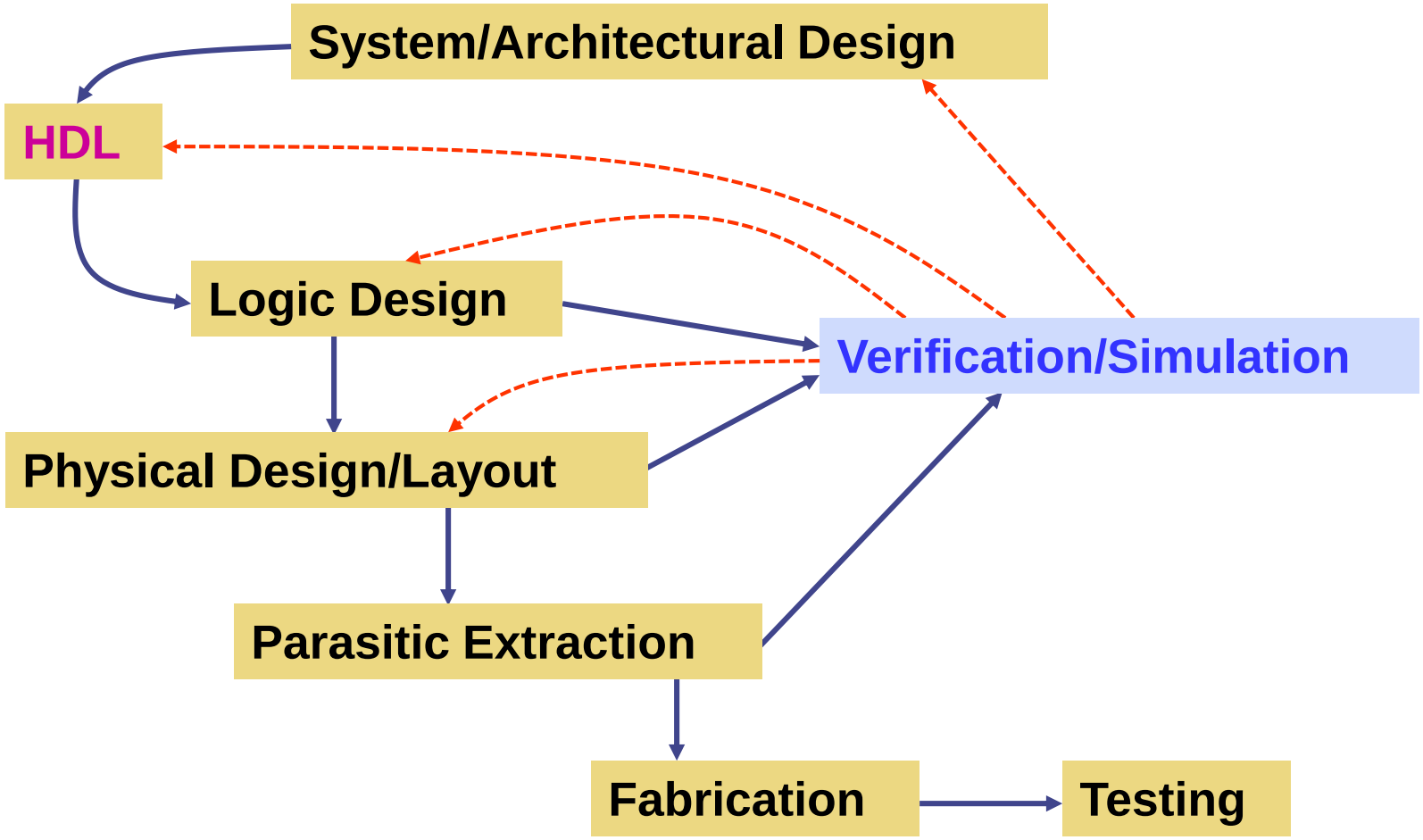
# Case Study RGB to YUV

# Basic Design Flow



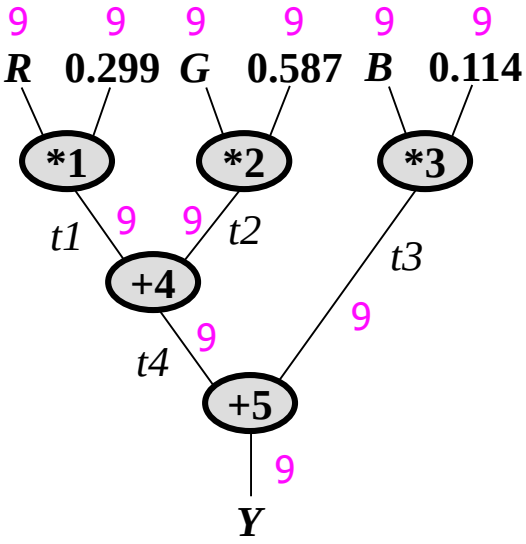
# RGB to YUV

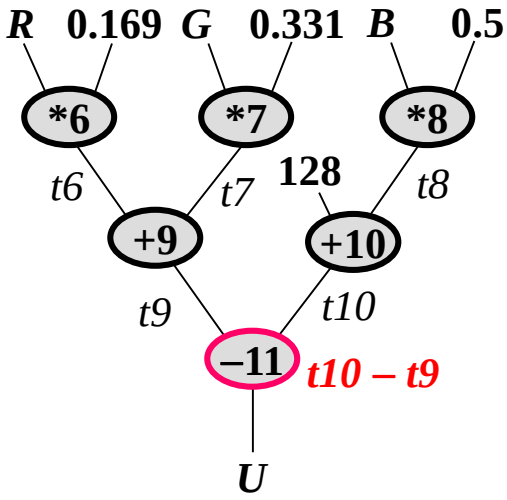
$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B \qquad Y \in [0, 255]$$

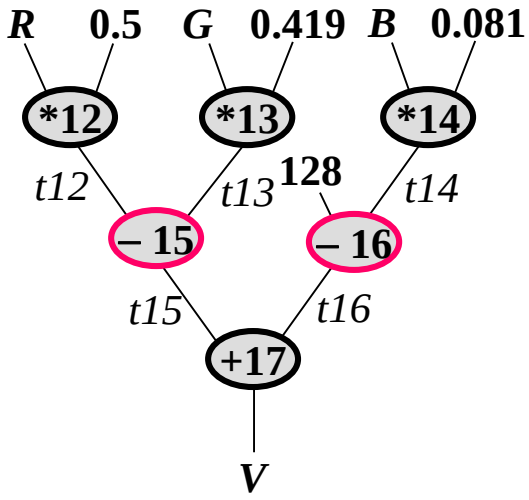
$$U = -0.169 \times R - 0.331 \times G + 0.5 \times B + 128 \qquad U \in [0, 255]$$

$$V = 0.5 \times R - 0.419 \times G - 0.081 \times B + 128 \qquad V \in [0, 255]$$

two's complement



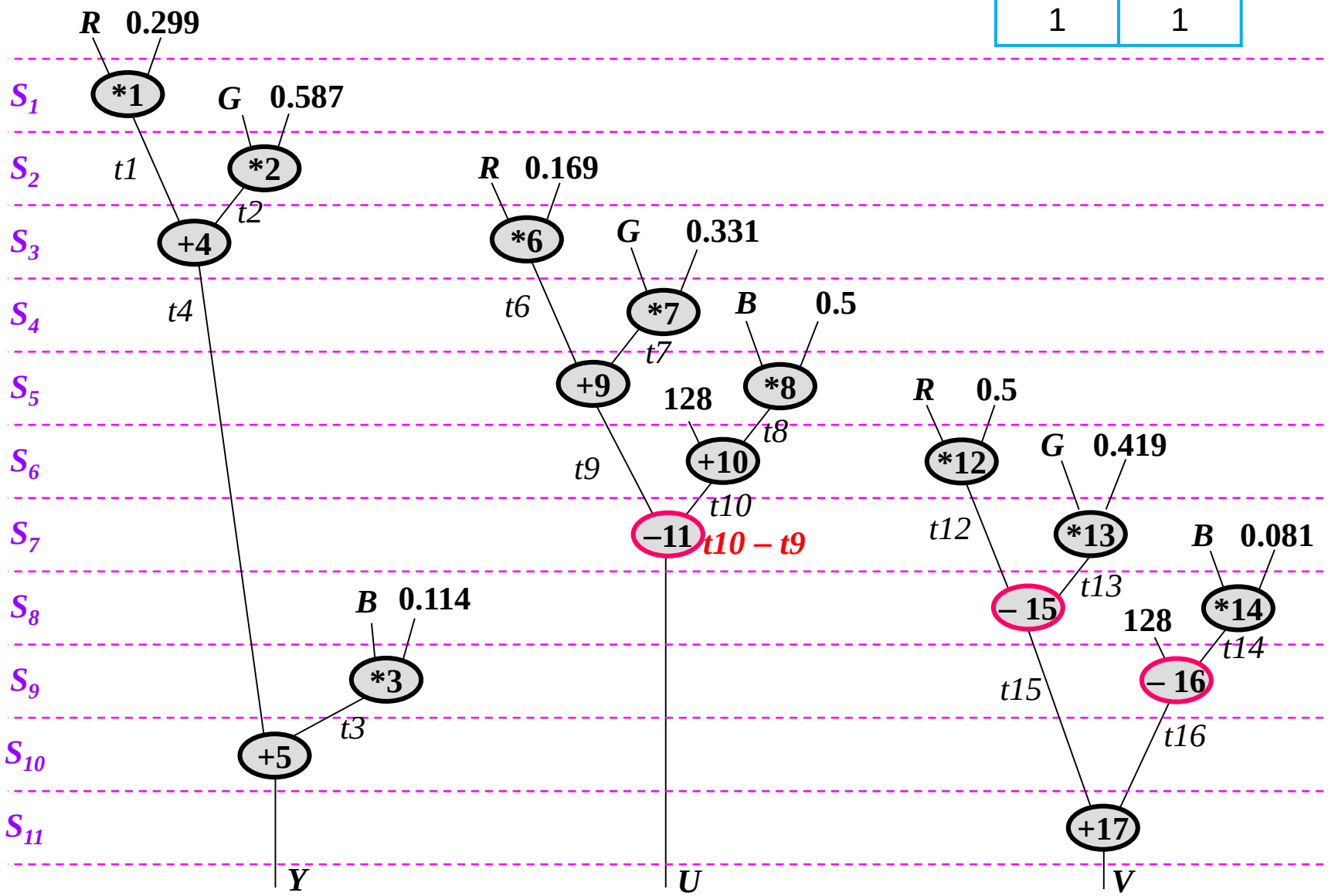



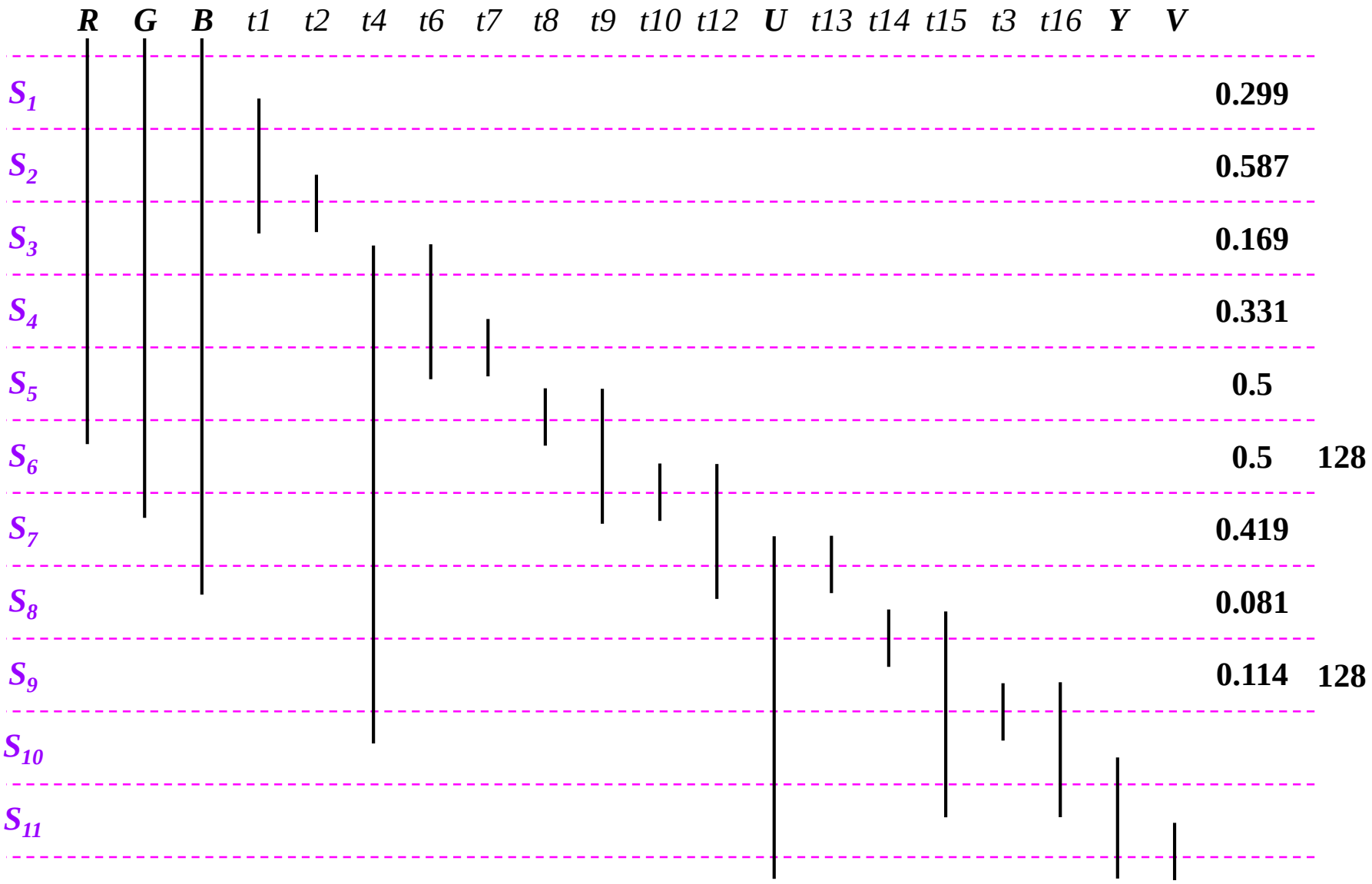
# Scheduling of RGB to YUV

**Resource constraint:**

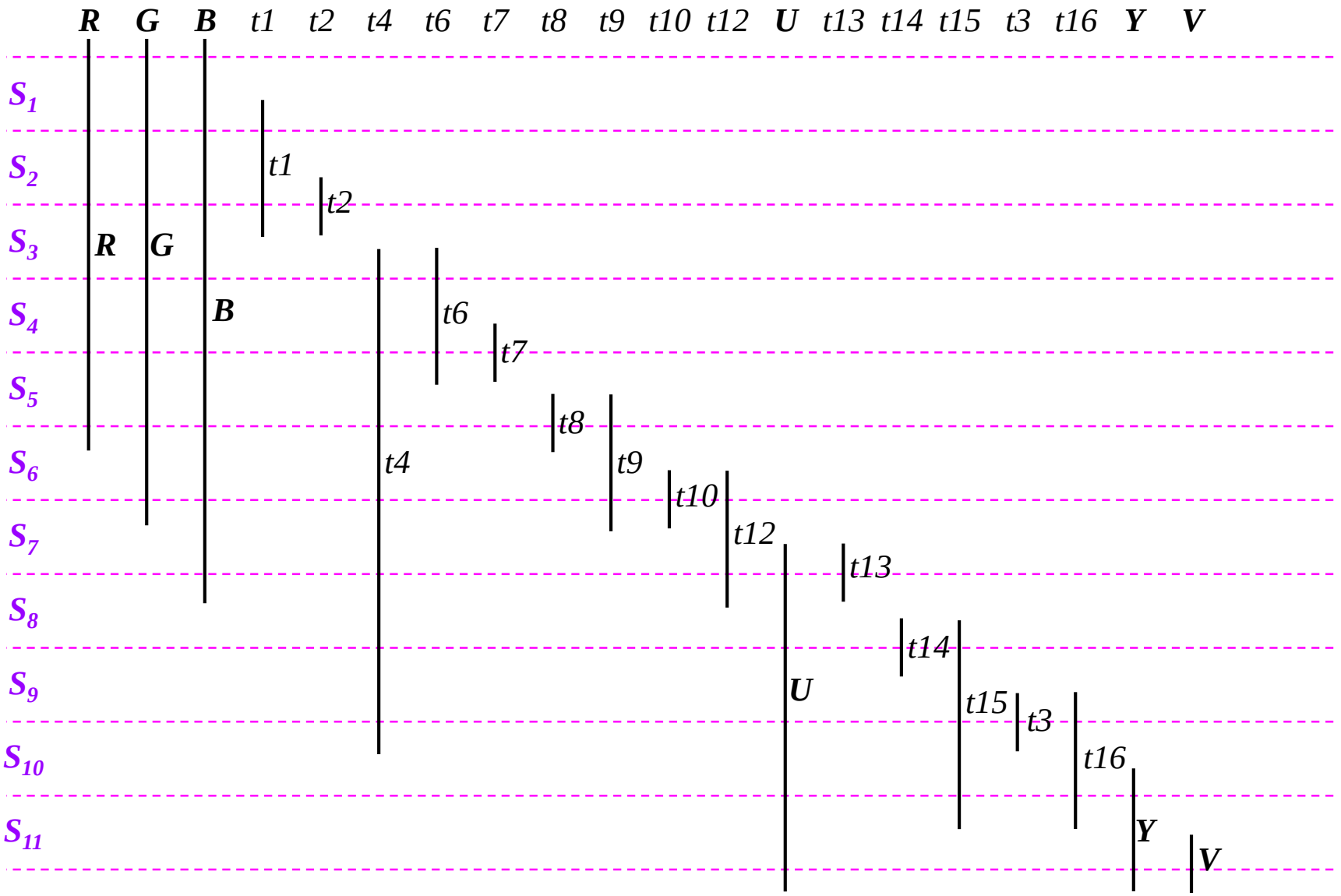
| * | + |
|---|---|
| 1 | 1 |



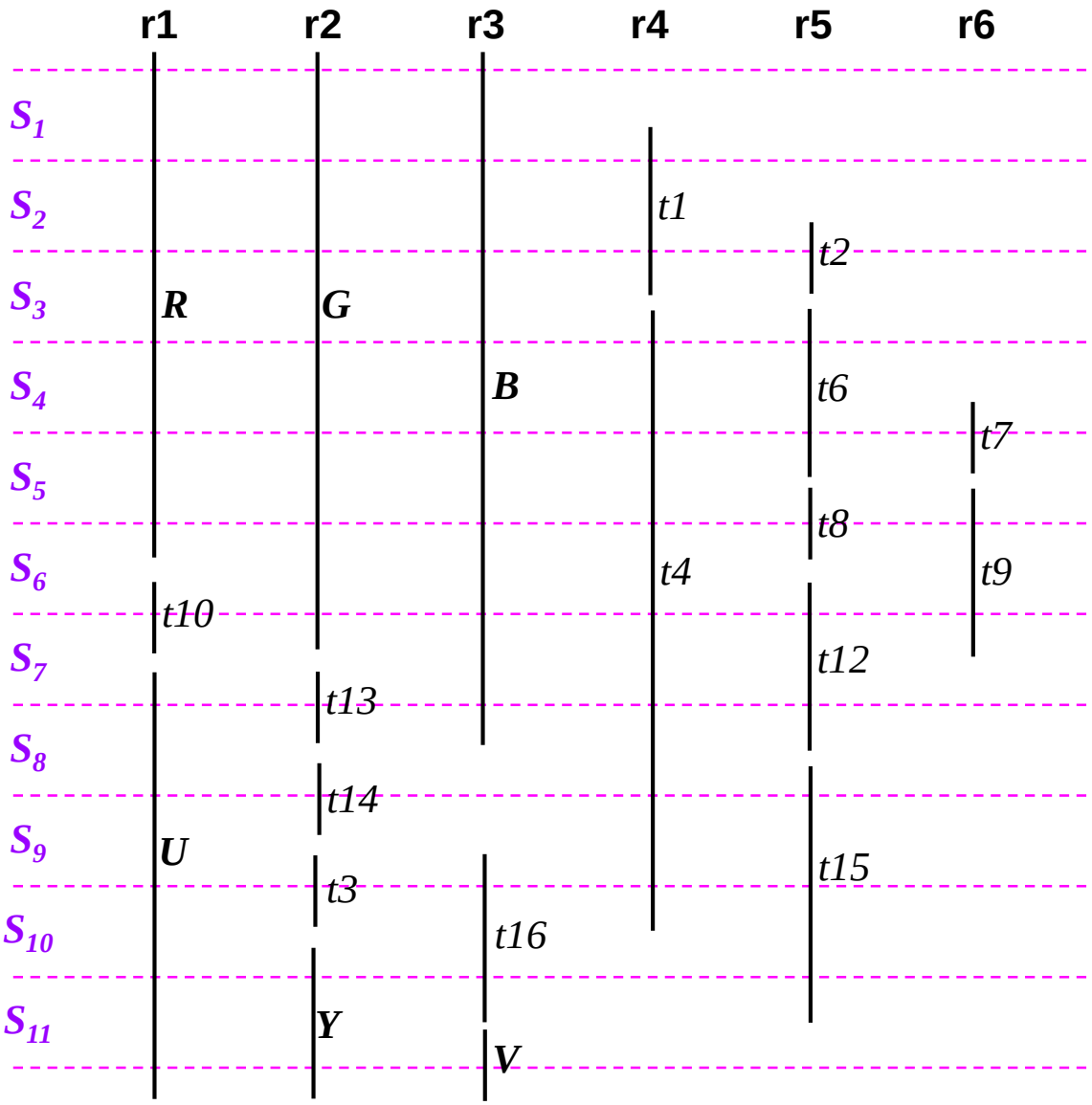
# Allocation of RGB to YUV (1/3)



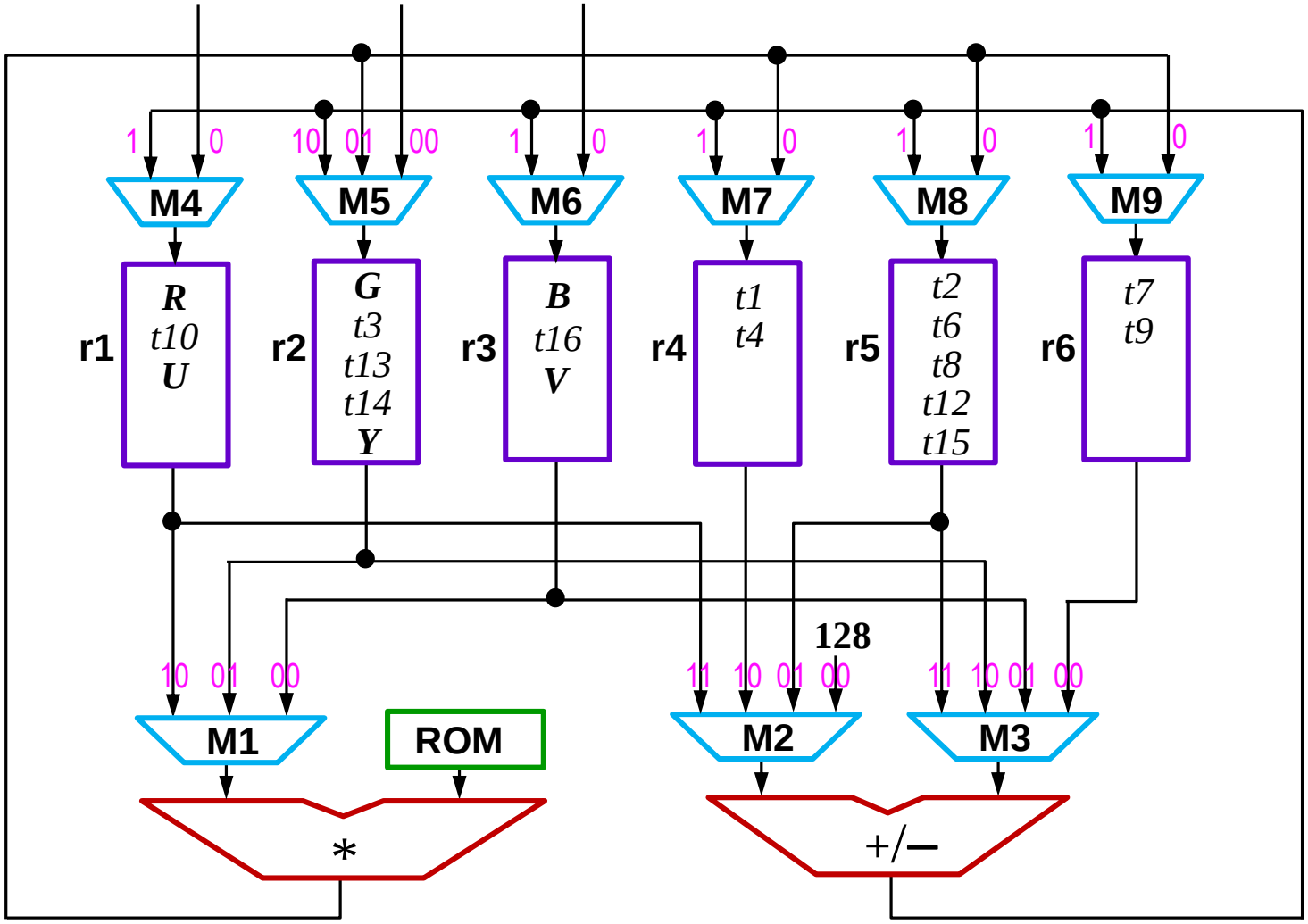
# Allocation of RGB to YUV (2/3)



# Allocation of RGB to YUV (3/3)



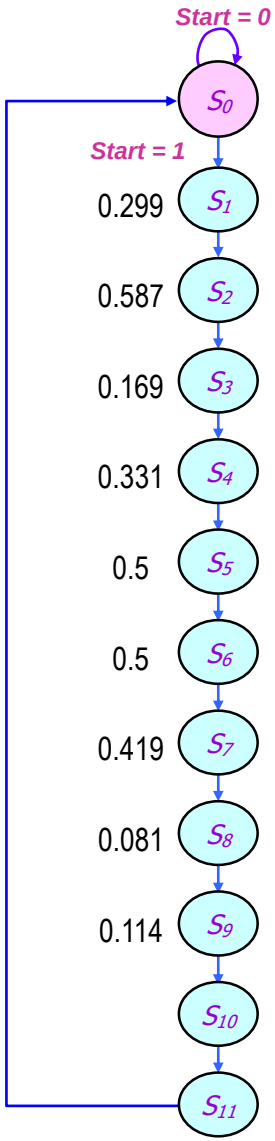
# Datapath of RGB to YUV




**ROM**

| address | value |
|---|---|
| 000 | 0.299 |
| 001 | 0.587 |
| 010 | 0.169 |
| 011 | 0.331 |
| 100 | 0.5 |
| 101 | 0.419 |
| 110 | 0.081 |
| 111 | 0.114 |

# STG and State Table of RGB to YUV (1/2)




| Present State | Input | Next State | Control Signals | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | r1 | r2 | r3 | r4 | r5 | r6 | ROM | done |
| S0 | Start = 0 | S0 | 1 | 1 | 1 | 0 | 0 | 0 | - | 0 |
| | Start = 1 | S1 | | | | | | | | |
| S1 | - | S2 | 0 | 0 | 0 | 1 | 0 | 0 | 000 | 0 |
| S2 | - | S3 | 0 | 0 | 0 | 0 | 1 | 0 | 001 | 0 |
| S3 | - | S4 | 0 | 0 | 0 | 1 | 1 | 0 | 010 | 0 |
| S4 | - | S5 | 0 | 0 | 0 | 0 | 0 | 1 | 011 | 0 |
| S5 | - | S6 | 0 | 0 | 0 | 0 | 1 | 1 | 100 | 0 |
| S6 | - | S7 | 1 | 0 | 0 | 0 | 1 | 0 | 100 | 0 |
| S7 | - | S8 | 1 | 1 | 0 | 0 | 0 | 0 | 101 | 0 |
| S8 | - | S9 | 0 | 1 | 0 | 0 | 1 | 0 | 110 | 0 |
| S9 | - | S10 | 0 | 1 | 1 | 0 | 0 | 0 | 111 | 0 |
| S10 | - | S11 | 0 | 1 | 0 | 0 | 0 | 0 | - | 0 |
| S11 | - | S0 | 0 | 0 | 1 | 0 | 0 | 0 | - | 1 |

# STG and State Table of RGB to YUV (2/2)

| Present State | Input | Next State | Control Signals | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 | M9 |
| S0 | Start = 0 | S0 | - | - | - | 0 | 00 | 0 | - | - | - |
| | Start = 1 | S1 | | | | | | | | | |
| S1 | - | S2 | 10 | - | - | - | - | - | 0 | - | - |
| S2 | - | S3 | 01 | - | - | - | - | - | - | 0 | - |
| S3 | - | S4 | 10 | 10 | 11 | - | - | - | 1 | 0 | - |
| S4 | - | S5 | 01 | - | - | - | - | - | - | - | 0 |
| S5 | - | S6 | 00 | 01 | 00 | - | - | - | - | 0 | 1 |
| S6 | - | S7 | 10 | 00 | 11 | 1 | - | - | - | 0 | - |
| S7 | - | S8 | 01 | 11 | 00 | 1 | 01 | - | - | - | - |
| S8 | - | S9 | 00 | 01 | 10 | - | 01 | - | - | 1 | - |
| S9 | - | S10 | 00 | 00 | 10 | - | 01 | 1 | - | - | - |
| S10 | - | S11 | - | 10 | 10 | - | 10 | - | - | - | - |
| S11 | - | S0 | - | 01 | 01 | - | - | 1 | - | - | - |

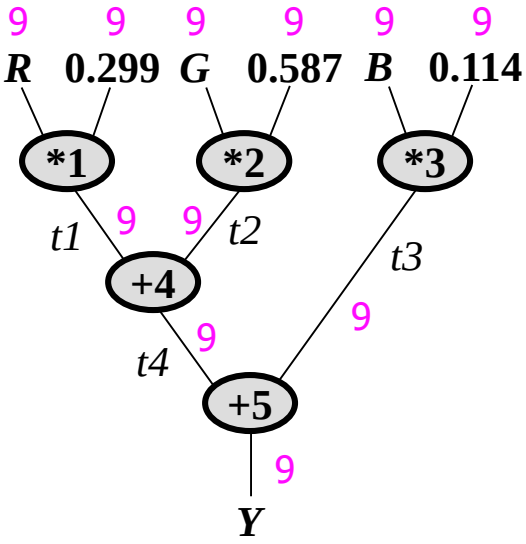
# High-level Transformations (1/2)

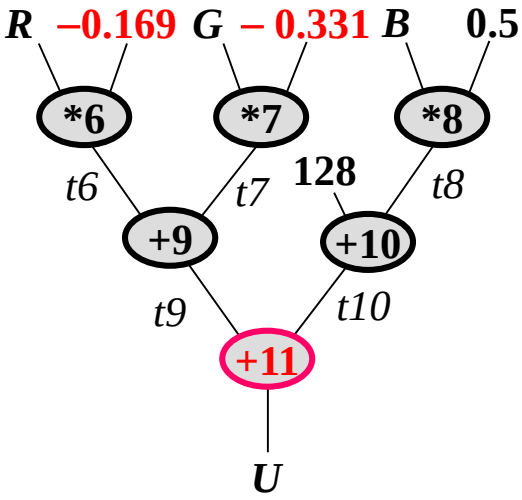
$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B \qquad Y \in [0, 255]$$

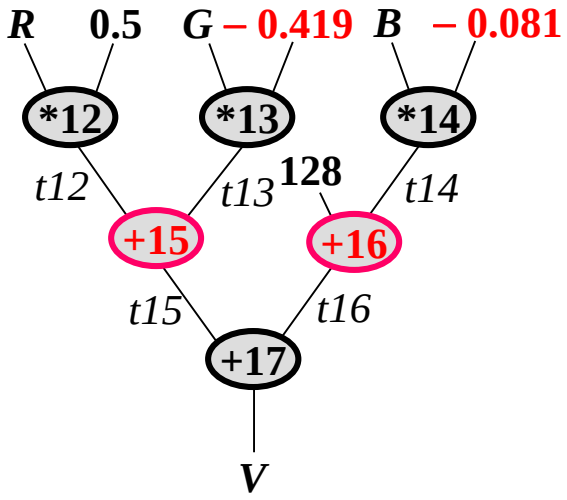
$$U = (-0.169) \times R + (-0.331) \times G + 0.5 \times B + 128 \qquad U \in [0, 255]$$

$$V = 0.5 \times R + (-0.419) \times G + (-0.081) \times B + 128 \qquad V \in [0, 255]$$

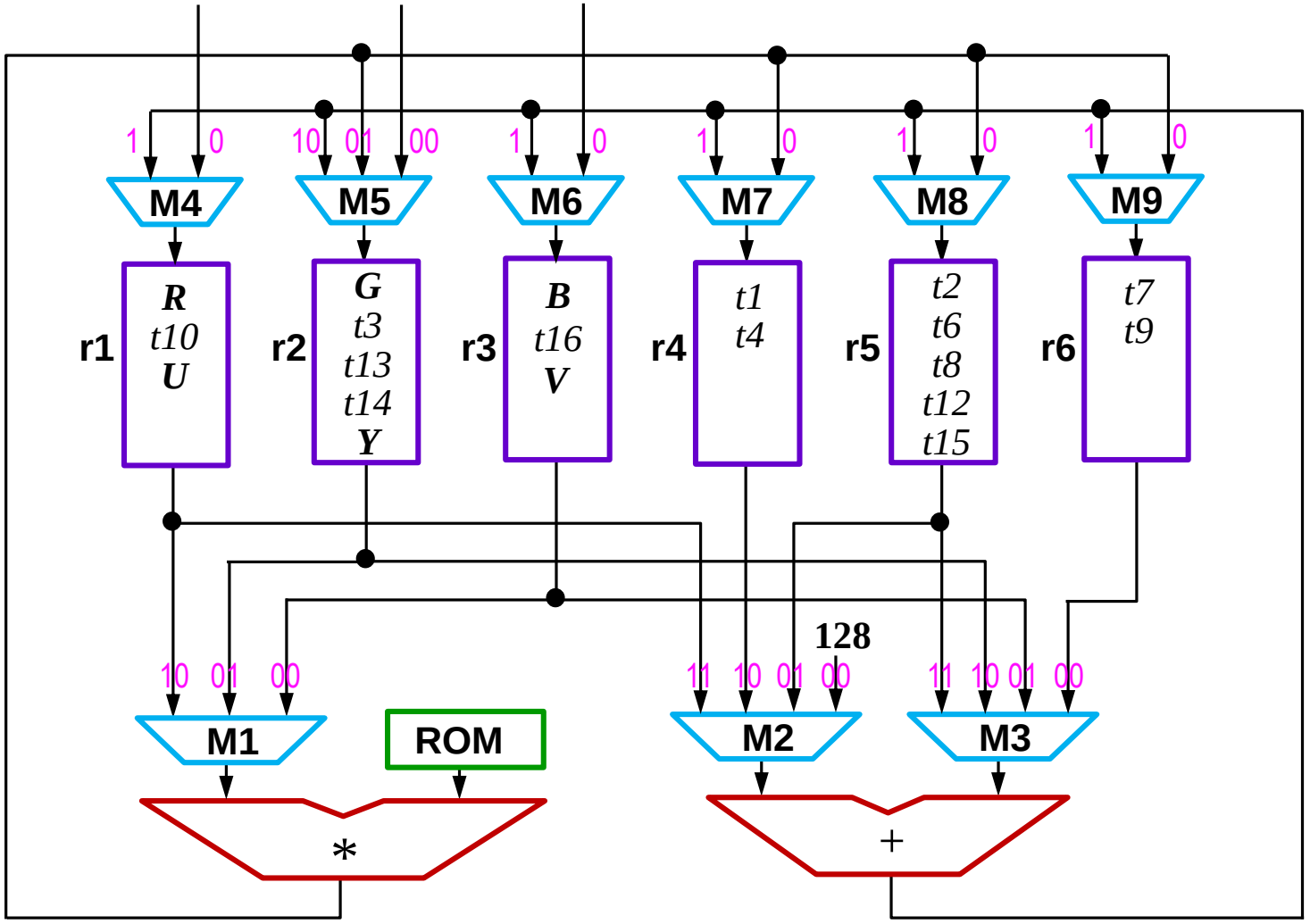
two's complement



# High-level Transformations (2/2)




**ROM**

| address | value |
|---|---|
| 000 | 0.299 |
| 001 | 0.587 |
| 010 | –0.169 |
| 011 | –0.331 |
| 100 | 0.5 |
| 101 | –0.419 |
| 110 | –0.081 |
| 111 | 0.114 |

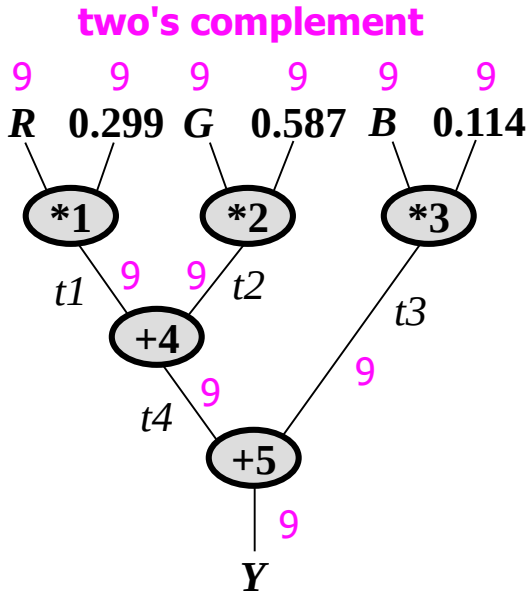
# Verilog of RGB to YUV (1/8)

Multipliper.v

```
`define bits 9

module Mul(A, B, Mul);
  input signed [`bits-1:0] A, B;
  output [`bits-1:0] Mul;
  wire s;
  wire [7:0] N;

  assign {s, Mul, N} = A * B;

endmodule
```

Add.v

```
module Add(A, B, Add);
  input signed [`bits-1:0] A, B;
  output [`bits-1:0] Add;

  assign Add = A + B;

endmodule
```



# Verilog of RGB to YUV (2/8)

MUX4.v

```
`define bits 9

module MUX4 (
  input  [`bits-1:0] A, B, C, D,
  input  [1:0] S,
  output reg [`bits-1:0] Y
 );

 always @(*) begin
   case (S)
     2'b00: Y = A;
     2'b01: Y = B;
     2'b10: Y = C;
     2'b11: Y = D;
   endcase
 end

endmodule
```

MUX3.v

```
`define bits 9

module MUX3 (
 input  [`bits-1:0] A, B, C,
 input  [1:0] S,
 output reg [`bits-1:0] Y
 );

 always @(*) begin
   case (S)
     2'b00: Y = A;
     2'b01: Y = B;
     2'b10: Y = C;
    default : Y = A;
   endcase
 end

endmodule
```

# Verilog of RGB to YUV (3/8)

Register.v

```
`define bits 9

module Register (
   input  [`bits-1:0] D,
   input  reset, clk, load,
   output reg [`bits-1:0] Q
  );

  always @(posedge clk or negedge reset) begin
     if(!reset)
         Q <= 9'b0;
     else if(load)
        Q <= D;
  end

endmodule
```

# Verilog of RGB to YUV (4/8)

ROM.v

```
module ROM (clk, addr, data);
  input       clk;
  input [2:0] addr;
  output reg [8:0] data;

  always @(*)
  begin
      case(addr)
          3'b000: data <= 9'b001001100;
          3'b001: data <= 9'b010010110;
          3'b010: data <= 9'b111010101;
          3'b011: data <= 9'b110101100;
          3'b100: data <= … ;
          …
       endcase
   end
endmodule
```

ROM

| address | value |
|---|---|
| 000 | 0.299 |
| 001 | 0.587 |
| 010 | –0.169 |
| 011 | –0.331 |
| 100 | 0.5 |
| 101 | –0.419 |
| 110 | –0.081 |
| 111 | 0.114 |

# Verilog of RGB to YUV (5/8)

## Datapath.v

```
`timescale 1ns/1ps
`define bits 9

module Datapath ( inportR, inportG, inportB, control, clk, rst_n, outportY, outportU, outportV, done );
  input  [`bits-1:0] inportR, inportG, inportB;
  input [21:0] control;
  input  clk, rst_n, done;
  output [`bits-1:0] outportY, outportU, outportV;

  wire [`bits-1:0] M1_OUT, M2_OUT, M3_OUT, M4_OUT, M5_OUT, M6_OUT, M7_OUT;
  wire [`bits-1:0] M8_OUT, M9_OUT, Fadd, Fmul, R1, R2, R3, R4, R5, R6, data;

  Register r1( .D(M4_OUT), .reset(rst_n), .clk(clk), .load(control[21:21]), .Q(R1) );
  Register r2( .D(M5_OUT), .reset(rst_n), .clk(clk), .load(control[20:20]), .Q(R2) );
  …

  Register r6( .D(M9_OUT), .reset(rst_n), .clk(clk), .load(control[16:16]), .Q(R6) );
  MUX3 M1 ( .A(R3), .B(R2), .C(R1), .S(control[12:11]), .Y(M1_OUT) );
  MUX4 M2 ( .A(9'b010000000), .B(R5), .C(R4), .D(R1), .S(control[10:9]), .Y(M2_OUT) );
  MUX4 M3 ( .A(R6), .B(R3), .C(R2), .D(R5), .S(control[8:7]), .Y(M3_OUT) );
  …

  MUX2 M9 ( .A(Fmul), .B(Fadd), .S(control[0:0]), .Y(M9_OUT) );
  Mul FU1 ( .A(M1_OUT), .B(data), .Mul(Fmul) );
  Add FU2 ( .A(M2_OUT), .B(M3_OUT), .Add(Fadd) );
  ROM FU3 ( .clk(clk), .addr(control[15:13]), .data(data) );

  Register r7(R2, rst_n, clk, done, outportY);
  Register r9(R1, rst_n, clk, done, outportU);
  Register r8(R3, rst_n, clk, done, outportV);
endmodule
```

# Verilog of RGB to YUV (6/8)

## Controller.v

```
`timescale 1ns/1ps
`define bits 9

`define S0 4'b0000
`define S1 4'b0001
`define S2 4'b0010
`define S3 4'b0011
`define S4 4'b0100
`define S5 4'b0101
`define S6 4'b0110
`define S7 4'b0111
`define S8 4'b1000
`define S9 4'b1001
`define S10 4'b1010
`define S11 4'b1011

module Controller ( start, rst_n, clk, done, control);
  input start, rst_n, clk;
  output reg   done;
  output reg   [21:0] control;

  reg [3:0] Current_State, Next_State;

  always @(posedge clk or negedge rst_n) begin
    if(!rst_n)   Current_State <= `S0;
    else   Current_State <= Next_State;
  end
```

```
always @(Current_State or start)
begin
   case (Current_State)
      `S0:
      begin
         control = 22'b1_1_1_0_0_0_000_00_00_00_0_00_0_0_0_0;
         done = 1'b0;
         if(~start)  Next_State = `S0;
         else  Next_State = `S1;
      end
      `S1:
      begin
         control = 22'b0_0_0_1_0_0_000_10_00_00_0_00_0_0_0_0;
         done = 1'b0;
         Next_State = `S2;
      end
       …
      `S11:
      begin
         control = 22'b0_0_1_0_0_0_000_00_01_01_0_00_1_0_0_0;
         done = 1'b1;
         Next_State = `S0;
      end
      default:
      begin
         control= 22'b0_0_1_0_0_0_000_00_01_01_0_00_1_0_0_0;
         done = 1'b1;
         Next_State = `S0;
      end
   endcase
end

endmodule
```

# Verilog of RGB to YUV (7/8)

RGB2YUV.v

```
`timescale 1ns/1ps
`define bits 9

module RGB2YUV (start, clk, rst_n, inportR, inportG, inportB, done, outportY, outportU, outportV);
  input  start, clk, rst_n;
  input  [`bits-1:0] inportR, inportG, inportB;
  output done;
  output [`bits-1:0] outportY, outportU, outportV;
  wire [21:0] control;

  Controller Controller( .start(start), .rst_n(rst_n), .clk(clk), .done(done), .control(control) );
  Datapath Datapath( .inportR(inportR), .inportG(inportG), .inportB(inportB), .control(control), .clk(clk),
        .rst_n(rst_n), .outportY(outportY), .outportU(outportU), .outportV(outportV) ,.done(done) );

endmodule
```

# Verilog of RGB to YUV (8/8)

## testbench.v

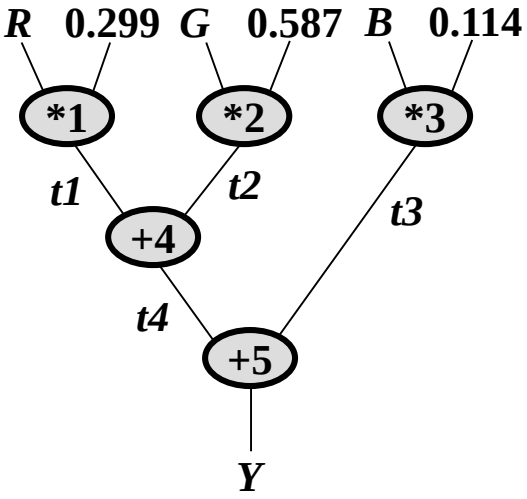
```
`timescale 1ns/1ns
`define bits 9

module testbench();
  parameter half_clk = 20;
  parameter clk_period = 2 * half_clk;
  integer imageIN,imageOUTY,imageOUTU,imageOUTV, i, cc, j;
  integer bmp_width, bmp_hight, data_start_index, bmp_size;
  reg [7:0]  bmp_data [0:2000000];
  reg rst,clk,start;
  wire [8:0] outY,outU,outV;
  wire done;

initial begin
  clk = 1'b0;
  rst = 1'b1;
  #(clk_period) rst = ~rst;
  #(clk_period) rst = ~rst;
end

always
  #(half_clk) clk = ~clk;

RGB2YUV rgbtuv(start,clk,rst,…,done,outY,outU,outV);
```

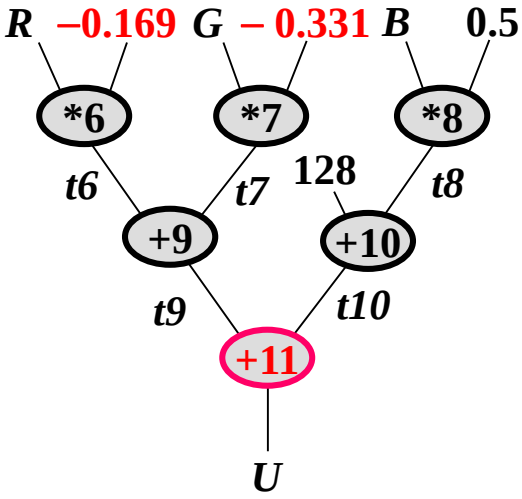
```
initial begin
  start= 1'b0;
  imageIN = $fopen("mountain256.bmp","rb");
  imageOUTY = $fopen("mountain256Y.bmp","wb");
  imageOUTU = $fopen("mountain256U.bmp","wb");
  imageOUTV = $fopen("mountain256V.bmp","wb");
  cc = $fread(bmp_data,imageIN);
  …

  for(j = 0; j < 54; j = j + 1) begin
    $fwrite(imageOUTY,"%c",bmp_data[j]);
    $fwrite(imageOUTU,"%c",bmp_data[j]);
    $fwrite(imageOUTV,"%c",bmp_data[j]);
  end

  …

  #(clk_period*2)
  $fclose(imageOUTY);
  $fclose(imageOUTU);
  $fclose(imageOUTV);
  $fclose(imageIN);
  $stop;

 end

endmodule
```

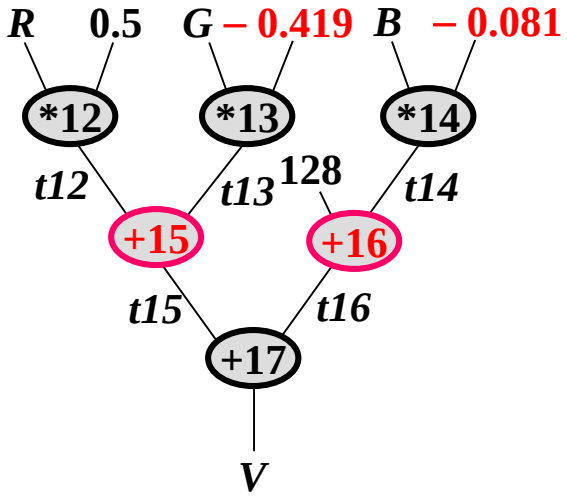
# Different Design (1/5)

**Resource constraints:**

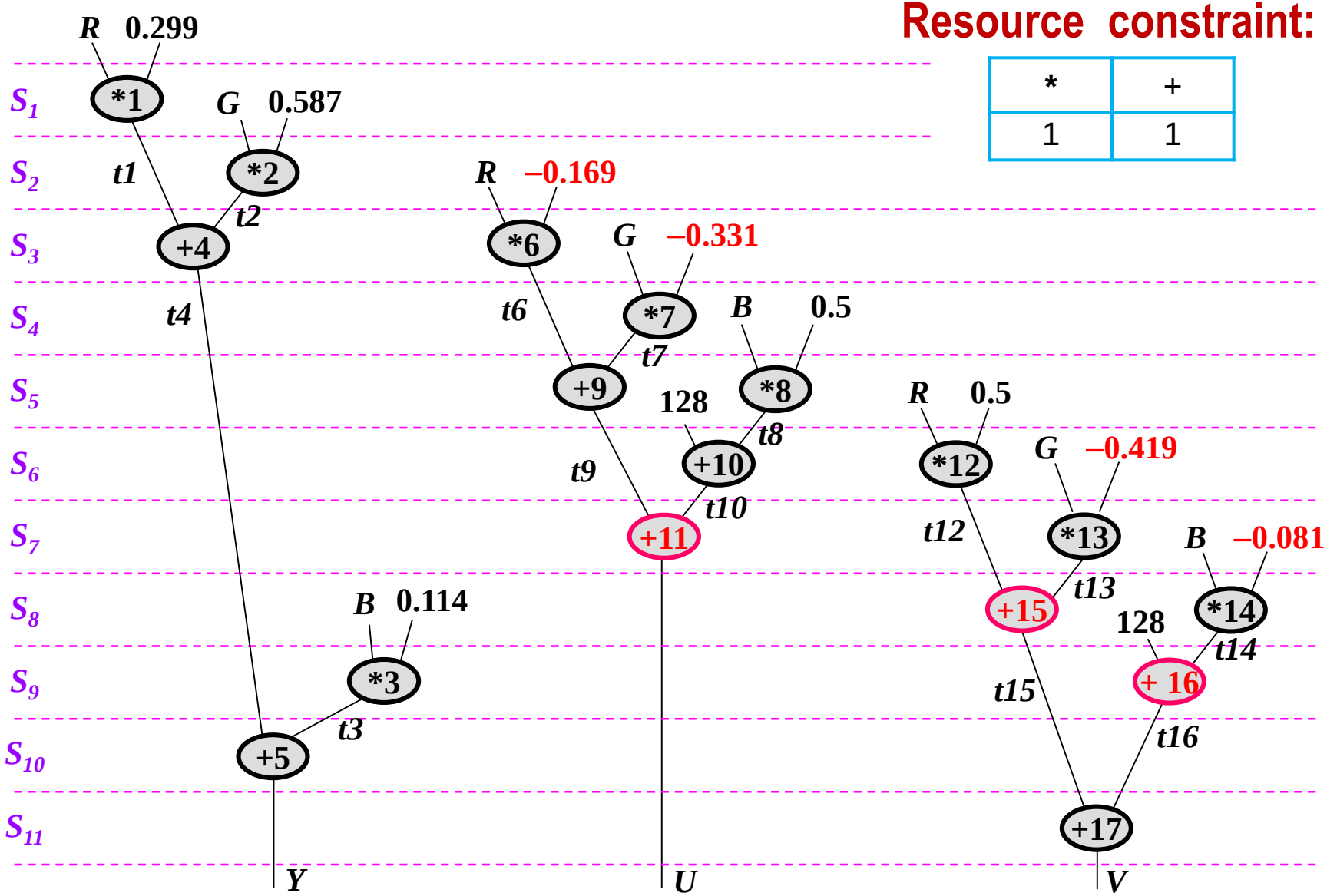
| * | + |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 1 | 3 |
| 2 | 1 |
| 2 | 2 |
| 2 | 3 |
| 3 | 1 |
| 3 | 2 |
| 3 | 3 |



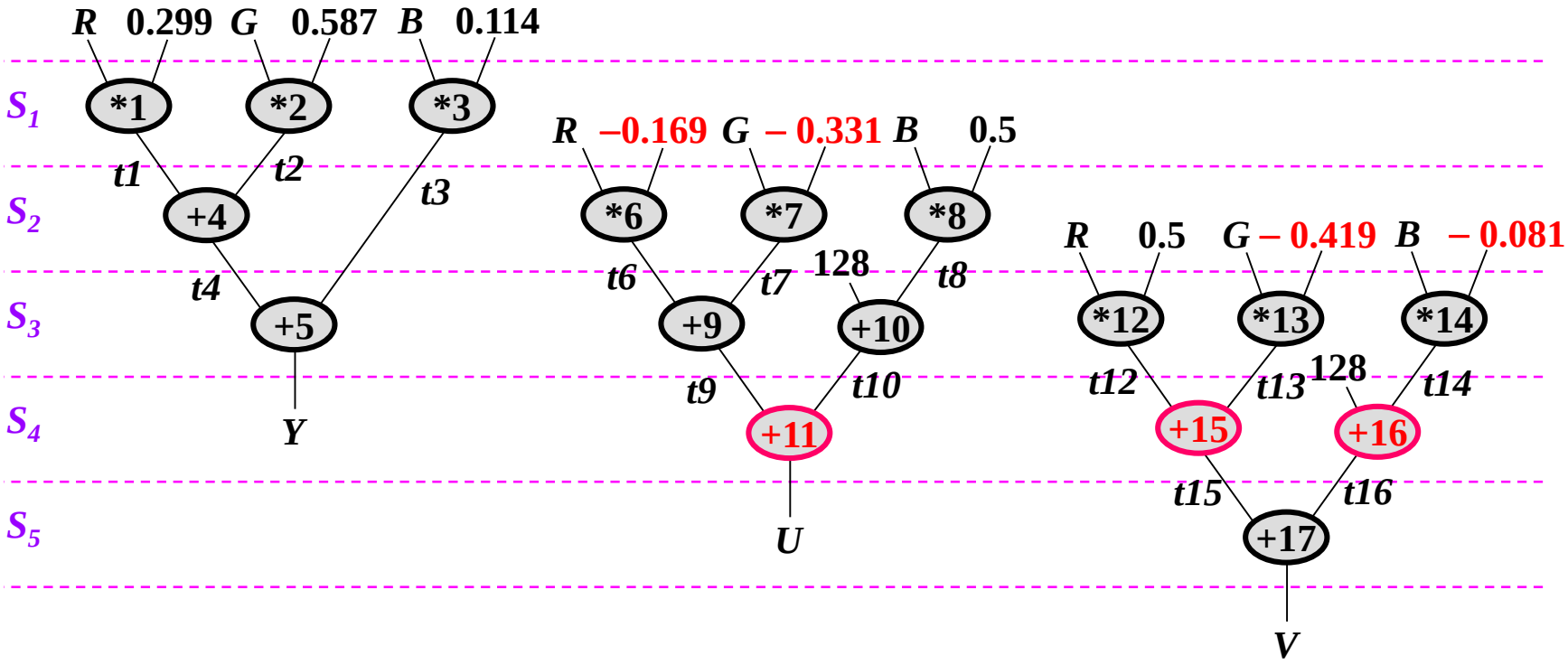
# Different Design (2/5)



# Different Design (3/5)

**Resource constraint:**

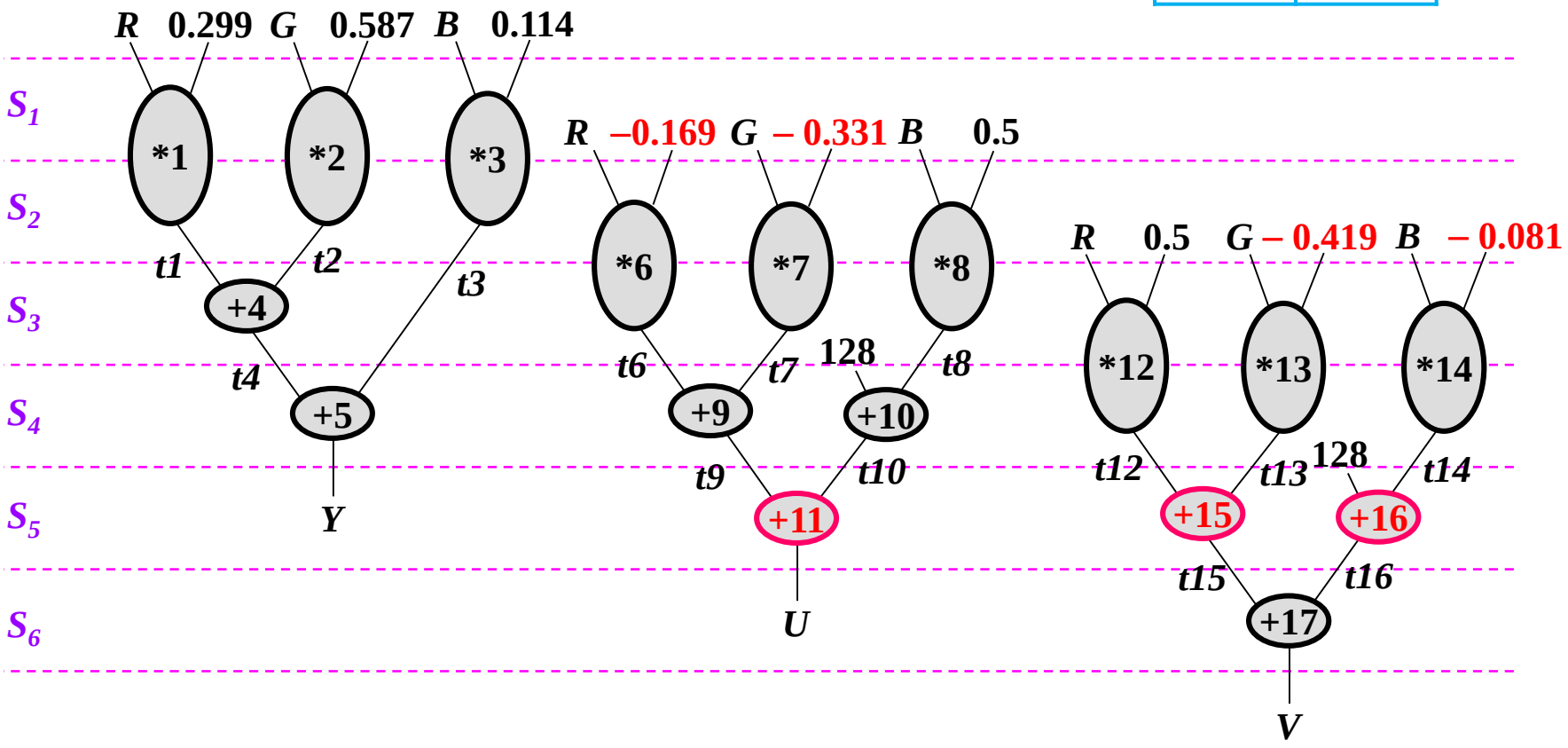
| * | + |
|---|---|
| 3 | 3 |



# Different Design (4/5)

**Resource constraint:**

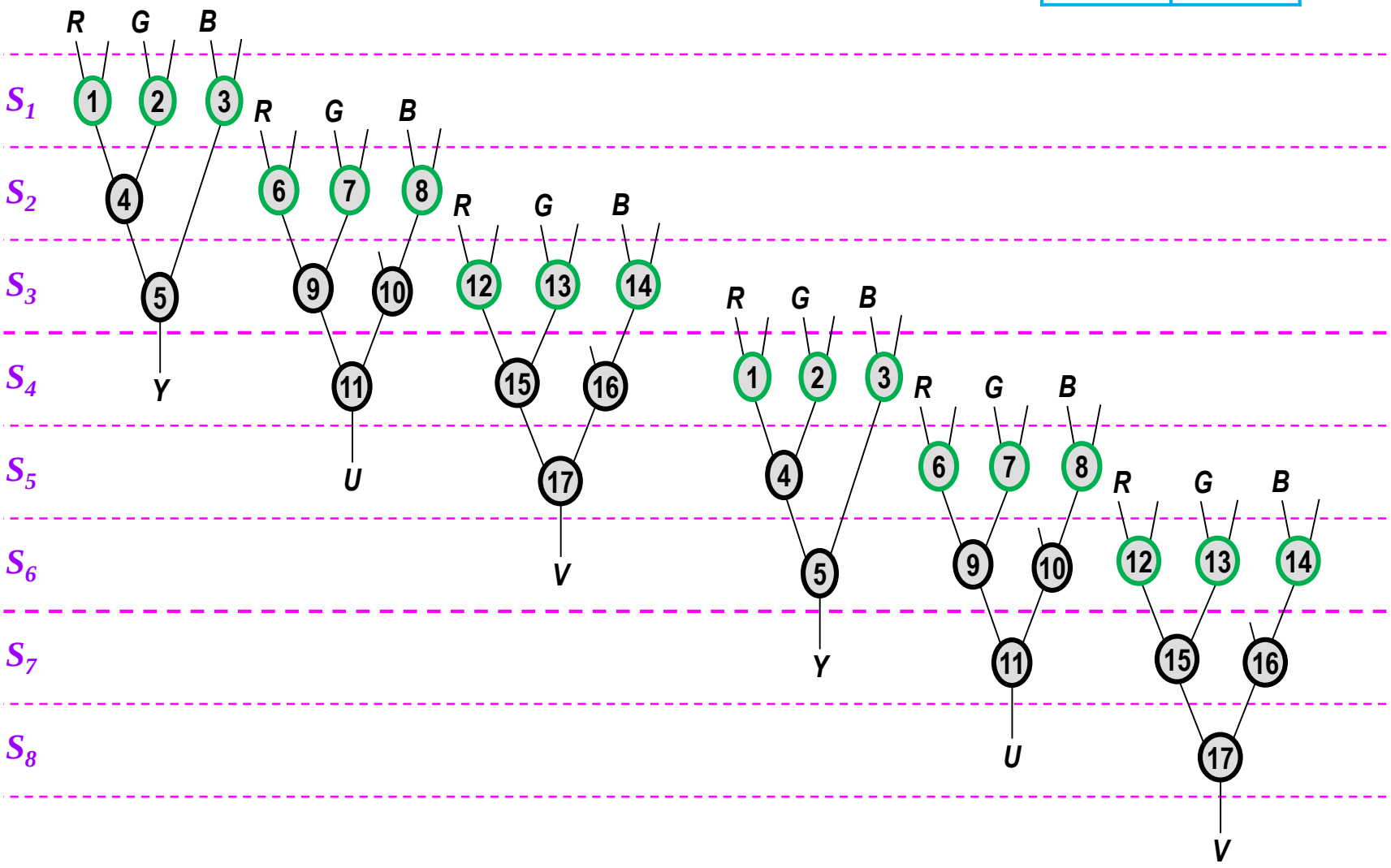
| * | + |
|---|---|
| 3 | 3 |



# Different Design (5/5)

**Resource constraint:**

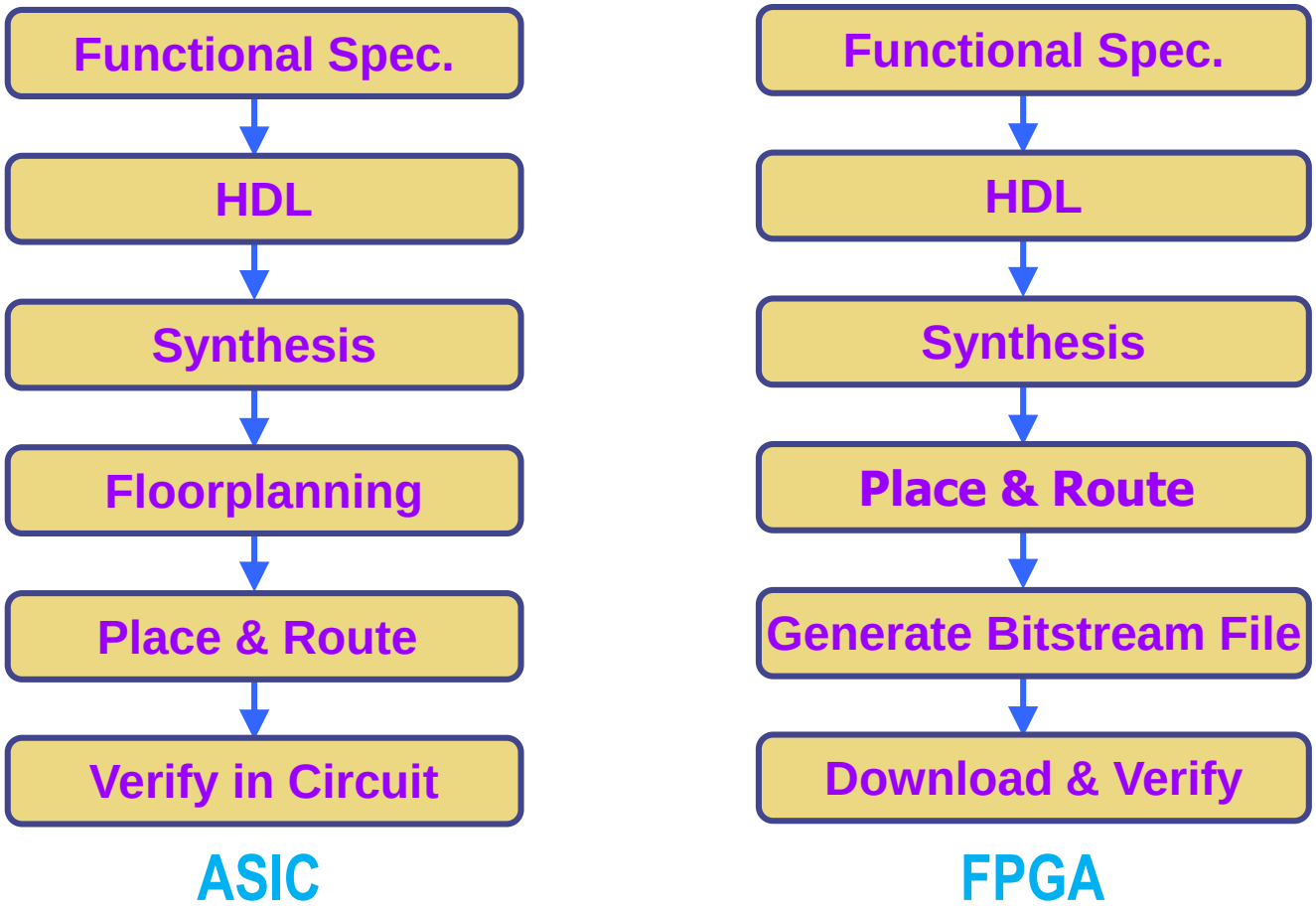
| * | + |
|---|---|
| 3 | 3 |



# Xilinx Vivado High-Level Synthesis

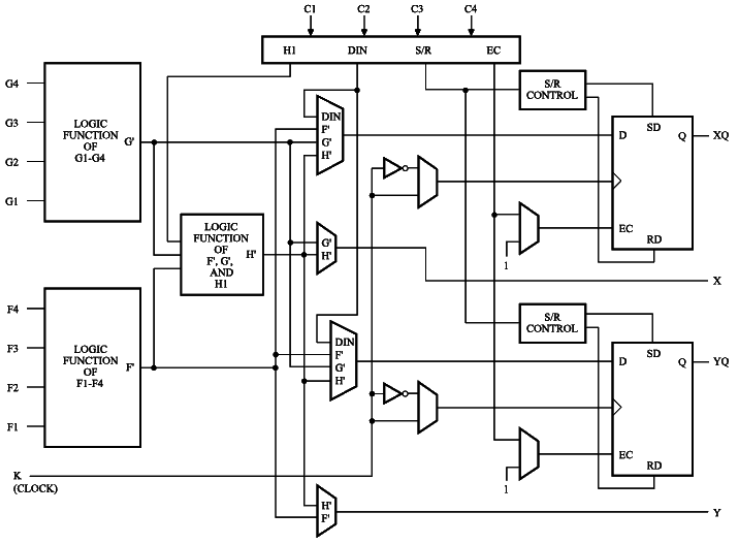
# VLSI Design Flow in SoC Era (6/6)

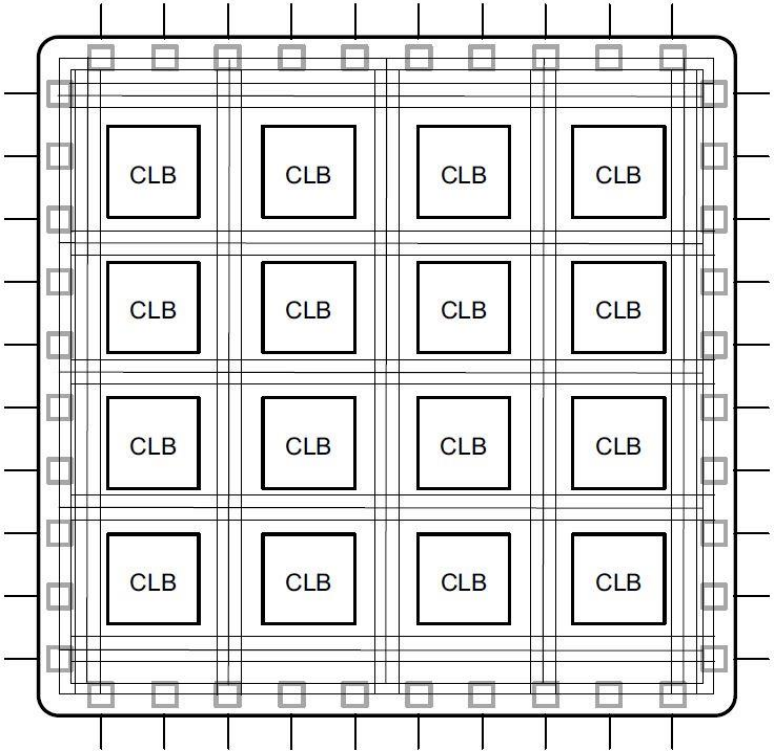
## Traditional ASIC and FPGA Design Flow



# Xilinx FPGA Architecture (1/5)

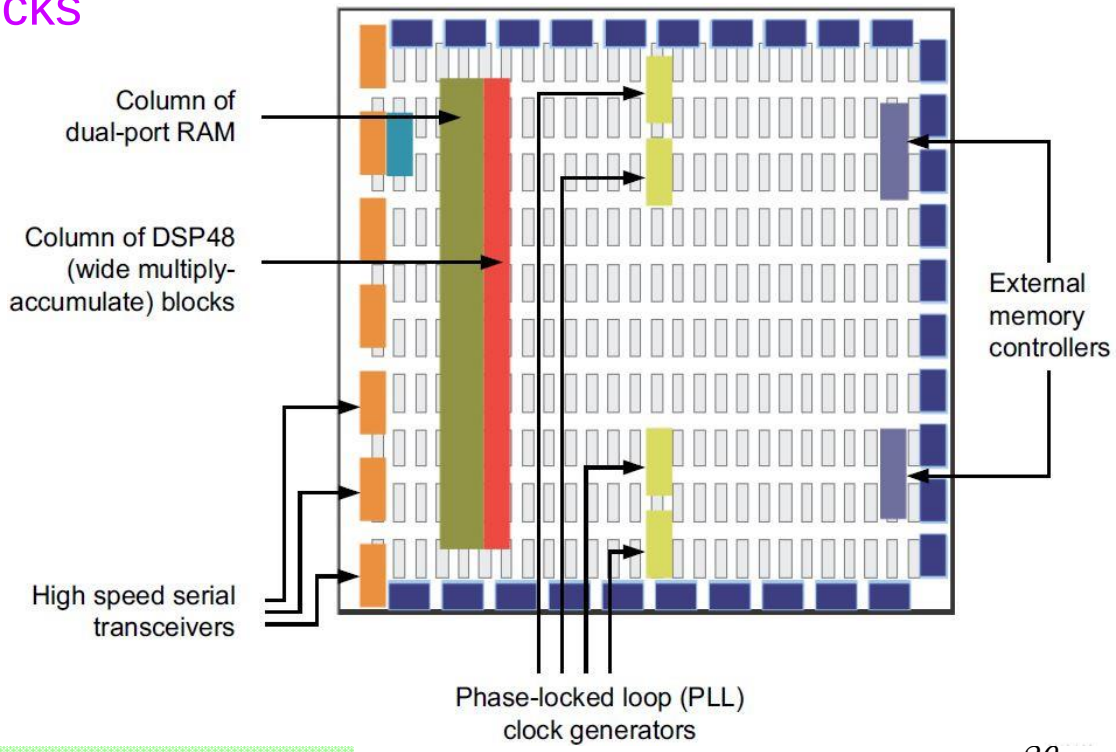
- The basic structure of an FPGA is composed of the following elements: [1.2][1.3]
  - Configurable Logic Blocks
    - Look-up table (LUT): implement any logic function of N (e.g., N = 4) Boolean variables
    - Flip-Flops (FFs)
  - Programmable interconnections
  - Input/Output (I/O) blocks



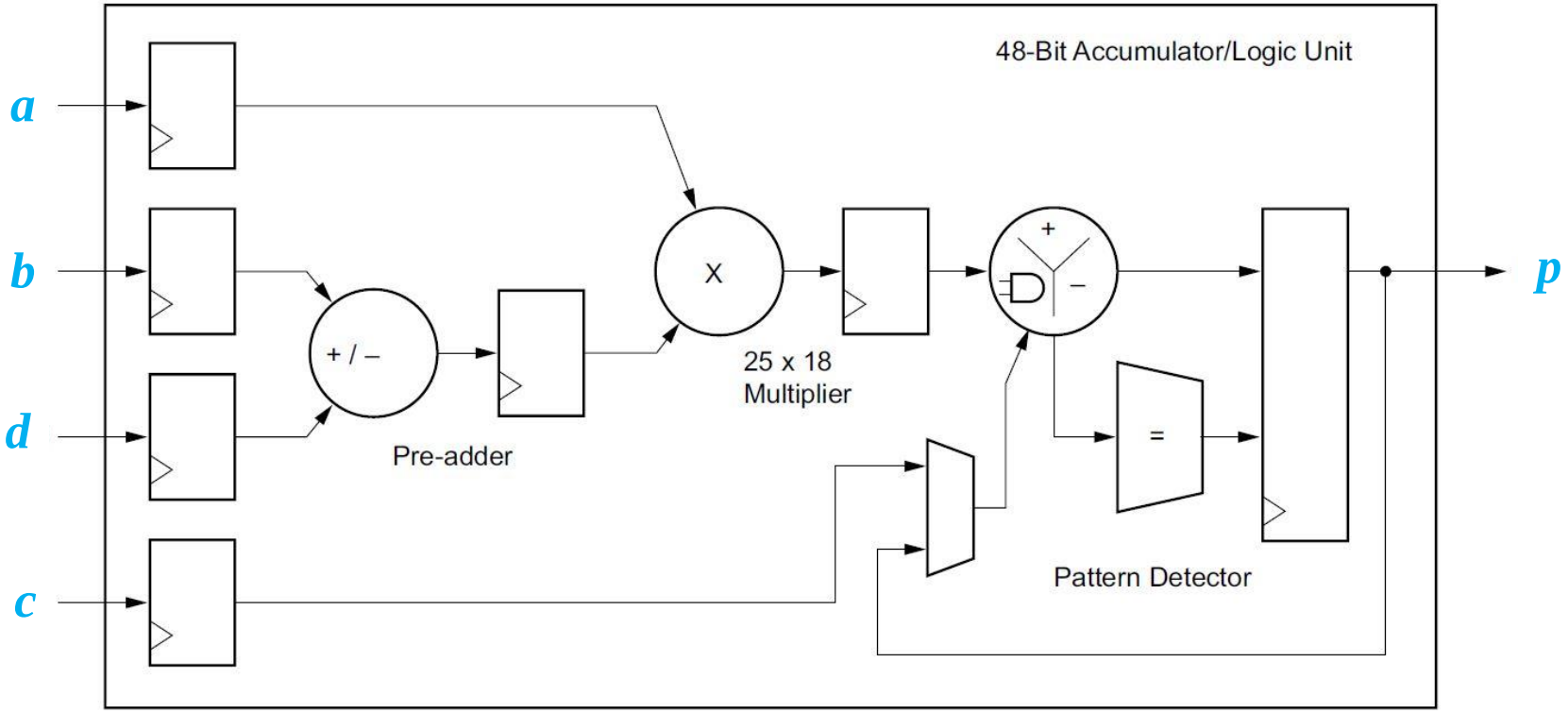
# Xilinx FPGA Architecture (2/5)

- Contemporary FPGA architectures (additional computational and data storage blocks) [1.2][1.3]
  - embedded memories for distributed data storage
  - phase-locked loops (PLLs) for different clock rates
  - high-speed serial transceivers
  - off-chip memory controllers
  - multiply-accumulate blocks



# Xilinx FPGA Architecture (3/5)

- The **DSP block** is an arithmetic logic unit (ALU) [1.2][1.3]




$p = a \times (b + d) + c$ or $p = a \times (b + d)$

# Xilinx FPGA Architecture (4/5)

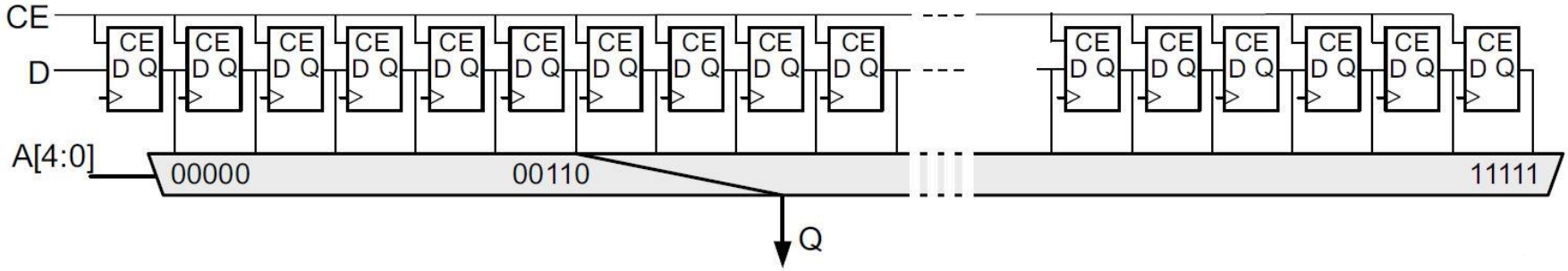
## *Storage Elements* [1.2][1.3]

- FPGA device includes embedded memory elements
  - that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers
  - these elements are block RAMs (BRAMs), UltraRAM blocks (URAMS), LUTs, and shift registers (SRLs)
- BRAM
  - a dual-port RAM module, can implement either a RAM or a ROM
  - two types of BRAM memories: can hold either 18 k or 36 k bits
- UltraRAM blocks
  - are dual-port, synchronous 288 Kb RAM with a fixed configuration of 4,096 bits deep and 72 bits wide
  - are available on UltraScale+ Devices and provide 8 times more storage capacity than the BRAM

# Xilinx FPGA Architecture (5/5)

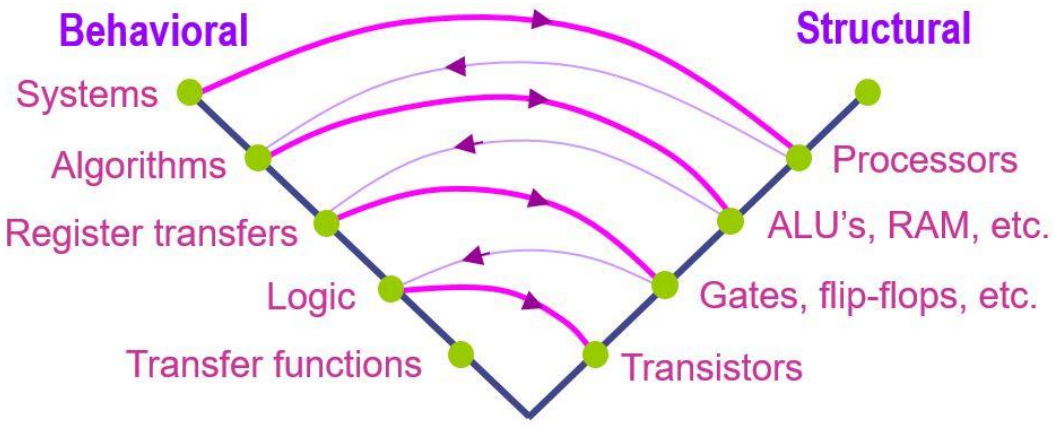
## *Storage Elements* [1.2][1.3]

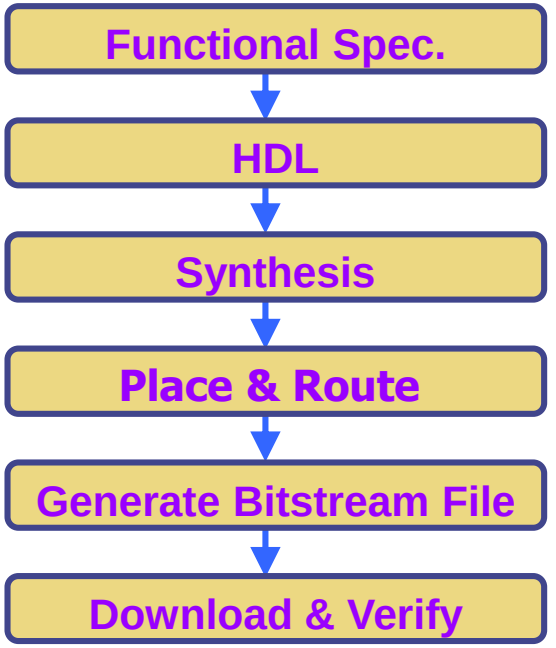
- LUT
  - is a small memory in which the contents of a truth table are written during device configuration
  - these blocks can be used as 64-bit memories and are commonly referred to as distributed memories
  - the fastest kind of memory available on the FPGA device
- shift register
  - a chain of registers connected to each other
  - this structure is to provide data reuse along a computational path, such as with a filter



# C-Based FPGA Design (1/8)

- Xilinx Vivado High-Level Synthesis (HLS) [1.2][1.3]
  - Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that you can synthesize into a Xilinx FPGA
  - You can write C specifications in C, C++, SystemC, or as an Open Computing Language (OpenCL) API C kernel, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power



# C-Based FPGA Design (2/8)

***SystemC*** [1.4]

- SystemC
  - is defined and promoted by the Open SystemC Initiative (OSCI)
  - has been approved by the IEEE Standards Association as IEEE 1666-2011 - the SystemC Language Reference Manual (LRM)
  - is an ANSI standard C++ class library (systemc.h) for system and hardware design
  - is often associated with electronic system-level (ESL) design, and with transaction-level modeling (TLM)
- SystemC is usually applied to
  - system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis
  - provide the hardware and software development team with an executable specification of the system

# C-Based FPGA Design (3/8)

## Features of SystemC

- Modules: component
- Ports: I/O ports
- Signals: wires
- Processes: functions, SC_THREAD & SC_METHOD
- Communication protocols: Channel & Interface
- Rich set of data types
  - introduce several data types which support the modeling of hardware (e.g., logic types, fixed point types, …)
- Clocks
- Event-based (cycle-accurate) simulation: ultra light-weight and fast
- Multiple abstraction levels

# C-Based FPGA Design (4/8)

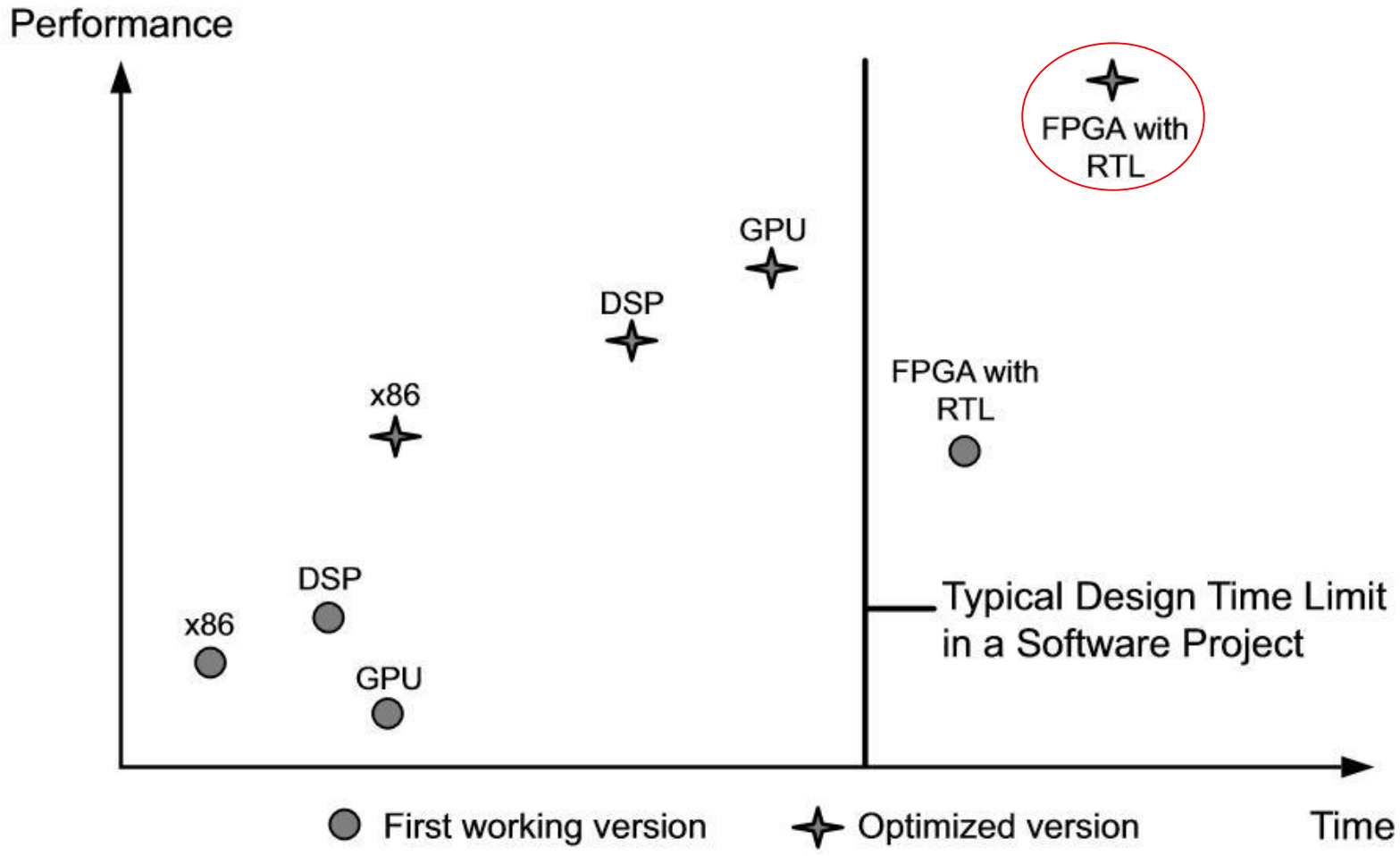
- Example code of an adder [1.4]

```
#include "systemc.h"

SC_MODULE(adder)                          // module (class) declaration
{
    sc_in<int> a, b;                       // ports
    sc_out<int> sum;

     void do_add()                         // process
    {
        sum.write(a.read() + b.read());    //or just sum = a + b
     }

    SC_CTOR(adder)                         // constructor
    {
        SC_METHOD(do_add);                 // register do_add to kernel
        sensitive << a << b;               // sensitivity list of do_add
    }
};
```

# C-Based FPGA Design (5/8)

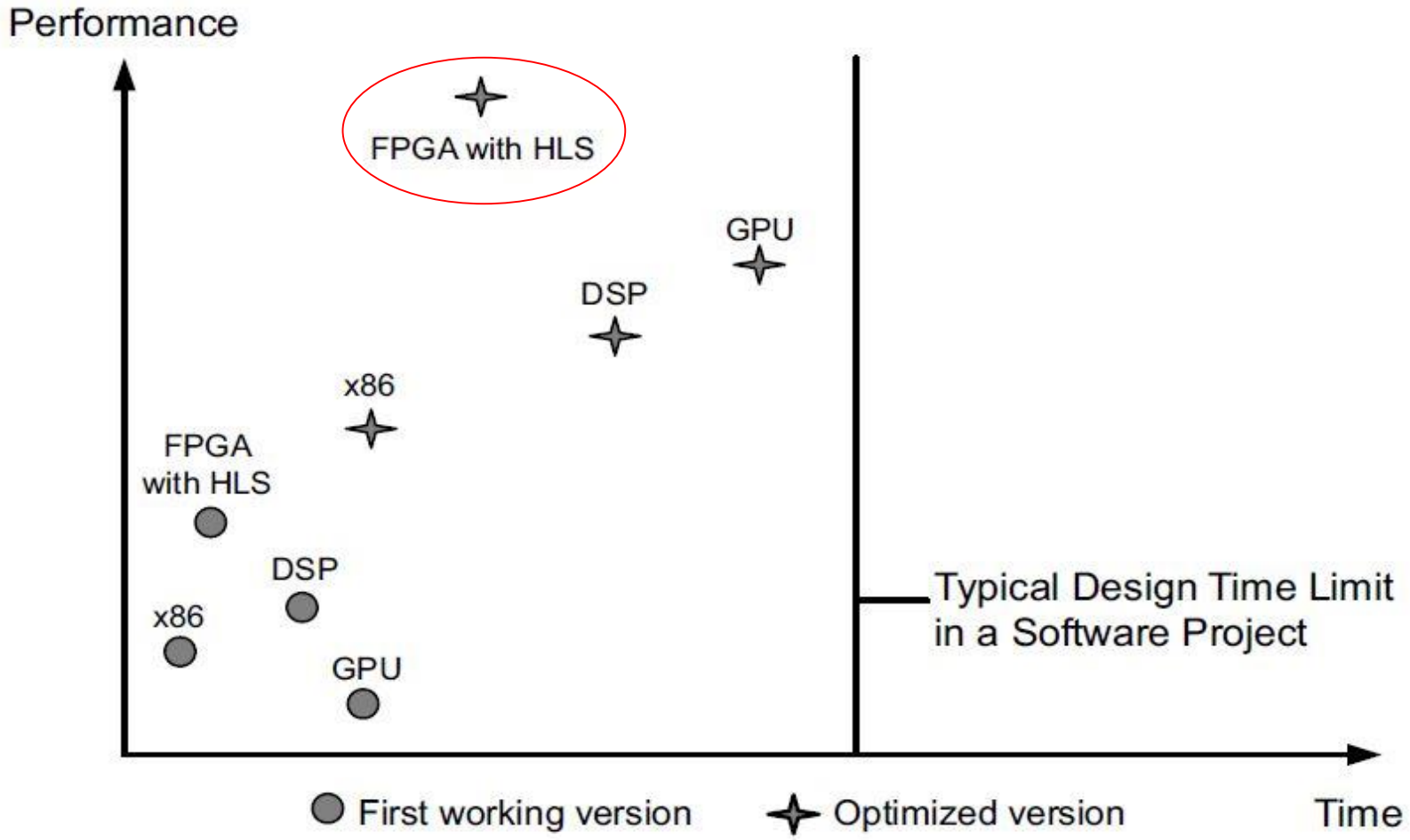
- High-level Synthesis Design Methodology [1.2][1.3]
  - Develop algorithms and verify at the C-level (reduce development and verification time)
  - Control the C synthesis process through optimization directives
  - Create multiple implementations from the C source code using optimization directives
  - Create readable and portable C source code
- High-Level Synthesis Benefits
  - Improved productivity for hardware designers
  - Improved system performance for software designers (new compilation target, the FPGA)

# C-Based FPGA Design (6/8)




**Design Time vs. Application Performance with RTL Design Entry**

# C-Based FPGA Design (7/8)




**Design Time vs. Application Performance with Vivado HLS Compiler**

# C-Based FPGA Design (8/8)

***Vivado High-Level Synthesis*** [1.2][1.3]

- Application code targeting the Vivado HLS compiler uses the same categories as any processor compiler
- Vivado HLS analyzes all programs in terms of:
  - Operations
  - Conditional statements
  - Loops
  - Functions
  - Arrays

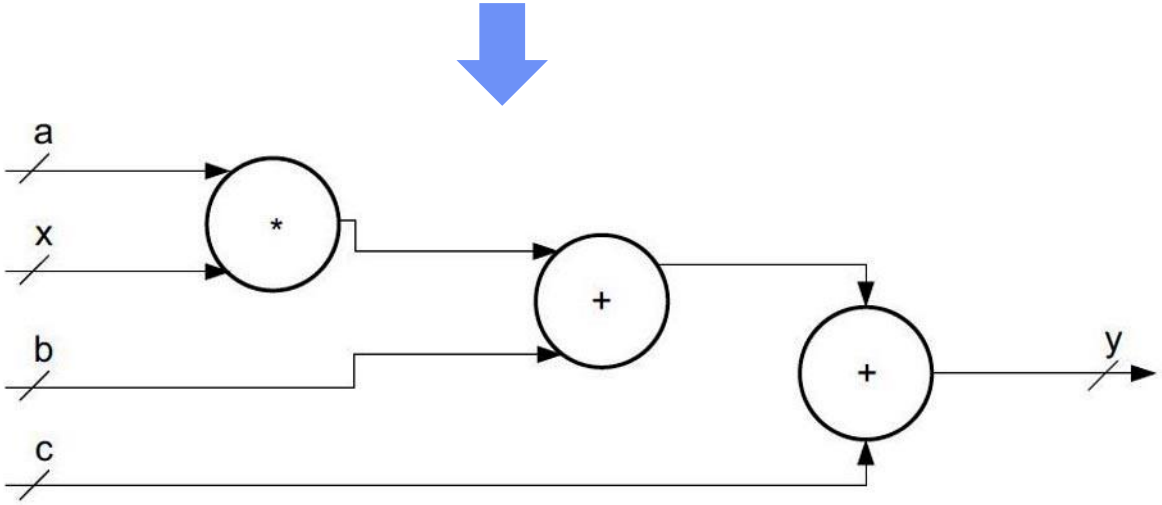
# High-Level Synthesis Basics (1/8)

- HLS includes the following phases [1.2][1.3]
  - Scheduling, Binding, Control logic extraction
- Scheduling
  - determines which operations occur during each clock cycle
    - length of the clock cycle or clock frequency
    - time it takes for the operation to complete, as defined by the target device
    - user-specified optimization directives
- Binding
  - determines which hardware resource implements each scheduled operation
- Control logic extraction
  - extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design

# High-Level Synthesis Basics (2/8)

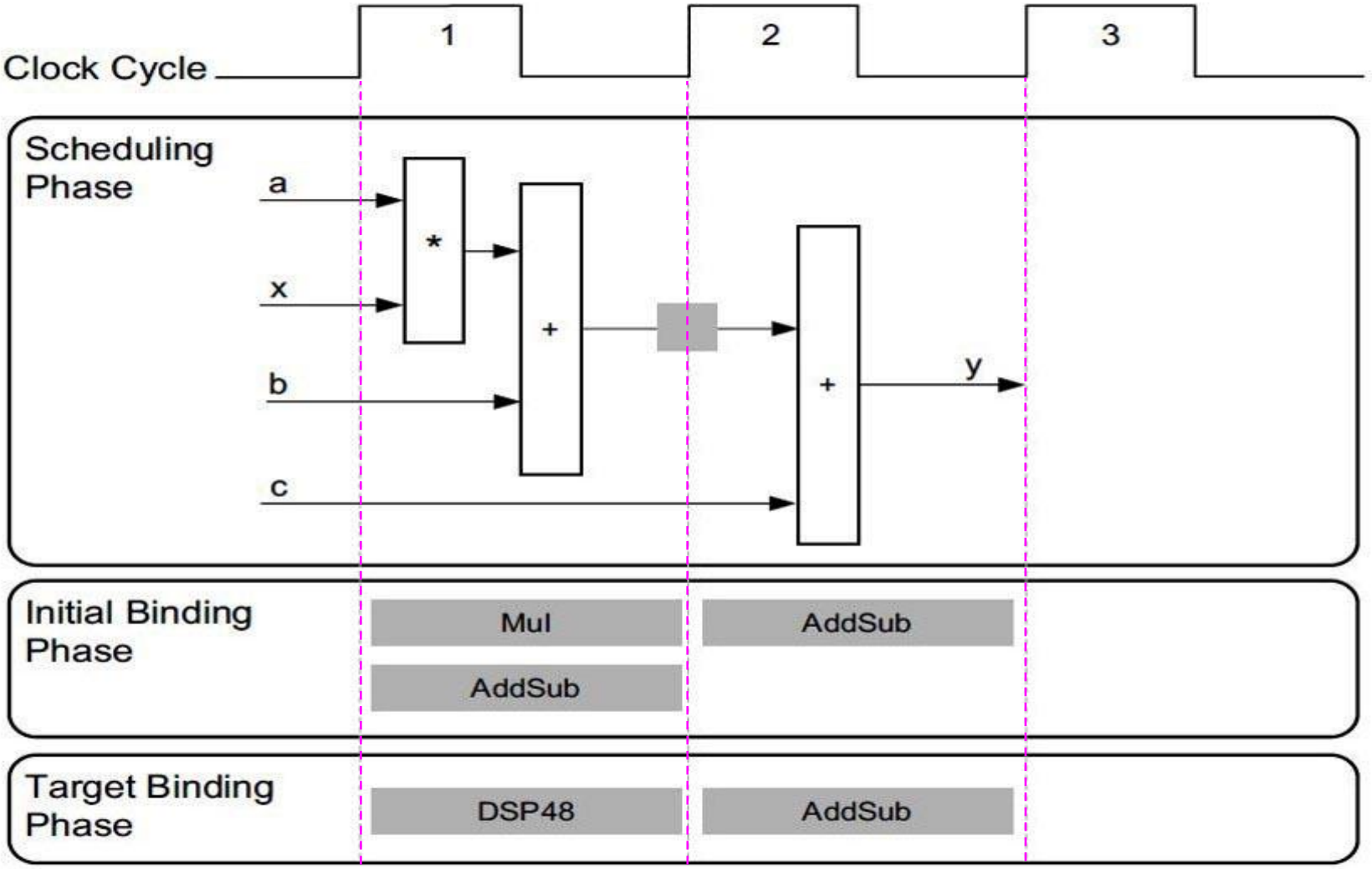
- Example: Scheduling and Binding [1.2][1.3]

```
int foo(char x, char a, char b, char c) {
    char y;
    y = a*x+b+c;
    return y;
}
```



# High-Level Synthesis Basics (3/8)

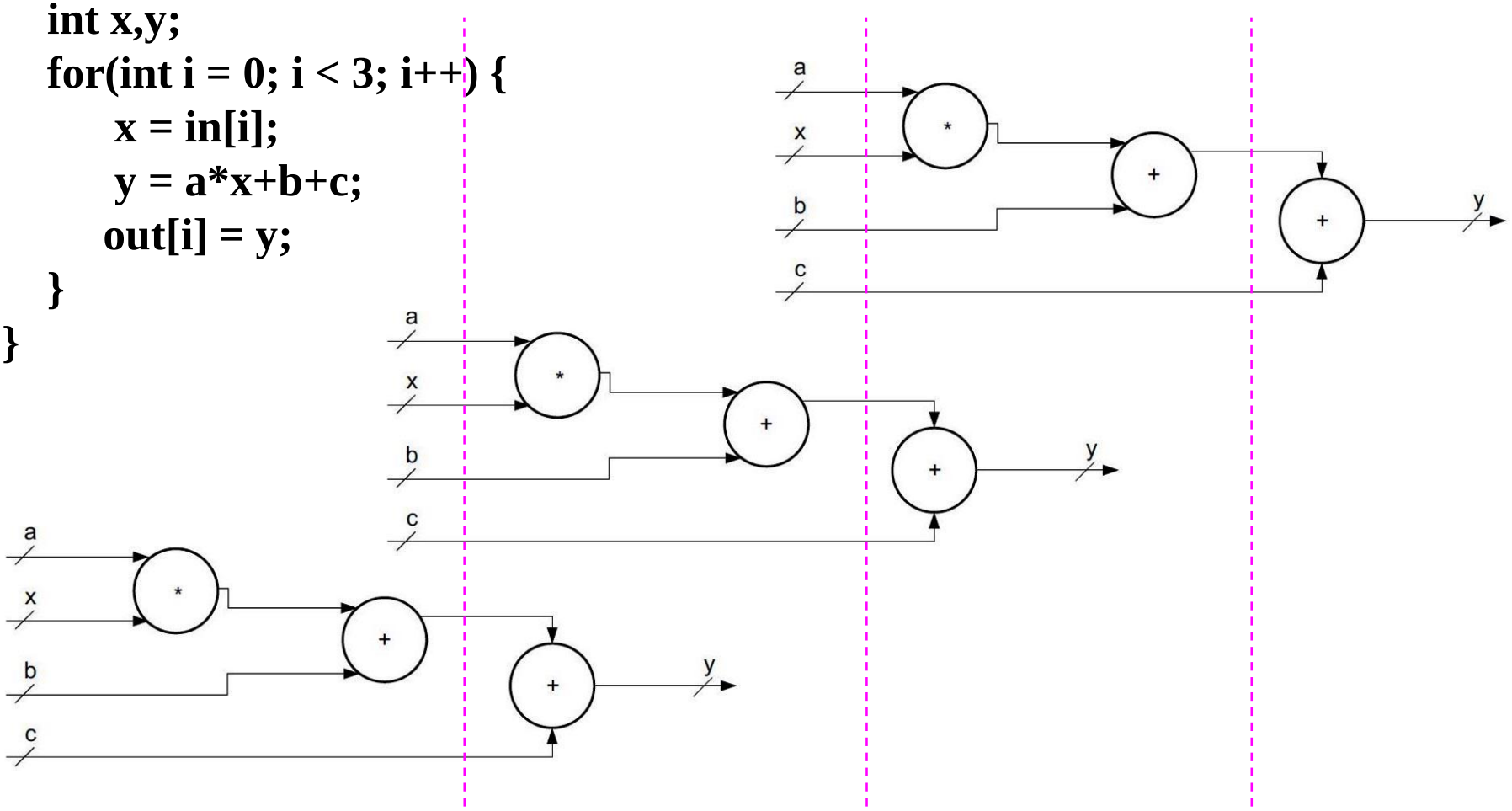
- Example: Scheduling and Binding [1.2][1.3]



# High-Level Synthesis Basics (4/8)

- Example: Control Logic Extraction [1.2][1.3]

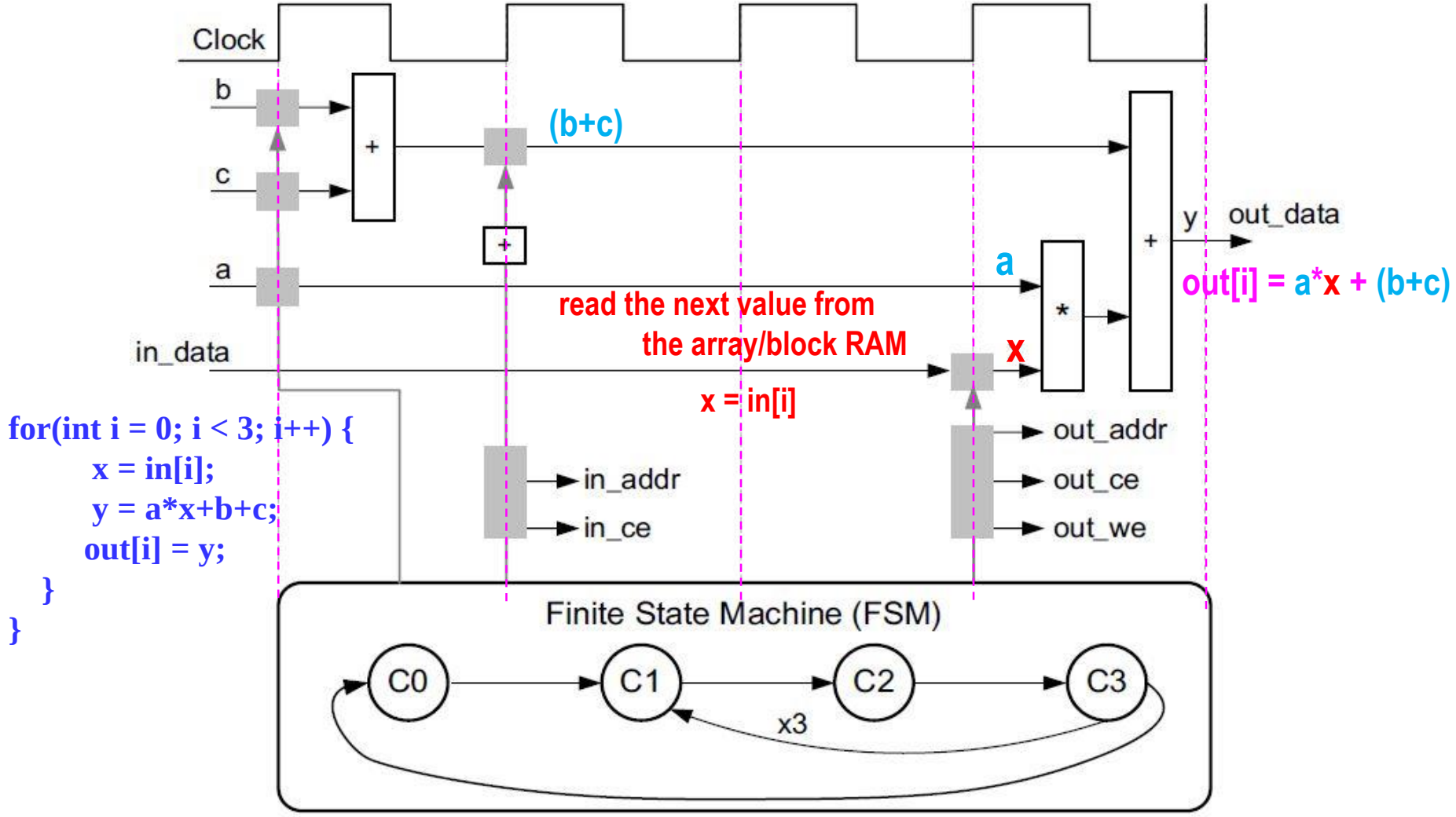
```
void foo(int in[3], char a, char b, char c, int out[3]) {
    int x,y;
    for(int i = 0; i < 3; i++) {
        x = in[i];
        y = a*x+b+c;
       out[i] = y;
    }
}
```



# High-Level Synthesis Basics (5/8)

- Example: Control Logic Extraction [1.2][1.3]

x = in[i];  out[i] = a*x+(b+c);



# High-Level Synthesis Basics (6/8)

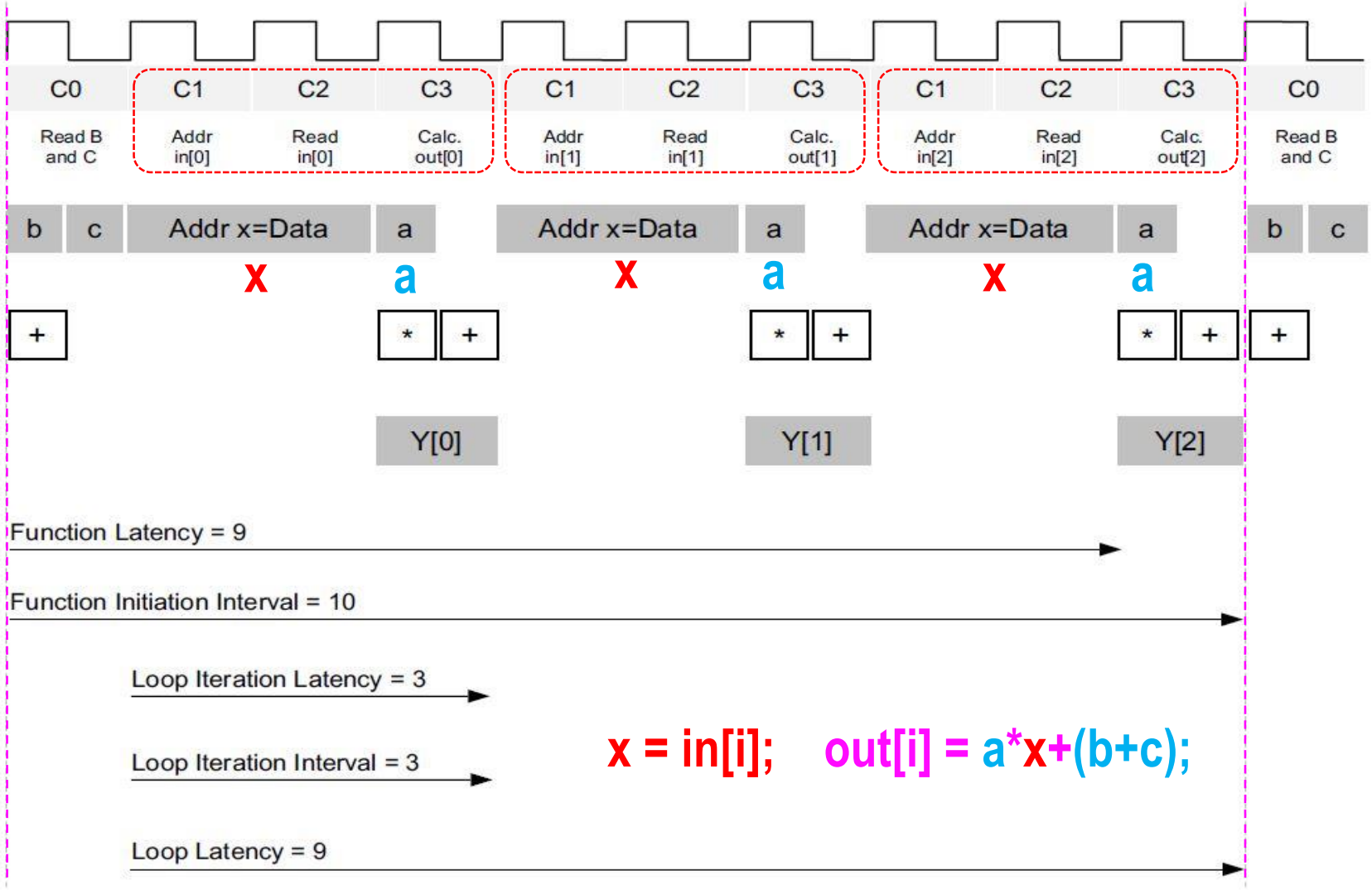
- HLS synthesizes the C code as follows [1.2][1.3]
  - Top-level function arguments synthesize into RTL I/O ports
  - C functions synthesize into blocks in the RTL hierarchy
  - Loops in the C functions are kept rolled by default
  - Arrays in the C code synthesize into block RAM or UltraRAM in the final FPGA design
- HLS creates the optimal implementation based on
  - default behavior, constraints, and any optimization directives you specify

# High-Level Synthesis Basics (7/8)

- The performance metrics in the synthesis report generated by high-level synthesis [1.2][1.3]
  - Area: look-up tables (LUT), registers, block RAMs, and DSP48s
  - Latency: number of clock cycles required for the function to compute all output values
  - Initiation interval (II): number of clock cycles before the function can accept new input data
  - Loop iteration latency: number of clock cycles it takes to complete one iteration of the loop
  - Loop initiation interval: number of clock cycle before the next iteration of the loop starts to process data
  - Loop latency: number of cycles to execute all iterations of the loop

# High-Level Synthesis Basics (8/8)

- Example: Latency and Initiation Interval [1.2][1.3]



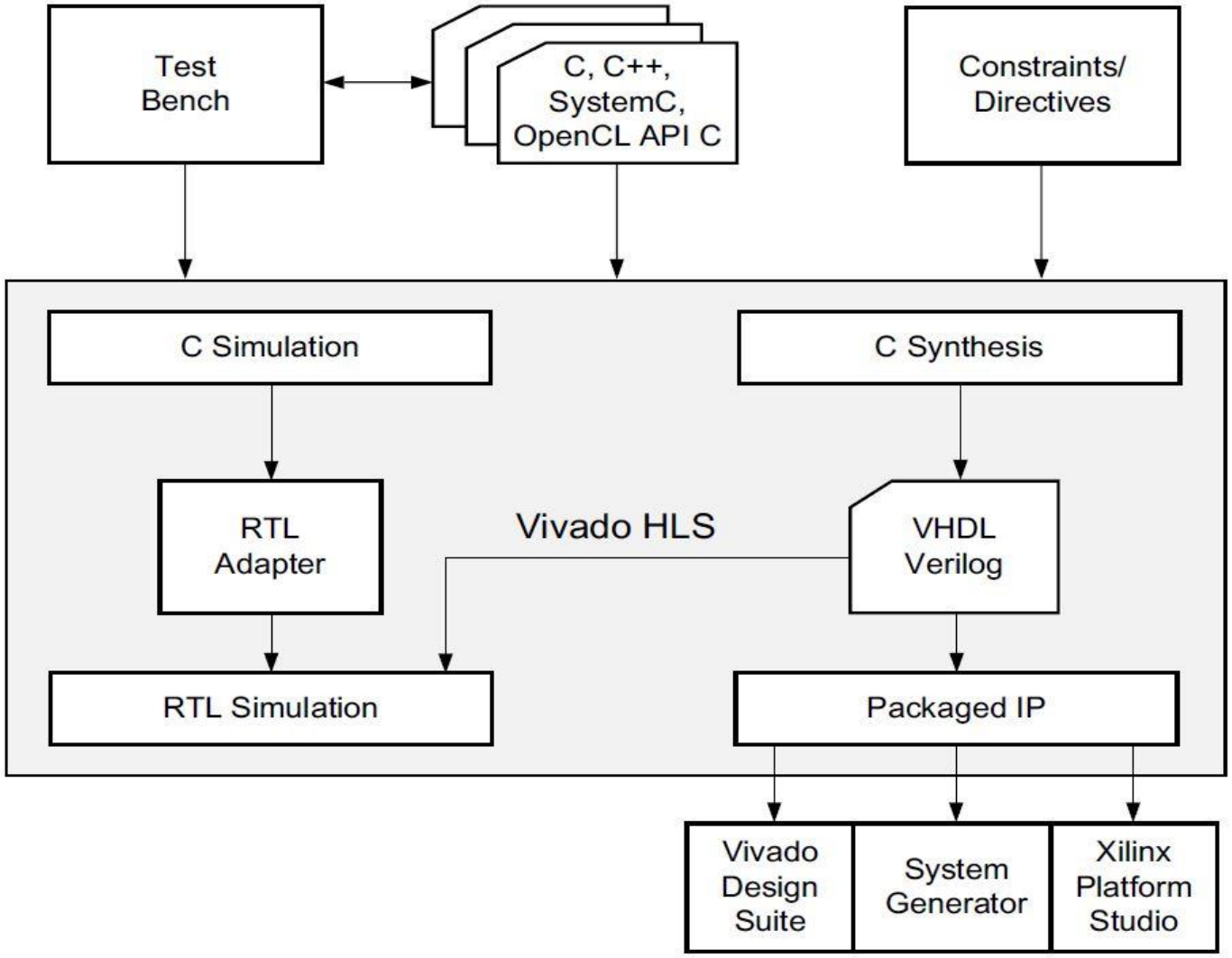
# Vivado High-Level Synthesis (1/12)

***Vivado HLS Design Flow*** [1.2][1.3]

- Compile, execute (simulate), and debug the C algorithm
- Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives
- Generate comprehensive reports and analyze the design
- Verify the RTL implementation using a pushbutton flow
- Package the RTL implementation into a selection of IP formats

# Vivado High-Level Synthesis (2/12)

- Vivado HLS Design Flow [1.2][1.3]



# Vivado High-Level Synthesis (3/12)

***Inputs to Vivado HLS*** [1.2][1.3]

- Vivado HLS graphical user interface (GUI) or Tcl commands can be used to give inputs to Vivado HLS
- C function written in C, C++, SystemC, or an OpenCL API C kernel
- Constraints
  - the clock period, clock uncertainty, and FPGA target
  - the clock uncertainty defaults to 12.5% of the clock period
- Directives
  - are optional and direct the synthesis process to implement a specific behavior or optimization
- C test bench and any associated files
  - Vivado HLS uses the C test bench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL Cosimulation

# Vivado High-Level Synthesis (4/12)

## *Outputs from Vivado HLS* [1.2][1.3]

- RTL implementation files in hardware description language (HDL) formats
  - VHDL (IEEE 1076-2000)
  - Verilog (IEEE 1364-2001)
  - using logic synthesis, you can synthesize the packaged IP into an FPGA bitstream
- Report files
  - the result of synthesis, C/RTL co-simulation, and IP packaging

# Vivado High-Level Synthesis (5/12)

***Test Bench, Language Support, and C Libraries*** [1.2][1.3]

- In any C program, the top-level function is called main()
- In the Vivado HLS design flow, you can specify any sub-function below main() as the top-level function for synthesis
- You cannot synthesize the top-level function main()
- Following are additional rules:
  - Only one function is allowed as the top-level function for synthesis
  - Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized
  - If you want to synthesize functions that are not in the hierarchy under the top-level function for synthesis, you must merge the functions into a single top-level function for synthesis
  - The verification flow for OpenCL API C kernels requires special handling in the Vivado HLS flow

# Vivado High-Level Synthesis (6/12)

***Test Bench*** [1.2][1.3]

- The C test bench includes the function main() and any sub-functions that are not in the hierarchy under the top-level function for synthesis
- These functions verify that the top-level function for synthesis is functionally correct by providing stimuli to the function for synthesis and by consuming its output
- Vivado HLS uses the test bench to compile and execute the C simulation
- During the compilation process, you can select the Launch Debugger option to open a full C-debug environment, which enables you to analyze the C simulation

# Vivado High-Level Synthesis (7/12)

## *Language Support* [1.2][1.3]

- Vivado HLS supports the following standards for C compilation/simulation:
  - ANSI-C (GCC 4.6)
  - C++ (G++ 4.6)
  - OpenCL API (1.0 embedded profile)
  - SystemC (IEEE 1666-2006, version 2.2)
- Synthesis is not supported for some constructs, including:
  - Dynamic memory allocation
  - Operating system (OS) operations

# Vivado High-Level Synthesis (8/12)

## *C Libraries* [1.2][1.3]

- Vivado HLS provides the following C libraries to extend the standard C languages:
  - Arbitrary precision data types
  - Half-precision (16-bit) floating-point data types
  - Math operations
  - Video functions
  - Xilinx IP functions, including fast fourier transform (FFT) and finite impulse response (FIR)
  - FPGA resource functions to help maximize the use of shift register LUT (SRL) resources

# Vivado High-Level Synthesis (9/12)

## *Synthesis, Optimization, and Analysis* [1.2][1.3]

- Synthesis, optimization, and analysis steps in the Vivado HLS design process:
  - 1. Create a project with an initial solution
  - 2. Verify the C simulation executes without error
  - 3. Run synthesis to obtain a set of results
  - 4. Analyze the results

# Vivado High-Level Synthesis (10/12)

## *Optimization* [1.2][1.3]

- You can apply different optimization directives to the design, including:
  - Instruct a task to execute in a pipeline, allowing the next execution of the task to begin before the current execution is complete
  - Specify a latency for the completion of functions, loops, and regions
  - Specify a limit on the number of resources used
  - Override the inherent or implied dependencies in the code and permit specified operations
    - for example, if it is acceptable to discard or ignore the initial data values, such as in a video stream, allow a memory read before write if it results in better performance
  - Select the I/O protocol to ensure the final design can be connected to other hardware blocks with the same I/O protocol

# Vivado High-Level Synthesis (11/12)

## *Analysis & RTL Verification* [1.2][1.3]

- Analysis
  - Vivado HLS automatically creates synthesis reports to help you understand the performance of the implementation
  - Analysis Perspective includes the Performance tab, which allows you to interactively analyze the results in detail
- RTL Verification
  - If you added a C test bench to the project, you can use it to verify that the RTL is functionally identical to the original C
  - The C test bench verifies the output from the top-level function for synthesis and returns zero to the top-level function main() if the RTL is functionally identical
  - Vivado HLS uses this return value for both C simulation and C/RTL co-simulation to determine if the results are correct

# Vivado High-Level Synthesis (12/12)

## *RTL Simulators & Export* [1.2][1.3]

- Vivado HLS automatically creates the infrastructure to perform the C/RTL co-simulation and automatically executes the simulation using one of the following supported RTL simulators:
  - Vivado Simulator (XSim)
  - ModelSim simulator
  - VCS
  - NCSim
  - Riviera
- RTL Export
  - Using Vivado HLS, you can export the RTL and package the final RTL output files as IP in any of Xilinx IP formats

# Lab 1: RTL Design of RGB to YUV (1/2)

- Color Space Conversion

$$
\begin{aligned}
Y &= 0.299 \times R + 0.587 \times G + 0.114 \times B \\
U &= -0.169 \times R - 0.331 \times G + 0.5 \times B + 128 \\
V &= 0.5 \times R - 0.419 \times G - 0.081 \times B + 128
\end{aligned}
$$

# Lab 1: RTL Design of RGB to YUV (2/2)

- Color space conversion circuit [1.5]
  - Xilinx Zedboard, Vivado HLS, Xilinx SDK

# References


- [1.1] https://www.arrow.com/en/research-and-events/articles/fpga-vs-cpu-vs-gpu-vs-microcontroller
- [1.2] https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf
- [1.3] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf
- [1.4] https://en.wikipedia.org/wiki/SystemC
- [1.5] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug871-vivado-high-level-synthesis-tutorial.pdf