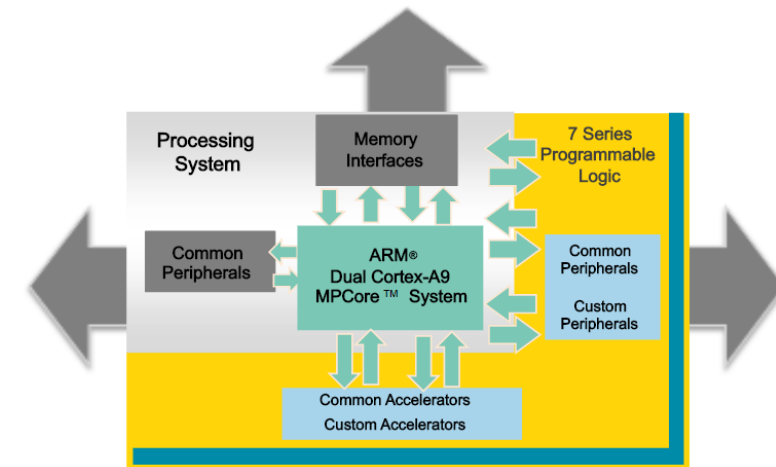
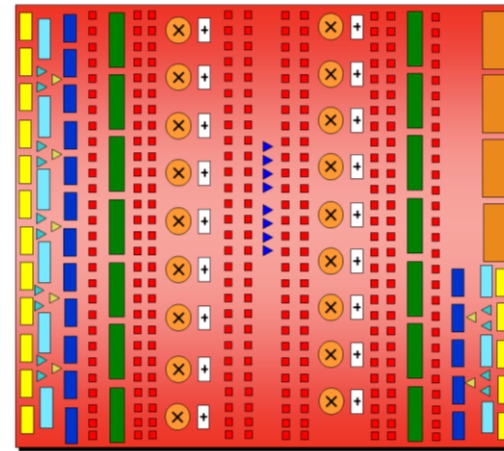


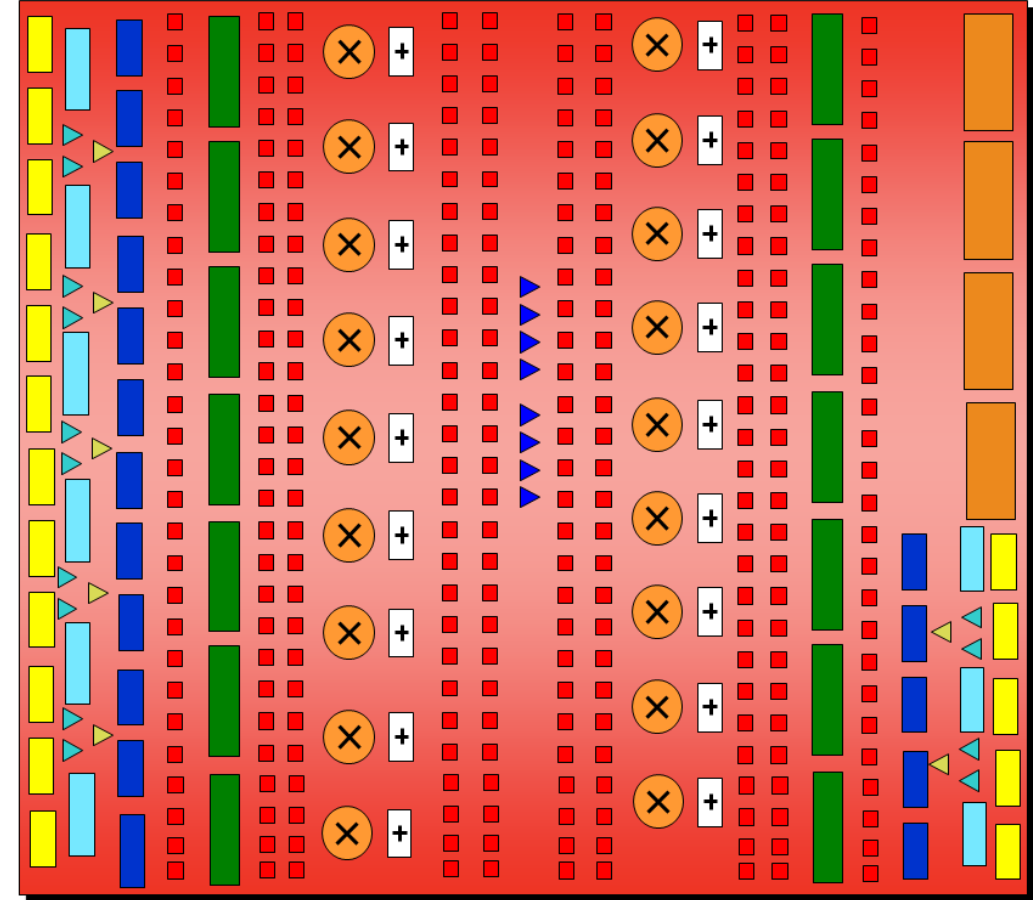
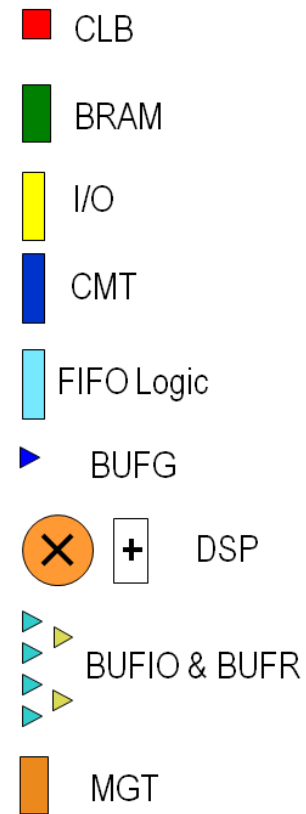


ECE 270: Embedded Logic Design



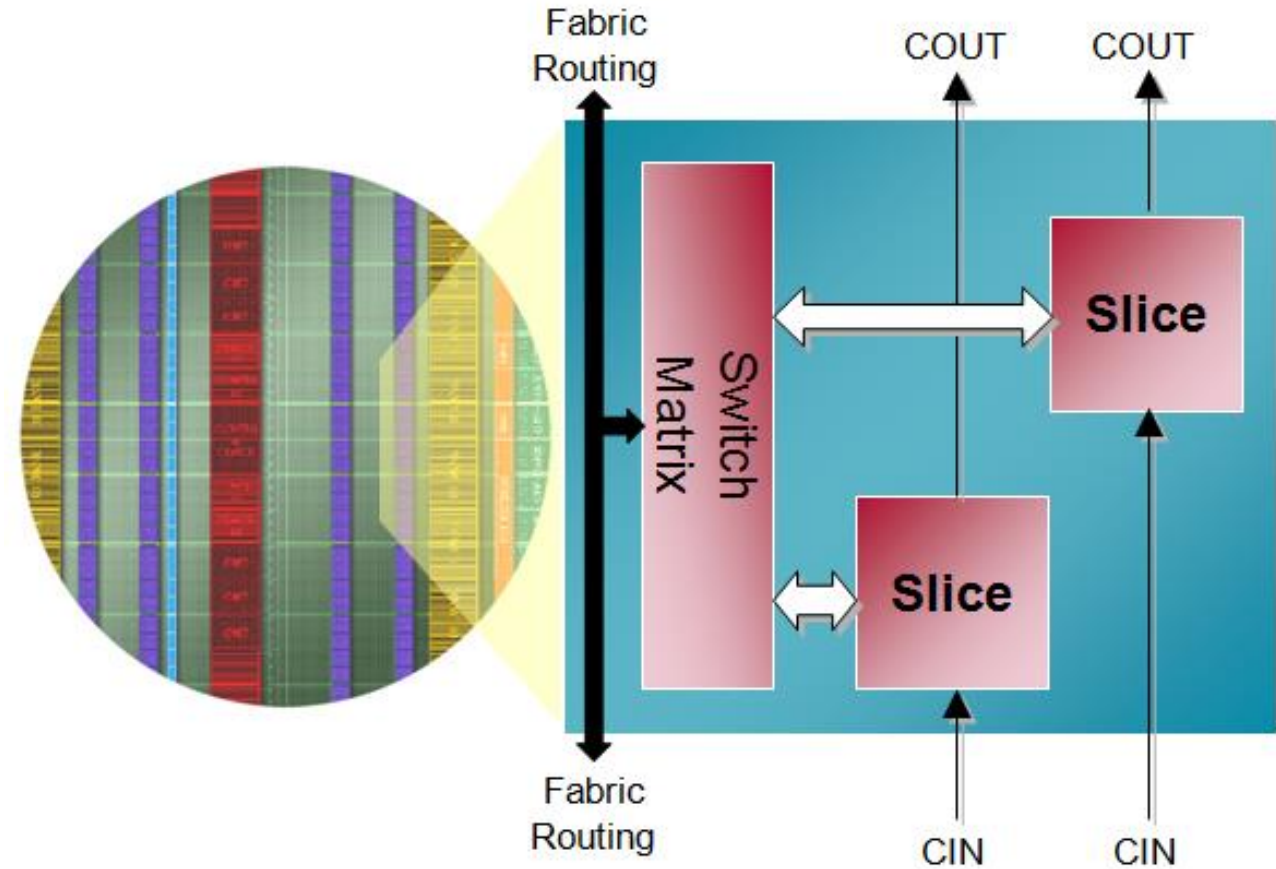
FPGA Architecture

- All **7-series** families share the same basic building blocks.
- The **mixture and number of these resources varies** across families

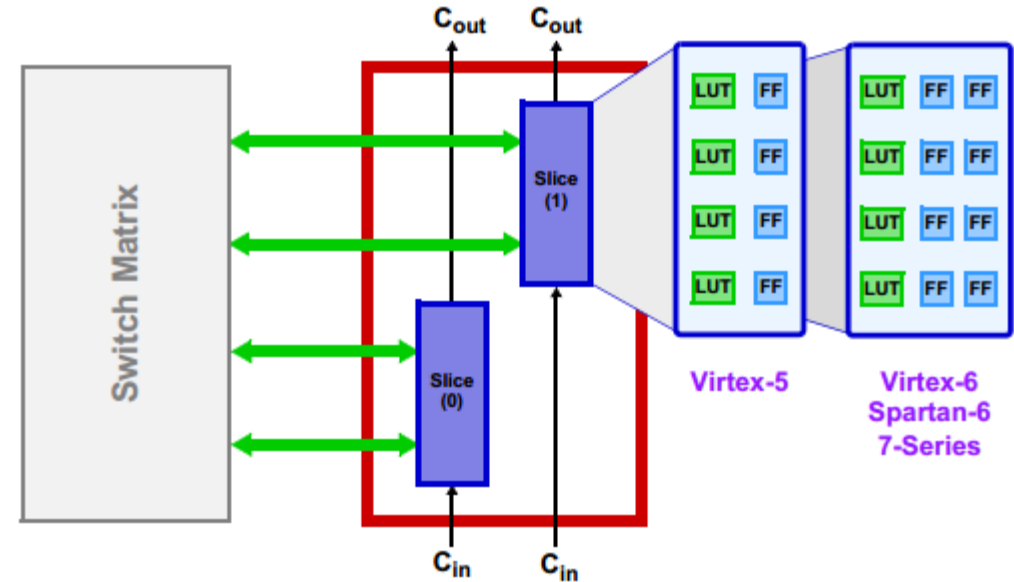
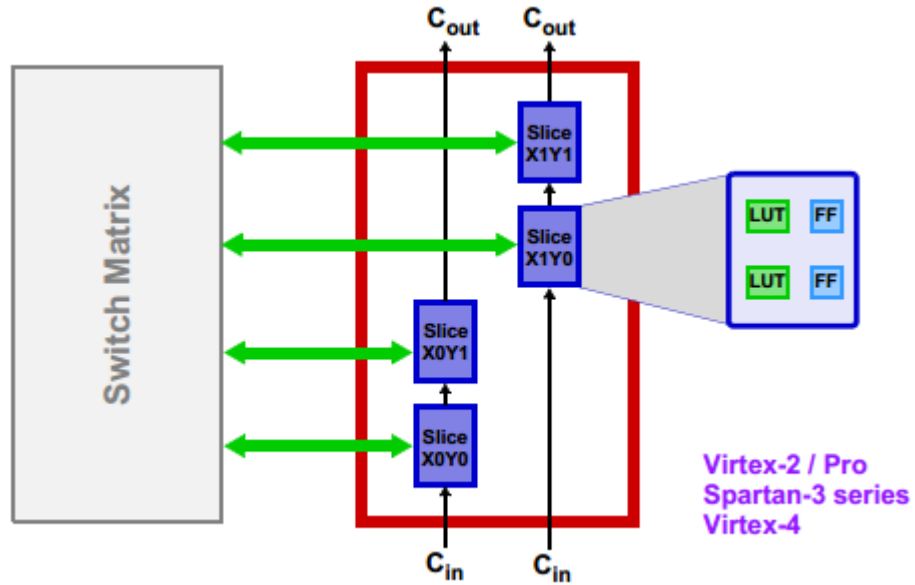


Configurable Logic Block (CLB)

- Primary resource for design in Xilinx FPGAs
- **CLB** contains more than one **slice**
- Connected to **switch matrix** for routing to other FPGA resources
- **Carry chain** runs vertically in a column from one slice to the one above

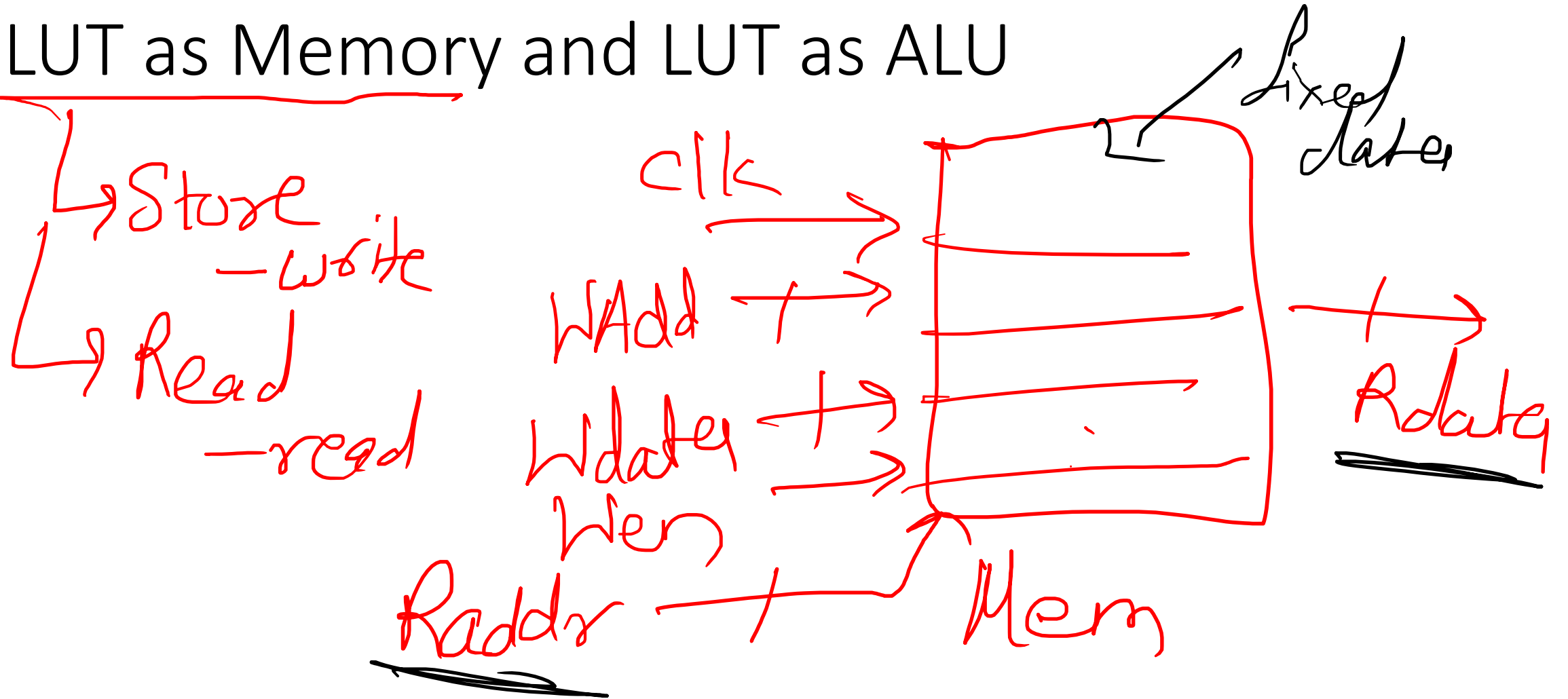


Configurable Logic Block (CLB)



Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains
2	8	16	2

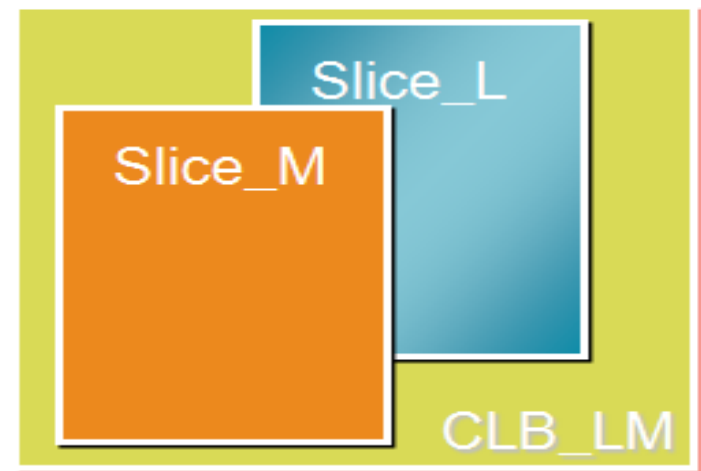
LUT as Memory and LUT as ALU



Types of CLB Slices

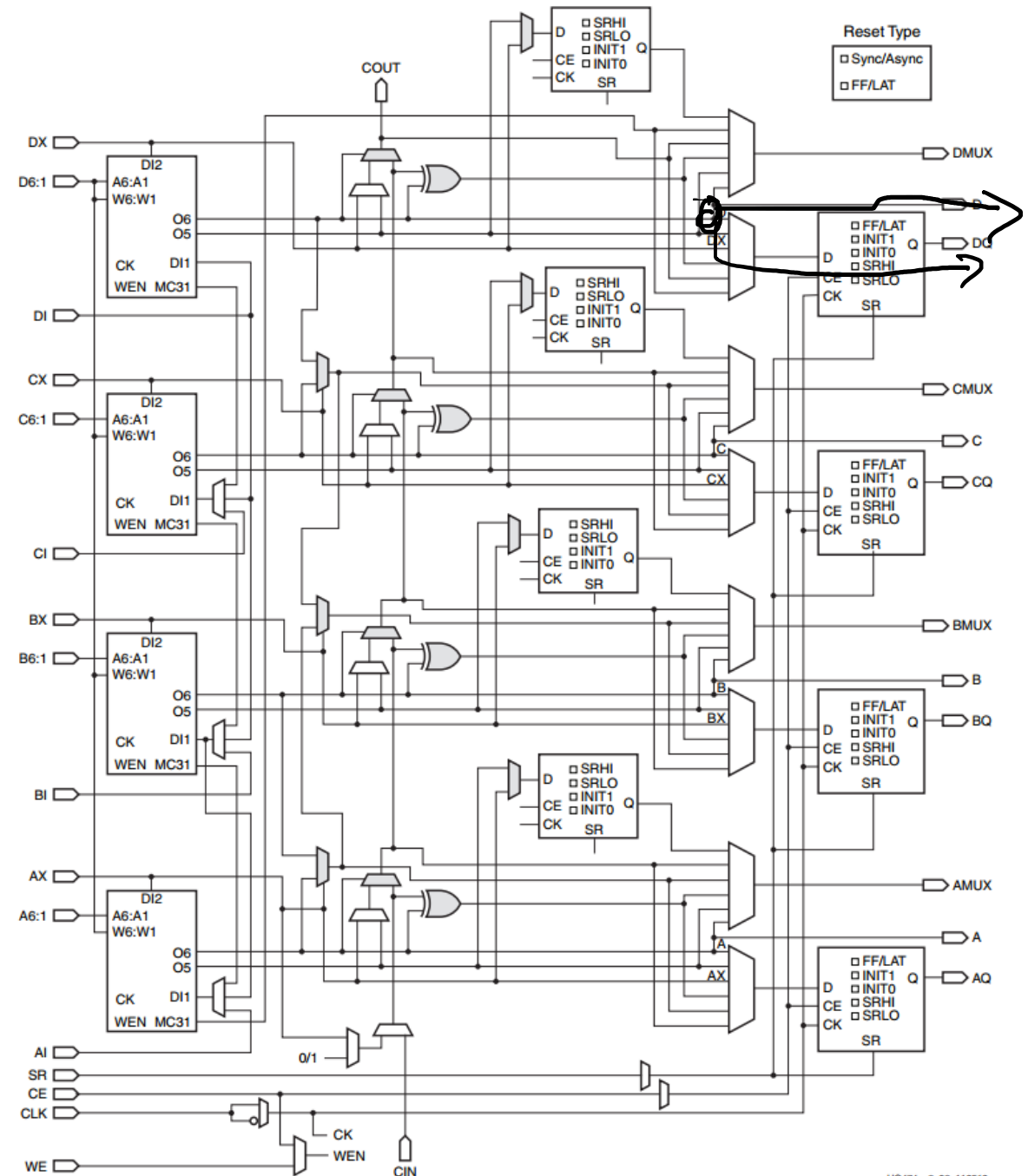
Shift Register

- **SLICEM: Full slice**
 - LUT can be used for logic and memory/SRL
- **SLICEL: Logic and arithmetic only**
 - LUT can only be used for logic (not memory/SRL)
- Each CLB can contain **two SLICEL** or a **SLICEL and a SLICEM**.
- In the 7-series FPGAs, **approximately ¼ of slices** are SLICEM, the remainder are SLICEL.



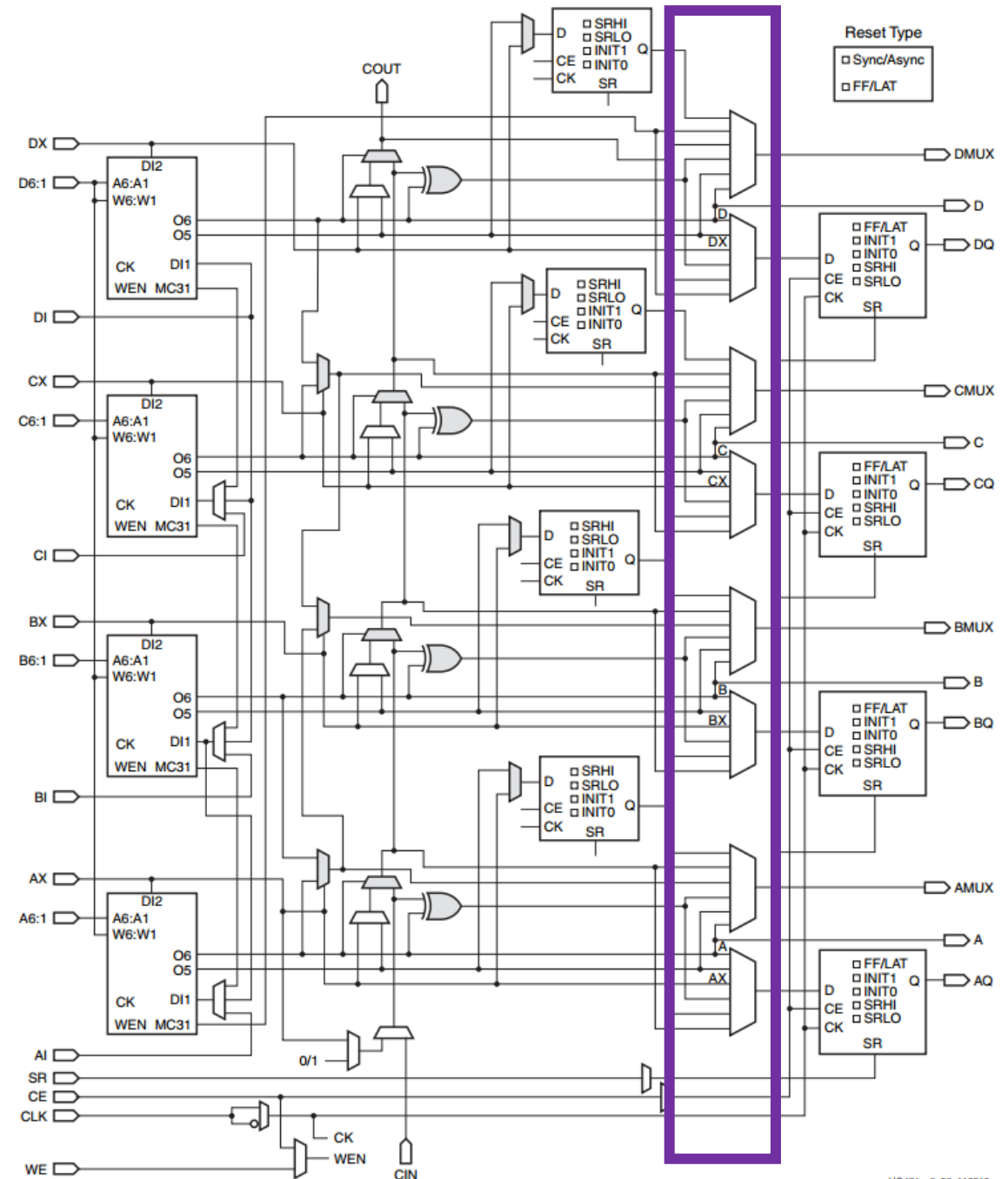
Slice Resource

- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.



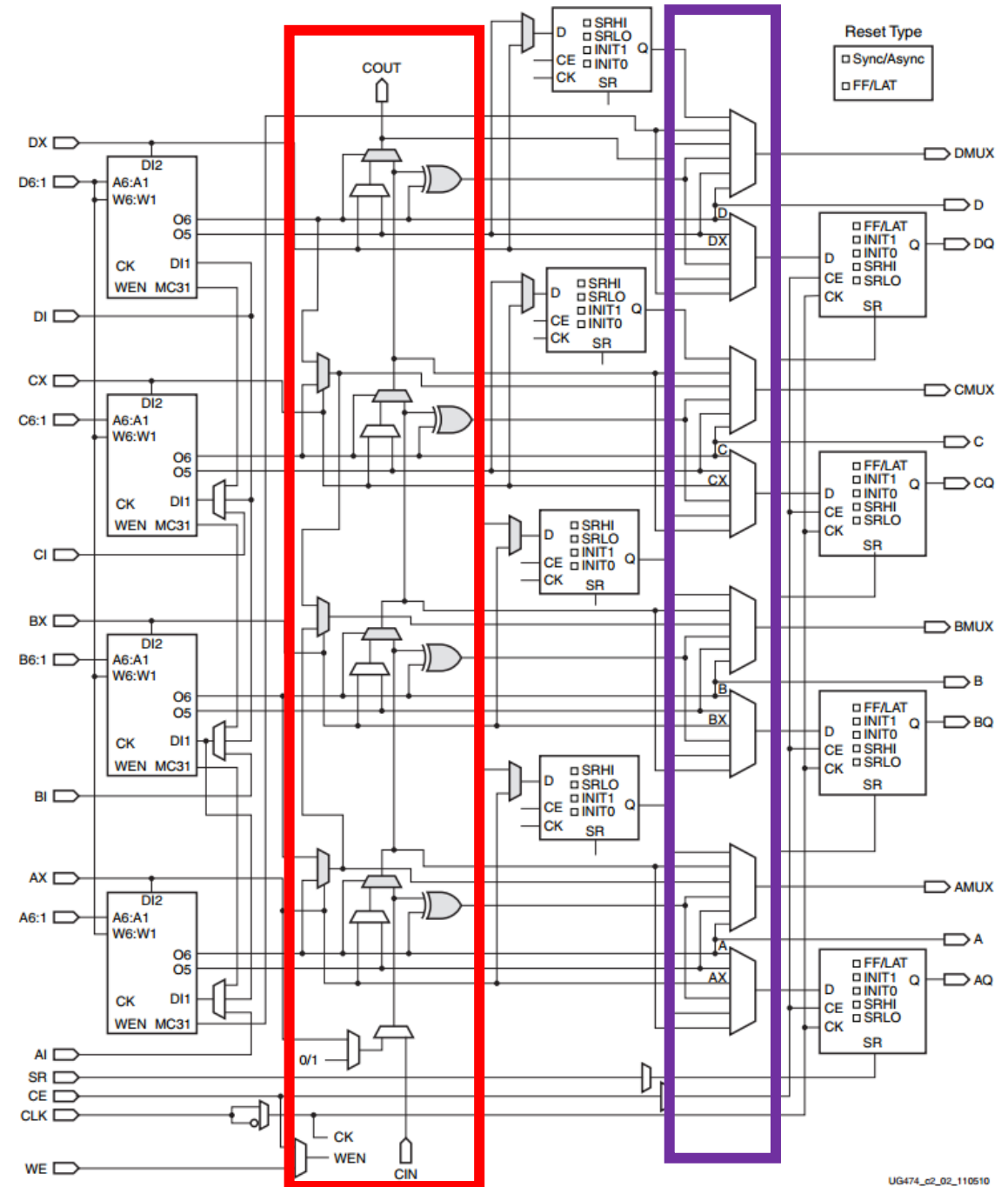
Slice Resource

- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.



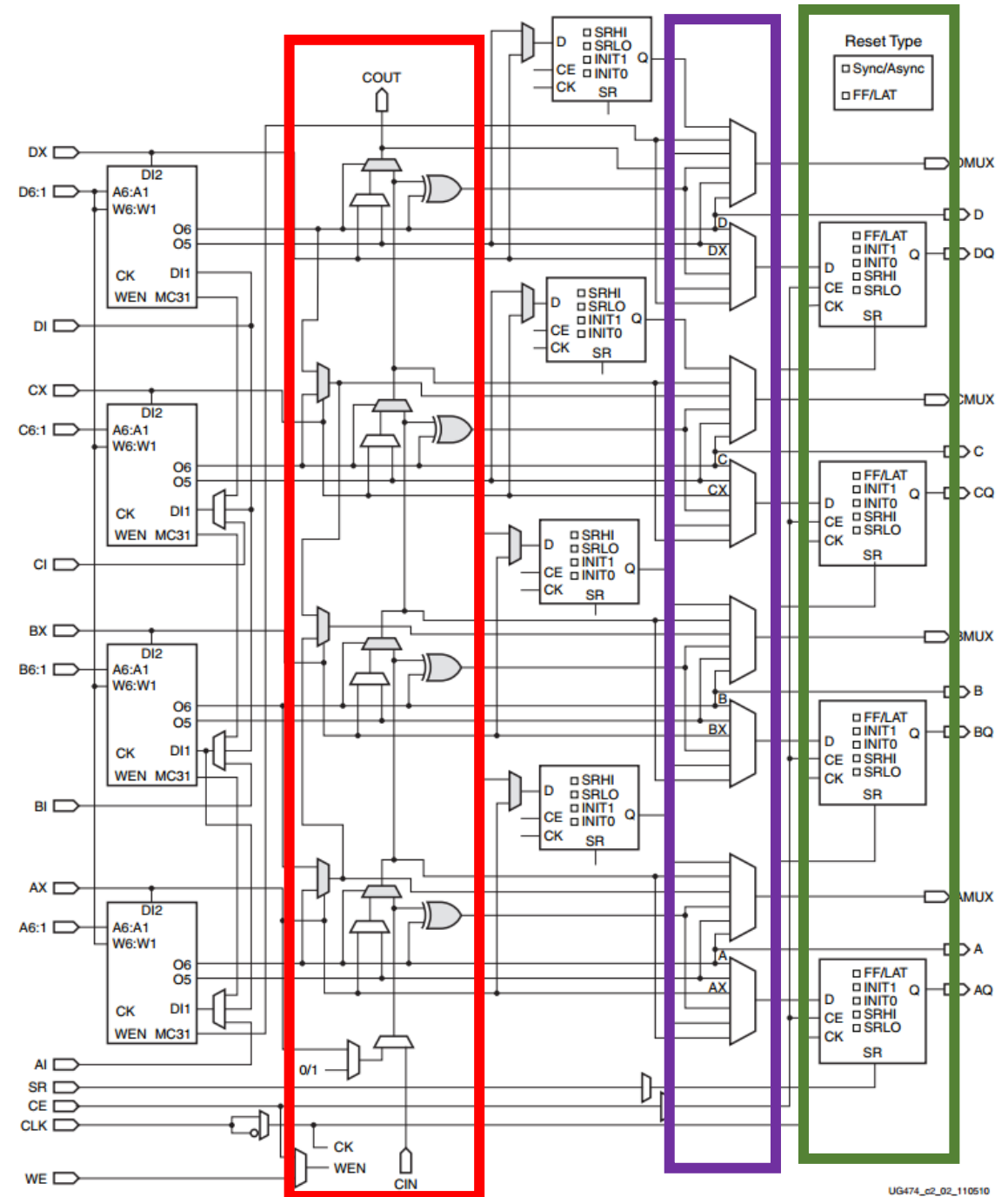
Slice Resource

- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.



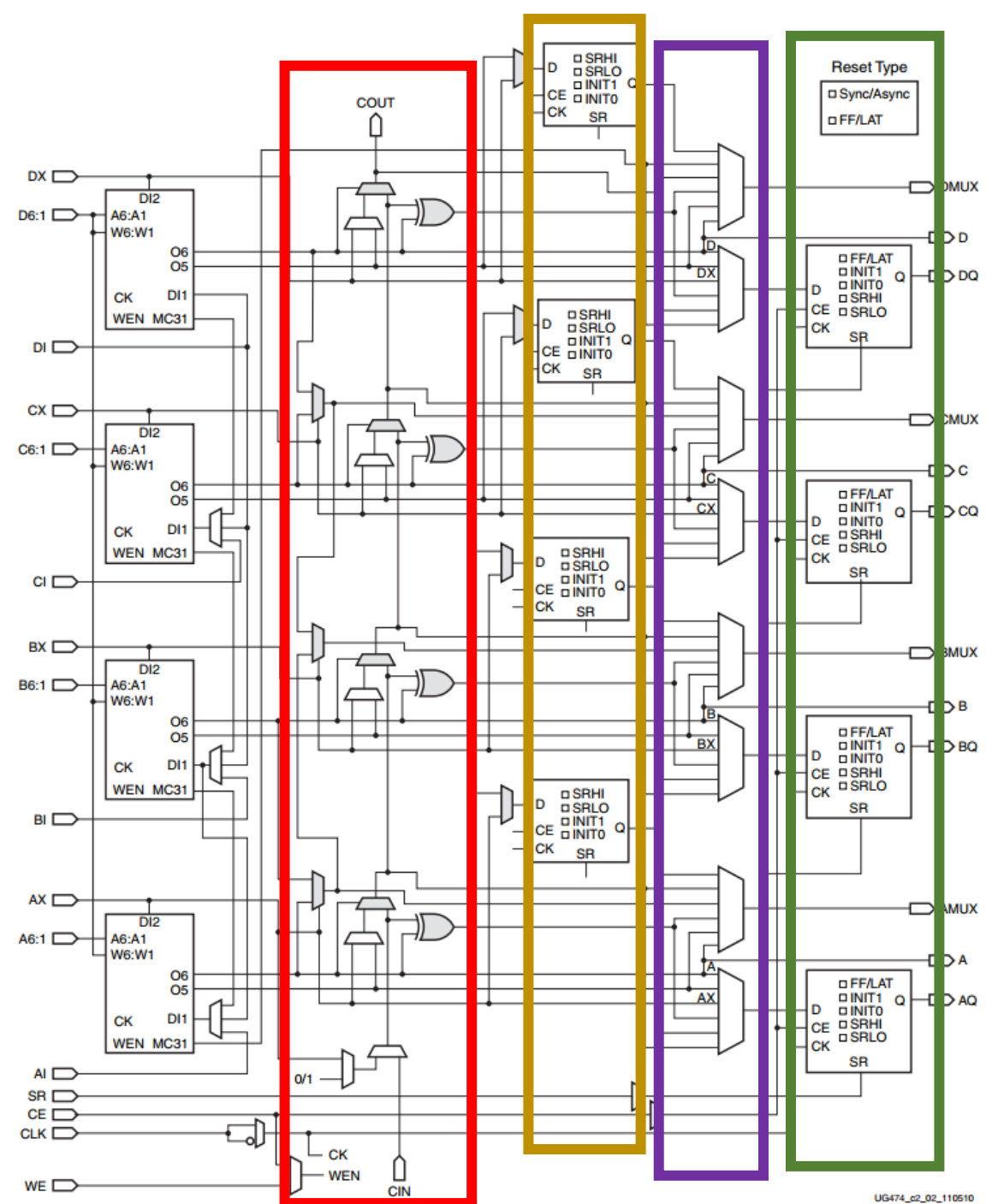
Slice Resource

- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.



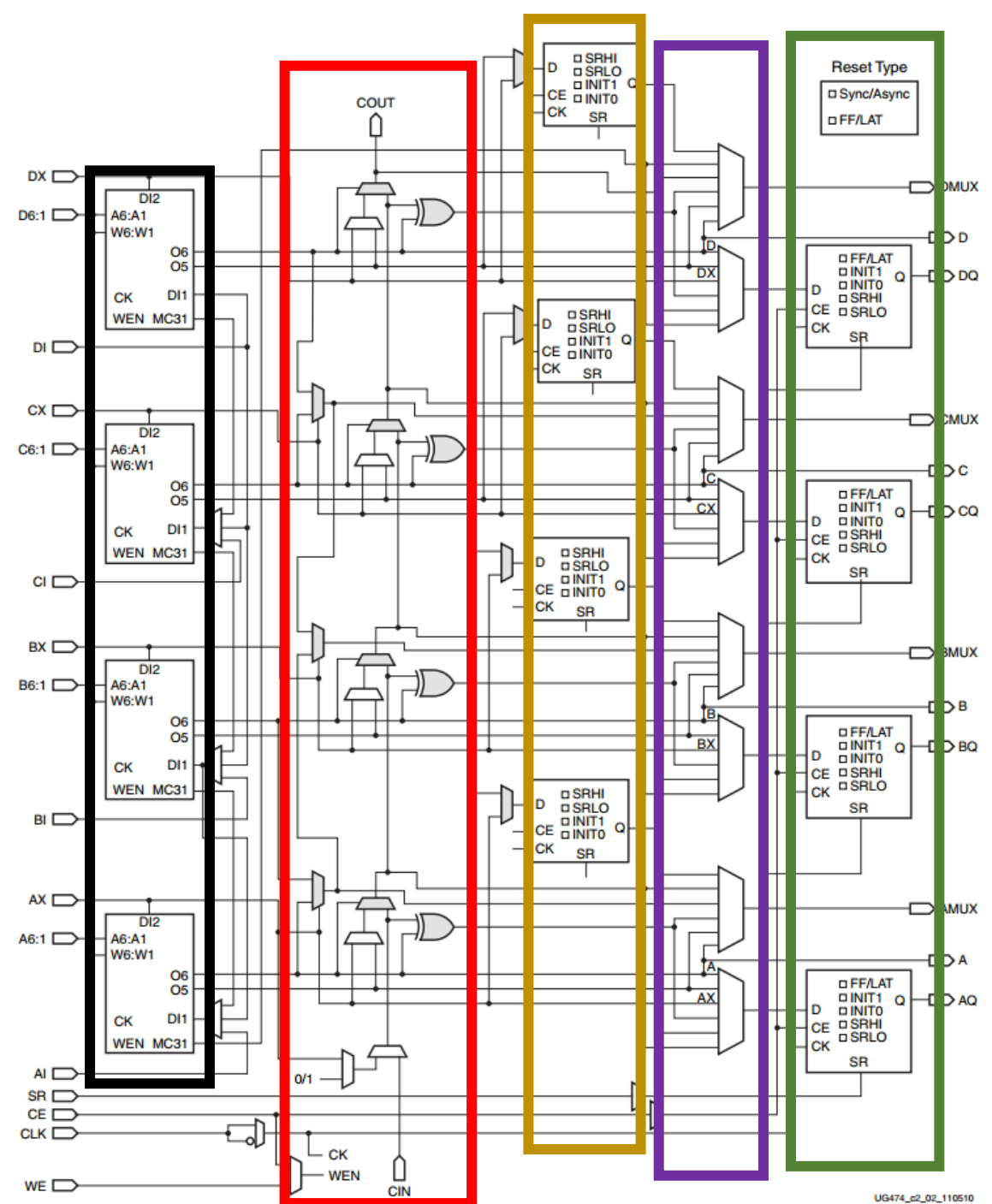
Slice Resource

- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.

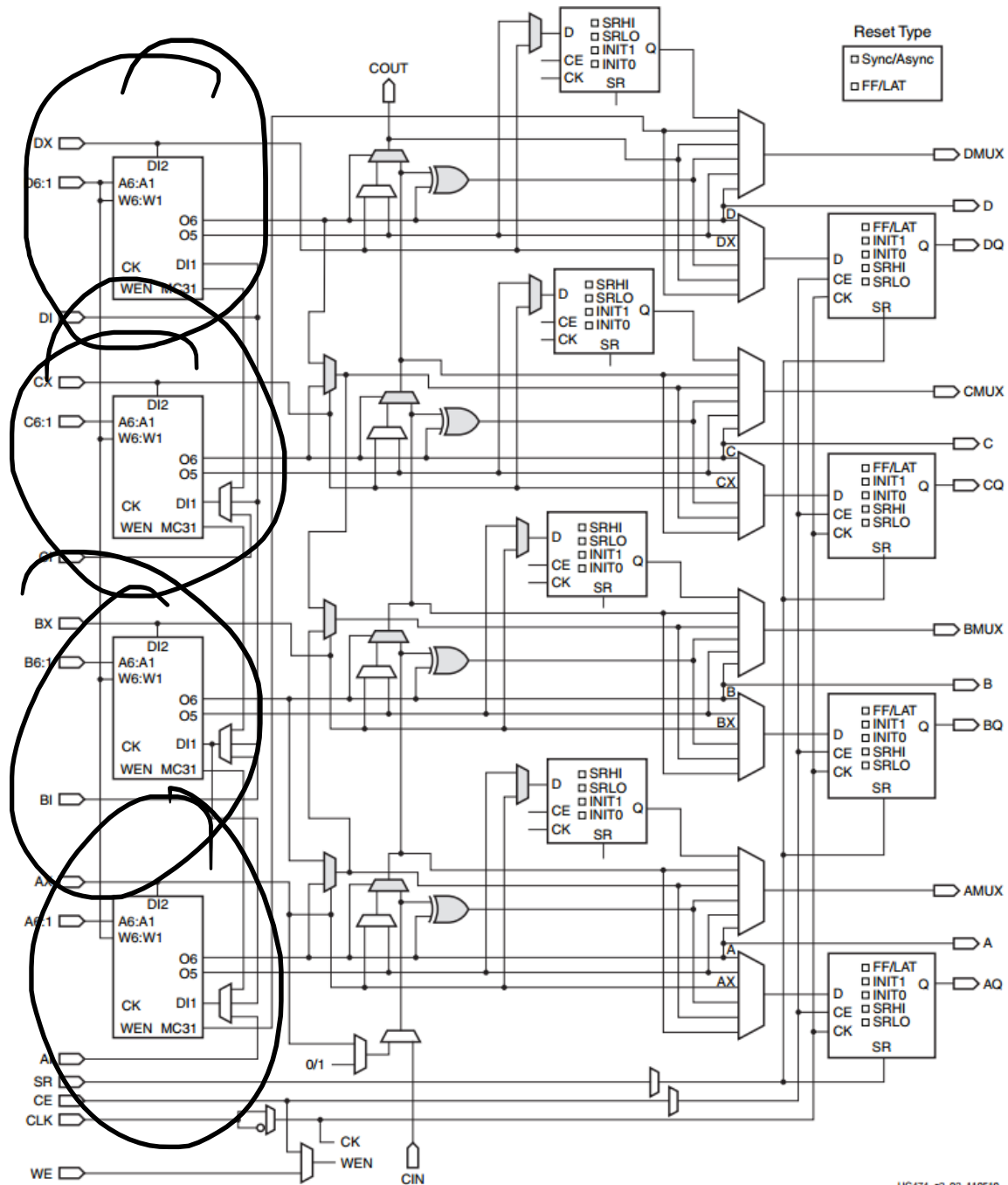


Slice Resource

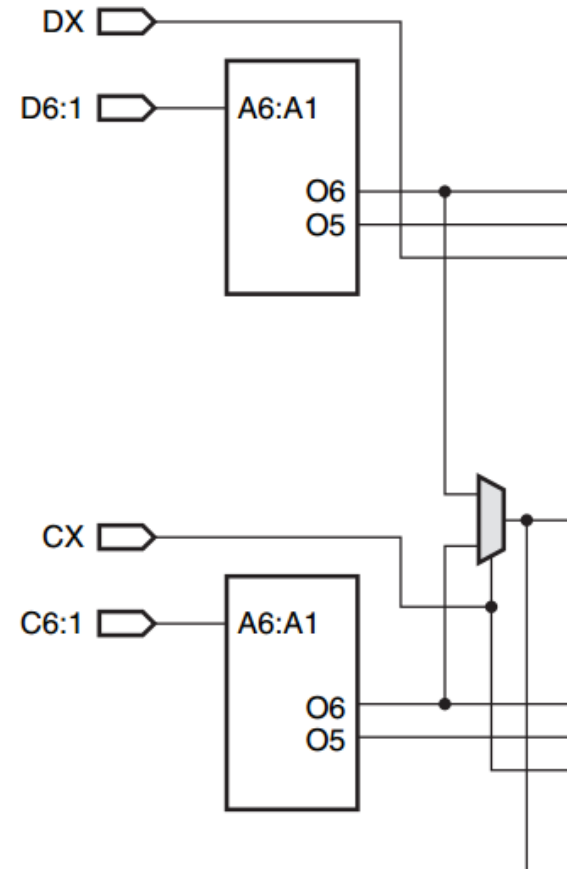
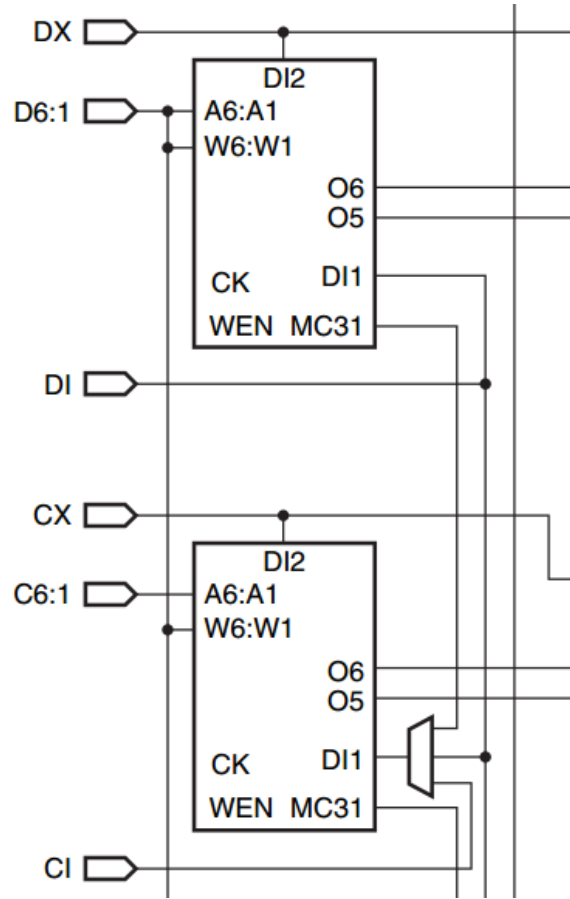
- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.



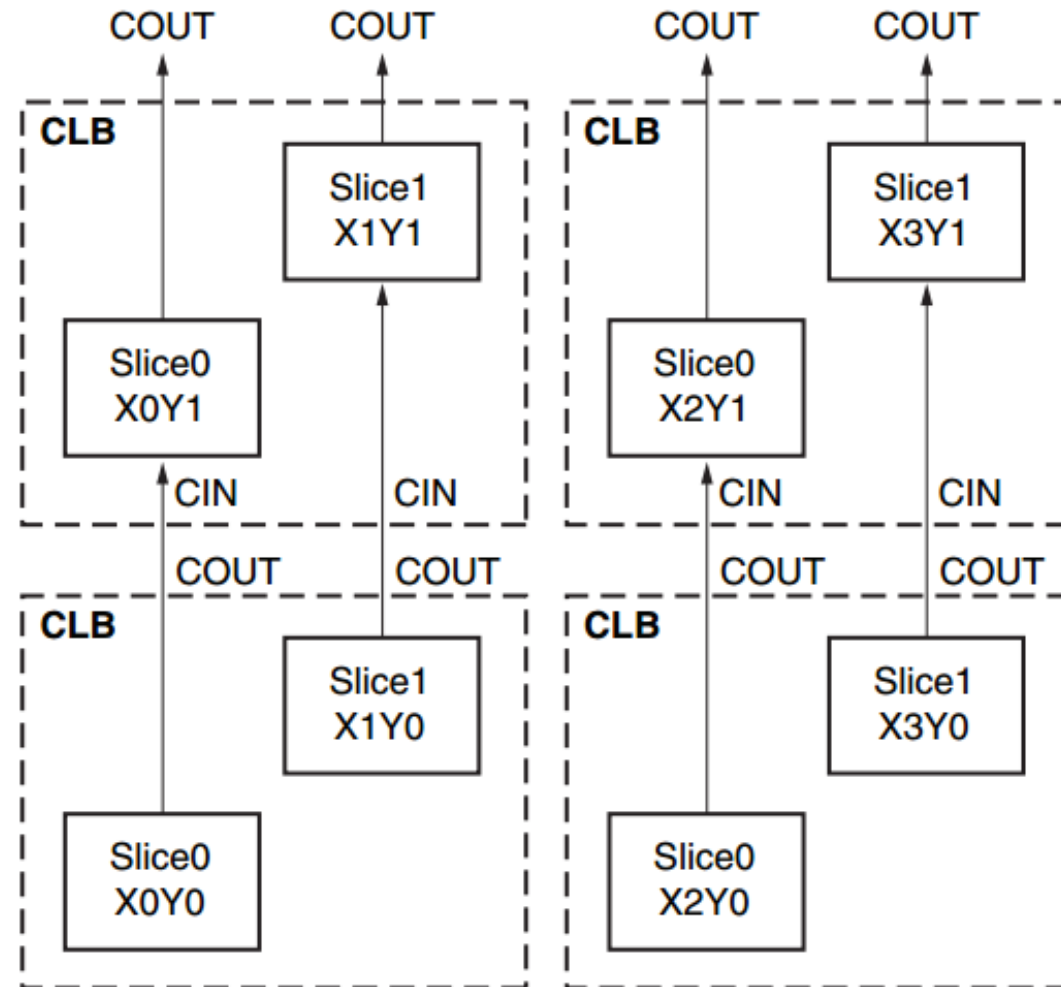
SLICEM



SLICEM Vs SLICEL

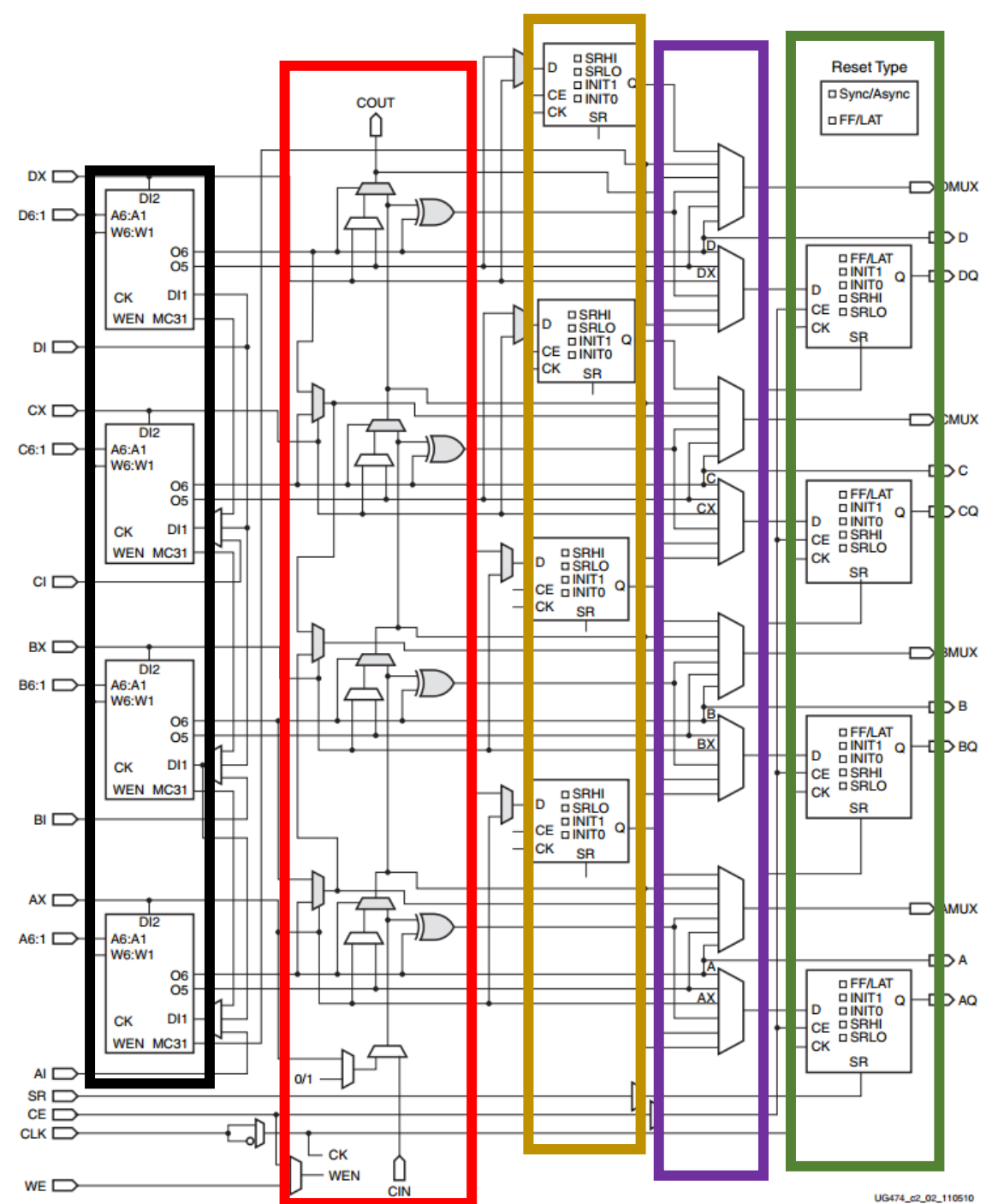


Configurable Logic Block (CLB)



Slice Resource

- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.



Vector and Memory

Verilog: Vectors

- Only “**net**” or “**reg**” data types can be declared as vectors (multiple bit width).
- Specifying vectors for **integer**, **real**, **realtime**, and **time** data types is **illegal**.
- Default: 1 bit (scalar). Example: **wire** [7:0] a_byte; **reg** [31:0] a_word;

```
reg [11:0] counter ;
```

```
reg a ;
```

```
reg [2:0] b ;
```

```
a = counter[7] ;    // bit seven is loaded into a
```

```
b = counter[4:2] ; // bits 4, 3, and 2 are loaded into b
```

Verilog: Vectors

- **MSB and LSB expressions** should be constant expressions and may be positive, negative, or zero.
- The LSB constant expression may be greater, equal or less than the MSB constant expression.
- **reg** [3:0] addr;

The 'addr' variable is a 4-bit vector register made up of addr[3] (the most significant bit), addr[2], addr[1], and addr[0] (the least significant bit).

- **wire** [-3:4] data;

The data variable is 8-bit vector net made up of data[-3] (msb), data[-2], data[-1], data[0], data[1], data[2], data[3], data[4] (lsb).

Verilog: Vectors

- 8-bit vector net called a_in:

```
wire [7:0] a_in ;
```

- A 32-bit storage register called address:

```
reg [31:0] address ;
```

- Set the value of the register to 32-bit decimal number equal to 3

```
address = 32'd3 ;
```

Address = 016001100
32bit

integer address

3 32'h3

Verilog: Vector Indexing

Assign OutSum = InA + InB;

// Break down a 40-bit vector string into 5 separate bytes

Use 5 bytes hard-coded slices

```
module indexarr ;  
  reg [39:0] str ; // string  
  initial  
  begin  
    str = "abcde" ;  
    $display("%s", str[7:0]) ;  
    $display("%s", str[15:8]) ;  
    $display("%s", str[23:16]) ;  
    $display("%s", str[31:24]) ;  
    $display("%s", str[39:32]) ;  
  end  
endmodule
```

// output: e, d, c, b, a

←←← } 4217g

SH = 1010001010111

1010111

11110001

Verilog: Vector Indexing

```
reg [63:0] word ;  
reg [3:0] byte_num ; //a value from 0 to 7.  
reg [7:0] byteN ;  
  
// If byte_num = 4  
byteN = word[byte_num*8+: 8] ; // = word[39:32]  
  
reg [31:0] a ;  
b = a[8+:16] ; // b = a[23:8]  
c = a[31-:8] ; // c = a[31-24]
```


Verilog: Vector Indexing

- Verilog allows indexing vectors using variable expression to perform dynamic parts select
- The syntax is as follows:

[base_expression +: width_expression] or
[base_expression -: width_expression]
- The base_expression can be a variable expression but **width_expression must be a constant**
- Offset direction indicates if the width_expression is added (+:) or subtracted (-:) from the base_expression

Verilog: Vector Indexing

// Break down a 40-bit vector string into 5 separate bytes

Use 5 bytes hard-coded slices

```
module indexarr ;  
  reg [39:0] str ; // string  
  initial  
    begin  
      str = "abcde" ;  
      $display("%s", str[7:0]) ;  
      $display("%s", str[15:8]) ;  
      $display("%s", str[23:16]) ;  
      $display("%s", str[31:24]) ;  
      $display("%s", str[39:32]) ;  
    end  
endmodule
```

Use indexed part select

```
module indexarr ;  
  reg [39:0] str ; // string  
  integer i ;  
  initial  
    begin  
      str = "abcde" ;  
      for (i = 0 ; i < 5 ; i = i + 1)  
        $display("%s", str[i*8+: 8]) ;  
    end  
endmodule
```

// output: e, d, c, b, a

Verilog: Memory

- Registers and memories can be declared in the same line
- `reg [3:0] mem[255:0], red;`
- This line declares **4-bit register 'red'** and **memory 'mem'**, which contains 256 4-bit words.

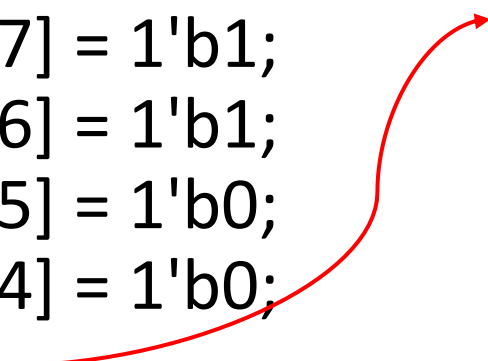
Verilog: Memory

- Elements of memory type can be accessed by memory index
- **reg** [7:0] mem [3:0], red;
mem[0] = 7;
red = mem[3];
mem[1] = red;

Verilog: Memory

- **Vector and memory declarations** are NOT the same.
- If a variable is declared as a **vector**, all bits can be assigned a value in one statement.
- If a variable is declared as **memory**, then a value to each element should be assigned separately

- **reg** [7:0] vect= 8'b11001010;
reg array[7:0];
array[7] = 1'b1;
array[6] = 1'b1;
array[5] = 1'b0;
array[4] = 1'b0;
array[3] = 1'b1;
array[2] = 1'b0;
array[1] = 1'b1;
array[0] = 1'b0;



Verilog: Memory

```
reg [7:0] my_reg [0:31];    // Array of 32 byte-wide registers
```

```
integer matrix [4:0] [0:255]; // 2-dimensional Array of integers
```

```
my_reg[15]; // Referencing the 16th byte of the array register
```

Verilog: Memory

```
wire [1:0] my_reg [0:3];  
wire [1:0] my_reg1 [3:0];  
  
assign my_reg[1]=2'b10;  
assign my_reg[3]=2'b11;  
assign my_reg1[1]=2'b10;  
assign my_reg1[3]=2'b11;
```

- The content of my_reg will be {Z,2,Z,3} and my_reg1 will be {3,Z,2,Z}

Verilog: Memory

```
reg [31:0] array2 [0:255][0:15] ;
```

- Select fourth byte from 101th row and 8th column.

```
wire [7:0] out2 = array2[100][7][31:24] ;
```

Verilog: Memory

1. Read 2nd byte from address 11 to data_out1.
2. Read 2nd and 3rd bytes from address 77 to data_out2.

```
reg [31:0] Data_RAM[0:255] ;  
output reg [7:0] data_out1 ;  
output reg [15:0] data_out2 ;
```

1. data_out1 = Data_RAM[11][15:8] ;
2. data_out2 = Data_RAM[77][23:8] ;

Verilog: Memory

- Memories are modeled as array of registers.

```
reg [7:0] my_memory[0:1023] ; // 1K bytes memory
// read a byte from address 511
data_out = my_memory[511] ;
// Write a byte to address 374
my_memory[374] = data_in ;
```

Self-study

🔲 Array declarations

```
reg [7:0] mema[0:255]; // declares a memory mema of 256 8-bit registers. The indices are 0 to 255
reg arrayb[7:0][0:255]; // declare a two-dimensional array of one bit registers
wire w_array[7:0][5:0]; // declare array of wires
integer inta[1:64];      // an array of 64 integer values
time chng_hist[1:1000] // an array of 1000 time values
integer t_index;
```

🔲 Assignment to array elements

```
mema = 0;      // Illegal syntax- Attempt to write to entire array
arrayb[1] = 0; // Illegal Syntax - Attempt to write to elements [1][0]..[1][255]
arrayb[1][12:31] = 0; // Illegal Syntax - Attempt to write to elements [1][12]..[1][31]
mema[1] = 0;    // Assigns 0 to the second element of mema
arrayb[1][0] = 0; // Assigns 0 to the bit referenced by indices [1][0]
inta[4] = 33559; // Assign decimal number to integer in array
chng_hist[t_index] = $time; // Assign current simulation time to element addressed by integer index
```