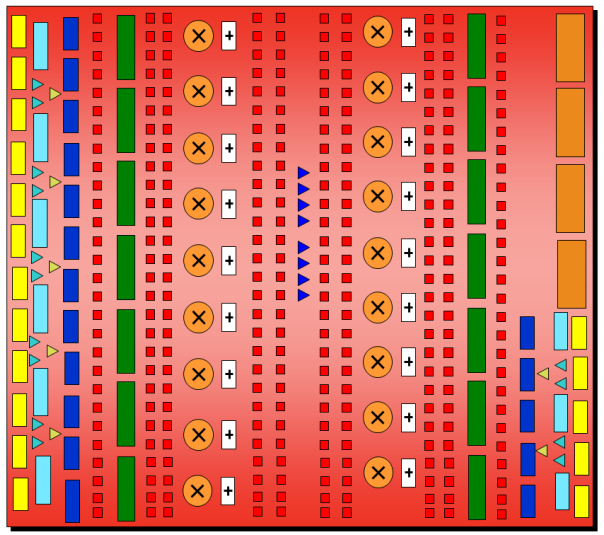
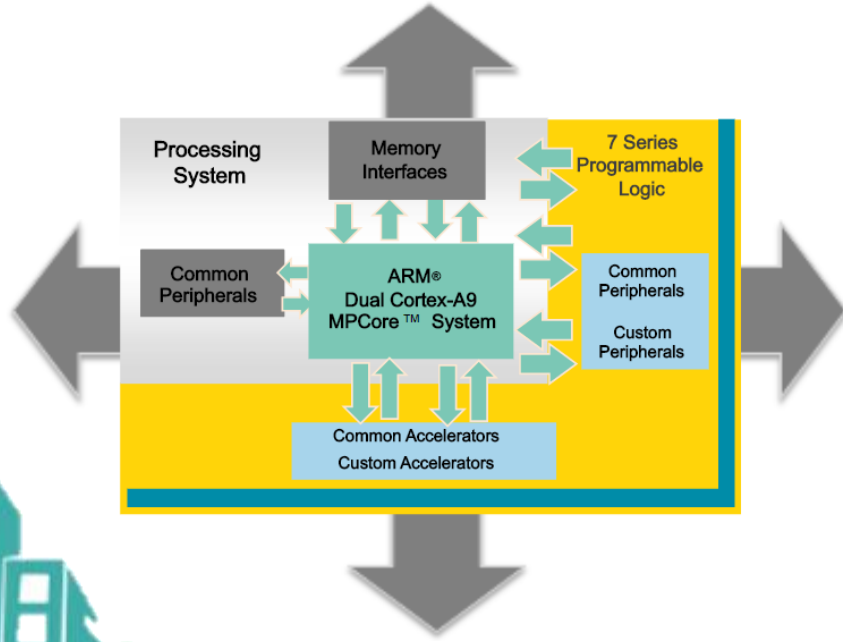
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

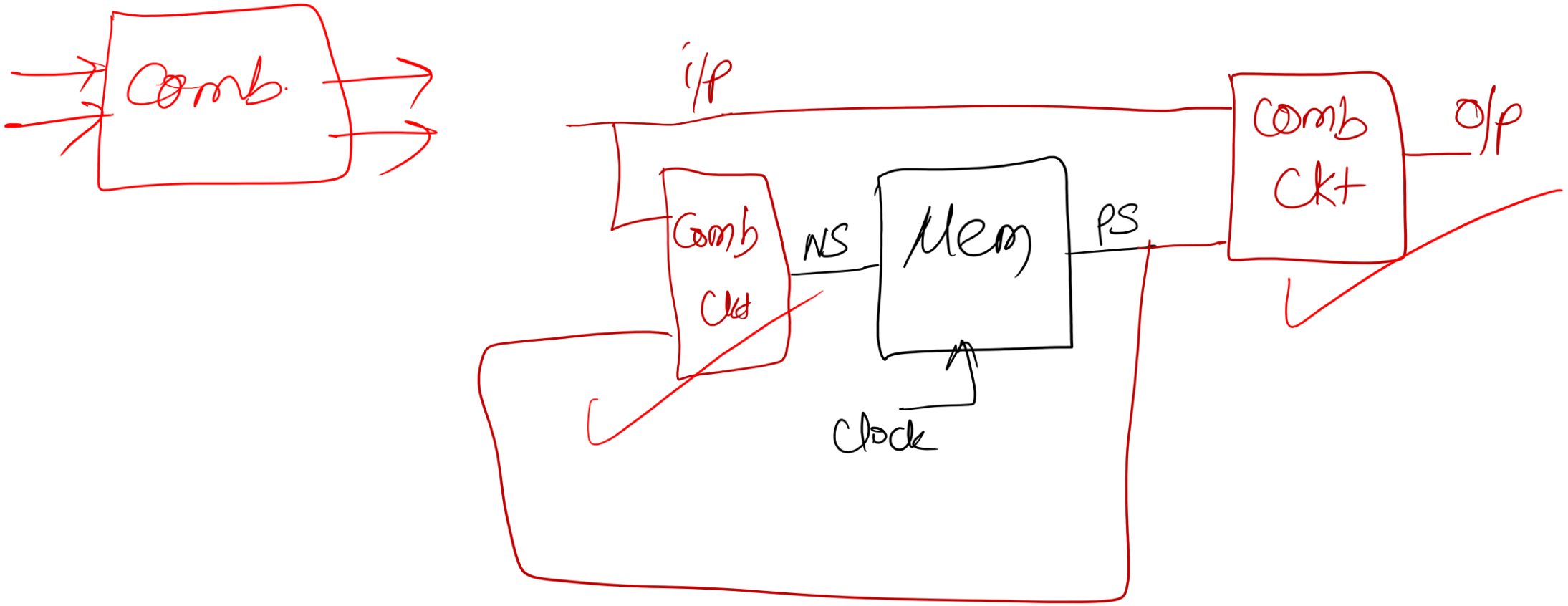
SDK
Software Development Kit
VIVADO™





Digital Circuits Revisited!

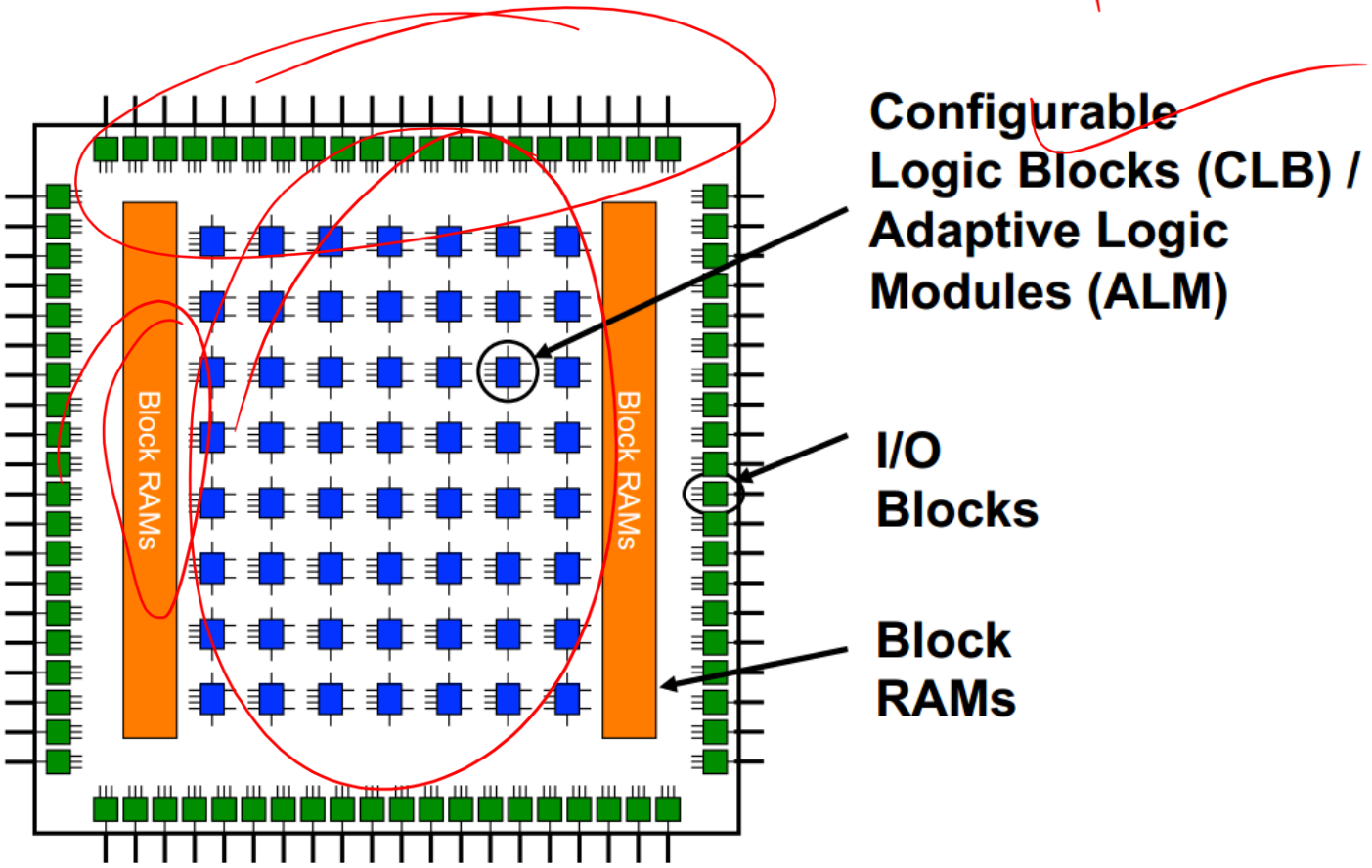
Combinational vs Sequential Circuits



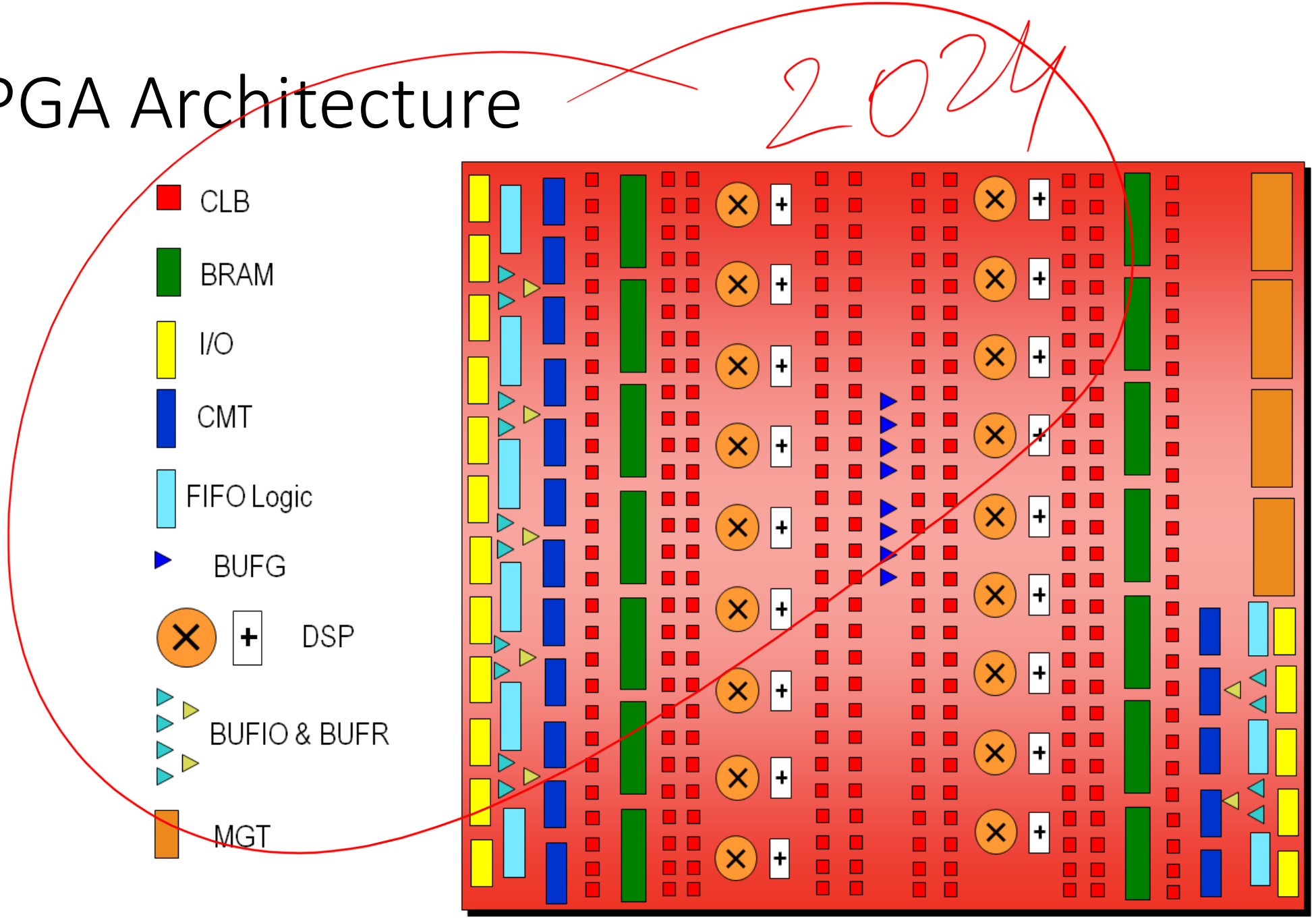
FPGA: Field Programmable Gate Array

1984

- **Array** of generic logic gates
- **Gates** where logic function can be programmed
- **Programmable** interconnection between gates
- **Field**: System can be reprogrammed in the field (After fabrication)

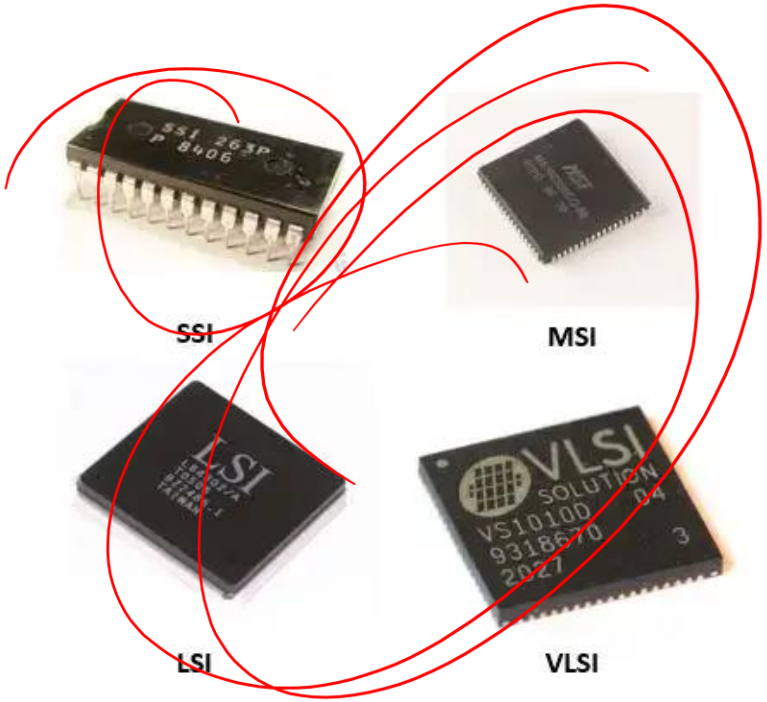


FPGA Architecture



Integrated Circuits (IC)

- Integrated circuit technology has improved to allow more and more components on chip



Integration Level	Numbers of Transistors	Equivalent Gates	Typical Functions of Systems	Typical Number I/Os
SSI	1-40	1-10	Single Circuit Function (e.g., Transistors)	14
MSI	40-400	10-100	Functional Network	24
LSI	400-4,500	100-1000	Hand Calculator or Digital Watch	48
VLSI	4,500-300,000	1,000-80,000	Microprocessor	64-300
ULSI	Over 300,000	Over 80,000	Small computer on a chip	300+
GSI	1 Billion	Over 100 Million	Supercomputer	10,000+

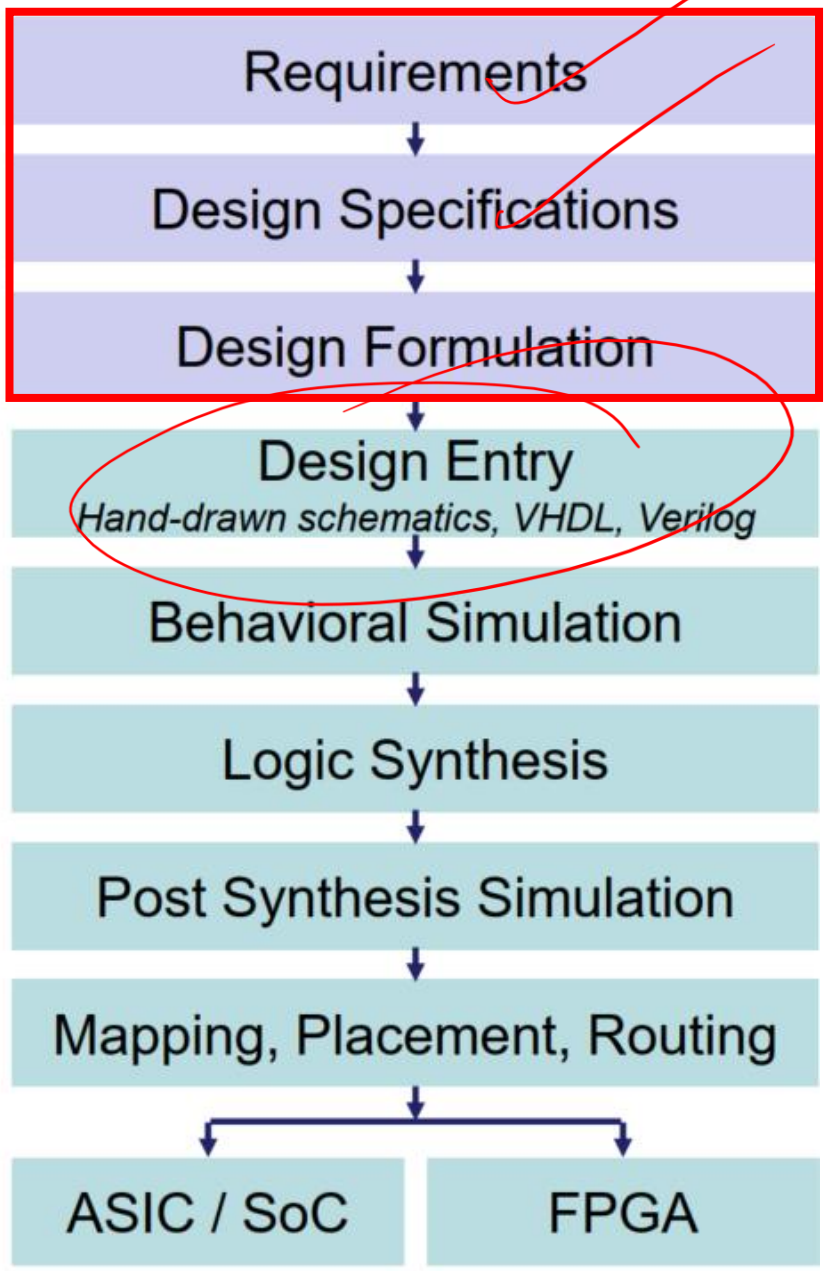
DC – Comb & Seq ckt.

– Breadboard ICs.

Generic FPGA Design Flow

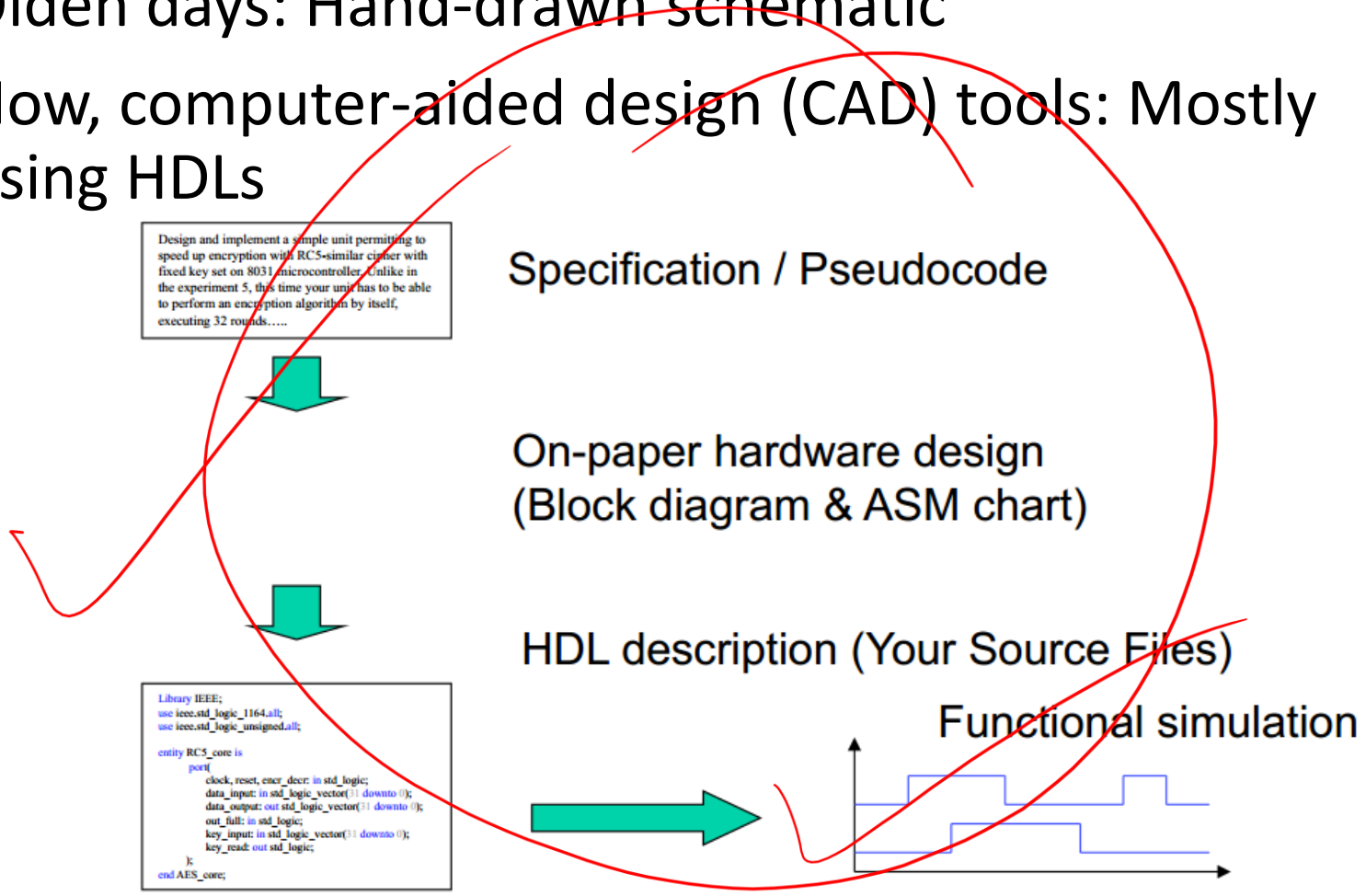
FPGA Design Flow

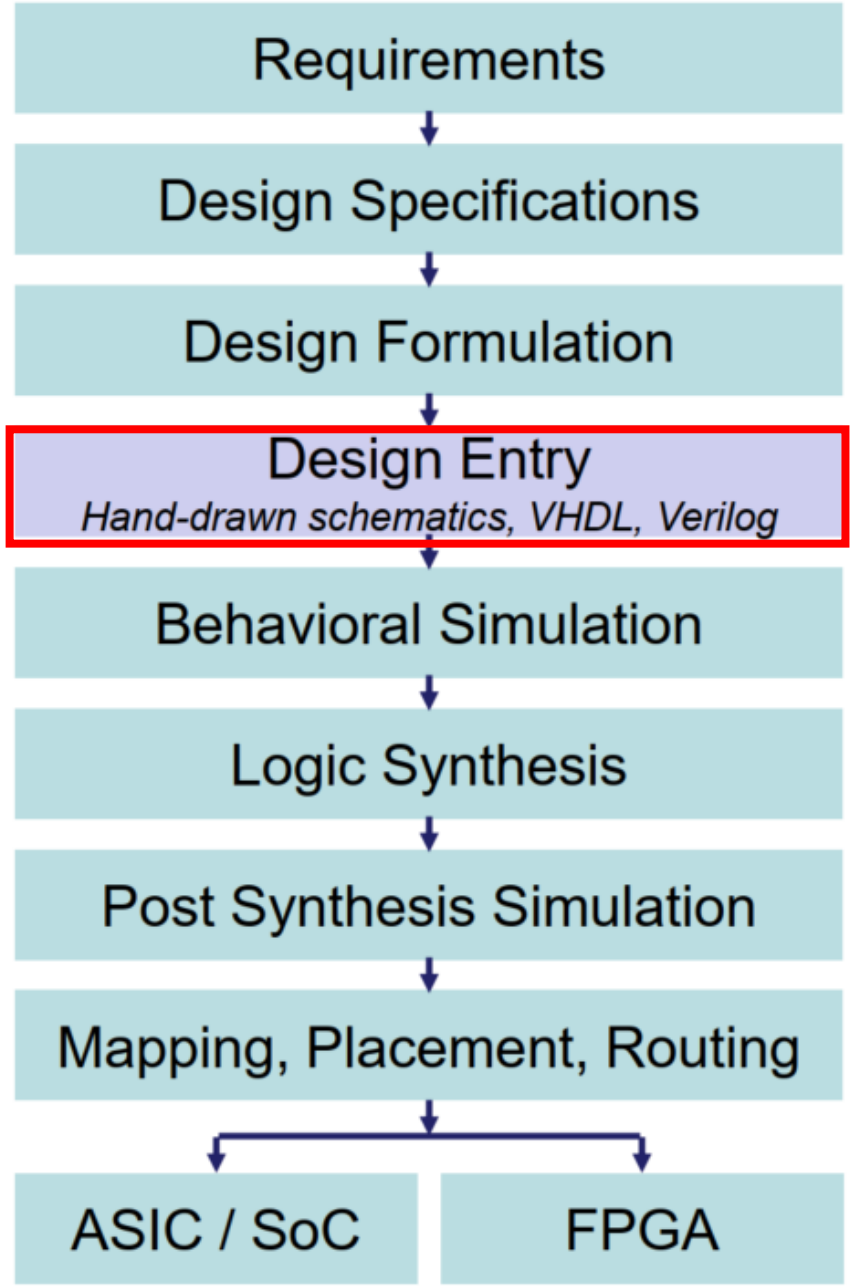
- All the designs start with design requirements and design specifications
- Next step is to formulate the design conceptually either at block diagram level or at an algorithmic level



FPGA Design Flow

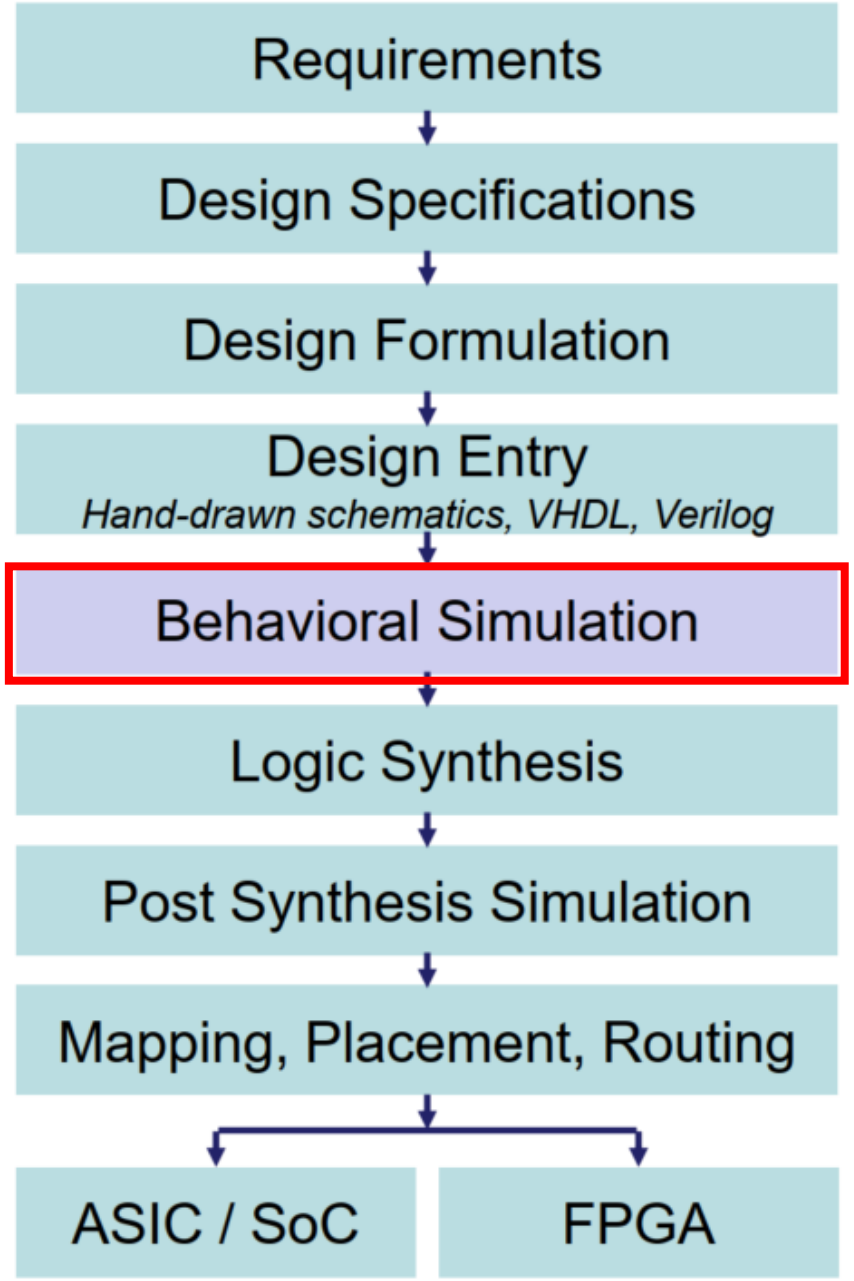
- Design Entry:
- Olden days: Hand-drawn schematic
- Now, computer-aided design (CAD) tools: Mostly using HDLs



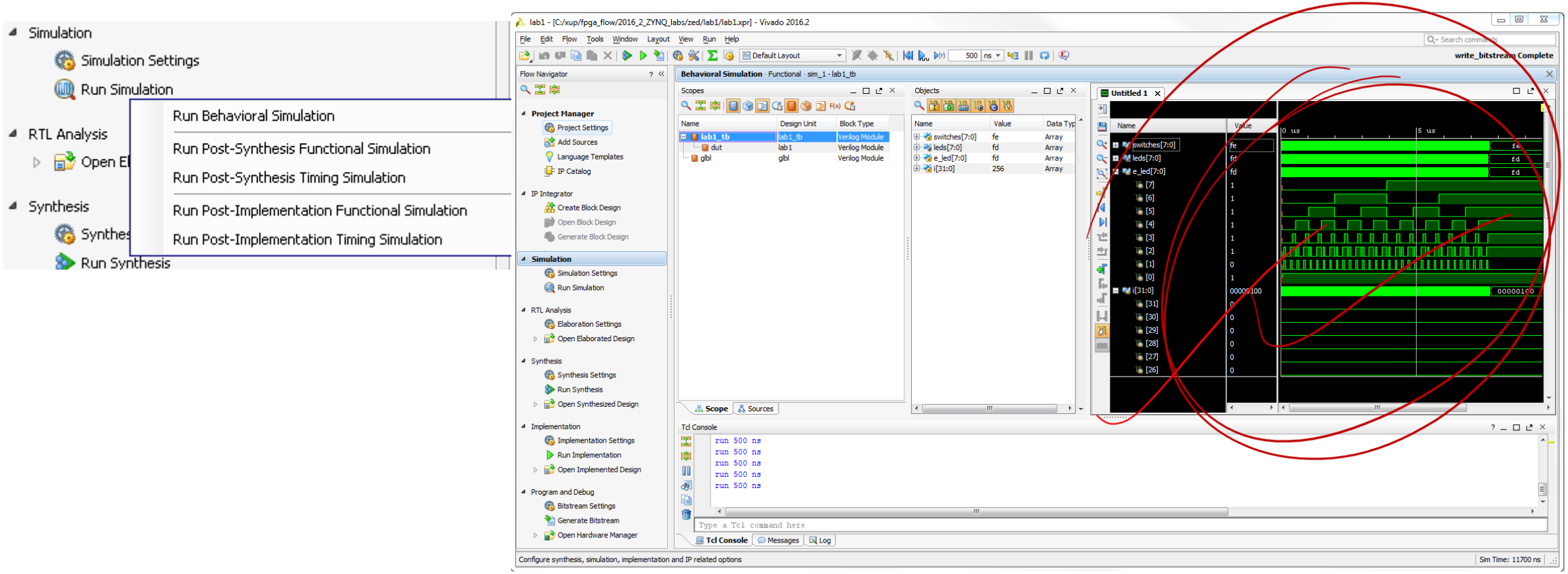


FPGA Design Flow

- Behavioral simulation to ensure that the design is functionally correct
- Can be done using: Test benches, test bench waveforms



Functional (HDL/RTL) Simulation



HDL/RTL Simulation

- Self-checking test bench: Contains output data and self-checking code that can be used to compare the data from later simulation runs
- When running a simulation with this kind of test bench, simulation outputs will be monitored. If a difference is detected between the predicted and the actual outputs, an error is reported.
- Automate the task of verifying simulation results i.e. no need to manually check waveform results

HDL/RTL Simulation

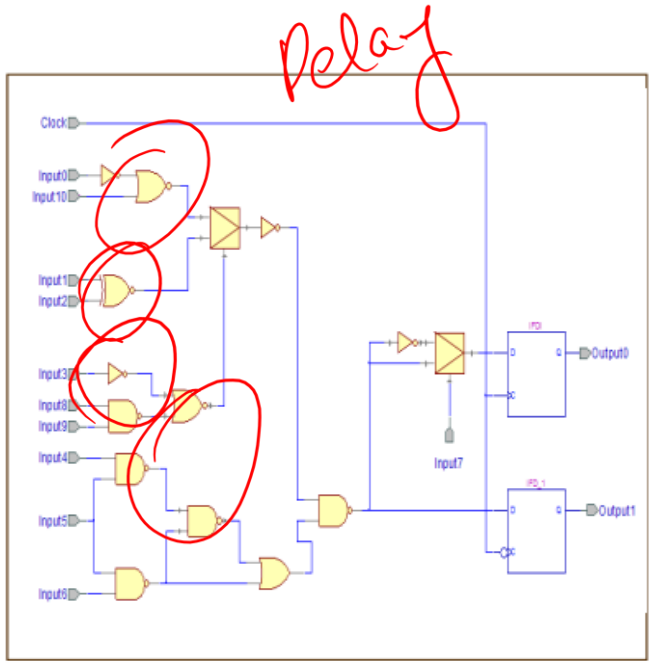
```
module testbench2();
  reg  a, b, c;
  wire y;

  // instantiate device under test
  sillyfunction dut(.a(a), .b(b), .c(c), .y(y));

  // apply inputs one at a time
  initial begin
     a = 0; b = 0; c = 0; #10; // apply input, wait
     if (y !== 1) $display("000 failed."); // check
     c = 1; #10;               // apply input, wait
     if (y !== 0) $display("001 failed."); // check
     b = 1; c = 0; #10;        // etc.. etc..
     if (y !== 0) $display("010 failed."); // check
  end
endmodule
```

FPGA Design Flow

VHDL description

```
architecture MLU_DATAFLOW of MLU is

signal A1:STD_LOGIC;
signal B1:STD_LOGIC;
signal Y1:STD_LOGIC;
signal MUX_0, MUX_1, MUX_2, MUX_3: STD_LOGIC;

begin
        A1<=A when (NEG_A='0') else
                not A;
        B1<=B when (NEG_B='0') else
                not B;
        Y<=Y1 when (NEG_Y='0') else
                not Y1;

        MUX_0<=A1 and B1;
        MUX_1<=A1 or B1;
        MUX_2<=A1 xor B1;
        MUX_3<=A1 xnor B1;

        with (L1 & L0) select
                Y1<=MUX_0 when "00",
                        MUX_1 when "01",
                        MUX_2 when "10",
                        MUX_3 when others;

end MLU_DATAFLOW;
```

Circuit netlist



Requirements

↓

Design Specifications

↓

Design Formulation

↓

Design Entry

Hand-drawn schematics, VHDL, Verilog

↓

Behavioral Simulation

↓

Logic Synthesis

↓

Post Synthesis Simulation

↓

Mapping, Placement, Routing

↓

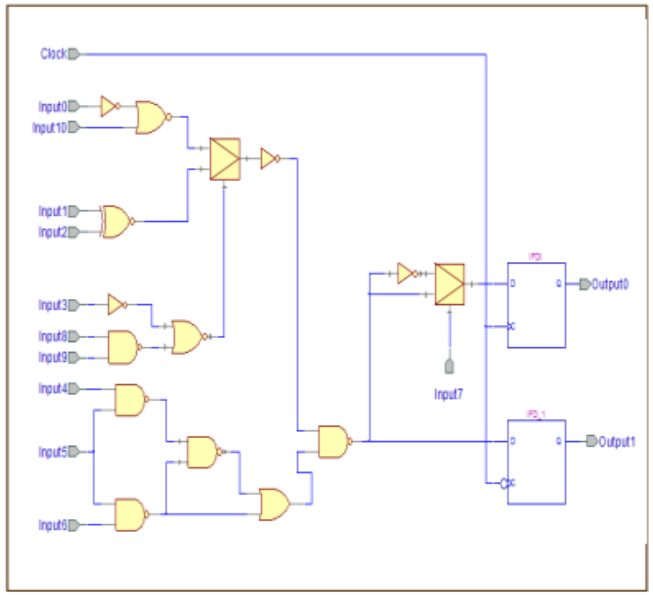
ASIC / SoC | FPGA

FPGA Design Flow

VHDL description

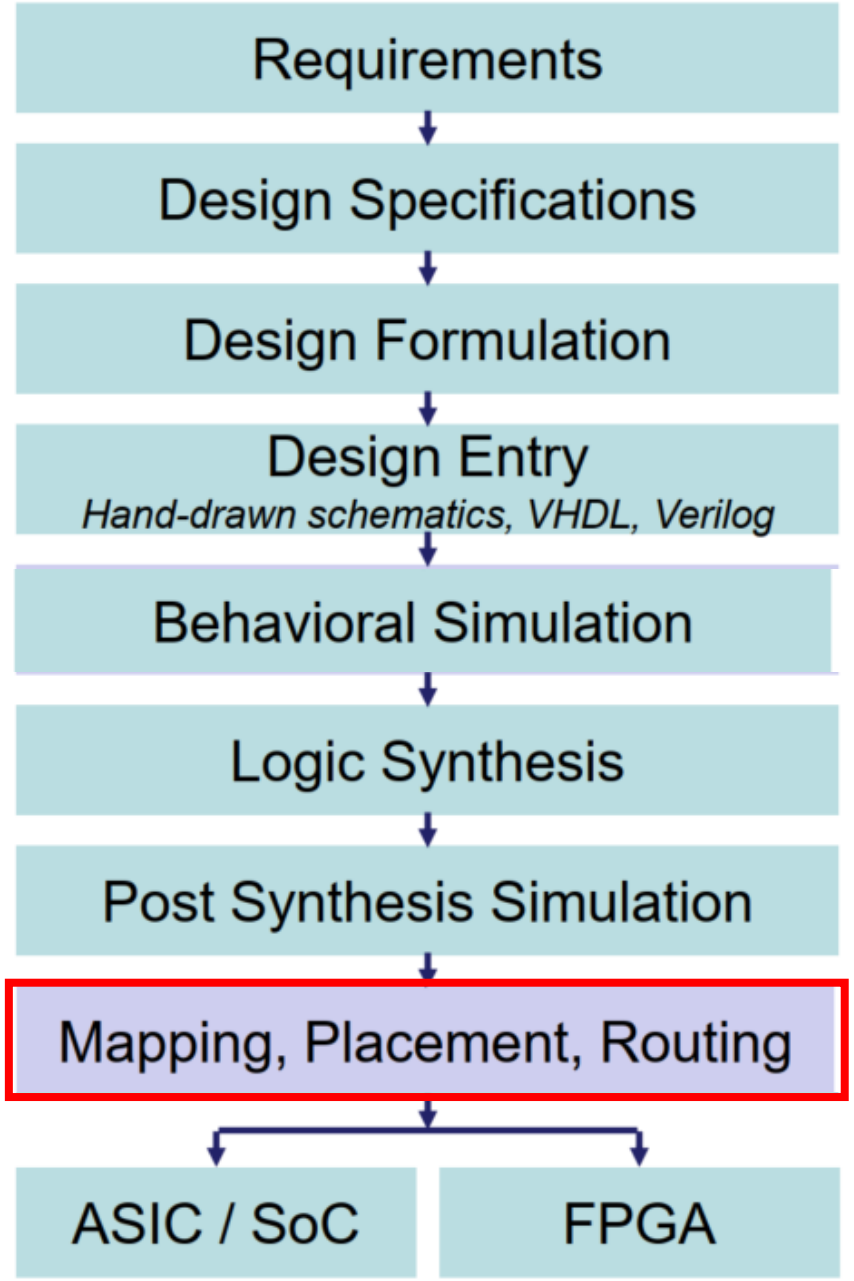
```
architecture MLU_DATAFLOW of MLU is

signal A1:STD_LOGIC;
signal B1:STD_LOGIC;
signal Y1:STD_LOGIC;
signal MUX_0, MUX_1, MUX_2, MUX_3: STD_LOGIC;

begin
        A1<=A when (NEG_A='0') else
                not A;
        B1<=B when (NEG_B='0') else
                not B;
        Y<=Y1 when (NEG_Y='0') else
                not Y1;

        MUX_0<=A1 and B1;
        MUX_1<=A1 or B1;
        MUX_2<=A1 xor B1;
        MUX_3<=A1 xnor B1;

        with (L1 & L0) select
                Y1<=MUX_0 when "00",
                        MUX_1 when "01",
                        MUX_2 when "10",
                        MUX_3 when others;

end MLU_DATAFLOW;
```

Circuit netlist



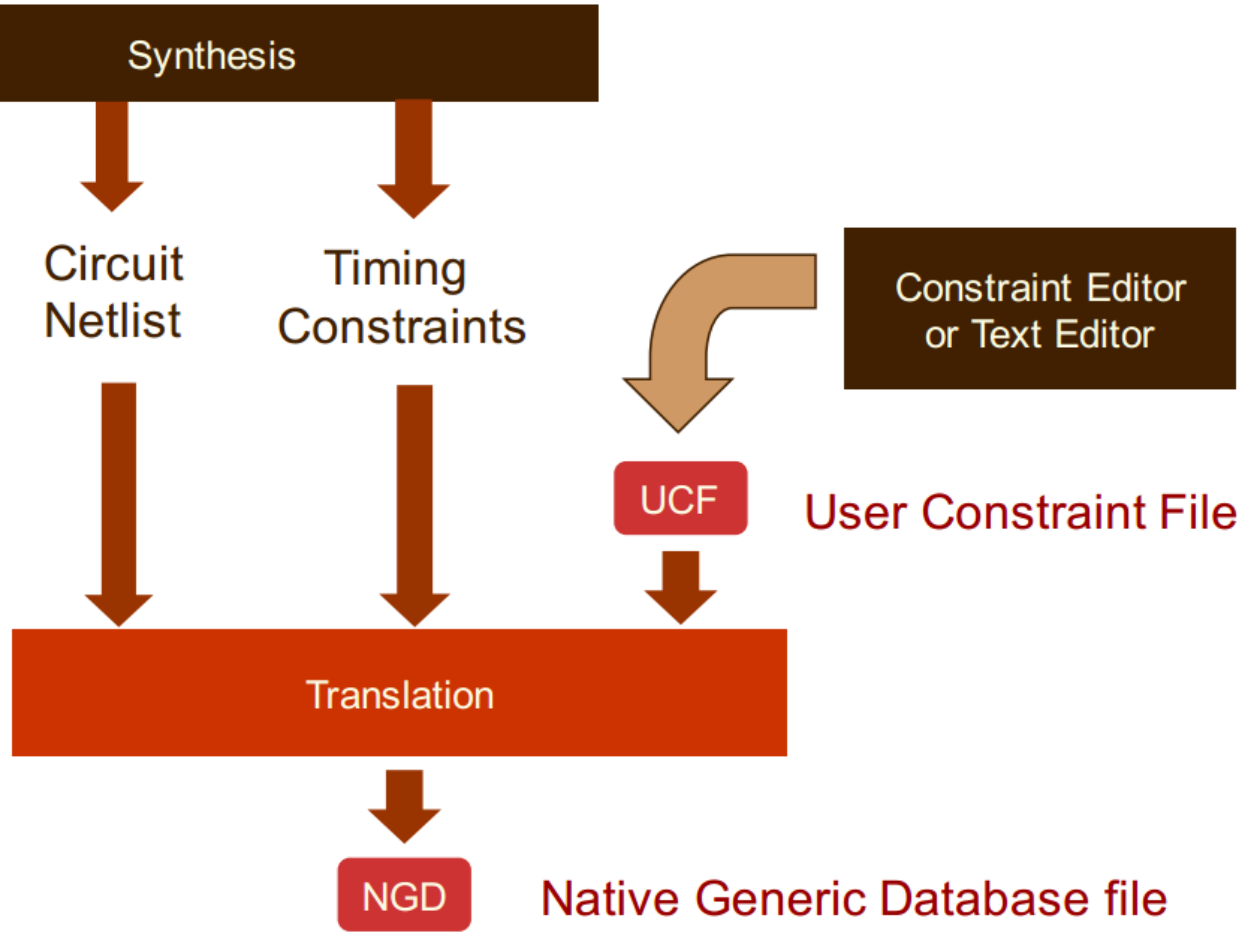
- Requirements
- Design Specifications
- Design Formulation
- Design Entry
  Hand-drawn schematics, VHDL, Verilog
- Behavioral Simulation
- Logic Synthesis
- **Post Synthesis Simulation**
- Mapping, Placement, Routing
- ASIC / SoC
- FPGA

FPGA Design Flow



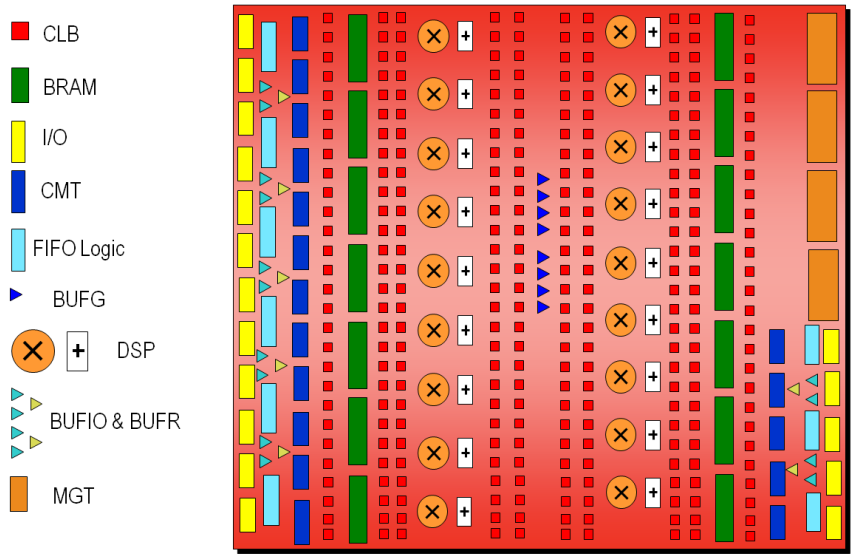
Translation

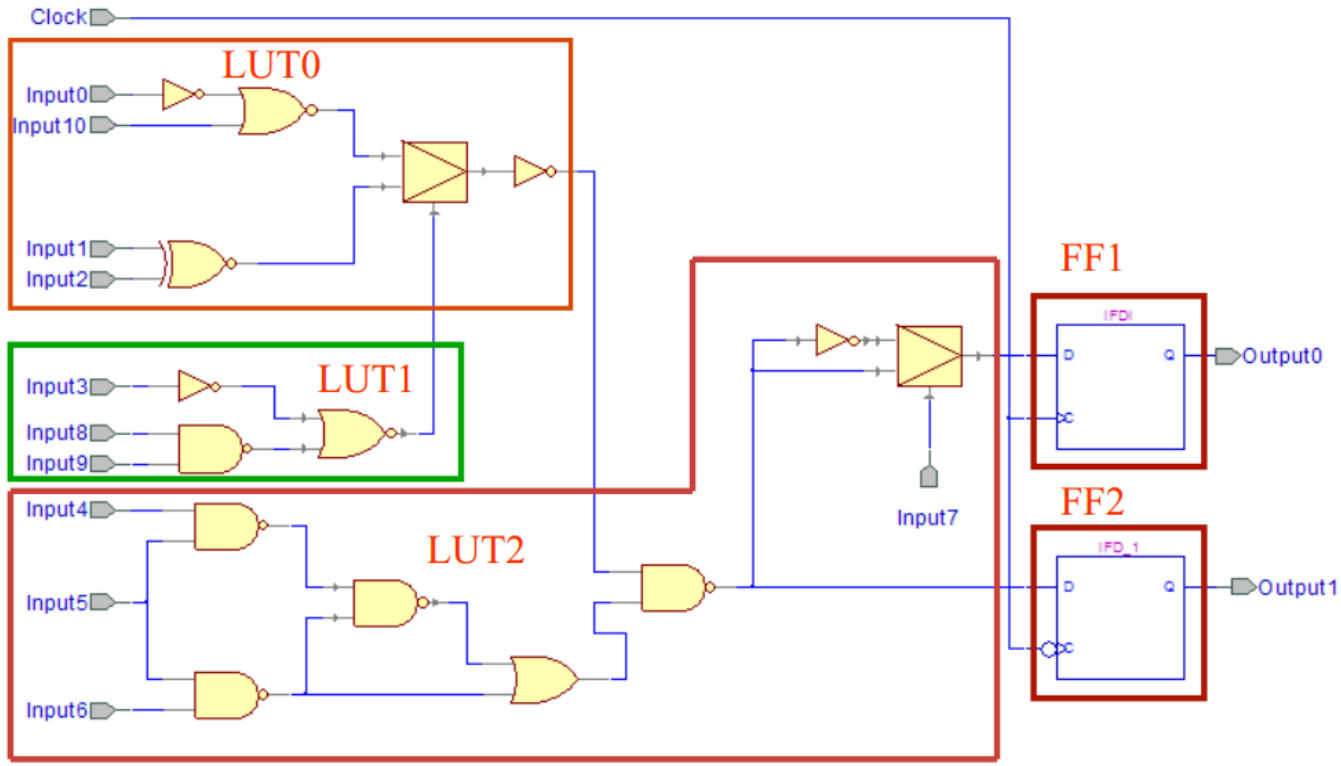
- Translate process merges all of the input netlists and design constraints and outputs a Xilinx native generic database (NGD) file



Mapping

- The Map process maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs.
- The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the FPGA.



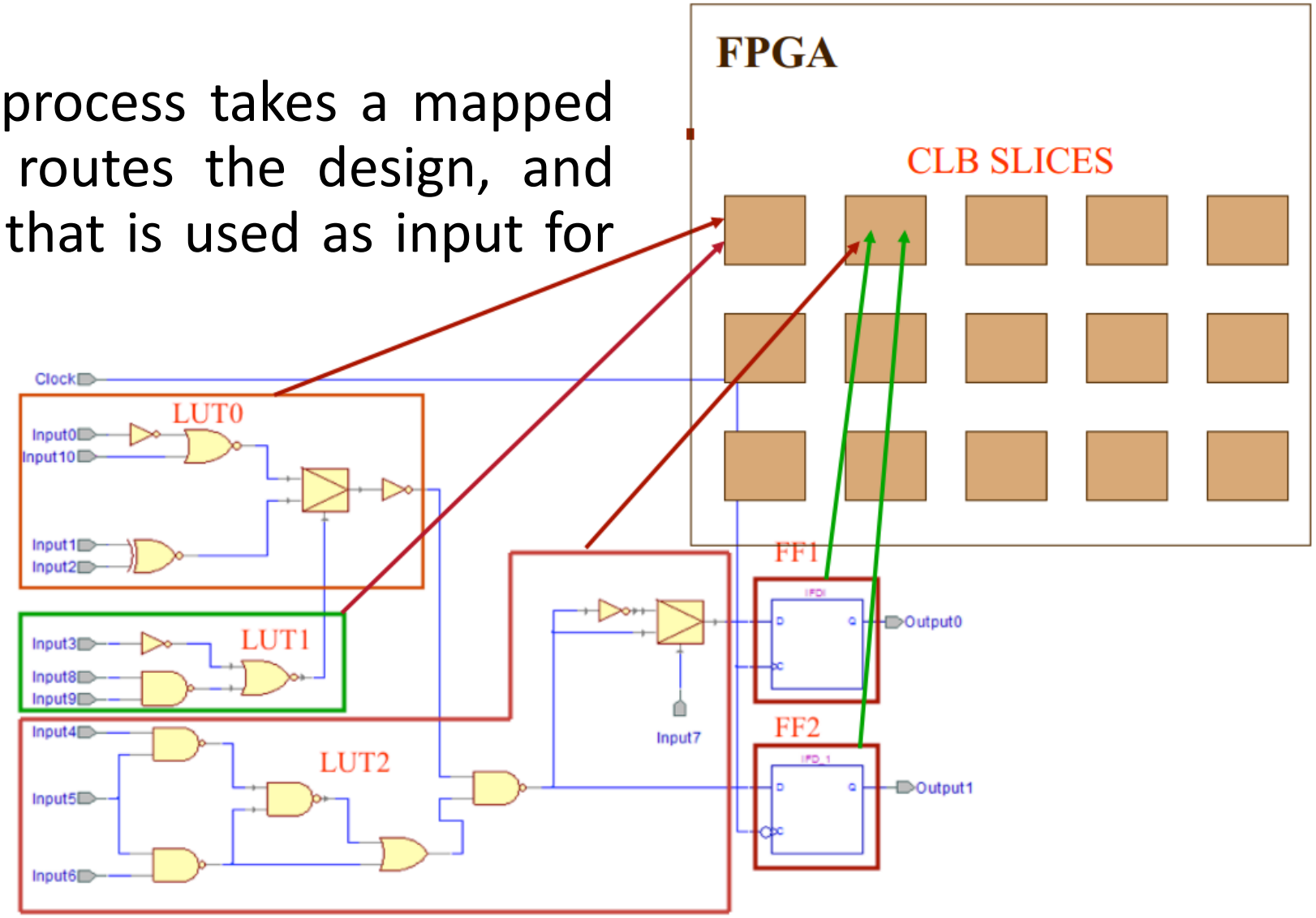


Post-map Static Timing

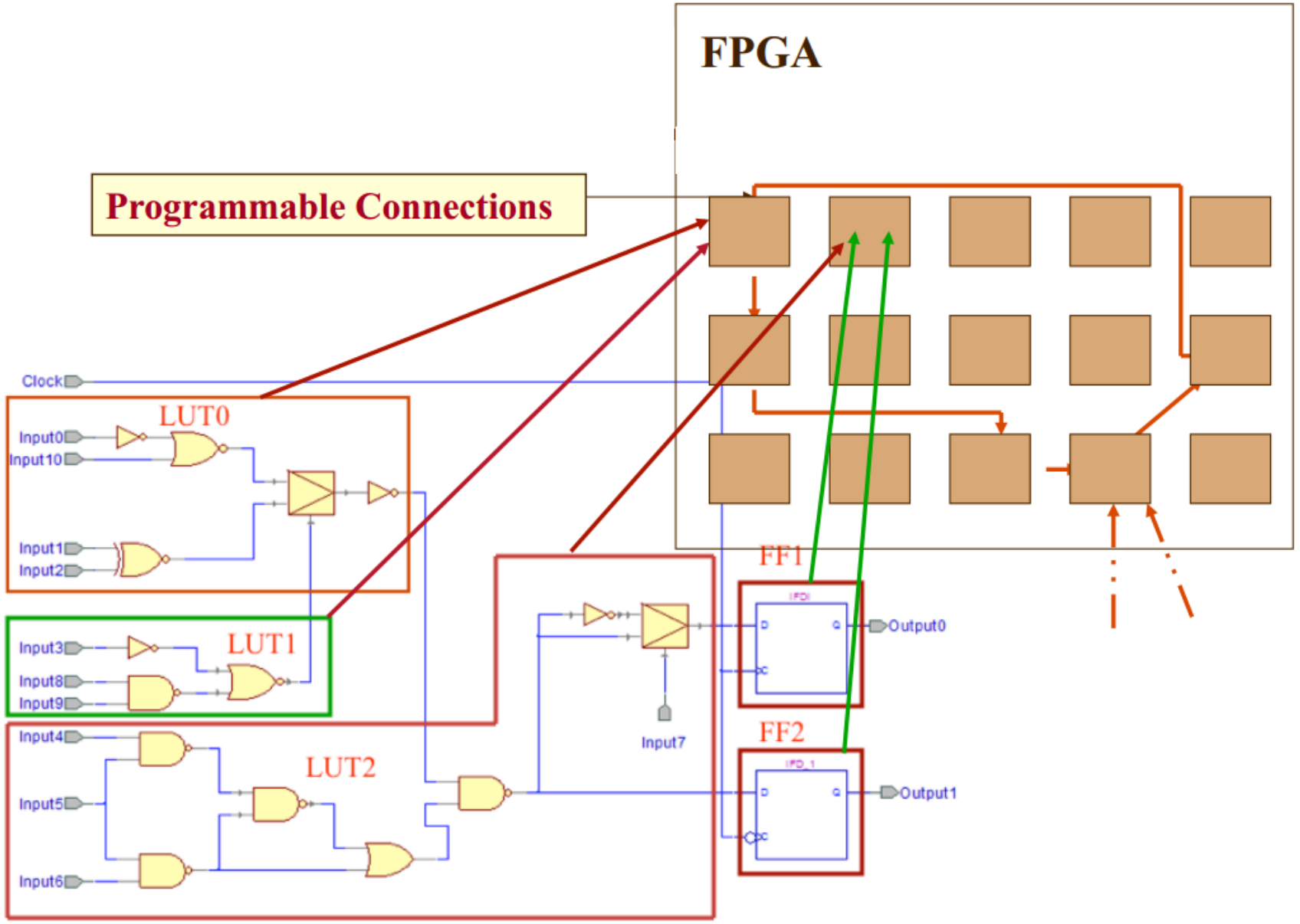
- You can generate a post-map static timing report for your design.
- It lists only the signal path delays in your design, derived from the design logic.
- Useful in evaluating timing performance of the logic paths, particularly if your design does not meeting timing requirements
- Route delays are not accounted (You can eliminate potential problems before investing time in examining routing delays)
- To eliminate problems, you may choose to redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the path.

Placing

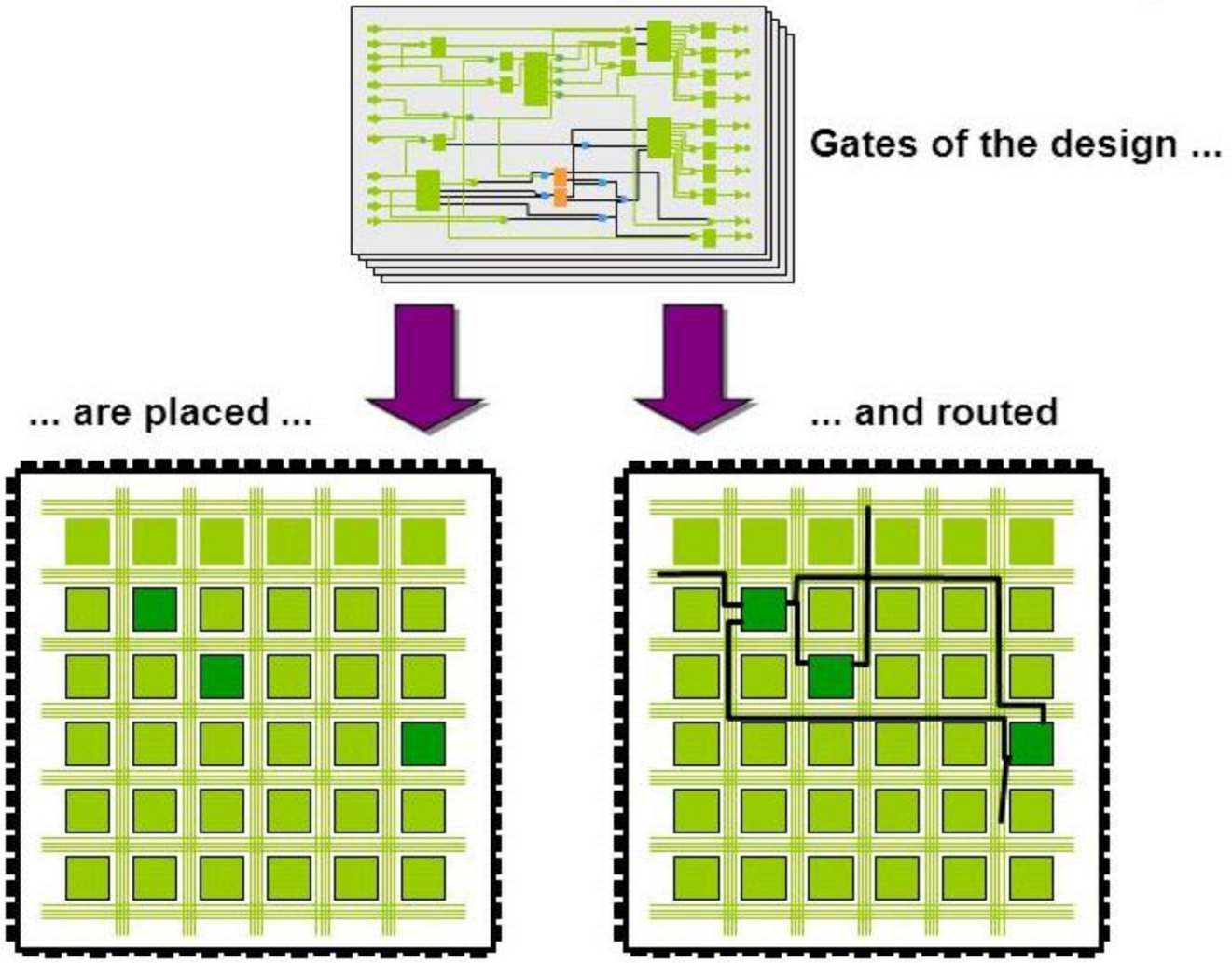
- The Place and Route process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bitstream generation.



Routing


FPGA
Programmable Connections
Clock
LUT0
Input0
Input10
Input1
Input2
LUT1
Input3
Input8
Input9
LUT2
Input4
Input5
Input6
Input7
FF1
FF2
Output0
Output1

Routing

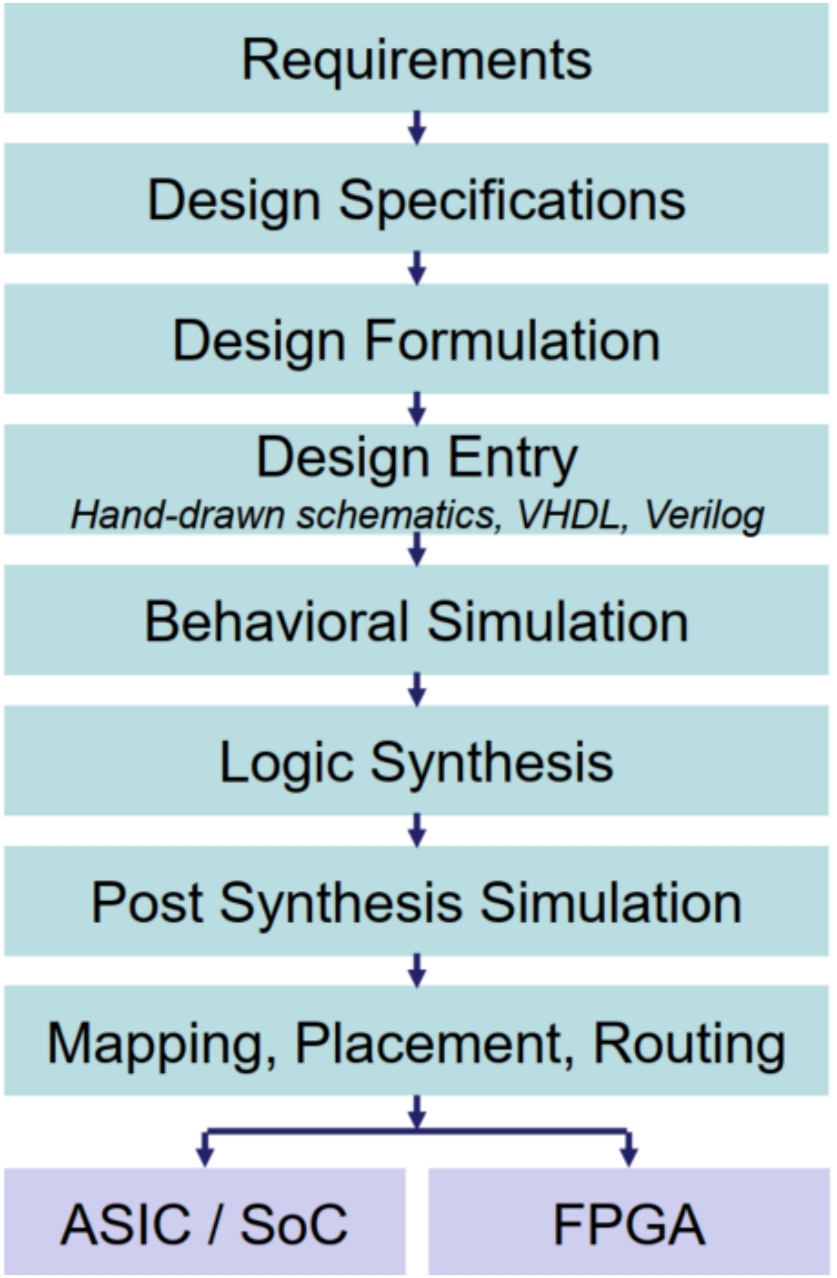


Post-place and Route Static Timing

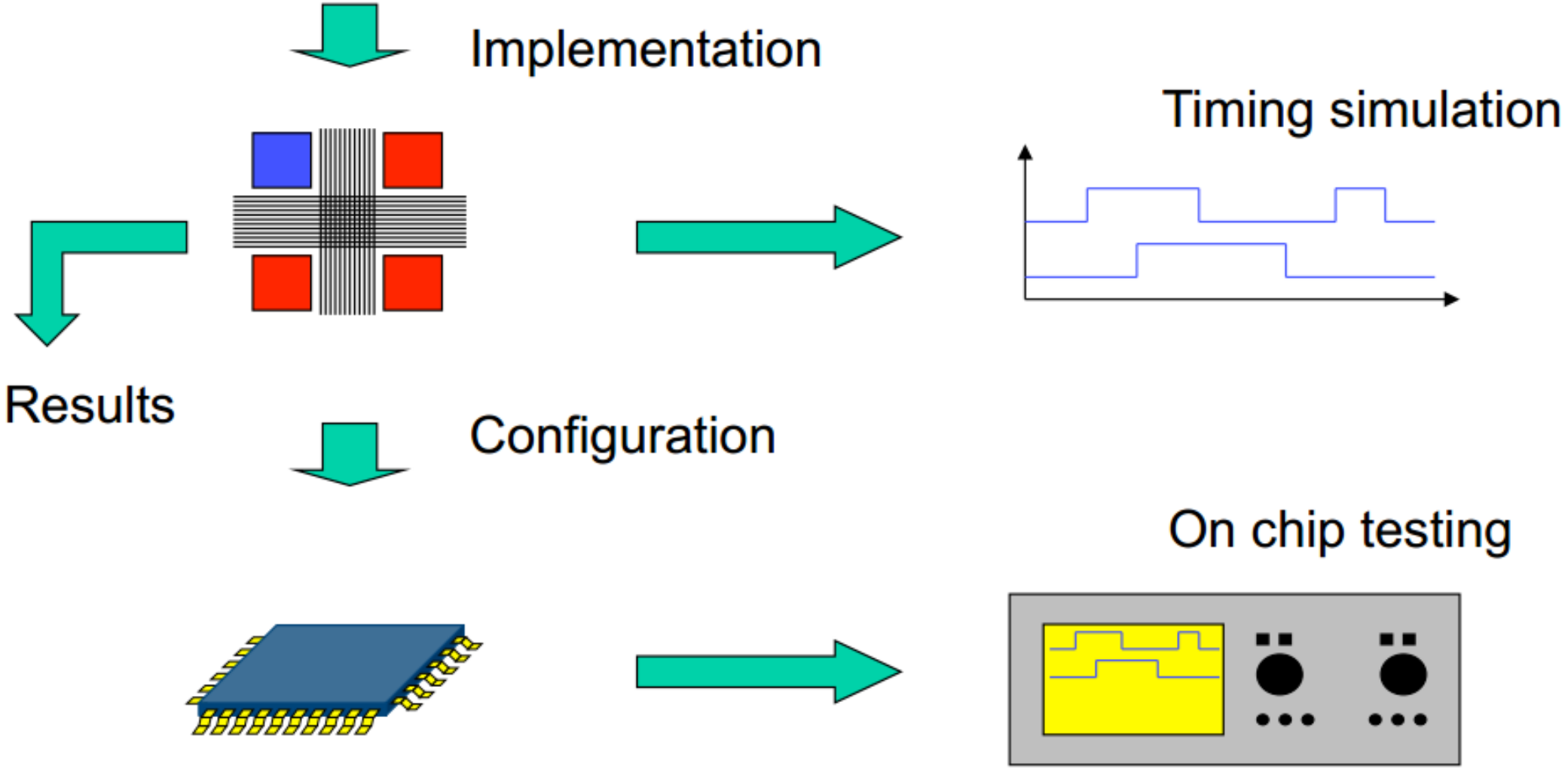
- Incorporates timing delay (signal and routing delay) information to provide a comprehensive timing summary of your design
- If you identify problems in the timing report, you can try fixing the problems by increasing the placer effort level, using re-entrant routing, or using multi-pass place and route.
- You can also redesign the logic paths to use fewer levels of logic, move to a faster device, or allocate more time for the paths.
- If a placed and routed design has met all of your timing constraints, then you can then create configuration data.

FPGA Design Flow

- FPGA programming simply involves writing a sequence of 0 and 1 into the programmable cells of FPGAs
- Once a design is implemented, you must create a file that the FPGA can understand. This file is called a bitstream: a BIT file (.bit extension)
- The BIT file can be downloaded directly to the FPGA or can be converted into a PROM file which stores the programming information.

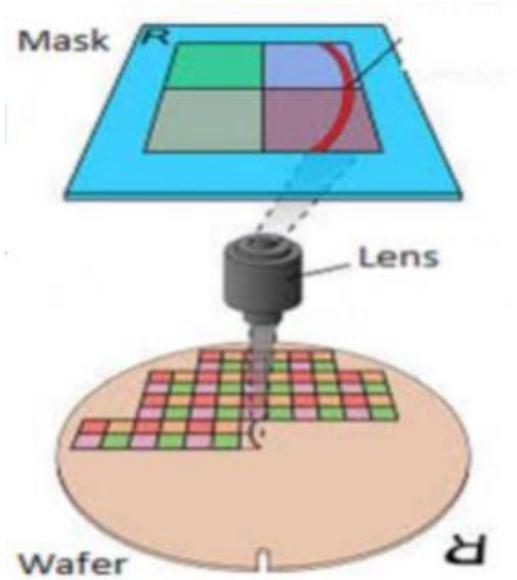


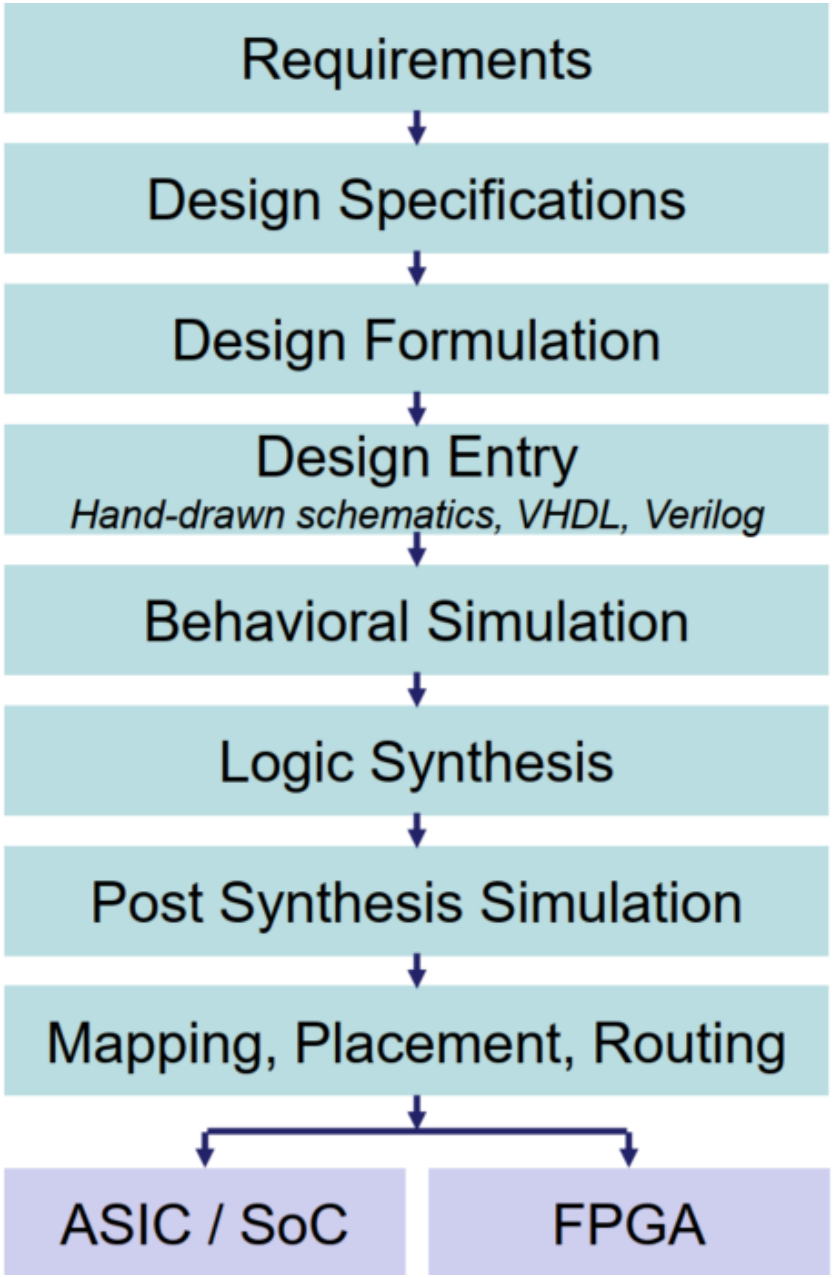
FPGA Design Process



ASIC Design Flow

- In ASIC, routed design is used to generate photomask for producing integrated circuits



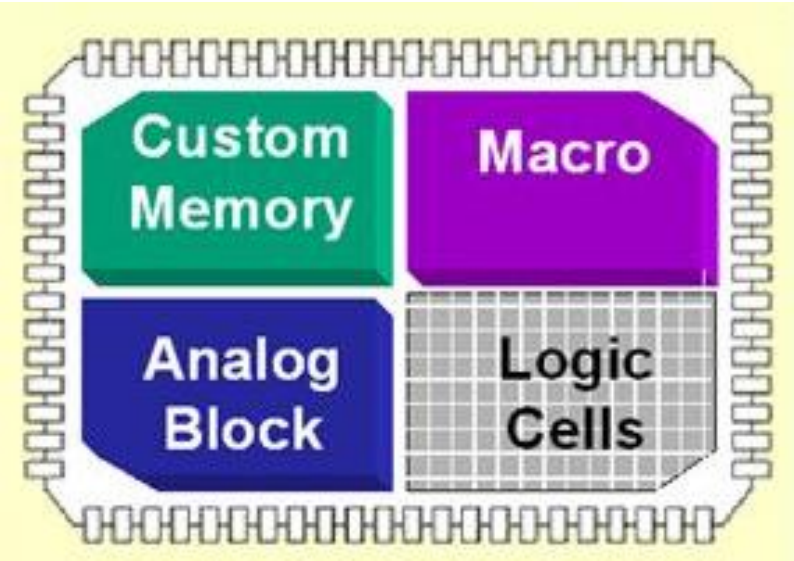


ASIC vs. FPGA vs. Microcontroller*

https://www.youtube.com/watch?v=vxSvQ-IcmHM

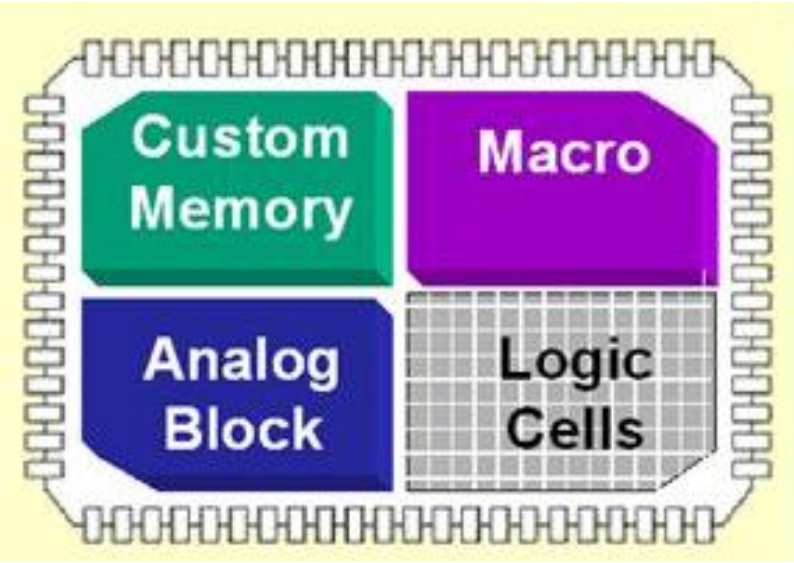
ASIC

- **ASIC**: Application Specific Integration Circuits

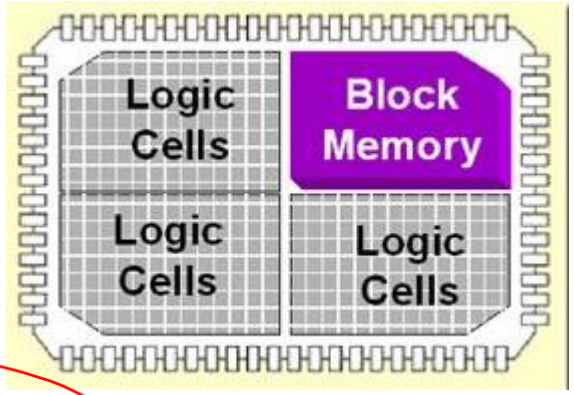


ASIC

- **ASIC**: Application Specific Integration Circuits
- High non-recurring engineering cost (one-time cost to research, design, develop and test a new product) and longest design cycle
- High cost for engineering change orders (Testing is critical! And hence, preferred when design is finalized)
- Lowest price for high volume production
- Fastest clock performance (high performance)
- Unlimited size and Low power
- Design and test tools are expensive
- Expensive IPs



FPGA



- Lowest cost for low to medium volume
- No NRE cost and Fastest time to market
- Field reconfigurable and partial reconfigurable
- Slower performance than ASIC (still 550 MHz)
- Limited size and steep learning curve
- Digital only*
- Industry often use FPGAs to prototype their chips before creating them (FPGA before ASIC).

FPGA/ASIC

FPGA vs ASIC

FPGA	ASIC
Reconfigurable circuitry after manufacturing	Fixed circuitry for product's lifespan
Suitable for digital designs only	Analog/mixed-signal circuitry can be fully implemented
Can be purchased as off-the-shelf products	Can only be designed as custom, private-label devices
Low-performance efficiencies, higher power consumption	Low power consumption, high-performance efficiencies
No non-recurring engineering (NRE) costs	NRE costs are part of the design process
Difficult to attain high-frequency rates	Operate at higher frequency rates
Faster time-to-market, high per unit costs	Long time-to-market, lower per unit costs
Are typically larger than ASICs	Can be much smaller than FPGA devices
Prototyping and validating with FPGAs is easier	Prototypes must be accurately validated to avoid design iterations
Lower barrier to entry for competitors	Higher barrier to entry for competitors

- Demand for specialized systems and short device life -> FPGAs

Microcontrollers

- Similar to simple computer placed in a single chip with all necessary components like memory, timers etc. embedded inside and performs a specific task.
- **Example:** Arduino, Pic
- Sequential execution, easy to use, control over **software**
- Consumes less power than FPGAs and mostly suitable for edge operations.
- Supports fixed as well as floating point operations
- **Microprocessors:** Completely different than microcontrollers.

Microprocessors

- ICs that come with a computer or CPU inside and are equipped with processing power. Examples: Pentium 3, 4, i5 etc.
- No peripherals such as RAM, ROM on the chip.
- Microprocessors form the heart of a computing system (general complex high-speed tasks) while microcontrollers drive embedded systems (specific tasks).
- Bulky due to the external peripherals

- **Expensive than micro-controllers**

Microprocessor vs FPGA vs ASIC

	Microprocessor	FPGA	ASIC
Example	ARM Cortex-A9	Virtex Ultrascale 440	Bitfury 16nm
Flexibility during development	Medium	High	Very high
Flexibility after development[1]	High	High	Low
Parallelism	Low	High	High
Performance[2]	Low	Medium	High
Power consumption	High	Medium	Low
Development cost	Low	Medium	High
Production setup cost[3]	None	None	High
Unit cost[4]	Medium	High	Low
Time-to-market	Low	Medium	High

[1]E.g. to fix bugs, add new functionality when already in production

[2]For a sufficiently parallel application

[3]Cost of producing the first chip

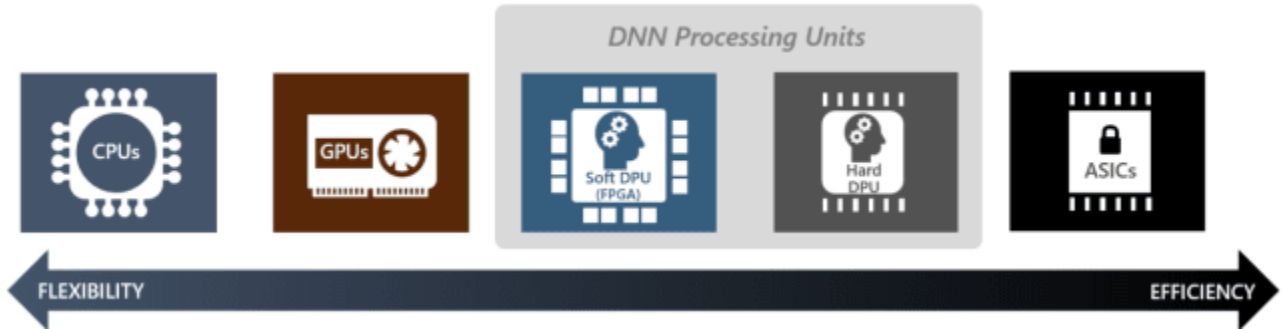
Microprocessor vs FPGA vs ASIC vs GPU

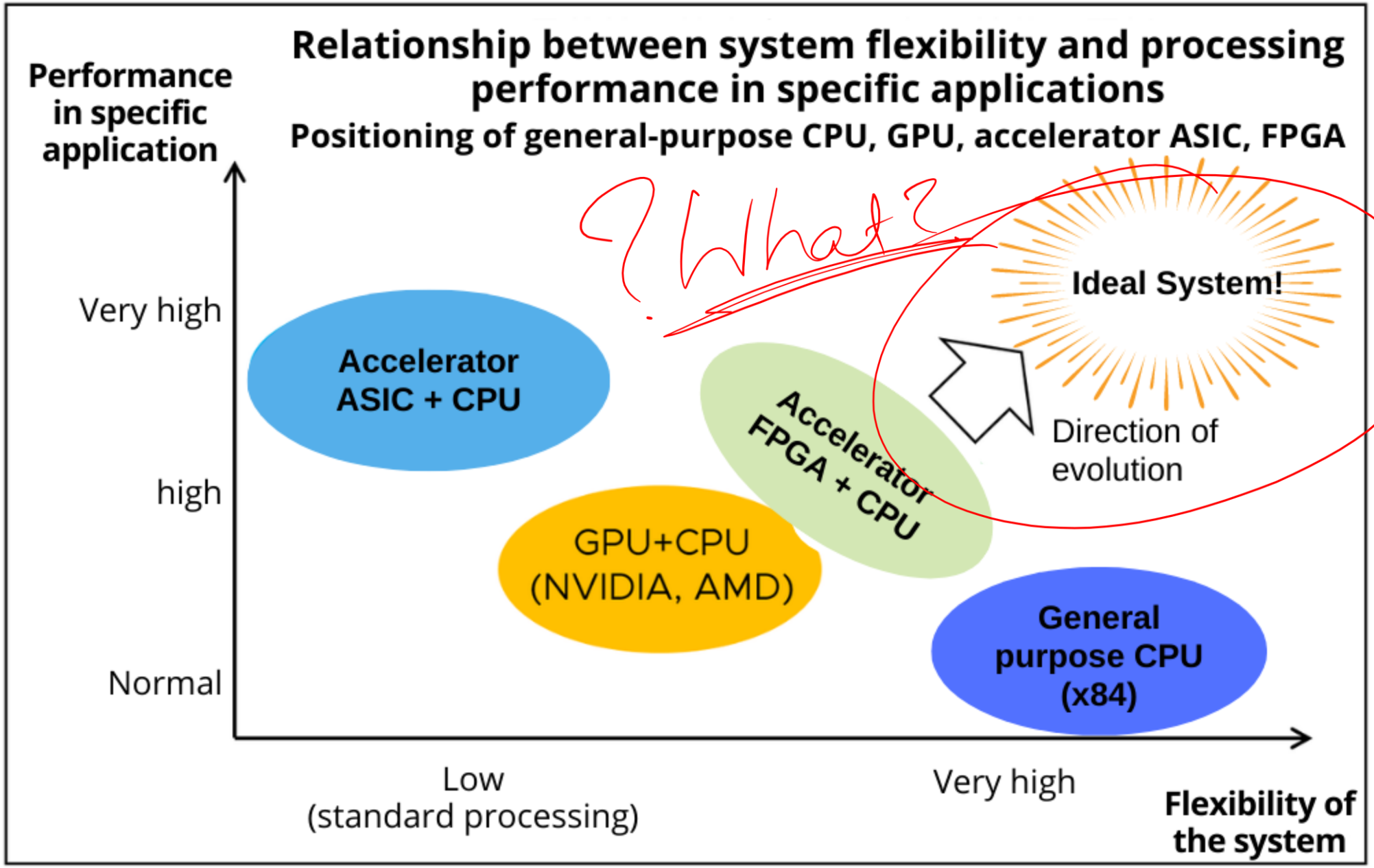
	Microprocessor	FPGA	ASIC	GPU
Example	ARM Cortex-A9	Virtex Ultrascale 440	Bitfury 16nm	Nvidia Titan X
Flexibility during development	Medium	High	Very high	Low
Flexibility after development[1]	High	High	Low	High
Parallelism	Low	High	High	Medium
Performance[2]	Low	Medium	High	Medium
Power consumption	High	Medium	Low	High
Development cost	Low	Medium	High	Low
Production setup cost[3]	None	None	High	None
Unit cost[4]	Medium	High	Low	High
Time-to-market	Low	Medium	High	Medium

[1]E.g. to fix bugs, add new functionality when already in production

[2]For a sufficiently parallel application

[3]Cost of producing the first chip




Relationship between system flexibility and processing performance in specific applications
Positioning of general-purpose CPU, GPU, accelerator ASIC, FPGA
Performance in specific application
Very high
high
Normal
Low (standard processing)
Very high
Flexibility of the system
What?
Ideal System!
Direction of evolution
Accelerator ASIC + CPU
Accelerator FPGA + CPU
GPU+CPU (NVIDIA, AMD)
General purpose CPU (x84)

Summary

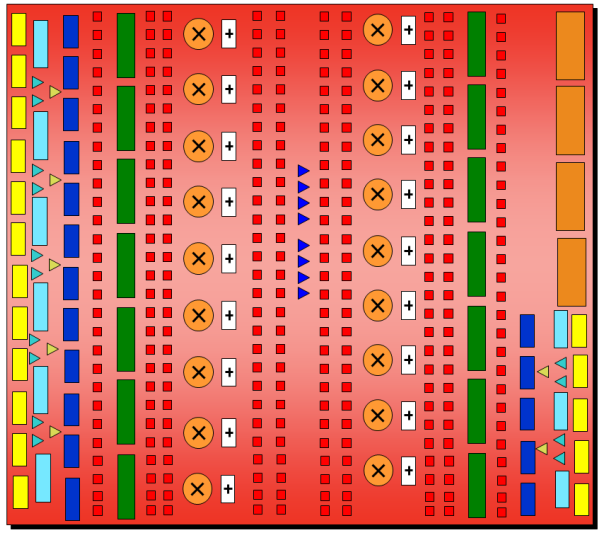
- **FPGA/ASIC:** Parallel execution, HDL (Verilog/VHDL), control over **hardware**
- You can make microcontroller inside FPGA/ASIC but not the other way round
- Time vs space limited
- FPGAs can perform any task while microcontrollers are limited by instruction sets while ASICs are application specific. FPGA are field-reconfigurable.
- Power consumption is high in FPGA
- FPGAs/ASICs can not be avoided in applications with stringent computational and memory requirements or applications with high level of determinism
- New world SoC: ARM + FPGA + GPU

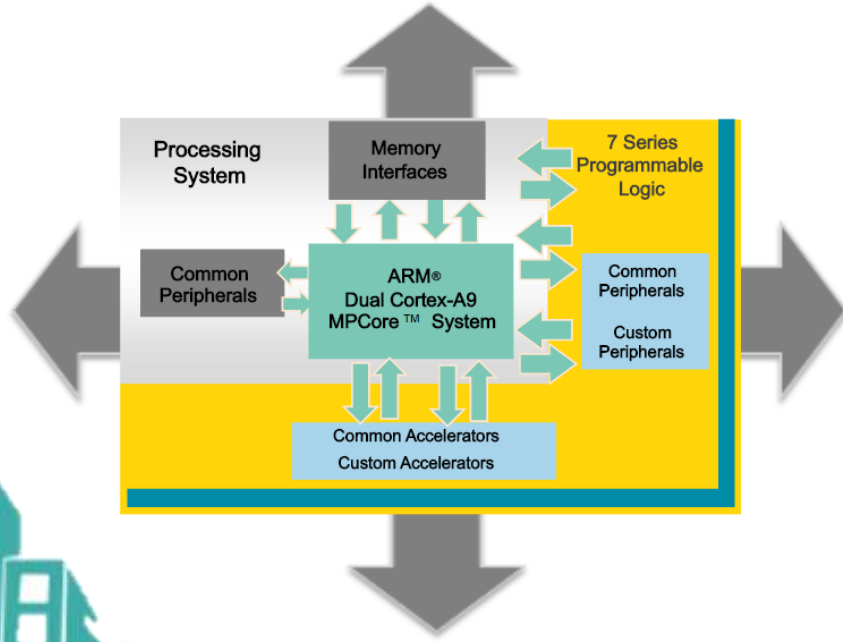
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

SDK
Software Development Kit
VIVADO™



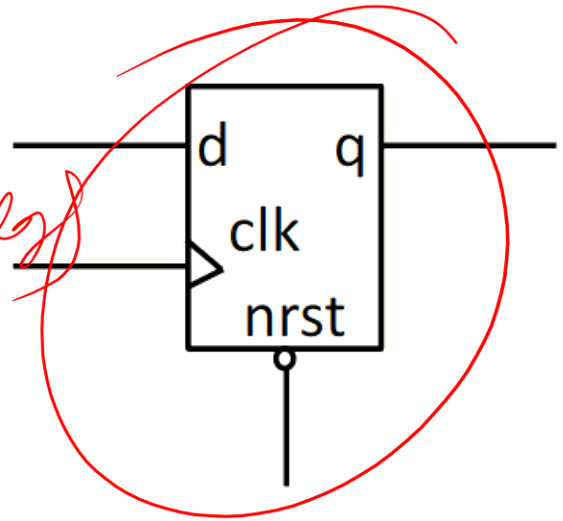


Verilog

```
module MUX (in1, in2, in3, in4, in5, in6, in7, in8, sel, out);
input in1, in2, in3, in4, in5, in6, in7, in8;
input [2:0] sel;
output reg out;

always @(*)
  begin
    if(sel==3'b000)
      out = in1;
    else if (sel==3'b001)
      out = in2;
```

Verilog: Module (Examples)

```
module D_FF(clk, nrst, d, q) ;
    input clk, nrst, d ;
    output reg q ;
    always @(posedge clk or negedge nrst)
                                    // Event-based Timing Control
      if (!nrst)                    // reset state
         q <= 0 ;
      else                          // normal operation
       q <= d ;
endmodule
```

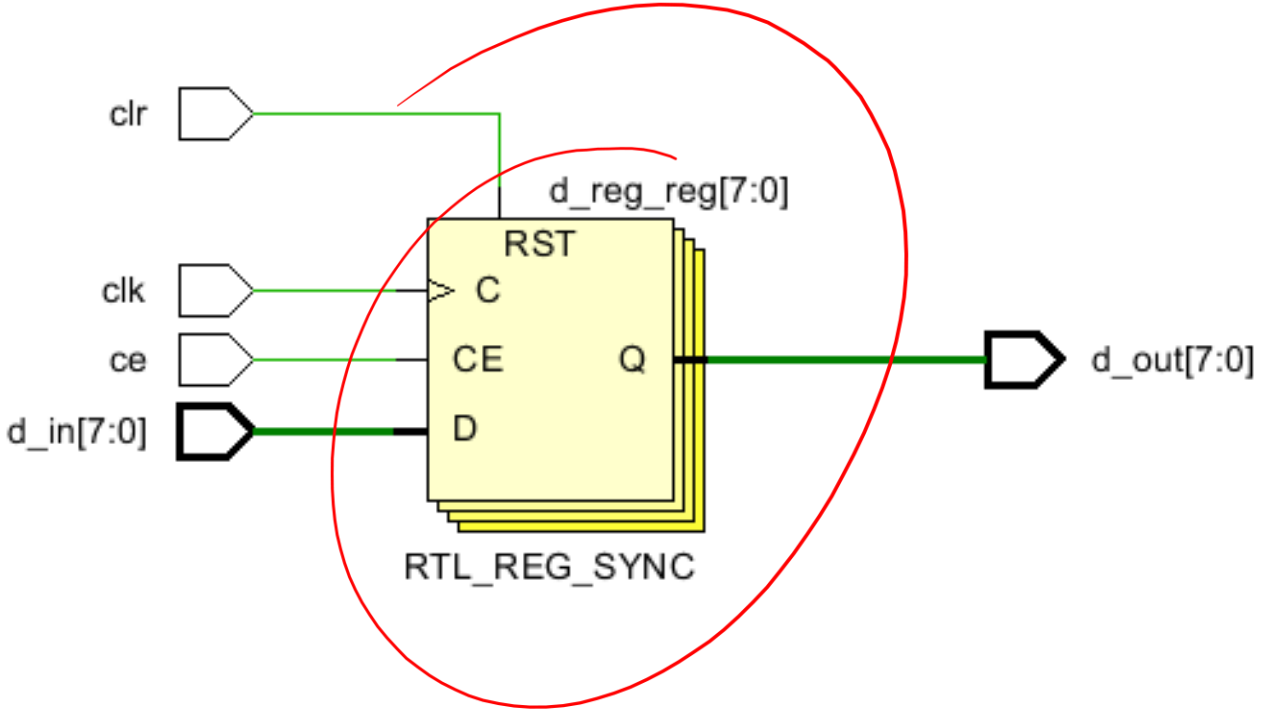


1) Design comb. ckt :- Use always@(*)

2) Design FFs :- Use always@(posedge clk)

Verilog: Register

```
module test_1(
    input [7:0] d_in,
    input ce,
    input clk,
    input clr,
    output [7:0] d_out
    );

    reg [7:0] d_reg;
    always@(posedge clk)
    begin
        if(clr)
            d_reg <= 8'b00000000;
        else if (ce)
            d_reg <= d_in;
    end
    assign d_out = d_reg;
endmodule
```



Register and Wire

Module Ports

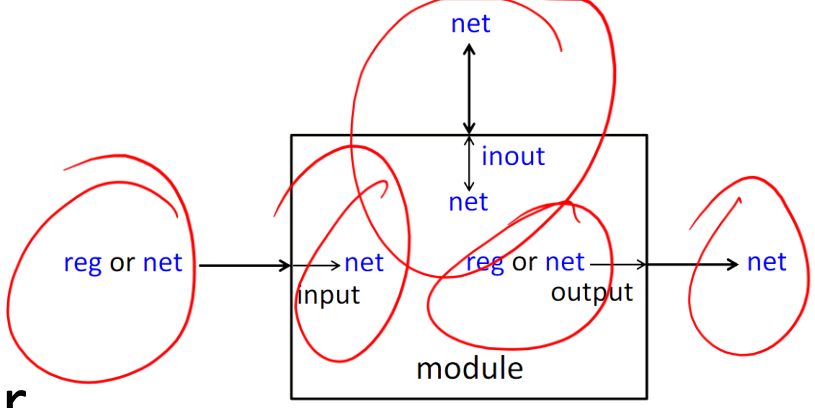
Verilog: Module Ports

reg / wire

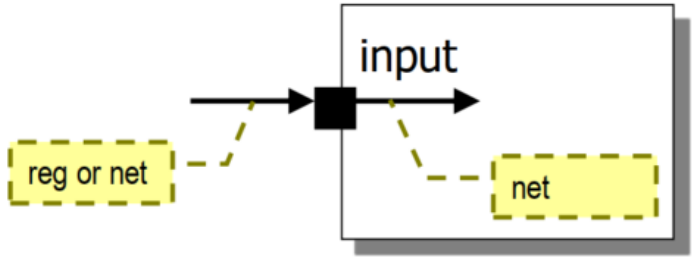
- Ports provide interface for the module to communicate with its environment
- Declaration: <Port direction> <width> <port_name>;
- Port direction can be *input, output, inout*.

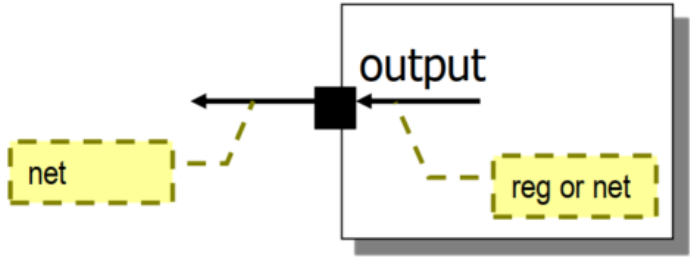
```
module my_module (my_input_port, my_inout_port,
                                  my_output_port ) ;
    input  [4:0]  my_input_port ;
    inout   my_inout_port ;
    output wire (or reg) [14:0] my_output_port ;
endmodule
```

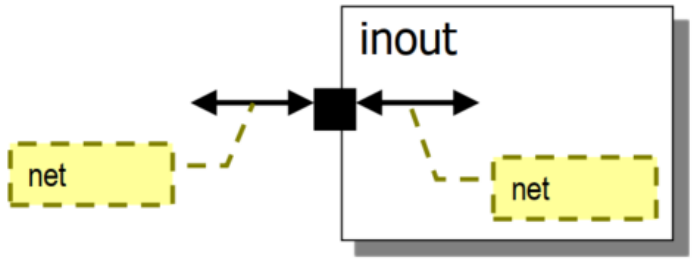
Verilog: Module Ports



- An input port specifies an internal name for a vector or scalar, driven by external entity.
- An output port specifies an internal name for a vector or scalar, driven by internal entity, available external to the module.
- An inout port specifies an internal name for a vector or scalar driven either by an internal or external entity.
- Input or inout ports cannot be declared as of type register.
- Port is always considered as net, unless declared elsewhere as reg (only for output port)



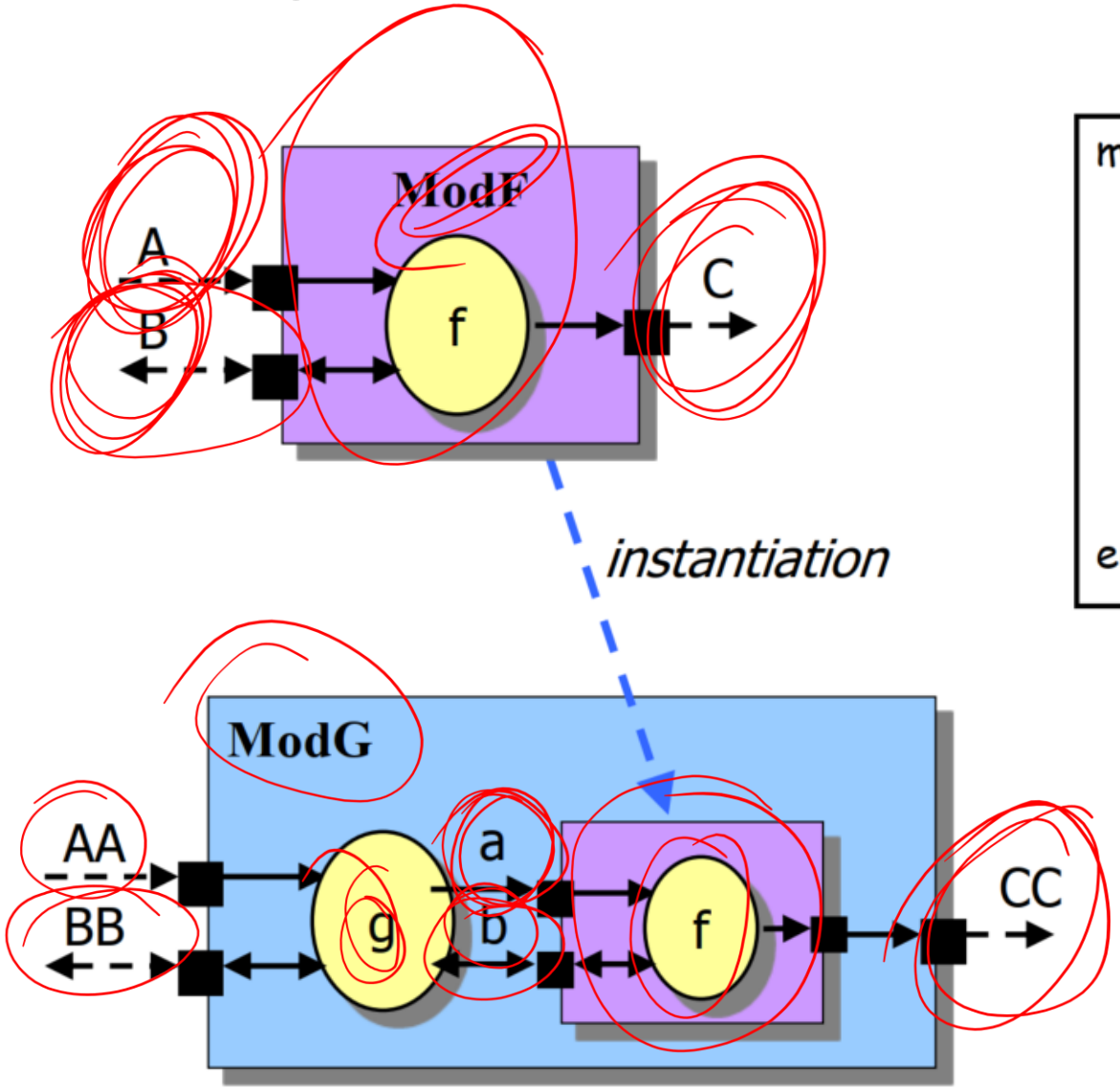




Module Interconnections

Verilog: Module Interconnections

named association




```
module ModG (AA, BB, CC);
   input        AA;
   inout  [7:0] BB;
   output [7:0] CC;
   wire         a;
   wire [7:0]   b;
   // description of 'g'
   ModF Umodgf(.A(a), .B(b), .C(CC));
endmodule
```

Port connection

Instance name

Module name

Verilog: Module Interconnections

- Ports of the instances could be connected by name or by order list.

```
module fa_tb ;
    reg [3:0] A, B ;
    reg CIN ;
    wire [3:0] SUM ;
    wire COUT ;
endmodule
```

```
module FA4 (sum, cout, a, b, cin) ;
    output wire [3:0] sum ;
    output wire cout ;
     input [3:0] a, b ;
     input cin ;
endmodule
```

Order Association

Name

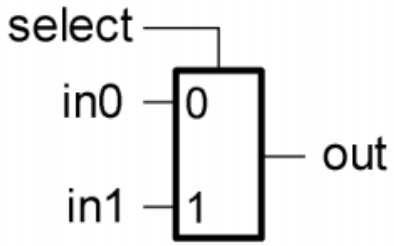
```
// Instantiate/connect by Positional association (order list):
FA4 fa_byorder (SUM, COUT, A, B, CIN) ;

// Instantiate/connect by Named association (port name):
FA4 fa_byname (.cout(COUT), .sum(SUM),.b(B), .cin(CIN), .a(A) ;
```

Verilog: Module Interconnections

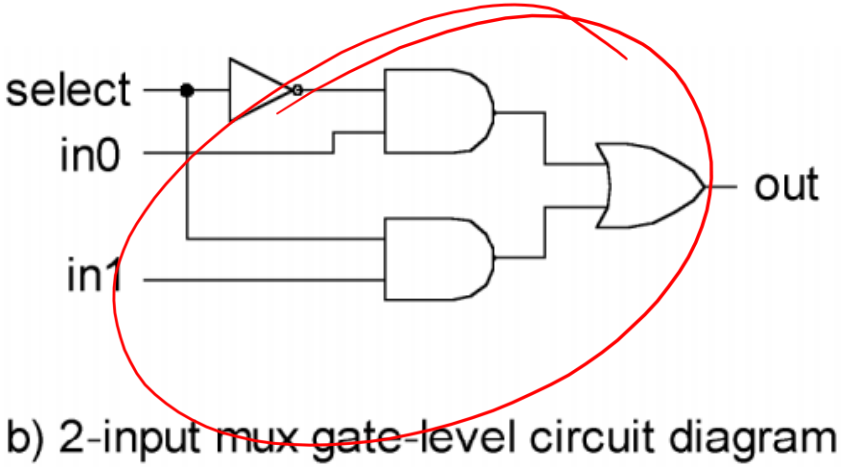
```
module topmod;
   wire [4:0] v;
   wire a,b,c,w;
   modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
   inout wa, wb;
   input c, d;
   tranif1 g1 (wa, wb, cinvert);
   not #(2, 6) n1 (cinvert, int);
   and #(6, 5) g2 (int, c, d);
endmodule
```

```
module topmod;
   wire [4:0] v;
   wire a,b,c,w;
   modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
   endmodule

module modB (wa, wb, c, d);
   inout wa, wb;
   input c, d;
   tranif1 g1(wa, wb, cinvert);
   not #(6, 2) n1(cinvert, int);
   and #(5, 6) g2(int, c, d);
endmodule
```

Multiplexer



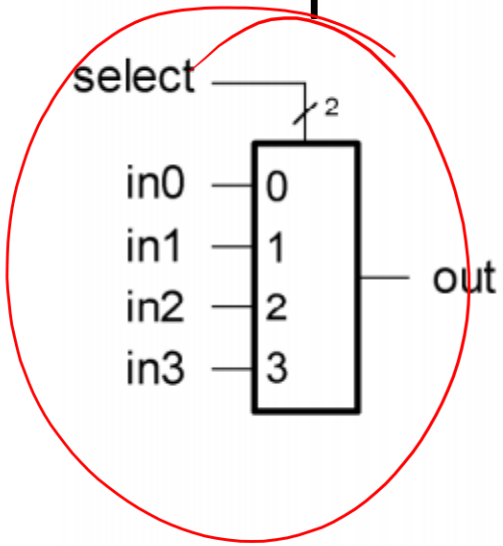
a) 2-input mux symbol

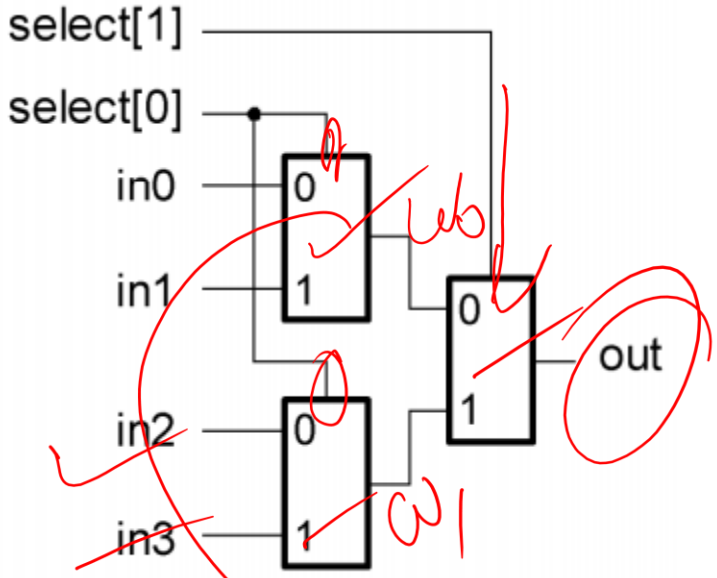


b) 2-input mux gate-level circuit diagram

```
module mux2 (in0, in1, select, out);
    input in0, in1, select;
    output out;
    wire s0, w0,w1;

    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or (out, w0,w1);

endmodule
```

Multiplexer






```
module mux2 (in0, in1, select, out);
   input in0, in1, select;
   output out;
   wire s0, w0,w1;

   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or (out, w0,w1);

endmodule
```

```
module mux4 (in0, in1, in2, in3, select, out);
   input in0, in1, in2, in3;
   input [1:0] select;
   output out;
   wire w0,w1;

   mux2
      mo (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
      m2 (.select(select[0]), .in0(in2), .in1(in3), .out(w1));
      mo (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule
```

If your design contains more than one module, put each in a separate file.

Instantiation: Prefer named association. It prevents incorrect connections for the ports of instantiated components.

3-bit Full Adder

```
module top_adder(
    input [2:0] A,
    input [2:0] B,
    output [3:0] Sum
    );
    wire c1,c2;

    fa in1(.A(A[0]),.B(B[0]),.C(1'b0),.Sum(Sum[0]),.Carry(c1));
    fa in2(.A(A[1]),.B(B[1]),.C(c1),.Sum(Sum[1]),.Carry(c2));
    fa in3(.A(A[2]),.B(B[2]),.C(c2),.Sum(Sum[2]),.Carry(Sum[3]));
endmodule
```

```
module fa(
    input A,
    input B,
    input C,
    output Sum,
    output Carry
    );

    assign Sum = A^B^C;
    assign Carry=((A^B)&C)|(A&B);
endmodule
```

3-bit Multiplier (Self Study)

```
module top_multiplier(
    input [2:0] A,
    input [2:0] B,
    output [5:0] Mul_Op
    );

    wire c1,c2,c3,c22,c32;
    wire s1,s2;
    assign Mul_Op[0]=A[0] & B[0];
    fa in1(.A(A[0] & B[1]),.B(A[1] & B[0]),.C(1'b0),.Sum(Mul_Op[1]),.Carry(c1));
    fa in2(.A(A[2] & B[0]),.B(A[1] & B[1]),.C(c1),.Sum(s1),.Carry(c2));
    fa in3(.A(A[0] & B[2]),.B(s1),.C(1'b0),.Sum(Mul_Op[2]),.Carry(c22));
    fa in4(.A(A[2] & B[1]),.B(1'b0),.C(c2),.Sum(s2),.Carry(c3));
    fa in5(.A(A[1] & B[2]),.B(s2),.C(c22),.Sum(Mul_Op[3]),.Carry(c32));
    fa in6(.A(A[2] & B[2]),.B(c3),.C(c32),.Sum(Mul_Op[4]),.Carry(Mul_Op[5]));
endmodule
```

```
module fa(
    input A,
    input B,
    input C,
    output Sum,
    output Carry
    );

    assign Sum = A^B^C;
    assign Carry=((A^B)&C)|(A&B);
endmodule
```

Homework

- Using module for 2:1 mux (data flow level approach), design 8:1 mux via module interconnections
- Design comparator for 2-bit inputs using data flow level approach.
- Using comparator for 1-bit inputs, design comparator for 2-bit inputs via module interconnections

Number Representation

Verilog: Number Representation

- Verilog HDL allows integer numbers to be specified as: Sized or Unsized numbers (Unsized is 32 bits)
- In a radix of binary, octal, decimal, or hexadecimal
- Syntax: **<size> '<radix> <value>**
  size in bits, radix in b, d, o, h
- Spaces are allowed between the size, radix and value
- Underscore character (_) is ignored and can be used to enhance readability. It cannot be the first character in number.

Verilog: Number Representation

```
549       // unsized decimal number
'h 8FF    // unsized hex number
'o765     // unsized octal number
4 'b11    // 4-bit binary number 0011
3 'b10x   // 3-bit binary number with LSB unknown
5 'd3     // 5-bit decimal number
```

32'd549
32'h8FF
32'o765
4'b0011
5'd3
⇓
00011

Verilog: Number Representation

```
792          // a decimal number
8d9          // Illegal, hexadecimal must be specified with ‘h
‘h 7d9       // an unsized hexadecimal number - 000007d9
‘o 7746      // an unsized octal number - 00000007746
1            // stored as 00000000000000000000000000000001

12 ‘h x      // a 12 bit unknown number
10 ‘d 17     // a 10 bit constant with the value 17
4 ‘b 110z    // a 4 bit binary number
8’hAA        // stored as 10101010
```

Verilog: Negative Numbers

- Any number that does not have negative sign prefix is a positive number. Or indirect way would be "Unsigned"
- Negative numbers can be specified by putting a minus sign before the size for a constant number, thus become signed numbers.
- Verilog internally represents negative numbers in 2's compliment format.

```
-4 'b11   // 4-bit two's complement of 0011 = 1101 = 4'hd

4'd-2     // Illegal specification
-5'ha     // stored as 10110

-4'b101   // stored as 1011
```

Verilog: Number Representation

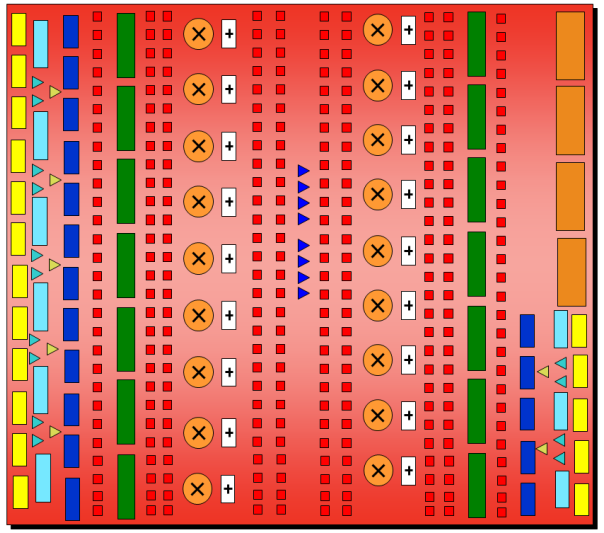
number	stored value	comment
5'b11010	11010	
5'b11_010	11010	_ ignored
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	00000000000000000000000000011010	extended to 32 bits
'hee	00000000000000000000000011101110	extended to 32 bits
1	00000000000000000000000000000001	extended to 32 bits
-1	11111111111111111111111111111111	extended to 32 bits

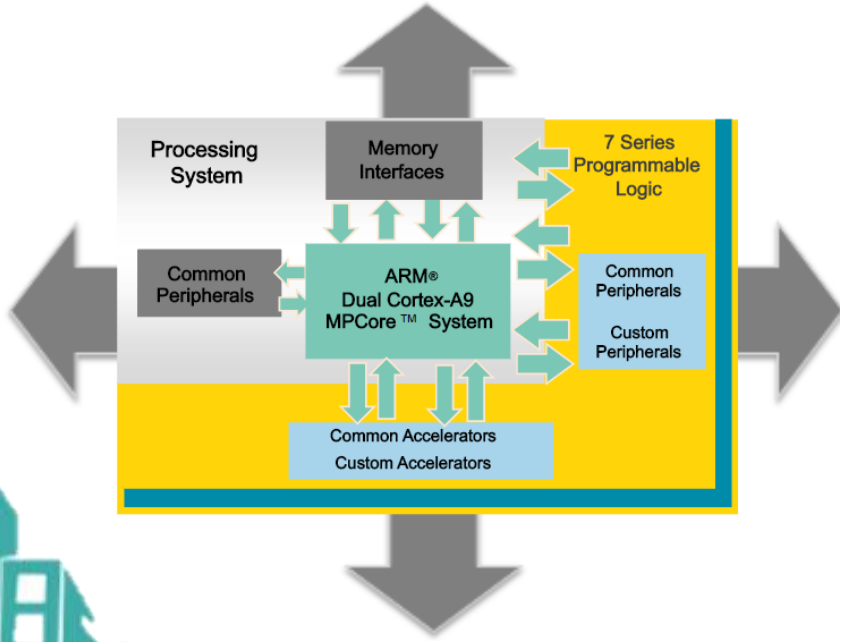
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

SDK
Software Development Kit
VIVADO™





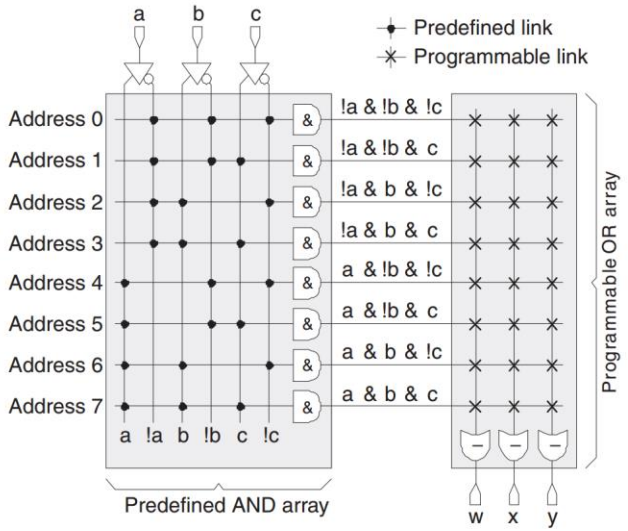
Evolution of Programmable Logic Device (PLD)

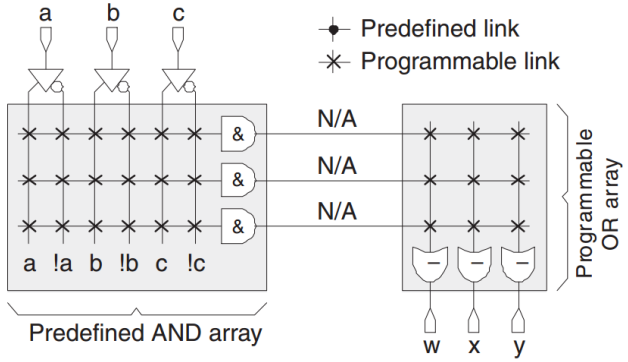
PLD

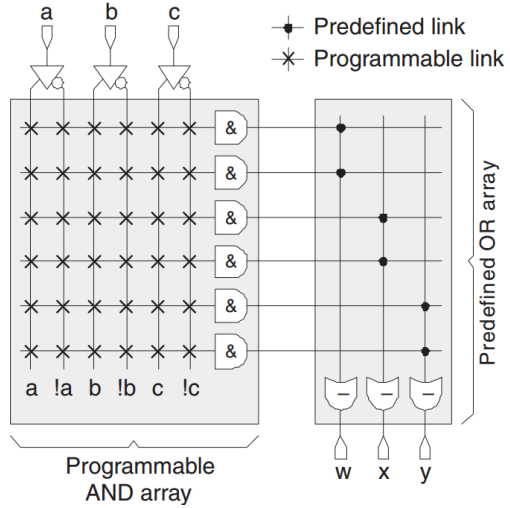
- **Programmable logic devices (PLD)**: Devices whose internal architecture is predetermined by manufacturer but which are created in such a way that they can be configured in the field to perform variety of functions
- How to make device field programmable?

PLD

- **Programmable logic devices (PLD)**: Devices whose internal architecture is predetermined by manufacturer but which are created in such a way that they can be configured in the field to perform variety of functions

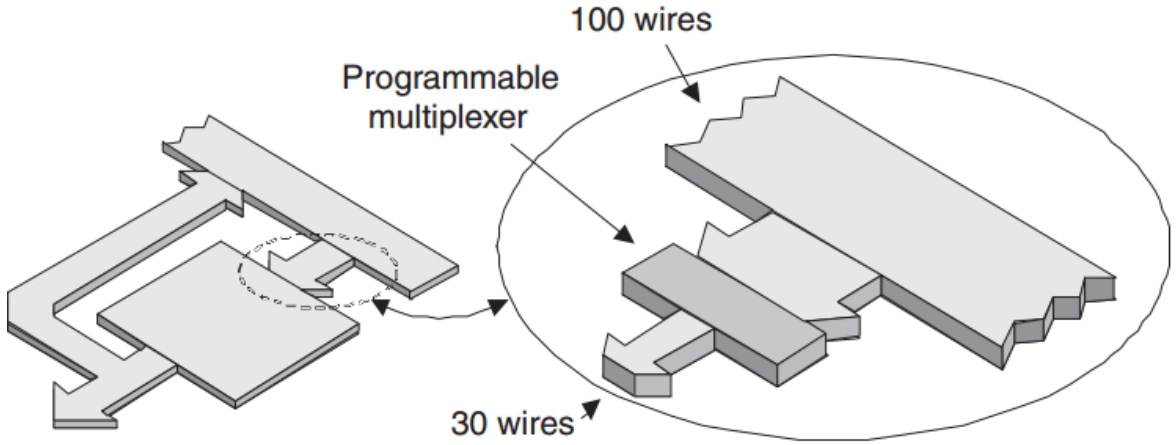


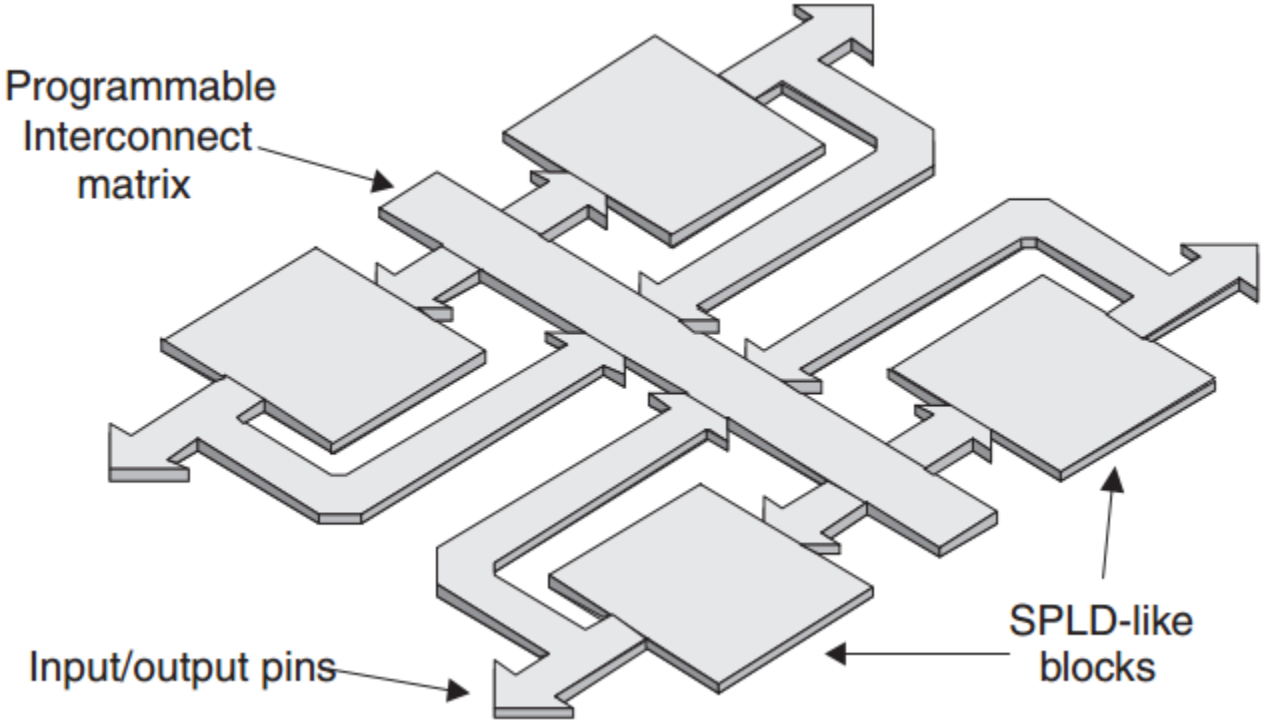




PAL: Complex PLDs

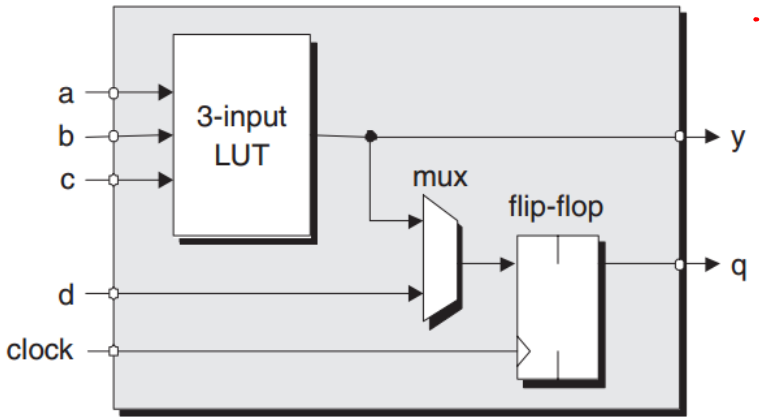
- Novelty: Central interconnect
- In addition to programming SPLD (PAL), connections can also be programmed using programmable interconnect matrix
- This leads to increase in the complexity of software tools

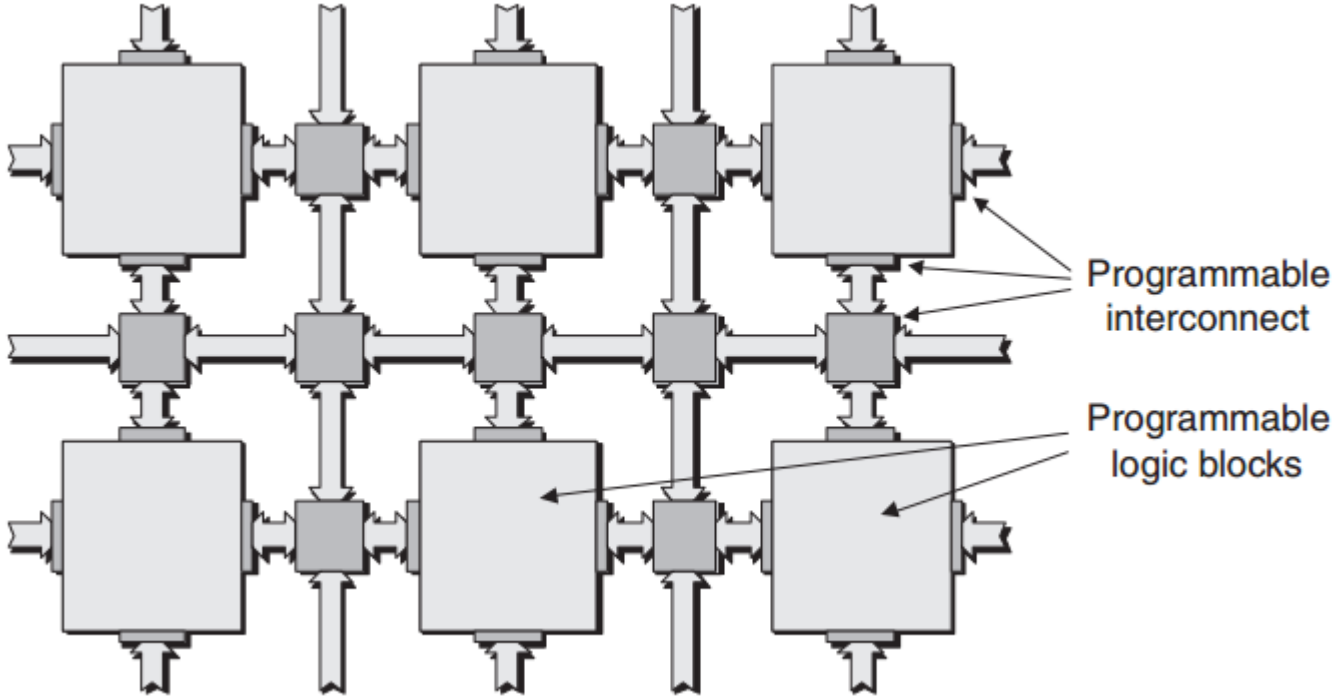




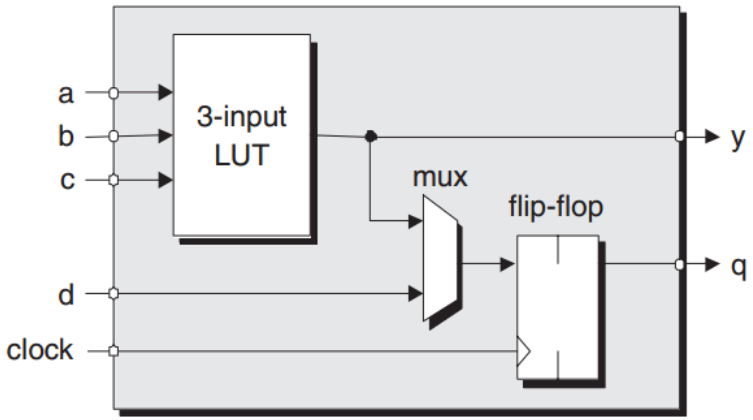
FPGA Architecture

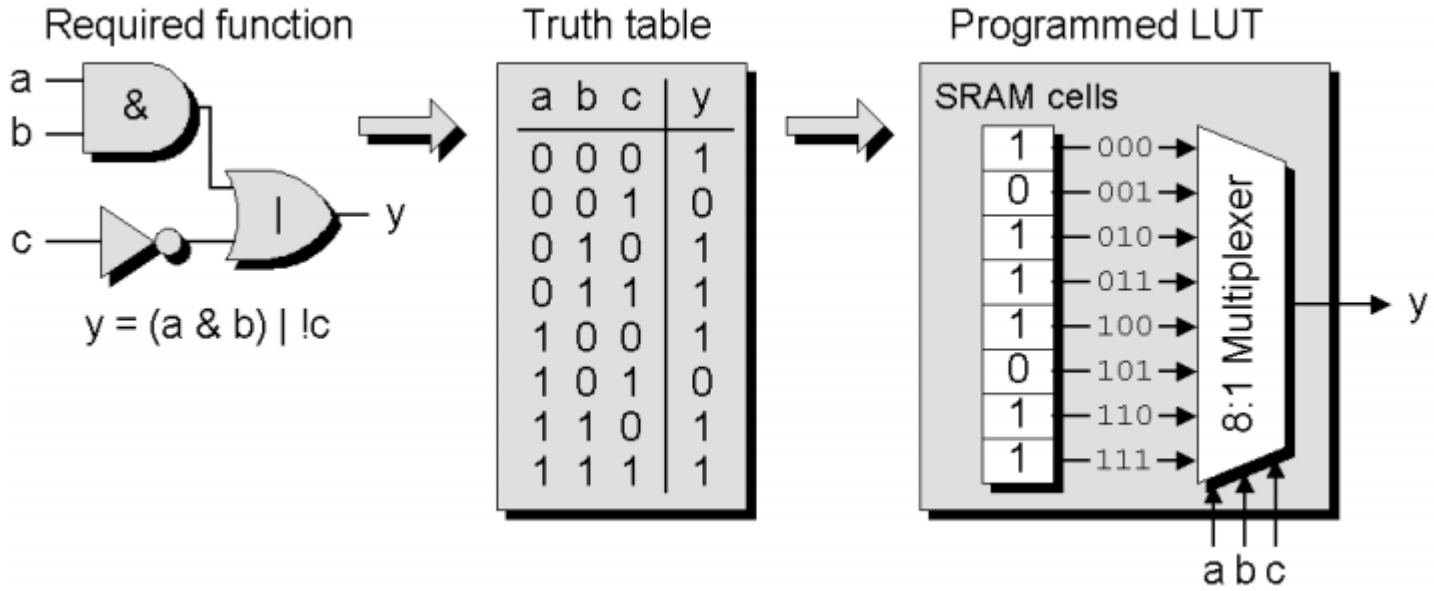
FPGA (1984)


a
b
c
d
clock
3-input
LUT
mux
flip-flop
y
q


Programmable
interconnect
Programmable
logic blocks

FPGA (1984)

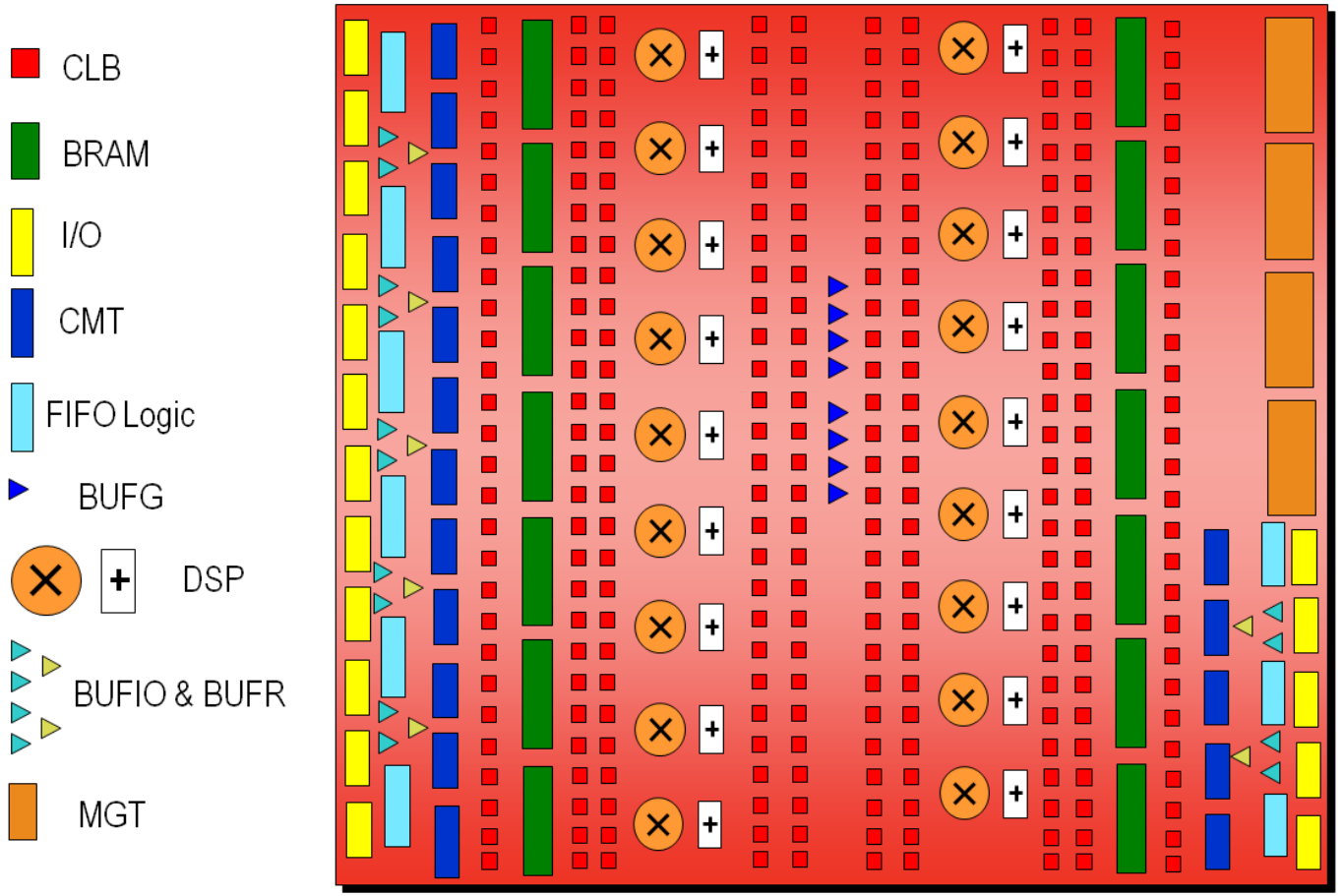

a
b
c
3-input
LUT
mux
flip-flop
y
d
q
clock


Required function
a
b
&
c
|
y
y = (a & b) | !c
Truth table
a b c | y
0 0 0 | 1
0 0 1 | 0
0 1 0 | 1
0 1 1 | 1
1 0 0 | 1
1 0 1 | 0
1 1 0 | 1
1 1 1 | 1
Programmed LUT
SRAM cells
1
0
1
1
1
0
1
1
000
001
010
011
100
101
110
111
8:1 Multiplexer
y
a b c

LUT as Memory and LUT as ALU

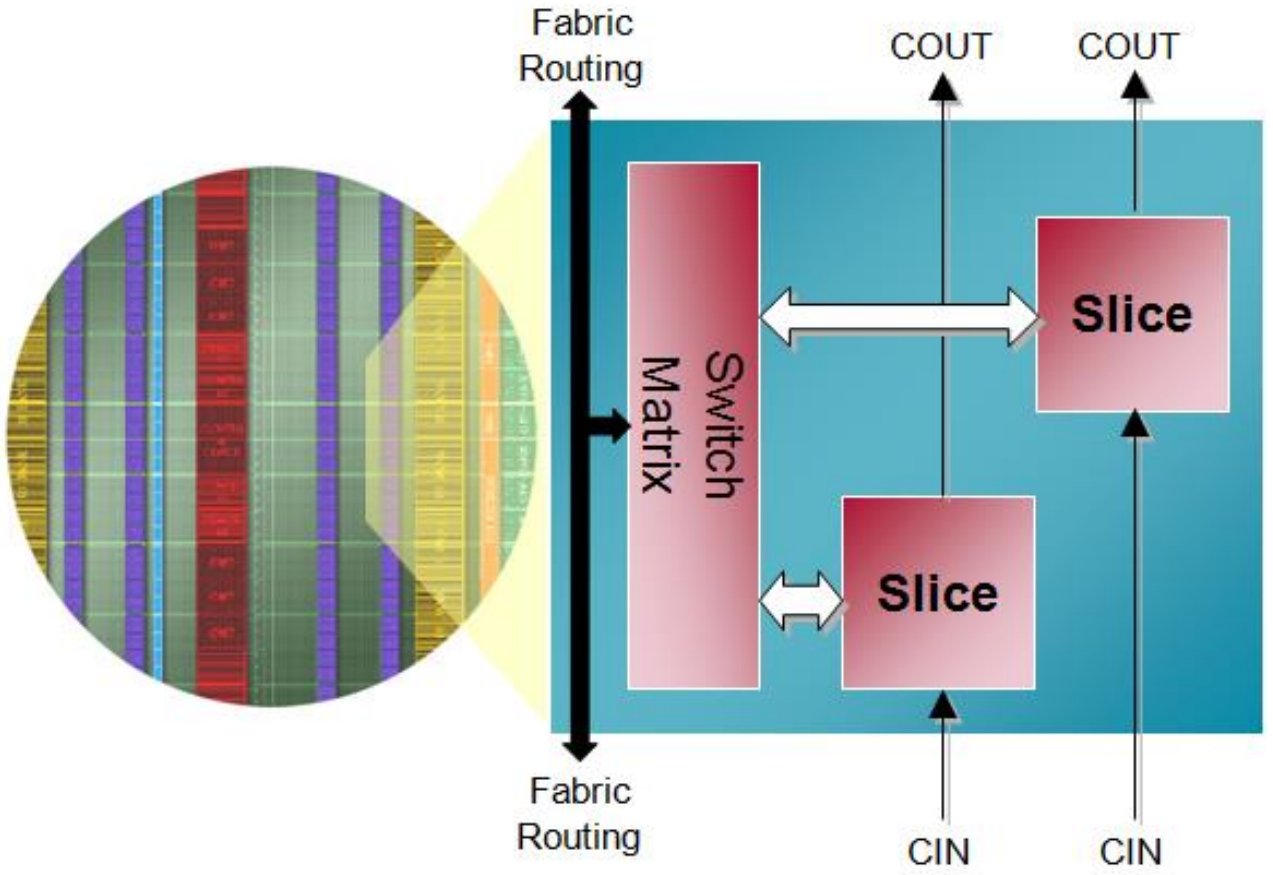
FPGA Architecture

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

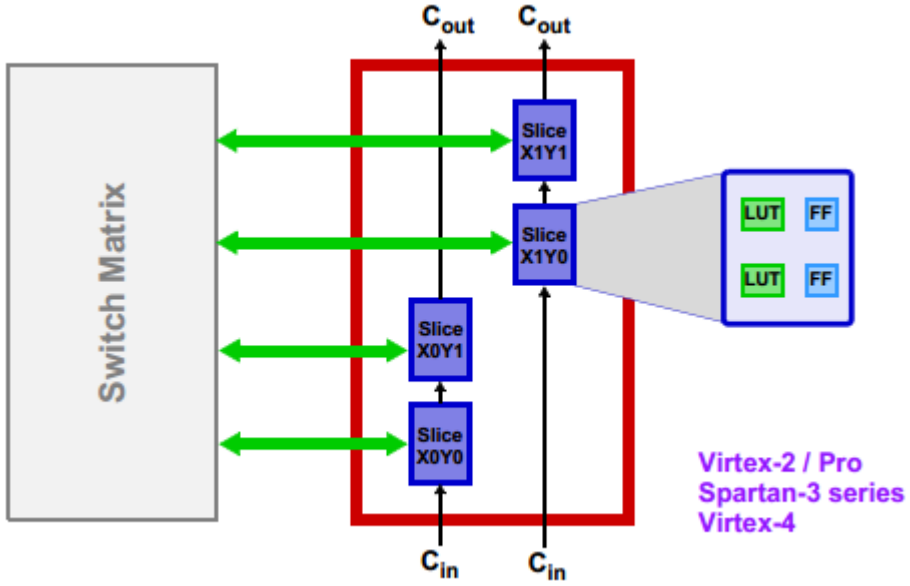


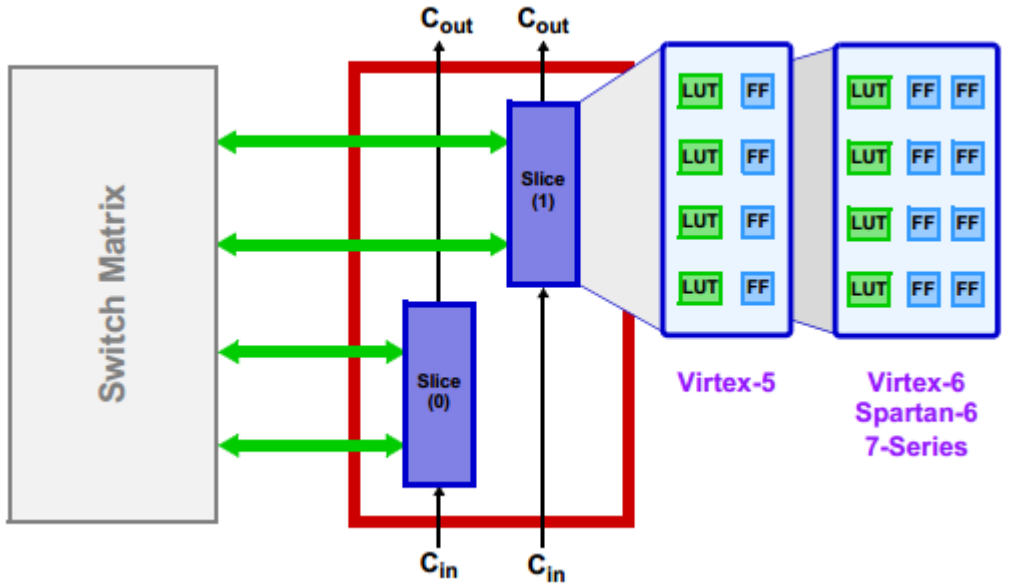
Configurable Logic Block (CLB)

- 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)

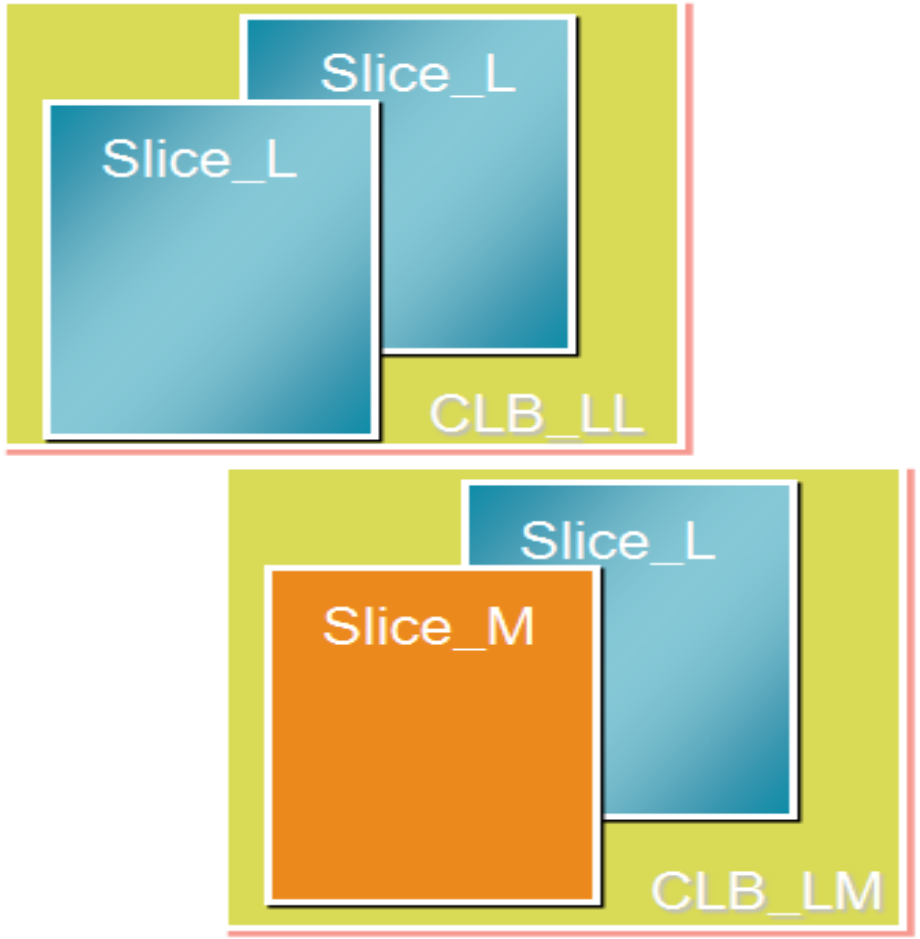




Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains
2	8	16	2

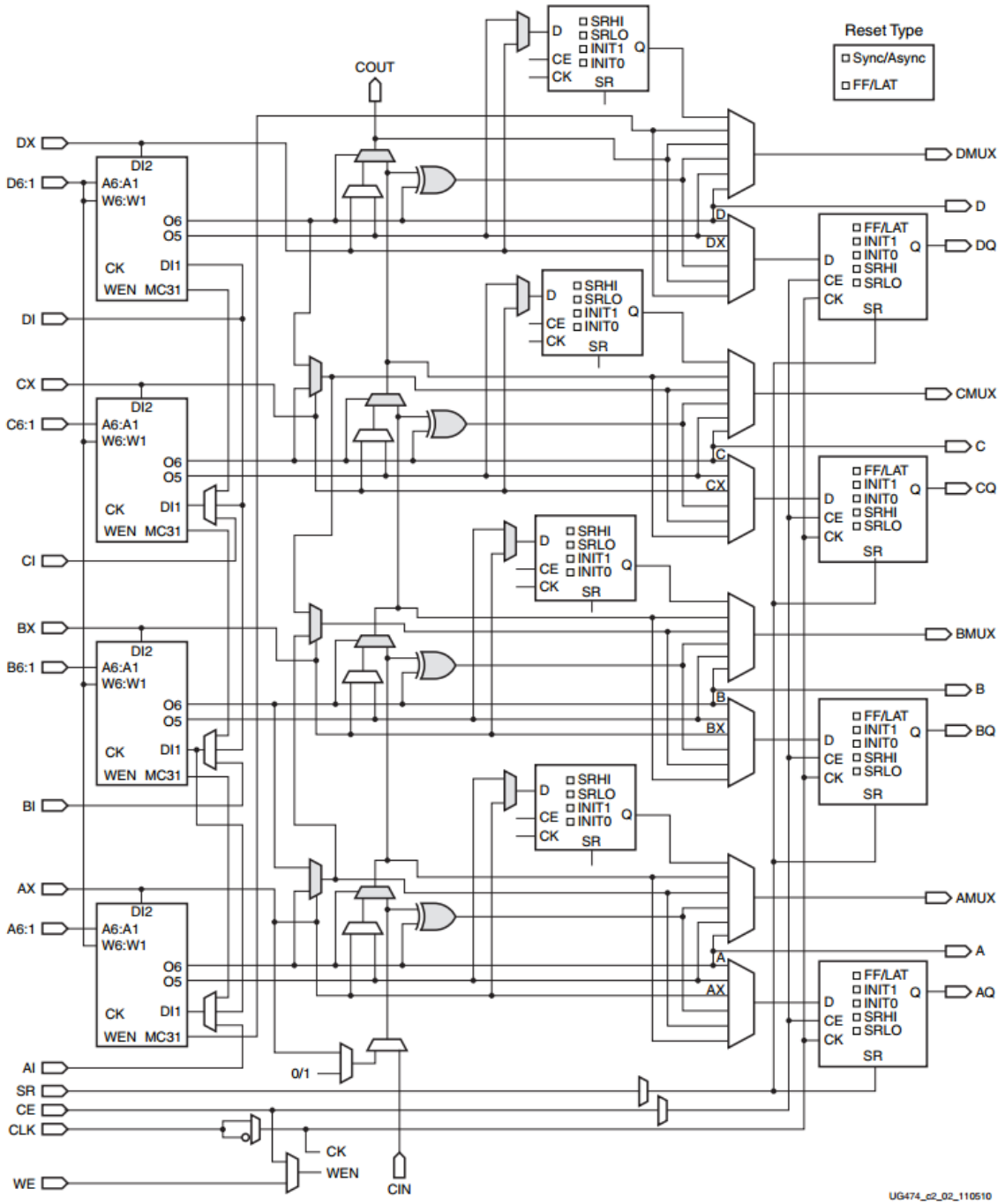
Types of CLB Slices

- **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.



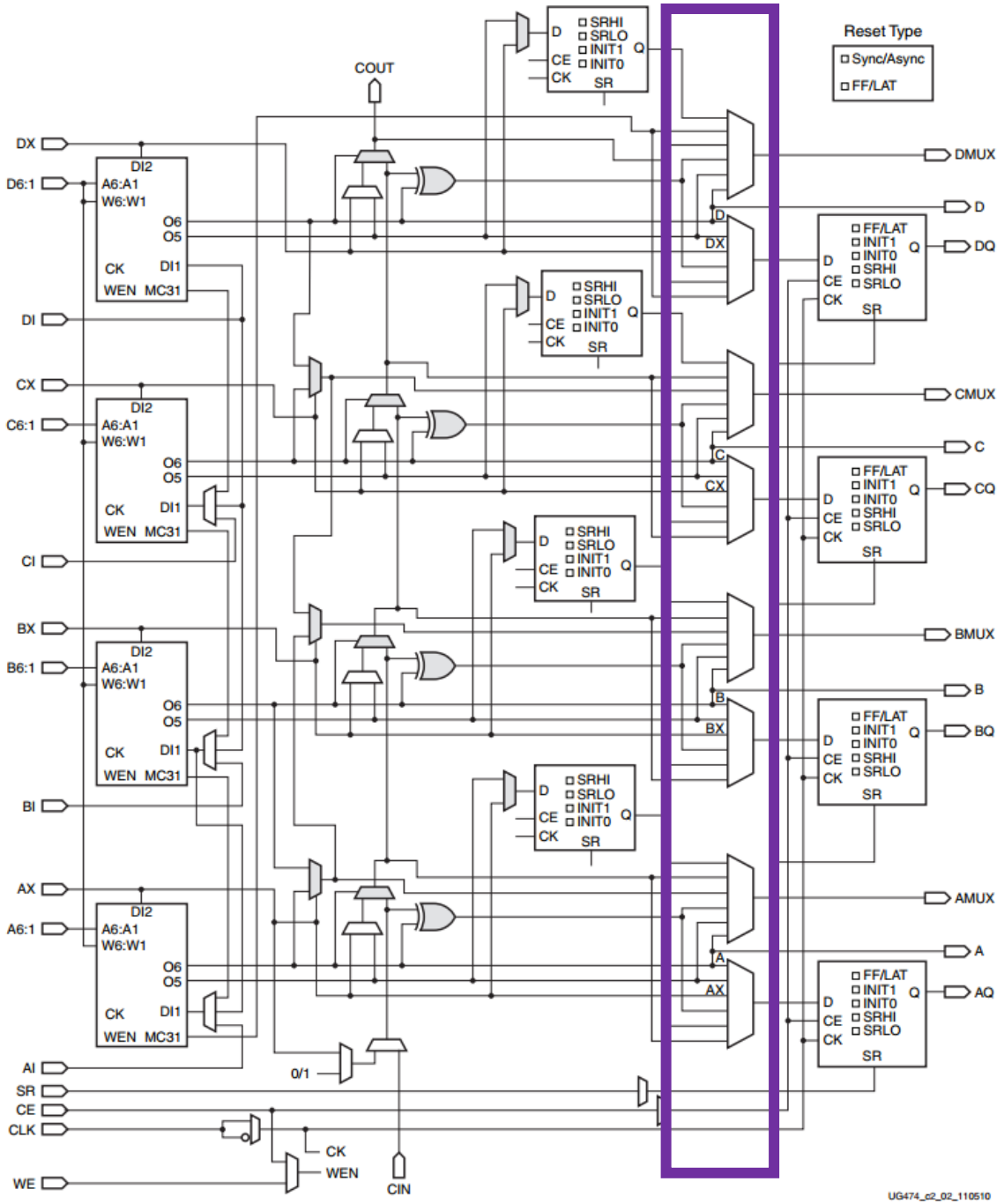
Slice Resource

- 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.



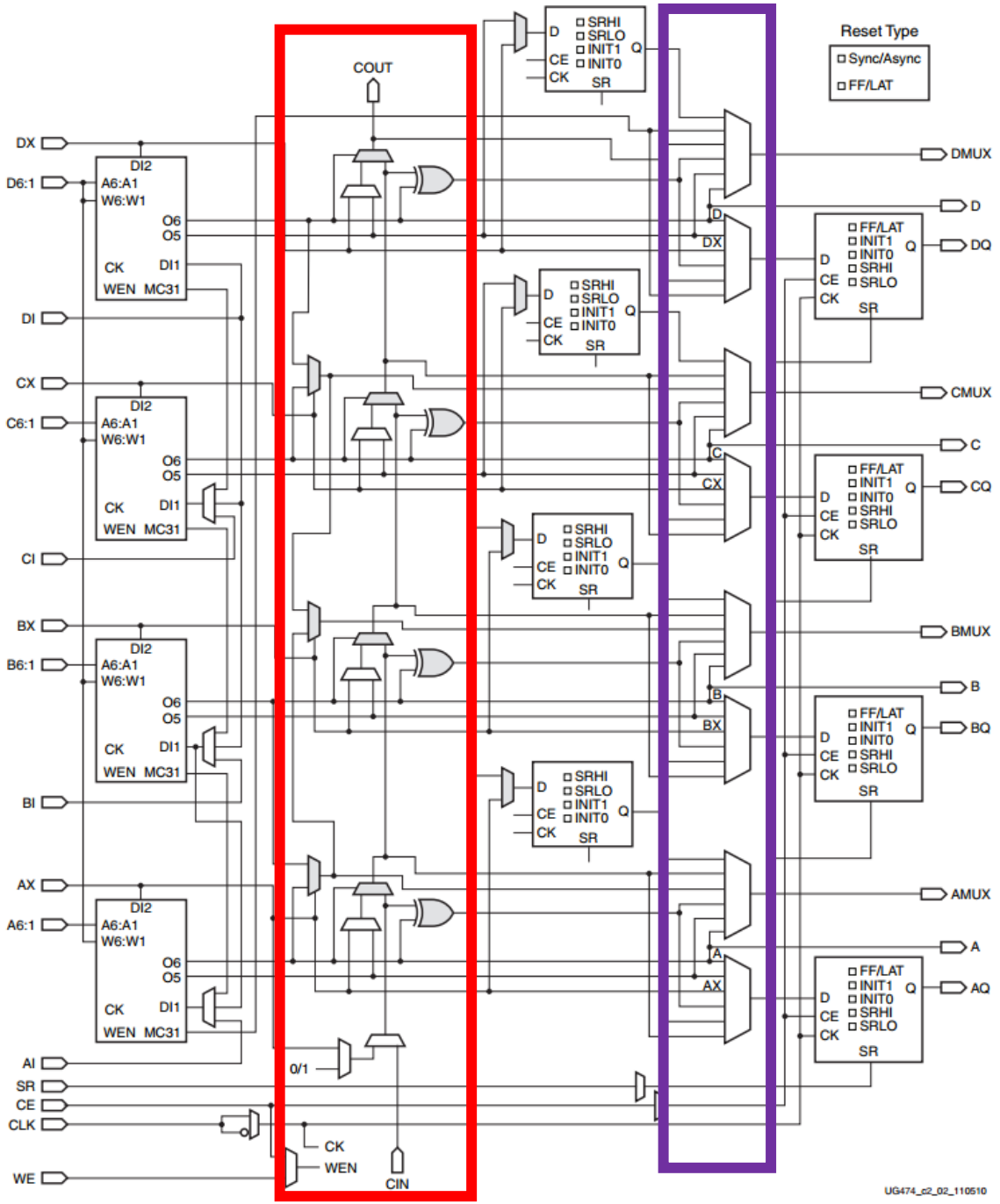
Slice Resource

- 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.



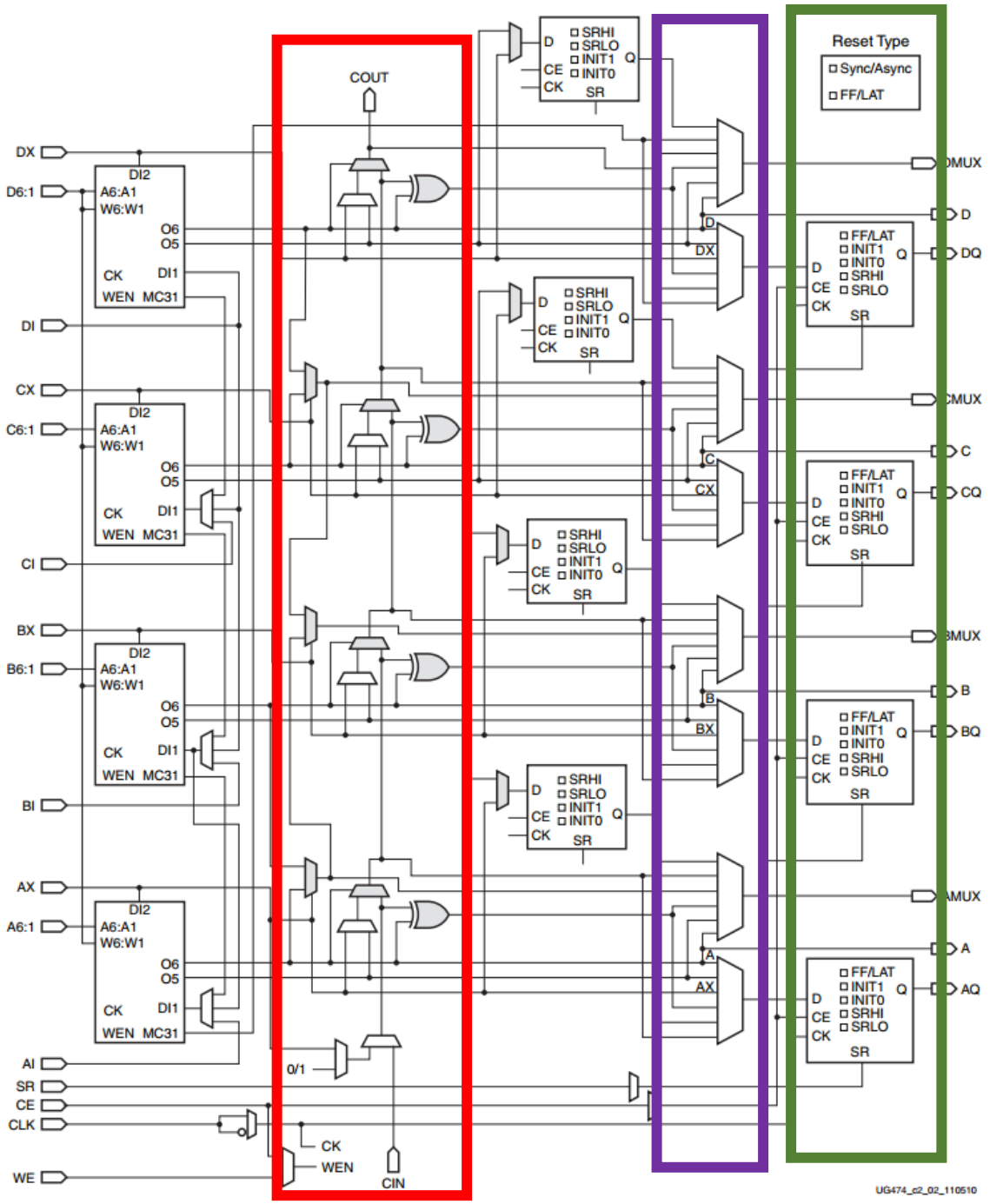
Slice Resource

- 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.



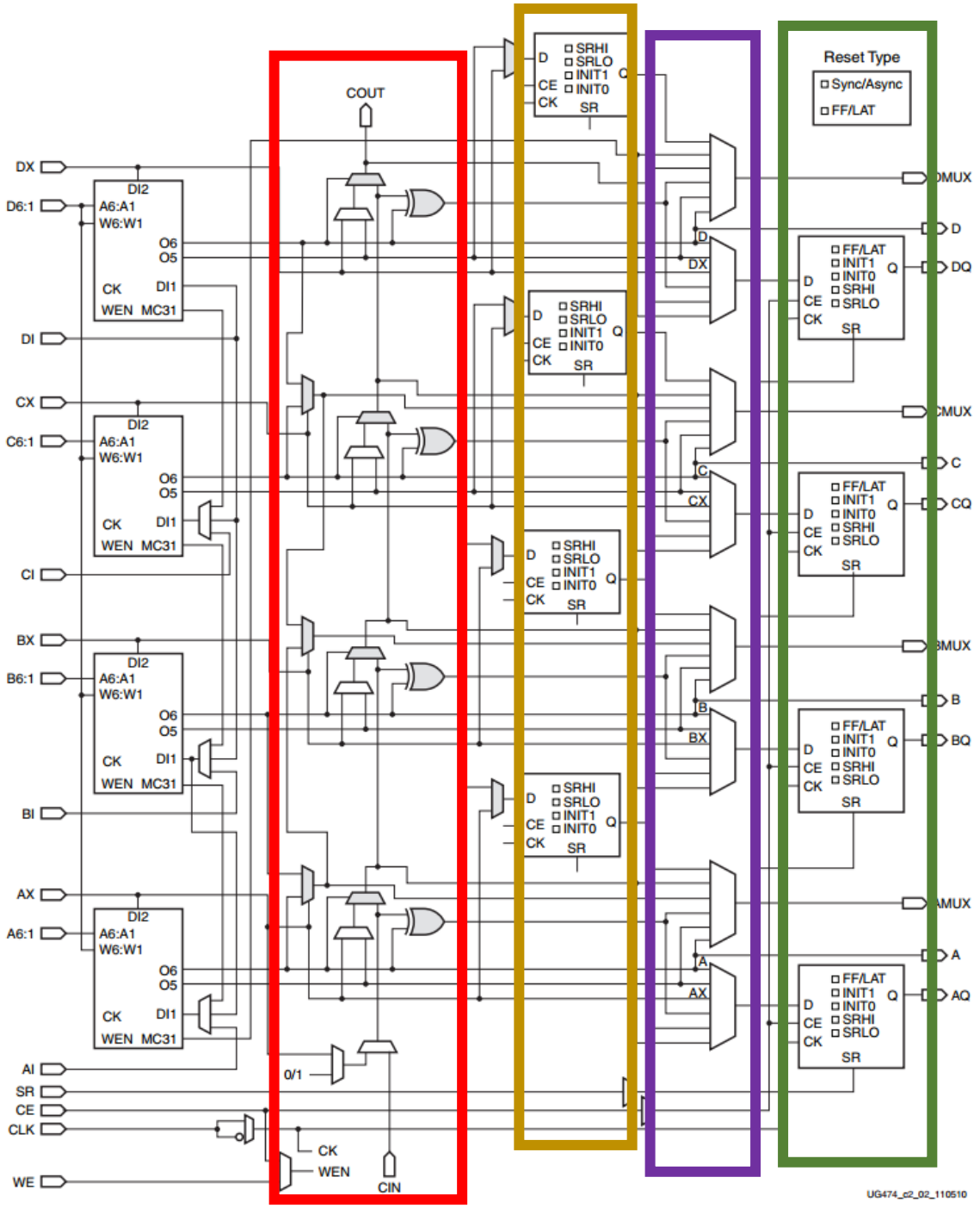
Slice Resource

- 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.



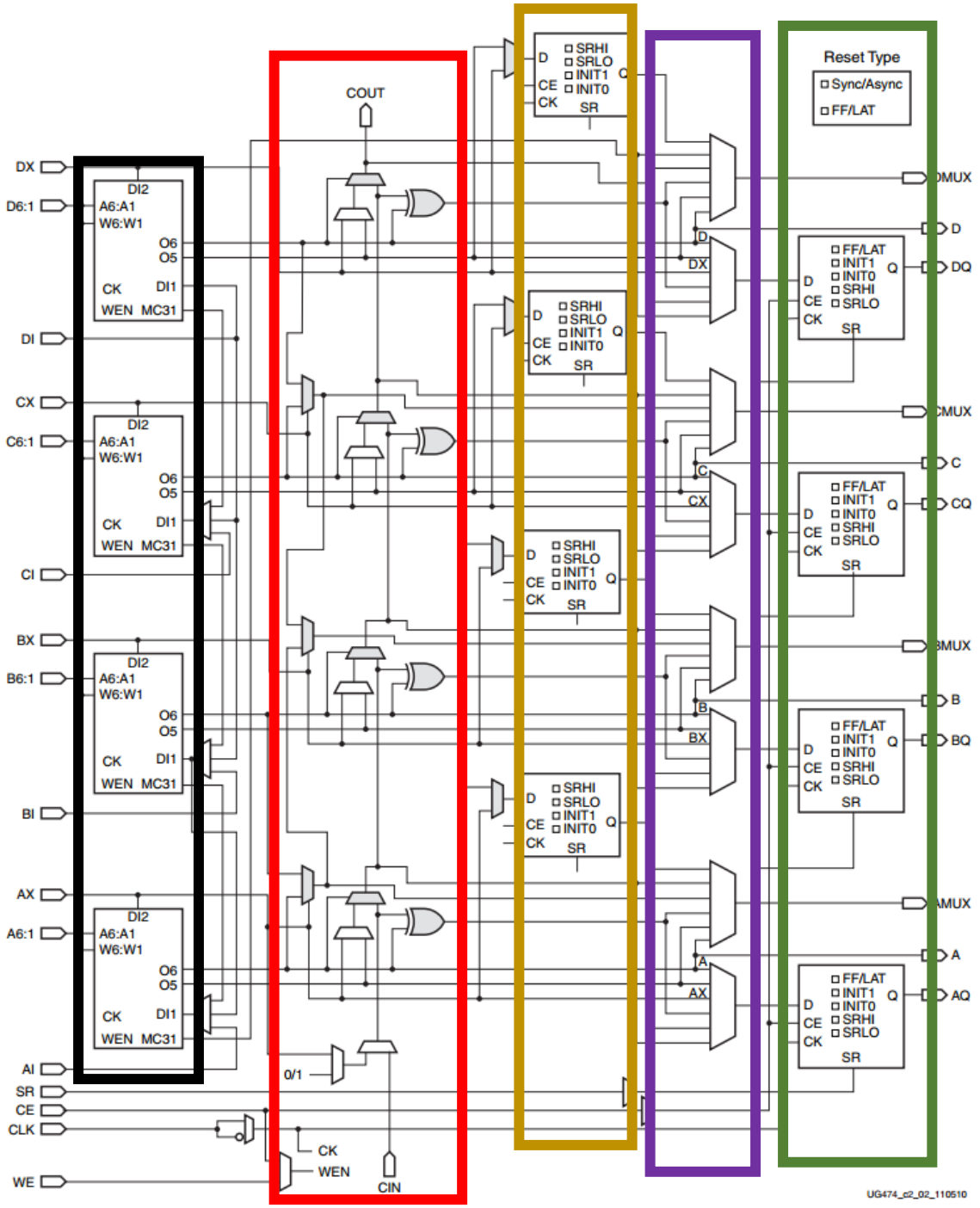
Slice Resource

- 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.

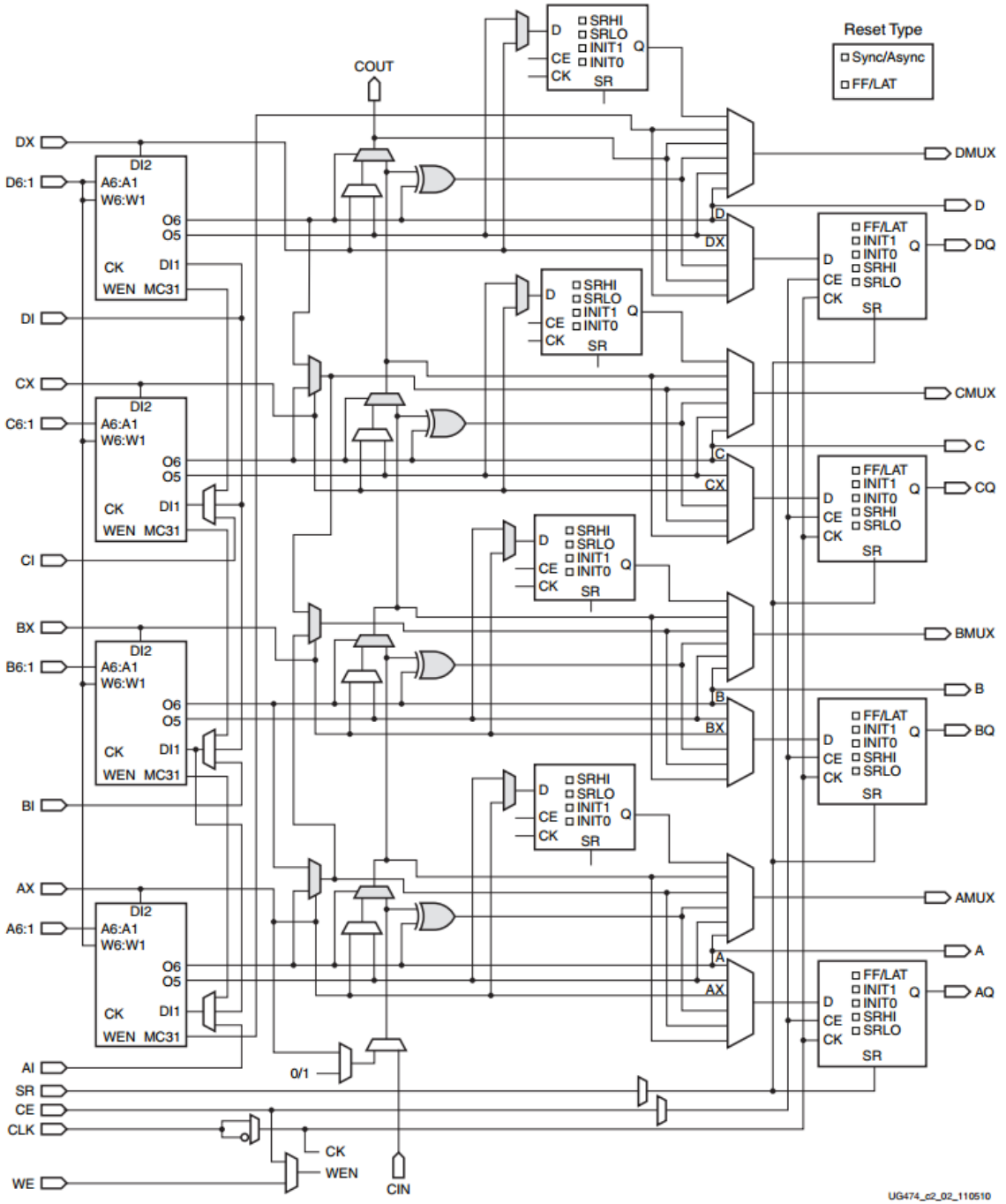


Slice Resource

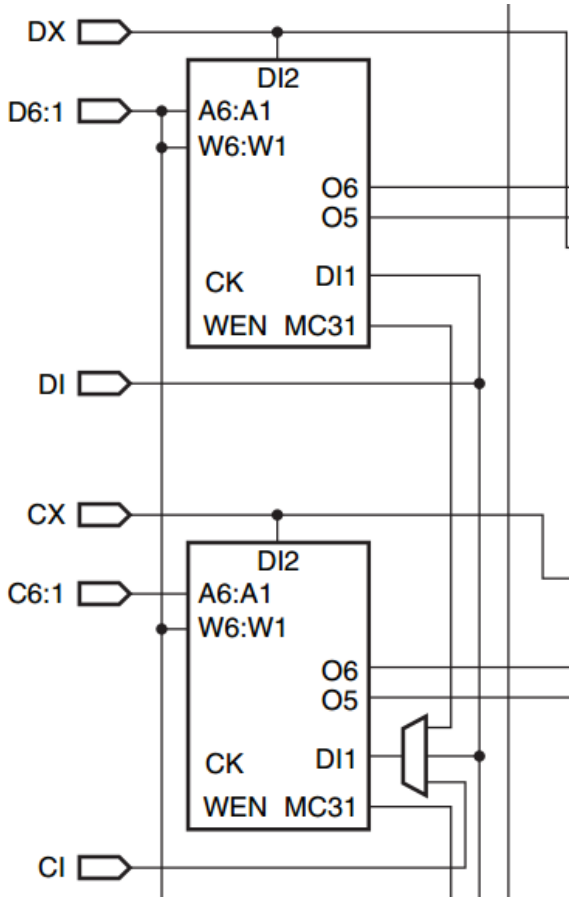
- Four six-input Look-Up Tables (LUT)
- Multiplexers
- Carry chains
- Four flip-flops/latches
- Four additional flip-flops
- Four 6-input LUTs and their eight flip-flops as well as multiplexers and arithmetic carry logic form a slice, and two slices form a CLB.

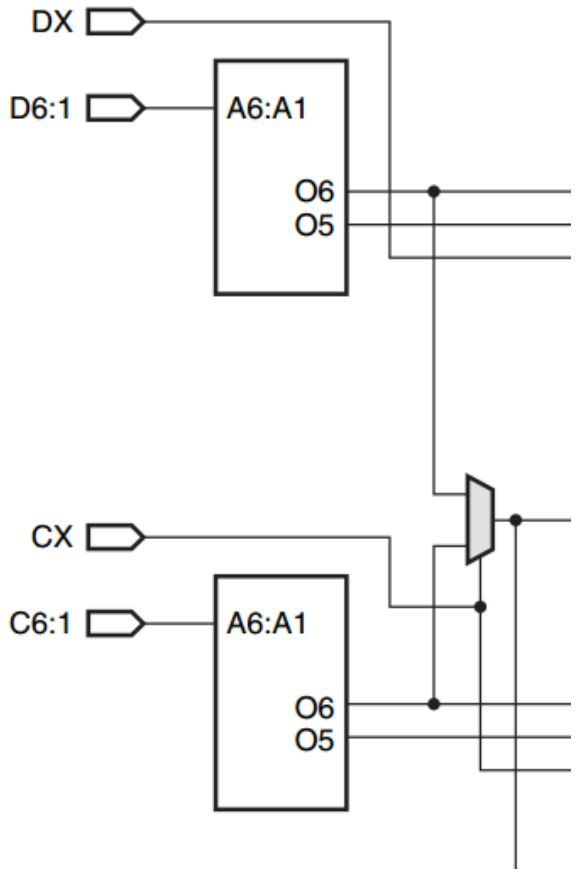


SLICEM

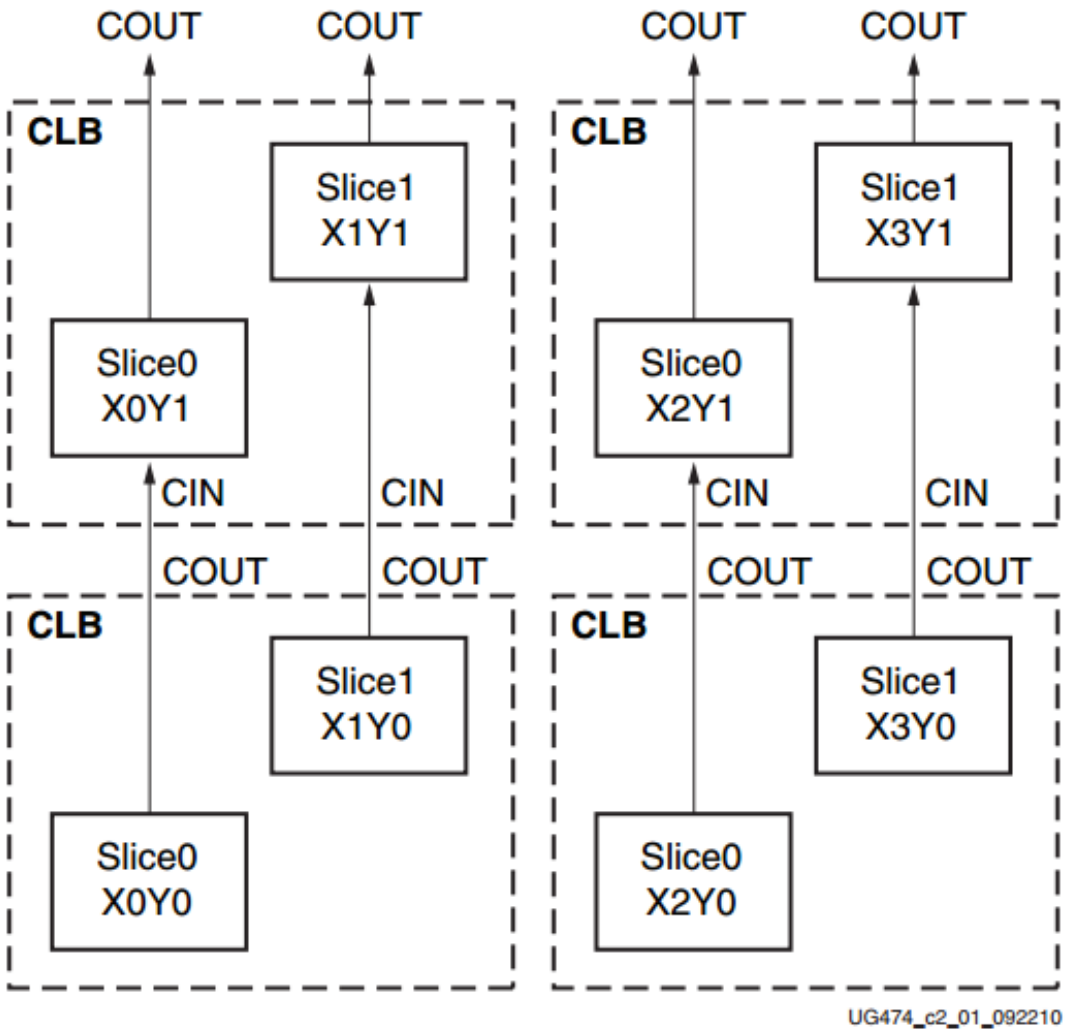

Reset Type
Sync/Async
FF/LAT
COUT
DX
D6:1
DI
CX
C6:1
CI
BX
B6:1
BI
AX
A6:1
AI
SR
CE
CLK
WE
CIN
DI2
A6:A1
W6:W1
O6
O5
CK
DI1
WEN
MC31
SRHI
SRLO
INIT1
INIT0
Q
D
CE
SR
FF/LAT
DMUX
DQ
CMUX
CQ
BMUX
BQ
AMUX
AQ
0/1
WEN
UG474_c2_02_110510

SLICEM Vs SLICEL



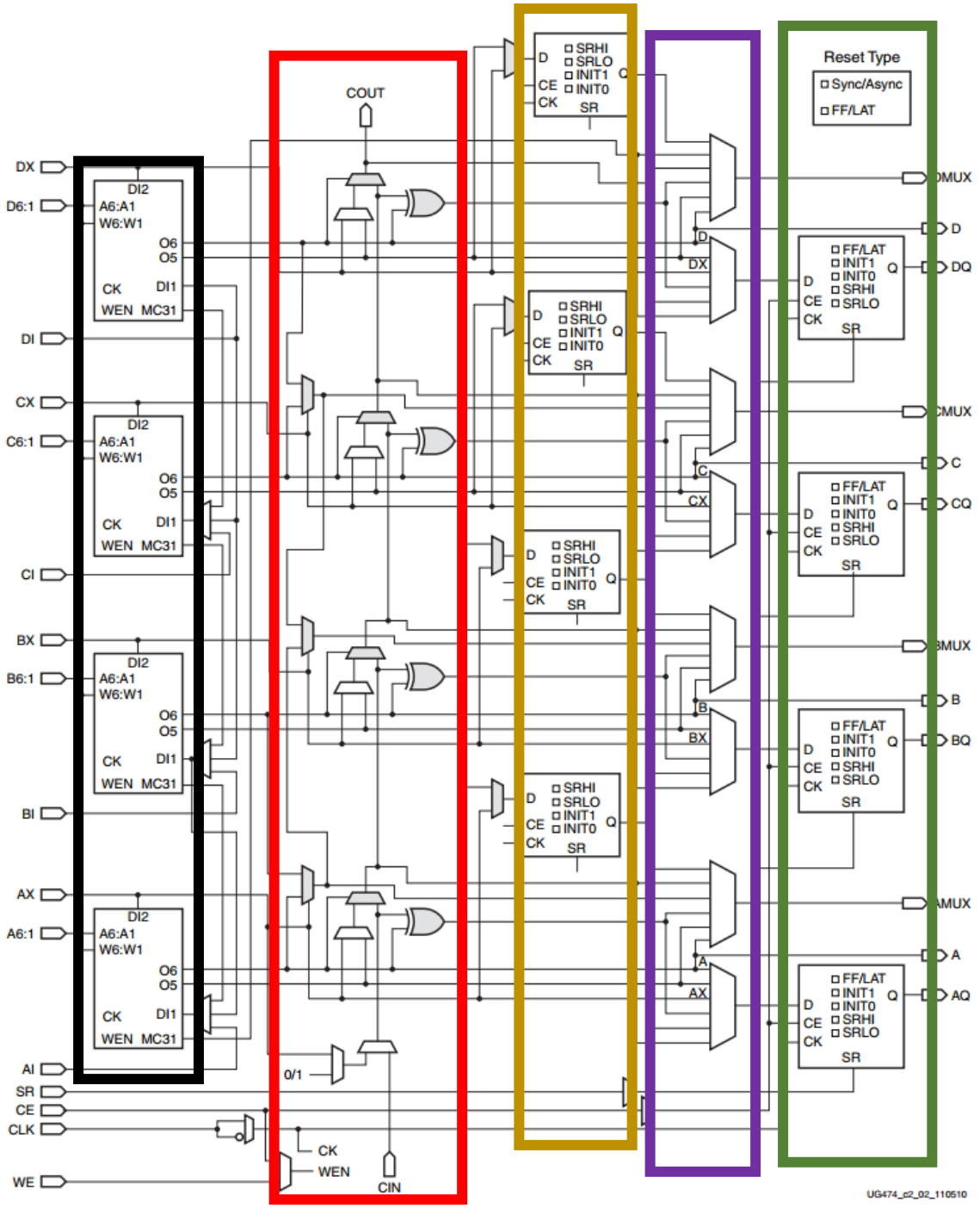


Configurable Logic Block (CLB)



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 ;
```

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

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

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
```

Verilog: Memory

- Registers and memories can be declared in the same line
- **reg** [3:0] mem[255:0], red;
- This line a 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-dimentional 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

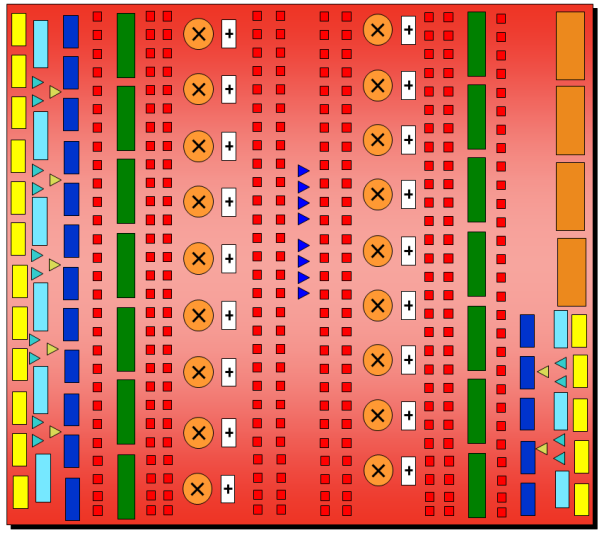
```
mema = 0;       // Illegal syntax- Attempt to write to entire array
arrayb[1] = 0;  // Illegal Syntax - Attempt to write to elements [1][0]..[1][255]
arrayb[1][12:31] = 0; // Illegal Syntax - Attempt to write to elements [1][12]..[1][31]
mema[1] = 0;      // Assigns 0 to the second element of mema
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
```

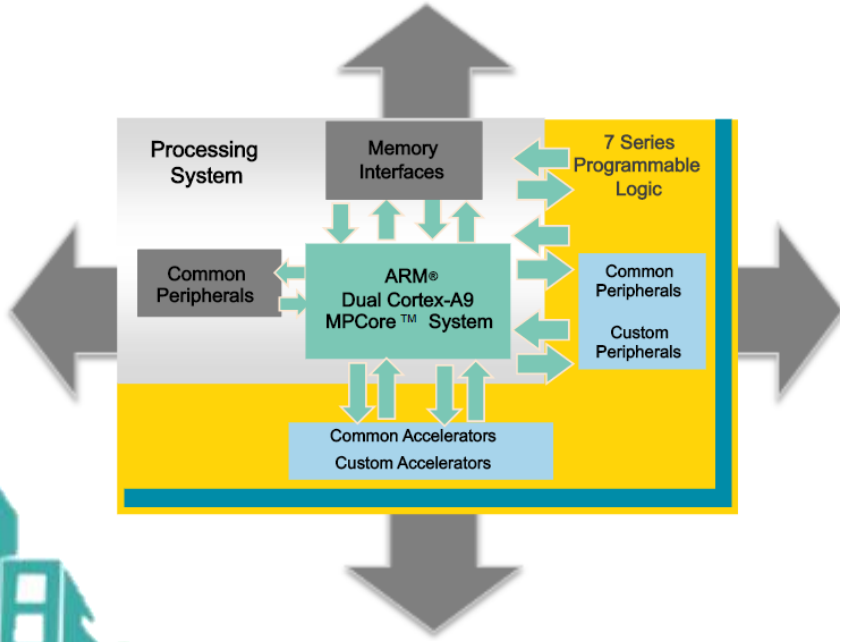
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

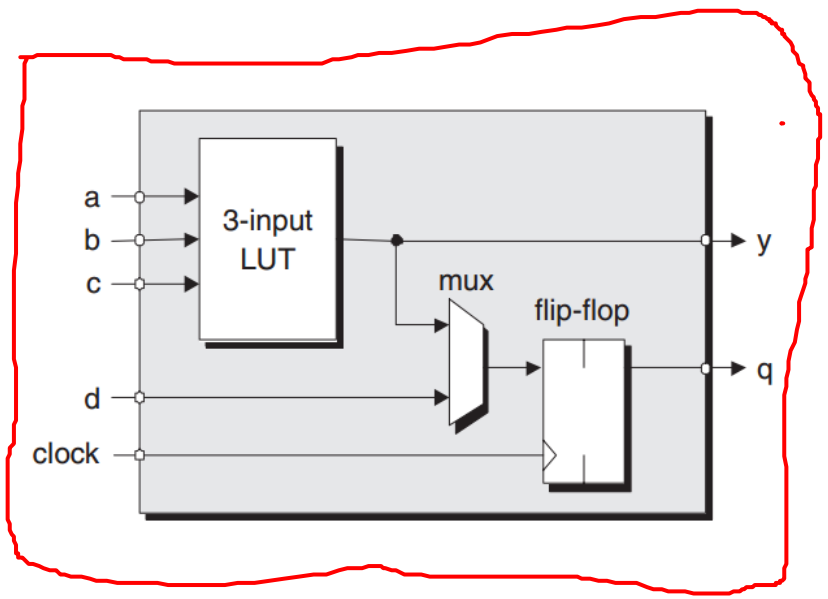
ECE 270: Embedded Logic Design

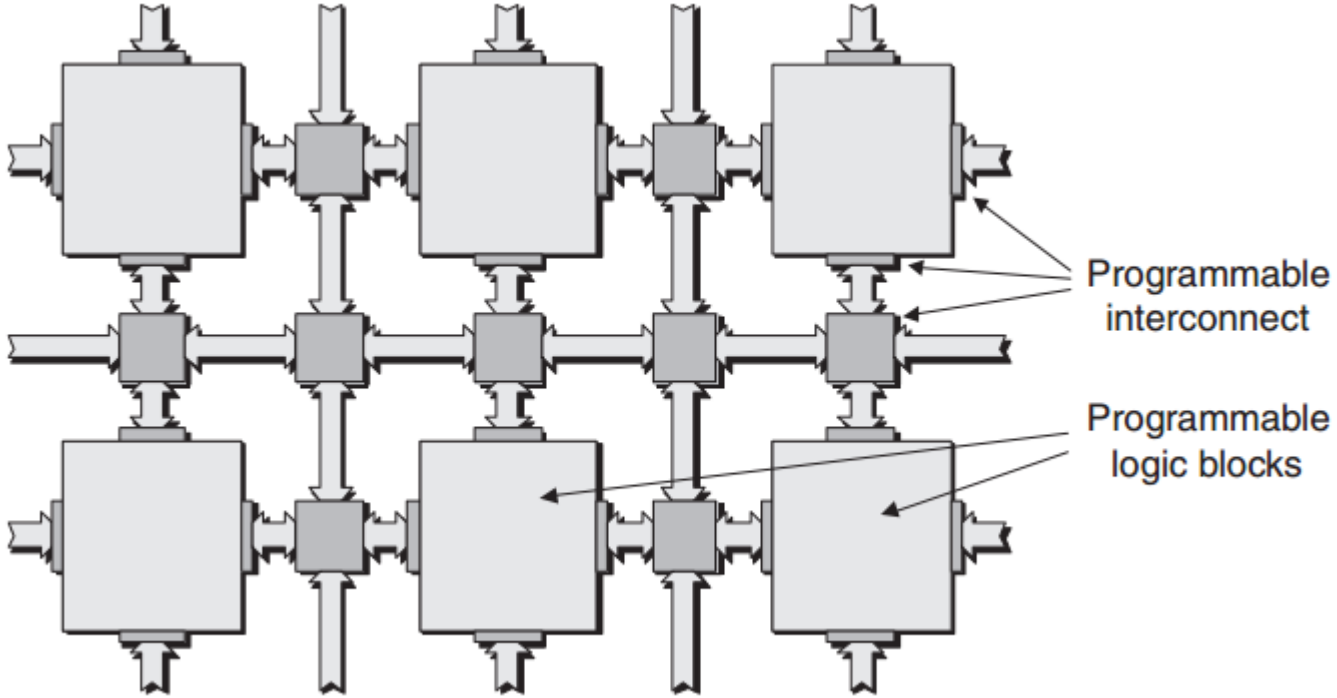
SDK
Software Development Kit
VIVADO™





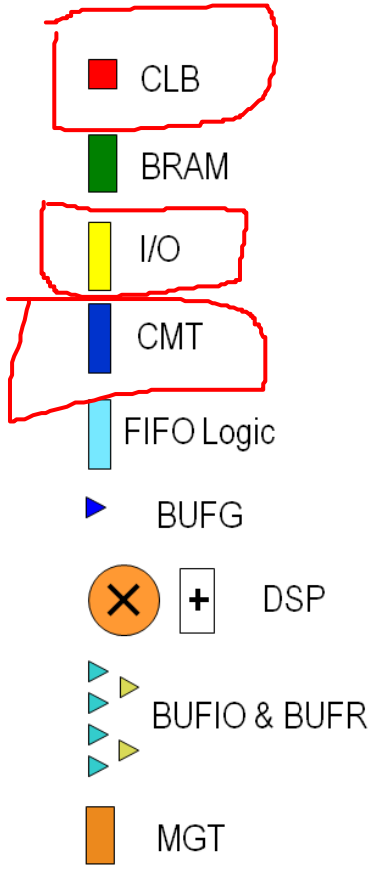
FPGA (1984)

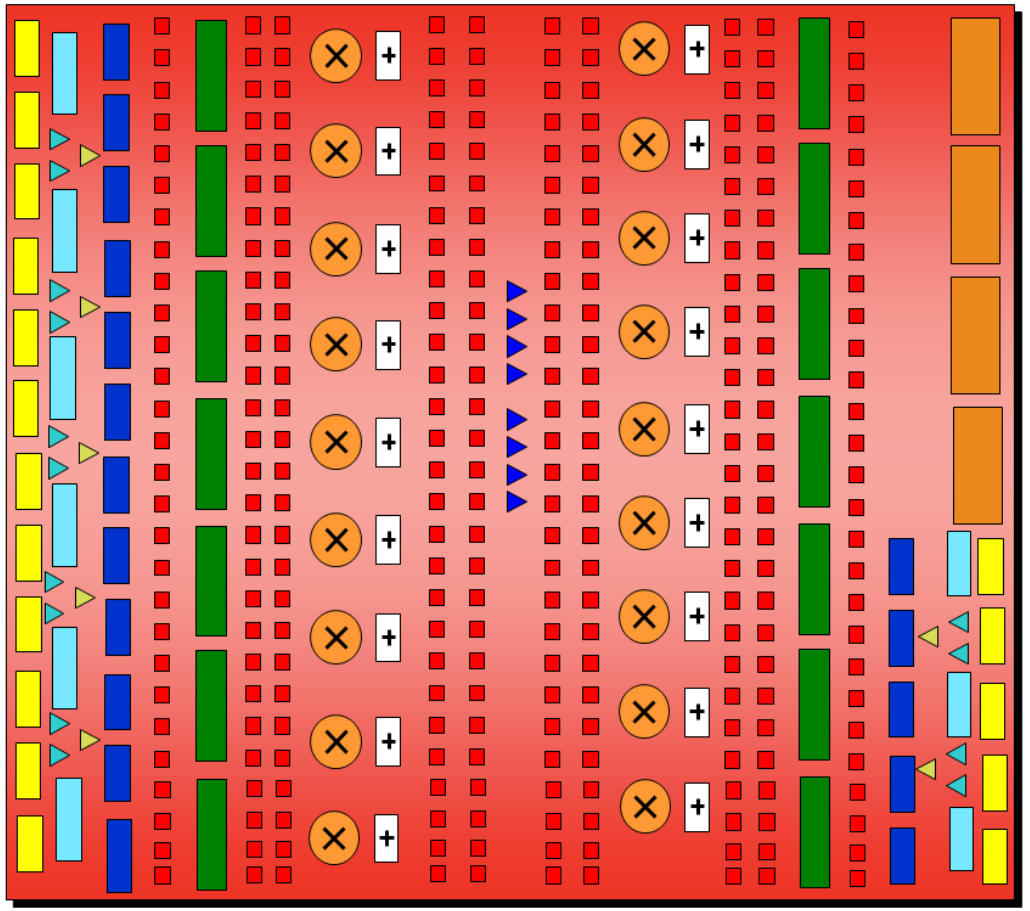

a
b
c
d
clock
3-input
LUT
mux
flip-flop
y
q


Programmable
interconnect
Programmable
logic blocks

FPGA Architecture

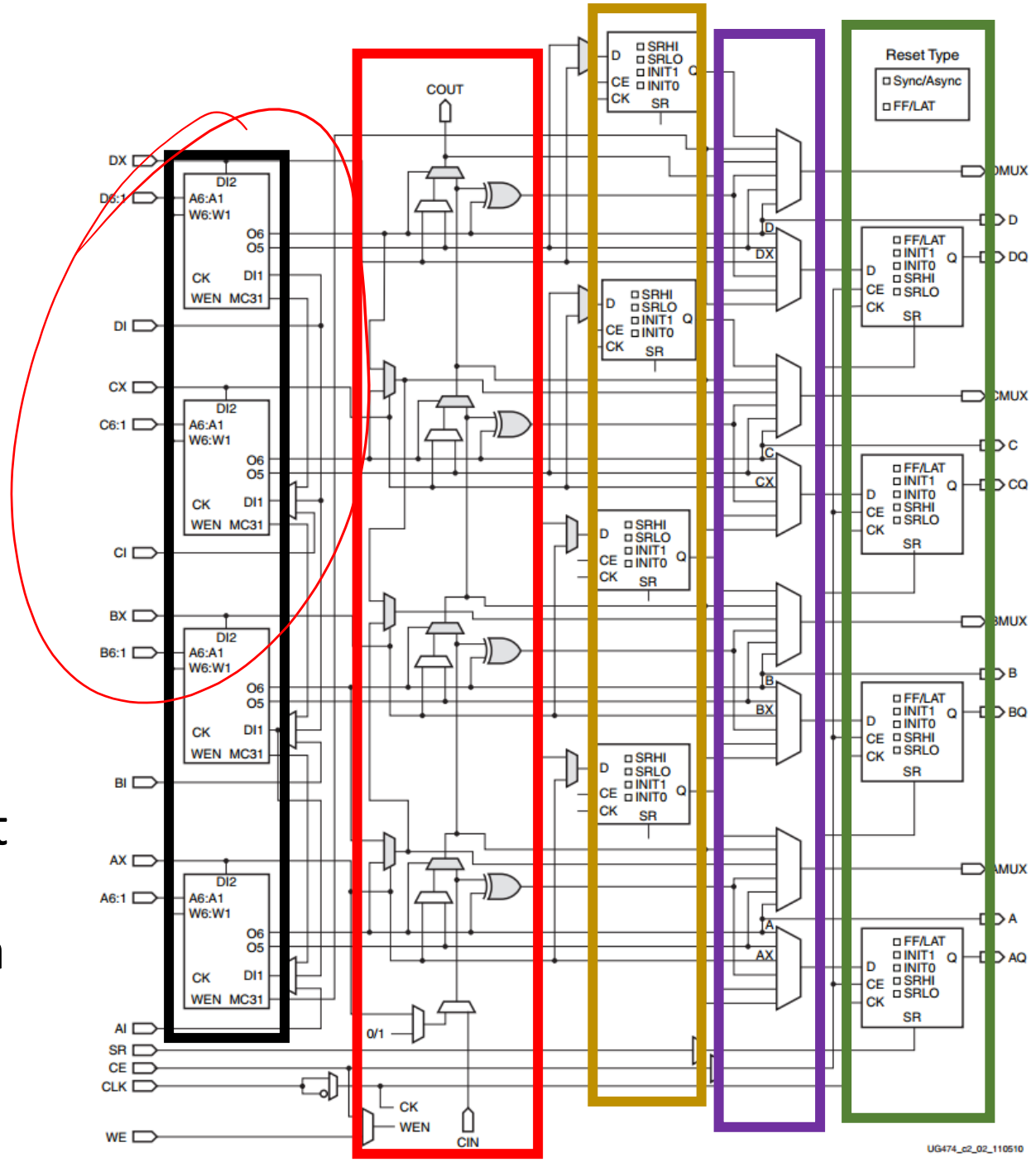
- All 7-series families share the same basic building blocks.
- The mixture and number of these resources varies across families



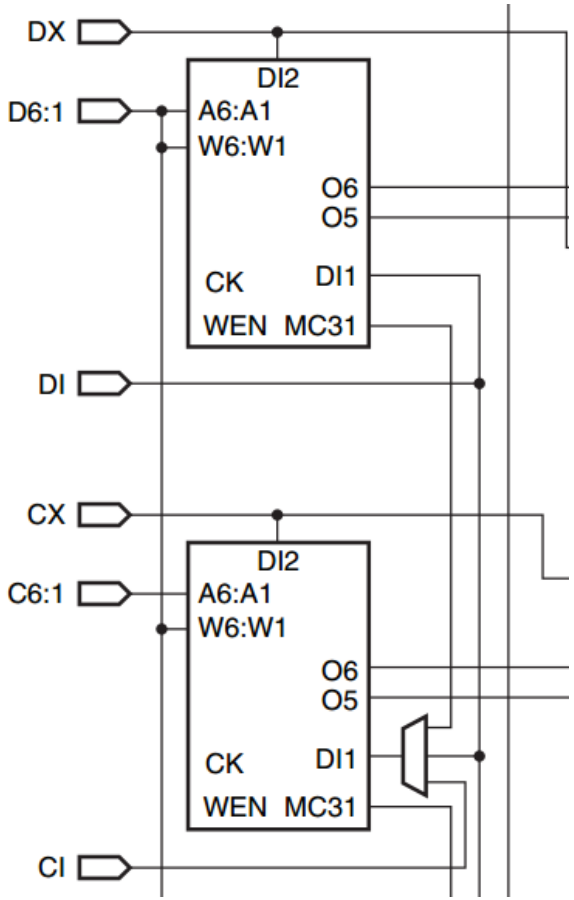


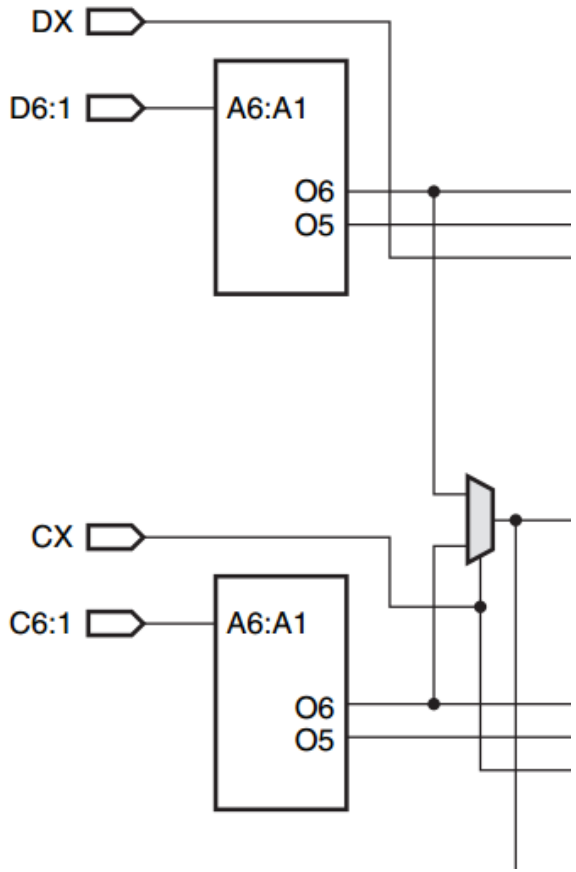
Slice Resource

- 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 Vs SLICEL

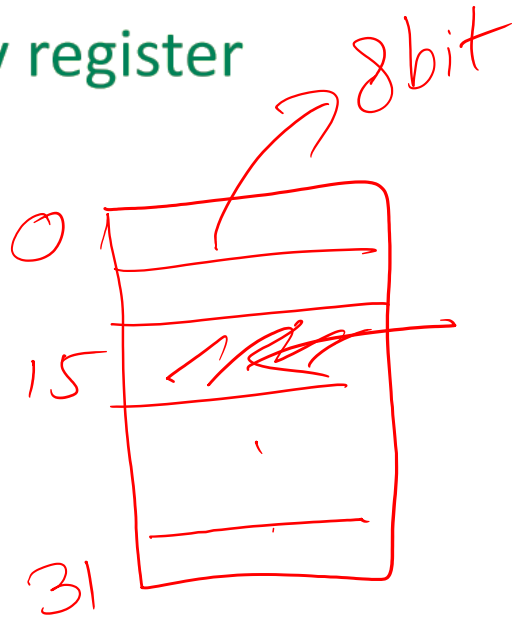




Vector and Memory

Verilog: Memory

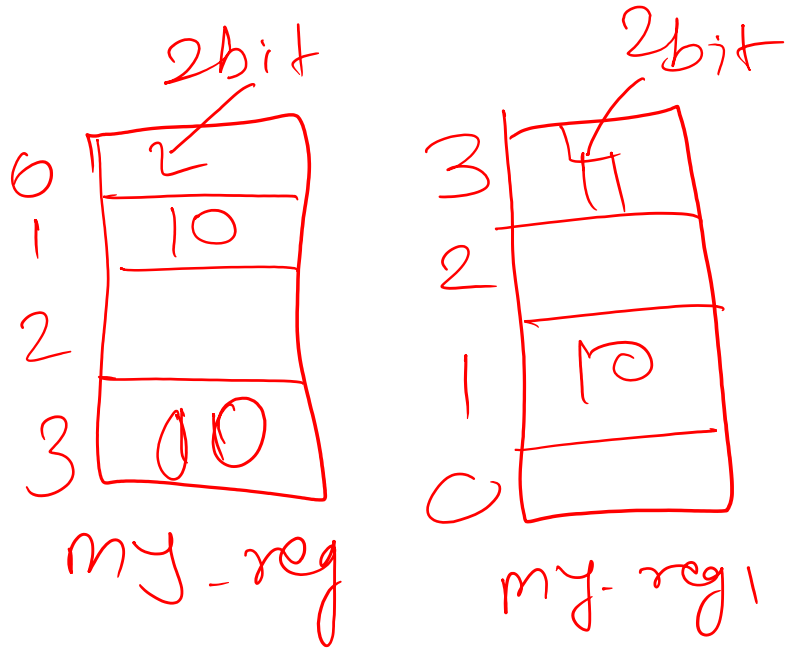
```
reg  [7:0] my_reg [0:31] ;       // Array of 32 byte-wide registers

integer matrix [4:0] [0:255] ; // 2-dimentional 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;
```

assign my_reg[3] = 2'b00;



- 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] ;
```

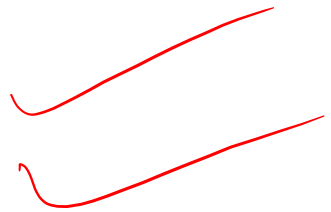
- Select fourth byte from 101th row and 8th column.

```
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
```

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

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

C = a[6:0]

// 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
```

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
```

Verilog: Register vs. Integer Numbers

- The **reg** register is a 1-bit wide data type. If more than one bit is required then range declaration should be used.
- The **integer** register is a 32-bit wide data type.
- Integer declarations cannot contain range specification.

```
module test_1(
    input [3:0] in_1,
    input [3:0] in_2,
    input sel,
    output integer [3:0] out_1
    );
```

Error: Cannot have packed dimensions of type integer

- Typically used for constants or loop variables in Verilog.
- Vivado will automatically trim any unused bits in integer. For e.g., if we declare an integer with a value of 255 then it will be trimmed to 8 bits.

Verilog: Real Numbers (Homework)

Verilog: Operators

- Operators are of three types:
  - Unary
  - Binary
  - Ternary
- Unary operators precede the operand
- Binary operators appear between two operands
- Ternary operators have two separate operators that separate three operands

```
a = ~ b ;        // ~ is a unary operator. b  is the operand
a = b && c ;  // && is the binary operator. a and b are operands
a = b ? c : d ; // ?: is a ternary operator. b, c and d are operands
```

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Reduction Operators
- Logical Operators
- Conditional Operators
- Relational Operators (HW)

Verilog: Bus Operators

A = 8'b10001011

Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010

Verilog: Bus Operators (Self-Study)

The concatenation operator { } provides mechanism to append multiple operands. Operands *must* be sized.

Examples:

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B, C} ;                    // Result Y is 4'b0010
Y = {A, B, C, D, 3'b001} ;      // Result Y is 11'b10010110001
Y = {A, B[0], C[1]} ;           // Result Y is 3'b101

assign {b [7:0], b[15:8]} = {a[15:8], a [7:0]} ; // Byte swap
assign FA_out = {cout, sum} ; // Full Adder output: carry out + Sum
```

Repetitive concatenation of the same number, can be expressed by using the replication constant.

```
Y = {4{A}} ;                    // Result Y is 4'b1111
Y = {4{A}, 2{B}, C} ;           // Result Y is 10'b1111000010
```

Verilog: Bus Operators

A = 8'b10001011

Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 =
>>	Shift right logical	A>>3 =

Verilog: Bus Operators

A = 8'b10001011

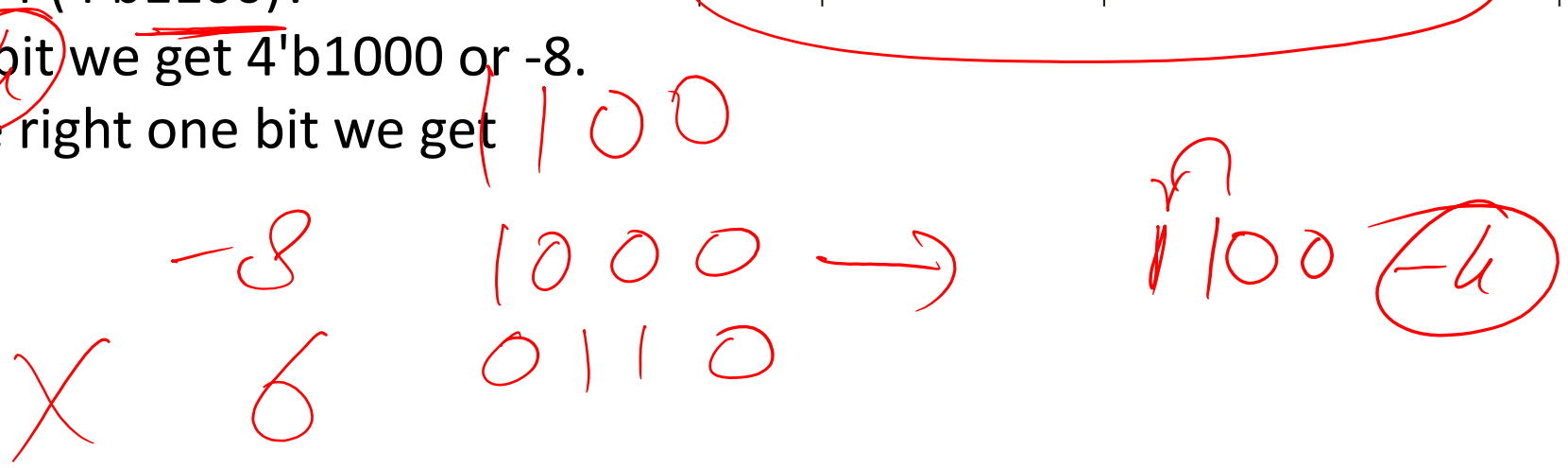
Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001

Verilog: Bus Operators

A = 8'b10001011

- Shifting bits is a very cheap (only signal renaming) way to perform multiplication and division by powers of two
- Take the value 6 (4'b0110). If we shift it to the left one bit we get 4'b1100 or 12 and if we shift it to the right one bit we get 4'b0011 or 3.
- Now what about the value -4 (4'b1100)?
- If we shift it to the left one bit we get 4'b1000 or -8. However if we shift it to the right one bit we get 4'b0110 or 6!

Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001



Verilog: Bus Operators

A = 8'b10001011

- Shifting bits is a very cheap (only signal renaming) way to perform multiplication and division by powers of two
- Take the value 6 (4'b0110). If we shift it to the left one bit we get 4'b1100 or 12 and if we shift it to the right one bit we get 4'b0011 or 3.
- Now what about the value -4 (4'b1100)?
- If we shift it to the left one bit we get 4'b1000 or -8. However if we shift it to the right one bit we get 4'b0110 or 6!
- This is why we need to use the arithmetic shift. If we use the arithmetic shift we get 4'b1110 or -2!

Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001
>>>	Shift right arithmetic	A>>>3 = 8'b11110001

a	a >> 2	a >>> 2	a << 2	a <<< 2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

Verilog: Operators

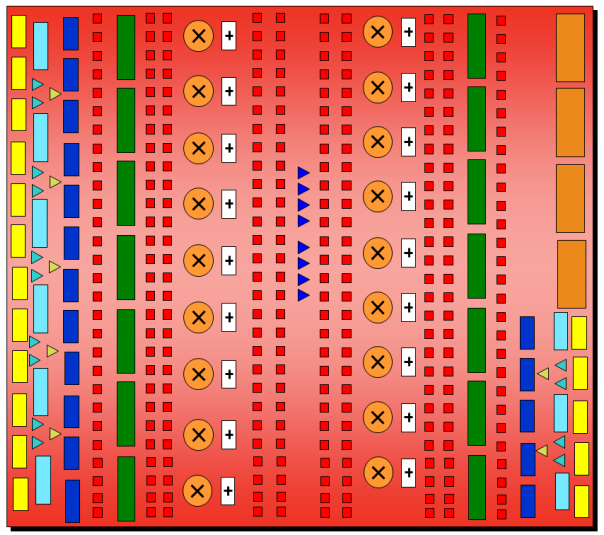
- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- Reduction Operators
- Conditional Operators
- Relational Operators (HW)

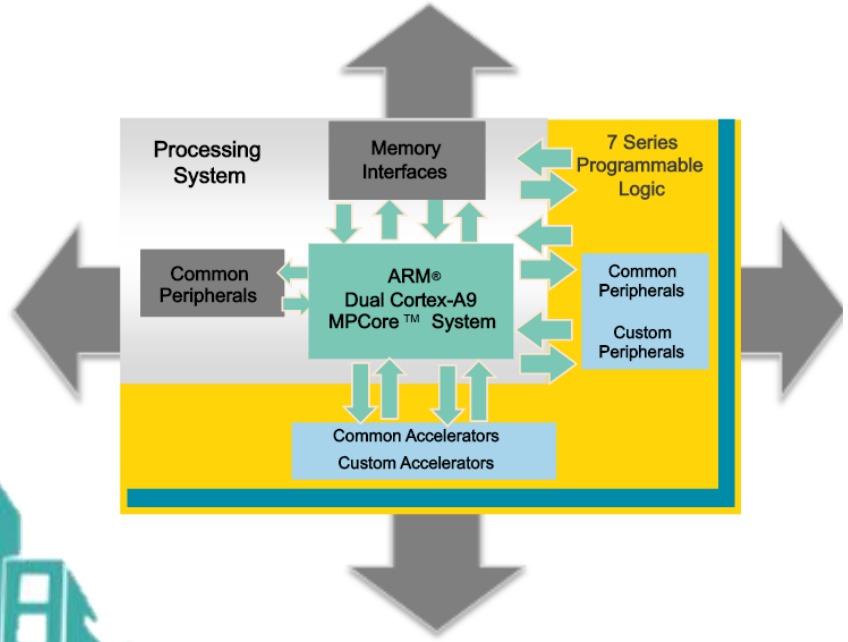
ECE | DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

SDK
Software Development Kit
VIVADO™





Verilog: Operators

- Operators are of three types:
  - Unary
  - Binary
  - Ternary
- Unary operators precede the operand
- Binary operators appear between two operands
- Ternary operators have two separate operators that separate three operands

```
a = ~ b ;        // ~ is a unary operator. b  is the operand
a = b && c ;  // && is the binary operator. a and b are operands
a = b ? c : d ; // ?: is a ternary operator. b, c and d are operands
```

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Reduction Operators
- Logical Operators
- Conditional Operators
- Relational Operators (HW)

Verilog: Bus Operators

A = 8'b10001011

Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001

Verilog: Bus Operators

A = 8'b10001011

- Shifting bits is a very cheap (only signal renaming) way to perform multiplication and division by powers of two
- Take the value 6 (4'b0110). If we shift it to the left one bit we get 4'b1100 or 12 and if we shift it to the right one bit we get 4'b0011 or 3.
- Now what about the value -4 (4'b1100)?
- If we shift it to the left one bit we get 4'b1000 or -8. However if we shift it to the right one bit we get 4'b0110 or 6!
- This is why we need to use the arithmetic shift. If we use the arithmetic shift we get 4'b1110 or -2!

Operator	Description	Example
[ ]	Bit/Part Select	A[0] = 1'b1; A[5:2] = 4'b0010 ;
{ }	Concatenation	{A[5:2],A[7:6],2'b01} = 8'b00101001
{{ }}	Replication	{3{A[7:6]}} = 6'b101010
<<	Shift left logical	A<<2 = 8'b00101100
>>	Shift right logical	A>>3 = 8'b00010001
>>>	Shift right arithmetic	A>>>3 = 8'b11110001

a	a >> 2	a >>> 2	a << 2	a <<< 2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- Reduction Operators
- Conditional Operators
- Relational Operators (HW)

Verilog: Arithmetic Operators

A = 8'b10001011 = 139
B = 8'b00001100 = 12

Operator	Description	Example
+	Addition	A + 12 = 151 = 8'b10010111
-	Subtraction	A – 10 = 129 = 8'b10000001
*	Multiplication	A * 3 = 417 = 9'b110100001
/	Division	A / 2 = 69 = 7'b1000101
%	Modulus	A % 5 = 4 = 3'b100
**	Power (exponent)	B ** 2 = 144 = 8'b10010000

Verilog: Arithmetic Operators

A = 8'b10001011 = 139
B = 8'b00001100 = 12

- For multiplication, many FPGAs have special resources (DSP48) dedicated to fast math.
- If these are available they will be used, however, if you run out of them a multiplication circuit will have to be generated (using CLB) which can be large and slow.
- Never multiply by a power of two, but use shift instead.
- Most tools are smart enough to do this for you but it is good practice not to rely on that.

Operator	Description	Example
+	Addition	A + 12 = 151 = 8'b10010111
-	Subtraction	A – 10 = 129 = 8'b10000001
*	Multiplication	A * 3 = 417 = 9'b110100001
/	Division	A / 2 = 69 = 7'b1000101
%	Modulus	A % 5 = 4 = 3'b100
**	Power (exponent)	B ** 2 = 144 = 8'b10010000

Verilog: Arithmetic Operators

A = 8'b10001011 = 139
B = 8'b00001100 = 12

- When two N-bit numbers are added or subtracted, an N+1-bit number is produced.
- If you add or subtract two N-bit numbers and store the value in an N-bit number you need to think about overflow.
- For example, if you have a two bit number 3 (2'b11) and you add it to 2 (2'b10) the result will be 5 (3'b101) but if you store the value in a two bit number you get 1 (2'b01).
- For multiplication of two N-bit numbers, the result will be an N*2-bit number.

Operator	Description	Example
+	Addition	A + 12 = 151 = 8'b10010111
-	Subtraction	A – 10 = 129 = 8'b10000001
*	Multiplication	A * 3 = 417 = 9'b110100001
/	Division	A / 2 = 69 = 7'b1000101
%	Modulus	A % 5 = 4 = 3'b100
**	Power (exponent)	B ** 2 = 144 = 8'b10010000

Verilog: Arithmetic Operators (Self-Study)

```
//suppose that: a = 4'b0011;
//               b = 4'b0100;
//               d = 6; e = 4; f = 2;
//then,
a + b  //add a and b; evaluates to  4'b0111
b - a  //subtract a from b; evaluates to  4'b0001
a * b  //multiply a and b; evaluates to  4'b1100
d / e  //divide d by e, evaluates to  4'b0001. Truncates fractional part
e ** f //raises e to the power f, evaluates to 4'b1111
       //power operator is most likely not synthesible
```

If any operand bit has a value "x", the result of the expression is all "x".
If an operand is not fully known the result cannot be either.

Verilog: Bitwise Operators

A = 8'b10001011

- Operate on each bit individually.
- When you perform a bitwise operator on multi-bit values, you are essentially using multiple gates to perform the bitwise operation.
- If the two values used by a bitwise operator are different in length, the shorter one is filled with zeros to make the lengths match.

Operator	Description	Example
~	Inverse / NOT	~A = 8'b01110100
&	AND	A[2] & A[1] = 1'b0
|	OR	A[2] | A[1] = 1'b1
^	XOR	A[2] ^ A[1] = 1'b1
~^	XNOR	A[2] ~^ A[1] = 1'b0

Verilog: Logical Operators

A = 8'b10001011

Operator	Description	Example
!	NOT	!A[1] = FALSE, !A[2] = TRUE
&&	AND	A[0] && A[1] = TRUE
||	OR	A[0] || A[2] = TRUE
==	EQUAL	A[3:0] == 4'b1011 = TRUE
!=	NOT EQUAL	A[3:0] != 4'b1011 = FALSE
<,<=,>,>=	COMPARE	A[3:0] < 13 = TRUE

A && B

Verilog: Logical and Bitwise Operators

a	b	a&b	a|b	a&&b	a||b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

Verilog: Reduction Operators

A = 8'b10001011

- **Reduce** the number of bits to one by performing the specified function on every bit.
- Similar to the bitwise operators, except they are performed on all the bits of a single value

Operator	Description	Example
&	AND	&A = A[0] & A[1] & … A[7] = 1'b0
~&	NAND	~&A = ~(A[0] & A[1] & … A[7]) = 1'b1
|	OR	|A = A[0] | A[1] | … A[7] = 1'b1
~|	NOR	~|A = ~(A[0] | A[1] | … A[7]) = 1'b0
^	XOR	^A = A[0] ^ A[1] ^ … A[7] = 1'b0
~^	XNOR	~^A = ~(A[0] ^ A[1] ^ … A[7]) = 1'b1

Verilog: Operators (Self-Study)

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a|b	a||b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

Verilog: Operators

- Bus Operators
- Arithmetic Operators
- Bitwise Operators
- Logical Operators
- Reduction Operators
- Conditional Operators
- Relational Operators (HW)

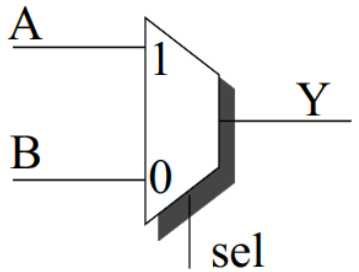
Verilog: Conditional Operator

- Can be used in place of *if* statement when one of the two or more values is to be selected for assignment

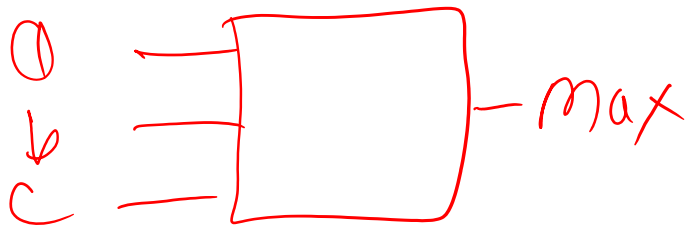
condition ? value_if_true : value_if_false

- Can be part of procedural or continuous assignment

assign Y = (sel)? A : B ;



Verilog: Conditional Operator



- Design maximal circuit to return the maximum of a, b and c.

```
assign max = (a>b) ? ((a>c) ? a : c) :
                     ((b>c) ? b : c);
```

- Design 4:1 Multiplexer using conditional operator
- Design 1-bit equality comparator using conditional operator

Verilog: Operator Precedence

- If no parentheses are used to separate operands, then Verilog uses the following rules of precedence (Good practice: use parentheses)

Operators	Operators Symbols	
Unary	+ - ! ~	Highest Precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >> >>>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~&, ^, ^~, |, ~|	
Logical	&&	
Conditional	?:	Lowest Precedence

Verilog: Hint for $signed

When you write this in Verilog:

```
wire [7:0] a;
wire [7:0] b;
wire less;
assign less = (a < b);
```

the comparison between a and b is unsigned, that is a and b are numbers in the range 0-255. Writing this instead:

```
wire [7:0] a;
wire [7:0] b;
wire less;
assign less = ($signed(a) < $signed(b));
```

means that the comparison treats a and b as *signed* 8-bit numbers, which have a range of -128 to +127. Another way of writing the same thing is:

```
wire signed [7:0] a;
wire signed [7:0] b;
wire less;
assign less = (a < b);
```

Verilog (Three Concepts)

- Difference between Register and Wire (Next two lectures)
- **Efficient Behavioral modelling**
- Difference between blocking and non-blocking assignments

Behavioral Modeling

Behavioral Modeling

- There are two basic statements in behavioral modeling: *initial* and *always*
- All other statements appear inside these statements.
- All initial and always blocks run in parallel
- All of them start at simulation time 0.
- Initial block starts at time 0 and executes only once. The initial statement provides a means of initiating input waveforms and initializing simulation variables before the actual description/simulation begins.
- Once the statements in the initial are exhausted, statement becomes inactive.

Behavioral Modeling

```
initial
    begin  /* multiple statements, need to be grouped –
                  begin/end */
        clock = 1'b0 ;   // clock initial logic state
        nrst = 1'b0      // reset initial logic state
    end

initial
    begin
        # 5 a = 1'b1 ;   // set a to 1 @ simulation time 5
        # 25 b = 1'b1 ; // set b to 1 @ simulation time 30
        # 50 $finish ;   // end simulation after 50 time ticks
    end
```

t=0 a=1, b=1;
t=50 a=0, b=1;
t=80 a=1, b=1;

initial
begin
d = #5 a&b;
end

t=0 t=5
t=5 t=85 #80 e = a&b;
e=1

Behavioral Modeling

```
module Reg_File
    ……
    reg [31:0] RegFile[0:31] ;
    integer i ;
    initial
      begin
          for(i = 0 ; i < 32 ; i = i + 1)
              RegFile[i] = 32'h0 ;
      end
    ……
endmodule
```

Behavioral Modeling

- *always block* starts at time 0 and executes statements continuously in a loop.
- Describes the function of a circuit.
- Can contain many statements like if, for, while, case
- Statements the always block are executed *sequentially* (= assignment) or in *parallel* (<= assignment)
- The *final* result describes the function of the circuit for current set of inputs.

```
// clock declaration, used mainly in Test Benches
always
    # 10 clock = ~clock ;   // Toggle clock every half-cycle
```

Behavioral Modeling

always block

- Always waiting for a change to a trigger signal
- Then executes the body

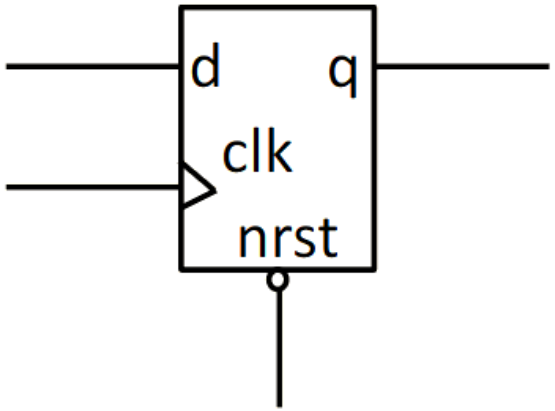
```
module and_gate (out, in1, in2) ;
   input   in1, in2 ;
  output  reg out ;


 always @(in1 or in2)
   begin
     out = in1 & in2 ;
   end
endmodule
```

Specifies when block is executed
I.e., triggered by which signals

Behavioral Modeling

```
module full_adder (sum, cout, a, b, cin) ;
   input a, b, cin ;
   output reg sum, cout ;   // implicit register

   always @(a or b or cin)  // Verilog 2001 allows (a, b, cin)
      {cout, sum} = a + b + cin ;
endmodule
      // If sensitivity list is too long, use (*), i.e. all inputs
```

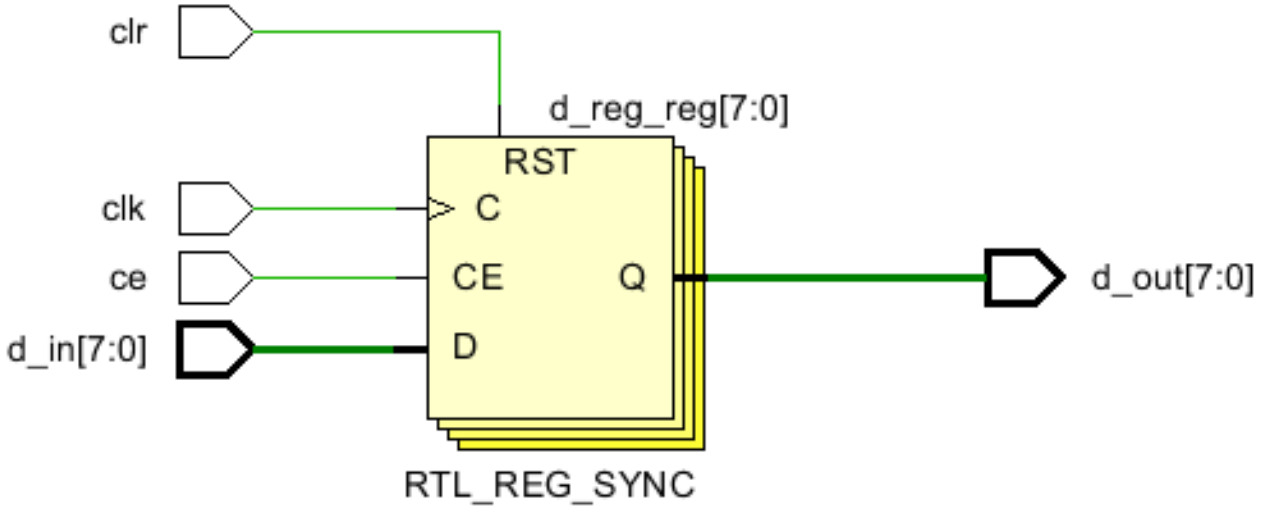
Verilog: Module (Examples)




```
module D_FF(clk, nrst, d, q) ;
    input clk, nrst, d ;
    output reg q ;
    always @(posedge clk or negedge nrst)
                                        // Event-based Timing Control
      if (!nrst)                        // reset state
        q <= 0 ;
      else                              // normal operation
        q <= d ;
endmodule
```

Verilog: Register

```
module test_1(
    input [7:0] d_in,
    input ce,
    input clk,
    input clr,
    output [7:0] d_out
    );

    reg [7:0] d_reg;
    always@(posedge clk)
    begin
        if(clr)
            d_reg <= 8'b00000000;
        else if (ce)
            d_reg <= d_in;
    end
    assign d_out = d_reg;
endmodule
```



Behavioral Modeling for Combinational Circuits

- While writing Verilog code for synthesis, we need to be aware of how the various language constructs are mapped to hardware.
- **Common errors**:

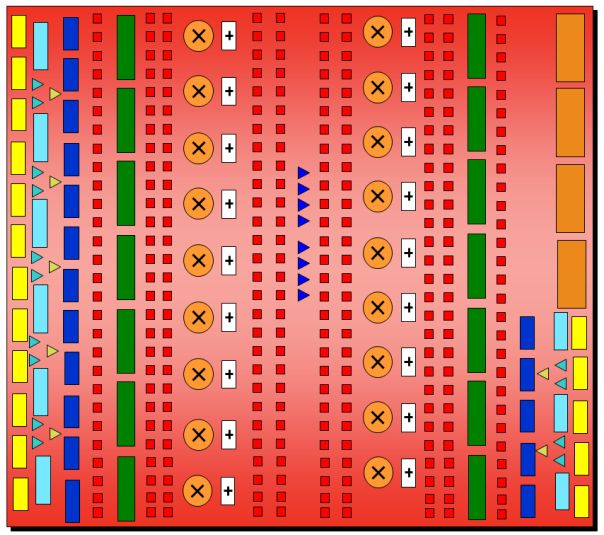
1. Variable assigned in multiple always blocks
2. Incomplete sensitivity list
3. Incomplete branch and incomplete output assignments

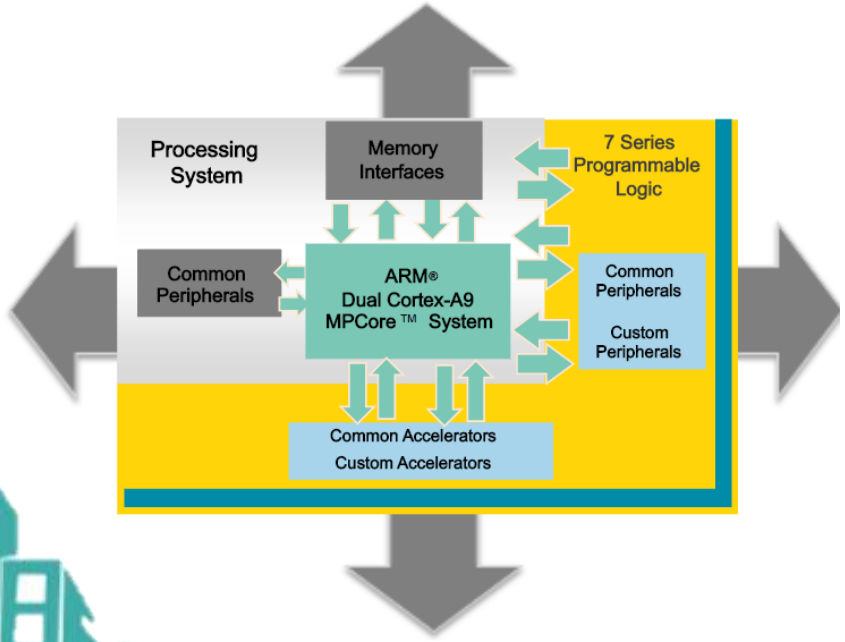
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

SDK
Software Development Kit
VIVADO™





Course Feedback

- Quiz will be announced few days in advance. Exact date and time will NOT be announced.
- **Tuesday Timetable:** Please reach out to academic team to explore if there is any alternative option.

Behavioral Modeling for Combinational Circuits

- While writing Verilog code for synthesis, we need to be aware of how the various language constructs are mapped to hardware.
- **Common errors**:

1. Variable assigned in multiple always blocks
2. Incomplete sensitivity list
3. Incomplete branch and incomplete output assignments

Behavioral Modeling: Guidelines

- Variable assigned in multiple always blocks

```
reg y;
reg a, b, clear;
. . .
always @*
   if (clear) y = 1'b0;

always @*
   y = a & b;
```

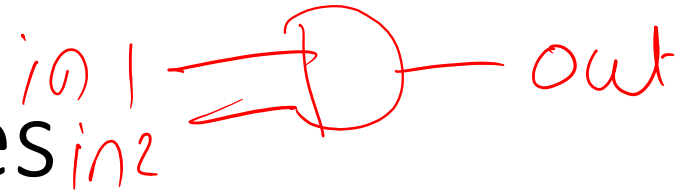
```
always @*
   if (clear)
      y = 1'b0;
   else
      y = a & b;
```

1) Separate ALWAYS block for every identifier

2) Don't update multiple identifiers in one ALWAYS Block.

- LHS code is not synthesizable since output y is driven by two blocks (i.e. two different circuits).
- No physical circuit exhibits such behaviour.

Behavioral Modeling: Guidelines



- Incomplete sensitivity list

```
module and_gate (out, in1, in2) ;
   input   in1, in2 ;
   output reg out ;

   always @(in1)    // should be (in1 or in2)
      begin
         out = in1 & in2 ;
      end
endmodule
```

- Leaving out an input trigger usually results in a sequential circuit
- **Use always@*** for combinational circuits

Behavioral Modeling: Guidelines

4) Else is MUST in if_else loop.

- Incomplete branch and incomplete output assignment

```
always @*
   if (a > b)          // eq not assigned in this branch
      gt = 1'b1;
   else if (a == b) // gt not assigned in this branch
      eq = 1'b1;
                       // final else branch is omitted
```

```
always @*
begin
   gt = 1'b0;   // default value for gt
   eq = 1'b0;   // default value for eq
   if (a > b)
      gt = 1'b1;
   else if (a == b)
      eq = 1'b1;
end
```

```
always @*
   if (a > b)
      begin
         gt = 1'b1;
         eq = 1'b0;
      end
   else if (a == b)
      begin
         gt = 1'b0;
         eq = 1'b1;
      end
   else    // i.e., a < b
      begin
         gt = 1'b0;
         eq = 1'b0;
      end
```

Behavioral Modeling: Guidelines

- Incomplete branch and incomplete output assignment

```
reg [1:0] s
. . .
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   default: y = 1'b1; // y gets 1 for 2'b01
endcase
```

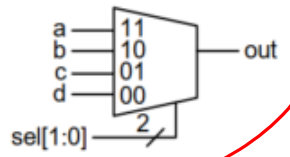
```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
   default: y = 1'bx; // y gets x for 2'b01
endcase
```

```
y = 1'b0;    // can also use y = 1'bx for don't-care
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

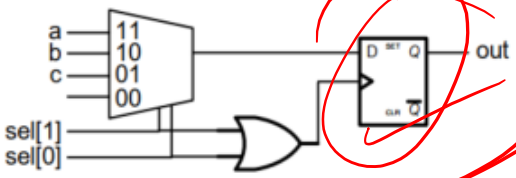
Case Statement: Full and Parallel

- Full Case Statement: all possible outcomes are accounted
- Parallel Case Statement: all stated alternatives are mutually exclusive

```
module full_par (sel, a, b, c, d, out);
input [1:0] sel;
input       a, b, c, d;
output      out; reg out;
always @ (sel or a or b or c or d)
  case (sel)
  2'b11:  out <= a;
  2'b10:  out <= b;
  2'b01:  out <= c;
  default: out <= d; // 2'b00
  endcase
endmodule
```



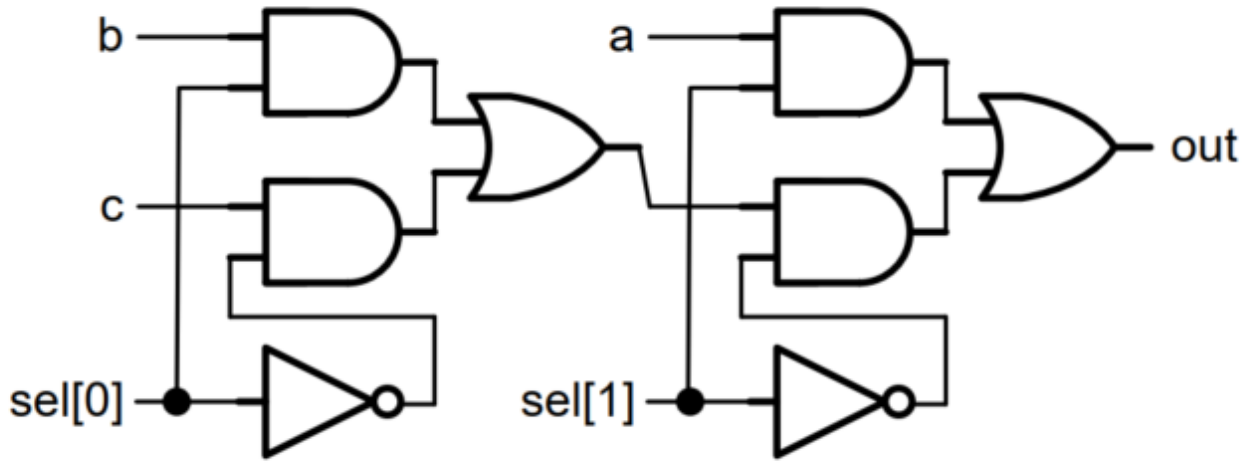

```
module par_not_full (sel, a, b, c, out);
input [1:0] sel;
input       a, b, c;
output      out; reg out;
always @ (sel or a or b or c)
  case (sel)
  2'b11:  out <= a;
  2'b10:  out <= b;
  2'b01:  out <= c;
  endcase
endmodule
```



Case Statement: Full and Parallel

- Full Case Statement: all possible outcomes are accounted
- Parallel Case Statement: all stated alternatives are mutually exclusive

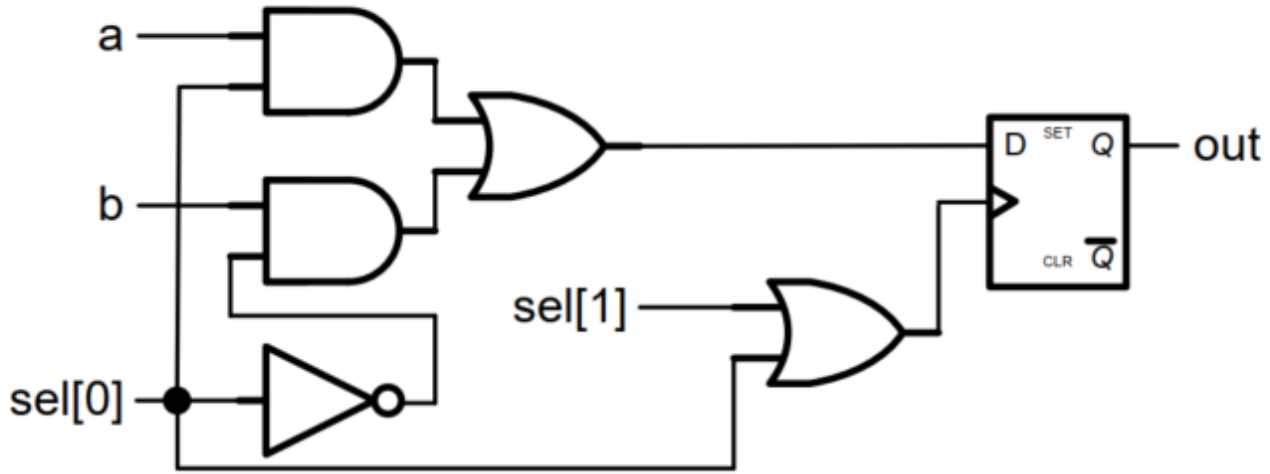
```
module full_not_par (sel, a, b, c, out);
input [1:0] sel;
input       a, b, c;
output      out; reg out;
always @ (sel or a or b or c)
   case (sel)
   2'b1?:    out <= a; // 2'b10, 2'b11
   2'b?1:    out <= b; // 2'b01, 2'b11
   default: out <= c; // 2'b00
   endcase
endmodule
// If the case is 2'b11 occurs, the first outcome gets higher priority because it is closer to the output.
```



Case Statement: Full and Parallel

- Full Case Statement: all possible outcomes are accounted
- Parallel Case Statement: all stated alternatives are mutually exclusive

```
module not_full_not_par (sel, a, b, out);
input [1:0] sel;
input       a, b;
output      out; reg out;
always @ (sel or a or b)
  case (sel)
  2'b1?:    out <= a; // 2'b10, 2'b11
  2'b?1:    out <= b; // 2'b01, 2'b11
  endcase
endmodule
```



Behavioral Modeling for Combinational Circuits

- If an always block executes and a variable is ***NOT*** assigned:
  - Variable keeps its old value
  - *NOT* combinational logic -> latch is inserted (implied memory)
  - This is usually *NOT* what you want

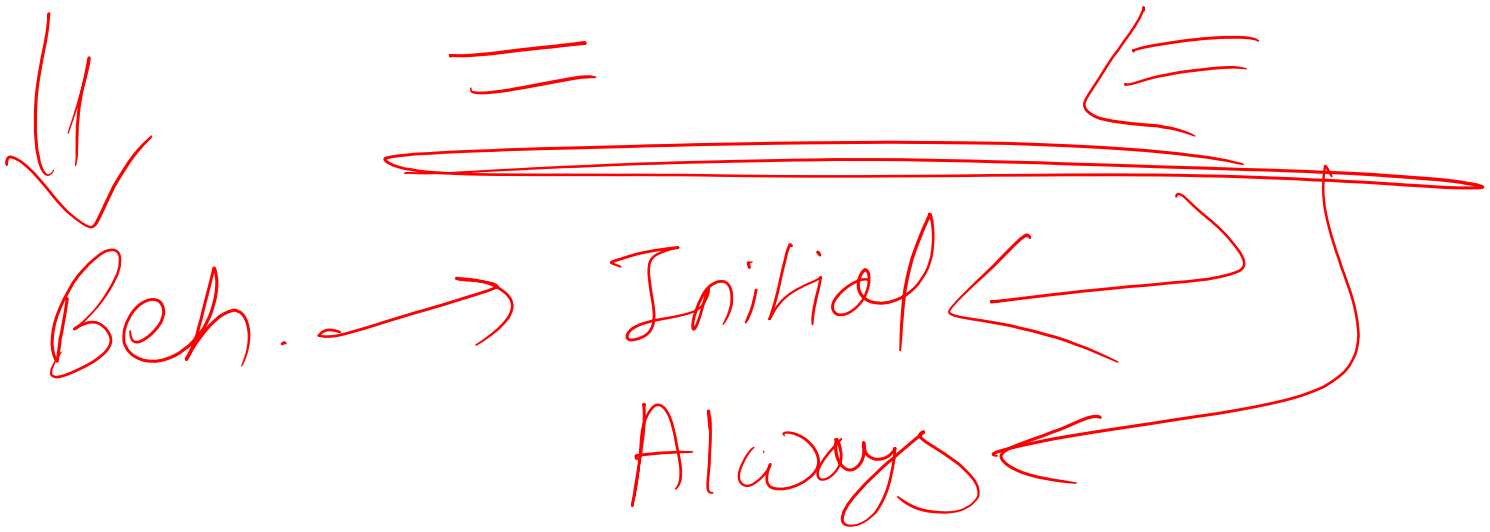
- Any variable assigned in an always block SHOULD be assigned for any (and every!) execution of the block
- Poorly coded always block leads to unnecessarily complex implementation or can not be synthesized at all.

Behavioral Modeling: Summary

- Assign a variable ONLY in a single always block
- **Use @*** to include all the desired identifiers automatically in the sensitivity list
- Make sure that **all branches** of the *if and case statements* are included
- Make sure that the outputs are assigned in all branches
- One way to satisfy previous two guidelines is to assign default values for outputs in the beginning of the always block
- ~~**Use blocking assignments for combinational circuits**~~
- Think hardware, not C or Python or MATLAB code

Verilog (Three Concepts)

- Difference between Register and Wire (Next two lectures)
- Efficient Behavioral modelling
- **Difference between blocking and non-blocking assignments**



Blocking Vs Non-Blocking Assignments

Behavioral Modeling

- ***Blocking* assignment** statements are executed in the order they are specified in a sequential block
- A blocking assignment will NOT block execution of statements that follows in a parallel block
- The = operator is used to specify blocking assignments
- ***Non-blocking* assignments** allow scheduling of assignments without locking execution of the statements that follow in a sequential block
- A <= operator is used to specify non-blocking assignments

Behavioral Modeling

```
reg_a = 16'b0 ;
```

```
reg [15:0] reg_a, reg_b ;
reg x, y, z ;
integer count ;
Initial
    begin
        x = 0 ; y = 0; z = 0 ; count = 0 ;
        reg_b = reg_a ;
        #15 reg_a[2] = 1'b1 ;
        #10 reg_b[15:13] = {x, y, z} ;
        count = count + 1 ;
    end
```

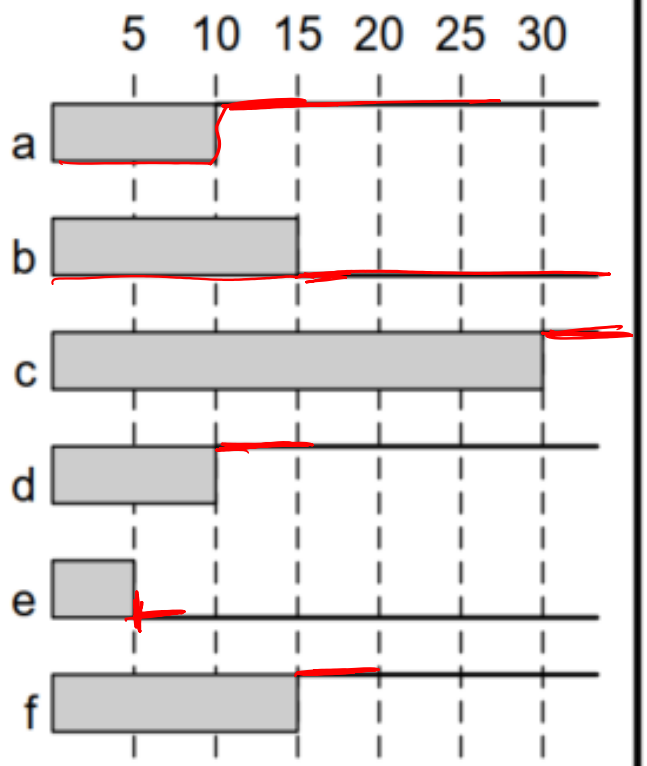
Behavioral Modeling

reg_a = 16'b0 ;

```
reg [15:0] reg_a, reg_b ;
reg x, y, z ;
integer count ;
Initial
  begin
    x <= 0 ; y <= 0; z <= 0 ; count <= 0 ;
    reg_b <= reg_a ;
    #15 reg_a[2] <= 1'b1 ;
    #10 reg_b[15:13] <= {x, y, z} ;
    count <= count + 1 ;
  end
```

Behavioral Modeling

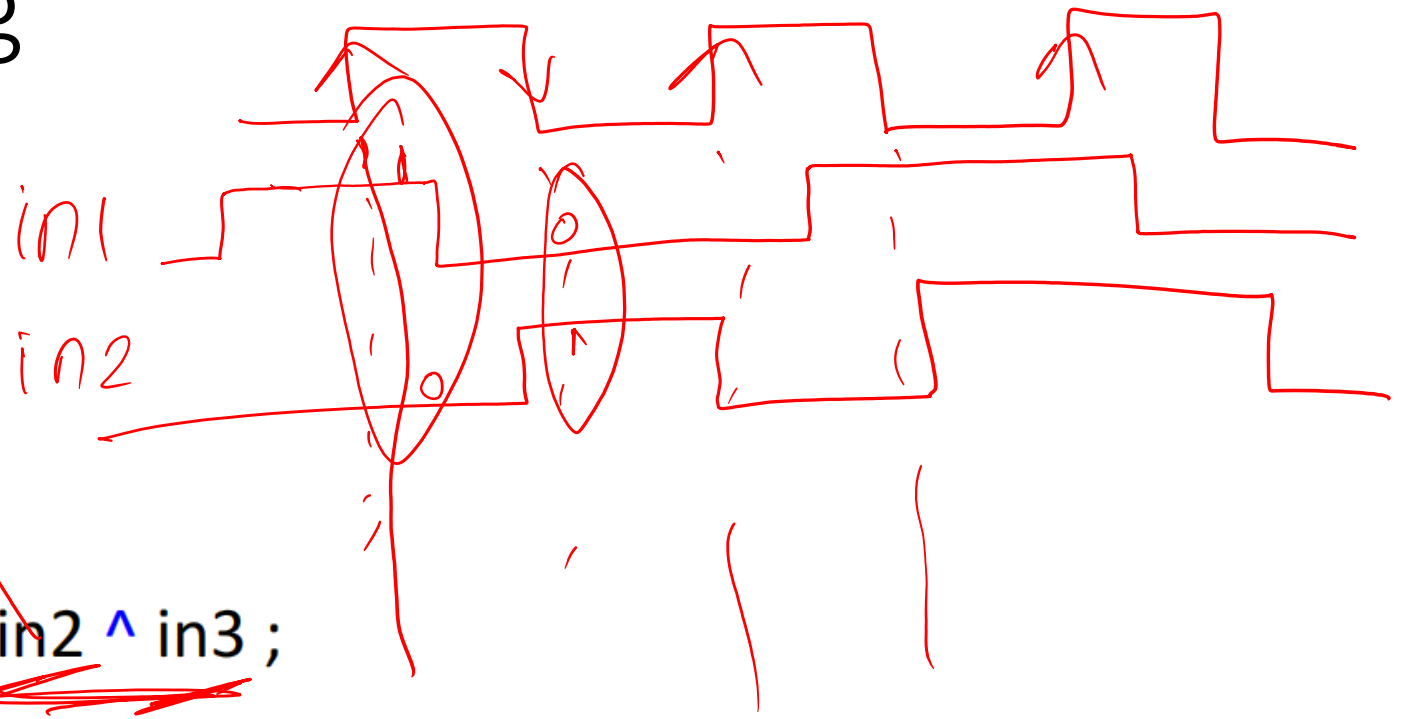
```
module top;
reg a, b, c, d, e, f;
initial begin
        a = #10  1;
        b = #5   0;
        c = #15  1;
end
initial begin
        d <= #10  1;
        e <= #5   0;
        f <= #15  1;
end
endmodule
```



Note that what are different with blocking and non-blocking assignments.

Behavioral Modeling

```
always @(posedge clock)

   begin reg1 <= #1 in1 ;

      reg2 <= @(negedge clock) in2 ^ in3 ;
      reg3 <= #1 reg1 ;
   end
```



Behavioral Modeling: Swapping

Race Conditions:

```
always @(posedge clock)
    a = b ;
always @(posedge clock)
    b = a ;
```

// Two concurrent always blocks with blocking

1) a=b 2) a=b 3) b=a
   b=a   b=b   a=a

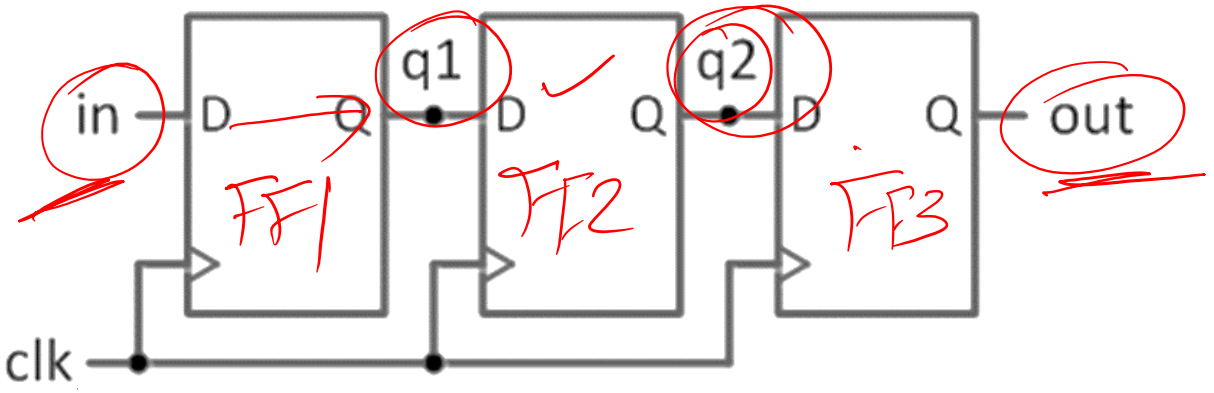
The values of both registers will not be swapped */

Race Conditions: /* Two concurrent always blocks with non- blocking

```
always @(posedge clock)
    a <= b ;
always @(posedge clock)
    b <= a ;
```

1) a=b 2) a=b_old 3) b=a_old
   b=a   b=b_old   a=b_old

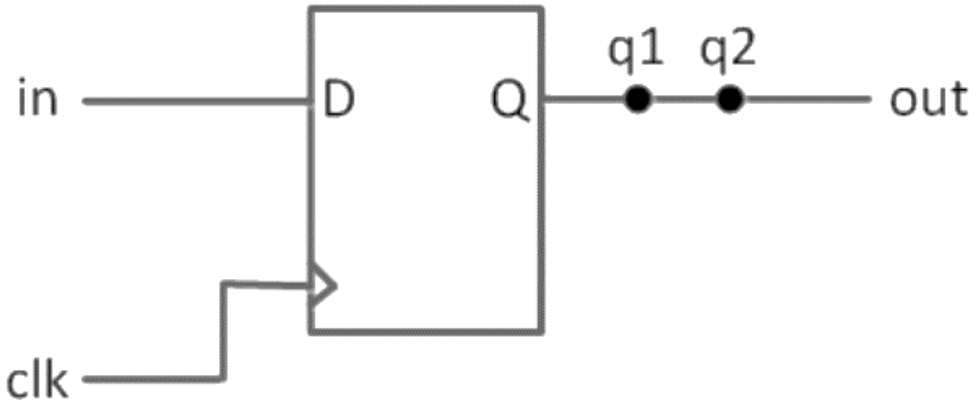
Behavioral Modeling



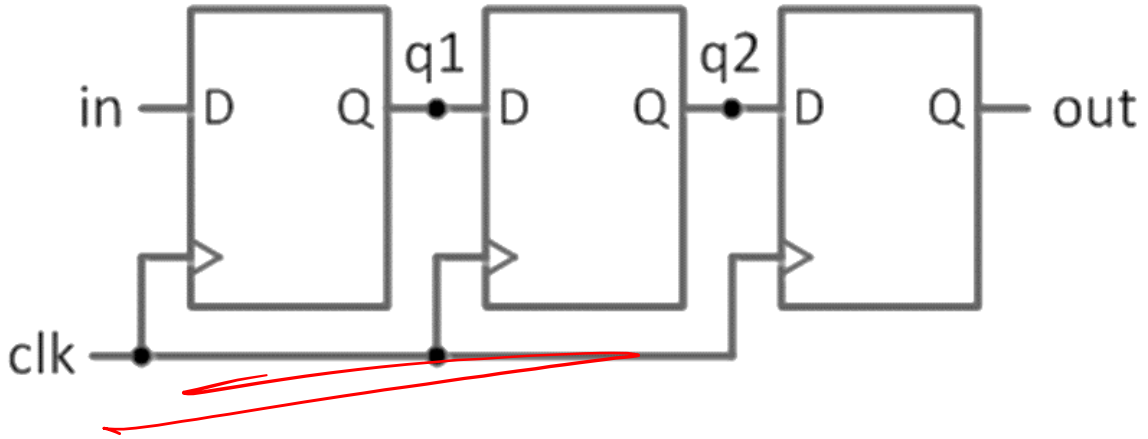

```
module blocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2 ;
   always @(posedge clk)
      begin
         q1 = in ;
         q2 = q1 ;
         out = q2 ;
      end
endmodule
```

out = in;

"at each rising clock edge, q1 = in,
after that, q2 = q1 = in
after that, out = q2 = q1 = in
Therefore, out = in"



Behavioral Modeling

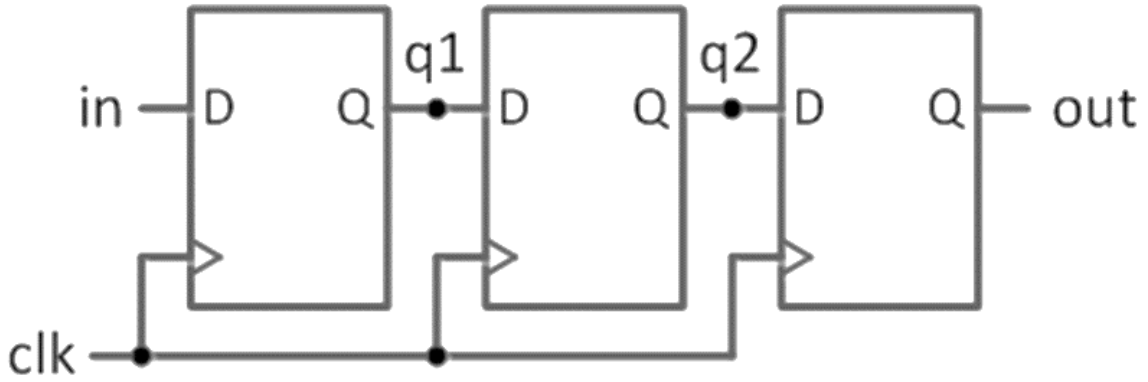



```
module nonblocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2;
   always @(posedge clk)
      begin
         q1 <= in ;
         q2 <= q1 ;
         out <= q2 ;
      end
endmodule
```

"at each rising clock edge, q1, q2 and out simultaneously receive the old values of in, q1 and q2. Therefore, out = q2"

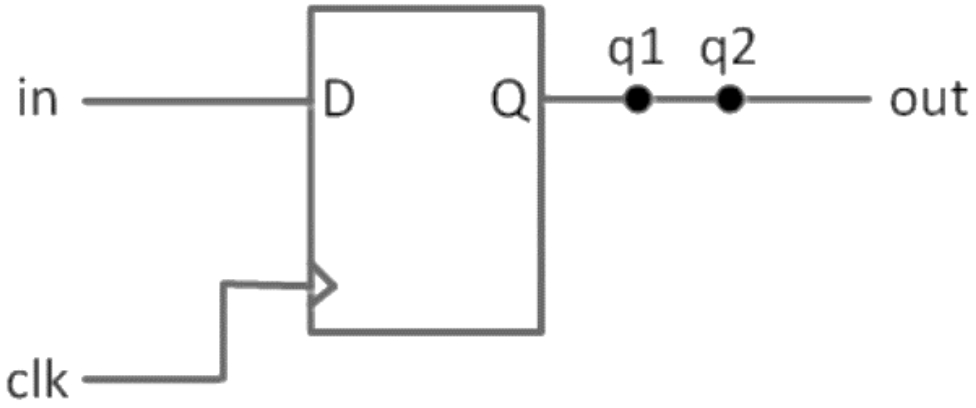
- Blocking assignments do not reflect the intrinsic behaviour of multi-stage sequential logic
- Use non-blocking assignments for sequential always blocks

Behavioral Modeling

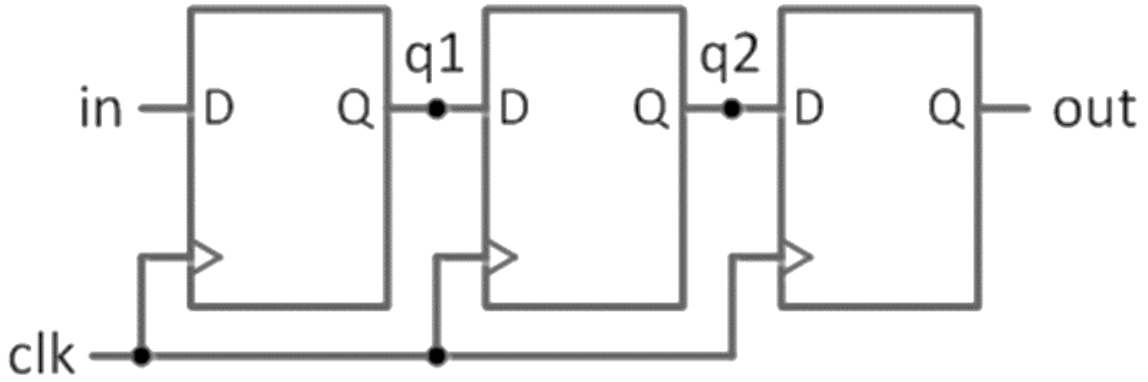



```
module blocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2 ;
   always @(posedge clk)
      begin
         q1 = in ;
         q2 = q1 ;
         out = q2 ;
      end
endmodule
```

"at each rising clock edge, q1 = in,
after that, q2 = q1 = in
after that, out = q2 = q1 = in
Therefore, out = in"



Behavioral Modeling

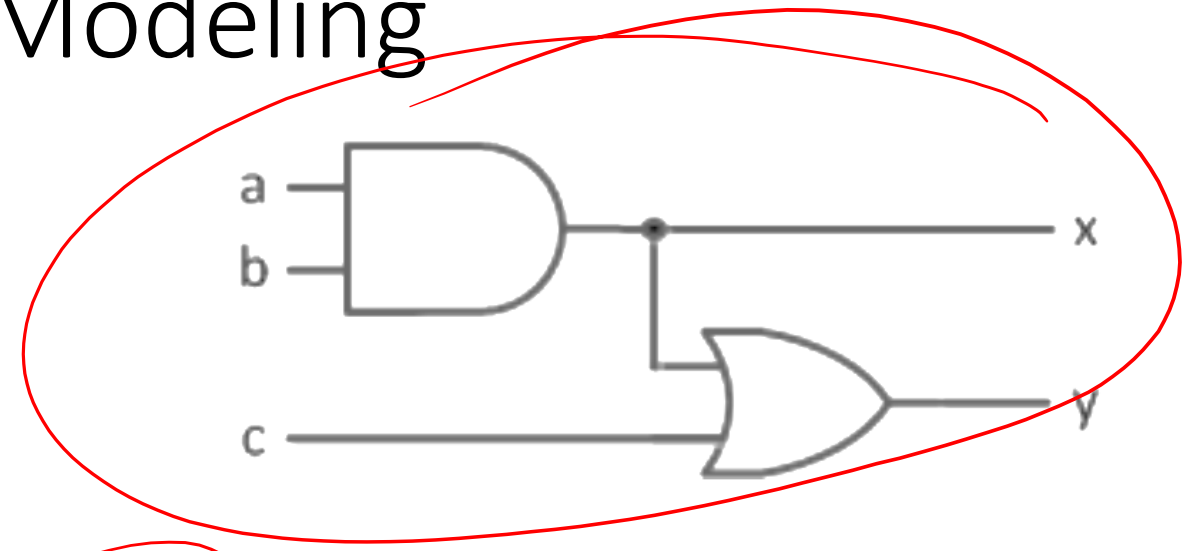



```
module nonblocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2;
   always @(posedge clk)
      begin
         q1 <= in ;
         q2 <= q1 ;
         out <= q2 ;
      end
endmodule
```

"at each rising clock edge, q1, q2 and out simultaneously receive the old values of in, q1 and q2. Therefore, out = q2"

- Blocking assignments do not reflect the intrinsic behaviour of multi-stage sequential logic
- Use non-blocking assignments for sequential always blocks

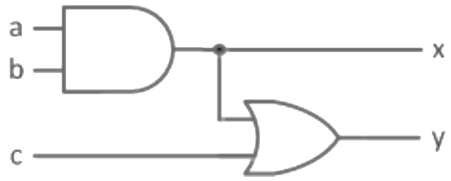
Behavioral Modeling




```
module blocking (a, b, c, x, y) ;
    input a, b, c ;
    output reg x, y ;
    always @ (a or b or c)
       begin
          x = a & b ;
          y = x | c ;
       end
endmodule
```

```
module nonblocking (a, b, c, x, y) ;
    input a, b, c ;
    output reg x, y ;
    always @ (a or b or c)
       begin
          x <= a & b ;
          y <= x | c ;
       end
endmodule
```

Behavioral Modeling




```
module blocking (a, b, c, x, y) ;
    input a, b, c ;
    output reg x, y ;
    always @ (a or b or c)
      begin
        x = a & b ;
        y = x | c ;
      end
endmodule
```

```
module nonblocking (a, b, c, x, y) ;
    input a, b, c ;
    output reg x, y ;
    always @ (a or b or c)
      begin
        x <= a & b ;
        y <= x | c ;
      end
endmodule
```

- Given initial conditions:

  a=1, b=1, c=0, x=1, y=1.
- a changes to 0. always block triggered.
- Blocking behaviour of simulator: 1st calculates x = a & b = 0. Then calculates y = x | c = 0
- Non-Blocking behavior of simulator: Concurrently calculates x(new) = a & b = 0. y = x(old) | c = 1
- Non-blocking assignment *do not* reflect the intrinsic behavior of multi-stage combinational logic
- While non-blocking assignments can be hacked to simulate correctly (expand sensitivity list), its not elegant
- **Guideline**: Use blocking assignments for combinational always blocks

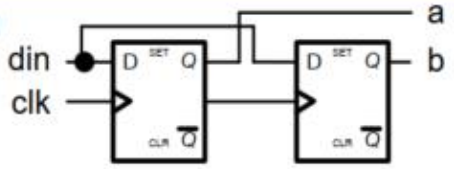
Behavioral Modeling

- When modelling sequential logic, use non-blocking assignments.
- When modelling combinational logic with an always block, use blocking assignments.
- When modelling both sequential and combinational logic within the same always block, use non-blocking assignments.
- Do not mix blocking and non-blocking assignments in the same always block.

Behavioral Modeling (Self Study)

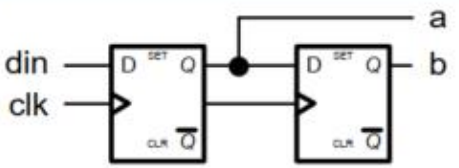
Blocking assignments (1/2)

```
always @ (posedge clk)  begin
   a = din;
   b = a;
end
```



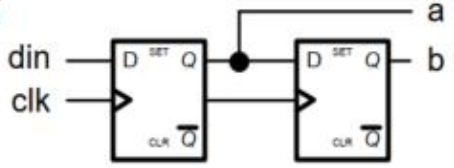
Non-blocking assignments (1/2)

```
always @ (posedge clk)  begin
   a <= din;
   b <= a;
end
```



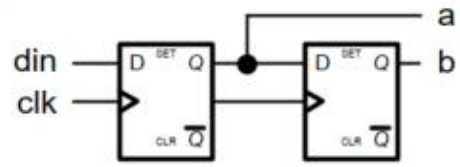
Blocking assignments (2/2)

```
always @ (posedge clk)  begin
   b = a;
   a = din;
end
```



Non-blocking assignments (2/2)

```
always @ (posedge clk)  begin
   b <= a;
   a <= din;
end
```

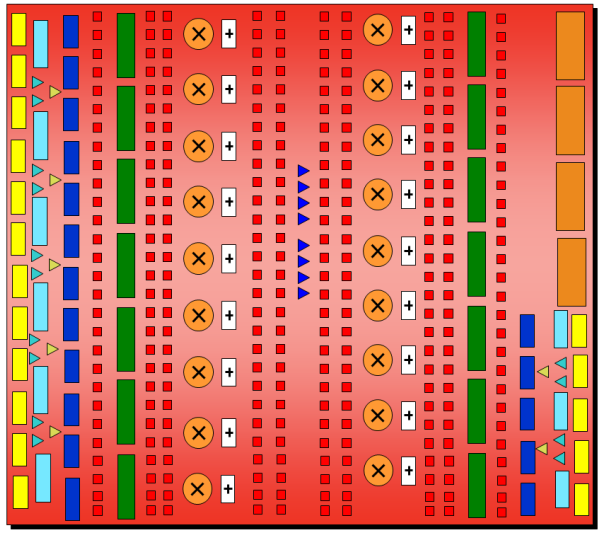


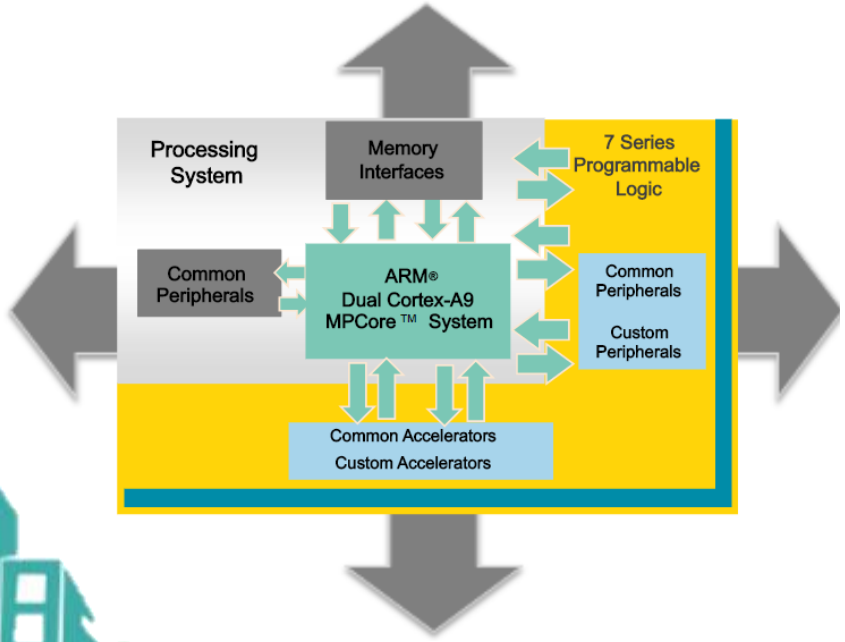
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

SDK
Software Development Kit
VIVADO™

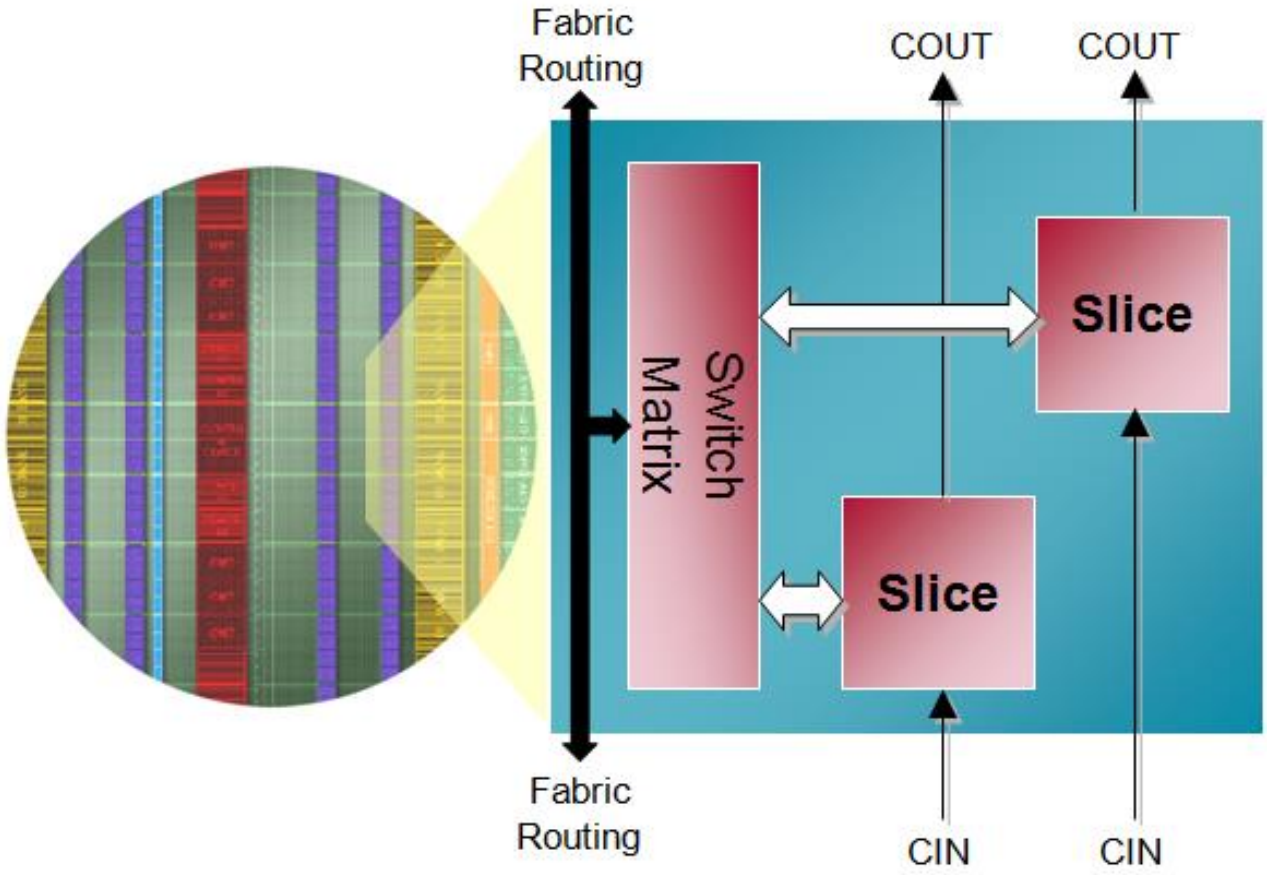




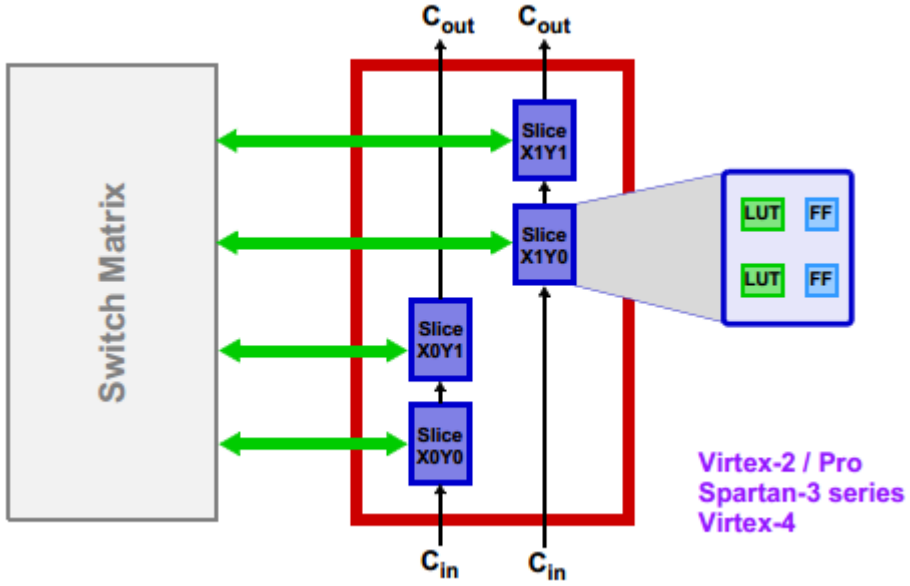
FPGA Architecture

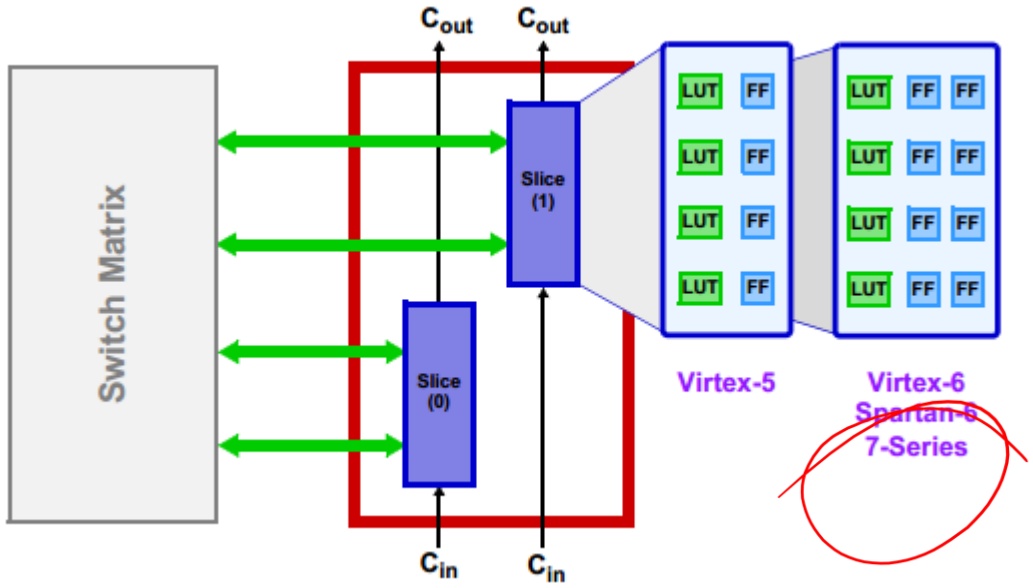
Configurable Logic Block (CLB)

- 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)

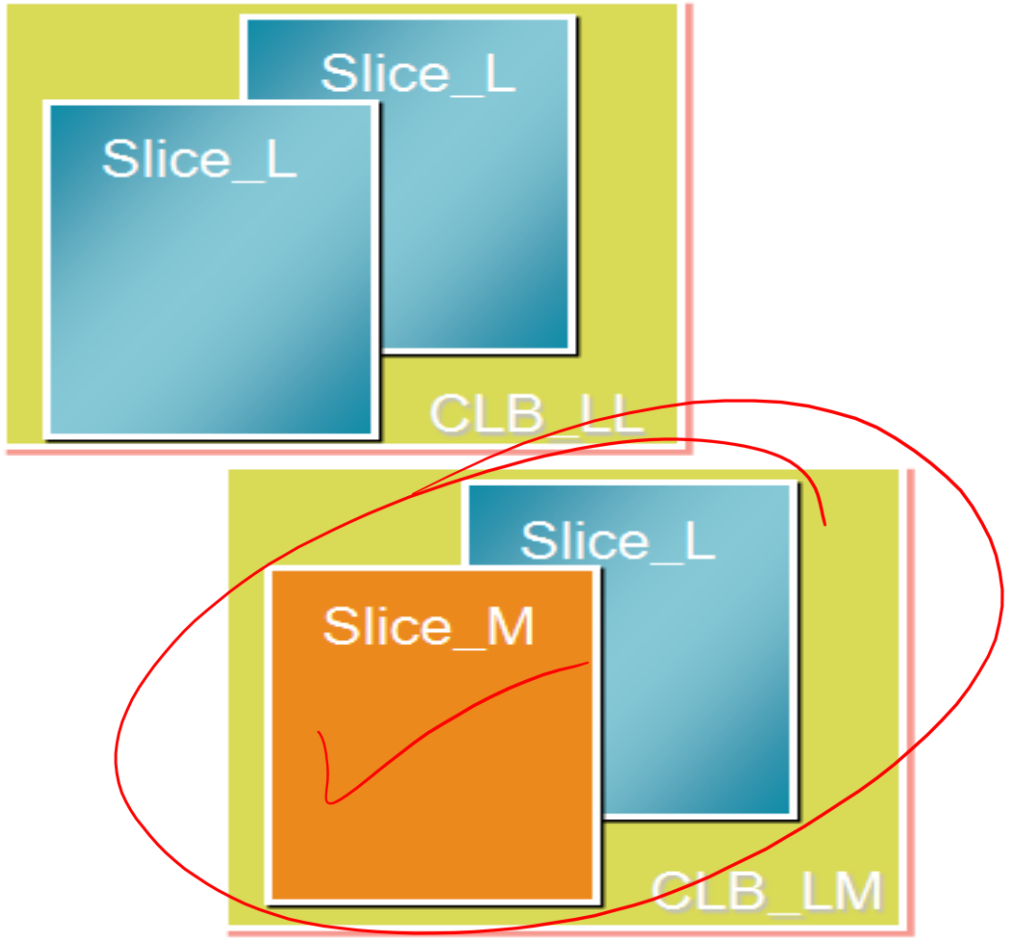




Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains
2	8	16	2

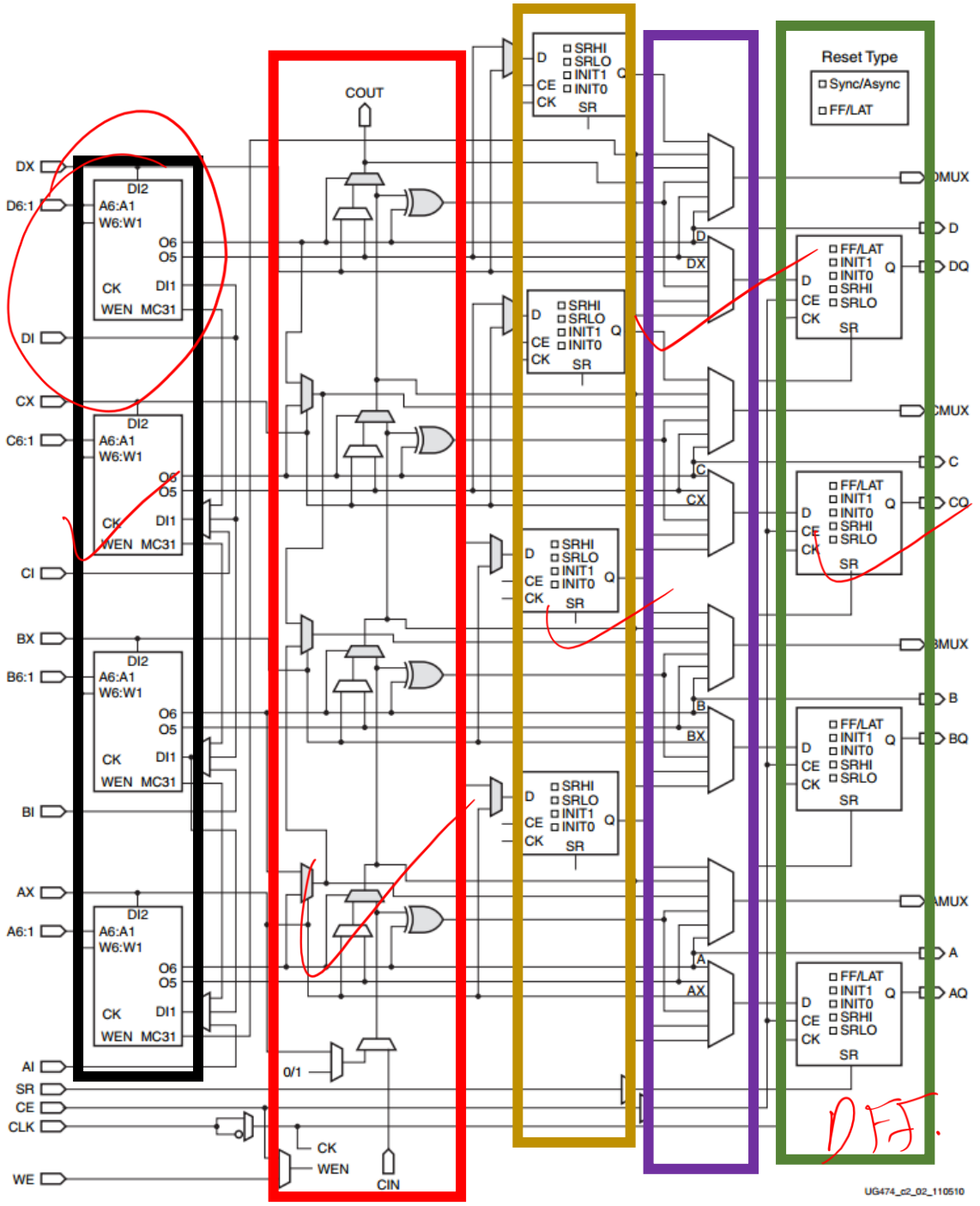
Types of CLB Slices

- **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.

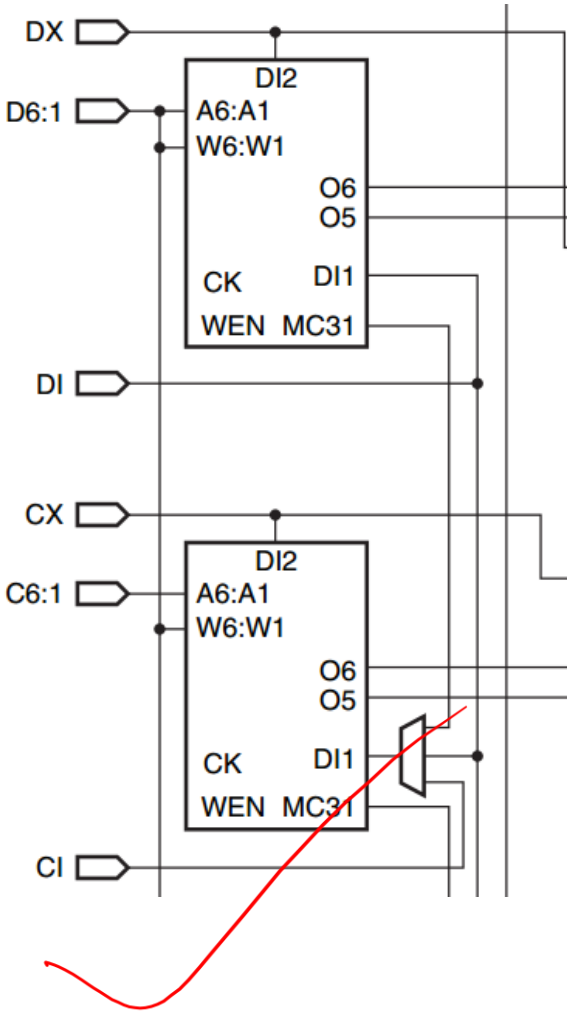


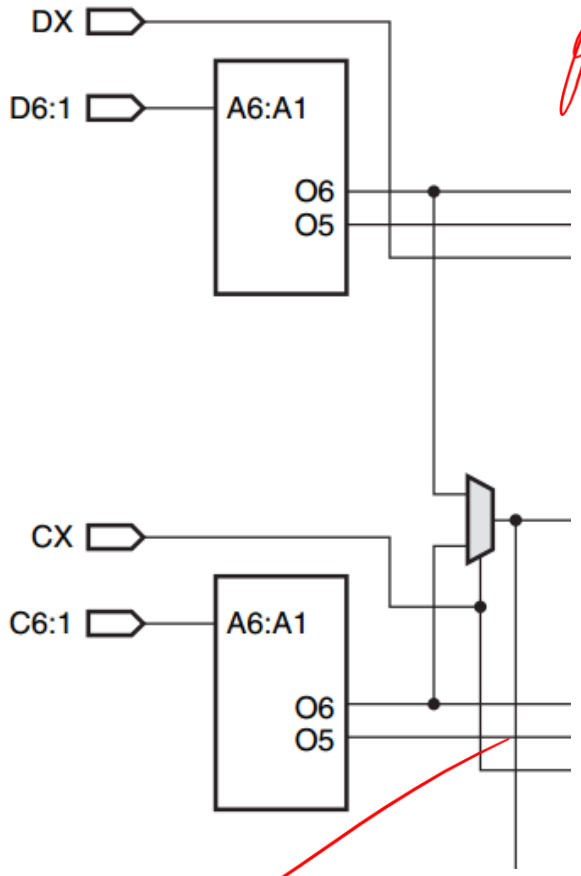
Slice Resource

- 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.



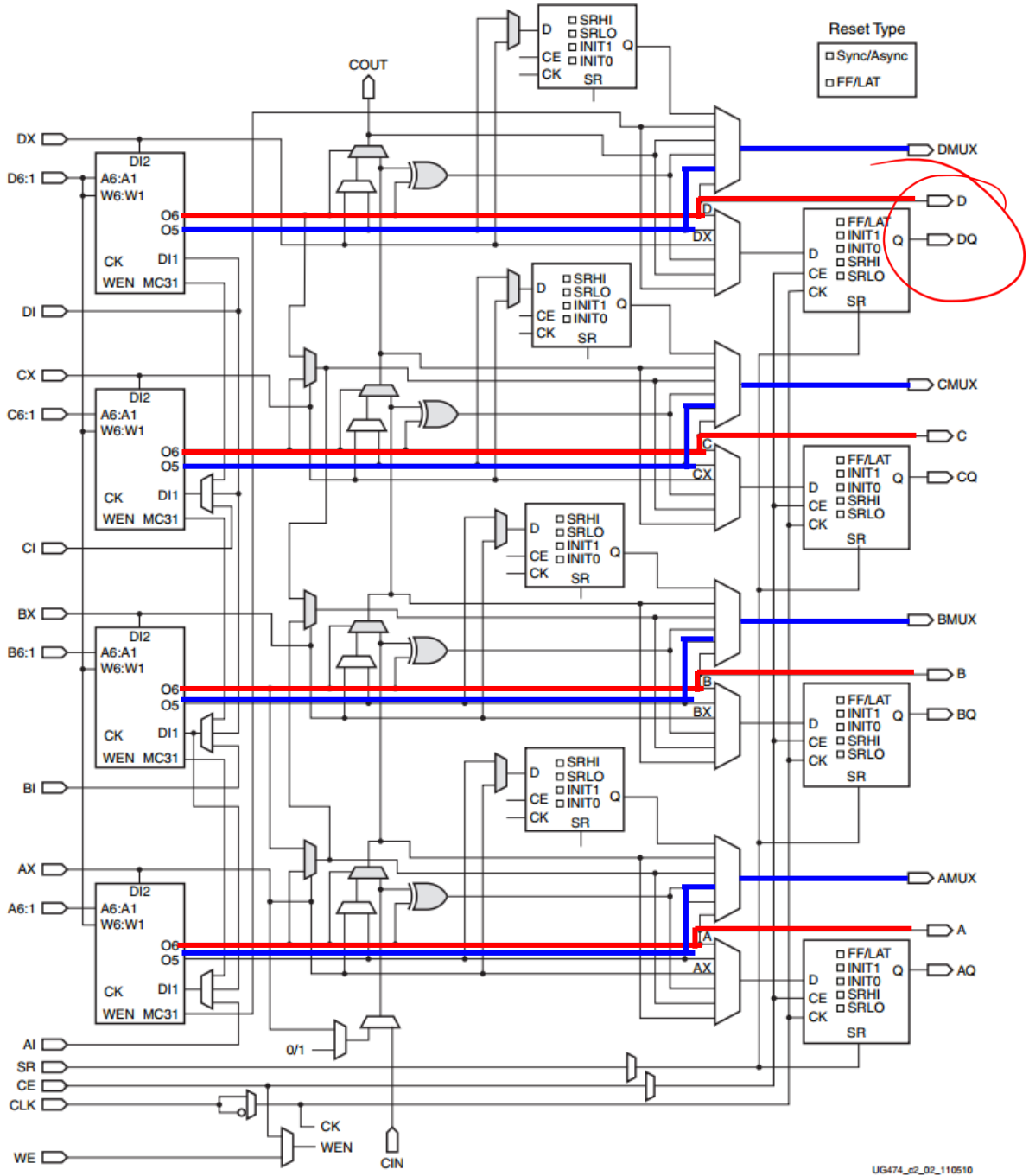
SLICEM Vs SLICEL





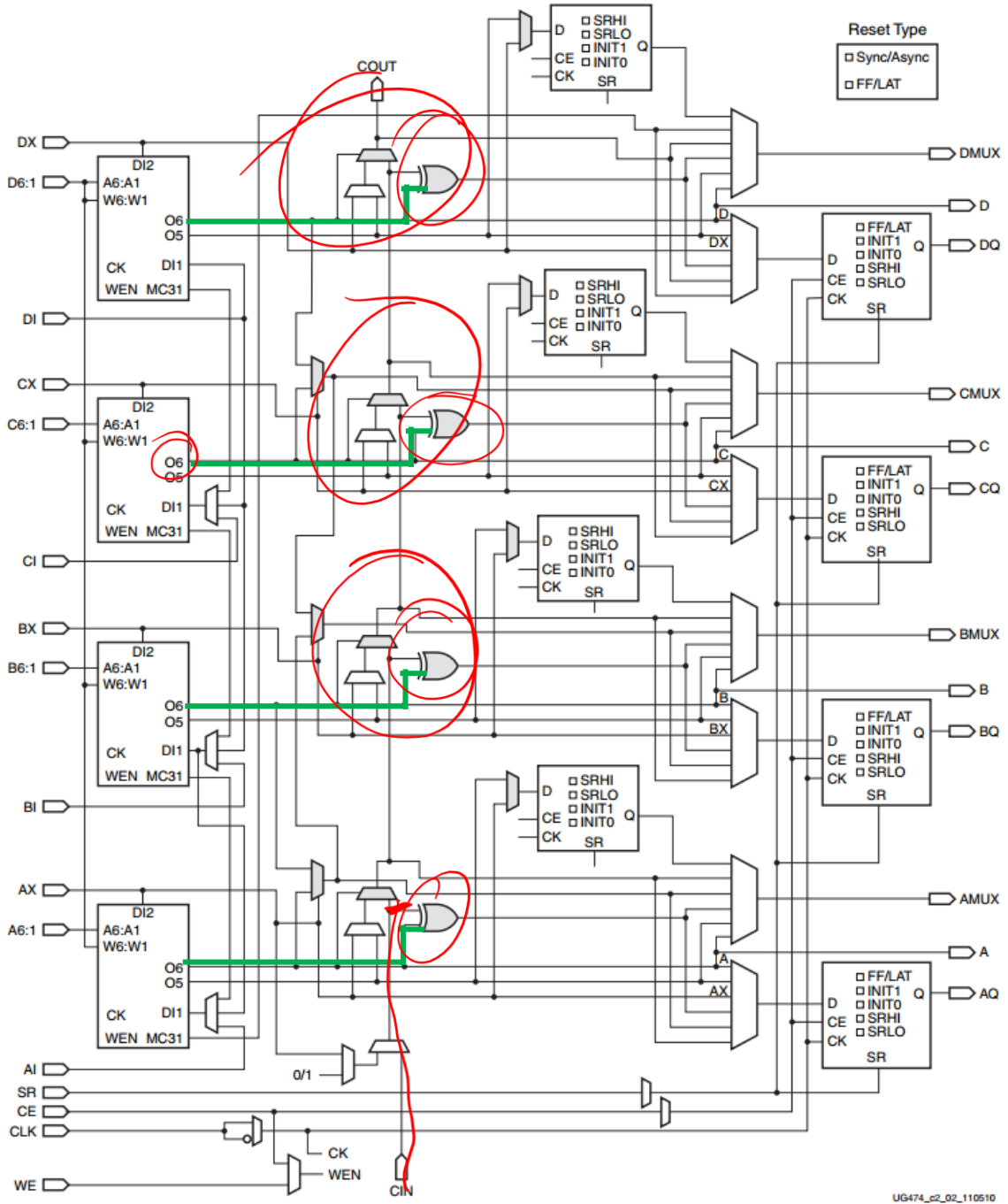
LUT

- Signals from the LUT can:
  - Exit the slice (through A, B, C, D output for O6 or AMUX, BMUX, CMUX, DMUX output for O5)



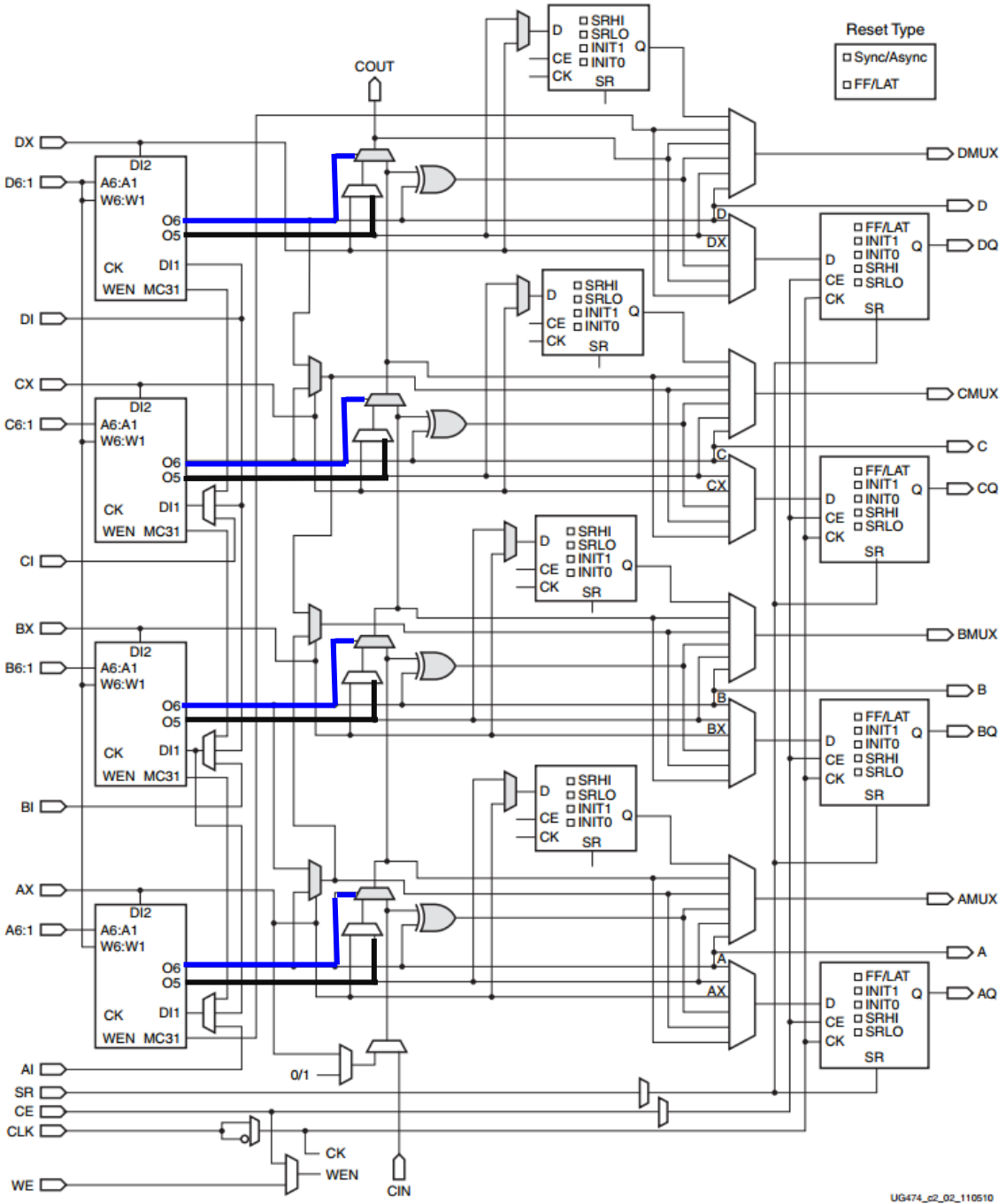
LUT

- Signals from the LUT can:
  - Exit the slice (through A, B, C, D output for O6 or AMUX, BMUX, CMUX, DMUX output for O5)
  - Enter the XOR dedicated gate from an O6 output



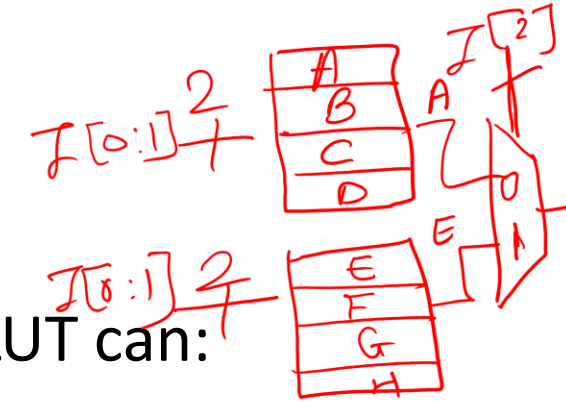
LUT

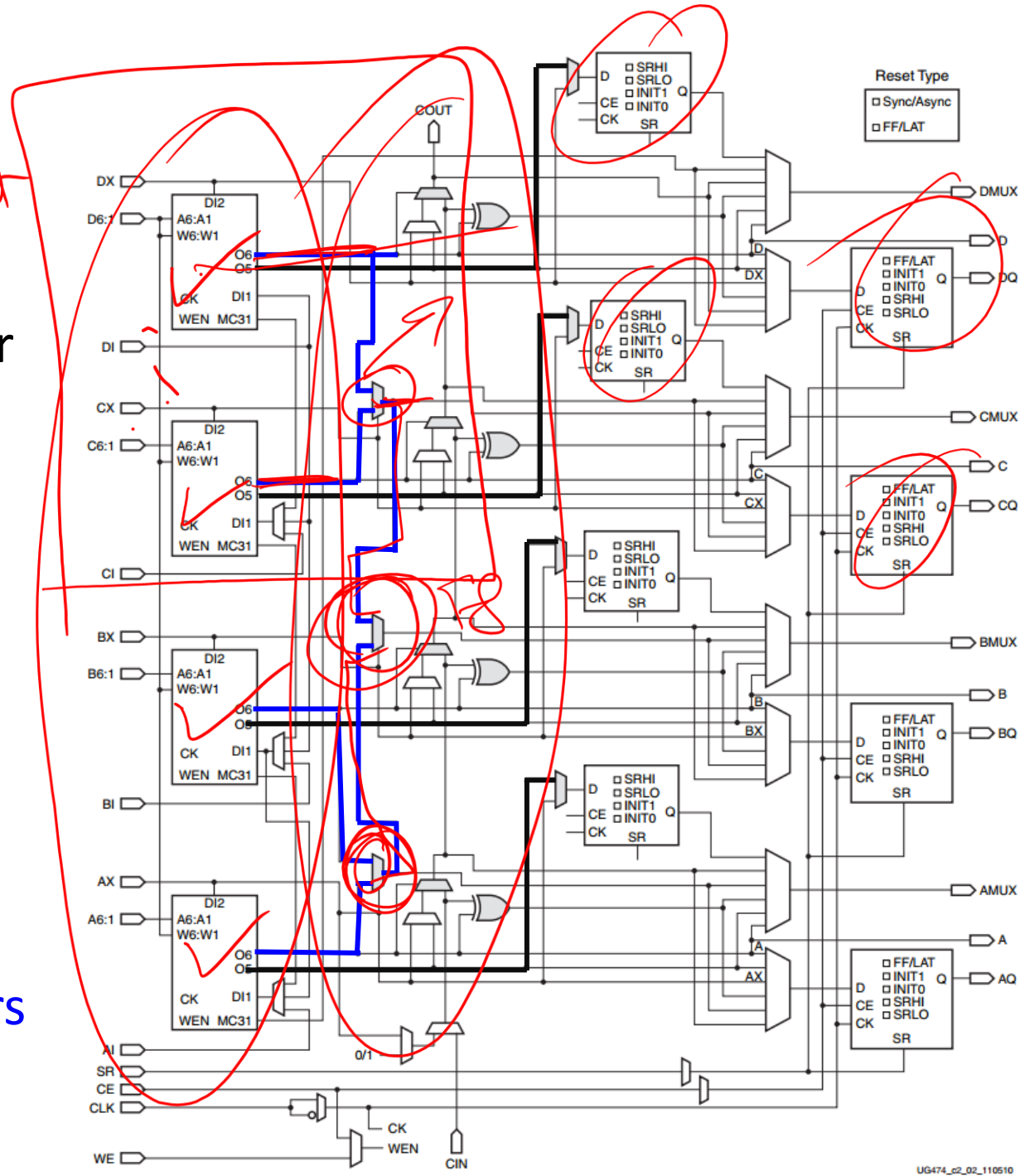
- Signals from the LUT can:
  - Exit the slice (through A, B, C, D output for O6 or AMUX, BMUX, CMUX, DMUX output for O5)
  - Enter the XOR dedicated gate from an O6 output
- **Enter the carry-logic chain from an O5 output**
- Enter the select line of the carry-logic multiplexer from O6 output



LUT

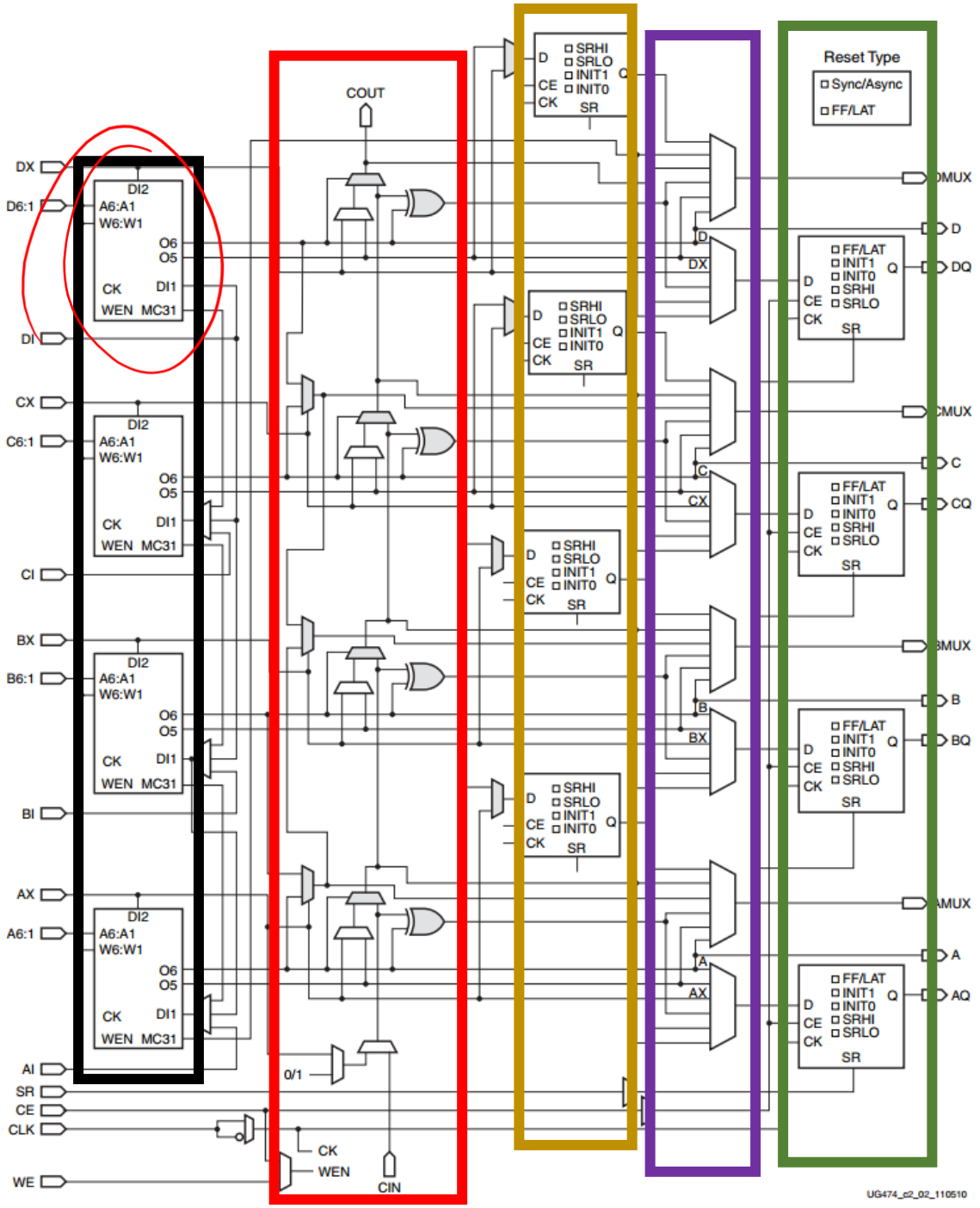
- Signals from the LUT can:
- Exit the slice (through A, B, C, D output for O6 or AMUX, BMUX, CMUX, DMUX output for O5)
- Enter the XOR dedicated gate from an O6 output
- Enter the carry-logic chain from an O5 output
- Enter the select line of the carry-logic multiplexer from O6 output
- **Feed the D input of the storage element**
- Go to F7AMUX/F7BMUX wide multiplexers from O6 output





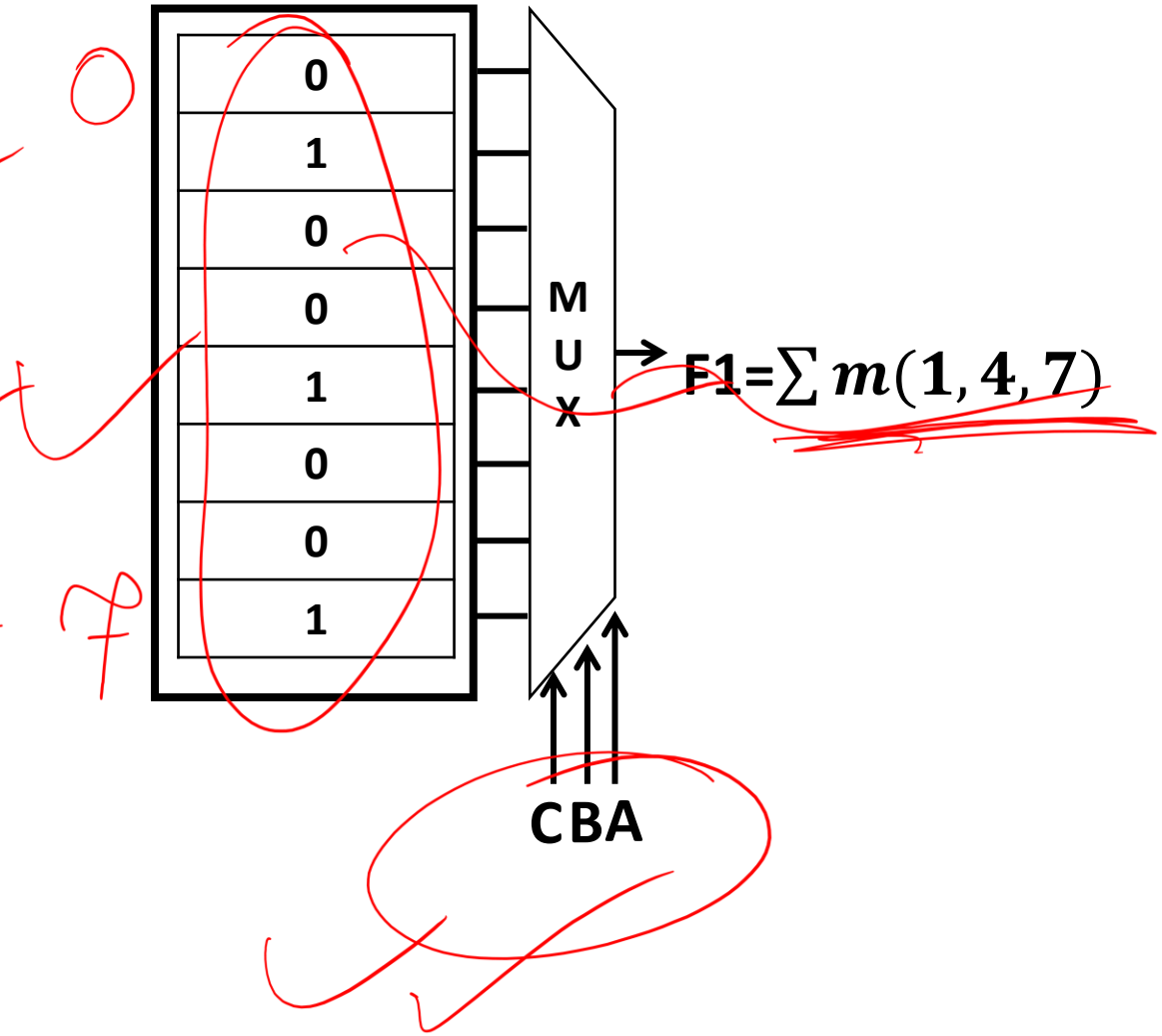
Slice Resource

- 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.



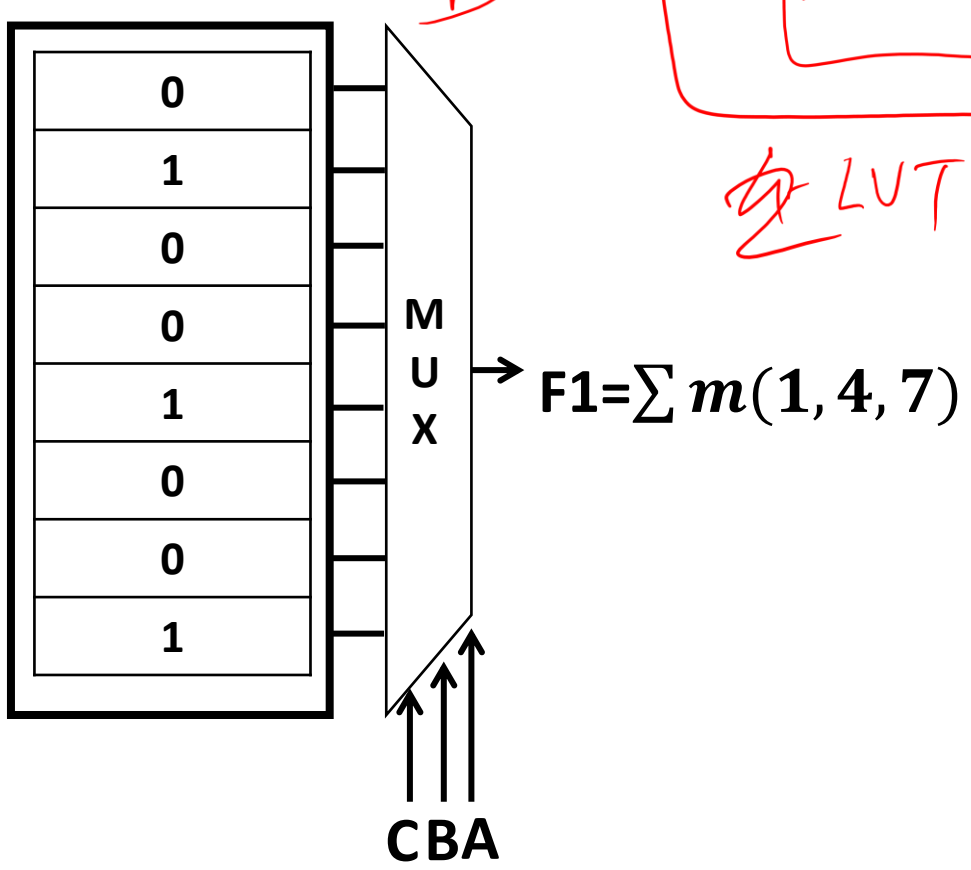
3 Input LUT

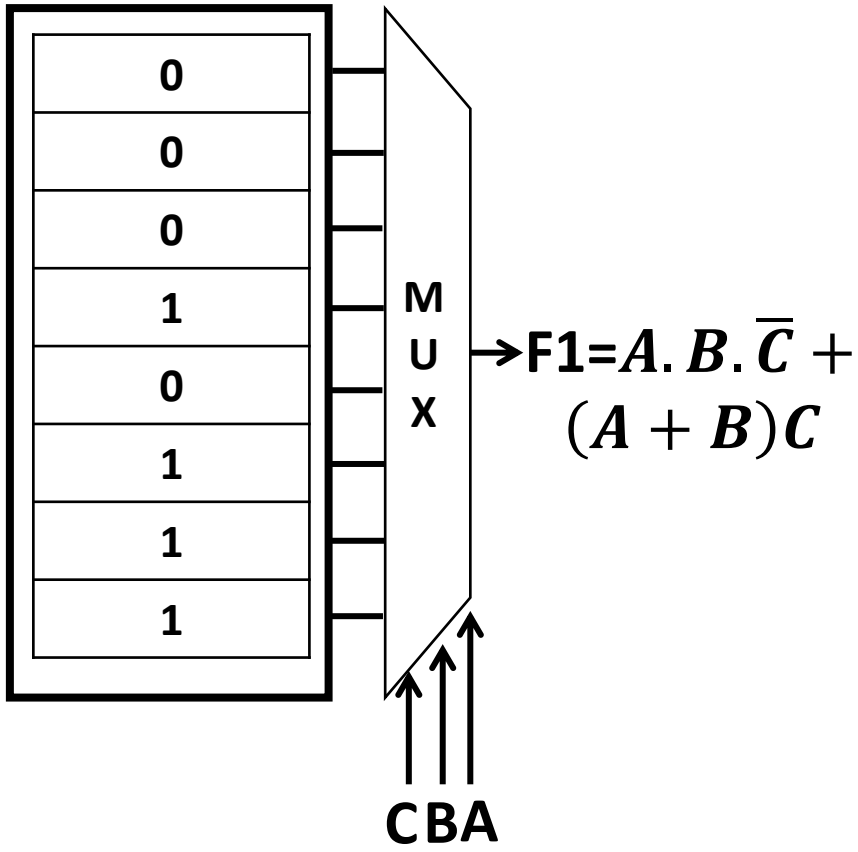
C	B	A	F1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



3 Input LUT

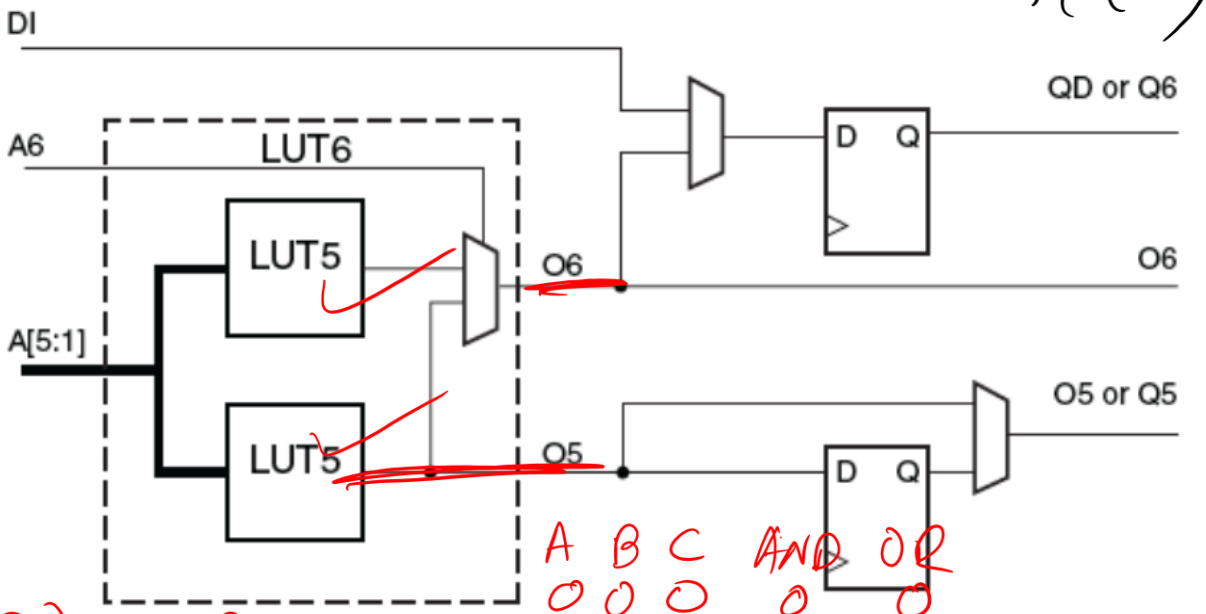

A
B
F1
F2
2 LUT


0
1
0
0
1
0
0
1
MUX
F1=∑ m(1, 4, 7)
CBA


0
0
0
1
0
1
1
1
MUX
F1=A. B. C̄ + (A + B)C
CBA

LUT

- There are six independent inputs (A inputs - A1 to A6) and two independent outputs (O5 and O6)



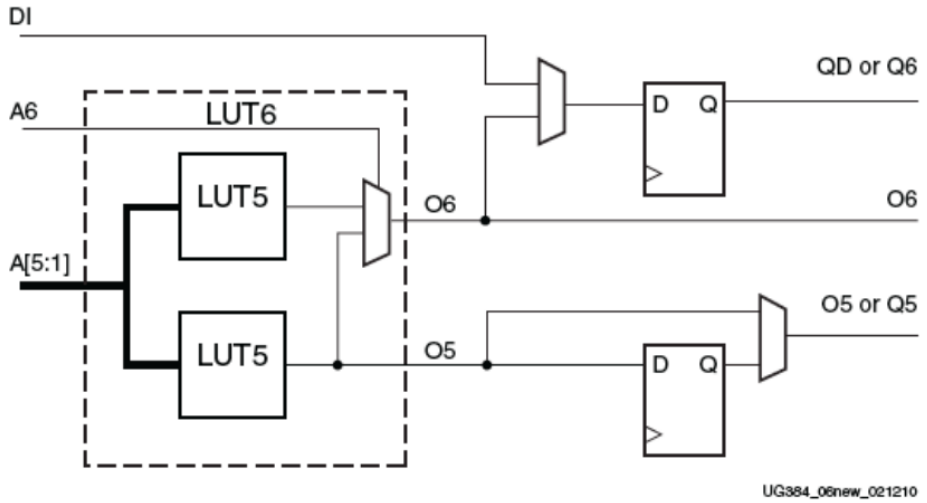
3-Input LUT for Logic

- Any single arbitrarily defined 3-input Boolean function
- A 3-input function uses: A1-A3 inputs and O3 output
- Two arbitrarily defined 2-input Boolean functions, as long as these two functions share common inputs
- Two 2-input or less functions use: A1–A2 inputs, A3 driven High, O2 and O3 outputs
- Two arbitrarily defined Boolean functions of 1 input

6-Input LUT for Logic

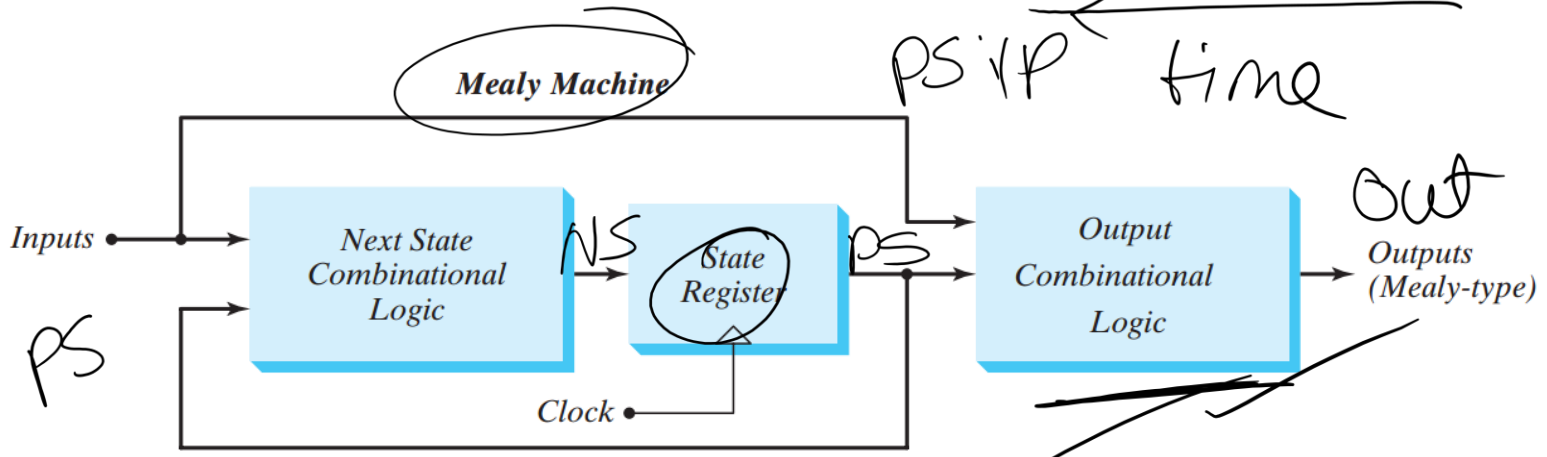
$f_1(A, B, C)$

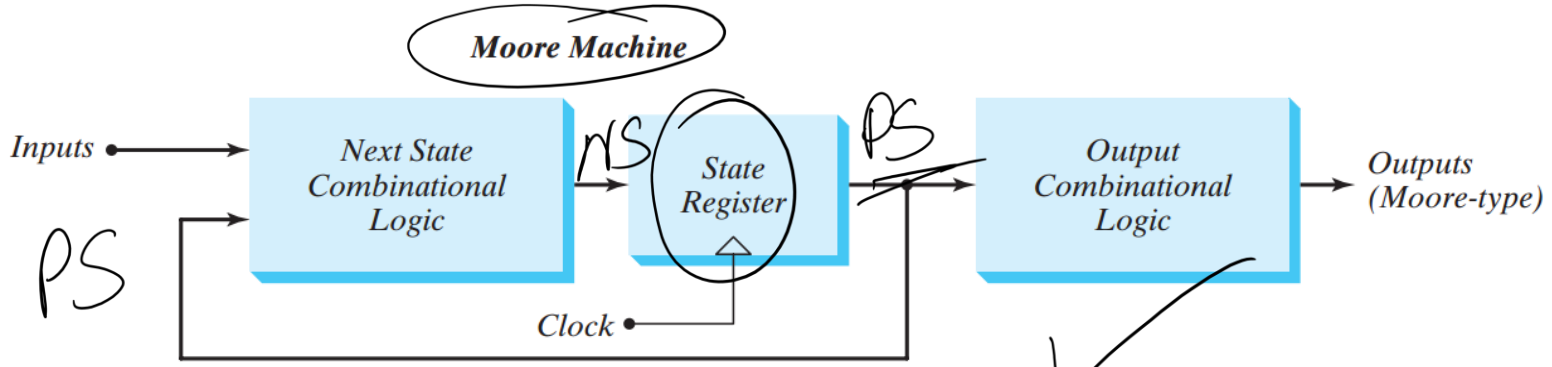
$f_2(D, E)$



- Any arbitrarily defined six-input Boolean function
- A six-input function uses: A1-A6 inputs and O6 output
- Two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs
- Two five-input or less functions use: A1–A5 inputs, A6 driven High, O5 and O6 outputs
- Two arbitrarily defined Boolean functions of 3 and 2 inputs or less

Finite State Machines



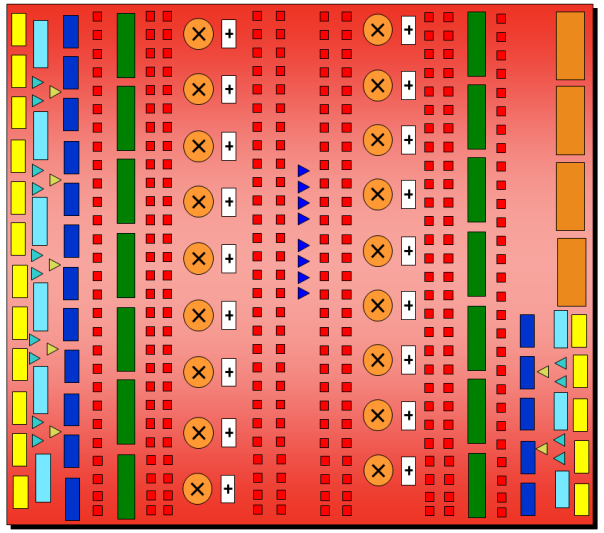


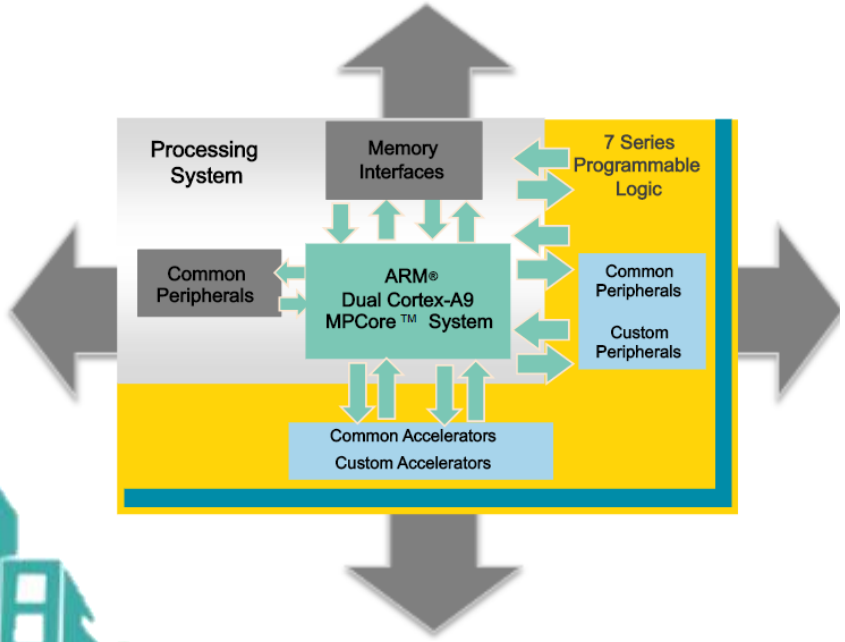
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

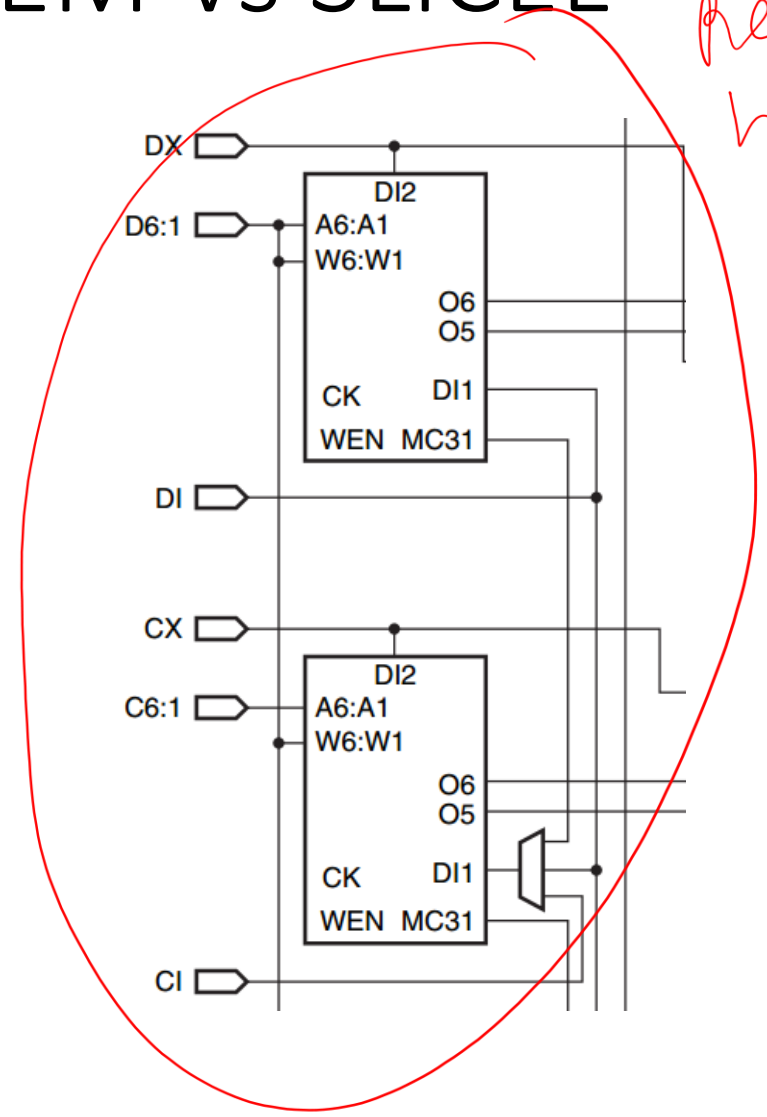
ECE 270: Embedded Logic Design

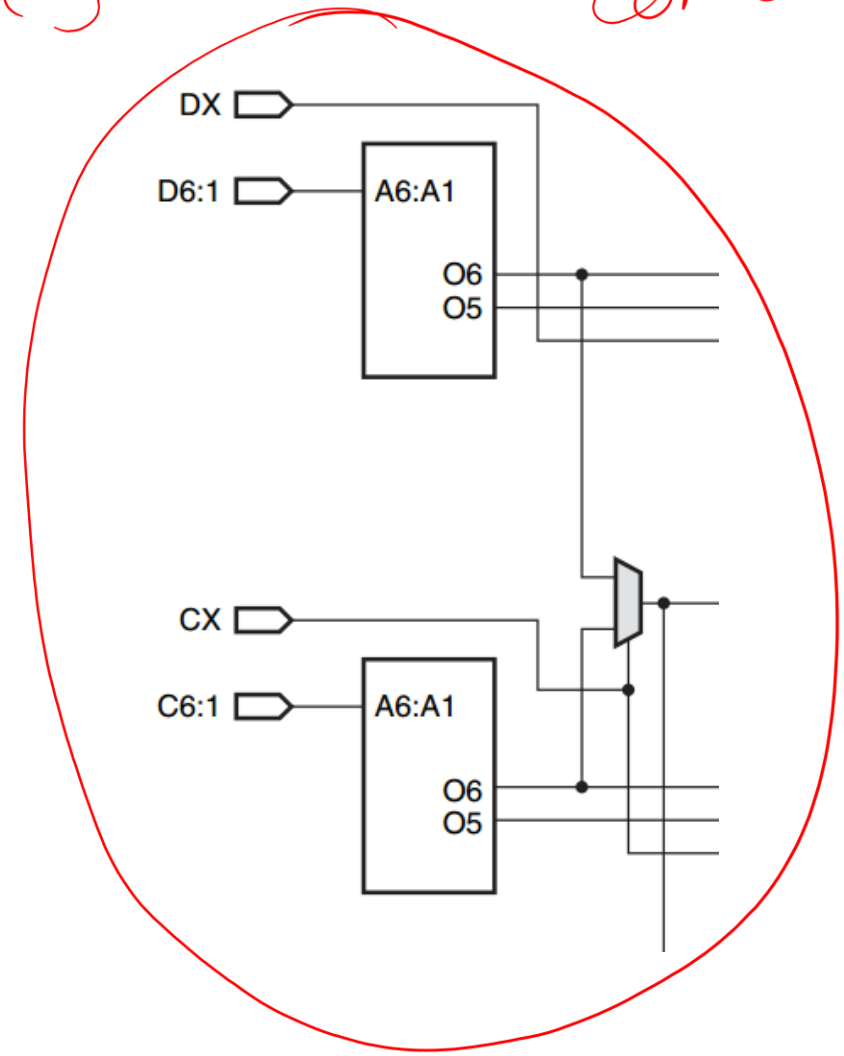
SDK
Software Development Kit
VIVADO™



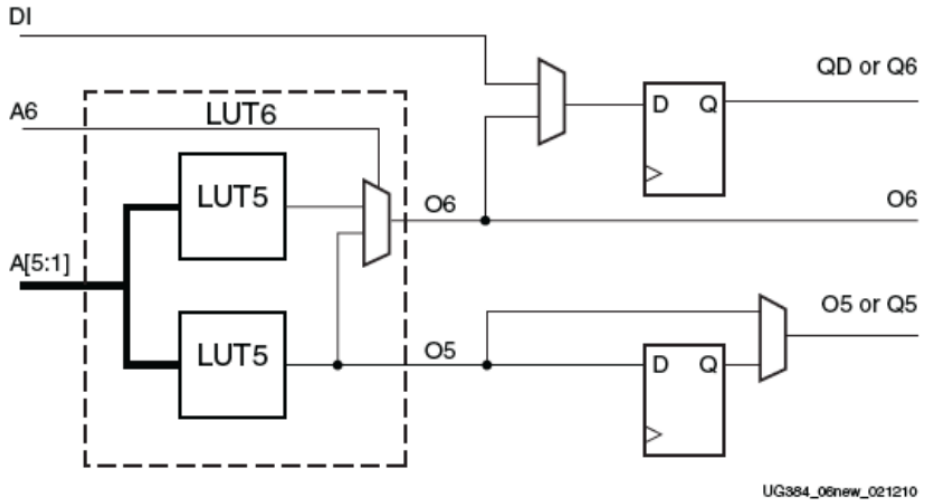


SLICEM Vs SLICEL



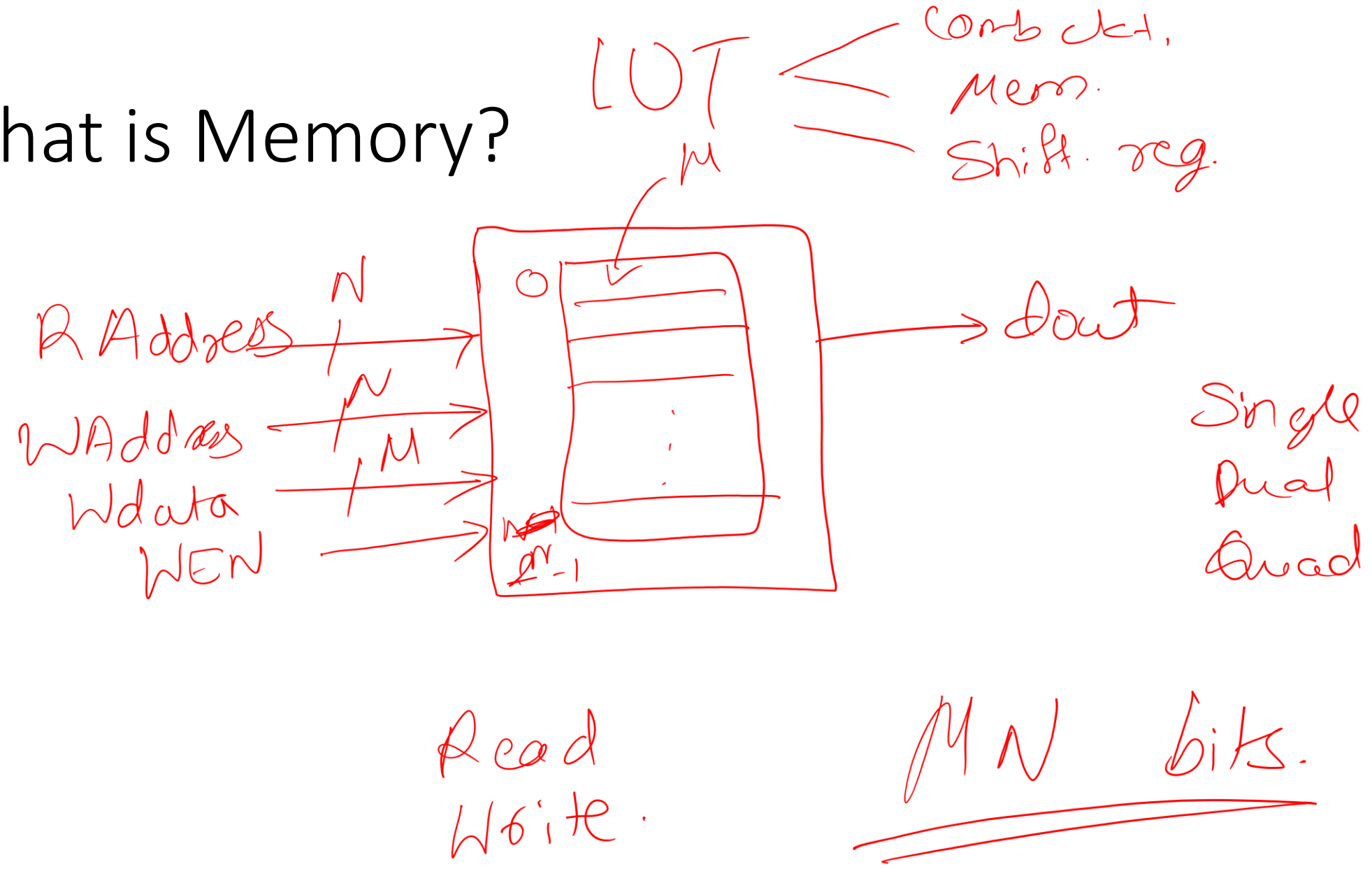


6-Input LUT for Logic



- Any arbitrarily defined six-input Boolean function
- A six-input function uses: A1-A6 inputs and O6 output
- Two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs
- Two five-input or less functions use: A1–A5 inputs, A6 driven High, O5 and O6 outputs
- Two arbitrarily defined Boolean functions of 3 and 2 inputs or less

What is Memory?

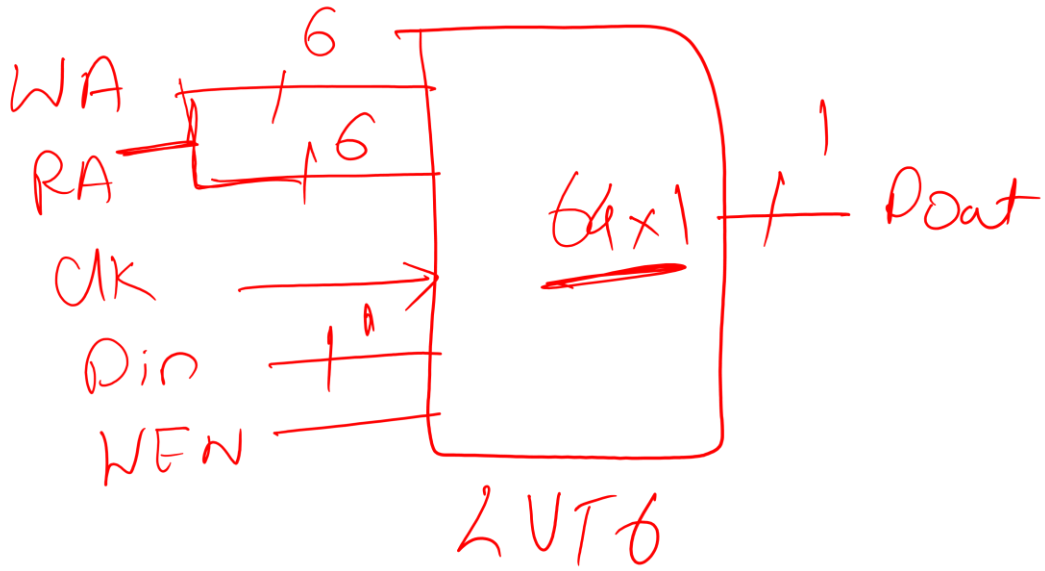


LUT as Memory

- Only in SLICEM
- Multiple LUTs in a SLICEM can be combined in various ways to store large amount of data
- Synchronous write operation, WEN must be high
- Asynchronous or synchronous read (using flip-flop in the same slice)

LUT as Memory

- **Single port:** Common address port for synch write and asynch read i.e. read and write addresses share the same address bus

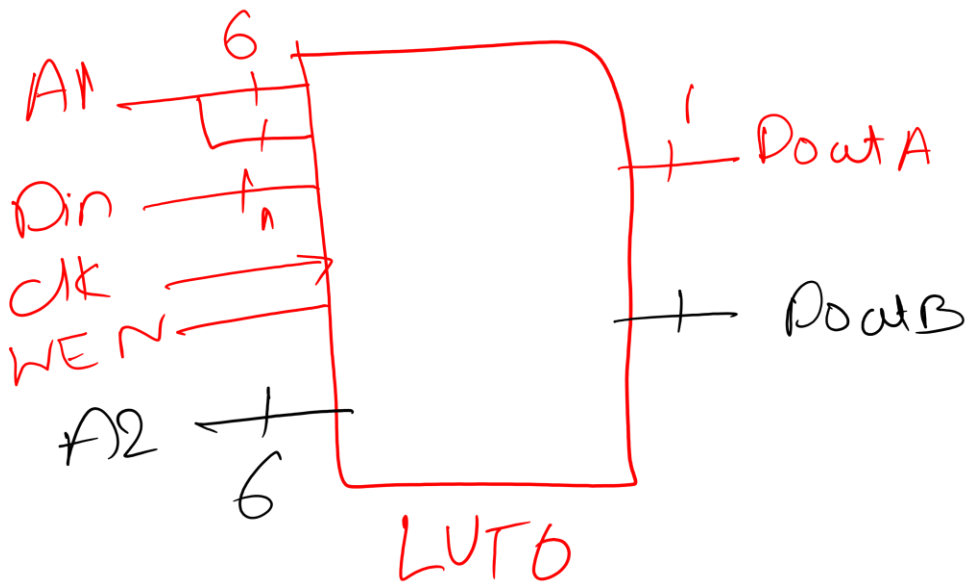


WEN=0 Read
WEN=1 Write Read.

LUT as Memory

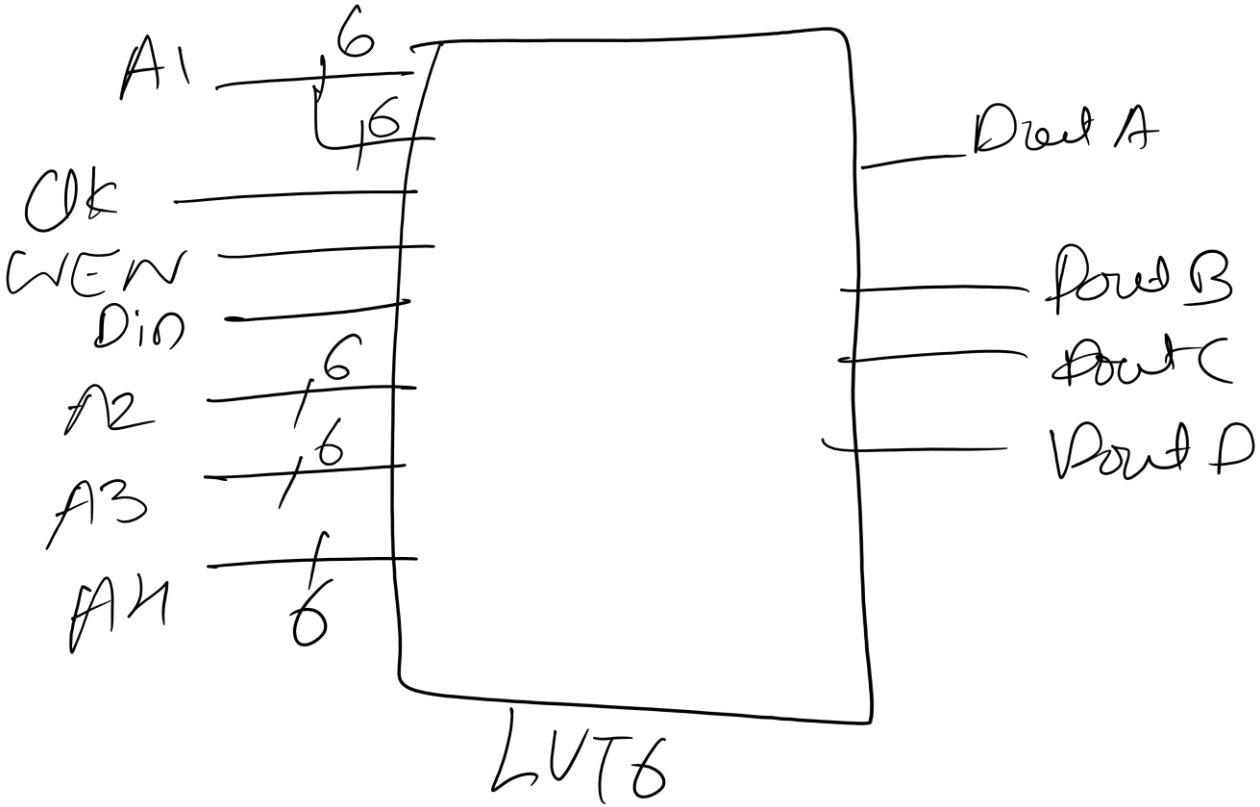
Quad Port

- **Dual port:** One port for synch write and asynch read, one port for asynch read



LUT as Memory

- **Quad port:** One port for synchronous write and asynchronous read and three ports for asynchronous reads



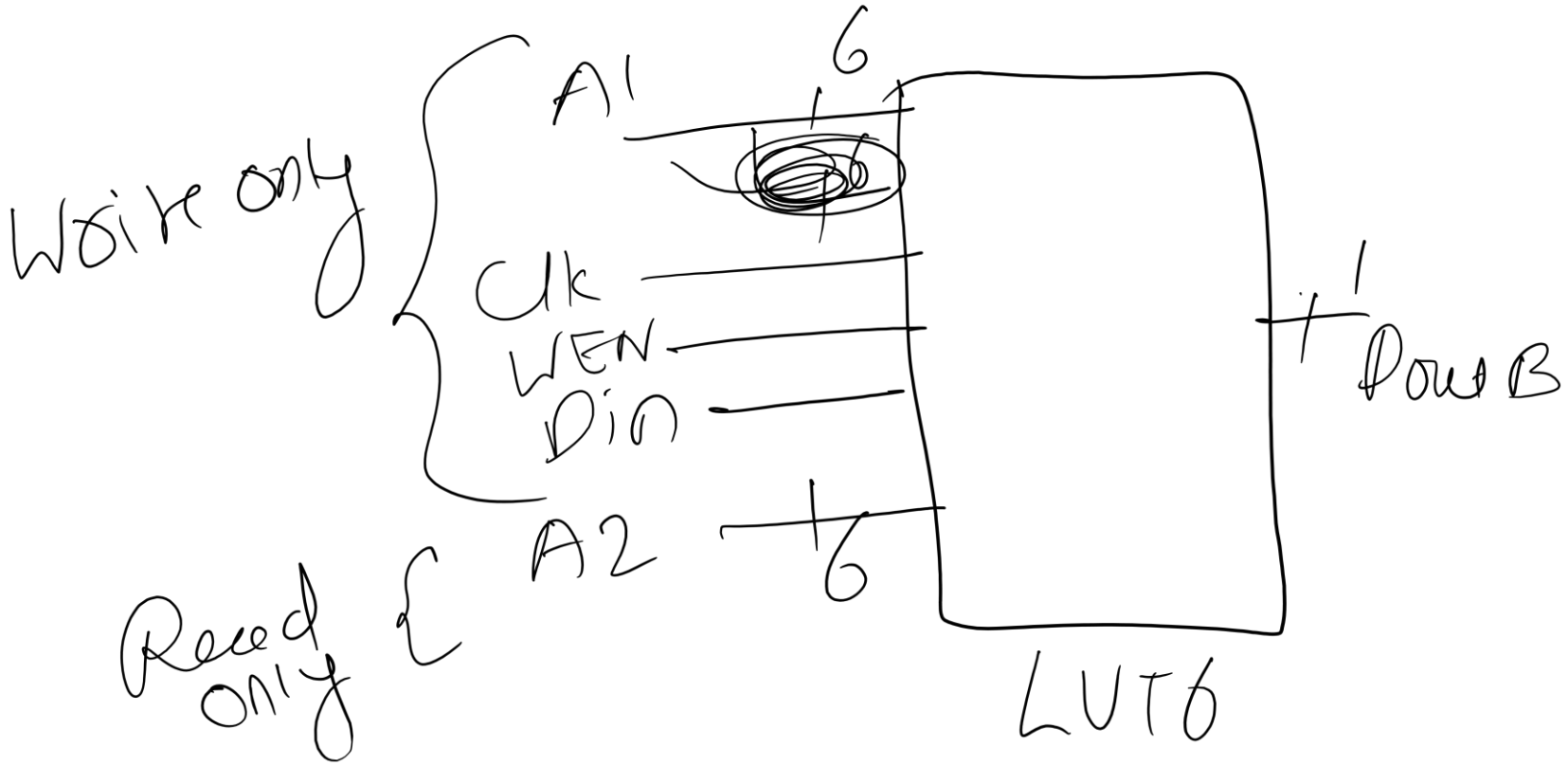
LUT as Memory

64x4 Quad Port

64x1 Single Port

128x2 Singe Port

- **Simple dual port:** One port for synchronous write (no data out/read port from the write port) and one port for asynchronous reads



1) Clock of all LUTs is common.

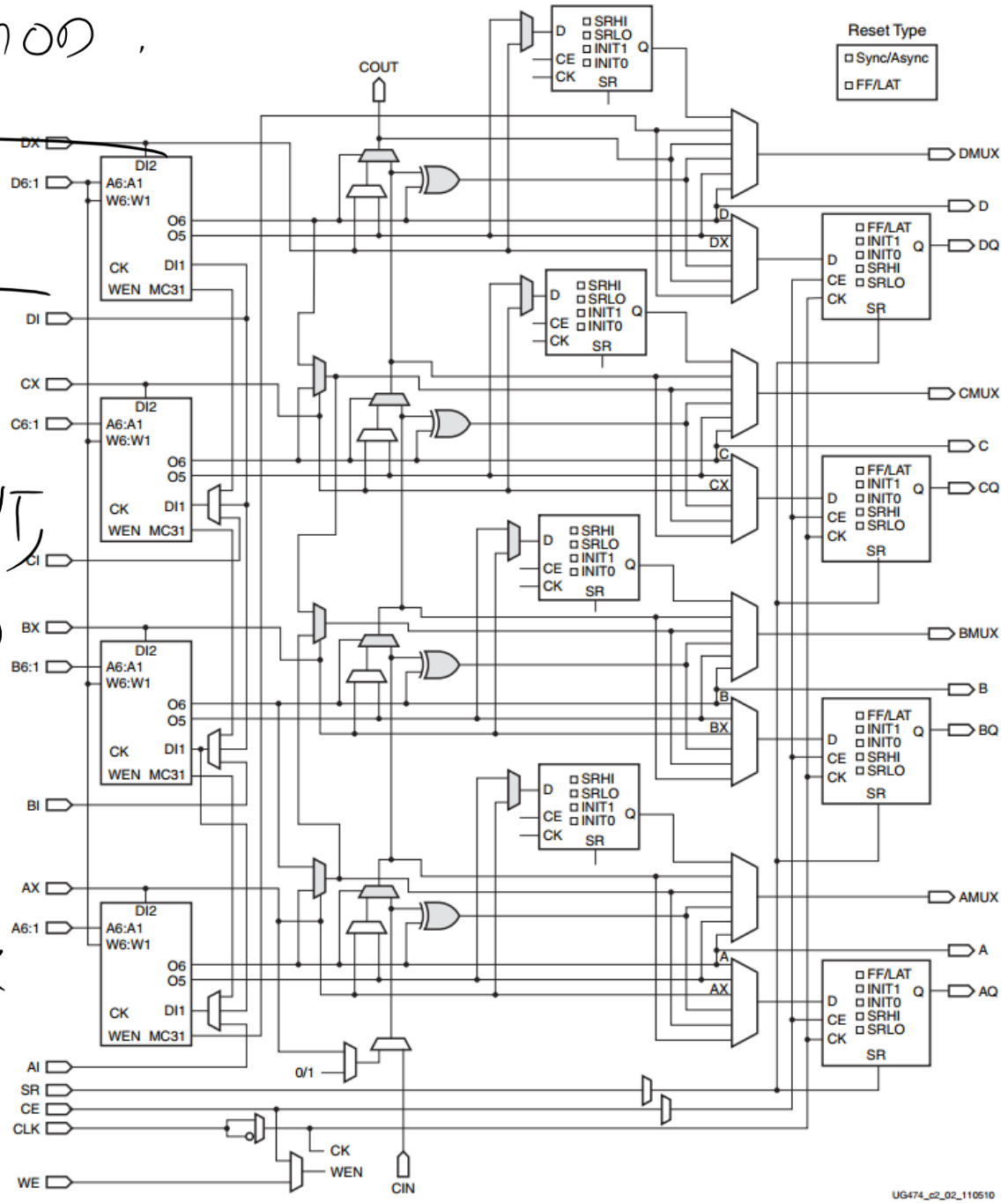
2) WEN ——o——

3) WA ——o——

← When you perform write opn on any LUT, write opn will happen on all LUTs. X

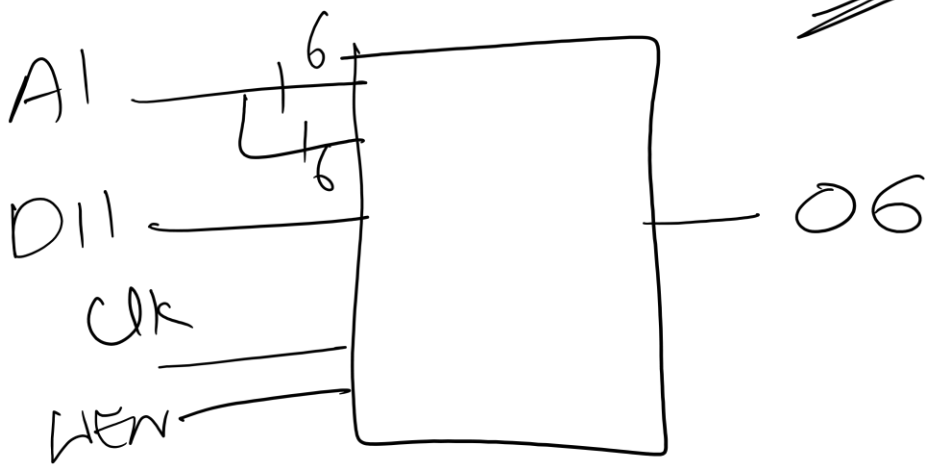
4) RA of all LUTs are diff

5) For first LUT, RA & WA are same.

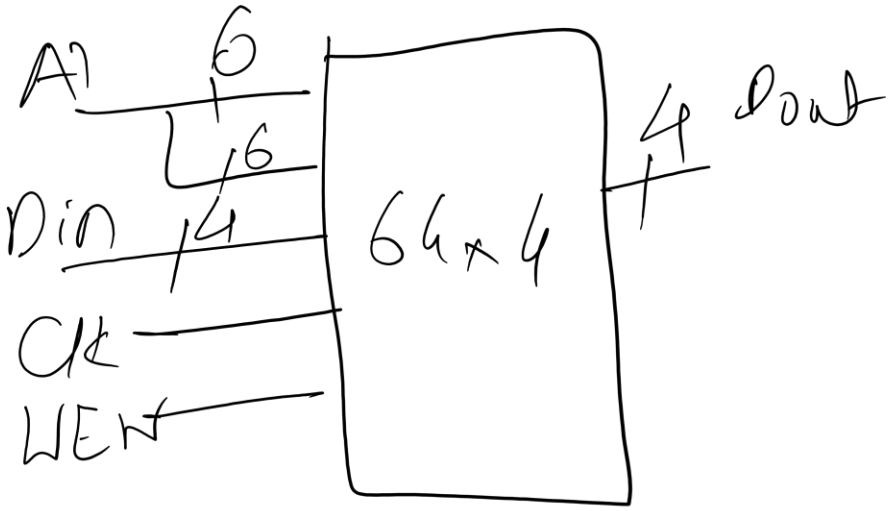


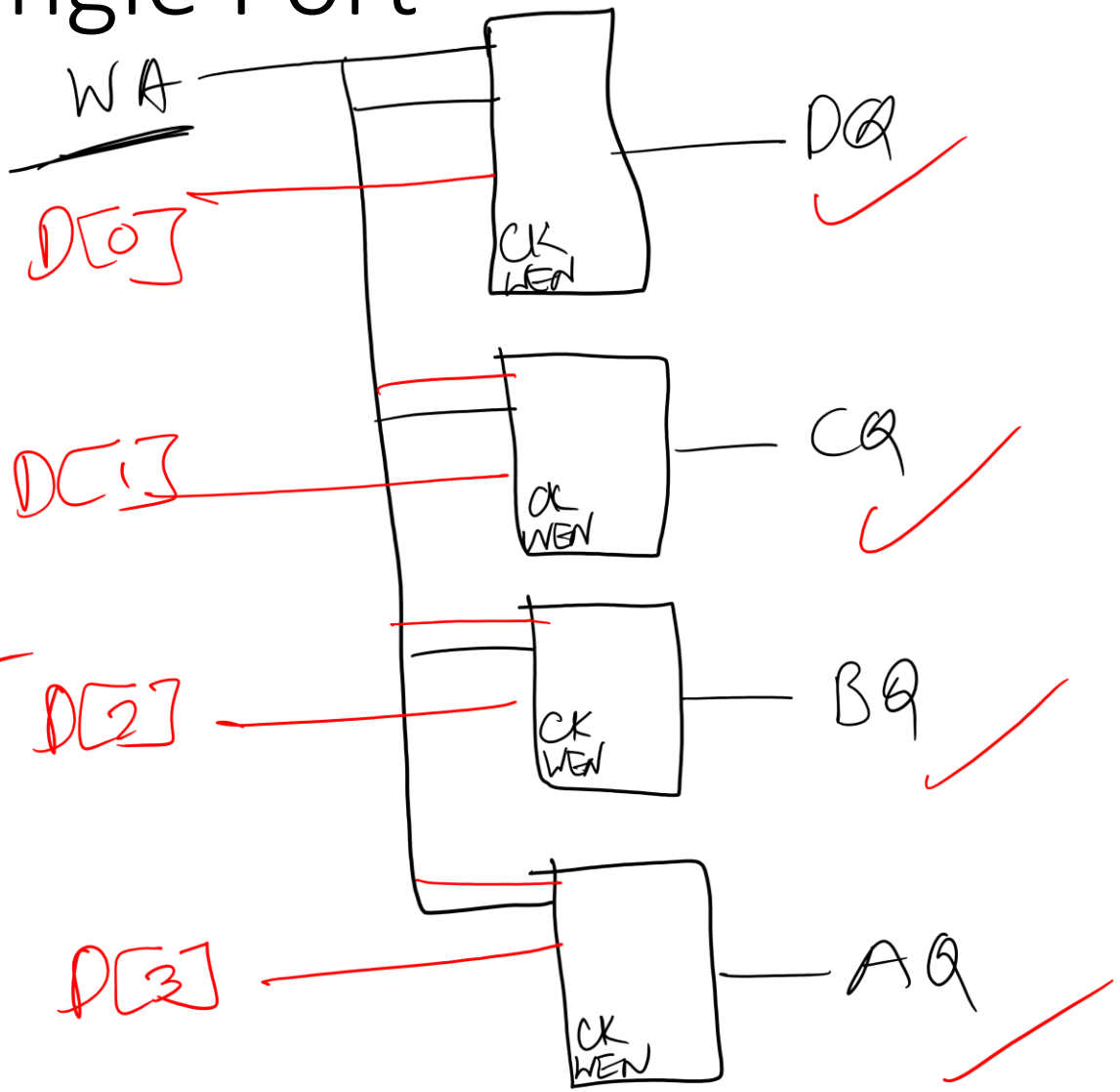
LUT as Memory: 64X1 Single Port

- How many LUTs are needed? 1



LUT as Memory: 64X4 Single Port





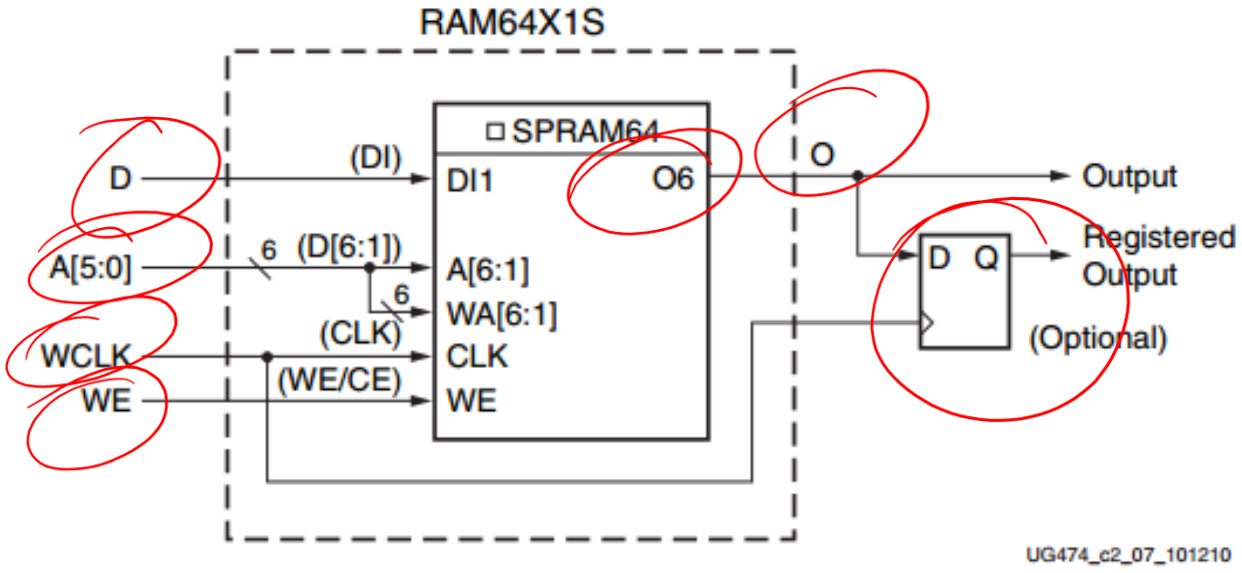
What is diff. bet^n

1) 64x4 Single port. ✓

2) four 64x1 Single port ✗

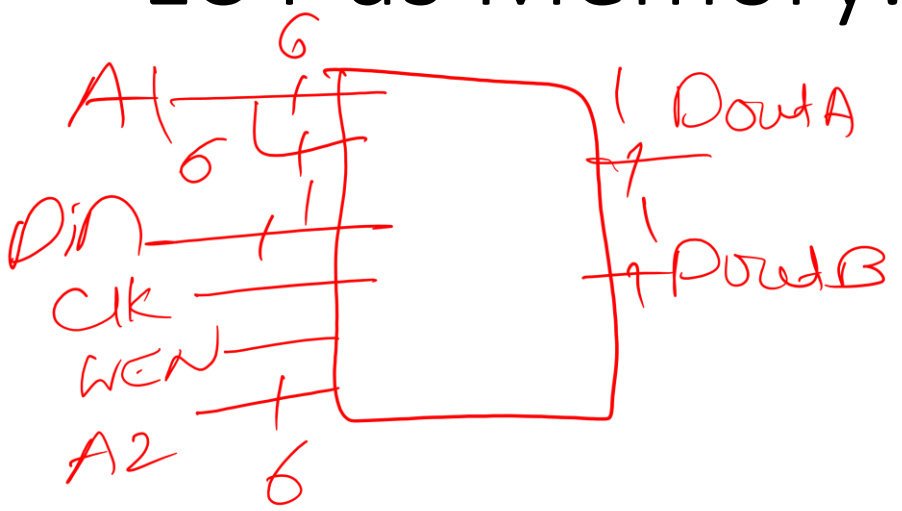
LUT as Memory: 64X1 Single Port

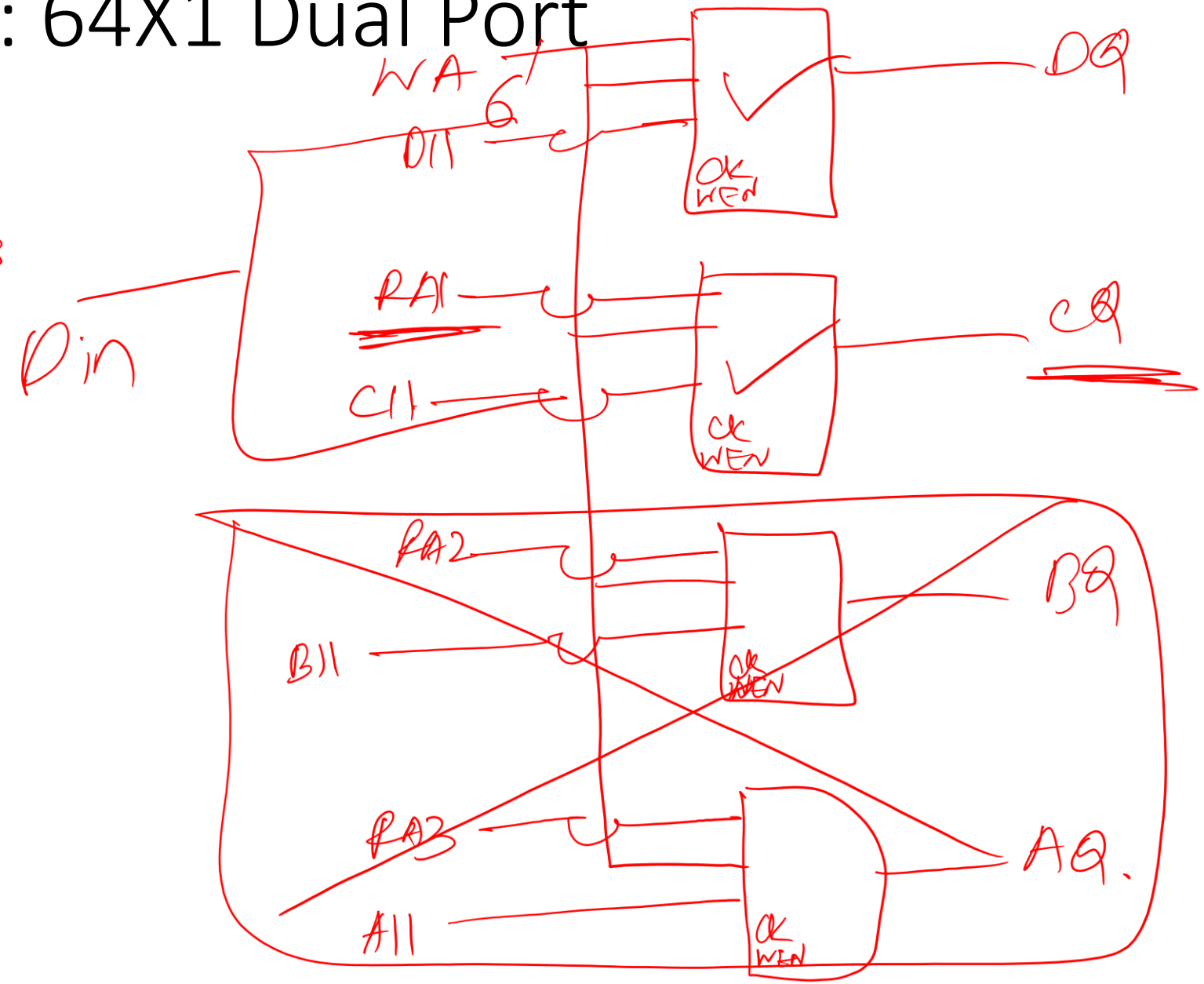
- **Single port:** Common address port for synch write and asynch read
- One SLICEM can have FOUR 64 x 1-bit memories as long as they share the same clock, write enable, and shared read and write port address inputs
- This configuration equates to a 64 x 4-bit single-port distributed RAM.
- What about O5 output?



64 X 1 Single Port Distributed RAM (RAM64X1S)

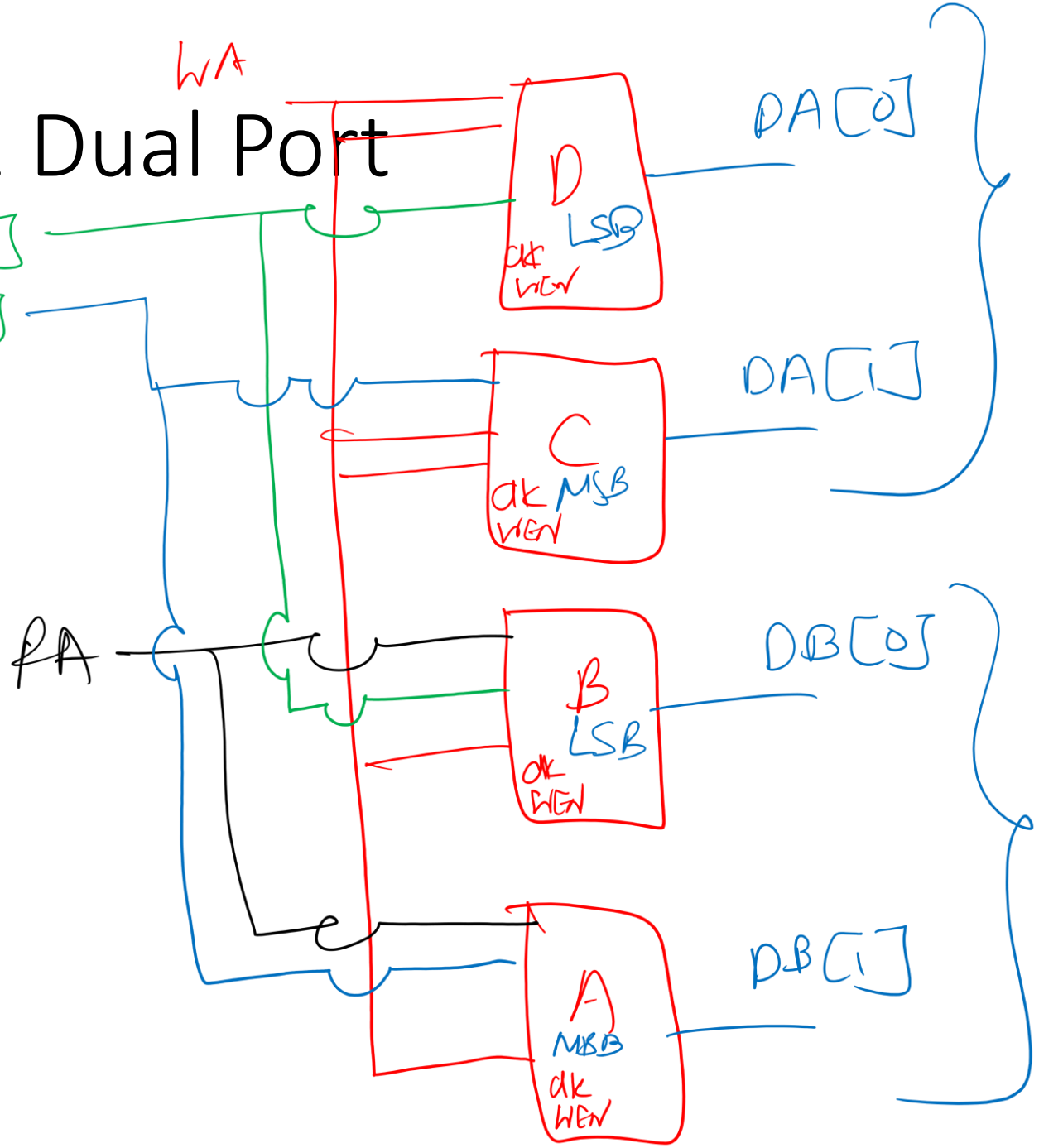
LUT as Memory: 64X1 Dual Port





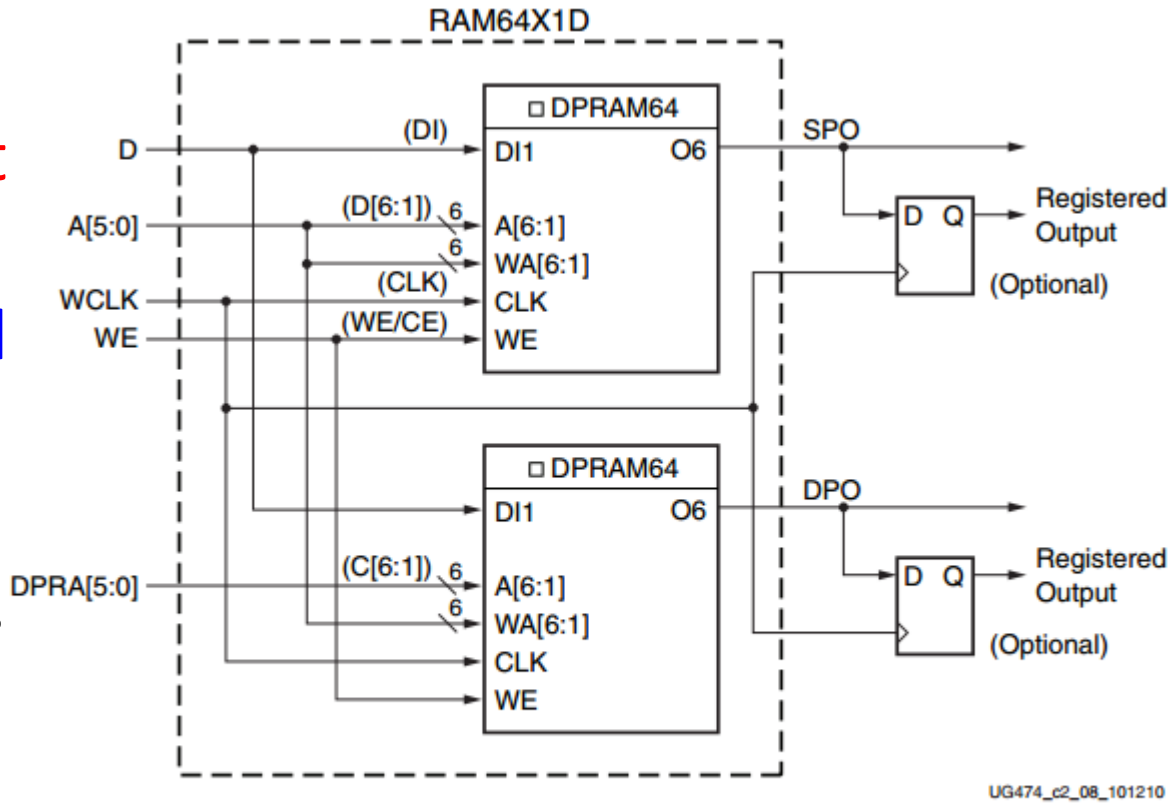
LUT as Memory: 64X2 Dual Port

64x1 Quad Port.



LUT as Memory: 64X1 Dual Port

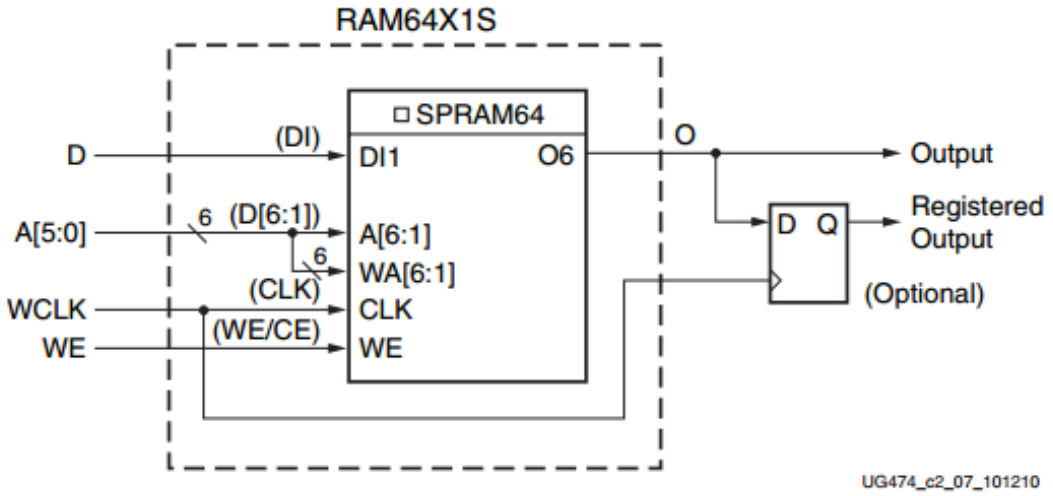
- One SLICEM can have TWO 64 x 1-bit memories as long as they share the same clock, write enable, and shared read and write port address inputs
- This configuration equates to a 64 x 2-bit dual-port distributed RAM.



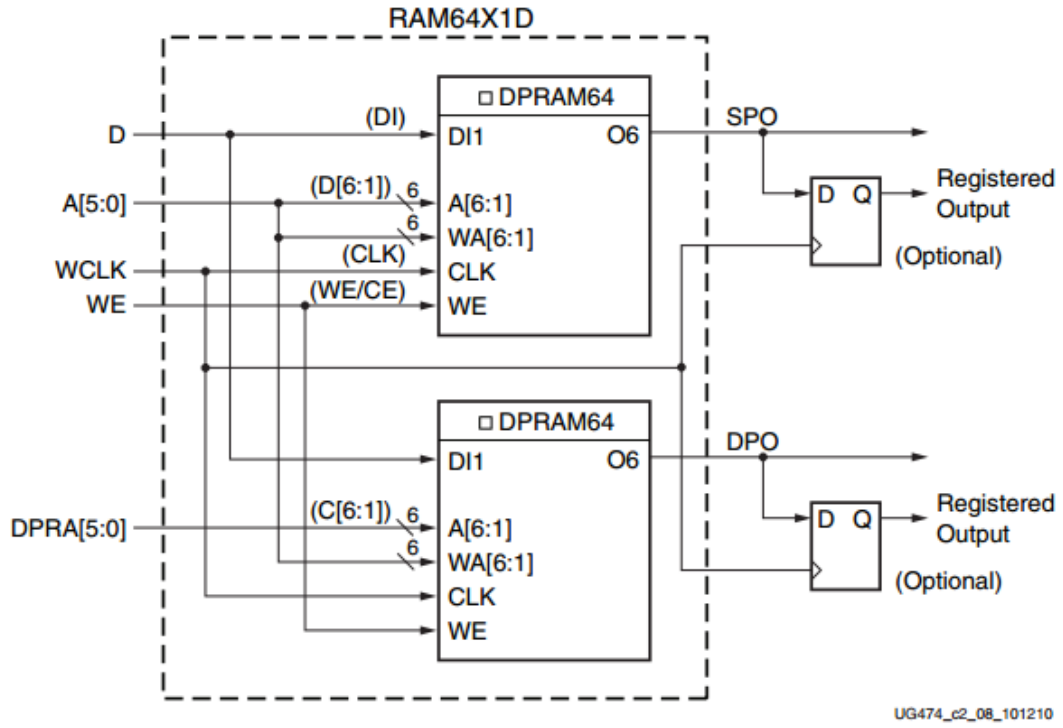
64 X 1 Dual Port Distributed RAM (RAM64X1D)

LUT as Memory

- **Synchronous Write Operation**
  The synchronous write operation is a single clock-edge operation with an active-High write-enable (WE) feature. When WE is High, the input (D) is loaded into the memory location at address A.



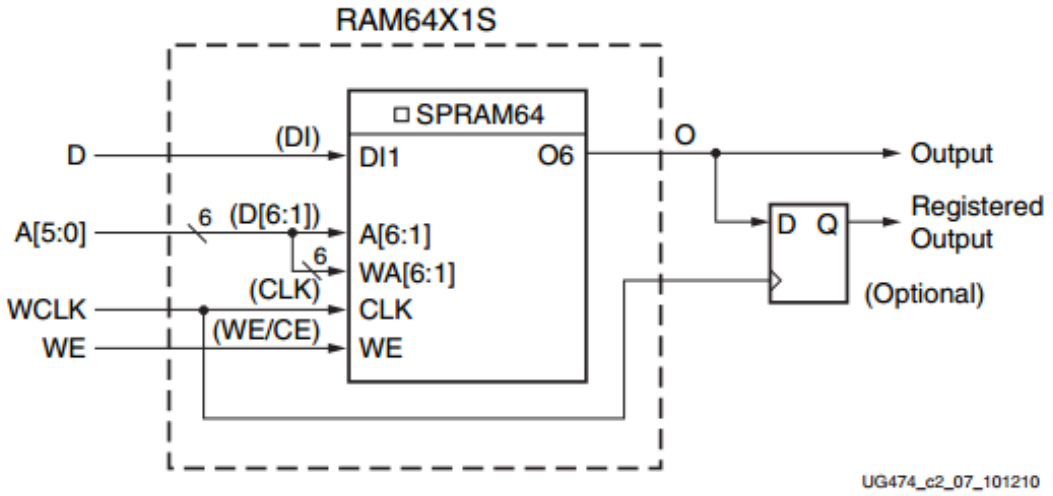
64 X 1 Single Port Distributed RAM (RAM64X1S)



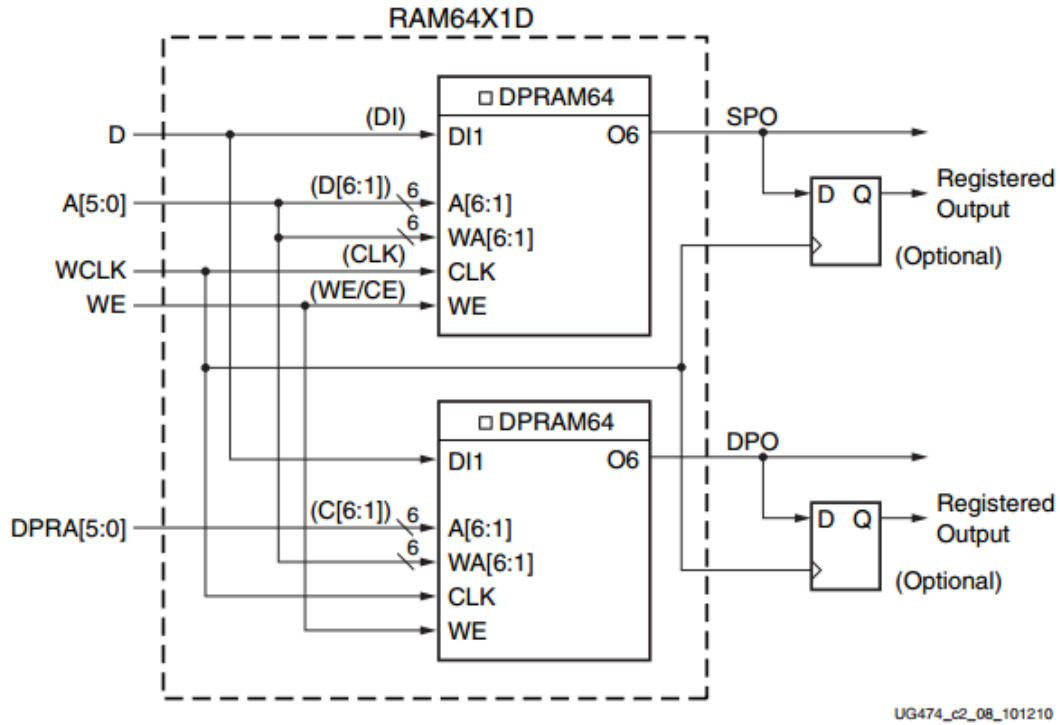
64 X 1 Dual Port Distributed RAM (RAM64X1D)

LUT as Memory

- **Asynchronous Read Operation**
  The output (or output SPO of dual-port mode) is determined by the address A for the single-port mode or address DPRA determines the DPO output of dual-port mode.
- Each time a new address is applied to the address pins, the data value in the memory location of that address is available on the output.
- This operation is asynchronous and independent of the clock signal.

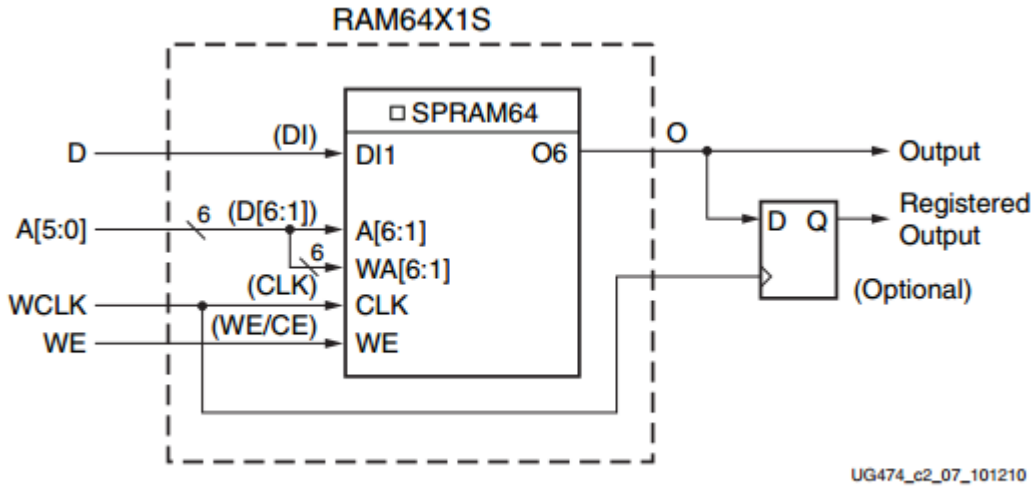


64 X 1 Single Port Distributed RAM (RAM64X1S)

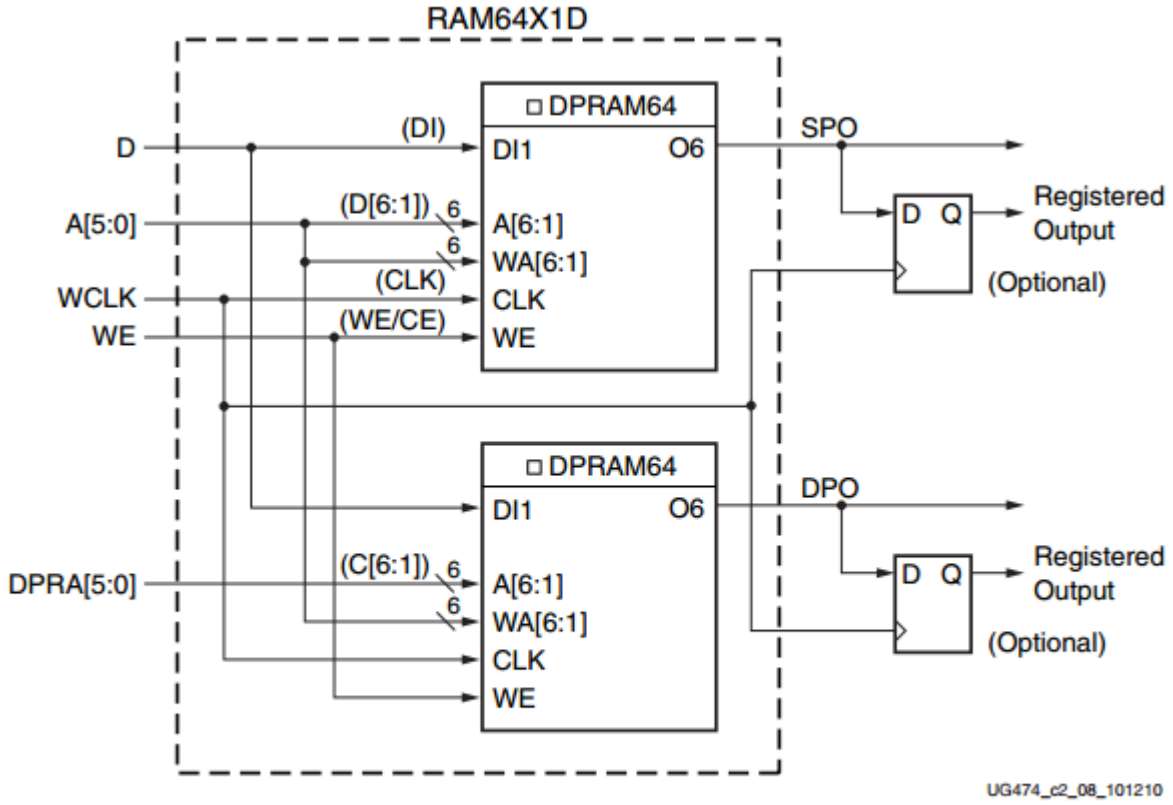


64 X 1 Dual Port Distributed RAM (RAM64X1D)

LUT as Memory: 64X1 Dual Port



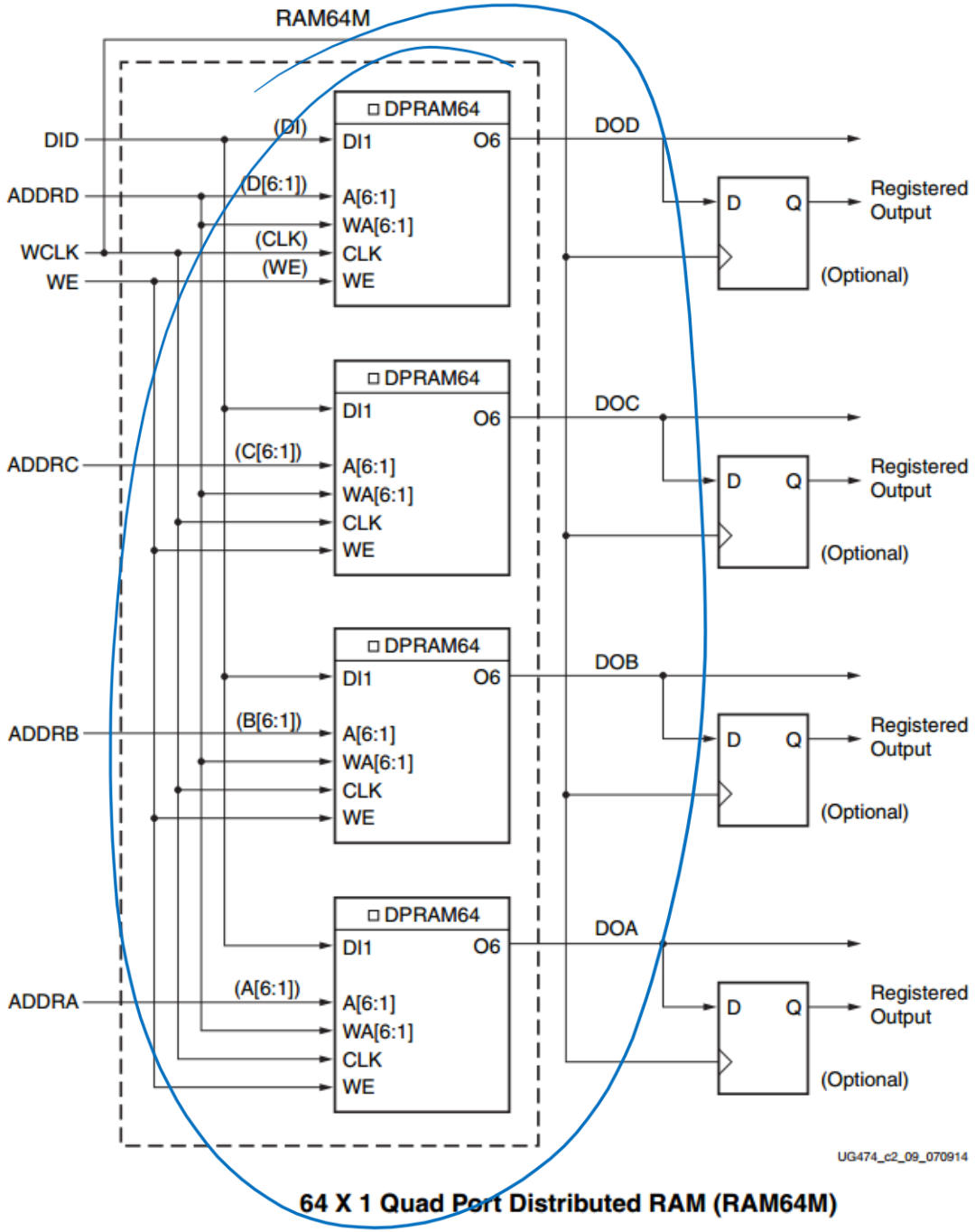
64 X 1 Single Port Distributed RAM (RAM64X1S)



64 X 1 Dual Port Distributed RAM (RAM64X1D)

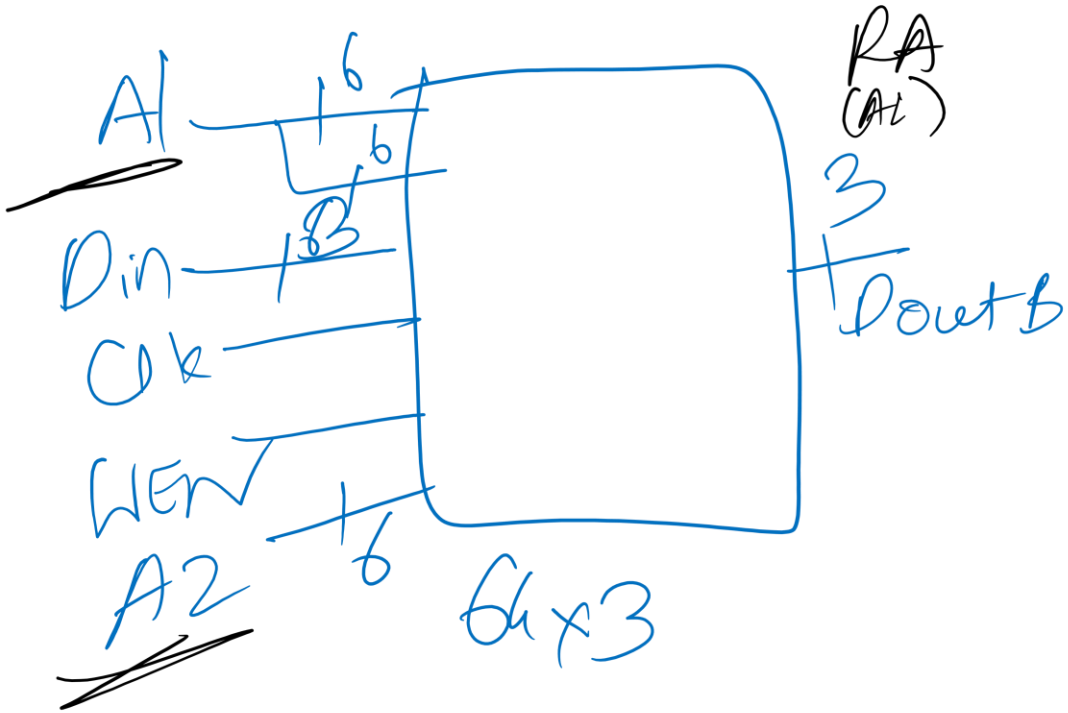
LUT as Memory: 64X1 Quad Port

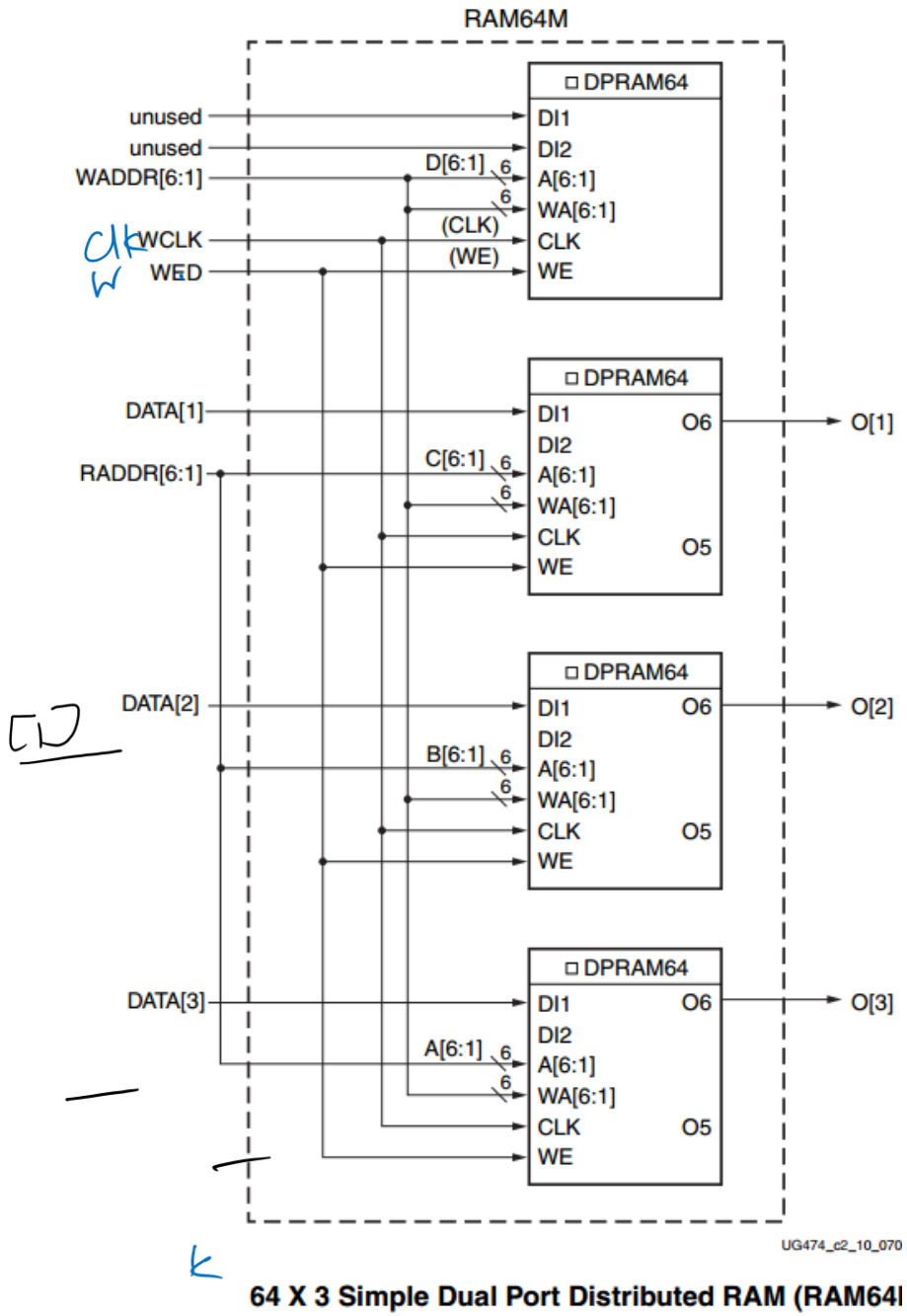
64x2 Quad Port. X



64 X 1 Quad Port Distributed RAM (RAM64M)

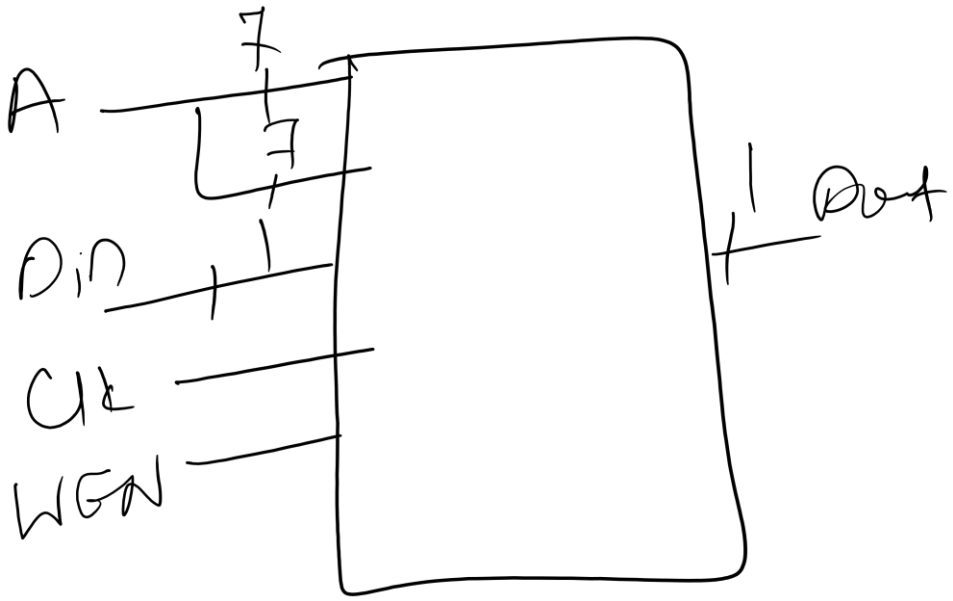
LUT as Memory: 64X3 Simple Dual Port





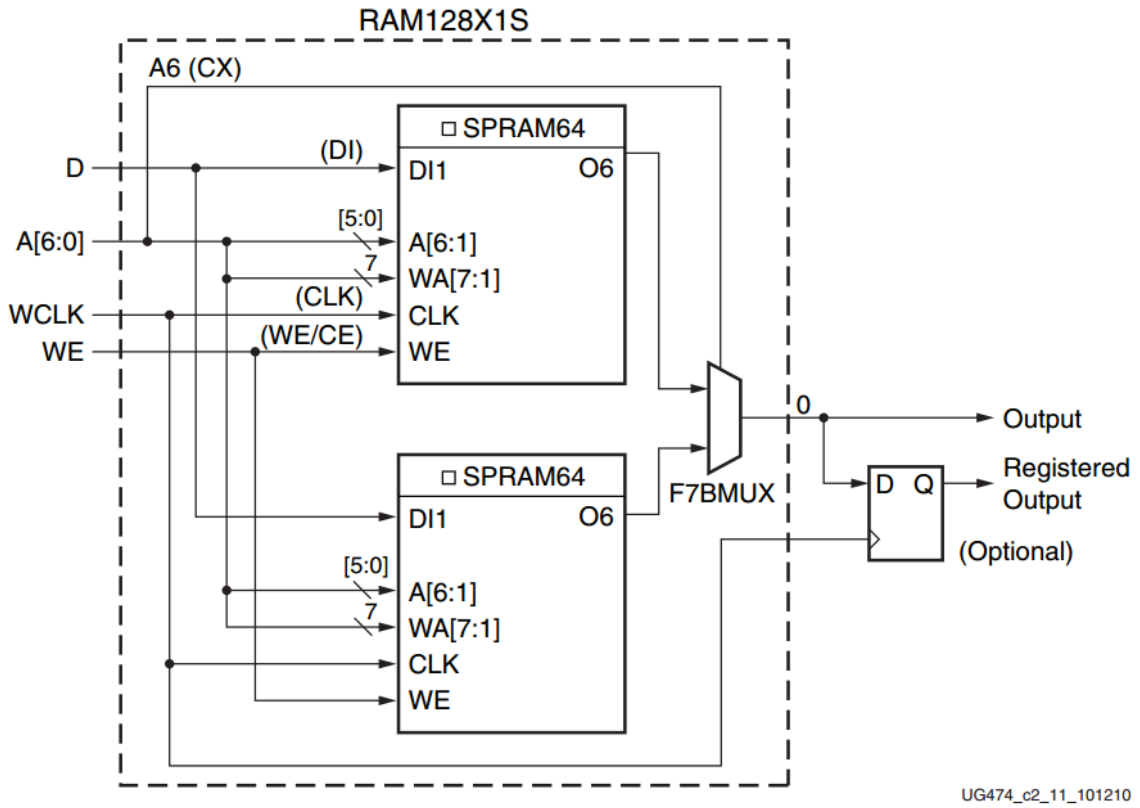
64 X 3 Simple Dual Port Distributed RAM (RAM64

LUT as Memory: 128X1 Single Port

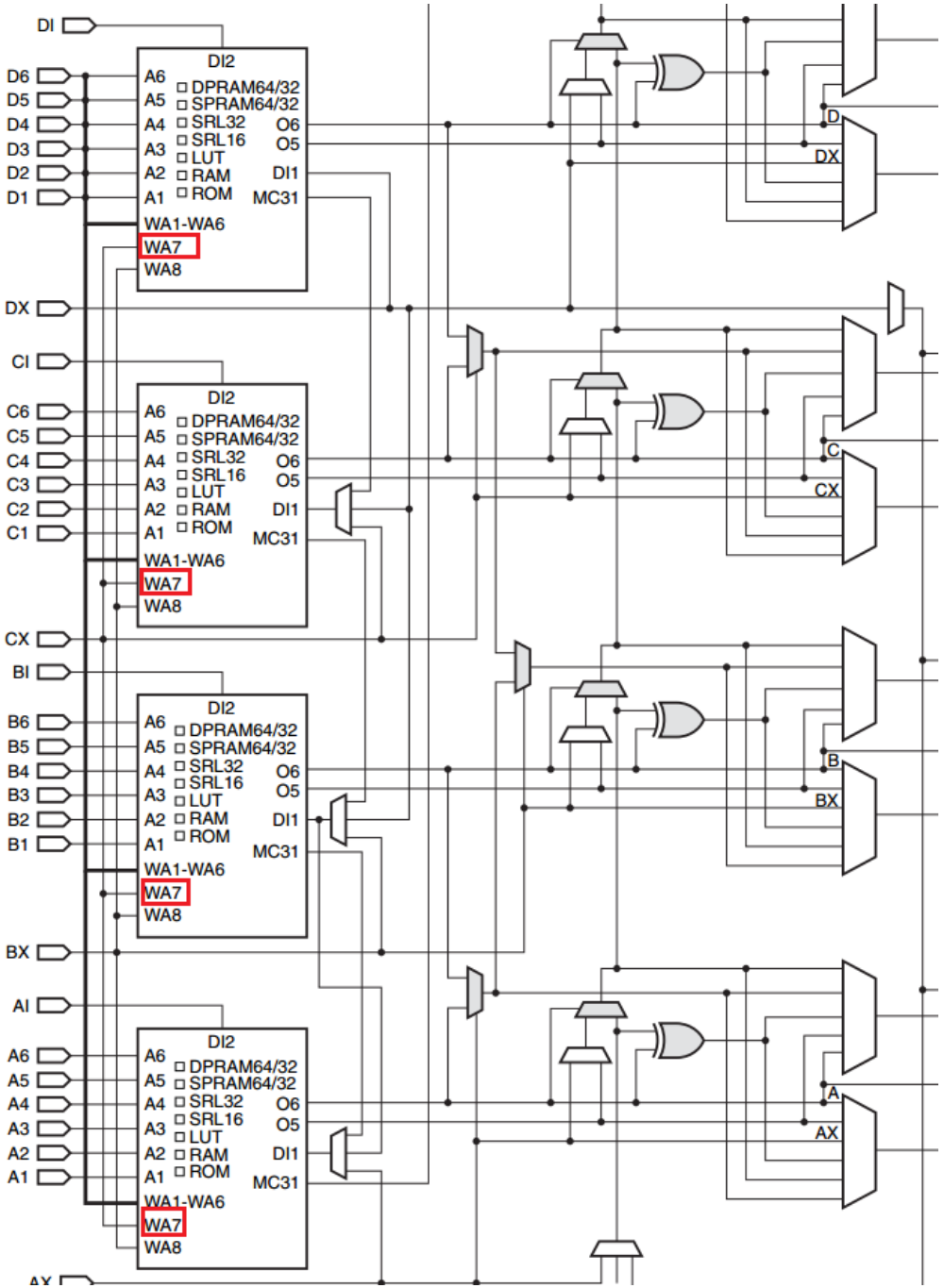


LUT as Memory: 128X1 Single Port

- Implementation of distributed RAM configurations with depth greater than 64 requires the usage of wide-function multiplexers (F7AMUX, F7BMUX, and F8MUX)
- One SLICEM can have TWO single port 128 x 1-bit memories as long as they share the same clock, write enable, and shared read and write port address inputs
- This configuration equates to a 128 x 2-bit single-port distributed RAM.

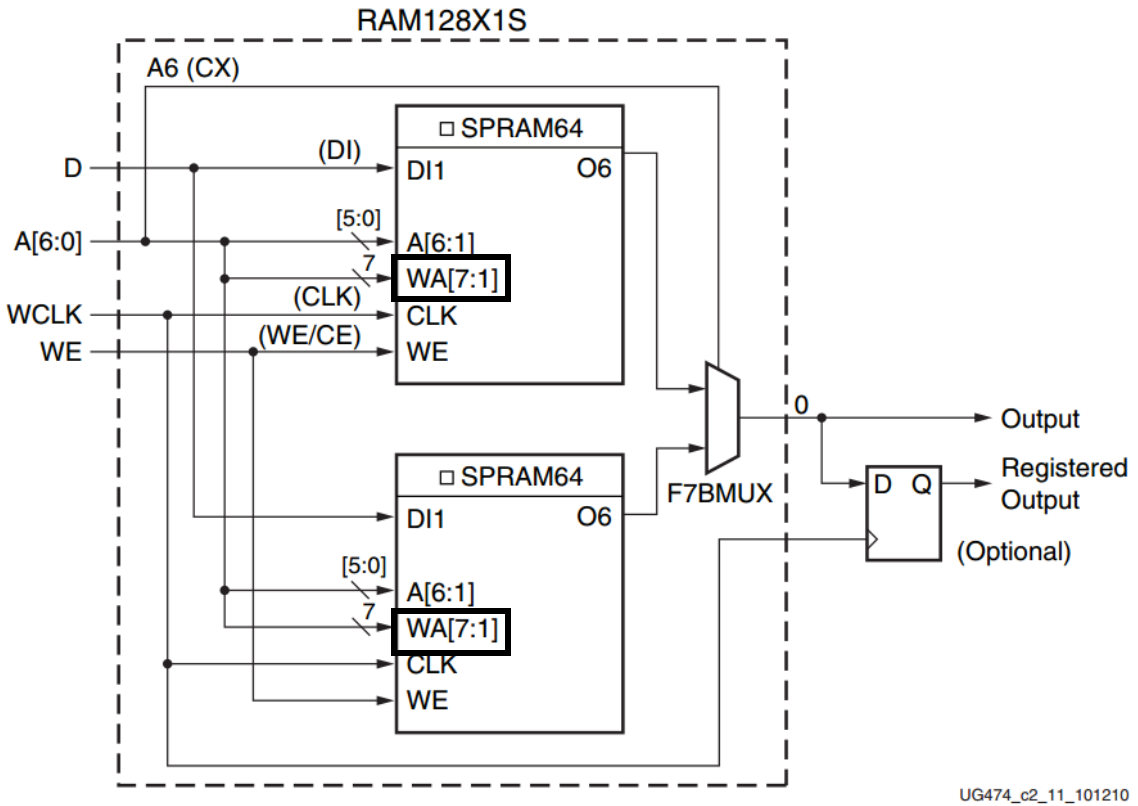


128 X 1 Single Port Distributed RAM (RAM128X1S)



WA7: Enable for 7-input LUT

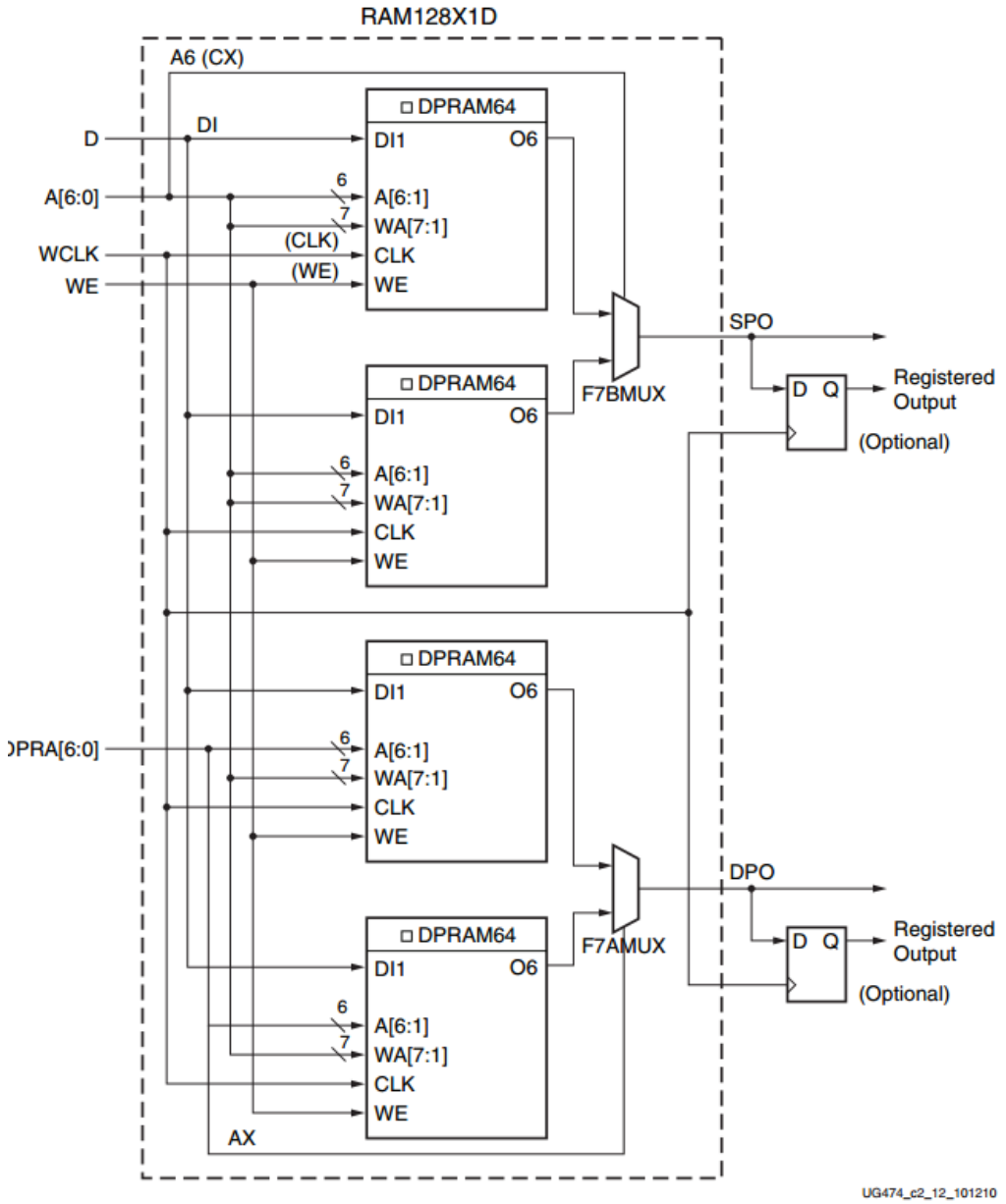
WA8: Enable for 8-input LUT



128 X 1 Single Port Distributed RAM (RAM128X1S)

LUT as Memory: 128X1 Dual Port

- 128X1 Quad Port?
- 128X2 Dual Port?



128 X 1 Dual Port Distributed RAM (RAM128X1D)

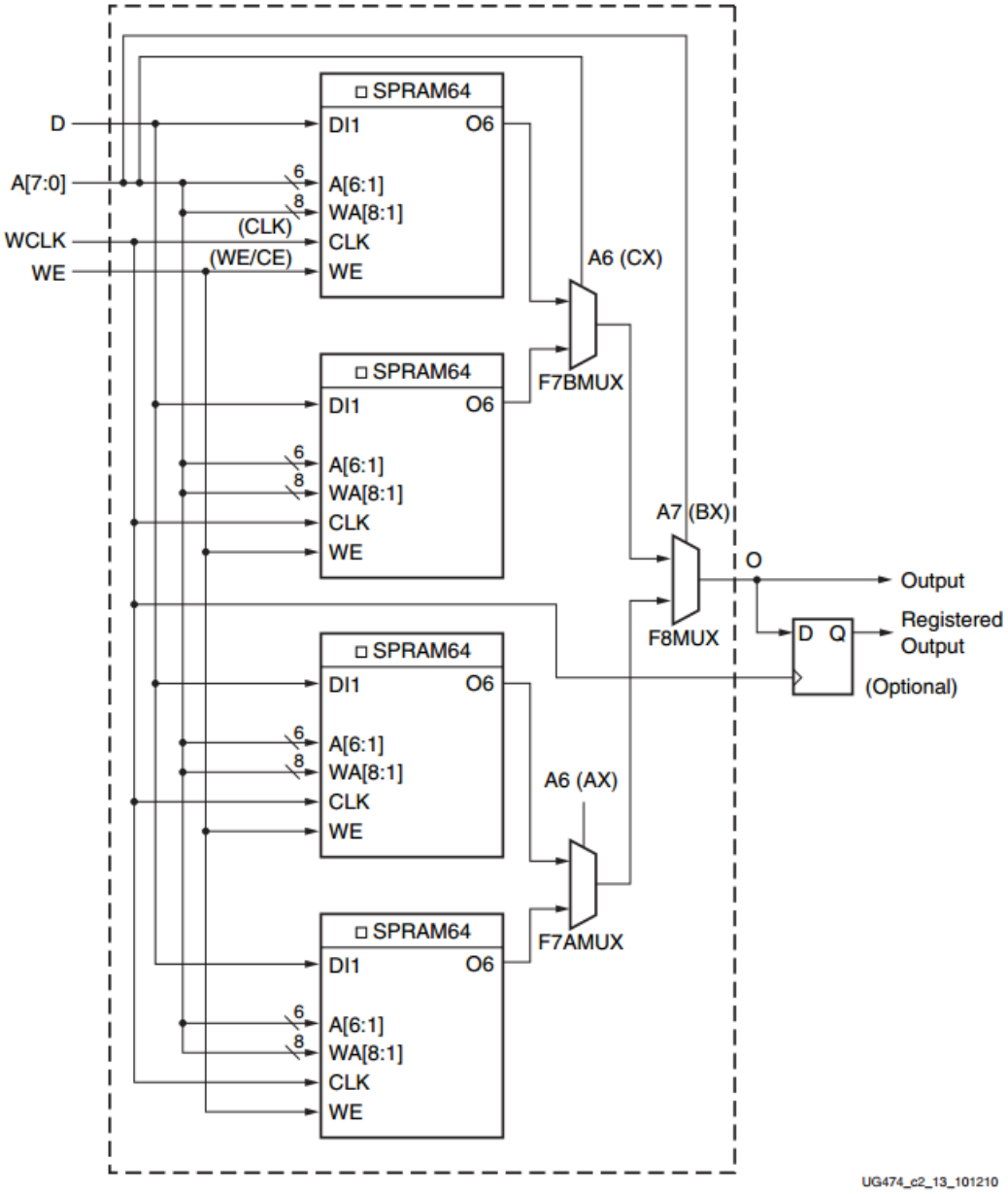
LUT as Memory: 256X1 Single Port

32 x 16 Single Port

256 x 1 Dual Port X

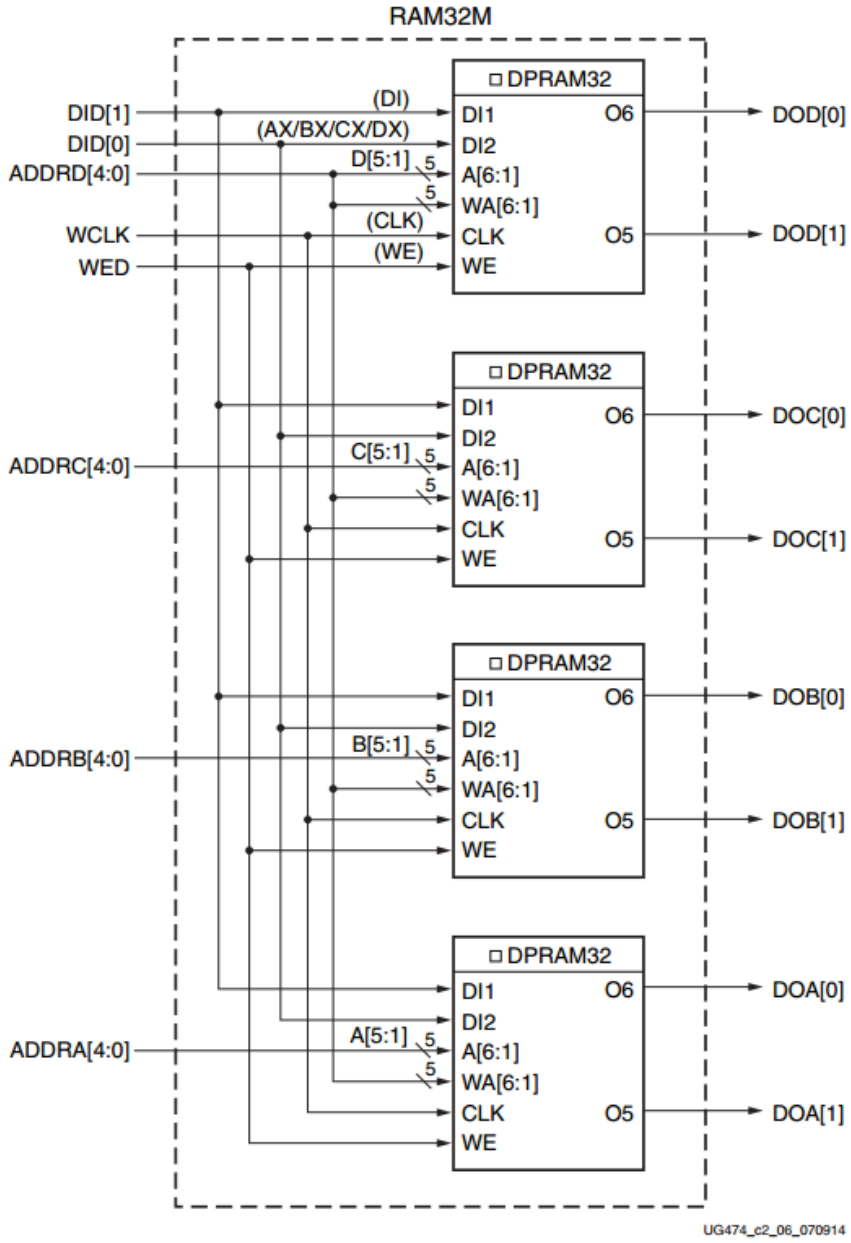
128 x 1 Quad Port X

128 x 1 Dual Port X



256 X 1 Single Port Distributed RAM (RAM256X1S)

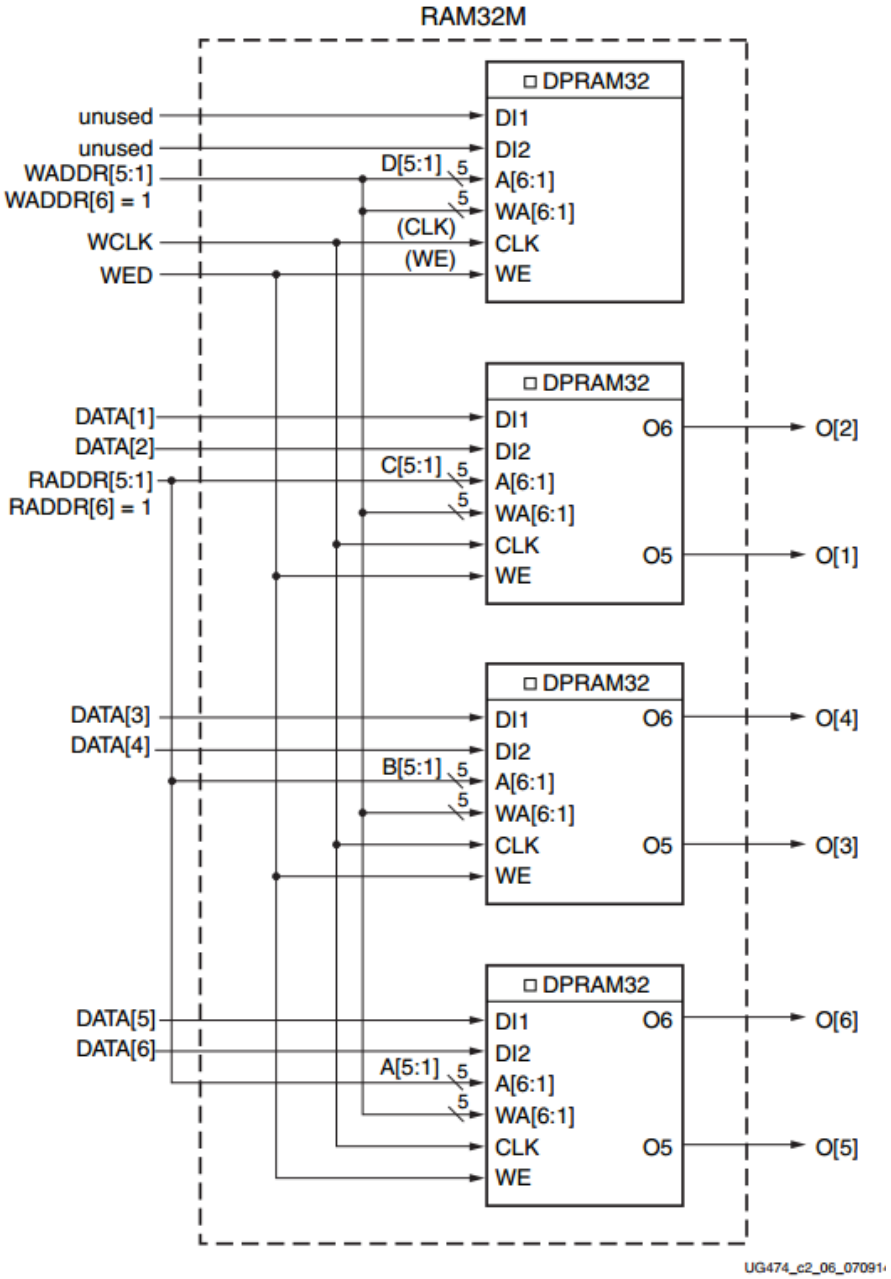
LUT as Memory: 32X2 Quad Port



32 X 2 Quad Port Distributed RAM (RAM32M)

LUT as Memory:
32X8 Simple Dual Port

LUT as Memory:
32X6 Simple Dual Port



32 X 6 Simple Dual Port Distributed RAM (RAM32M)

LUT as Memory

RAM	Description	Primitive	Number of LUTs
32 x 1S	Single port	RAM32X1S	1
32 x 1D	Dual port	RAM32X1D	2
32 x 2Q	Quad port	RAM32M	4
32 x 6SDP	Simple dual port	RAM32M	4
64 x 1S	Single port	RAM64X1S	1
64 x 1D	Dual port	RAM64X1D	2
64 x 1Q	Quad port	RAM64M	4
64 x 3SDP	Simple dual port	RAM64M	4
128 x 1S	Single port	RAM128X1S	2
128 x 1D	Dual port	RAM128X1D	4
256 x 1S	Single port	RAM256X1S	4

Single Port	Dual Port	Simple Dual Port	Quad Port
32x2	32x2**D**	32x6**SDP**	32x2**Q**
32x4	32x4**D**	64x3**SDP**	64x1**Q**
32x6	64x1**D**		
32x8	64x2**D**		
64x1	128x1**D**		
64x2			
64x3			
64x4			
128x1			
128x2			
256x1			

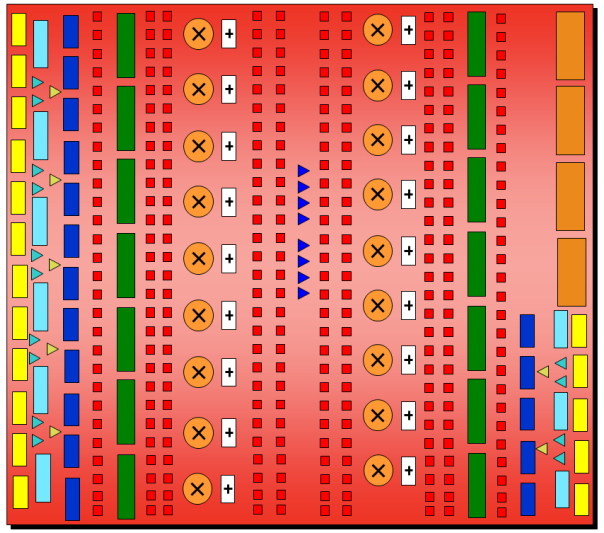
Each Port Has Independent Address Inputs

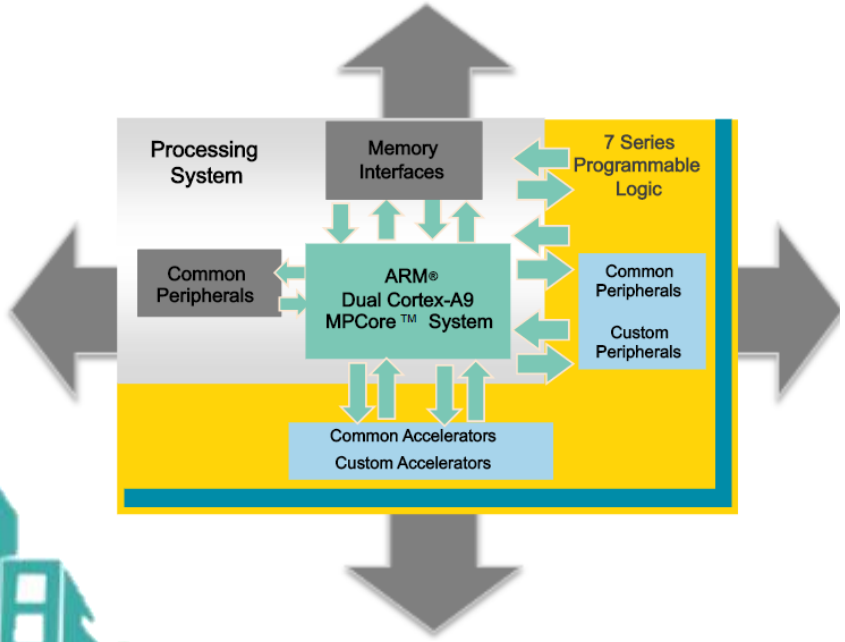
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

ECE 270: Embedded Logic Design

SDK
Software Development Kit
VIVADO™





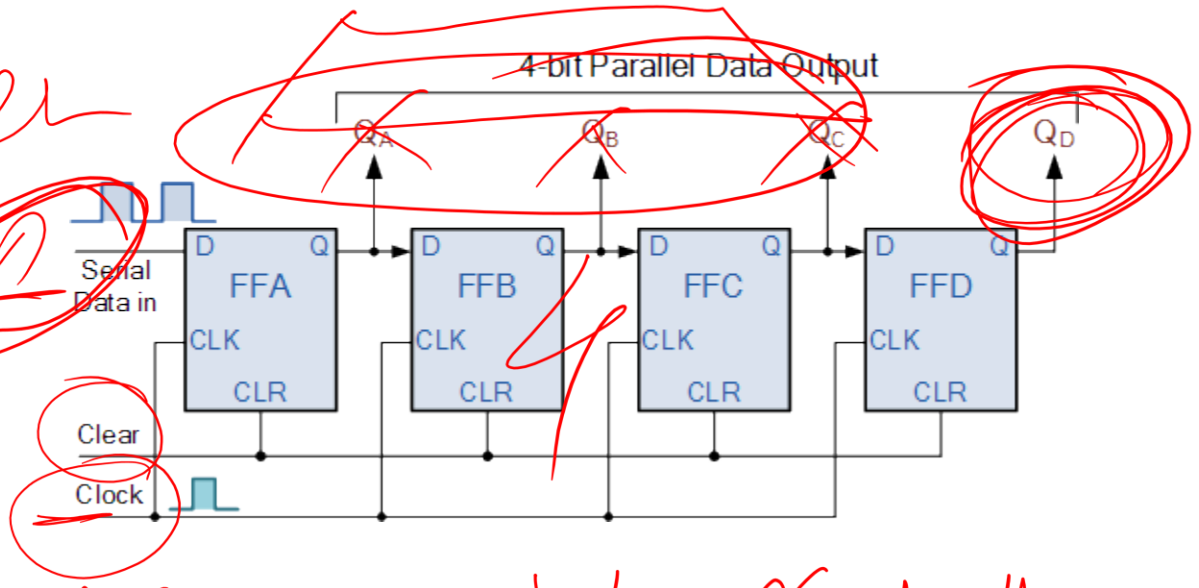
SISO

LUT as SRL

Sequential Circuits

- 4-bit serial shift register:

```
module ShiftReg(
input wire clk,
input wire clr,
input wire data_in,
output reg [3:0] Q
);

//      4-bit Shift Register
always @(posedge clk or posedge clr)
begin
    if(clr == 1)
        Q <= 0;
    else
    begin
        Q[3] <= data_in;
        Q[2:0] <= Q[3:1];
    end
end
endmodule
```

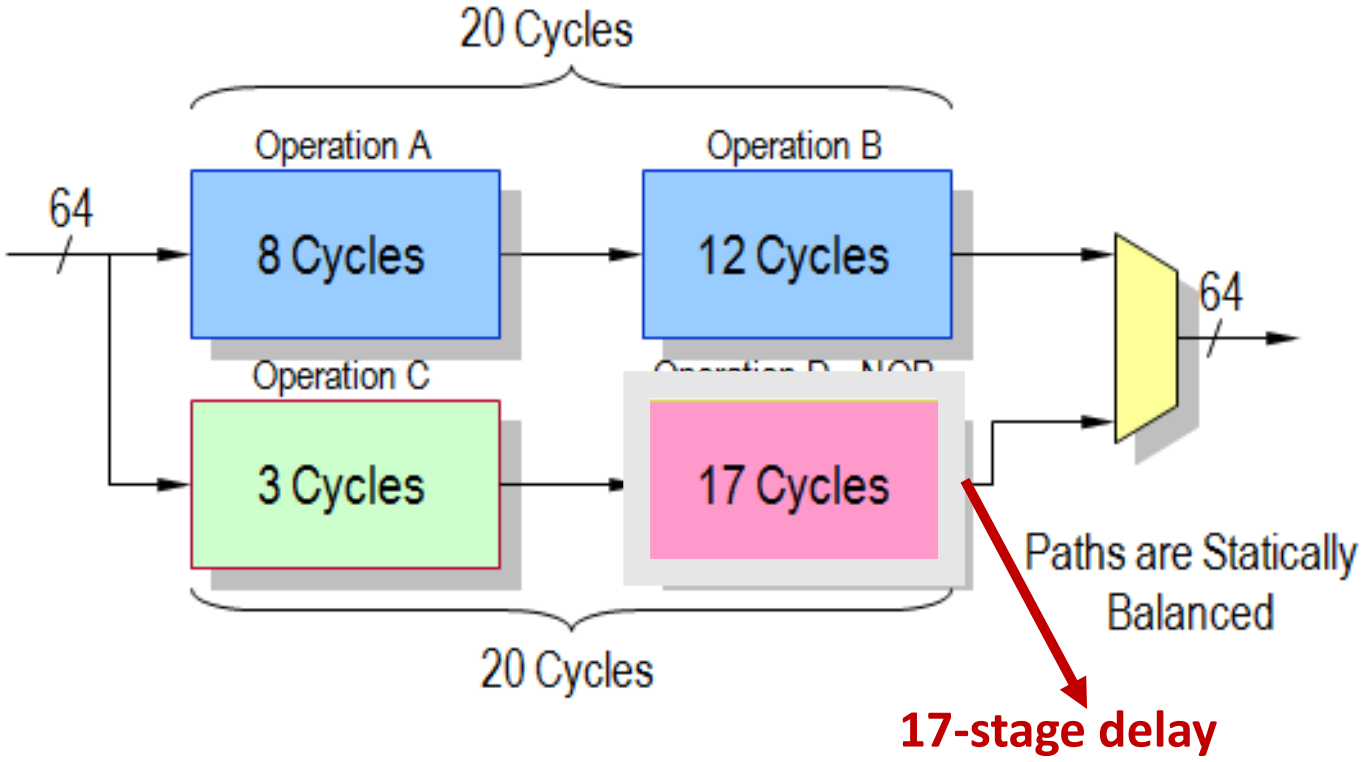


module(in, clr, clk, Q_D)
input in, clr, clk;
output reg Q_D;
wire Q_A, Q_B, Q_C;
always @(posedge clk or posedge clr)
if(clr)
$Q_A \leftarrow 0$;
else $Q_A \leftarrow$ in;

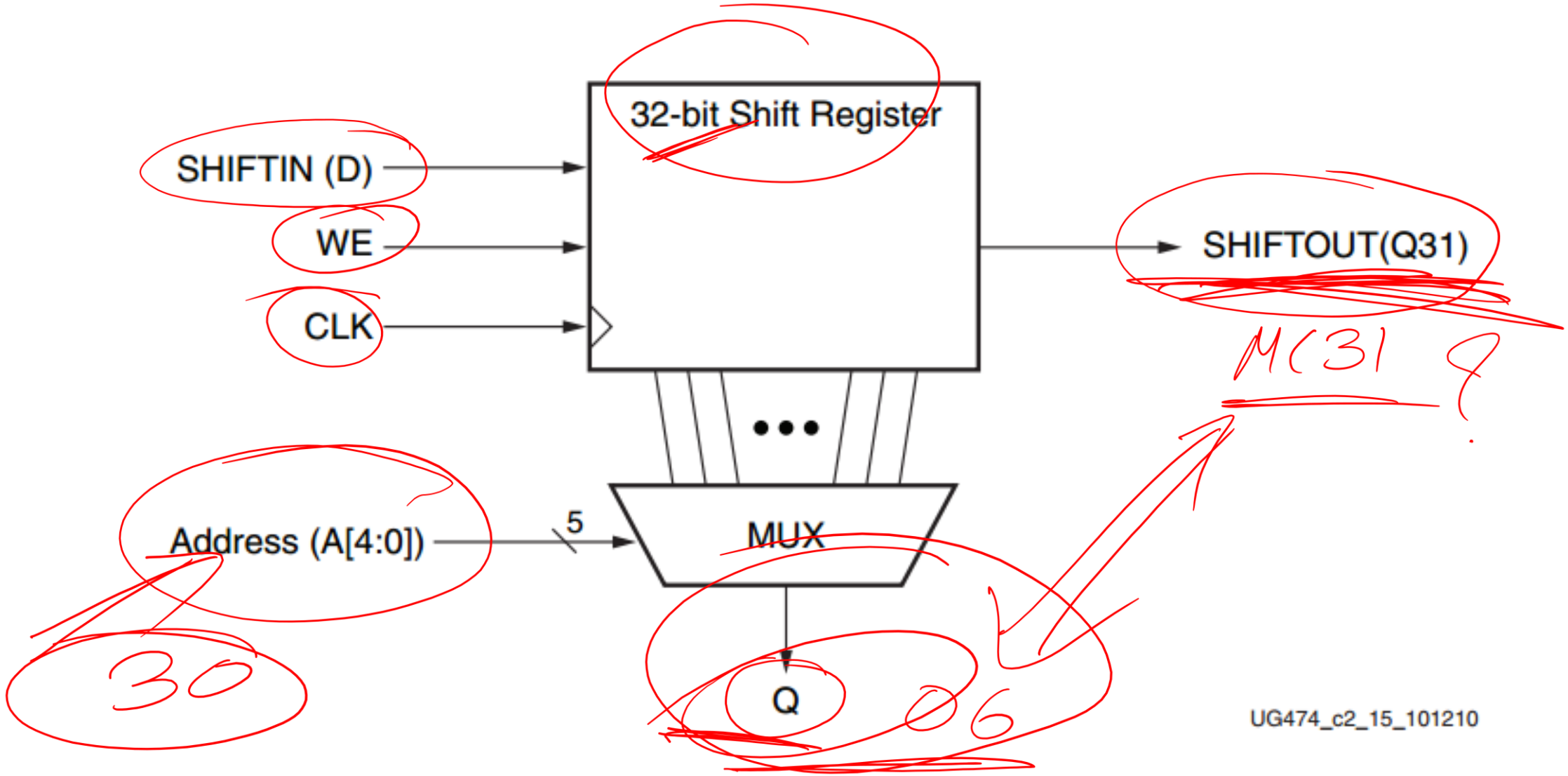
always @(posedge clk or posedge clr)
if(clr)
$Q_B \leftarrow 0$;
else $Q_B \leftarrow Q_A$;

Shift Register Using FFs

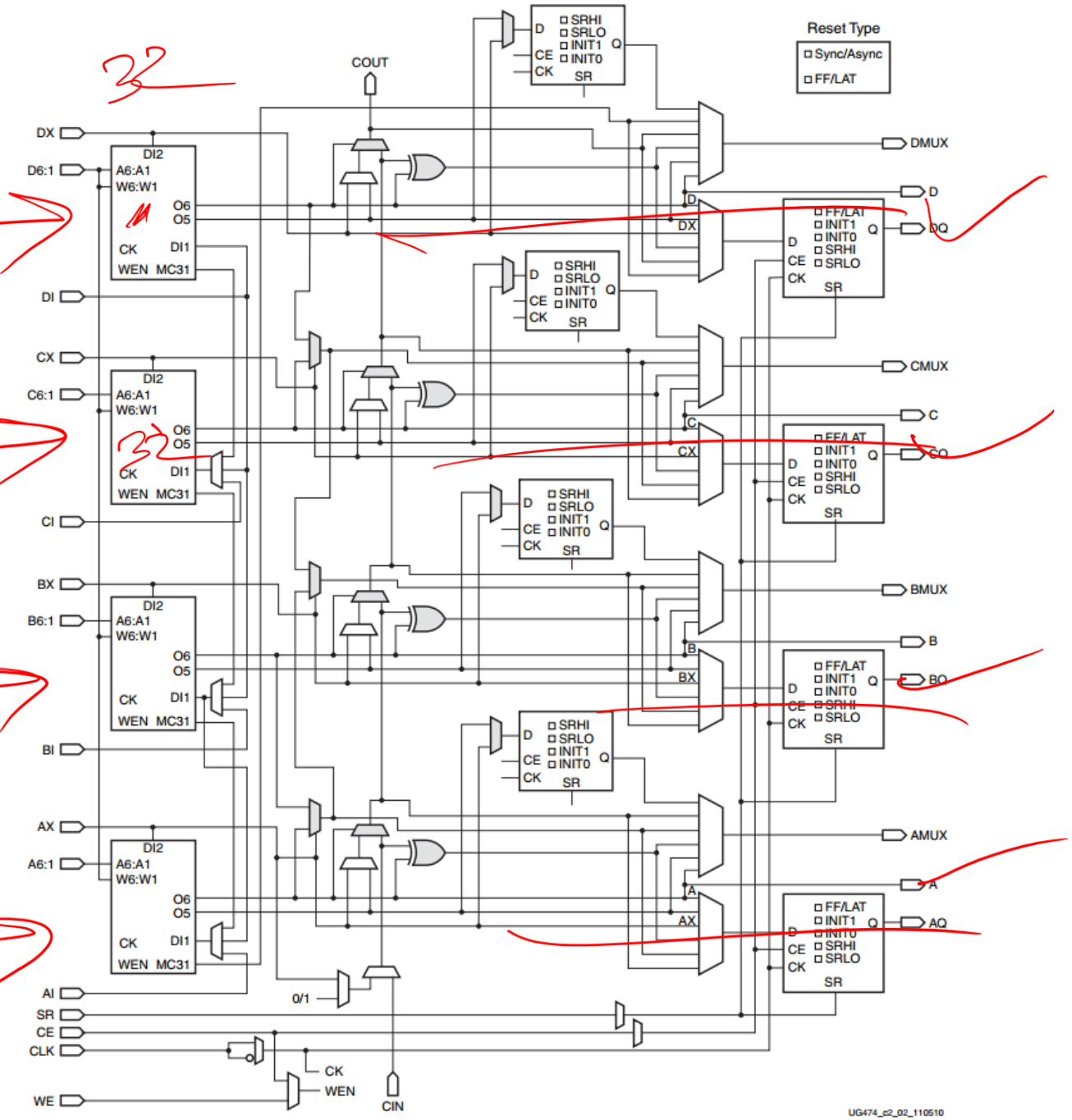
- Operation D - NOP must add 17 pipeline stages of 64 bits each
  - 1,088 flip-flops (hence 136 slices)



LUT as Shift Registers (only in SLICEM)

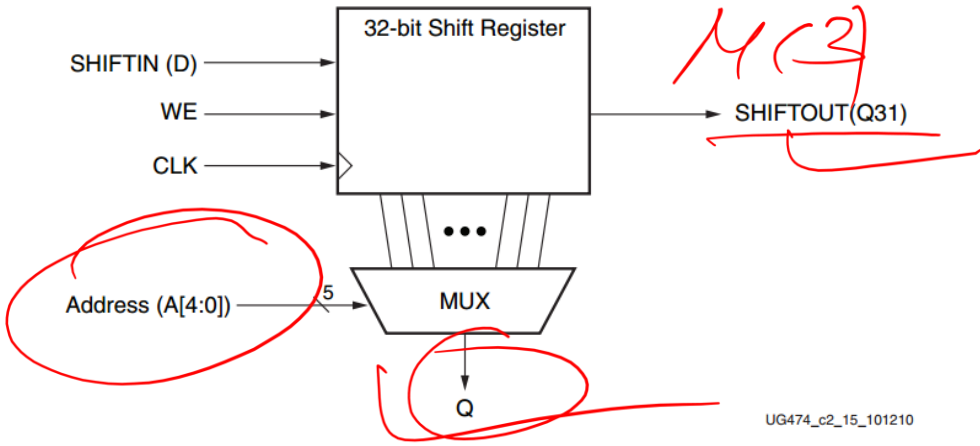


SLICEM


Reset Type
Sync/Async
FF/LAT
COUT
DX
D6:1
DI
CX
C6:1
CI
BX
B6:1
BI
AX
A6:1
AI
SR
CE
CLK
WE
DI2
A6:A1
W6:W1
O6
O5
CK
DI1
WEN
MC31
SRHI
SRLO
INIT1
INIT0
D
CE
Q
DMUX
DQ
CMUX
CQ
BMUX
BQ
AMUX
AQ
0/1
CIN
UG474_c2_02_110510

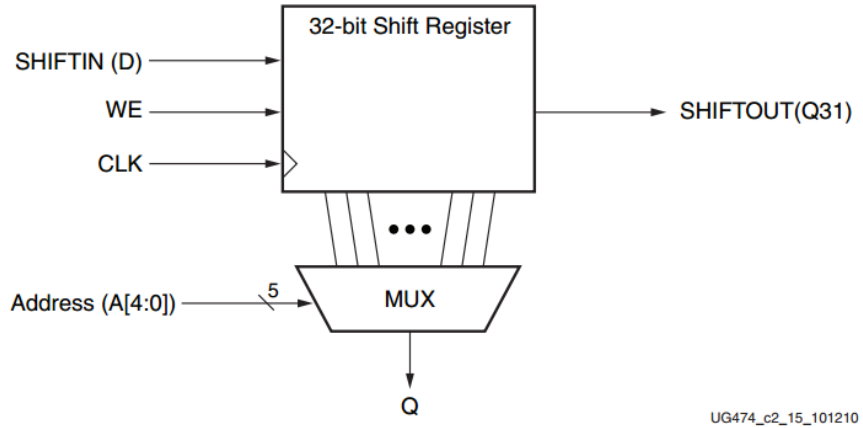
LUT as Shift Registers (only in SLICEM)

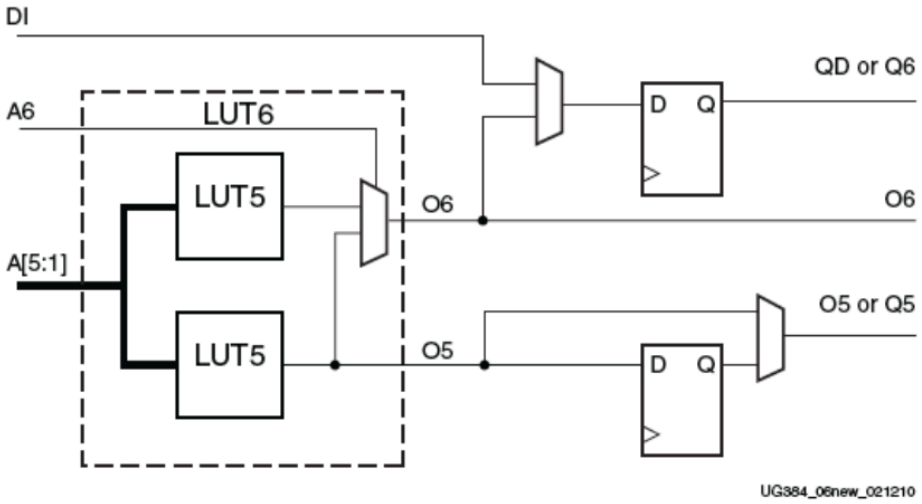
- Shift register functions include:
  - Write operation (Synchronous with a clock input and an optional clock enable)
  - Fixed read access to Q31
  - Dynamic read access:
    - Performed through 5-bit address bus (LSB is unused)
    - Any of the 32 bits can be read out asynchronously by controlling the address
    - Useful for smaller shift registers
  - Flip-flop can be used for synchronous read with one additional latency
  - **Set/reset is not supported**

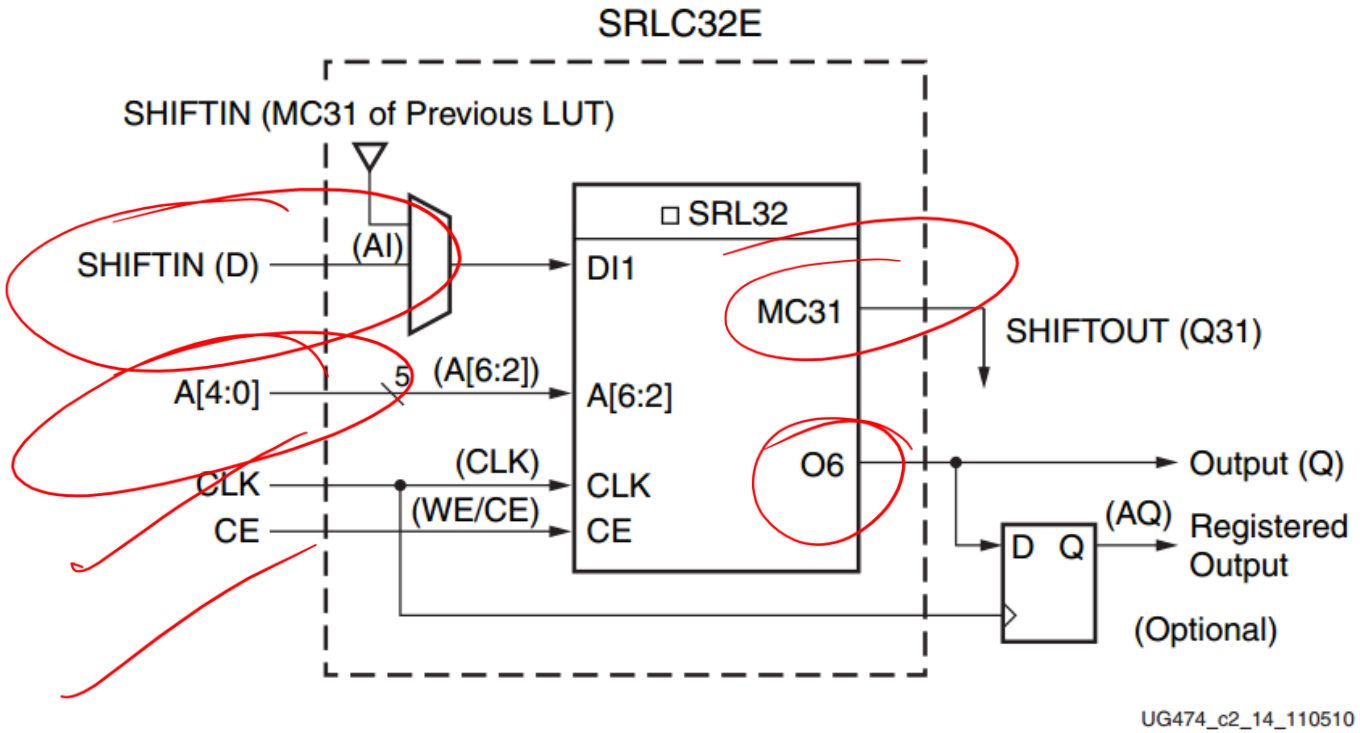


LUT as Shift Registers (only in SLICEM)

- A SLICEM LUT can be configured as a 32-bit shift register without using the flip-flops available in a slice.

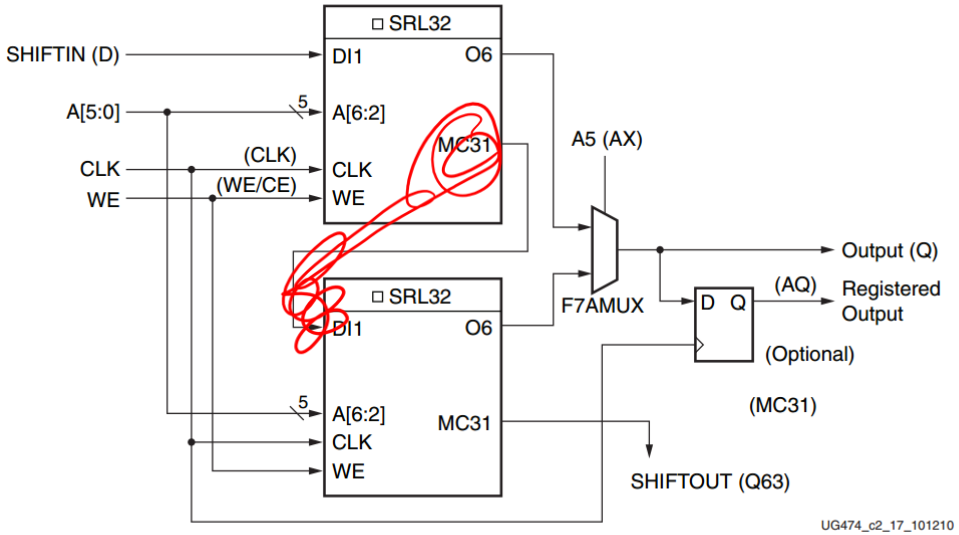






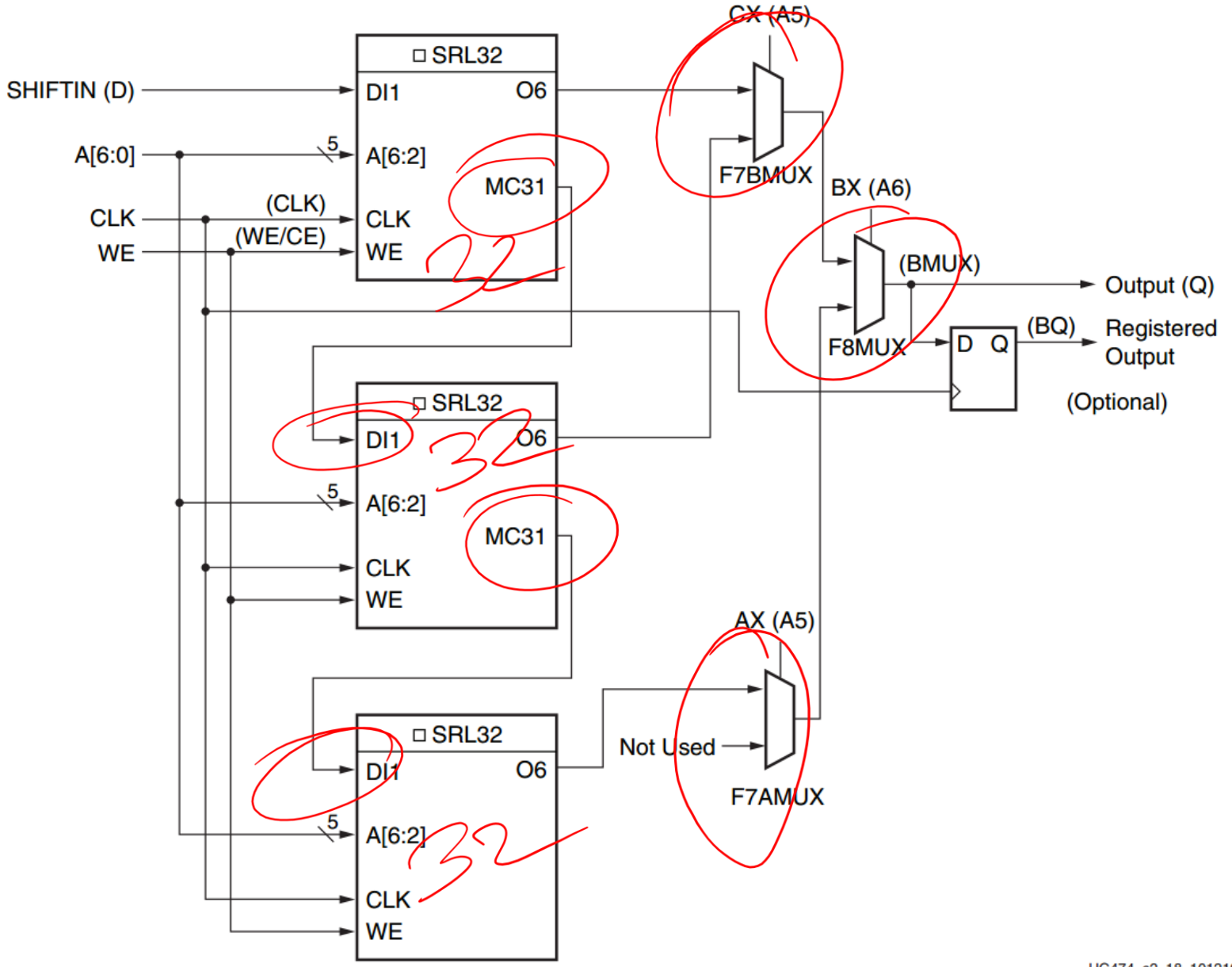
LUT as Shift Registers: 64 Bit

- MC31 output and a dedicated connection between LUTs allows connecting the last bit of one shift register to the first bit of the next, without using the LUT O6 output.
- Longer shift registers can be built with dynamic access to any bit in the chain.
- The shift register chaining and the F7AMUX F7BMUX, and F8MUX multiplexers allow up to a 128-bit shift register with addressable access to be implemented in one SLICEM



64-Bit Shift Register Configuration

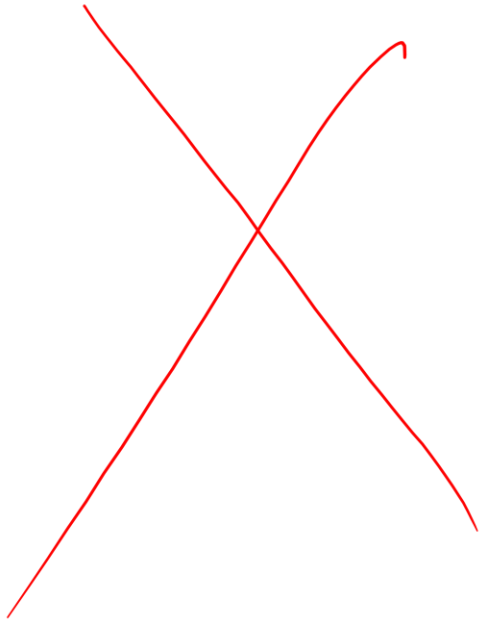
Shift Register
96 Bit



96-Bit Shift Register Configuration

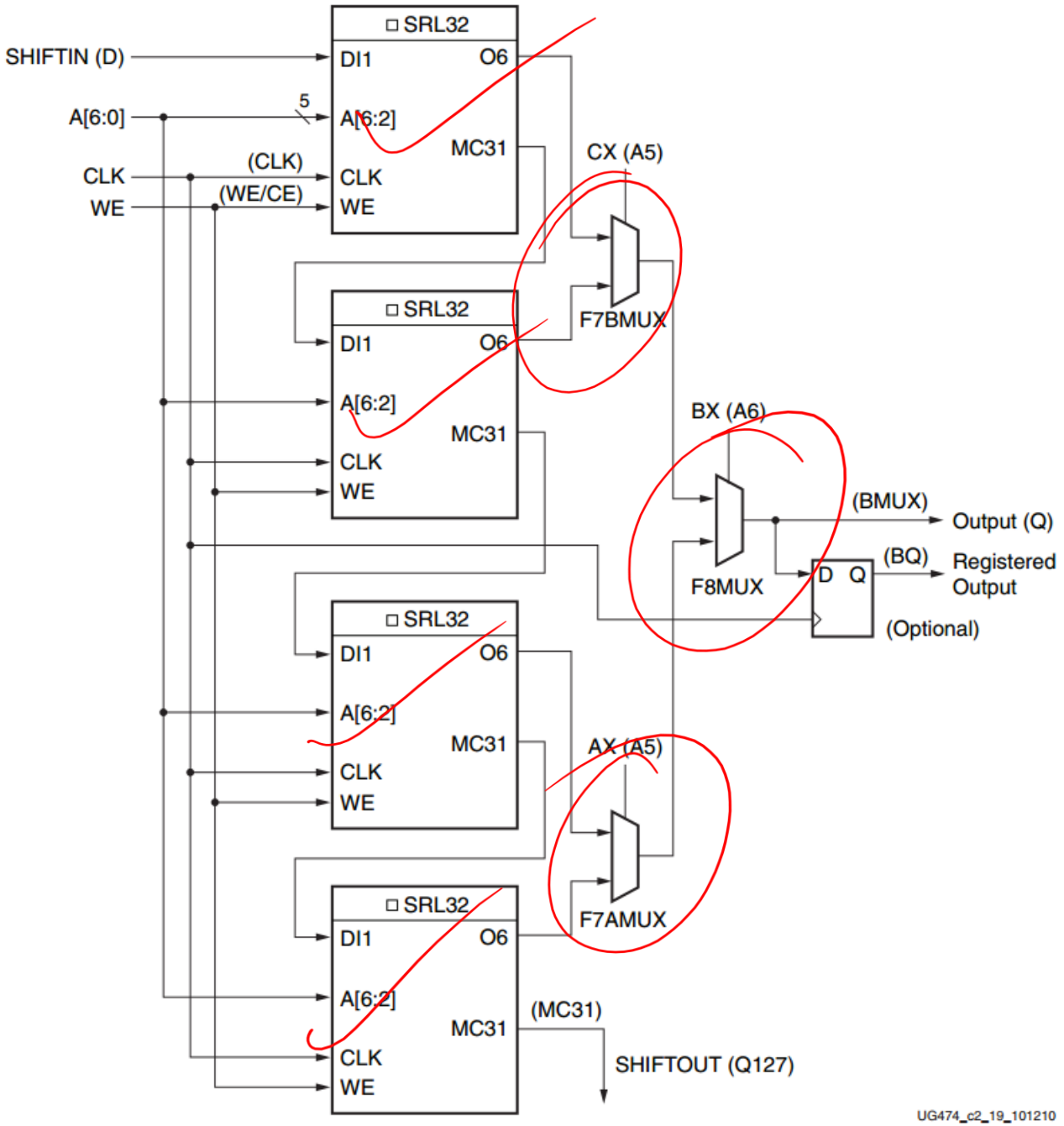
Shift Register

256 Bit

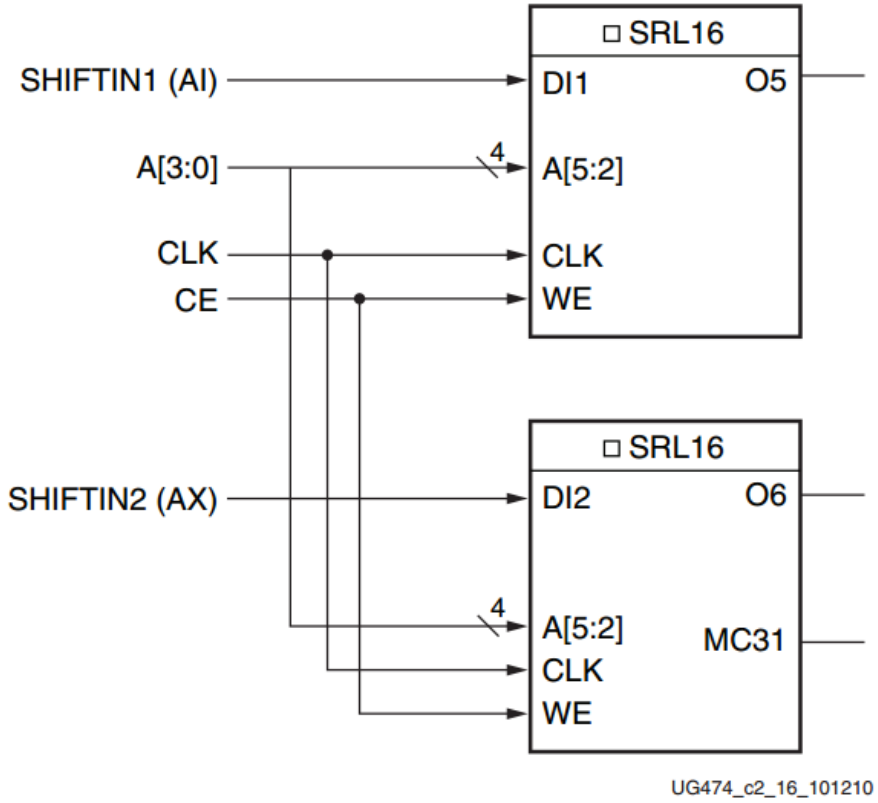


Slice8

Shift Register
128 Bit



Dual 16-bit SRL in One LUT



Dual 16-Bit Shift Register Configuration

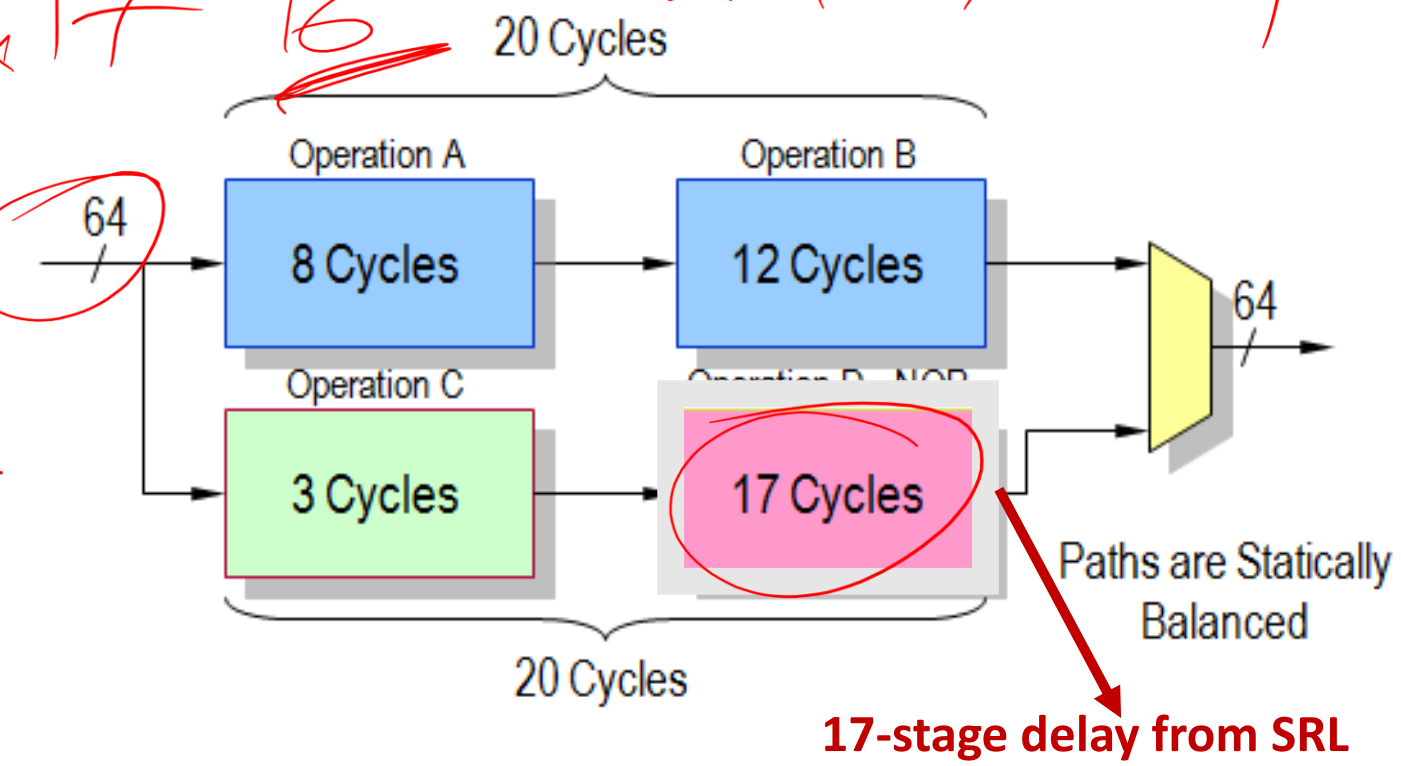
LUT as SRL

- There are **no set or reset capabilities**, it is not loadable, and data can only be read serially.
- Each LUT6 can implement a maximum delay of 32 clock cycles. The SRLs within a slice can be **cascaded** for longer shift registers (up to 128).
- The shift register length can be changed asynchronously by changing the value applied to the address pins (A). This means that you can dynamically change the pipeline delay associated with an SRL.

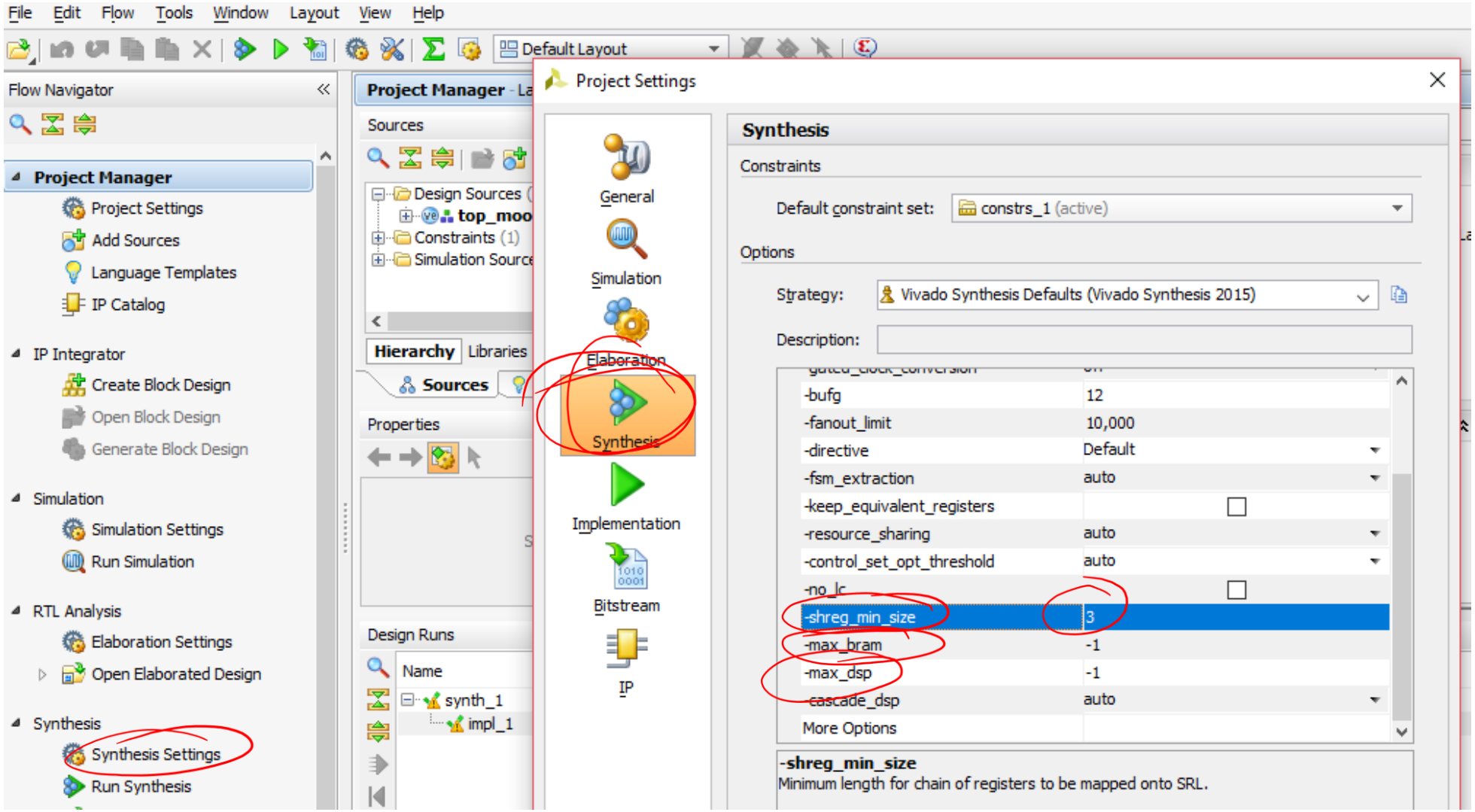
SRL Configurations in One Slice (4 LUTs)
16x1, 16x2, 16x4, 16x6, 16x8
32x1, 32x2, 32x3, 32x4
64x1, 64x2
96x1
128x1

Shift Register LUT Example

- Operation D - NOP must add 17 pipeline stages of 64 bits each
  - 1,088 flip-flops (hence 136 slices) or
  - 64 SRLs (hence 16 slices)



Shift Register LUT Example

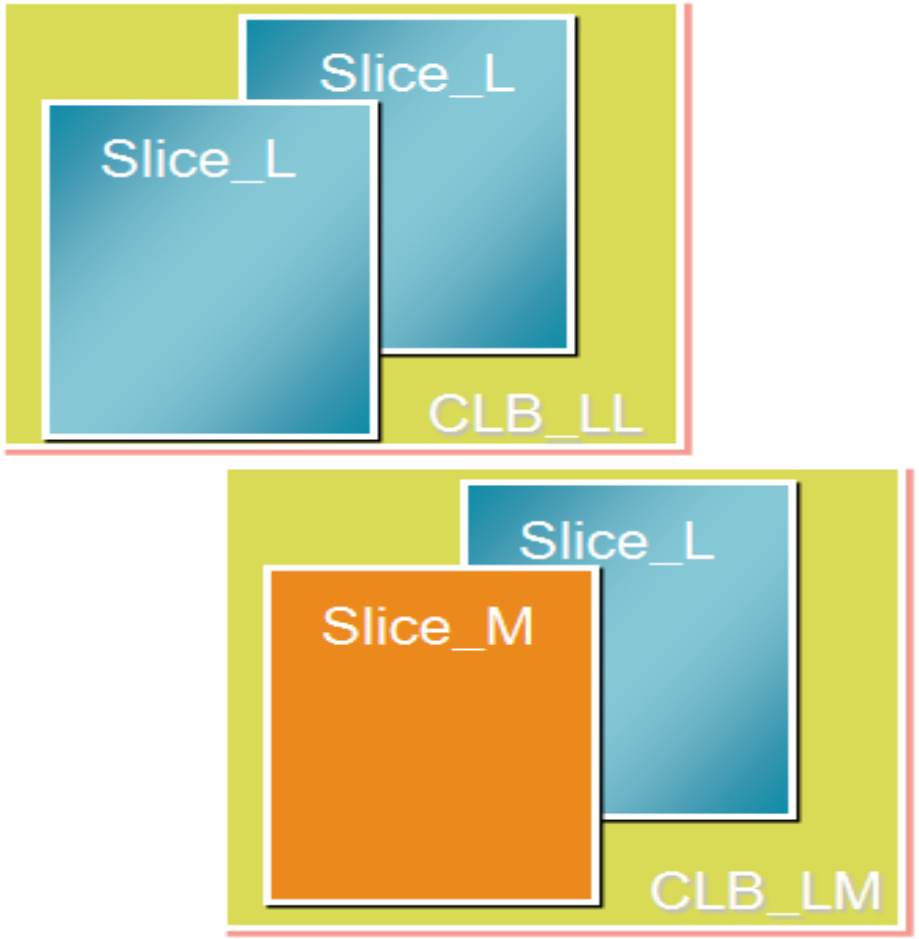


Types of CLB Slices

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM(1)	Shift Registers(1)
2	8	16	2		

Types of CLB Slices

Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM(1)	Shift Registers(1)
2	8	16	2		

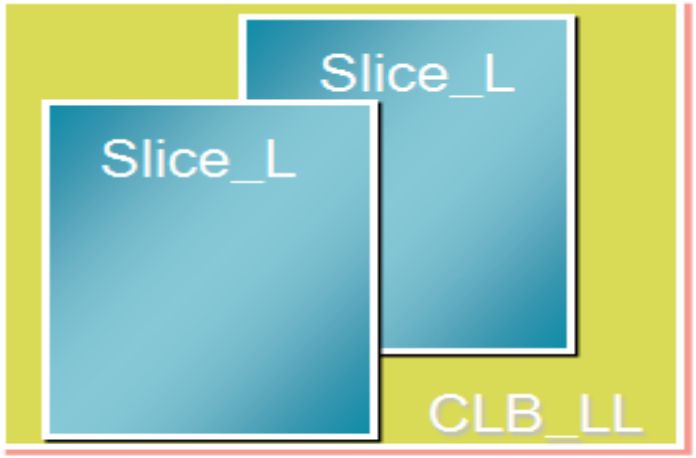


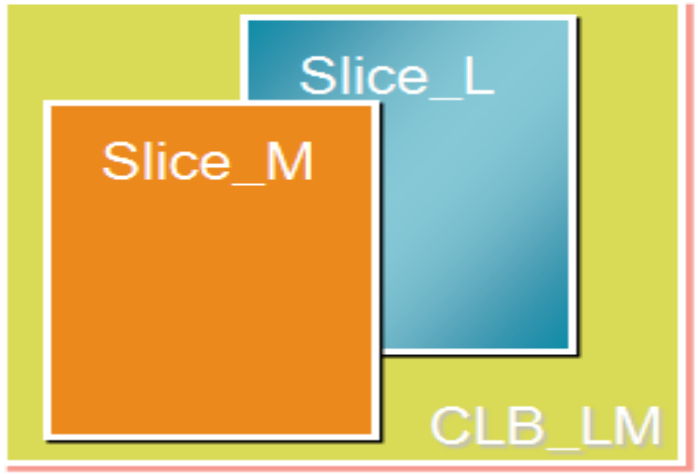
Types of CLB Slices

BRAM / FIFO

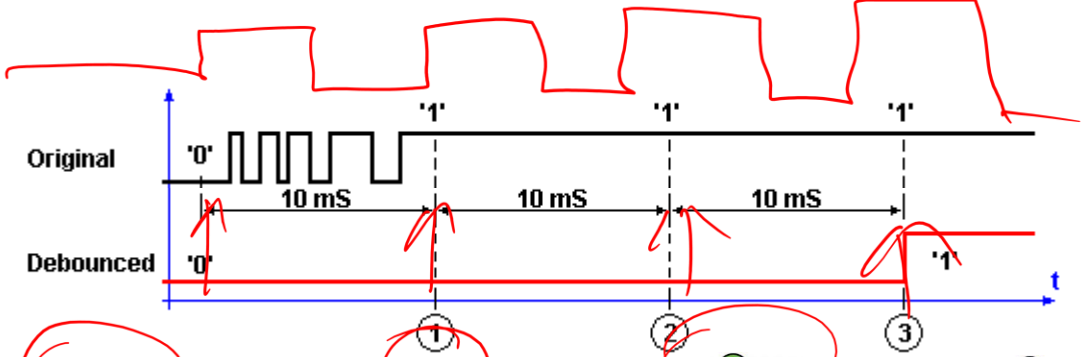
Slices	LUTs	Flip-Flops	Arithmetic and Carry Chains	Distributed RAM(1)	Shift Registers(1)
2	8	16	2	256 bits	128 bits

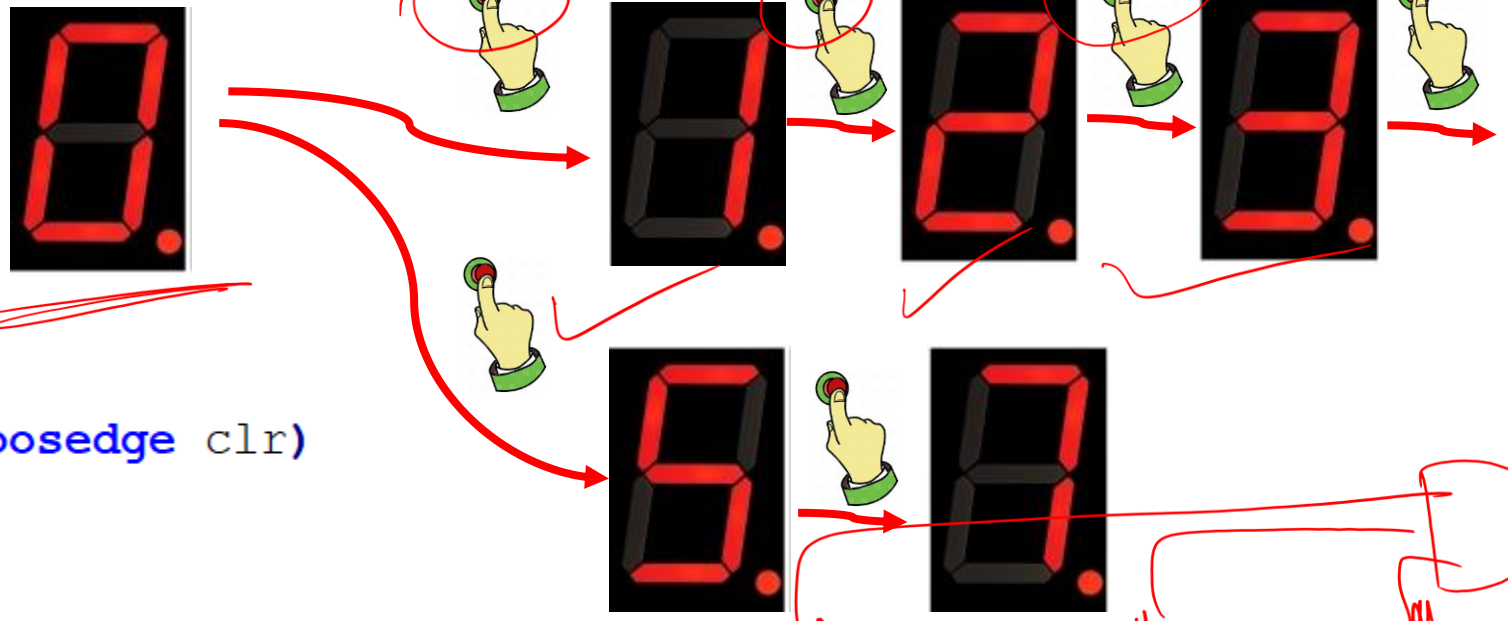
LUT ⇒ CLB



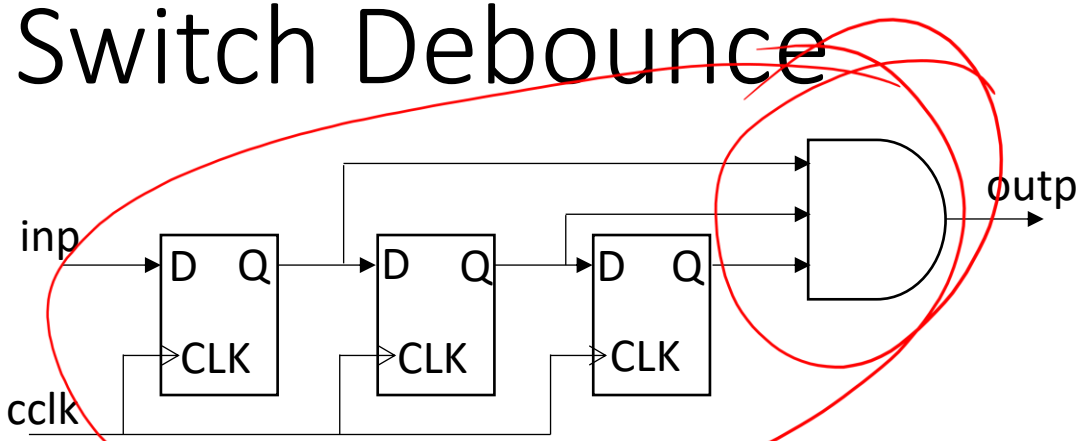


```
module count_3b (
input wire clr ,
input wire push_b ,
output reg [2:0] q
);

// 3-bit counter
always @(posedge push_b or posedge clr)
  begin
    if(clr == 1)
      q <= 0;
    else
      q <= q + 1;
  end

endmodule
```





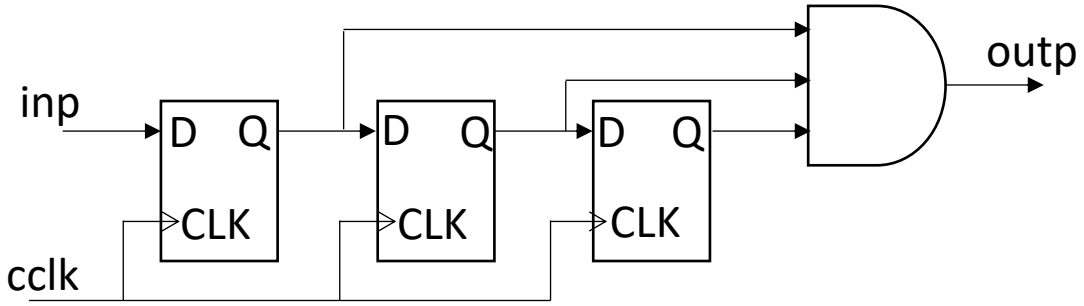
Switch Debounce

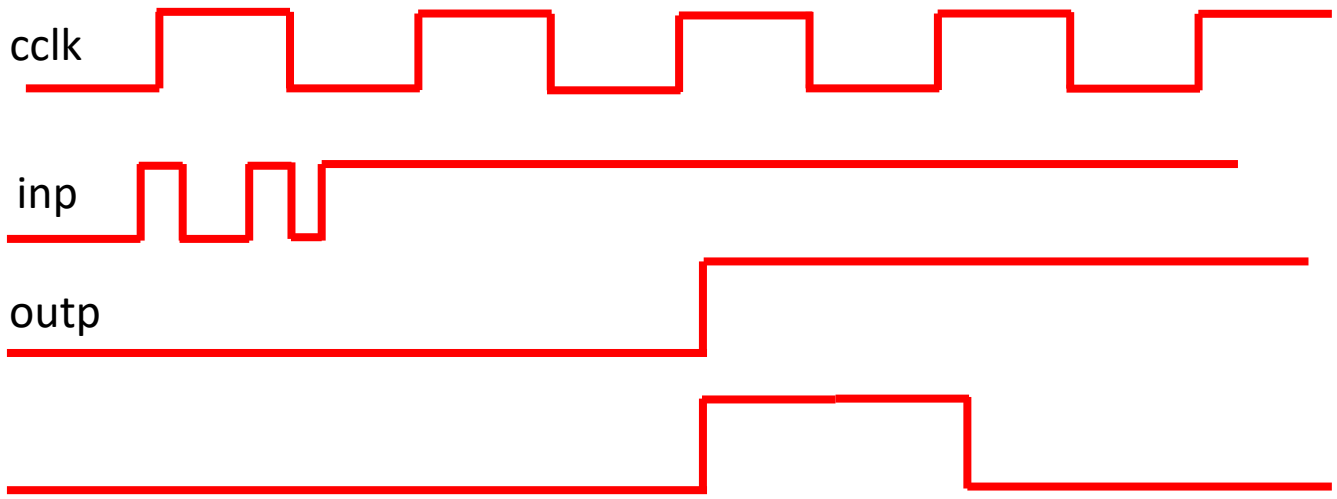


- When you press any of the pushbuttons on FPGA board, they may bounce slightly for a few milliseconds before settling down.
- This means that instead of the input to the FPGA going from 0 to 1 cleanly, it may bounce back and forth between 0 and 1 for a few milliseconds.
- Clock, cclk, frequency must be low enough that the switch bouncing is over before three clock periods.

```
module debounce (
input wire inp ,
input wire cclk ,
input wire clr ,
output wire outp
);
reg delay1;
reg delay2;
reg delay3;

always @(posedge cclk or posedge clr)
begin
    if(clr == 1)
      begin
        delay1 <= 1'b0;
        delay2 <= 1'b0;
        delay3 <= 1'b0;
      end
    else
      begin
        delay1 <= inp;
        delay2 <= delay1;
        delay3 <= delay2;
      end
end
assign outp = delay1 & delay2 & delay3;

endmodule
```

Clock Pulse

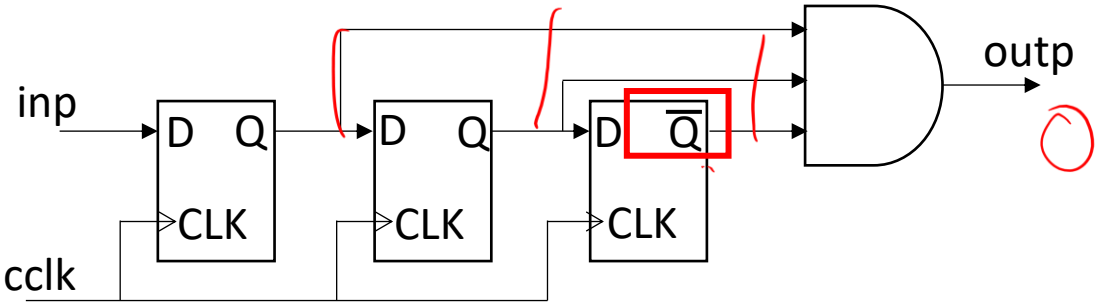


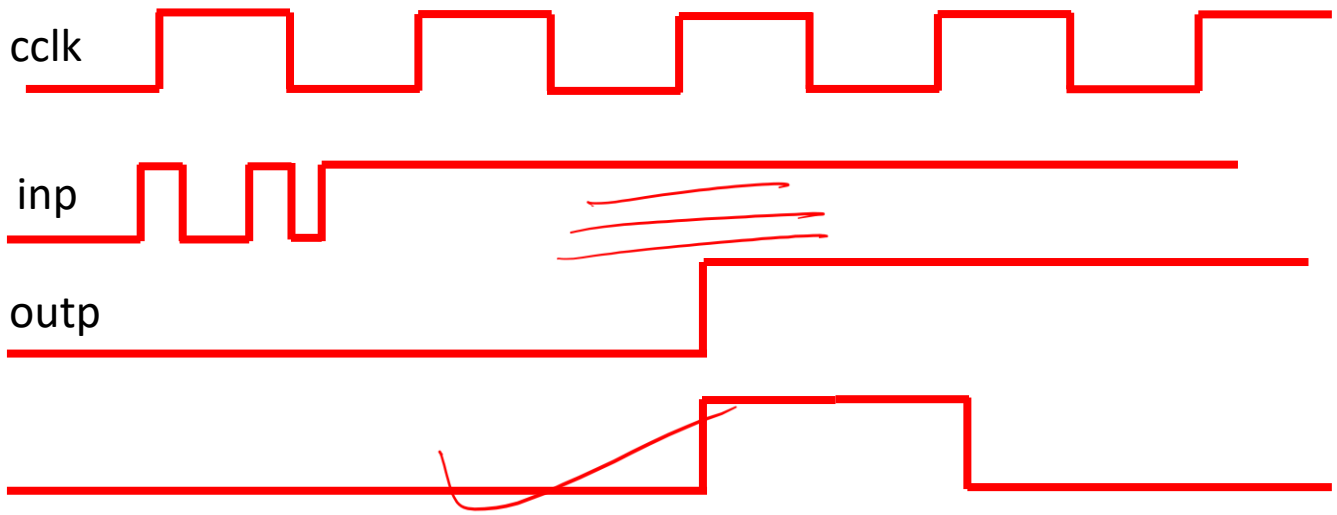



```
module debounce (
input wire inp ,
input wire cclk ,
input wire clr ,
output wire outp
);
reg delay1;
reg delay2;
reg delay3;

always @(posedge cclk or posedge clr)
begin
    if(clr == 1)
      begin
        delay1 <= 1'b0;
        delay2 <= 1'b0;
        delay3 <= 1'b0;
      end
    else
      begin
        delay1 <= inp;
        delay2 <= delay1;
        delay3 <= delay2;
      end
end
assign outp = delay1 & delay2 & delay3;

endmodule
```

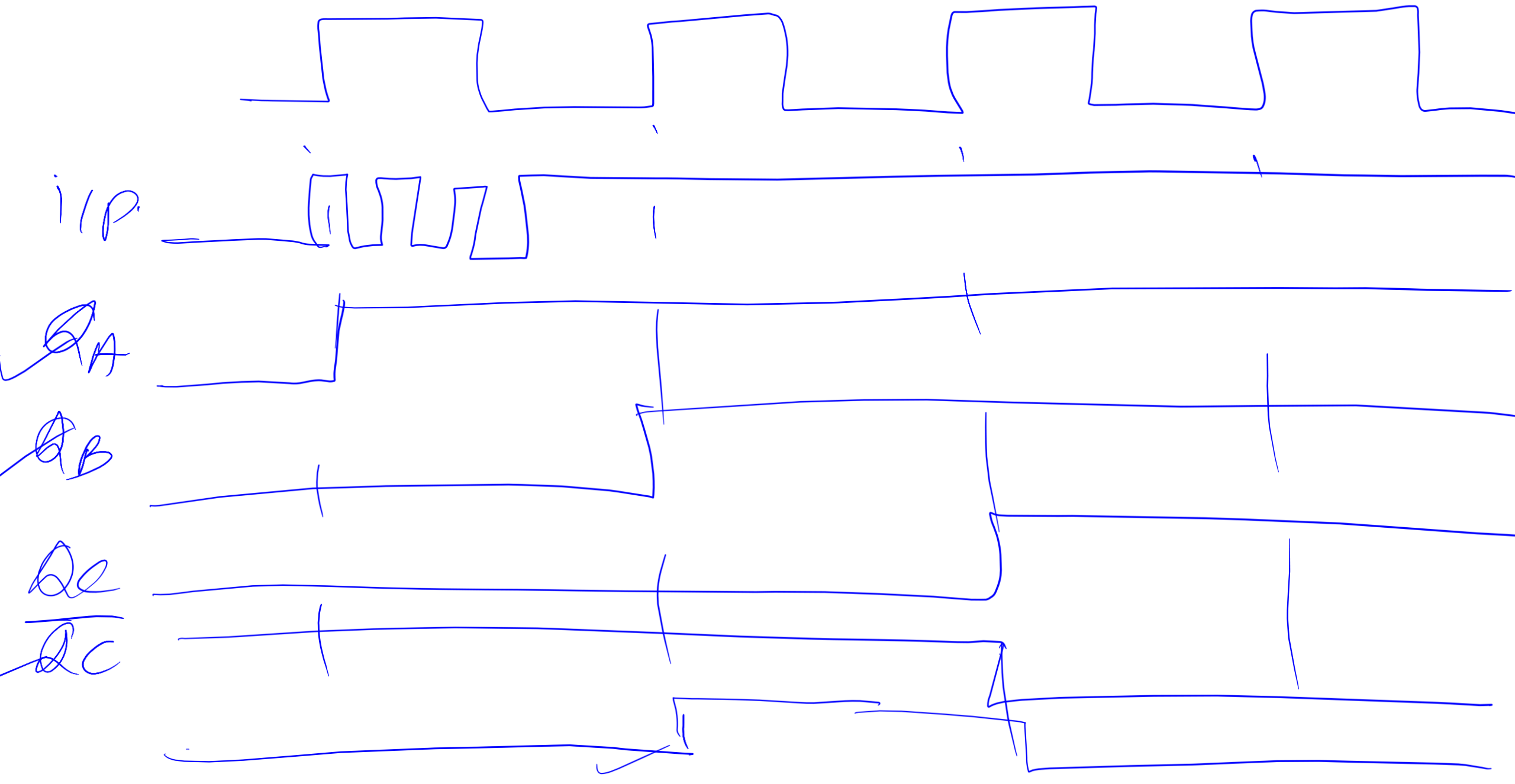
Clock Pulse

- Circuit to produce a single clean clock pulse






```
module clkp (
input wire inp ,
input wire cclk ,
input wire clr ,
output wire outp
);
reg delay1;
reg delay2;
reg delay3;

always @(posedge cclk or posedge clr)
begin
    if(clr == 1)
      begin
        delay1 <= 1'b0;
        delay2 <= 1'b0;
        delay3 <= 1'b0;
      end
    else
      begin
        delay1 <= inp;
        delay2 <= delay1;
        delay3 <= ~delay2;
      end
end
assign outp = delay1 & delay2 & delay3;

endmodule
```

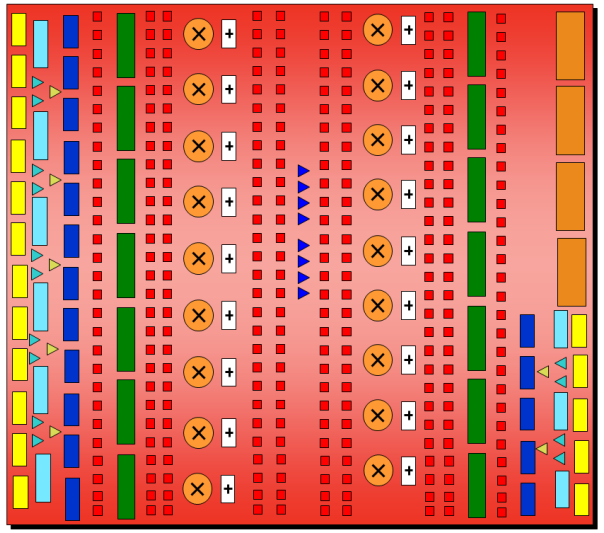

i/p
Q_A
Q_B
Q_C
$\overline{Q_C}$

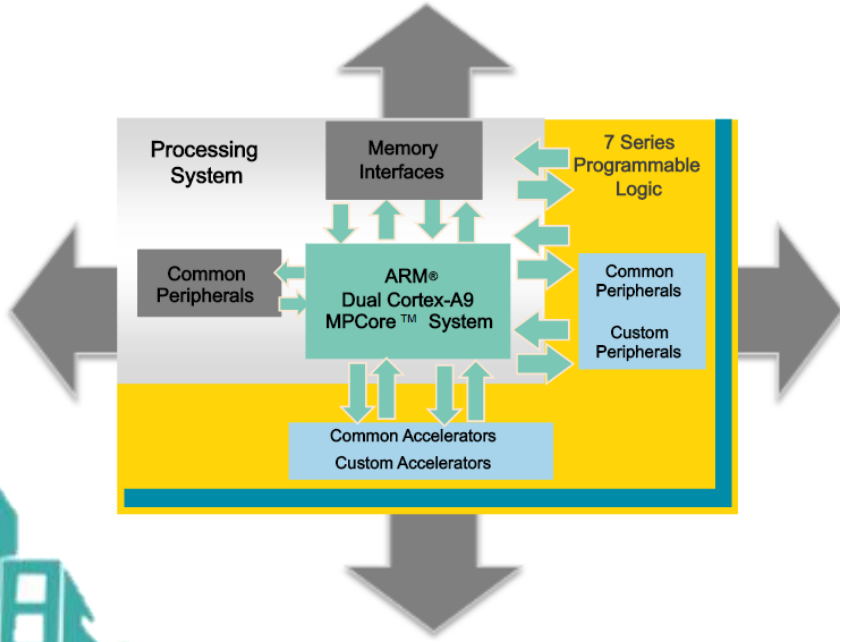
DEPARTMENT OF ELECTRONICS & COMMUNICATIONS ENGINEERING

A2A
Algorithms to Architecture

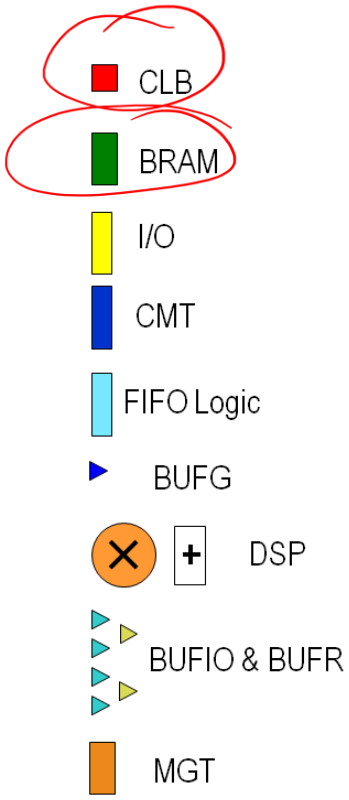
ECE 270: Embedded Logic Design

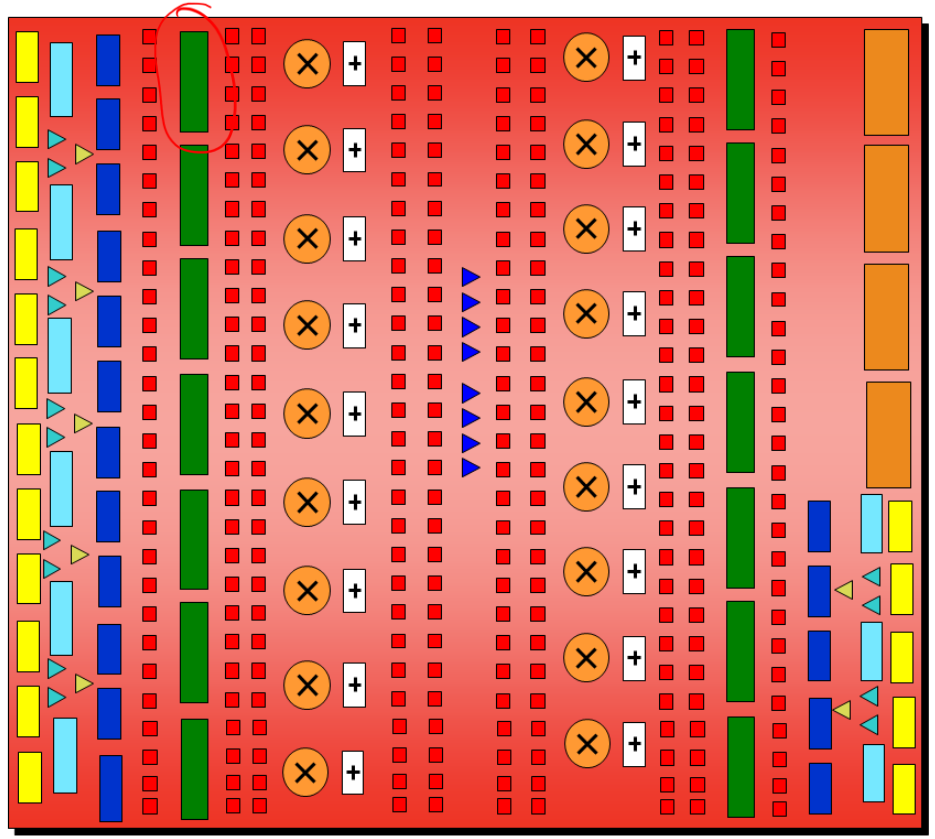
SDK
Software Development Kit
VIVADO™





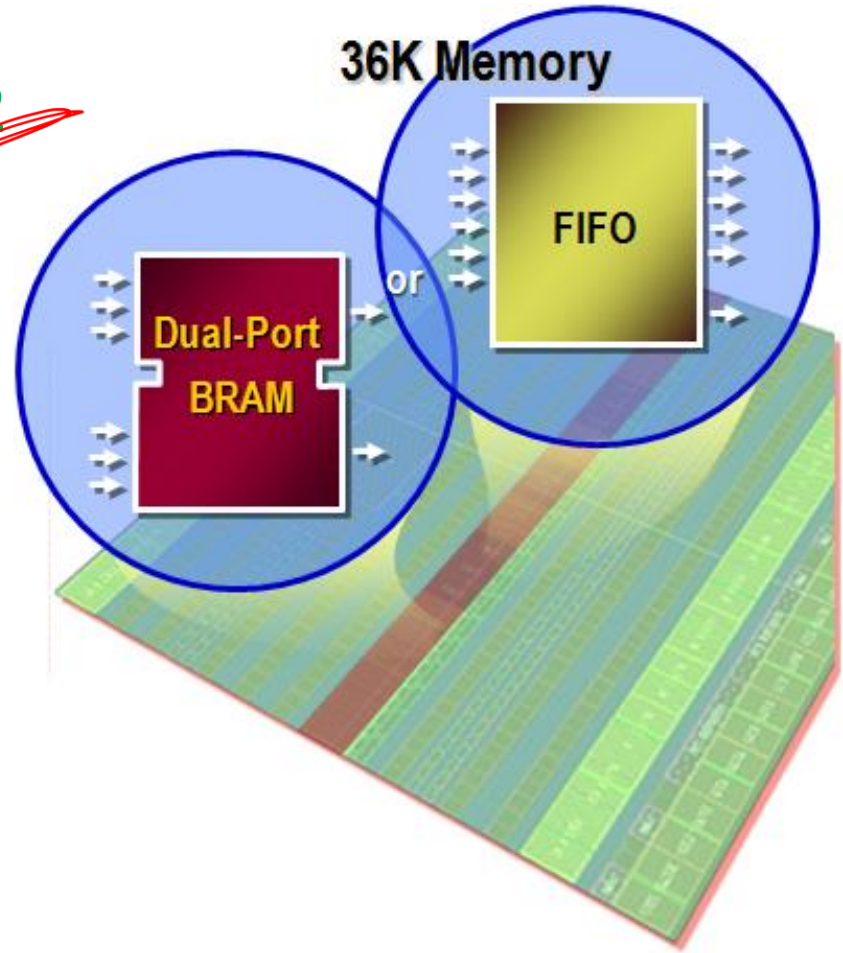
Block RAM





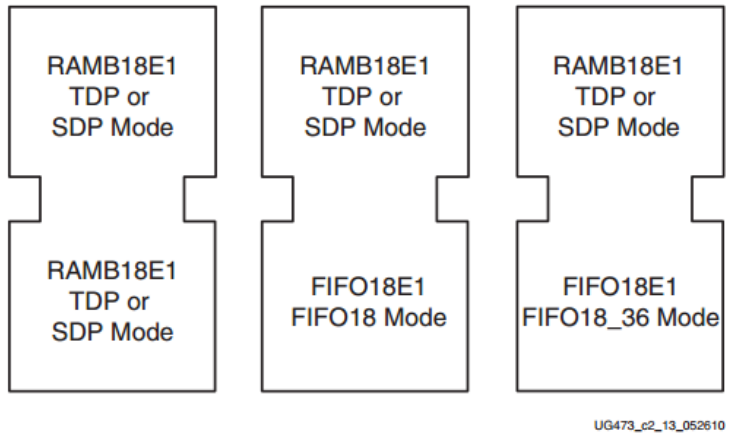
Block RAM

- All members of the 7-series families have dual-port 36 Kb block RAM with port widths of up to 72
- Fully synchronous operation
- Multiple configuration options: True dual port, simple dual port, single port

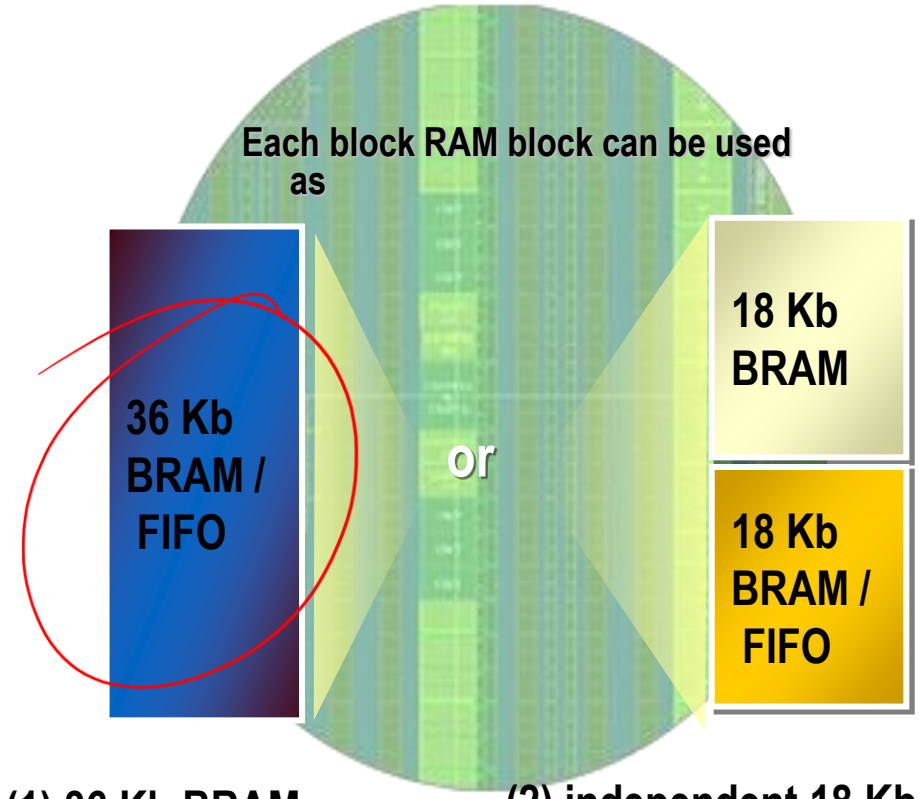


Block RAM

- Each BRAM can be segmented as:
  - 36 Kb BRAM
  - 36 Kb FIFO
  - Independent 18 Kb BRAMs
  - 18 Kb FIFO and 18 Kb BRAM



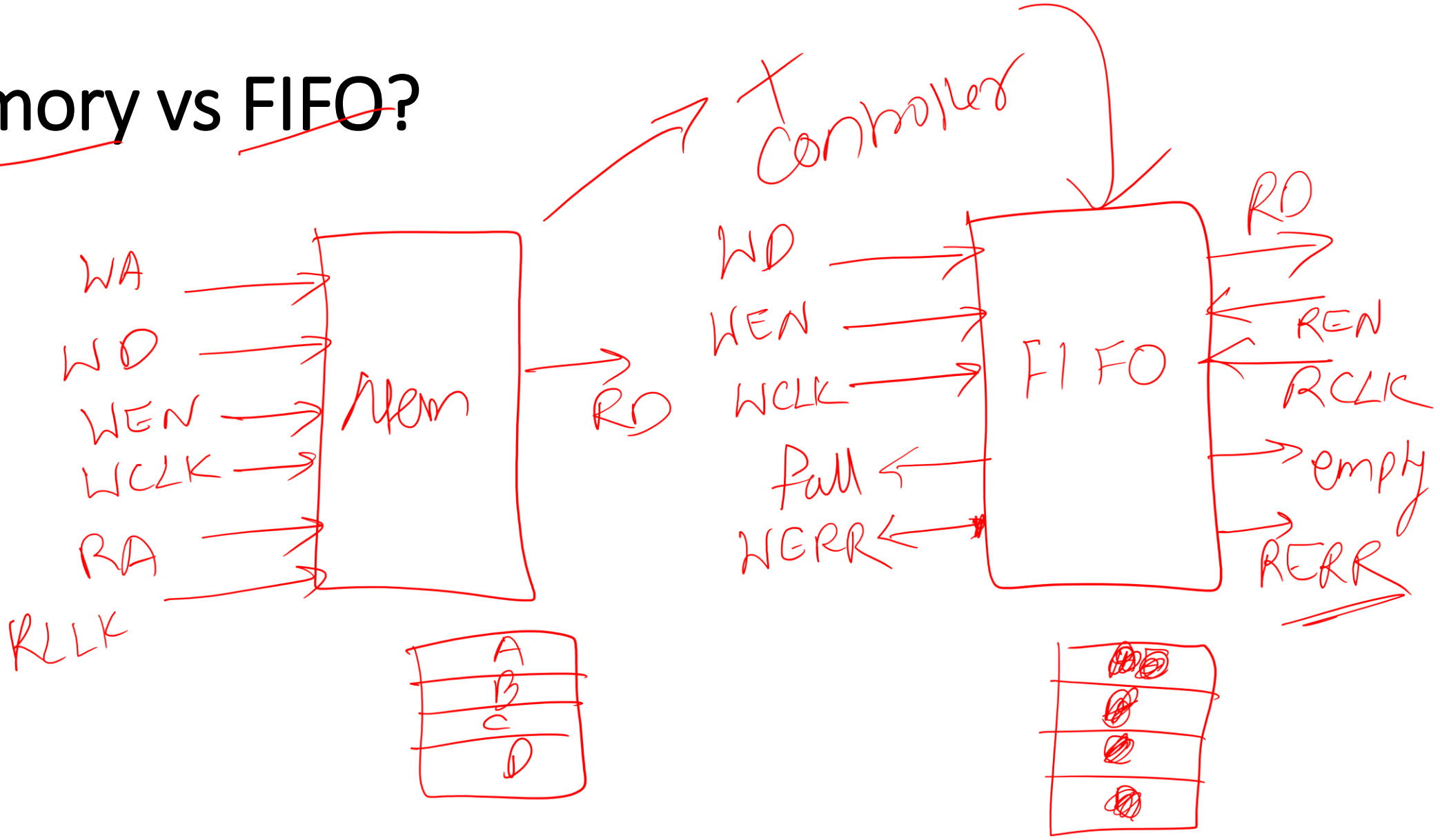
Legal Block RAM and FIFO Combinations



(1) 36 Kb BRAM OR (1) 36 Kb or FIFO

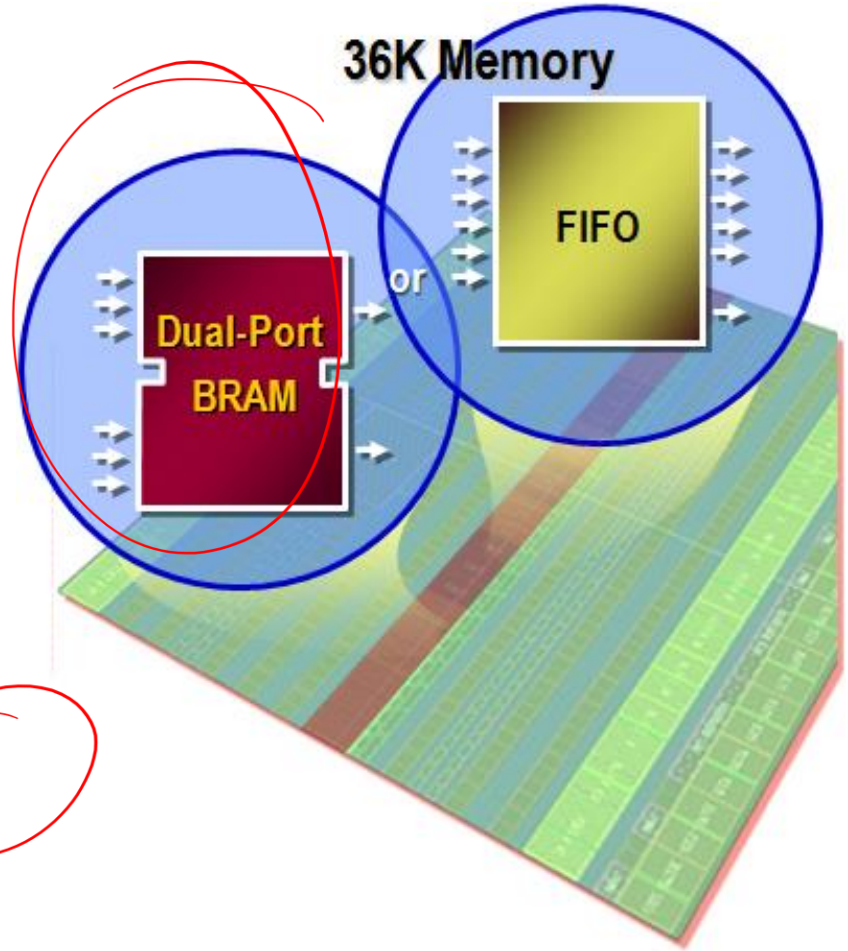
(2) independent 18 Kb block RAMs OR (1) 18 Kb FIFO + (1) 18 Kb block RAM

Memory vs FIFO?



Block RAM

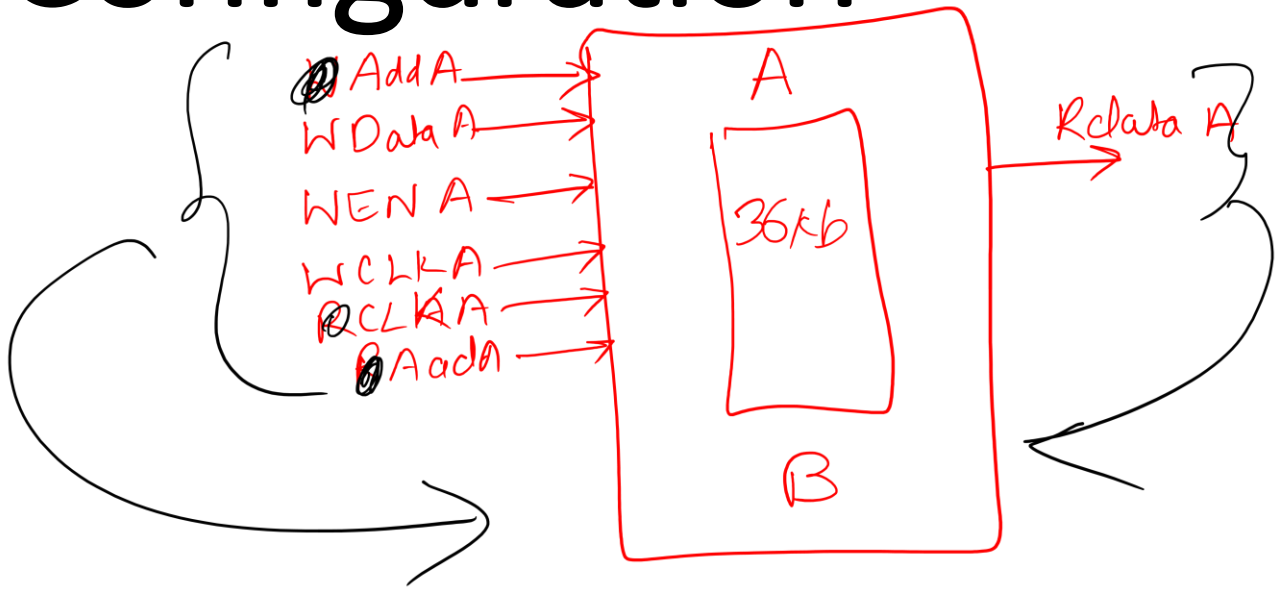
- Each BRAM can be segmented as:
  - 36 Kb BRAM
  - 36 Kb FIFO
  - Independent 18 Kb BRAMs
  - 18 Kb FIFO and 18 Kb BRAM
- Dedicated hardware to convert BRAM in to FIFO
- Integrated cascade logic to build larger memories
- Integrated error correction logic to fix bit errors*



LUT → 1-bit

BRAM - Multibit R & W

Block RAM Configuration

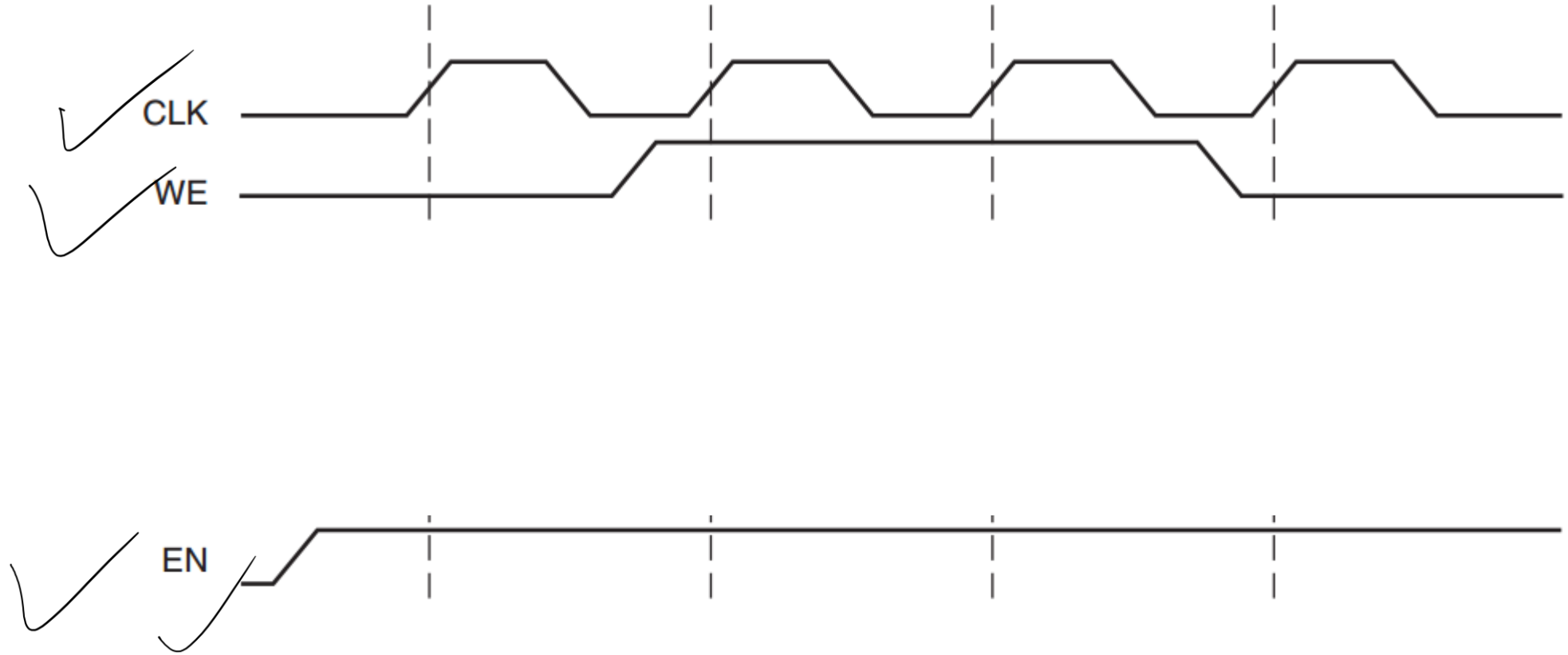


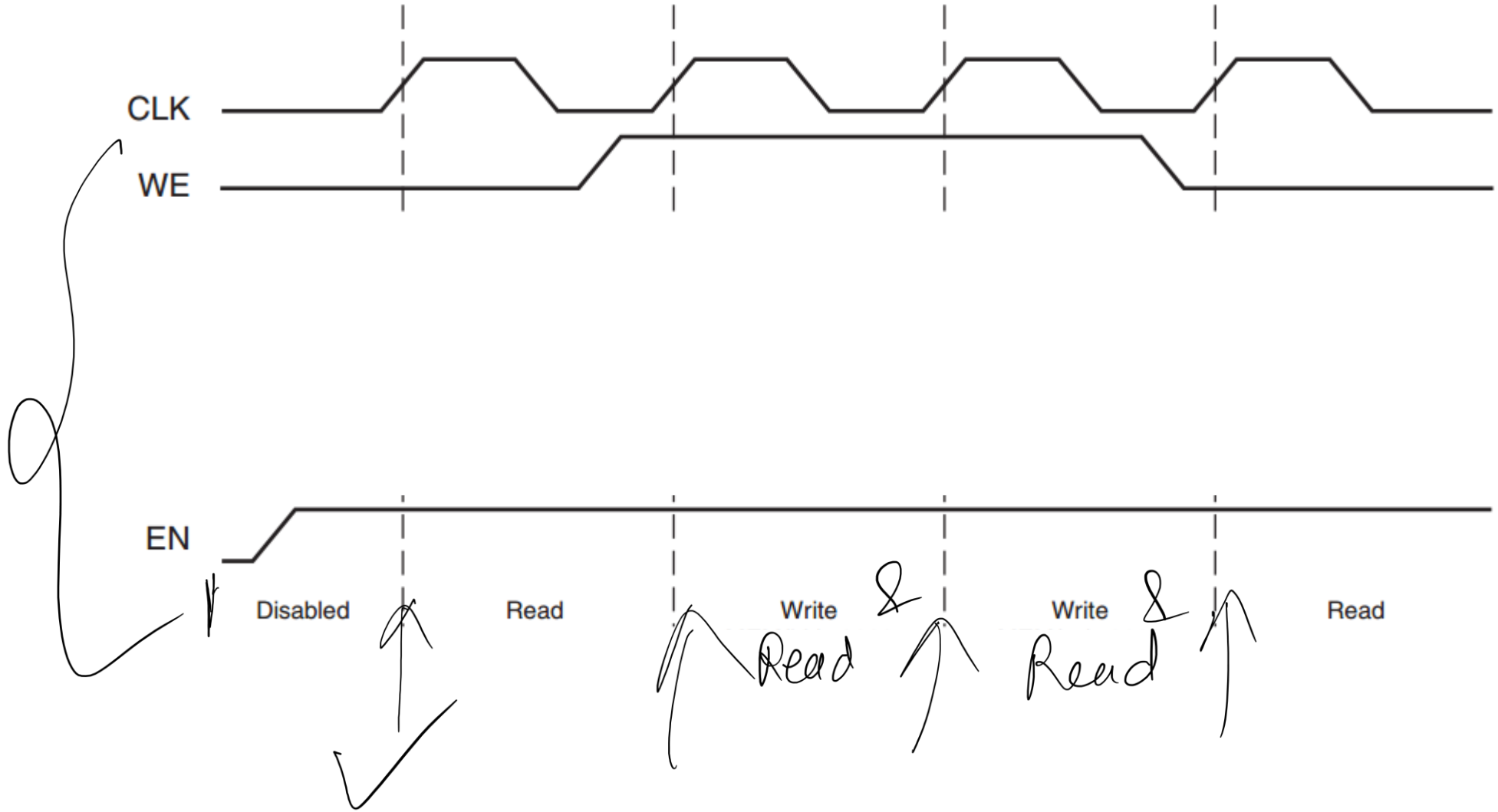
Block RAM: Ports

Block RAM: Ports

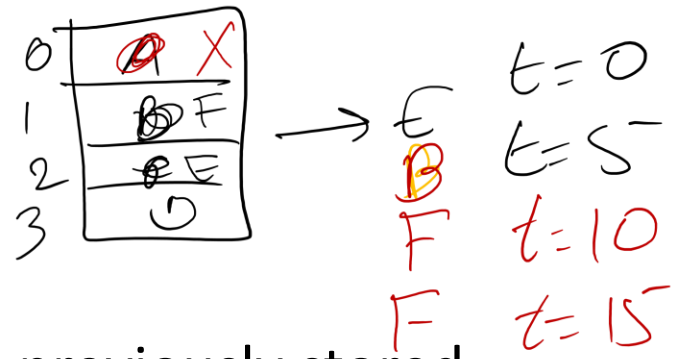
- Two ports (A and B) are symmetrical and totally independent, sharing only the stored data
- Each port can be configured in one of the available widths, independent of the other port.
- In addition, the read port width can be different from the write port width for each port.
- The memory content can be initialized or cleared by the configuration bitstream.

```
; Sample initialization file for a
; 8-bit wide by 16 deep RAM
memory_initialization_radix = 16;
memory_initialization_vector =
12, 34, 56, 78, AB, CD, EF, 12, 34, 56, 78, 90, AA, A5, 5A, BA;
```

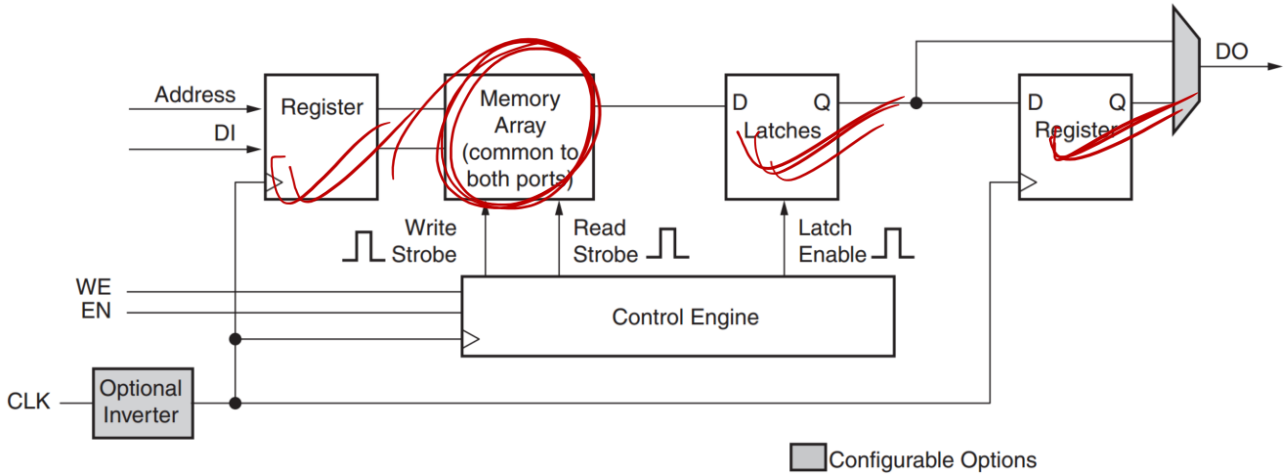

CLK
WE
EN

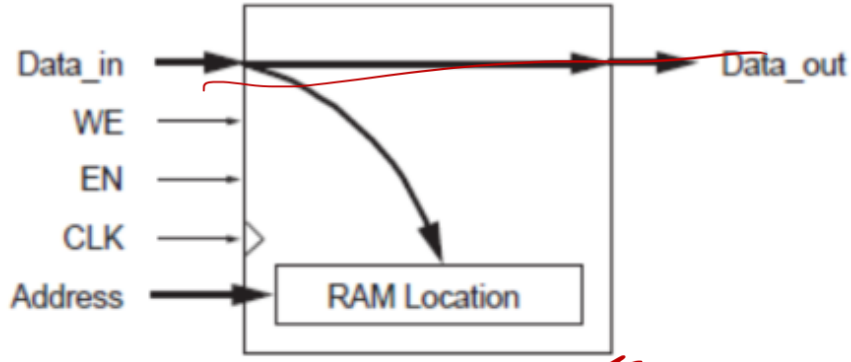

CLK
WE
EN
Disabled
Read
Write
Write
Read
Read
Read

Block RAM (Configuration Modes)



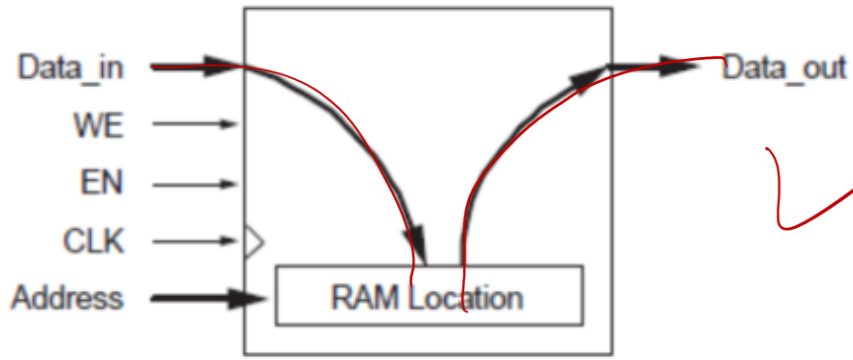
- During a write operation, the data output can reflect either the previously stored data, the newly written data, or can remain unchanged.
- Configurable write mode
  - WRITE_FIRST: Data written on DIA is available on DOA
  - READ_FIRST: Old contents of RAM at ADDRA is presented on DOA
  - NO_CHANGE: The DOA holds its previous value (saves power)





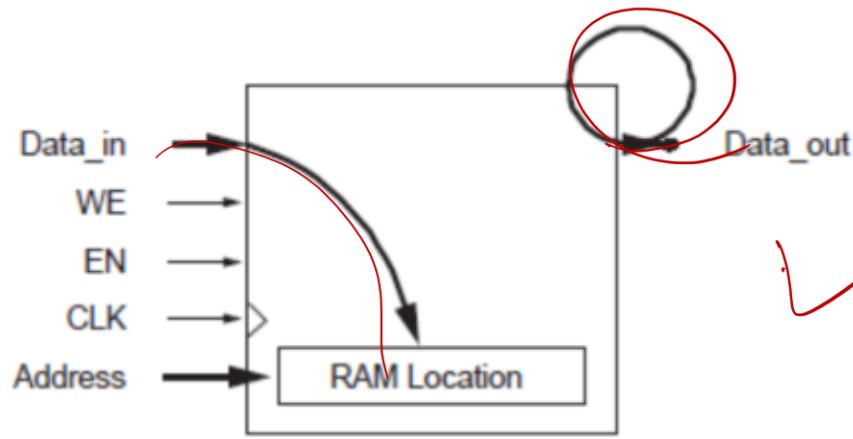
WRITE_MODE = WRITE_FIRST

X463_11_062503



WRITE_MODE = READ_FIRST

X463_13_062503

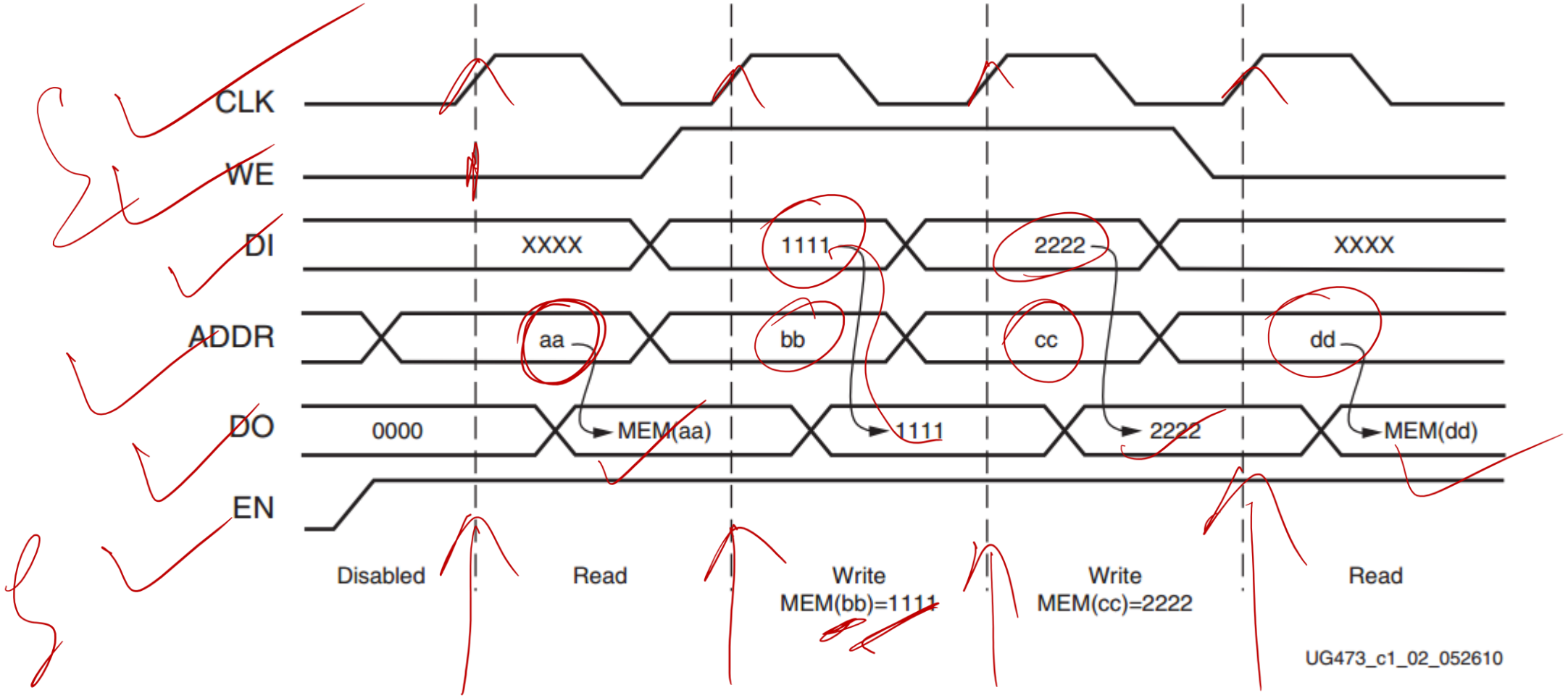


WRITE_MODE = NO_CHANGE

X463_15_062503

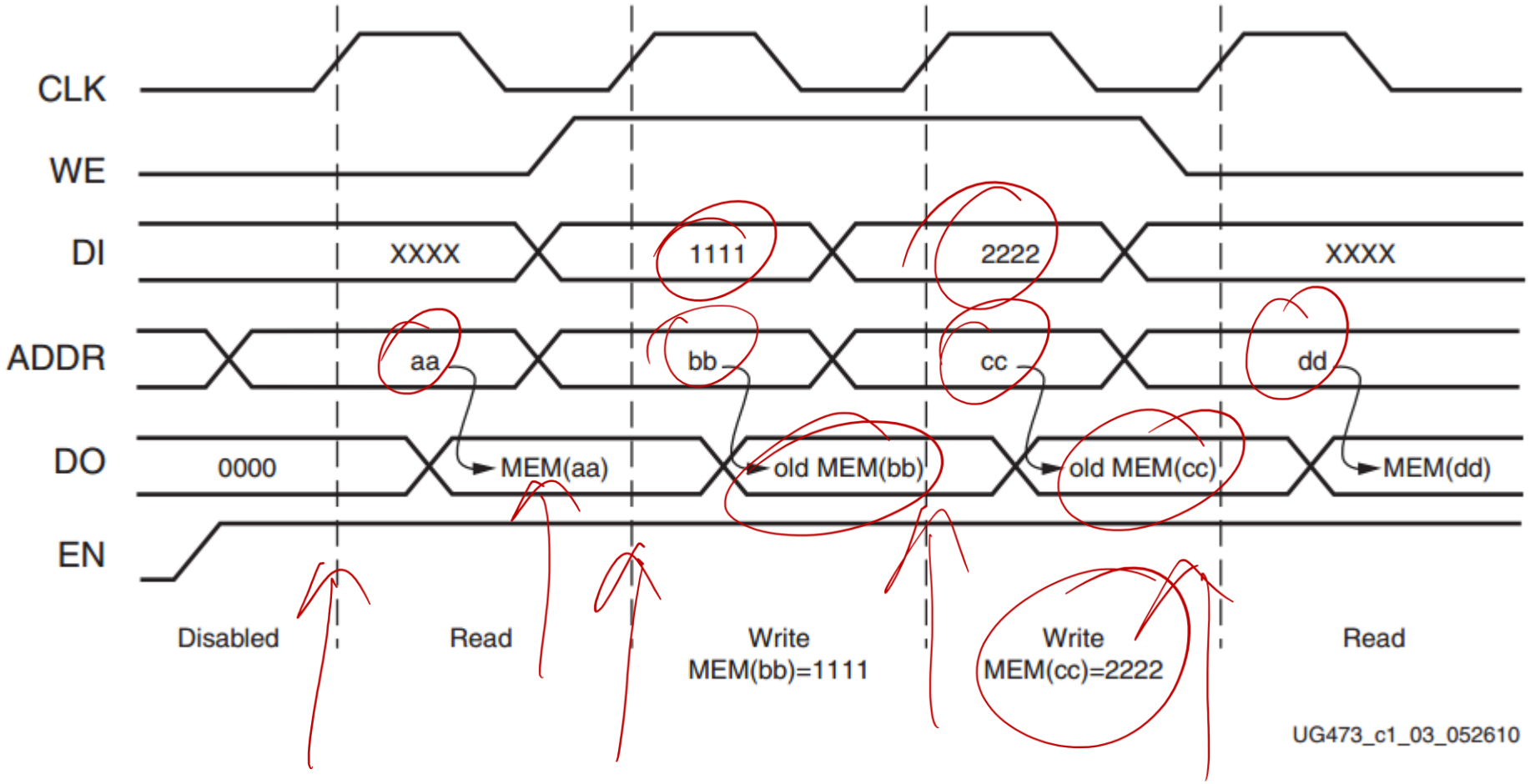
These waveforms correspond to latch mode when the optional output pipeline register is not used.

Block RAM: WRITE_FIRST



WRITE_FIRST Mode Waveforms

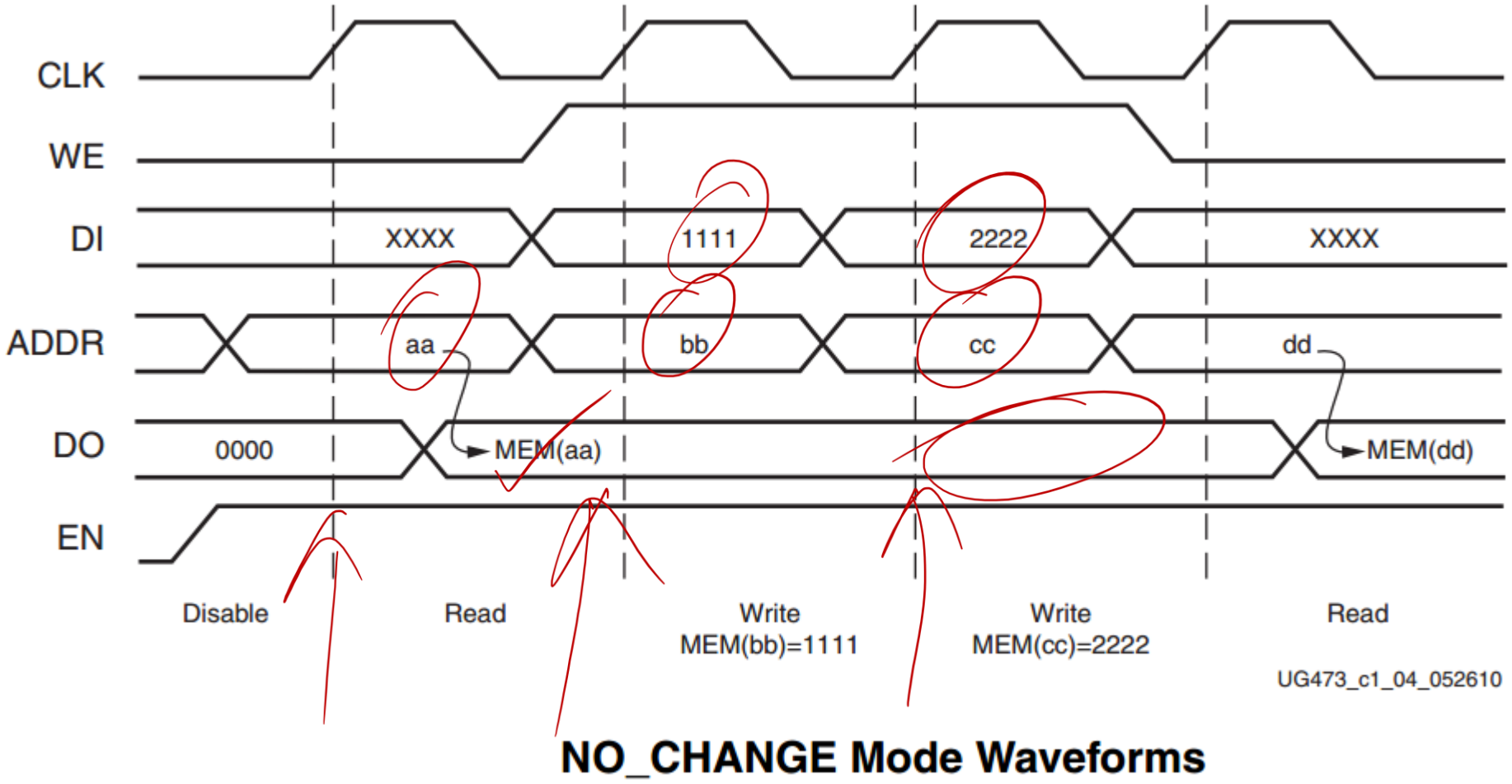
Block RAM: READ_FIRST



READ_FIRST Mode Waveforms

Block RAM: NO_CHANGE

Most power efficient



NO_CHANGE Mode Waveforms