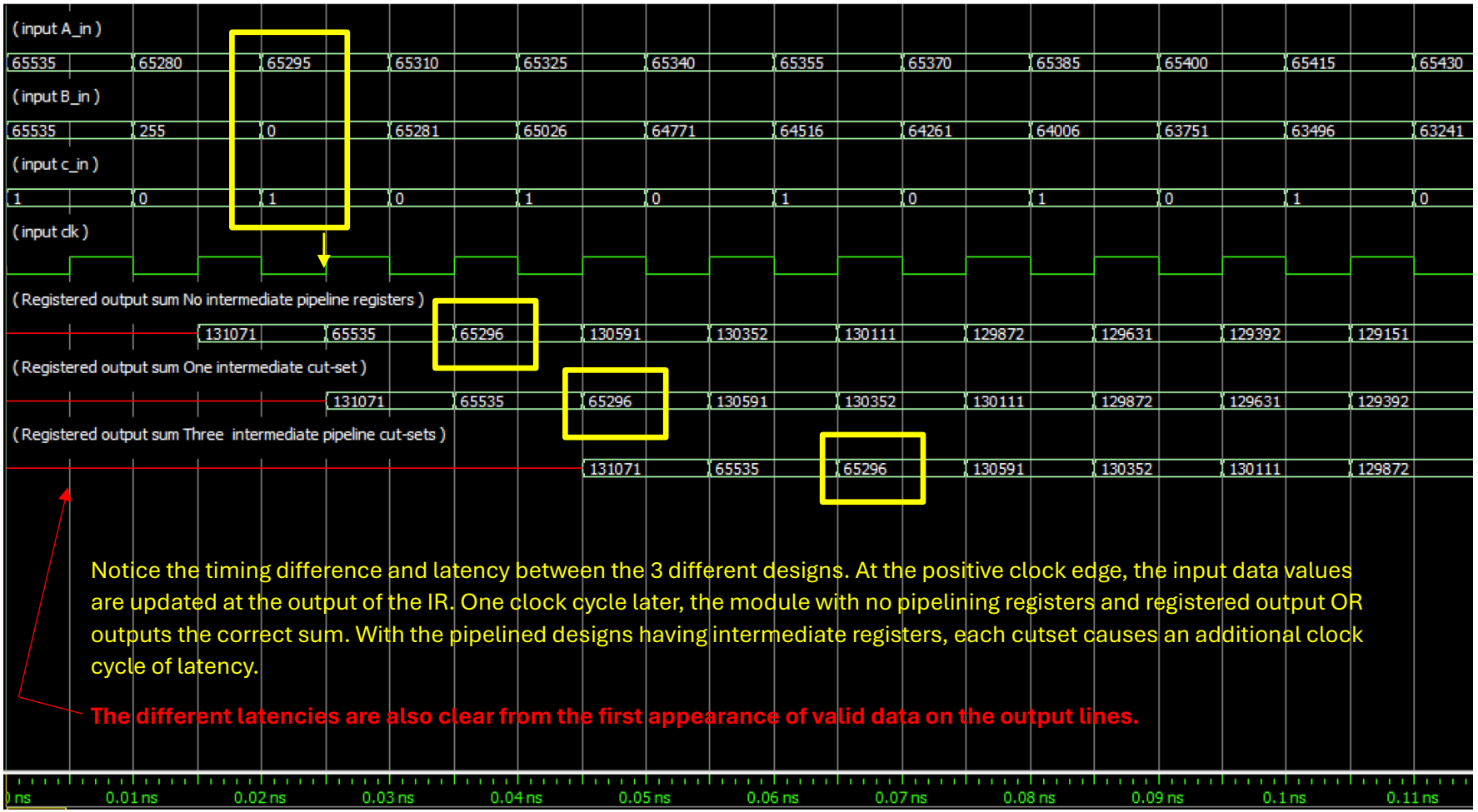
```
module Pipeline_adder_4 ( A_in, B_in, c_in, clk, sum_out);

input  [15:0]    A_in, B_in;     // 16 bit input words
input            c_in;           // carry in
input            clk;
output [16:0]    sum_out;        // including the carry_out accounting for maximum A_in and B_in.

// Internal Registers:

reg [32:0]  IR;     // Input Register     = input words (2*16 = 32) and carry_in (1)
reg [28:0]  PR1;    // Pipeline Register1 = LSBs0 & carry_in (4) + 2 MSBs (2*12) + carry_LSB (1)
reg [24:0]  PR2;    // PipeLine register2 = LSBs1 & carry_0 (4) + 2 MSB (2*8) + carry_1 (1) + Sum0  (4)
reg [20:0]  PR3;    // PIpeline register3 = LSBs2 & carry_1 (4) + 2 MSB (2*4) + carry_2 (1) + Sum01 (8)
reg [16:0]  OR;     // output register    = the total sum (16) + carry_out (1)

assign sum_out = OR;

always @ (posedge clk)
begin

// Load Input Register with new data words:
IR[0]      <=  c_in;            // carry in
IR[ 4: 1]  <=  A_in [ 3: 0];    // 4bit of A_in[ 3: 0] ;
IR[ 8: 5]  <=  B_in [ 3: 0];    // 4bit of B_in[ 3: 0];
IR[12: 9]  <=  A_in [ 7: 4];    // 4bit of A_in[ 7: 4];
IR[16:13]  <=  B_in [ 7: 4];    // 4bit of B_in[ 7: 4];
IR[20:17]  <=  A_in [11: 8];    // 4bit of A_in[11: 8];
IR[24:21]  <=  B_in [11: 8];    // 4bit of B_in[11: 8];
IR[28:25]  <=  A_in [15:12];    // 4bit of A_in[15:12];
IR[32:29]  <=  B_in [15:12];    // 4bit of B_in[15:12];

// Load Pipeline Register 1:
PR1[ 4: 0]  <= IR[8:5] + IR[4:1] + IR[0];    // Adding LSBs and the carry_in of previous entries
PR1[28: 5]  <= IR[32:9];                     // passing the MSBs for data coherency


// Load Pipeline Register 2:
PR2[ 3: 0]  <= PR1 [ 3: 0];                  // LSBs sum passes to PR2 register for data coherency
PR2[ 8: 4]  <= PR1 [12: 9] + PR1[8:5] + PR1[4];
PR2[24: 9]  <= PR1 [28:13];                  // passing the MSBs for data coherency

// Load Pipeline Register 3:
PR3[ 7: 0]  <= PR2 [ 7: 0];                  // LSBs sum passes to PR3 register for data coherency
PR3[12: 8]  <= PR2 [16:13] + PR2[12:9] + PR2[8];
PR3[20:13]  <= PR2 [24:17];                  // passing the MSBs for data coherency


// loaad the Output Register:
OR[11: 0]  <= PR3[11: 0];                    // passing the LSBs sum from PR3 as is
OR[16:12]  <= PR3[20:17] + PR3[16:13] + PR3[12];

end

endmodule
```

To compare all the designs together, a top module was created to instantiate them all, and compare their output values and latency.

```
module Pipeline_adder ( A_in, B_in, c_in, clk, Sum_none, Sum_One_cutset, Sum_three_cutsets);

input  [15:0]    A_in, B_in;      // 16 bit input words
input            c_in;            // carry in
input            clk;
output [16:0]    Sum_none;           // No intermediate registers
output [16:0]    Sum_One_cutset;     // One set of intermediate registers
output [16:0]    Sum_three_cutsets;  // Three sets of intermediate registers

NoPipeline_adder M1 ( A_in, B_in, c_in, clk, Sum_none);
Pipeline_adder_8 M2 ( A_in, B_in, c_in, clk, Sum_One_cutset);
Pipeline_adder_4 M3 ( A_in, B_in, c_in, clk, Sum_three_cutsets);

endmodule


module Pipeline_tb ();

reg  [15:0]    A_in, B_in;      // 16 bit input words
reg            c_in;            // carry in
reg            clk;
wire [16:0]    Sum_none;           // No intermediate registers
wire [16:0]    Sum_One_cutset;     // One set of intermediate registers
wire [16:0]    Sum_three_cutsets;  // Three sets of intermediate registers

Pipeline_adder UUT ( A_in, B_in, c_in, clk, Sum_none, Sum_One_cutset, Sum_three_cutsets);

initial
begin
clk = 0;
forever  begin
#5 clk = ~clk;    end
end

initial
begin
     A_in = 16'hFF_FF;    B_in = 16'hFF_FF;    c_in = 1'b1;                    // Maximum possible sum
#10  A_in = 16'hFF_00;    B_in = 16'h00_FF;    c_in = 1'b0;
forever  begin
#10  A_in = A_in + 16'h000F;  B_in = B_in - 16'h00FF;     c_in = ~ c_in;      // random input values
end
end

endmodule
```

The simulation results show how all the sums are consistent for three designs and providing the correct values. It is also clear that the latency (by clock cycles) increases with the addition of intermediate registers. While we kept the clock periods here the same, it is clear that the clock periods for the pipelined designs can be shorter allowing higher clock frequencies, and this would mean higher throughput.

Higher clock frequencies mean that different input data can be fed into the module at a higher rate. At every positive clock edge, the module outputs a calculated sum, and the module is operating at better efficiency. At any instance, any 4Bit adder block is actively operating at an addition operation. No block is idle waiting for its turn or just waiting for the other blocks to complete their calculations. This is like an automated pipeline in a manufacturing setup!

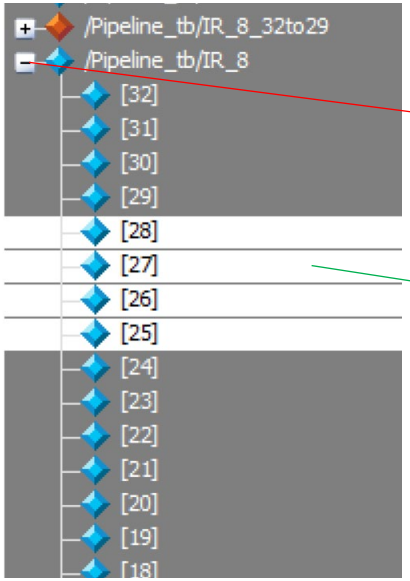


The Pipeline_adder_4 with three intermediate cut-sets has a total latency of 4 clock cycles. (one caused by OR and 3 caused by the 3 PRs). The Pipeline_adder_8 with one intermediate cut-set has a total latency of 2 clock cycles. (one caused by OR and one caused by the PR). The NoPipeline_adder with no intermediate cut-sets has a total latency of one clock cycle caused by the OR.

To verify the flow of data between the pipelined registers and how they are concatenated in the design, internal probes are added in the testbench.

```
module Pipeline_tb ();

reg  [15:0]    A_in, B_in;     // 16 bit input words
reg            c_in;           // carry in
reg            clk;
wire [16:0]    Sum_none;           // No intermediate registers
wire [16:0]    Sum_One_cutset;     // One set of intermediate registers
wire [16:0]    Sum_three_cutsets;  // Three sets of intermediate registers

Pipeline_adder UUT ( A_in, B_in, c_in, clk, Sum_none, Sum_One_cutset, Sum_three_cutsets);

wire [32:0]  IR_8;    // Input Register    = input words (2*16 = 32) and carry_in (1)
wire [24:0]  PR_8;    // Pipeline Register = the sum of LSBs & carry_in (8) + 2 MSBs (16) + carry_LSB (1)
wire [16:0]  OR_8;

assign IR_8 = UUT.M2.IR;
assign PR_8 = UUT.M2.PR;
assign OR_8 = UUT.M2.OR;

wire [32:0]  IR_4;     // Input Register     = input words (2*16 = 32) and carry_in (1)
wire [28:0]  PR1_4;    // Pipeline Register1 = LSBs0 & carry_in (4) + 2 MSBs (2*12) + carry_LSB (1)
wire [24:0]  PR2_4;    // PipeLine register2 = LSBs1 & carry_0 (4) + 2 MSB (2*8) + carry_1 (1) + Sum0  (4)
wire [20:0]  PR3_4;    // PIpeline register3 = LSBs2 & carry_1 (4) + 2 MSB (2*4) + carry_2 (1) + Sum01 (8)
wire [16:0]  OR_4;

assign IR_4  = UUT.M3.IR;
assign PR1_4 = UUT.M3.PR1;
assign PR2_4 = UUT.M3.PR2;
assign PR3_4 = UUT.M3.PR3;
assign OR_4  = UUT.M3.OR;

wire [16:0]  comb_SUM;
assign comb_SUM = UUT.M1.comb_add;

initial
begin
clk = 0;
forever  begin
#5 clk = ~clk;   end
end

initial
begin
     A_in = 16'hFF_FF;   B_in = 16'hFF_FF;   c_in = 1'b1;                  // Maximum possible sum
#10  A_in = 16'hFF_00;   B_in = 16'h00_FF;   c_in = 1'b0;
forever  begin
#10  A_in = A_in + 16'h000F;  B_in = B_in - 16'h00FF;    c_in = ~ c_in;     // random input values
end
end

endmodule
```

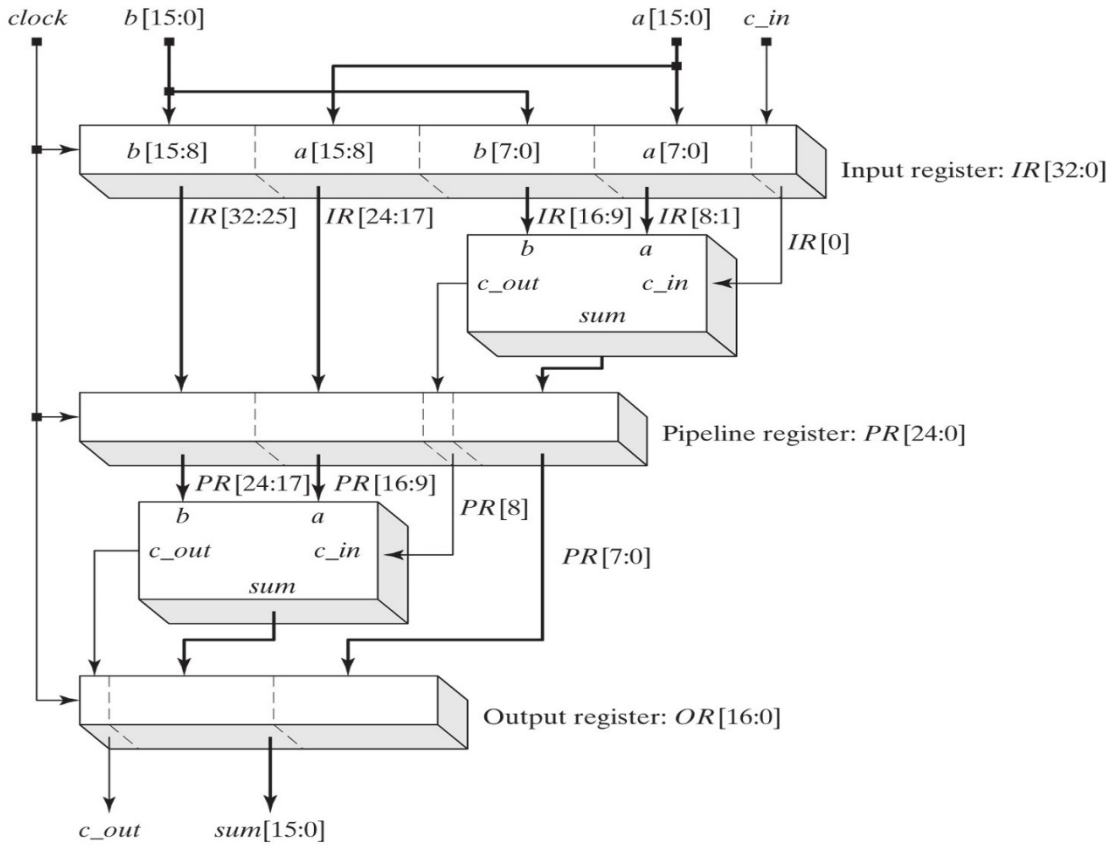
In the waveform window, we can expand and combine signals as needed to form the required groups for testing.

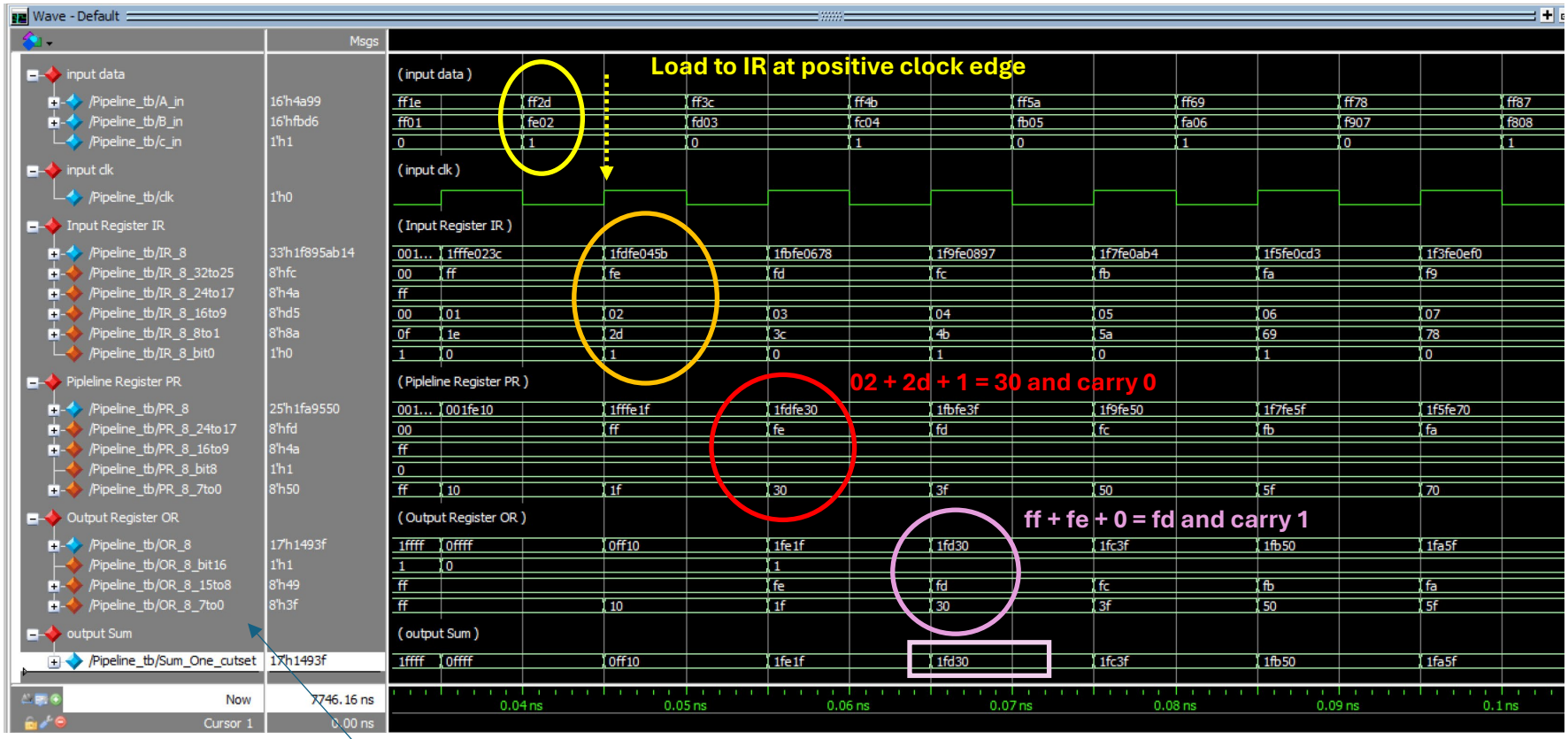


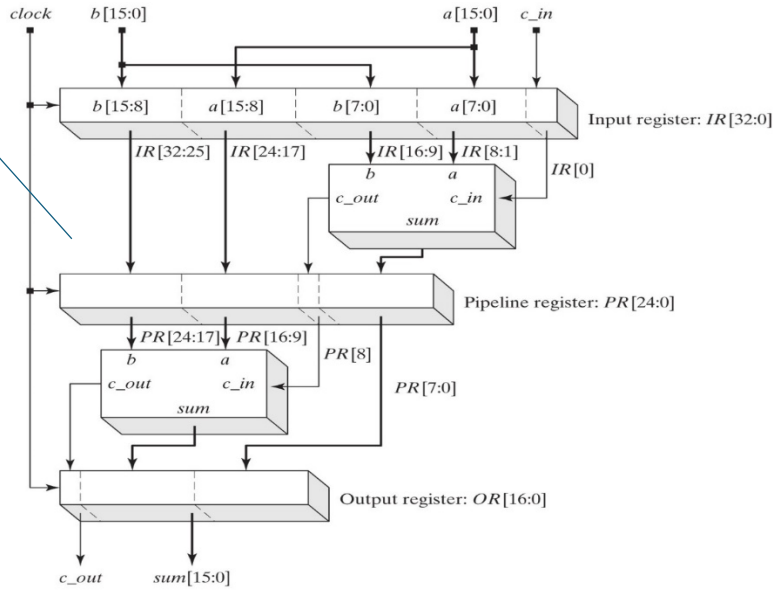
If you click on the **[+]** sign, it will change into a **[-]** and expand the bus into **single bits** numbered by their index order.

Select the bits that you would like to combine for observation and testing, and then right click, and select **combine signals**.

Hexadecimal format is optimal for this setup as it combines each 4 bits into one hexadecimal value, which makes it easier to track for each word was split into bytes to save into the registers.







Notice how over each clock cycle, the data propagates from one register to the next. When we run the timing analysis (in lesson 12), Quartus will calculate the longest propagation delays, and as designers, we will need to select a clock frequency that can accommodate the longest calculation paths for the combinational logic between the registers.