A Comprehensive Study of DSP48E1, DSP48E2, and DSP58 Primitives

Rajesh Kumar Mahato[1], Abhidan Jung Thapa[2]

1, 2 - LogicTronix Technologies

Introduction

Digital Signal Processing (DSP) is a key technology that enables modern systems to be more advanced and efficient, especially in areas like communication, multimedia, and AI [1]. In FPGA-based hardware design, numerical representation plays an important role, with fixed-point and floating-point being the two main approaches. Fixed-point computation is resource-efficient and easy to implement, making it suitable for high-speed and low-power applications. However, it offers limited precision and range. In contrast, floating-point computation provides higher accuracy and a wider range, but requires more hardware resources [2]. FPGAs include dedicated DSP blocks to accelerate these computations. For example, the Xilinx 7 Series FPGAs use the DSP48E1 primitive for efficient fixed-point operations. In more advanced platforms like the AMD Versal ACAP, the DSP58 Engine supports floating-point arithmetic based on the IEEE 754 single-precision floating-point standard [3].

DSP48E1

The DSP48E1 slice is a specific digital signal processing (DSP) primitive in Xilinx 7-series FPGA architectures, which is optimally suited to perform arithmetic operations with high speed (such as multiplication, addition, and accumulation). The DSP48E1 offers a specialized hardware block, unlike the general-purpose logic (CLB), which is much more effective in improving the performance, consuming less power and optimizing the use of resources in applications with high levels of compute-intensive workloads. It integrates a 25 X 18 two-complement multiplier, a 48-bit accumulator, a pre-adder and a reconfigurable arithmetic logic unit (ALU) into one unit. The architecture allows the designers to execute complicated DSP algorithms like FIR filters, FFTs, and MAC (multiply-accumulate) operations at low logic overhead. Pipelining and cascading is also supported by the DSP48E1 slice so that multiple slice can be linked together to perform large scale arithmetic operations [4].

Key Features of DSP48E1 Slice

- **Dedicated Multiplier (25 × 18)**
  - Supports signed (two's complement) arithmetic
  - High-speed multiplication optimized in hardware
- **48-bit Accumulator / MAC Unit**
  - Enables multiply-accumulate (MAC) operations
  - Can function as a **counter (up/down)** or accumulator
- **Fixed-Point Arithmetic Support**
  - Highly efficient for fixed-point DSP operations

- Reduces resource usage compared to floating-point implementations
- Ideal for filters, control systems, and embedded DSP

- **Pre-Adder Block**

  - Performs addition before multiplication
  - Useful in symmetric FIR filters to reduce DSP slice usage

- **Pattern Detector**

  - Detects specific bit patterns in output
  - Used for rounding, saturation, overflow detection, and thresholding

- **SIMD (Single Instruction Multiple Data) Mode**

  - Supports parallel operations:
  - Dual 24-bit or quad 12-bit operations
  - Improves throughput for vector processing

- **Arithmetic Logic Unit (ALU)**

  - Supports add, subtract, logic, and compare operations
  - Can implement multiple arithmetic functions dynamically

- **Cascading Capability**

  - Dedicated cascade paths (PCIN/PCOUT)
  - Enables building large multipliers, filters, and wide adders without routing delay

- **Pipeline Registers**

  - Multiple pipeline stages for high-frequency operation
  - Improves timing and reduces power consumption

- **Wide Function Support**

  - 96-bit accumulation using cascading
  - Supports wide bus multiplexing and barrel shifting

- **Dynamic Control (OPMODE, ALUMODE)**

  - Runtime reconfiguration of operations
  - Enables flexible DSP designs without changing hardware

- **Logic Unit Integration**

  - Can perform bitwise operations (AND, OR, XOR)
  - Extends functionality beyond arithmetic.

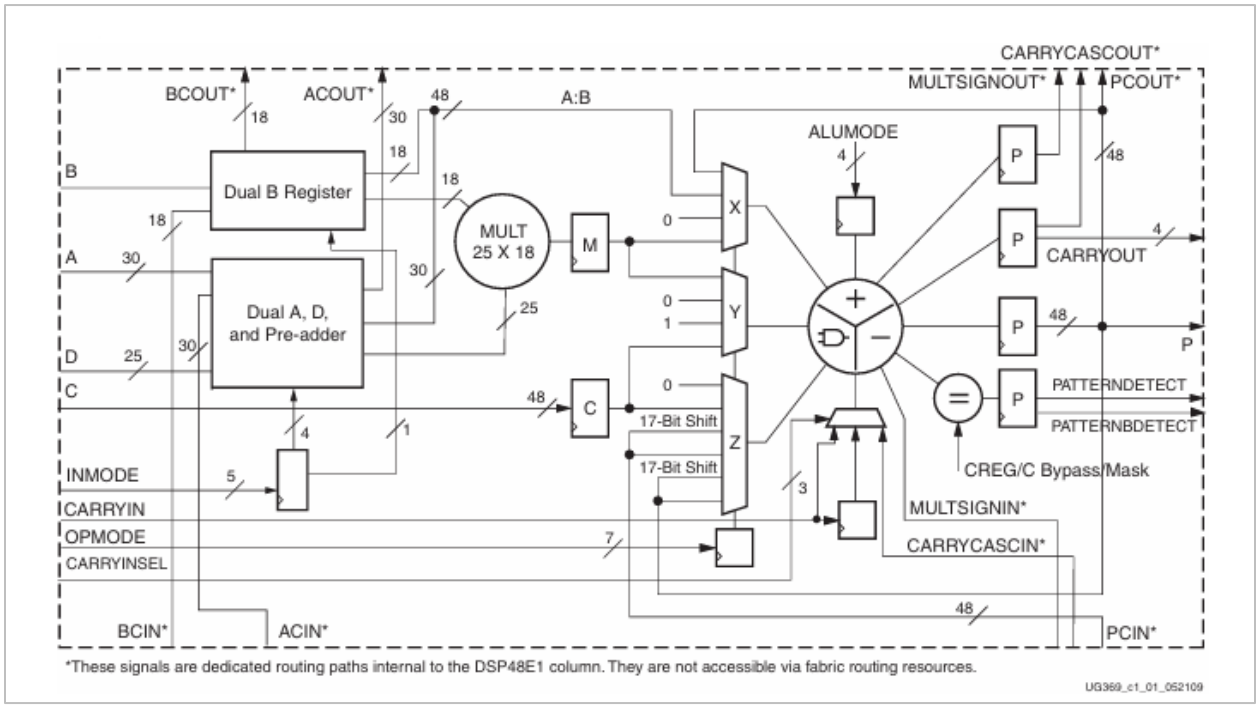

Figure 1: 7-Series FPGA DSP48E1 Slice [4]

Here are some code examples demonstrating the use of the DSP48E1 slice for multiplication, multiply-accumulate operations, and pattern detection. The following code covers all attributes with manual control, along with simulation outputs across different pipeline stages, as shown below.

Verilog Code:

```
module dsp48E1_ (
    input  wire         clk,
    input  wire         rst,     // Active high reset
    // Data inputsss
    input  wire [29:0]  A,
    input  wire [17:0]  B,
    input  wire [24:0]  D,        // Pre-adder input
    input  wire [47:0]  C_in,
    // Outputs
    output wire [47:0]  P,
    output wire         PATTERNDETECT
);

    // ============================================================
    // REGISTERED CONTROL SIGNALS (IMPORTANT FOR TIMING)
    // ============================================================
    reg [6:0] OPMODE_reg;
    reg [3:0] ALUMODE_reg;
    reg [4:0] INMODE_reg;
    reg [2:0] CARRYINSEL_reg;
    always @(posedge clk) begin
       if (rst) begin
            OPMODE_reg  <= 7'd0;
```

```
        ALUMODE_reg <= 4'd0;
        INMODE_reg  <= 5'd0;
        CARRYINSEL_reg <=3'b0;
    end else begin
      //  OPMODE_reg  <= 7'b0000101; // x->M, y->M,z->0
        OPMODE_reg <= 7'b0100101;  // x->M, y->M, z->P
        ALUMODE_reg <= 4'b0000; // x+y+z+cin
        INMODE_reg  <= 5'b00000; // it select the A2 and B2 as a input ports
        CARRYINSEL_reg<= 3'b000;
    end
end

// ============================================================
//  DSP48E1 INSTANCE
// ============================================================
DSP48E1 #(

    // ================= PIPELINE =================
    .AREG(1),
    .ACASCREG(1),
    .BREG(1),
    .BCASCREG(1),
    .MREG(1),
    .PREG(1),
    .CREG(1),

    // Control pipeline
    .ALUMODEREG(1),
    .OPMODEREG(1),
    .INMODEREG(1),
    .CARRYINREG(1),
    .CARRYINSELREG(1),

    // Pre-adder
    .DREG(1),
    .ADREG(1),
    .USE_DPORT("FALSE"),    // TRUE-> ENABLE PRE-ADDER
                           // FALSE -> DISABLE PRE-ADDER

    // Multiplier
    .USE_MULT("MULTIPLY"), // MULTIPLY,NONE,DYNAMIC
    .USE_SIMD("ONE48"),    //ONE48,TWO24,FOUR12

    // ================= PATTERN DETECTOR =================
    .USE_PATTERN_DETECT("PATDET"),   //  NO_PATDET,PATDET
    .PATTERN(48'd50),      // Detect 50
    .MASK(48'd0),         //When a MASK bit is set to 1, the
                //corresponding pattern bit is ignored. When a MASK bit is set to
                                    //0, the pattern bit is compared.
    .SEL_MASK("MASK"),               // MASK, C, ROUNDING_MODE1,ROUNDING_MODE2
    .SEL_PATTERN("PATTERN"),         // PATTERN,C
    .AUTORESET_PATDET("NO_RESET")   // NO_RESET, RESET_MATCH,RESET_NOT_MATCH

) dsp_inst (

    // ====================================================
    //  DATA PATH
    // ====================================================
    .A(A),
    .B(B),
    .C(C_in),
    .D(D),
```

```
        // ================================================
        //  CONTROL PATH (REGISTERED)
        // ================================================
        .OPMODE(OPMODE_reg),
        .ALUMODE(ALUMODE_reg),
        .INMODE(INMODE_reg),
        .CARRYINSEL(CARRYINSEL_reg),


        // ================================================
        //  CLOCK / ENABLE
        // ================================================
        .CLK(clk),
        .CEA1(1'b1),
        .CEA2(1'b1),
        .CEB1(1'b1),
        .CEB2(1'b1),
        .CEC(1'b1),
        .CED(1'b1),
        .CEAD(1'b1),
        .CEALUMODE(1'b1),
        .CECTRL(1'b1),
        .CECARRYIN(1'b1),
        .CEINMODE(1'b1),
        .CEM(1'b1),
        .CEP(1'b1),

        // ================================================
        //  RESET
        // ================================================
        .RSTA(rst),
        .RSTB(rst),
        .RSTC(rst),
        .RSTD(rst),
        .RSTP(rst),
        .RSTCTRL(rst),
        .RSTALUMODE(rst),
        .RSTINMODE(rst),
        .RSTM(rst),
        .RSTALLCARRYIN(rst),

        // ================================================
        // OTHER SIGNALS
        // ================================================
        .CARRYIN(1'b0),
        .PCIN(48'd0),
        .ACIN(30'd0),
        .BCIN(18'd0),
        .CARRYCASCIN(1'b0),
        .MULTSIGNIN(1'b0),

        // ================================================
        //  OUTPUT
        // ================================================
        .P(P),
        .PATTERNDETECT(PATTERNDETECT)
    );
endmodule
```

Test Bench Code:

```
module dsp48E1_TB();
  reg         clk;
  reg         rst;
  reg [29:0]  A;
  reg [17:0]  B;
  reg [24:0]  D;        // Pre-adder input
  reg [47:0]  C_in;
   wire [47:0]  P;
   wire         PATTERNDETECT;

   dsp48E1_ uut(
   .clk(clk),
   .rst(rst),
   .A(A),
   .B(B),
   .D(D),
   .C_in(C_in),
   .P(P),
   .PATTERNDETECT(PATTERNDETECT)
   );

   initial begin
   clk<=0;
   rst<=1'b0;
   end
   initial forever begin #50 clk <= ~clk; end
   initial begin

   @(posedge clk)
   A<=30'b111111111111111111111111111011; //-5
   B<=18'd10; //10
   C_in<=48'd0;
   D<= 25'd0;
   end
endmodule
```

Simulation Result:

Case 1:

```
OPMODE_reg  <= 7'b0000101; // x->M, y->M,z->0
Areg <=1;
A<=30'b111111111111111111111111111011; //-5
B<=18'd10; //10
PATTERN(48'd50),      // Detect 50
```

Latency is 3 clock cycle

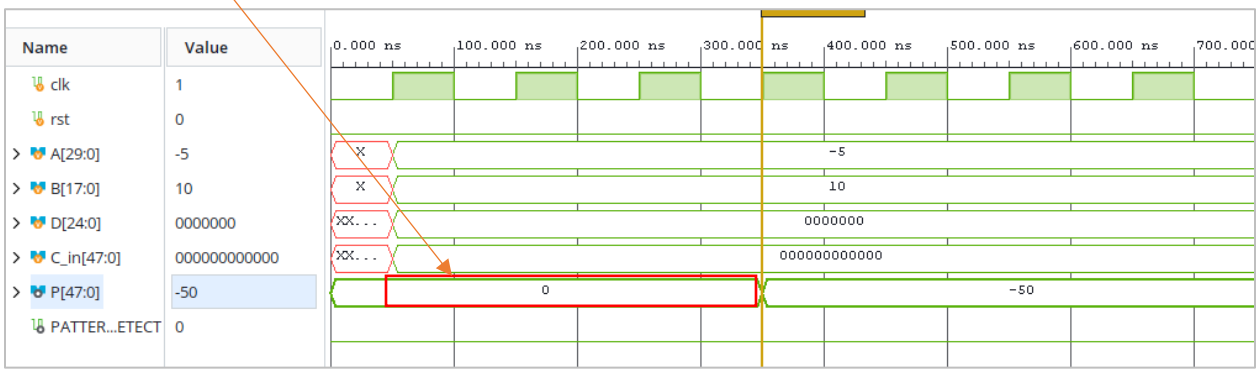

Figure 2: Case 1, DSP48E1 simulation result

Case 2:

```
Areg <=2;
OPMODE_reg  <= 7'b0000101; // x->M, y->M,z->0
A<=30'd5; //5
B<=18'd10; //10
PATTERN(48'd50),      // Detect 50
```

Latency is 4 clock cycle

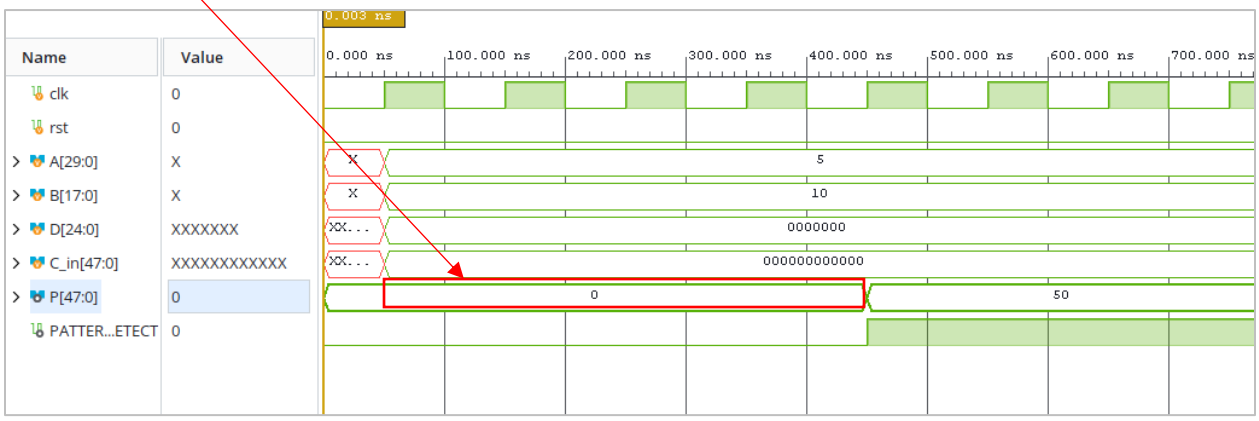

Figure 3: Case 2, DSP48E1 simulation result

Case 3:

```
Areg <=2;
OPMODE_reg <= 7'b0100101;  // x->M, y->M, z->P
A<=30'd5; //5
B<=18'd10; //10
PATTERN(48'd50),       // Detect 50
```

Latency is 4 clock cycle

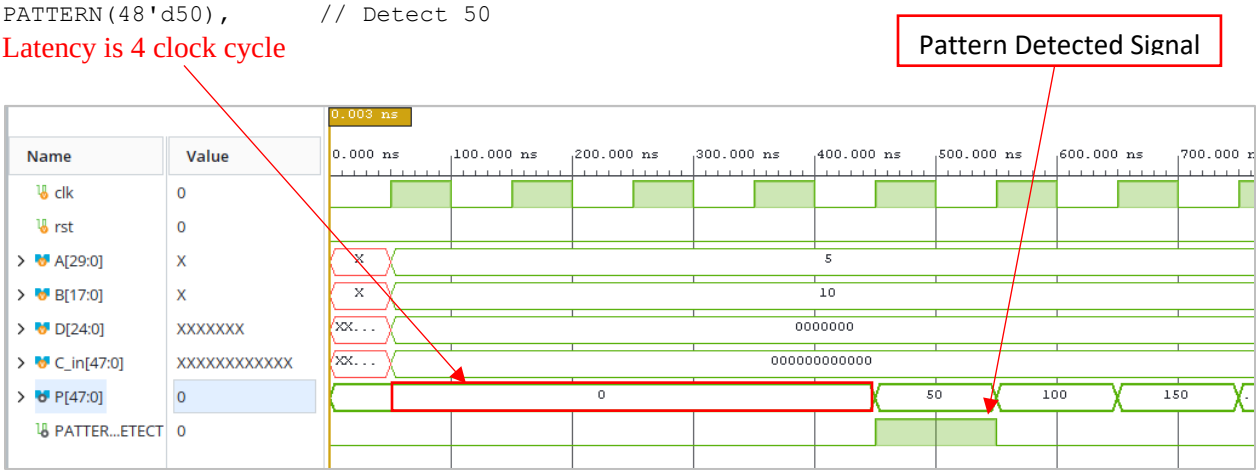

Figure 4: Case 3, DSP48E1 simulation result

These three simulation cases explain how pipeline registers affect the latency of a system. For higher clock frequencies, pipeline registers help meet timing requirements. On the other hand, for lower clock frequencies, using fewer pipeline stages can reduce system latency. For example, when AREG is set to 1, the latency is 3 clock cycles. When AREG is set to 2, the latency increases by one clock cycle, making it 4 clock cycles. Additionally, by changing control signals such as OPMODE, we can modify the operating mode of the primitive. This allows the system to handle different mathematical operations dynamically.

Resource Utilization Report:

```
DSP Final Report (the ' indicates corresponding REG is set)
+------------+-------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|Module Name | DSP Mapping | A Size | B Size | C Size | D Size | P Size | AREG | BREG | CREG | DREG | ADREG | MREG | PREG |
+------------+-------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
|dsp48E1_    | Dynamic     | -      | -      | -      | -      | 48     | -    | -    | -    | -    | -     | 1    | 1    |
+------------+-------------+--------+--------+--------+--------+--------+------+------+------+------+-------+------+------+
```

```
3. DSP
------

+-----------------+------+-------+------------+-----------+-------+
|    Site Type    | Used | Fixed | Prohibited | Available | Util% |
+-----------------+------+-------+------------+-----------+-------+
| DSPs            |    1 |     0 |          0 |       120 |  0.83 |
|   DSP48E1 only |    1 |       |            |           |       |
+-----------------+------+-------+------------+-----------+-------+
```

Figure 5: Resource Utilization Report of DSP48E1

DSP48E2

The DSP48E2 is the special digital signal processing unit used in the Xilinx Ultra scale architecture, and that is optimally used to perform arithmetic operations with fully parallel hardware. It is an extended and compatible version of the DSP48E1 slice used in 7-series FPGA. The slice contains a pre-adder, multiplier of 27*18, and a 48-bit arithmetic and logic unit (ALU) that allows high-speed multiplication, accumulation and logic operations to be done in a smaller and less-power consuming hardware block [5].

Features and Supported Operations

- **27-bit pre-adder with optional D register to enhance A or B input processing**
- **Selectable A or B input to the pre-adder for flexible coefficient handling**
- **Pre-adder output can be routed to both multiplier inputs to support squaring operations**
- **27 X 18 two's complement signed multiplier with dynamic bypass capability**
- **48-bit ALU supporting:**
  - Addition, Subtraction, and Accumulation
  - Four-Input arithmetic operations
  - Bitwise logic functions: AND, OR, NOT, NAND, NOR, XOR and XNOR
- **Single instruction Multiple Data (SIMD) modes:**
  - Dual 24-bit add/subtraction/accumulate
  - Quad 12-bit add/subtraction/accumulate
- **Pattern detector supporting:**
  - Overflow and Underflow detection
  - Convergent and symmetric rounding
  - Terminal count detection and auto-reset
- **Wide XOR functionality configurable from multiple 12-bit segments up to 96-bit XOR**
- **Cascading support through dedicated internal paths for:**
  - Wide accumulators
  - Large adders and counters
  - FIR Filters and complex arithmetic
- **Optional input, pipeline, and output registers to maximize performance and efficiency.**

- **Internal multiplier and XOR logic can be gated off when unused to reduce power consumption**

Control Registers and Operating Modes

- **OPMODE (9-bit control word):**
  - Selects inputs for W, X, Y, and Z multiplexers
  - Determine how multiplier output, accumulator feedback, and external inputs are combined
  - Enable dynamics switching between arithmetic functions.
- **ALUMODE (4-bit control word):**
  - Control ALU operation type
  - Selects between add, subtract, accumulate, and logic operations
  - Enable dynamic selection of bitwise logic functions
- **INMODE (5-bit control word):**
  - Controls A, B and D input register selection
  - Configure pre-adder behavior, including add/ subtraction and masking
  - Support balanced pipelining when switching between multiply and add operations
- **CARRYINSEL (3-bit control word):**
  - Selects carry input source for the second-stage adder
  - Supports rounding and wide add/subtract operation
- **Optional control signal registers**
  - OPMODE, ALUMODE, and CARRYINSEL can be registered
  - Enable higher clock frequency and stable dynamic operation
- **Dynamic control capability:**
  - Allows a single DSP48E2 slice to perform multiple functions across clock cycles
  - Supports both sequential and cascaded operations without reconfiguration

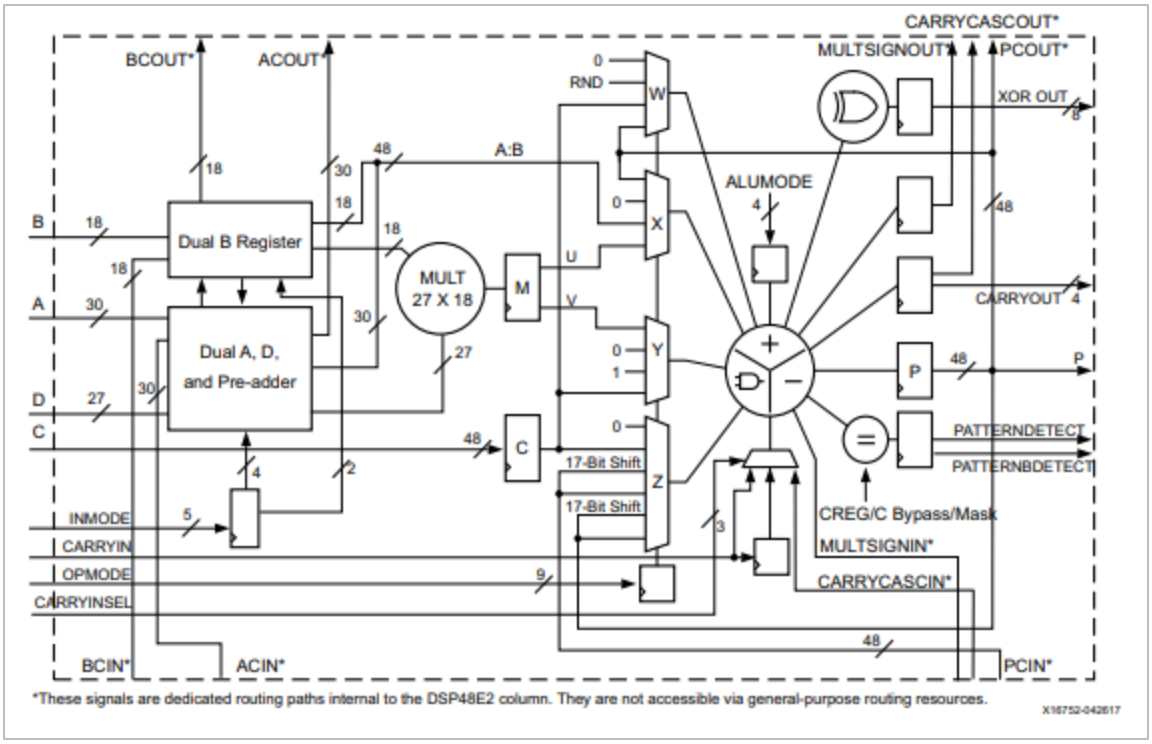

Figure 6: DSP48E2 Architecture [5]

Verilog Code (Source file):

```
module dsp48E2_ (
    input  wire         clk,
    input  wire         rst,     // Active high reset
    // Data inputs
    input  wire [29:0]  A,
    input  wire [17:0]  B,
    input  wire [26:0]  D,        // UPDATED: Pre-adder input (25 -> 27 bits)
    input  wire [47:0]  C_in,
    // Outputs
    output wire [47:0]  P,
    output wire         PATTERNDETECT
);
    // ============================================================
    // REGISTERED CONTROL SIGNALS (9-bit OPMODE for E2)
    // ============================================================
    reg [8:0] OPMODE_reg;      // UPDATED: 7 -> 9 bits
    reg [3:0] ALUMODE_reg;
    reg [4:0] INMODE_reg;
    reg [2:0] CARRYINSEL_reg;

    always @(posedge clk) begin
        if (rst) begin
            OPMODE_reg      <= 9'd0;
            ALUMODE_reg     <= 4'd0;
            INMODE_reg      <= 5'd0;
            CARRYINSEL_reg  <= 3'b0;
        end else begin
            // UPDATED OPMODE: 9-bit format [8:7]=W, [6:4]=Z, [3:2]=Y, [1:0]=X
            // Legacy 0100101 (X=01, Y=01, Z=010) becomes 9'b0000100101
```

```
        //OPMODE_reg      <= 9'b000000101; // only multiplication
        OPMODE_reg      <=9'b000100101; // multiplication with accumulation
        ALUMODE_reg     <= 4'b0000;      // x+y+z+cin
        INMODE_reg      <= 5'b00000;     // Select A2 and B2
        CARRYINSEL_reg  <= 3'b000;  // select carryin
    end
end

// ============================================================
//  DSP48E2 INSTANCE (Updated for UltraScale)
// ============================================================
DSP48E2 #(
    // ================= PIPELINE =================
    .AREG(1),  // 0,1,2
    .ACASCREG(1), //0,1,2
    .BREG(1),     //0,1,2
    .BCASCREG(1), //0,1,2
    .MREG(1),
    .PREG(1),
    .CREG(1),
    // Control pipeline
    .ALUMODEREG(1),
    .OPMODEREG(1),
    .INMODEREG(1),
    .CARRYINREG(1),
    .CARRYINSELREG(1),
    // Pre-adder
    .DREG(1),
    .ADREG(1),
    // UPDATED: USE_DPORT (E1) -> AMULTSEL (E2)
    .AMULTSEL("A"),       // "AD" enables pre-adder, "A" bypasses it
    .BMULTSEL("B"),        // Selects B register path
    .PREADDINSEL("A"),     // Selects A for pre-adder
    // Multiplier
    .USE_MULT("MULTIPLY"), // NONE, MULTIPLY, DYNAMIC
    .USE_SIMD("ONE48"),
    .RND(48'd0),
    .USE_WIDEXOR("FALSE"),
    .XORSIMD("XOR24_48_96"),

    // ================= PATTERN DETECTOR =================
    .USE_PATTERN_DETECT("PATDET"),
    .PATTERN(48'd50),
    .MASK(48'd0),
    .SEL_MASK("MASK"),
    .SEL_PATTERN("PATTERN"),
    .AUTORESET_PATDET("NO_RESET")

) dsp_inst (
    // ====================================================
    //  DATA PATH
    // ====================================================
    .A(A),
    .B(B),
    .C(C_in),
    .D(D),

    // ====================================================
```

```
        //  CONTROL PATH (REGISTERED)
        // ====================================================
        .OPMODE(OPMODE_reg),
        .ALUMODE(ALUMODE_reg),
        .INMODE(INMODE_reg),
        .CARRYINSEL(CARRYINSEL_reg),

        // ====================================================
        //  CLOCK / ENABLE
        // ====================================================
        .CLK(clk),
        .CEA1(1'b1),
        .CEA2(1'b1),
        .CEB1(1'b1),
        .CEB2(1'b1),
        .CEC(1'b1),
        .CED(1'b1),
        .CEAD(1'b1),
        .CEALUMODE(1'b1),
        .CECTRL(1'b1),
        .CECARRYIN(1'b1),
        .CEINMODE(1'b1),
        .CEM(1'b1),
        .CEP(1'b1),

        // ====================================================
        //  RESET
        // ====================================================
        .RSTA(rst),
        .RSTB(rst),
        .RSTC(rst),
        .RSTD(rst),
        .RSTP(rst),
        .RSTCTRL(rst),
        .RSTALUMODE(rst),
        .RSTINMODE(rst),
        .RSTM(rst),
        .RSTALLCARRYIN(rst),

        // ====================================================
        // OTHER SIGNALS
        // ====================================================
        .CARRYIN(1'b0),
        .PCIN(48'd0),
        .ACIN(30'd0),
        .BCIN(18'd0),
        .CARRYCASCIN(1'b0),
        .MULTSIGNIN(1'b0),

        // ====================================================
        //  OUTPUT
        // ====================================================
        .P(P),
        .PATTERNDETECT(PATTERNDETECT)
    );
endmodule
```

Test Bench file:

```
module DSP48E2_tb();
 reg         clk;
 reg         rst;     // Active high reset
    // Data inputs
 reg [29:0]  A;
 reg [17:0]  B;
 reg [26:0]  D;        // UPDATED: Pre-adder input (25 -> 27 bits)
 reg [47:0]  C_in;
    // Outputs
 wire [47:0]  P;
 wire         PATTERNDETECT;


 dsp48E2_ uut(
 .clk(clk),
 .rst(rst),
 .A(A),
 .B(B),
 .D(D),
 .C_in(C_in),
 .P(P),
 .PATTERNDETECT(PATTERNDETECT)
 );
initial begin
clk<=0;
rst<=0;
end
initial forever begin #50; clk=~clk; end
initial begin
@(posedge clk)
//A=30'd5;
A=30'b111111111111111111111111111011;
B=18'd10;
D=27'd0;
C_in=48'd0;
end
endmodule
```

Simulation Result:

Case 1

```
Areg=1;
Breg=1;
A=30'd5;
B=18'd10;
OPMODE_reg      <= 9'b000000101; // only multiplication
ALUMODE_reg     <= 4'b0000;      // x+y+z+cin
INMODE_reg      <= 5'b00000;     // Select A2 and B2
CARRYINSEL_reg  <= 3'b000;  // select carryin
.PATTERN(48'd50),  // detect 50 in output stream
```

Latency is 3 clock cycle

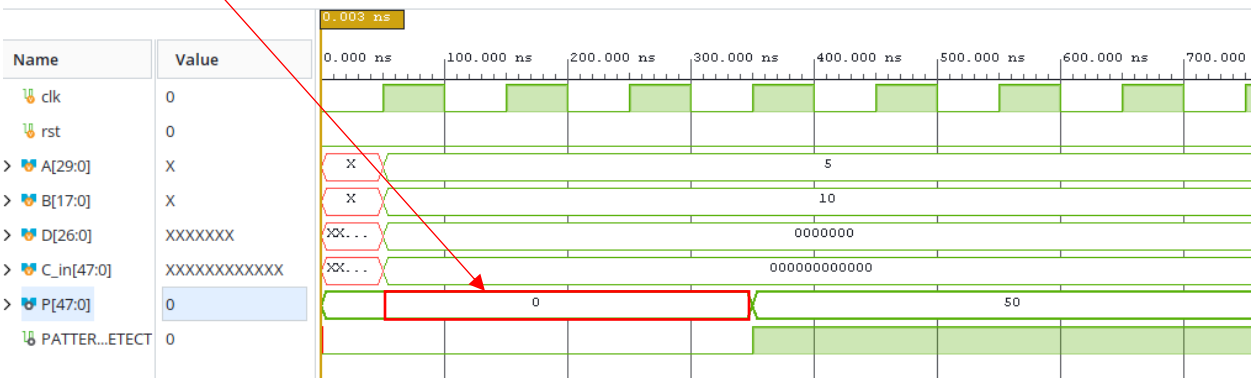

Figure 7: Case 1, DSP48E2 Simulation Result

Case 2

```
Areg=2;
Breg=2;
A=30'd5;
B=18'd10;
OPMODE_reg      <= 9'b000000101; // only multiplication
ALUMODE_reg     <= 4'b0000;      // x+y+z+cin
INMODE_reg      <= 5'b00000;     // Select A2 and B2
CARRYINSEL_reg  <= 3'b000;  // select carryin
.PATTERN(48'd50),  // detect 50 in output stream
```

Latency is 4 clock cycle

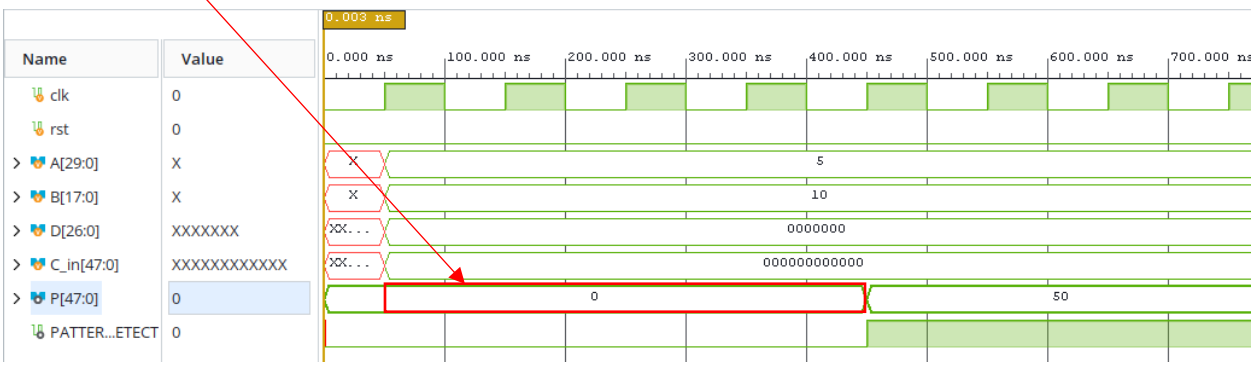

Figure 8: Case 2, DSP48E2 Simulation Result

Case 3

```
Areg=2;
Breg=2;
A=30'd5;
B=18'd10;
OPMODE_reg      <=9'b000100101; // multiplication with accumulation
ALUMODE_reg     <= 4'b0000;      // x+y+z+cin
INMODE_reg      <= 5'b00000;     // Select A2 and B2
CARRYINSEL_reg  <= 3'b000;  // select carryin
.PATTERN(48'd50),  // detect 50 in output stream
```

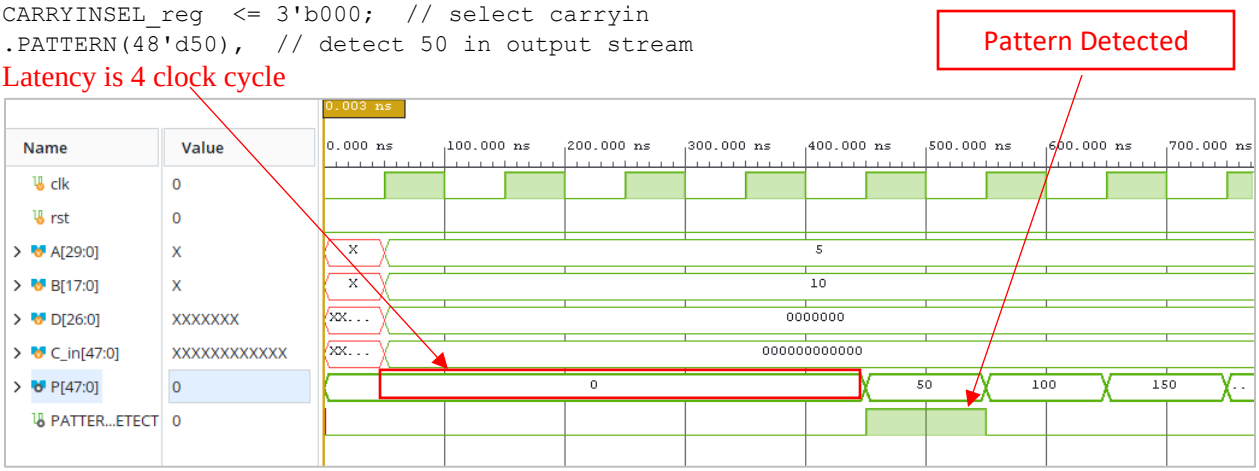

Figure 9: Case 3, DSP48E2 Simulation Result

The role of pipeline registers and latency in this primitive is similar to that of the DSP48E1. However, this primitive includes additional features that make it more advanced and better suited for DSP applications and dynamic usage.

Resource Utilization Report:

3. ARITHMETIC

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	1	0	0	1824	0.05
DSP48E2 only	1				

DSP Final Report (the ' indicates corresponding REG is set)

Module Name	DSP Mapping	A Size	B Size	C Size	D Size	P Size	AREG	BREG	CREG	DREG	ADREG	MREG	PREG
dsp48E2_	Dynamic	-	-	-	-	48	-	-	-	-	-	1	1

Figure 10: Resource Utilization Report of DSP48E2

DSP58

The Versal DSP Engine primitives are the fundamental hardware building blocks embedded in the Versal Adaptive Compute Acceleration Platforms (ACAPs) that used to implement high-performance arithmetic operations in AMD Versal ACAP. These primitives represent special DSP hardware, that optimized for scalable signal processing workloads and are intended for use in Versal devices where it compute numerical operations like: multiply-accumulate, complex number, vector processing and floating-point computations, and these are efficiently implemented in hardware. Primitives in the Versal DSP Engine are DSP58, DSPCPLX and DSPFP32, one representing each functional mode of the Versal DSP Engine, and it intended to be instantiated directly in hardware design to provide advanced DSP functionality [6].

Key Features of Versal DSP primitives

The Versal DSP Engine primitives combine multiple arithmetic capabilities and extensible processing modes into a hardware block:

- **DSP58 Primitive**
  - Implements a 27*24 fixed-point multiply with a 58-bit accumulator/logic unit.
  - Includes a pre-adder for flexible input combinations.
  - Supports cascaded DSP chaining for large accumulators and hierarchical functions.

- **DSPCPLX Primitive**

  - Combine two DSP58 units to form complex 18*18 multiply unit.
  - Supports real and imaginary components simultaneously.

- **DSPFP32 Primitive**

  - Specialized block for single-precision floating-point arithmetic.
  - Provide both floating multiply and floating-point add outputs and status flags (overflow, underflow and invalid)

- **Mixed Modes and Functional Flexibility**

  - DSP58 supports SIMD arithmetic, vector dot products, and optional wide logic units.
  - DSPCPLX supports paired DSP execution for complex signal processing flows.
  - DSPFP32 handles floating-point arithmetic directly within the DSP primitive.

Control Word and Configuration Ports

Each primitive provides a set of control and configuration inputs that dictate how the hardware operates:

- OPMODE (9-bit control word): Defines the arithmetic or logic function applied in the DSP58 and DSPCPLX primitives by selecting input paths and operation mode.
- ALUMODE (4-bit control word): Determines the ALU function such as add, subtract, or logic operation.
- CARRYINSEL (3-bit control word): Selects the carry input source for the second-stage adder.
- INMODE (5-bit control word): Controls how inputs are registered and pre-adder behavior.
- FPOPMODE (6-bit control word): For DSPFP32, controls floating-point specific operation modes of FP arithmetic.

- Clock enables and reset signals: Provide fine-grain control over internal register enables and resets to manage power and pipelining.

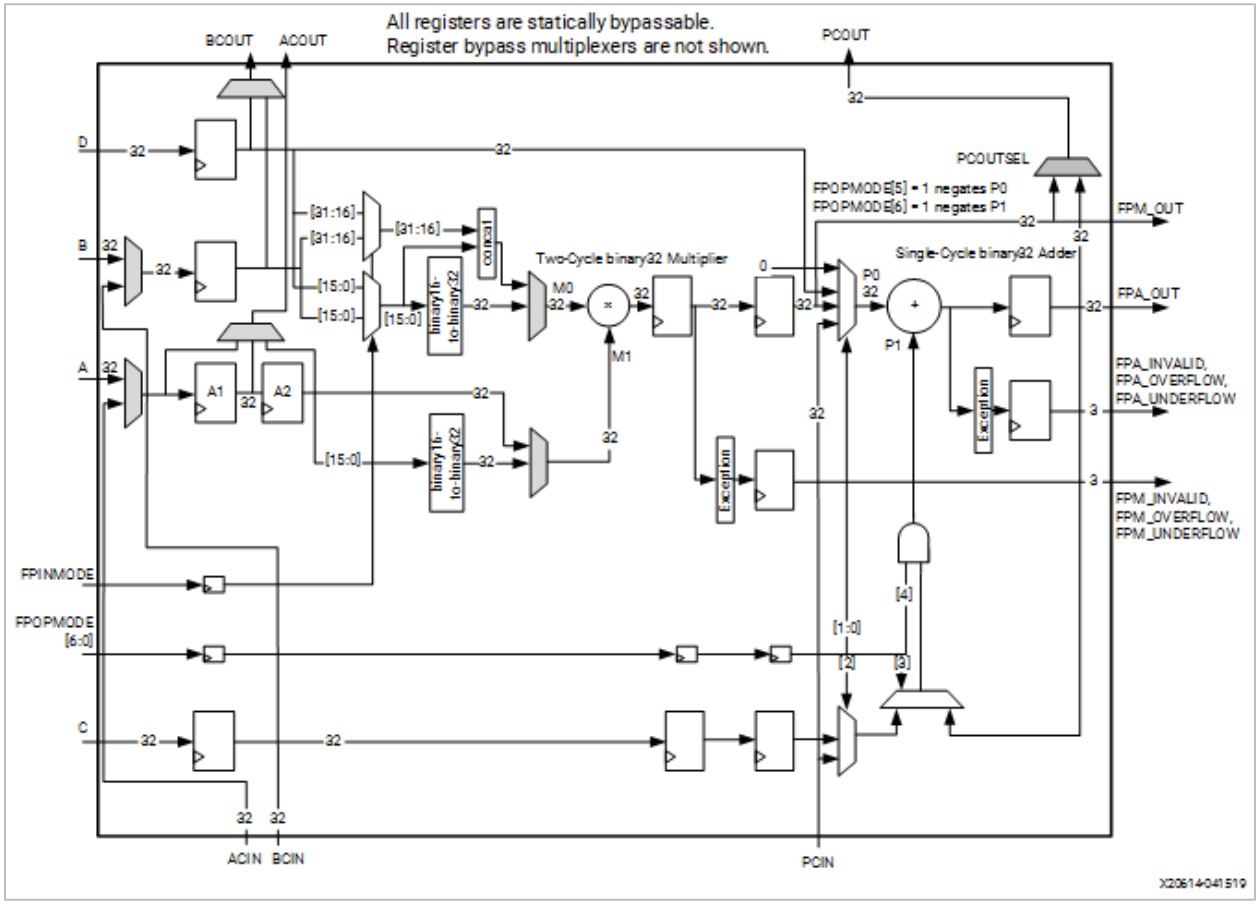

Figure 11: DSP58 Engine Architecture [6]

The following code example explains each attribute and port, and how they are used in the design to properly control every register. This example explores the use of the DSP58 engine in floating-point mode for multiplication and multiply-accumulate operations. The input value should be in the IEEE754 single precision standard format.

Verilog Code (Source File):

```
module DSP58_ (
    input  wire        clk,       // Clock input
    input  wire        rst,         // Synchronous reset
    input  wire [31:0] A,           // Floating-point input A (FP32)
    input  wire [31:0] B,           // Floating-point input B (FP32)
    output wire [31:0] P,           // Floating-point result (FP32)
    output wire        invalid_o,   // Invalid operation flag (NaN, 0*Inf etc.)
    output wire        overflow_o,  // Overflow flag (result too large)
    output wire        underflow_o // Underflow flag (result too small)

);
```

```
/*****************************************************************************/
    // Local Parameters: Pipeline & Register Configuration

/*****************************************************************************/
    localparam AREG        = 1;   // Pipeline stages for A input
    localparam FPBREG      = 1;   // Pipeline stages for B input
    localparam FPMPIPEREG  = 1;   // Pipeline stages inside multiplier
    localparam FPM_PREG    = 1;   // Pipeline stages for multiplier output
    localparam FPA_PREG    = 1;   // Pipeline stages for A internal
    localparam FPOPMREG    = 1;   // Pipeline stages for multiplier post-processing


/*****************************************************************************/
    // Internal Signals

/*****************************************************************************/
    reg        fpinmode;         // Floating-point input mode signal
    reg  [6:0]  fpopmode;         // Floating-point output mode register

    // Decompose input A into FP fields
    wire [7:0]  a_exp_i  = A[30:23];  // Exponent field
    wire [22:0] a_man_i  = A[22:0];   // Mantissa field
    wire        a_sign_i = A[31];     // Sign bit

    // Decompose input B into FP fields
    wire [7:0]  b_exp_i  = B[30:23];  // Exponent field
    wire [22:0] b_man_i  = B[22:0];   // Mantissa field
    wire        b_sign_i = B[31];     // Sign bit

    // Intermediate FPA output (internal to DSP block)
    wire [31:0] y_o;

  always @(posedge clk) begin
  if(rst)begin
  fpinmode <= 1'b1;  // 1 for B as input and 0 for D as a input port
  fpopmode <= 7'd0;
  end
  else begin
  fpinmode <=1'b1;
 // fpopmode <= 7'b0000001;  // only multiplication between two operand
  fpopmode <= 7'b0010001;  // multiplication and accumulation
  end
  end


/*****************************************************************************/
    // DSPFP32 Instantiation: Floating-Point Multiplier

/*****************************************************************************/
    DSPFP32 #(
        // --- Input A ---
        .ACASCREG (1),         // Cascade register stages for A
        .AREG     (AREG),      // Pipeline stages for A
        .A_INPUT  ("DIRECT"),  // Direct input (not cascaded)

        // --- Input B ---
        .B_INPUT    ("DIRECT"),// Direct input for B
```

```
    .FPBREG     (FPBREG),  // Pipeline stages for B

    // --- Internal pipeline registers ---
    .FPA_PREG     (FPA_PREG),     // Pipeline stages for A processing
    .FPCREG       (0),             // C input register stages
    .FPDREG       (1),             // D input register stages
    .FPMPIPEREG   (FPMPIPEREG),   // Multiplier pipeline stages
    .FPM_PREG     (FPM_PREG),     // Post-multiplier pipeline stages
    .FPOPMREG     (FPOPMREG),     // Multiplier output pipeline
    .INMODEREG    (0),             // Input mode register stages

    // --- Output selection ---
    .PCOUTSEL     ("FPM"),        // Select output from FPM stage

    // --- Reset mode ---
    .RESET_MODE   ("SYNC"),       // Synchronous reset

    // --- Operation ---
    .USE_MULT     ("MULTIPLY")    // Floating-point multiply
)
DSPFP32_inst (
    // --- FPA Outputs ---
    .FPA_OUT(P),               // Internal FPA output
    .FPA_INVALID(invalid_o),     // Invalid operation
    .FPA_OVERFLOW(overflow_o),   // Overflow
    .FPA_UNDERFLOW(underflow_o), // Underflow
    // --- Input operands ---
    .A_EXP(a_exp_i),
    .A_MAN(a_man_i),
    .A_SIGN(a_sign_i),
    .B_EXP(b_exp_i),
    .B_MAN(b_man_i),
    .B_SIGN(b_sign_i),
    // --- Clock enables ---
    .CEA1(1'b1),                 // Enable A register
    .CEA2(1'b1),
    .CEB(1'b1),                  // Enable B register
    .CEC(1'b0),                  // C input not used
    .CED(1'b0),                  // D input not used
    .CEFPA(1'b1),                // Enable FPA stage
    .CEFPM(1'b1),                // Enable FPM stage
    .CEFPMPIPE(1'b1),            // Enable FPM pipeline
    .CEFPINMODE(1'b1),               // FP input mode enable
    .CEFPOPMODE(1'b1),           // FP output mode enable
    // --- Clock & mode signals ---
    .CLK(clk),
    .FPINMODE(fpinmode),
    .FPOPMODE(fpopmode),
    .PCIN(),
    // --- Reset signals ---
    .RSTA(rst),
    .RSTB(rst),
    .RSTC(rst),
    .RSTD(rst),
    .RSTFPA(rst),
    .RSTFPINMODE(rst),
    .RSTFPM(rst),
    .RSTFPMPIPE(rst),
```

```
        .RSTFPOPMODE(rst)
    );
endmodule
```

TestBench :

```
module dsp58_tb();
   reg        clk;       // Clock input
   reg       rst;         // Synchronous reset
   reg [31:0] A;           // Floating-point input A (FP32)
   reg [31:0] B;           // Floating-point input B (FP32)
    wire [31:0] P;           // Floating-point result (FP32)
    wire        invalid_o;   // Invalid operation flag (NaN, 0*Inf etc.)
    wire        overflow_o;  // Overflow flag (result too large)
    wire        underflow_o;

DSP58_ uut(
.clk(clk),
.rst(rst),
.A(A),
.B(B),
.P(P),
.invalid_o(invalid_o),
.overflow_o(overflow_o),
.underflow_o(underflow_o)
);
initial begin
clk=0;
rst=0;
forever begin
#50;
clk=~clk;
end
end
initial begin
@(posedge clk)
A=32'b01000000101000000000000000000000; //5
B=32'b01000001001000000000000000000000; //10
end
endmodule
```

Simulation Result:

Case 1:

```
Localparam AREG         = 1;
Fpopmode <= 7'b0000001; // only multiplication between two operand
```

Latency is 4 clock cycle

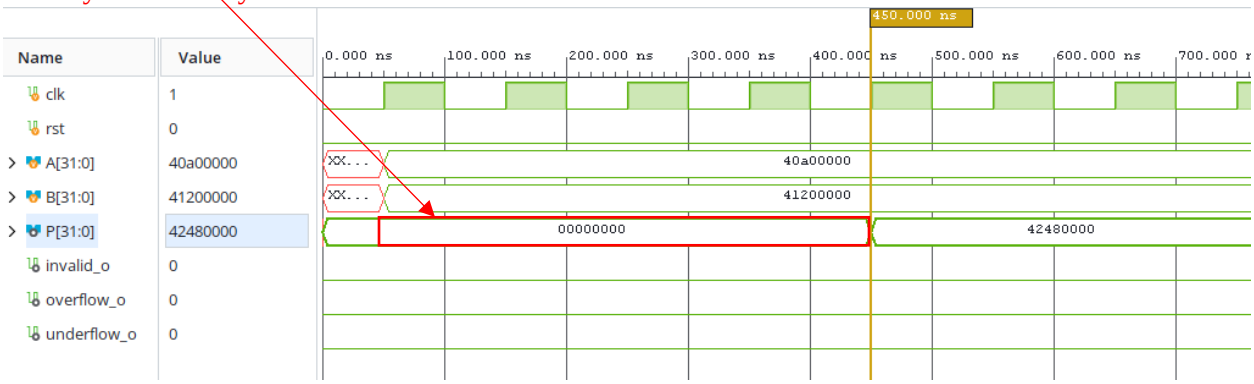

Figure 12: Case 1, DSP58 Simulation Result

Case 2:

```
Localparam AREG         = 2;
Fpopmode <= 7'b0000001; // only multiplication between two operand
```

Latency is 5 clock cycle

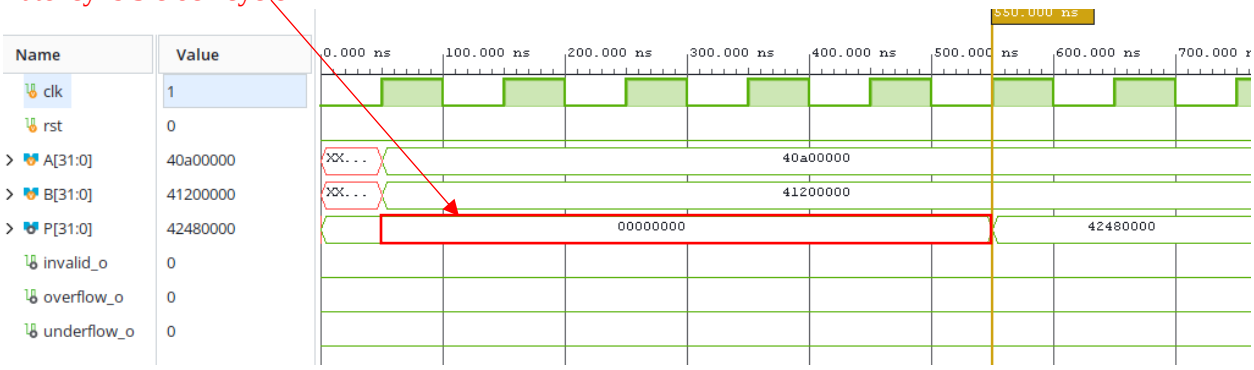

Figure 13: Case 2, DSP58 Simulation Result

Case 3:

```
localparam AREG         = 2;
fpopmode <= 7'b0010001;  // multiplication and accumulation
```

Latency is 5 clock cycle

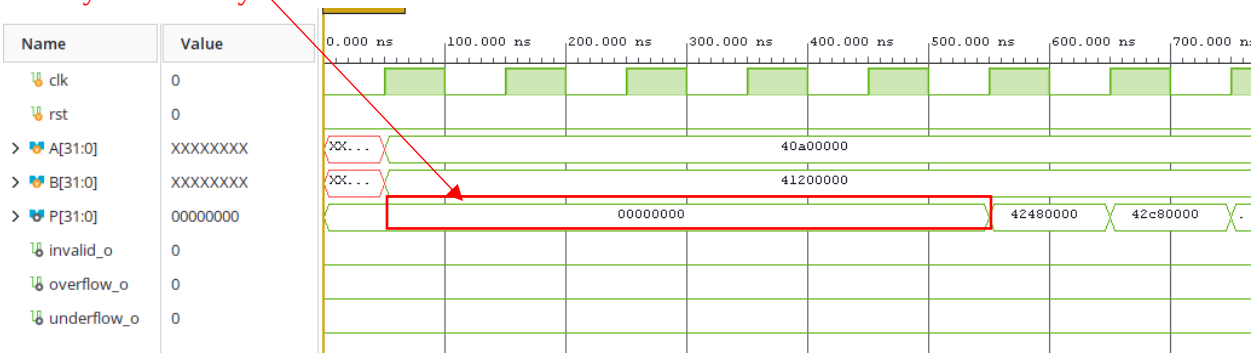

Figure 14: Case 3, DSP58 Simulation Result

The DSP58 engine can perform floating-point computations, which makes it more robust and suitable for applications such as radar, medical systems, and artificial intelligence. When AREG = 1 and BREG = 1, the computation requires 4 clock cycles. When AREG = 2 and BREG = 2, the latency increases to 5 clock cycles.

Resource Utilization Report:

```
---------------------------------------------------------------------------------
Starting Synthesize : Time (s): cpu = 00:00:10 ; elapsed = 00:00:09 . Memory (MB): peak = 1360.504 ; gain = 0.000
---------------------------------------------------------------------------------
INFO: [Synth 8-6157] synthesizing module 'DSP58_' [D:/DSP58_Experiments/DSP58_Experiments.srcs/sources_1/new/DSP58_.v:18]
INFO: [Synth 8-6157] synthesizing module 'DSPFP32' [E:/FPGA/2025.2/Vivado/scripts/rt/data/unisim_comp.v:48999]
    Parameter ACASCREG bound to: 1 - type: integer
    Parameter AREG bound to: 2 - type: integer
    Parameter A_INPUT bound to: DIRECT - type: string
    Parameter B_INPUT bound to: DIRECT - type: string
    Parameter FPA_PREG bound to: 1 - type: integer
    Parameter FPBREG bound to: 1 - type: integer
    Parameter FPCREG bound to: 0 - type: integer
    Parameter FPDREG bound to: 1 - type: integer
    Parameter FPMPIPEREG bound to: 1 - type: integer
    Parameter FPM_PREG bound to: 1 - type: integer
    Parameter FPOPMREG bound to: 1 - type: integer
    Parameter INMODEREG bound to: 0 - type: integer
    Parameter PCOUTSEL bound to: FPM - type: string
    Parameter RESET_MODE bound to: SYNC - type: string
    Parameter USE_MULT bound to: MULTIPLY - type: string
```

```
3. ARITHMETIC
-------------

+--------------------+------+-------+------------+-----------+-------+
|      Site Type     | Used | Fixed | Prohibited | Available | Util% |
+--------------------+------+-------+------------+-----------+-------+
| DSP Slices         |    1 |     0 |          0 |      1968 |  0.05 |
|   DSP58            |    0 |     0 |            |           |       |
|   DSPCPLX          |    0 |     0 |            |           |       |
|   DSPFP32          |    1 |     0 |            |           |       |
|   DSP48E5          |    0 |     0 |            |           |       |
| DSP Imux registers |    0 |     0 |            |           |       |
|   Pipelining       |    0 |       |            |           |       |
+--------------------+------+-------+------------+-----------+-------+
```

Figure 15: DSP58 Resource Utilization Report

Conclusion

This article provides a detailed explanation of the DSP48E1, DSP48E2, and DSP58 slices in FPGA, along with coding examples, which help optimize DSP-related applications.

The DSP58 engine has one additional clock cycle of latency compared to the other two primitives however, it offers a wider range of features and supports floating-point computations. This makes the DSP58 engine more robust and suitable for advanced applications such as artificial intelligence, medical systems, and space science.

References

[1] J. G. Proakis and D. G. Manolakis, “DIGITAL SIGNAL PROCESSING Principles, Algorithms, m l Applications,” 1996.

[2] U. Meyer-Baese, “Signals and Communication Technology Digital Signal Processing with Field Programmable Gate Arrays.” [Online]. Available: http://www.springer.com/series/4748

[3] G. Brignone, R. Bosio, F. Ottati, C. Sansoè, and L. Lavagno, “SILVIA: Automated Superword-Level Parallelism Exploitation via HLS-specific LLVM Passes for Compute-Intensive FPGA Accelerators,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 2, Mar. 2025, doi: 10.1145/3705324.

[4] Xilinx and Inc, “7 Series DSP48E1 Slice User Guide (UG479),” 2011. [Online]. Available: http://www.xilinx.com/legal.htm#tos.

[5] Xilinx and Inc, “UltraScale Architecture DSP Slice User Guide.” [Online]. Available: www.xilinx.com

[6] “Versal ACAP DSP Engine Architecture Manual AM004 (v1.2.1),” 2022. [Online]. Available: www.xilinx.com