Lecture 22
EE 421 / CS 425
Digital System Design

Fall 2025

Shahid Masud

Topics

- **Special Features in FPGA**
- Sequential Implementation on CLB
- Memory
- Multipliers
- DSP Slices
- FIR and Symmetric Filters

Recap and Summarize

- **Faults and Testing**
- Examples of Path Sensitization Method
- EXOR Method for Fault Generation
- What is Design for Testability?
- BIST and SCAN technique

Specialized Modules in FPGAs

- Dedicated Memory
  - Single Port and Dual Port Embedded Memory Blocks – Block RAM
- Dedicated Arithmetic Units
  - Adders, Multipliers, Multipliers – Accumulators, Fast Carry Logic
- Digital Signal Processing Blocks – DSP Slice
  - FFT Butterfly Modules, FIR / IIR Filters, ALU, Floating Point Arithmetic
  - IP Core Libraries for Encryption, Video Compression, Cloud Applications, etc.
- Embedded Microprocessors
  - PowerPC, Microblaze, NIOS, ARM, MIPS, etc.
- Content Addressable Memory (CAM)
  - used in Branch Prediction, Caches inside CPU
- More and more features keep appearing in new FPGA devices
  - High Speed Interfaces, Security Features, RISC-V Support, etc.

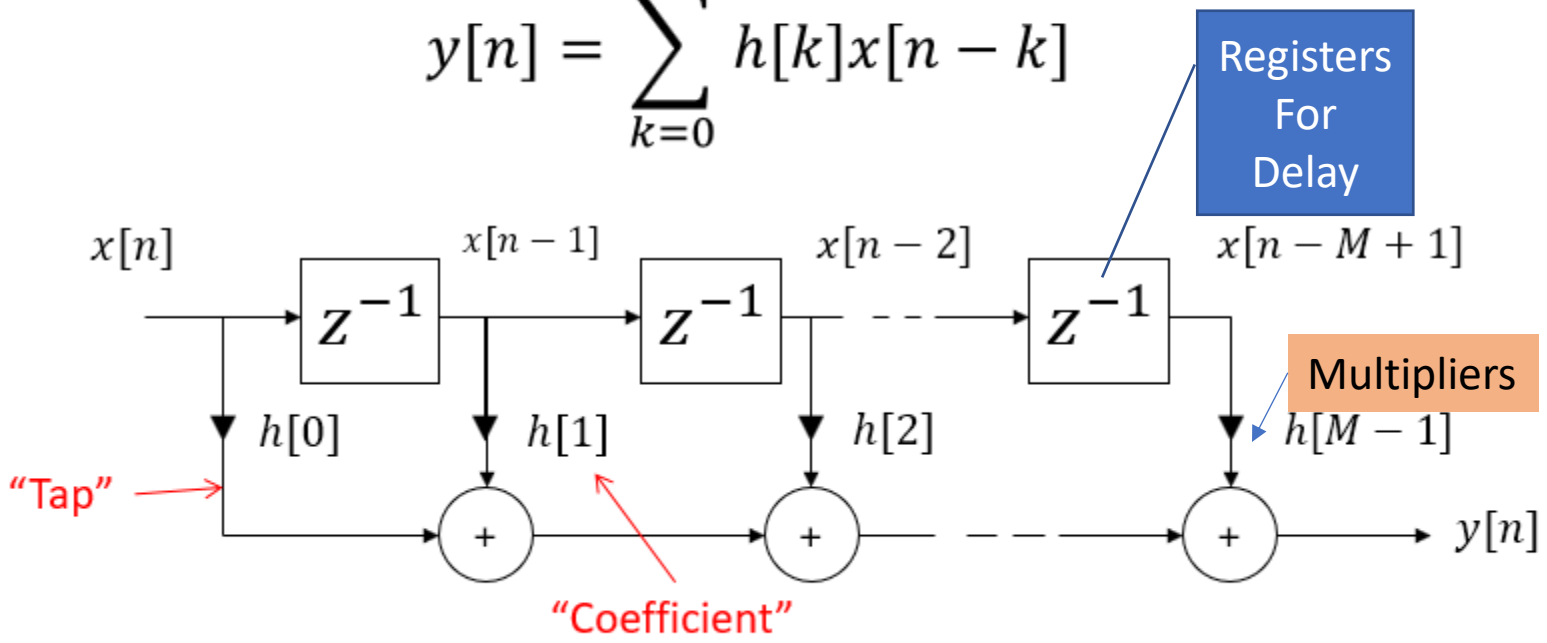
DSP Features in modern FPGA

Some Example Implementations

FIR Filter Design

- FIR system is easily implemented directly from convolution summation

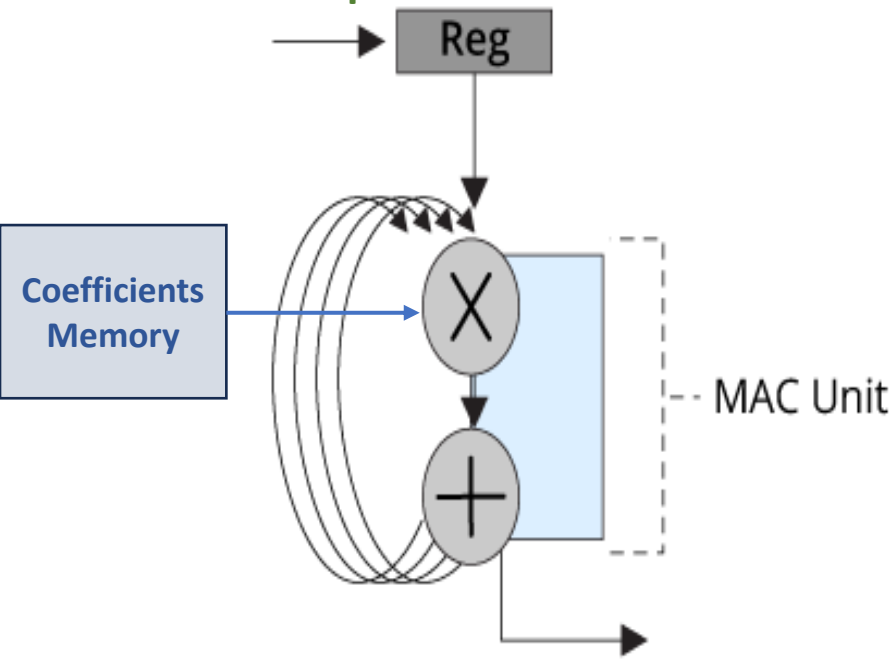
$$y[n] = \sum_{k=0}^{M-1} h[k]x[n-k]$$

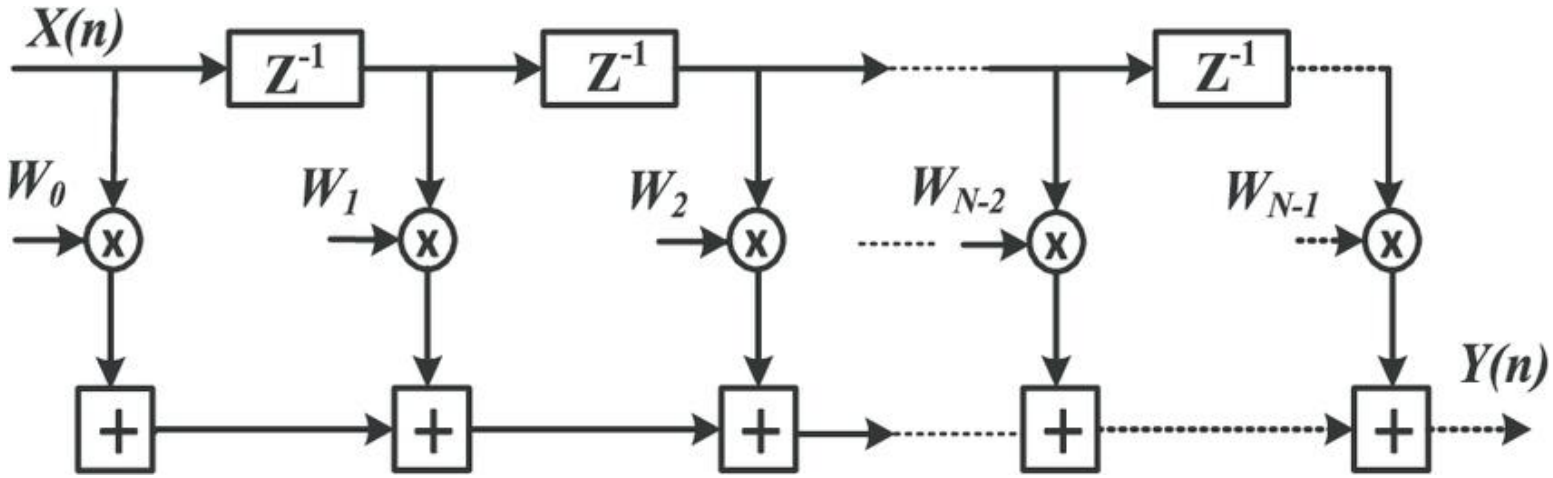


Implementation of DSP Filters

Implementation of FIR filters in Digital Signal Processing

MAC – Multiplier Accumulator

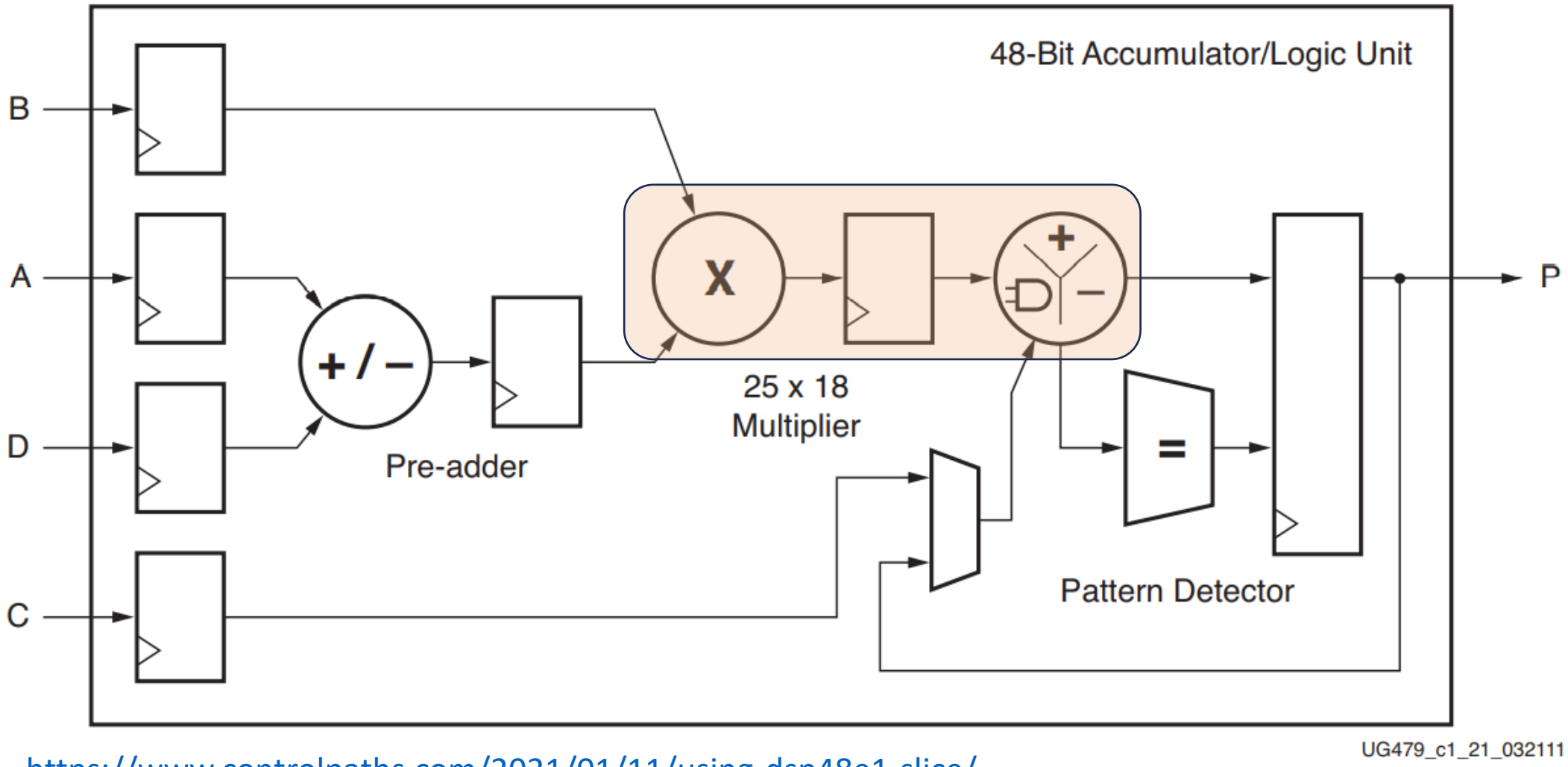




FIF filter mapping on Software Programmable Device

FIF filter mapping on a configurable Hardware Device

Basic Xilinx DSP48 Slice Architecture



https://www.controlpaths.com/2021/01/11/using-dsp48e1-slice/

MAC for Folded FIR Filter

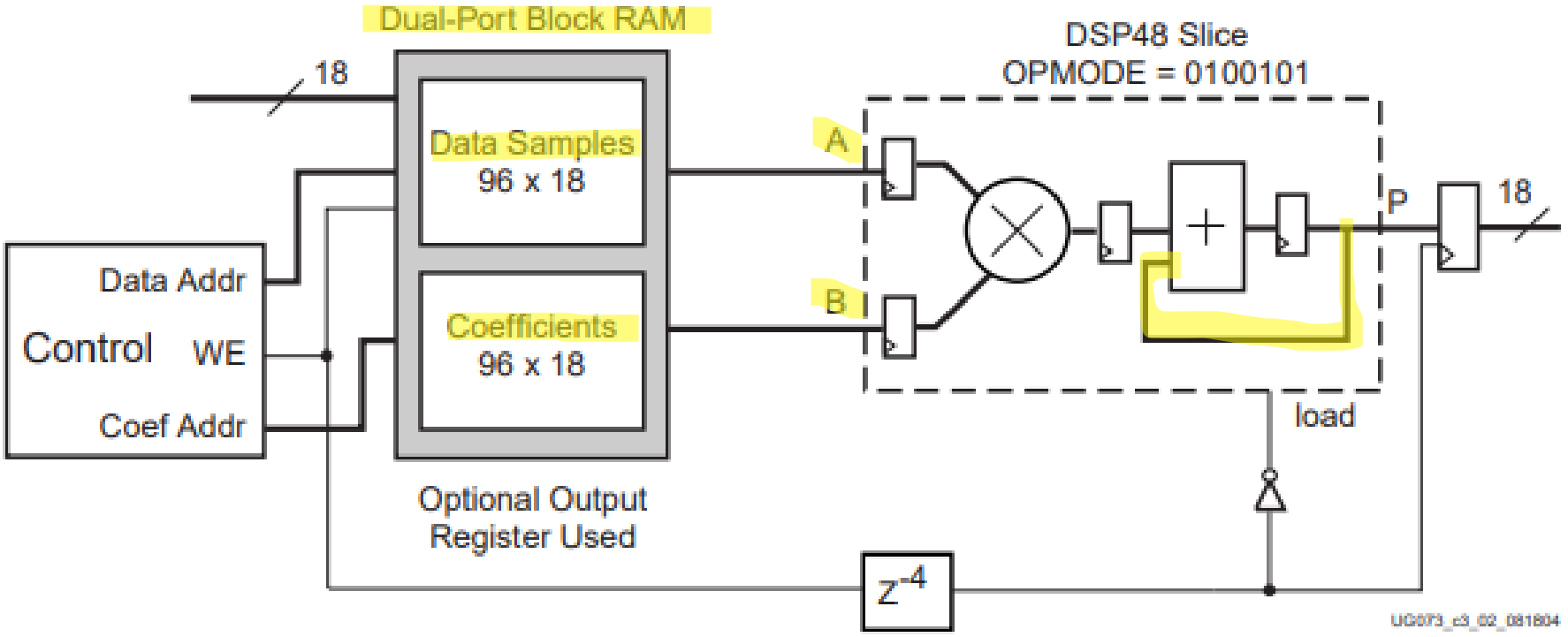


Figure 4-1: **Single-Multiplier MAC FIR Filter**

MAC Engine for FIR Filter in FPGA

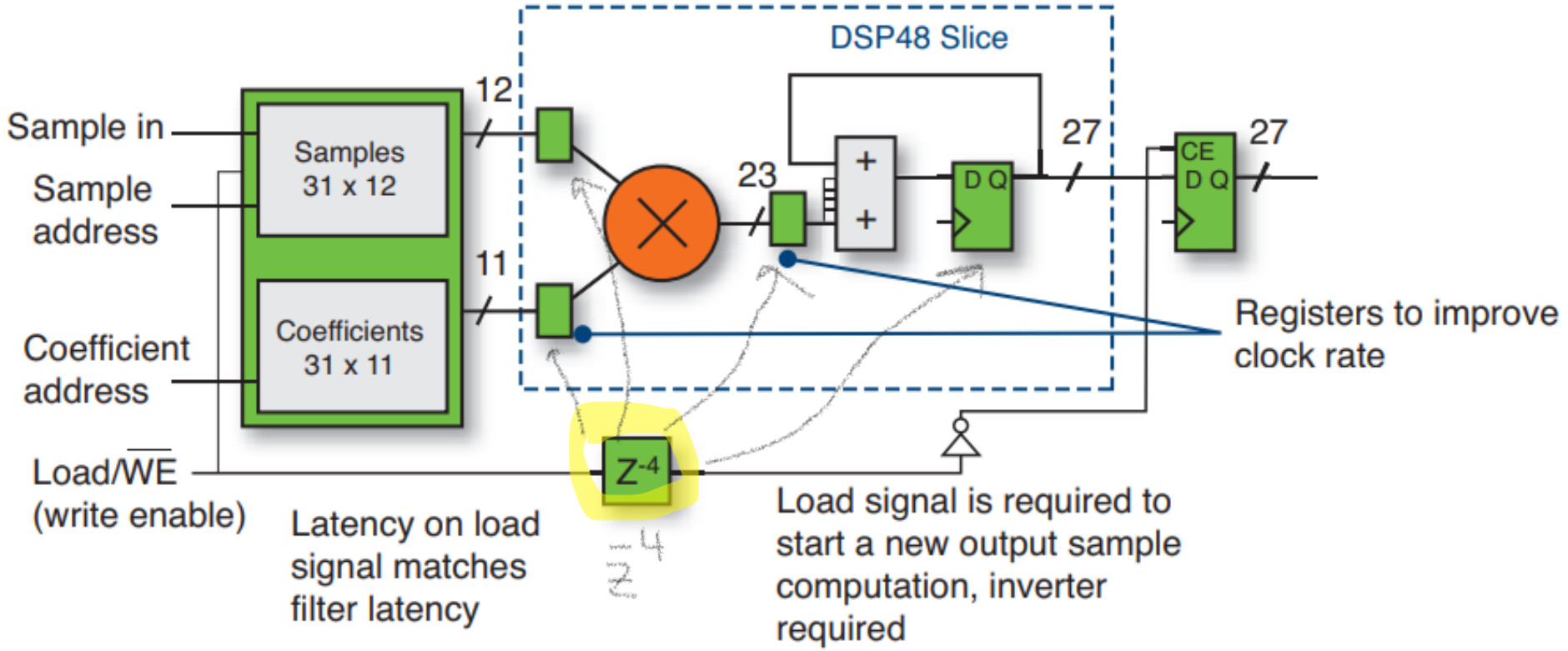


Figure 4 – MAC engine FIR filter in an FPGA

Coefficients Control

The control logic consists of two counters. One counter drives the address of the coefficient section of the dual-port block RAM, while the other controls the address for the data buffer. A comparator controls an enable to the data buffer counter to disable the count for one cycle every output sample, and writes a new sample into the data buffer every N cycles. A simplified diagram of the control logic and the memory is shown in Figure 4-3.

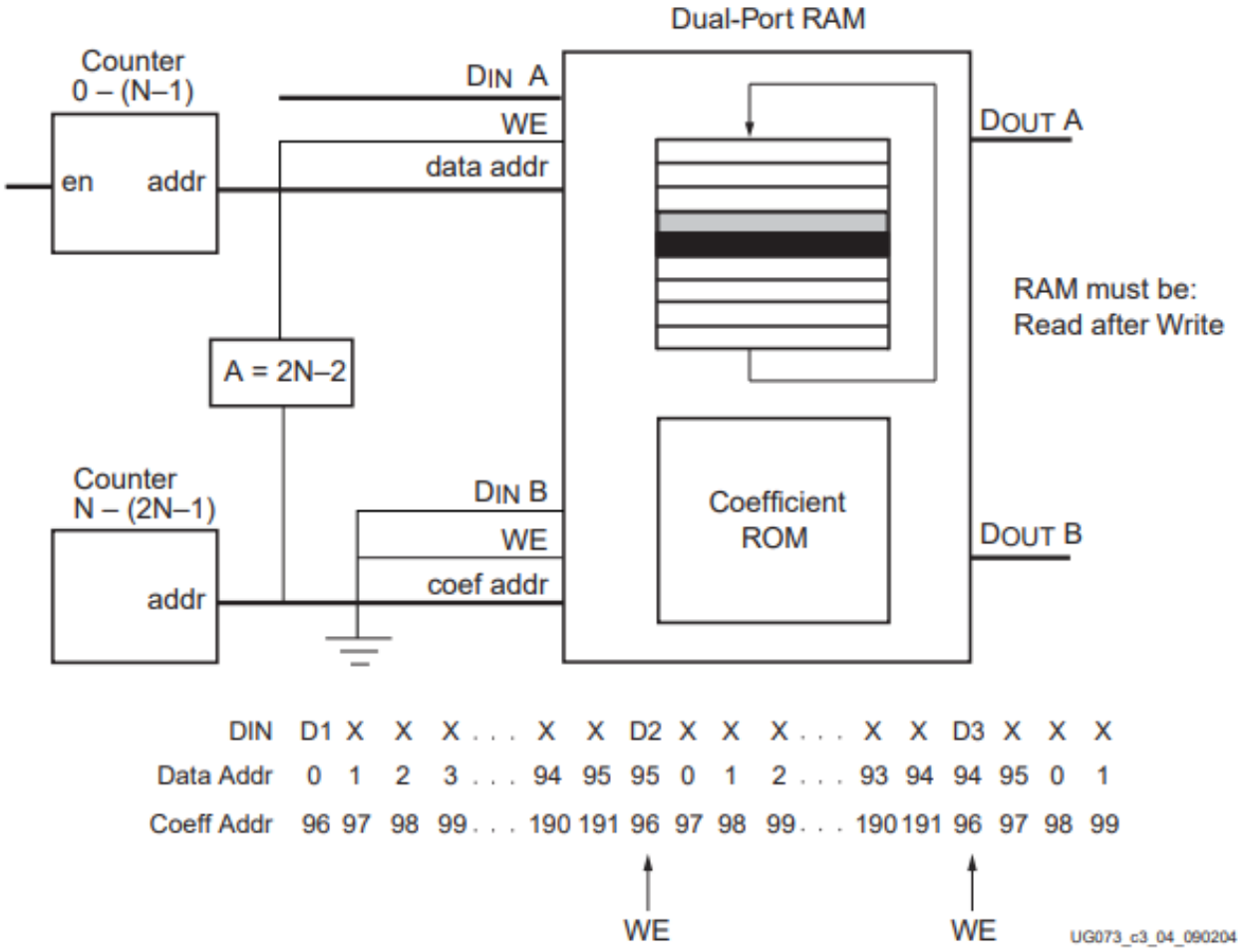


Figure 4-3: **Control Logic and Memory**

Distributed RAM MAC Based FIR

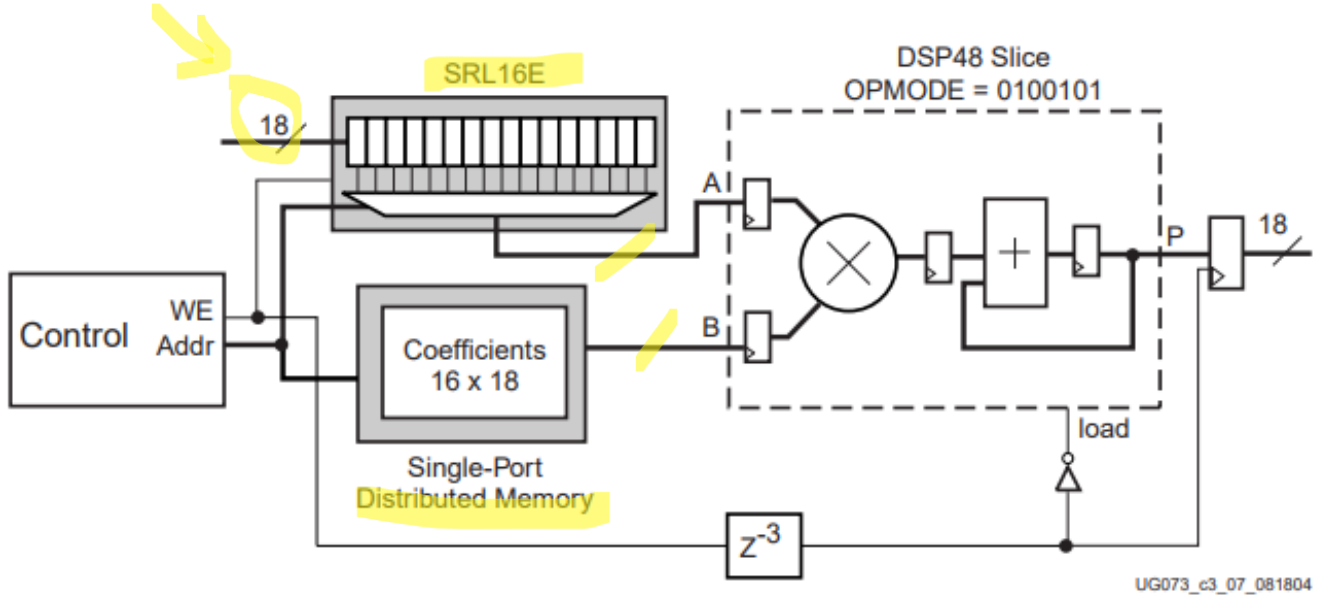


Figure 4-6: **Tap-Distributed RAM MAC FIR Filter**

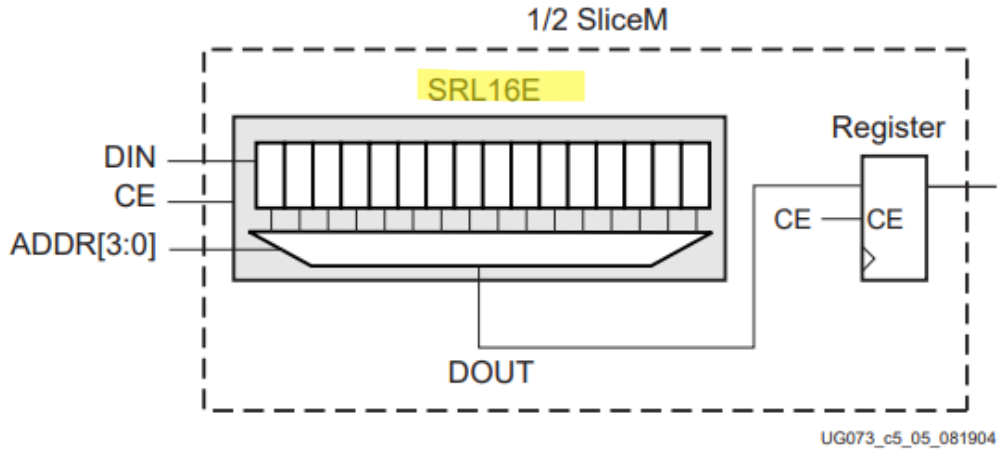


Figure 6-5: **Single Bit of One Input Memory Buffer**

Using Distributed RAM for Data and Coefficient Buffers

For smaller-sized MAC FIR filters (typically those under 32 taps), it can be considered wasteful to use block RAM as a means to store filter input samples and coefficients. Using block RAM for a 16-tap, 18-bit filter, for example, only uses up to 3% of the memory block. Block RAMs are not as abundant as the smaller distributed RAMs found inside the slice, making them an excellent option for smaller FIR filters. Figure 4-6 illustrates the MAC FIR filter implementation using distributed RAM for the coefficient bank and an SRL16E for the data buffer.

- Use SRL16s/SRL32s in the CLB and block RAM to store filter coefficients or act as a register file or memory elements in conjunction with the DSP48E2 slice. The bit pitch of the input bits is designed to pitch match the CLB and block RAM.

Block RAM and DSP Slice

Two DSP48E2 slices with a dedicated interconnect form each DSP tile (see Figure 1-2). The DSP tiles stack vertically in a DSP48E2 column. The height of a DSP tile is the same as five configurable logic blocks (CLBs) and also matches the height of one 36K block RAM. The block RAM can be split into two 18K block RAMs. Each DSP48E2 slice aligns horizontally with an 18K block RAM, providing optimal connectivity between resources.

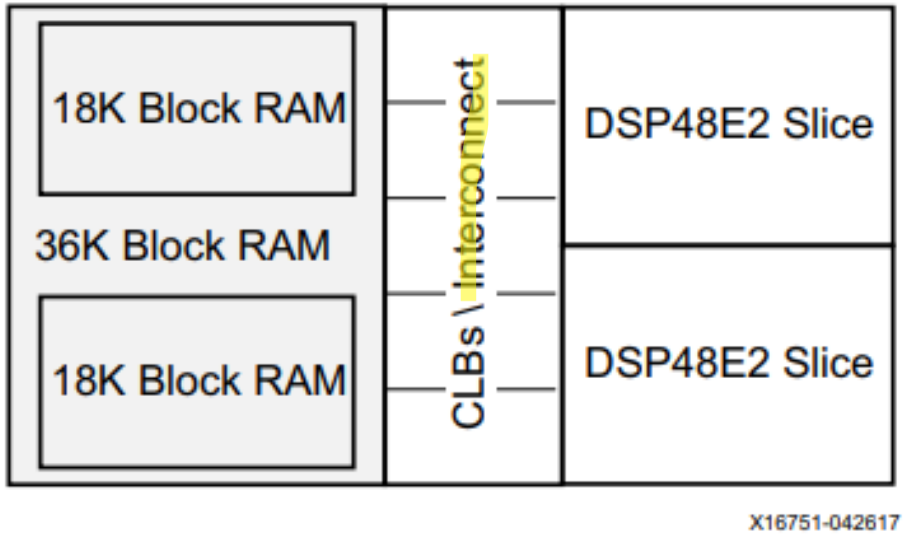
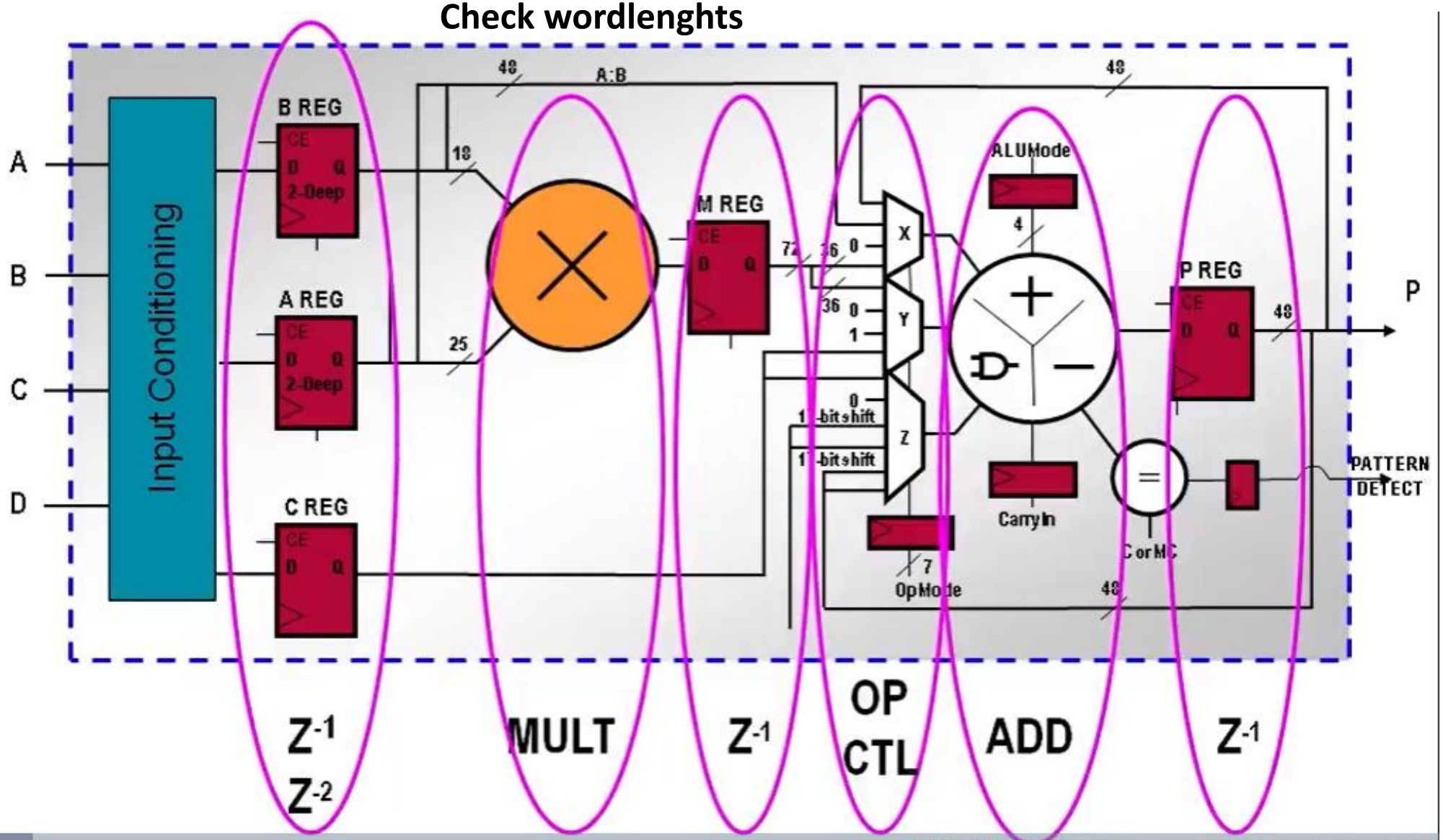


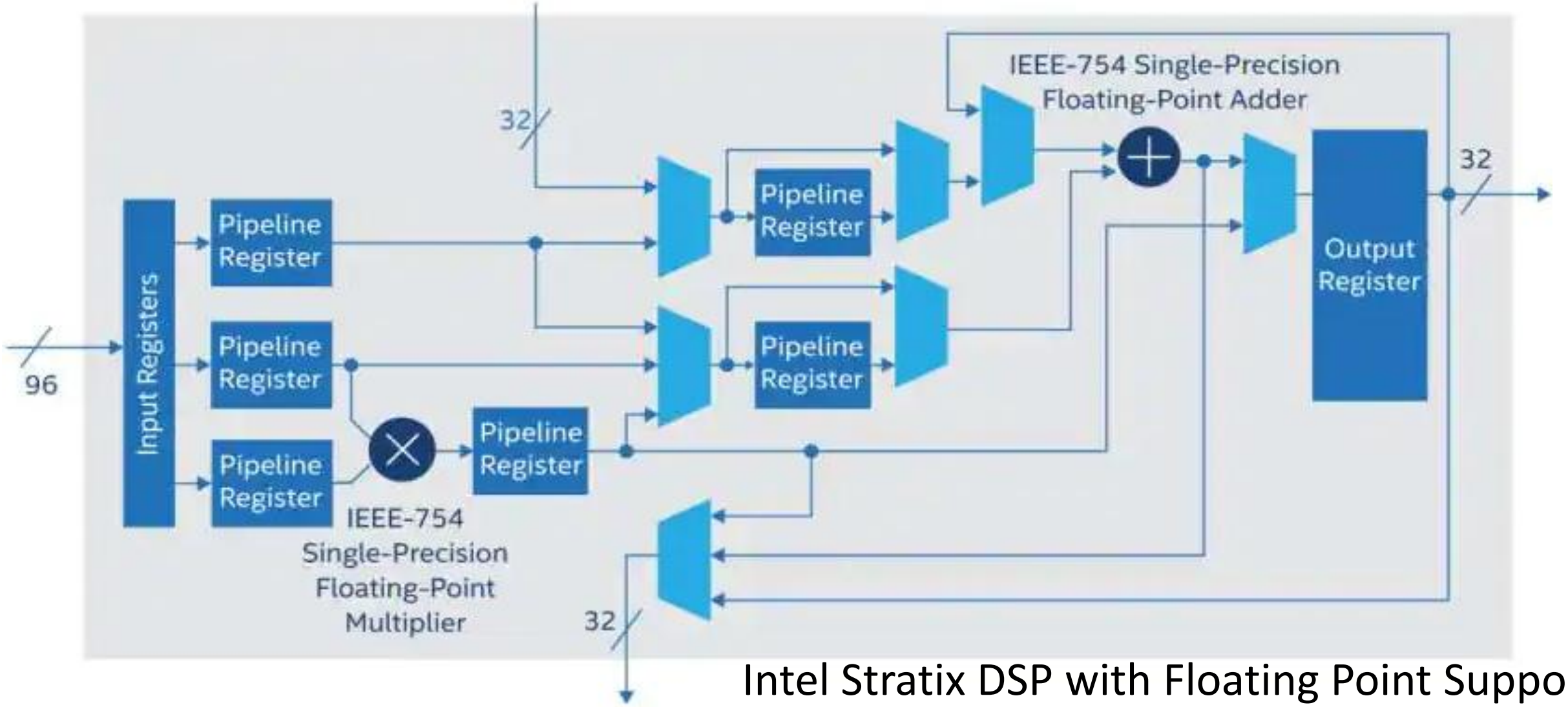
Figure 1-2: **DSP Tile**

DSP Slice Features

Check wordlenghts



Intel Stratix DSP Slice with Floating Point



Intel Stratix DSP with Floating Point Support

Intel® Stratix® 10 Device DSP Block: Single-Precision Floating Point

Complex Multiplier Approach

On multiplying two complex numbers: $z_1 = a_1 + ib_1$ and $z_2 = a_2 + ib_2$, the product obtained is written as:

$z_1z_2 = (a_1 + ib_1)(a_2 + ib_2)$

Multiplying Complex Numbers

$$(2+3i)(-6-2i)$$
$$= -12 - 4i - 18i - 6i^2$$
$$= -12 - 22i - 6(-1)$$
$$= -12 - 22i + 6$$
$$= -6 - 22i$$

Fully Pipelined, Complex, 18 x 18 Multiplier Use Model

Complex multiplication used in many DSP applications combines operands having both real and imaginary parts into results with real and imaginary parts. Two operands A and B, each having real and imaginary parts, are combined as shown in the following equations:

$$(A\_real \times B\_real) - (A\_imaginary \times B\_imaginary) = P\_real$$

$$(A\_real \times B\_imaginary) + (A\_imaginary \times B\_real) = P\_imaginary$$

The real and imaginary results use the same slice configuration with the exception of the adder/subtracter. The adder/subtracter performs subtraction for the real result and addition for the imaginary result.

Figure 2-22 shows several DSP48 slices used as a complex, 18-bit x 18-bit multiplier.

Pipeline Complex Multiplication

Fully Pipelined, Complex, 18 x 18 MAC Use Model

The differences between complex multiply and complex MAC implementations using several DSP48 slices is illustrated with the next set of equations. As shown, the addition and subtraction of the terms only occur after the desired number of MAC operations.

For N Cycles:

Slice 1 = (A_real × B_imaginary) accumulation
Slice 2 = (A_imaginary × B_real) accumulation
Slice 3 = (A_real × B_real) accumulation
Slice 4 = (A_imaginary × B_imaginary) accumulation

Last Cycle:

Slice 1 + Slice 2 = P_imaginary
Slice 3 – Slice 4 = P_real

During the last cycle, the input data must stall while the final terms are added. To avoid having to stall the data, instead of using the complex multiply implementation shown in Figure 2-23 and Figure 2-24, use the complex multiply implementation shown in Figure 2-25.

Pipeline Cplx Mult

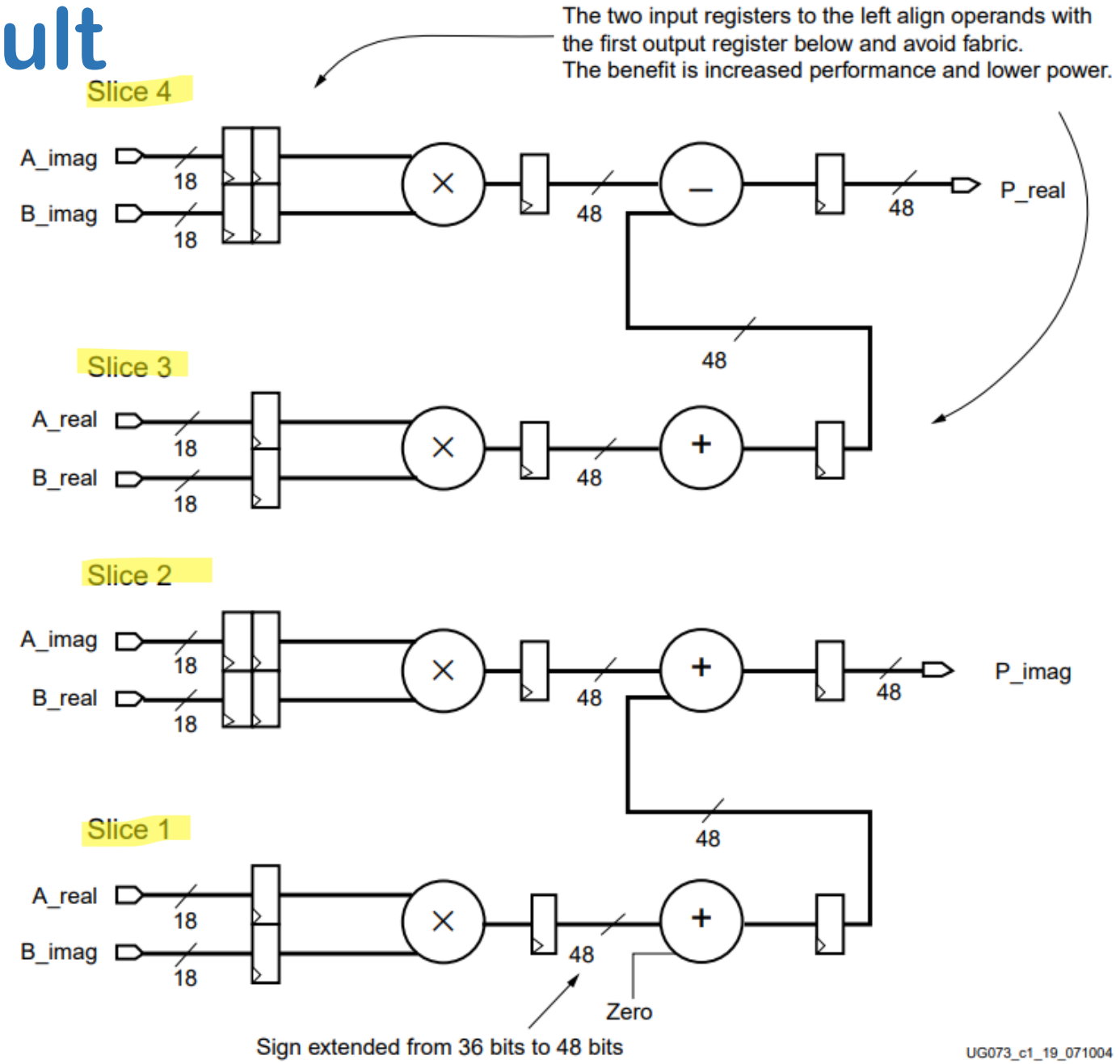


Figure 2-22: Pipelined, Complex, 18 x 18 Multiply

Pipel Cmplx Mu N c

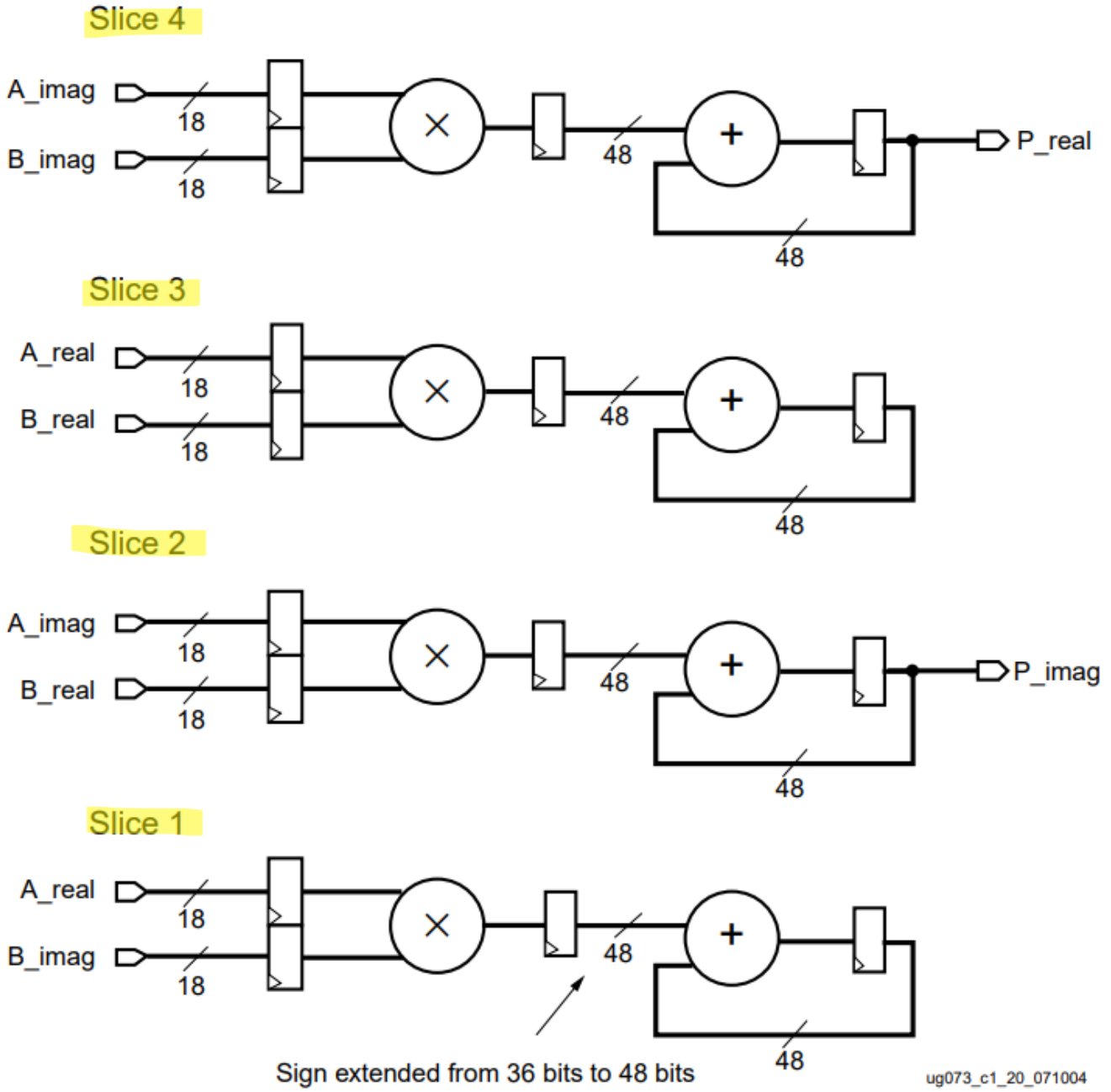


Figure 2-23: **Fully Pipelined, Complex, 18 x 18 MAC (N Cycles)**

Last Cycle

In Figure 2-24, the N+1 cycle adds the accumulated products, and the input data stalls one cycle.

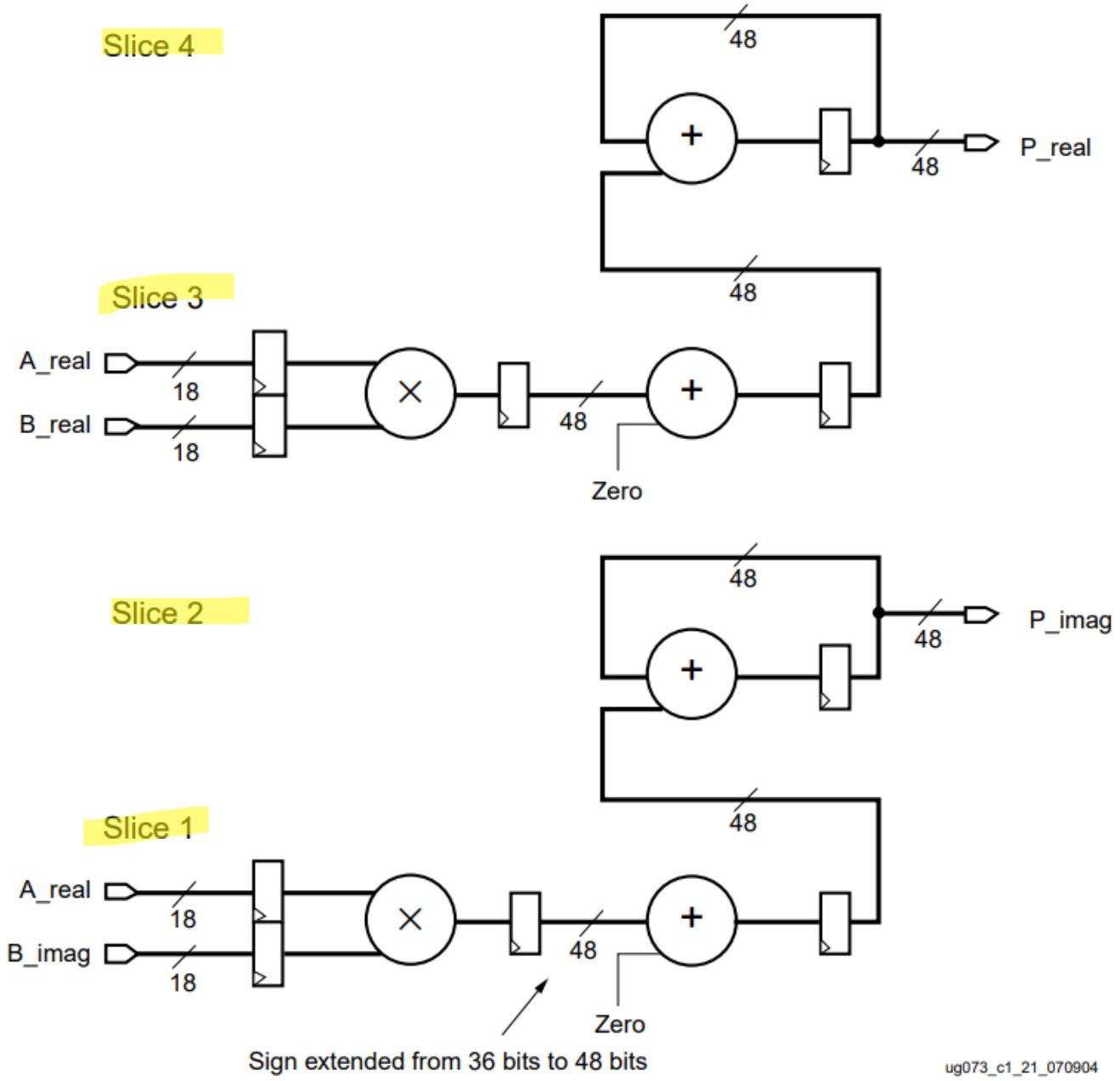


Figure 2-24: **Fully Pipelined, Complex, 18 x 18 MAC (Last or N+1 Cycle)**

Barrel Shifter Operation

2.1 Logical Right Shifter

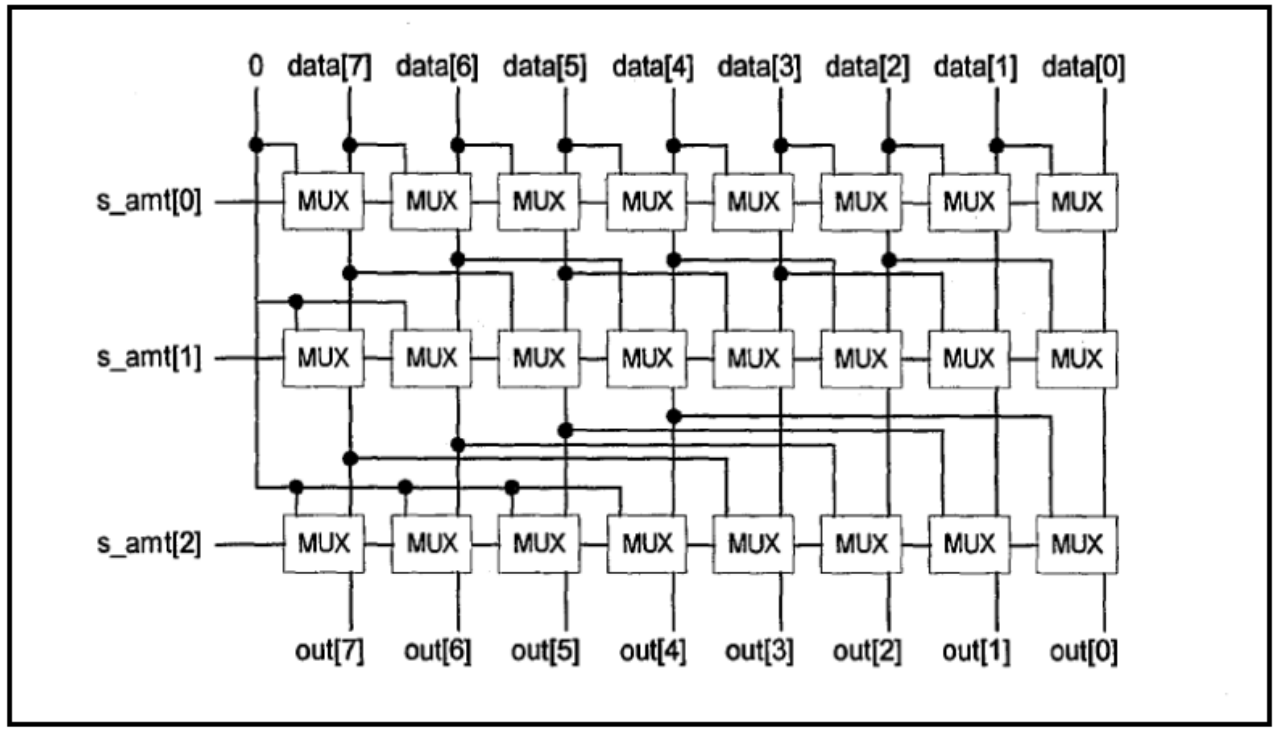


Figure 2.1 Logical Right Shifter

2.2 Right Rotator

The only difference lies in the Hardwired zeros. Instead the lost bits are rotated back in upper positions as shown below. The operation is similar to that of right shifter.

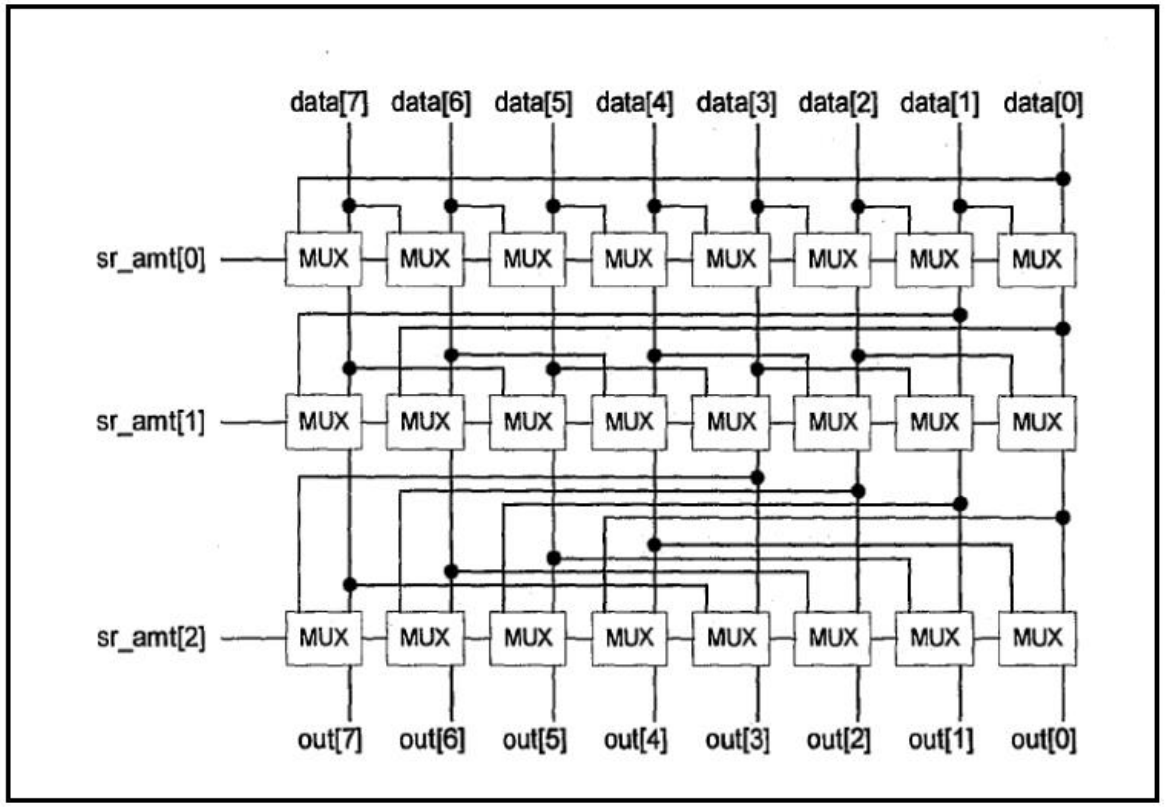


Figure 2.2 Right Rotator

https://github.com/trivedidvijen/Design-and-Optimization-of-Barrel-Shifter/blob/master/Project%20Report.pdf

Barrel Shifter Mo

Dynamic, 18-bit Circular Barrel Shifter Use Model

The barrel shift function is very useful when trying to realign data very quickly. Using two DSP48 slices, an 18-bit circular barrel shifter can be implemented. This implementation shifts 18 bits of data left by the number of bit positions represented by n. The bits shifted out of the most-significant part reappear in the lower significant part of the answer completing the circular shift. The equations in Figure 2-28 describe what value is carried out of the first slice, what this value looks like after shifting right 17 bits, and finally what is visible as a result.

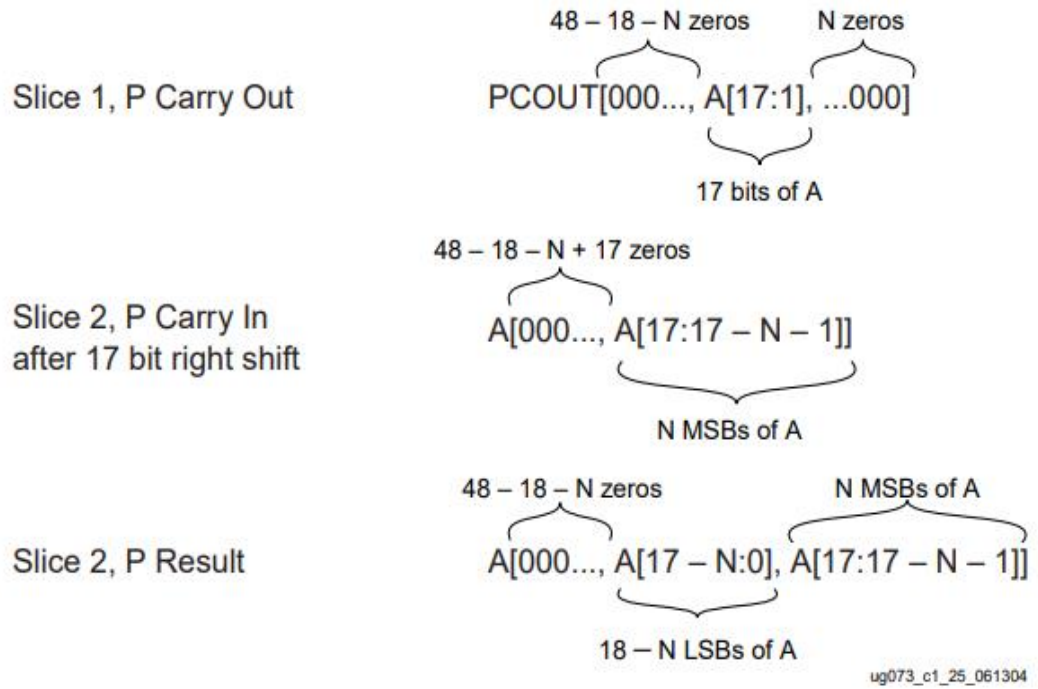


Figure 2-28: **Circular Barrel Shifter Equations**

Figure 2-29 shows the DSP48 used an 18-bit circular barrel shifter. The P register for slice 1 contains leading zeros in the MSBs, followed by the most-significant 17 bits of A, followed by n trailing zeros. If n equals zero, then are no trailing zeros and the P register contains leading zeros followed by 17 bits of A.

Barrel Shifter Using DSP Slice

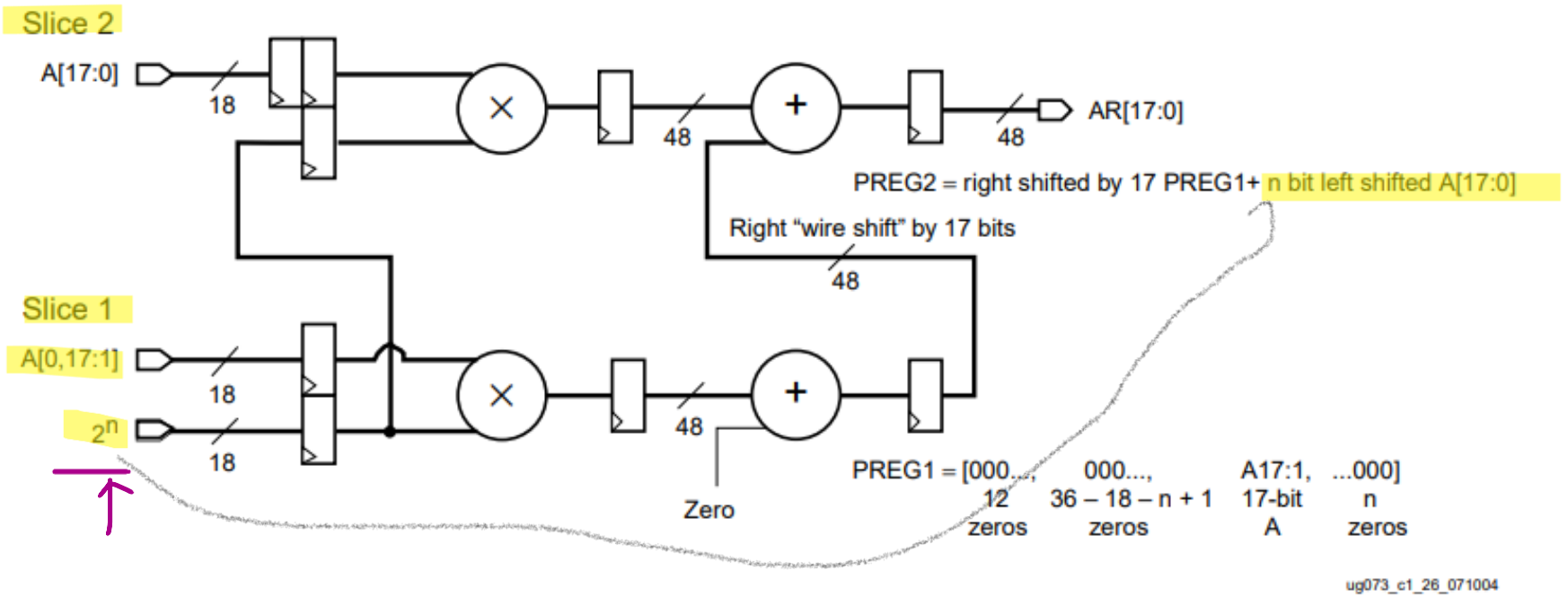


Figure 2-29: Dynamic 18-Bit Barrel Shifter

Table 2-14: **Miscellaneous DSP48 Implementations**

Single Slice Mode	Cycle	Inputs			Function and OPMODE[6:0]		Output
		A	B	C			
18-bit Barrel Shifter	0	A[17:0]	B[17:0]	X	Multiply	0x05	
	1	A[17:0]	B[17:0]	X	Multiply Accumulate	0x25	P
	2	A[17:0]	B[17:0]	X	Multiply	0x05	
	3	A[17:0]	B[17:0]	X	Multiply Accumulate	0x25	P

Barrel Shifter Operation

Barrel Shifter

An 18-bit barrel shifter can be implemented using the two DSP48 tiles in the DSP slice. To barrel shift the 18-bit number A[17:0] two positions to the left, the output from the barrel shifter is A[15:0], A[17], and A[16]. This operation is implemented as follows.

The first DSP48 is used to multiply {0,A[17:1]} by 2^2. The output of this DSP48 tile is now {0,A[17:1],0,0}. The output from the first tile is fed into the second DSP48 tile over the PCIN/PCOUT signals, and is passed through the 17-bit right-shifted input. The input to the Z multiplexer becomes {0,A[17],A[16]}, or {0,A[17:0],0,0} shifted right by 17 bits.

The multiplier inputs to the second DSP48 tile are A = A[17:0] and B = 2^2. The output of this multiplier is {A[17:0], 0,0}. This output is added to the 17-bit right-shifted value of {0,A[17],A[16]} coming from the previous slice. The 18-bit output of the adder is {A[15:0],A[17],A[16]}. This is the initial A input shifted by two to the left.

The Verilog code is in the reference design file `barrelshifter_18bit.v`, and the VHDL code is in the reference design file `barrelshifter_18bit.vhd`).

Square Root Architecture

Square Root

The square root of an integer number can be calculated by successive multiplication and subtraction. This is similar to the subtraction method used to divide two numbers. The square root of an N-bit number will have N/2 (rounded up) bits. If the square root is a fractional number, N/2 clocks are needed for the integer part of the answer, and every following clock gives one bit of the fraction part. The logic needed to compute this is shown in Figure 3-2.

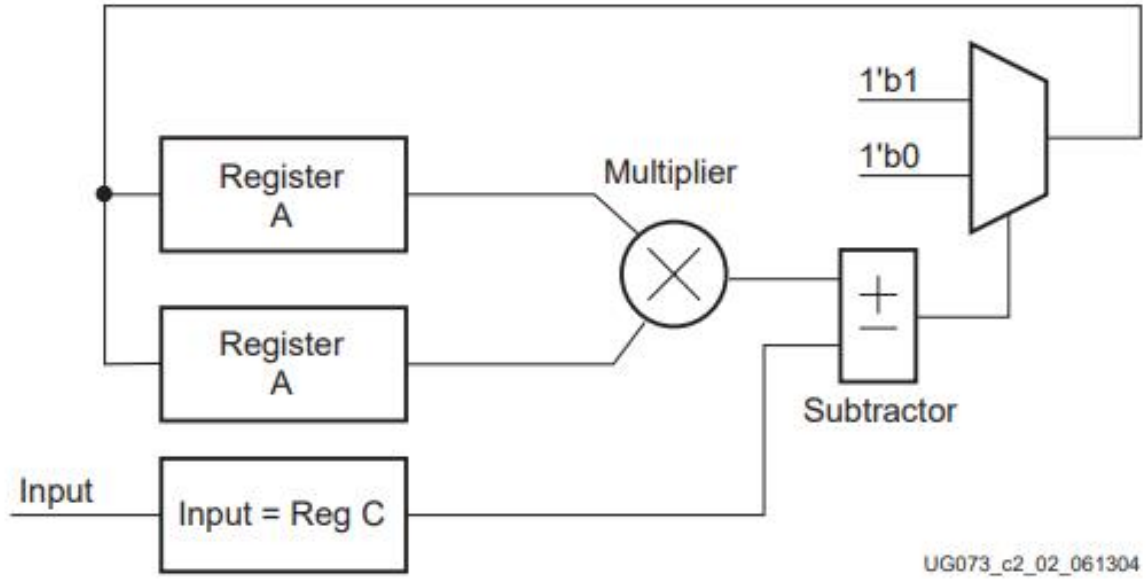


Figure 3-2: **Square Root Logic**

Square Root Method using DSP Slice

The square root for an 8-bit number can be calculated as follows:

$$\sqrt{X} = Y.Z$$

Y is the integer part of the root, and Z is the fraction part. Register A refers to the registers found on the A input to the DSP48 slice, and Register C refers to the registers found on the C input to the DSP48 slice

1. Read the number into Register C. Set Register A to `8'b10000000`.
2. Calculate Register C – (Register A * Register A).
3. If step 2 is positive, set Register A[(8-clock)] = 1,
   Register A[(8-clock)-1] = 1

   If step 2 is negative, set Register A[(8-clock)] = 0,
   Register A[(8-clock)-1] = 1
4. Repeat steps 1 to 3.

Four clocks are required to calculate the integer part of the value (Y). The number of clocks required for the fraction part (Z) depends on the precision required. For an 8-bit input value, the value in Reg_A after eight clocks includes the integer part given by the four MSBs and the fractional part given by the four LSBs.

Square Root Example

The binary of value of 11 decimal is `1011`. Expressed as an 8-bit number, it becomes `0000,1011`. Store this value as `0000,1011,0000,0000`. The last eight bits are necessary because the result is an 8-bit number, and 8 bits * 8 bits gives a 16-bit multiplication result.

Clock	Step	Action
1	1	Register A = 1000,0000
1	2	0000,1011,0000,0000 – (1000,0000 * 1000,0000)
1	3	Step 2 is negative. Set Register A to 0100,0000
2	1	Register A = 0100,0000
2	2	0000,1011,0000,0000 – (0100,0000 * 0100,0000)
2	3	Step 2 is negative. Set Register A to 0010,0000
3	1	Register A = 0010,0000
3	2	0000,1011,0000,0000 – (0010,0000 * 0010,0000)
3	3	Step 2 is positive. Set Register A to 0011,0000
4	1	Register A = 0011,0000
4	2	0000,1011,0000,0000 – (0011,0000* 0011,0000)
4	3	Step 2 is positive. Set Register A to 0011,1000
5	1	Register A = 0011,1000
5	2	0000,1011,0000,0000 – (0011,1000* 0011,1000)
5	3	Step 2 is negative. Set Register A to 0011,0100
6	1	Register A = 0011,0100
6	2	0000,1011,0000,0000 – (0011,0100* 0011,0100)
6	3	Step 2 is positive. Set Register A to 0011,0110
7	1	Register A = 0011,0110
7	2	0000,1011,0000,0000 – (0011,0110* 0011,0110)
7	3	Step 2 is negative. Set Register A to 0011,0101
8	1	Register A = 0011,0101
8	2	0000,1011,0000,0000 – (0011,0101* 0011,0101)
8	3	Step 2 is positive.

The output is in Register A and is `0011,0101`. The final answer is `11.0101`.

The Verilog code for this implementation (8-bit input, 8 clocks) is in `SQRT.v`, and the VHDL code is in `SQRT.vhd`.

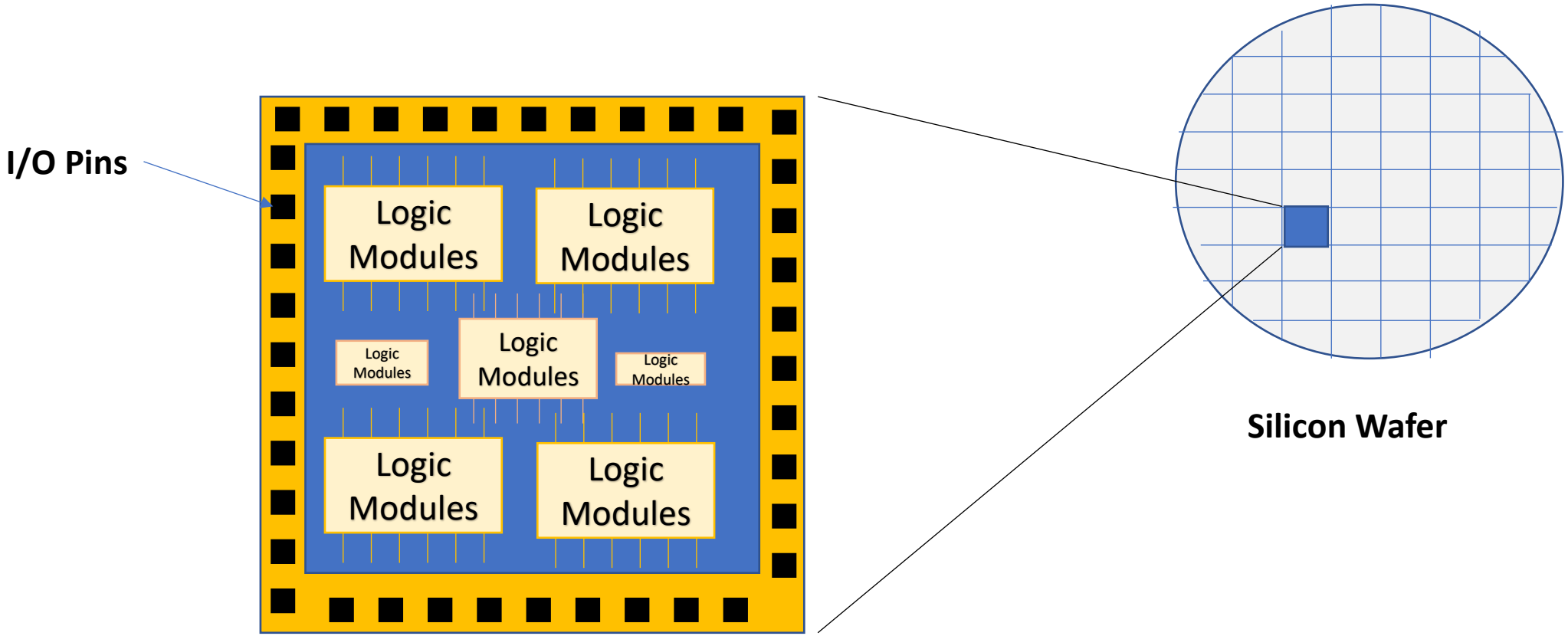
Faults, Testing and Testability

Topics: Faults and Testing, Examples of path sensitization method for fault tests, EX-OR method truth table, EX-OR method Boolean expression, testing of sequential elements using Scan cells

Types of Circuit Failure

- The domain of hardware related failure
- Permanent Failure: Incorrect behaviour at all times
- Intermittent Failure: Occurs randomly for finite time duration
- Transient Failure: Occurs in presence of certain environmental conditions such as high temperature, radiation, etc.
- Reasons for Failure: Wafer defects, impurities in clean room, mask mis-alignment, process imperfections, vibrations in equipment

Faults and Failure in Logic Circuit Chip



Number of internal inputs and outputs is much more than the number of physical I/O pins available

Production testing

- Detection of permanent errors caused by manufacturing defects. Involves two major steps:
  - Test Generation, and
  - Fault Simulation
- Failure modes are called ‘Faults’
- Set of vectors generated to detect ‘Faults’ is called ‘Fault-Simulation’
- ‘Fault-Models’ consider the logic effects that result from the physical faults in a circuit
- When a circuit fails to behave correctly, implies that the logic realized is different from logic that was specified for design

Chip Level Faults

Chip Level Fault Type	Degradation Fault	Open Circuit	Short Circuit
Leakage or Short between package leads	Yes		Yes
Broken or missing wire bonding		Yes	
Surface contamination or moisture	Yes		
Metal migration, stress peeling		Yes	Yes
Metallization		Yes	Yes

Gate Level Faults

Gate Level Fault Type	Degradation Fault	Open Circuit	Short Circuit
Contact Open		Yes	
Gate to Source short circuit	Yes		Yes
Field Oxide Parasitic Device	Yes		Yes
Gate Oxide Flaw, Spiking	Yes		Yes
Mask Misalignment	Yes		Yes

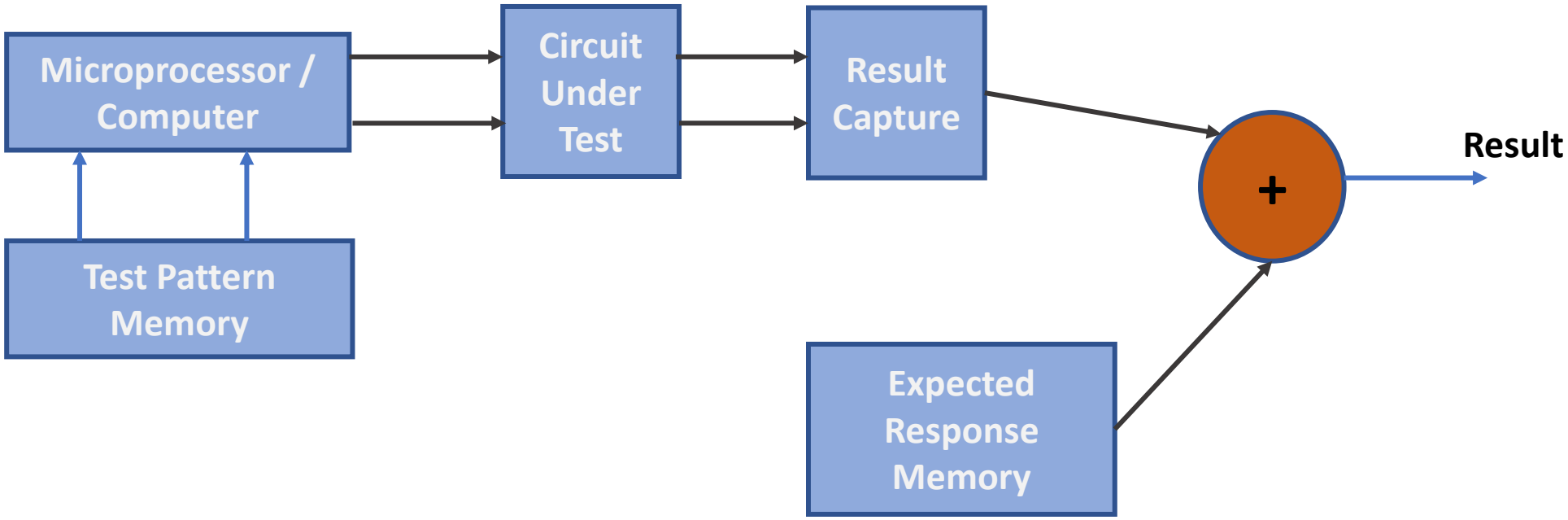
Fault Types

- Stuck Faults: A signal line is shorted to supply or ground permanently
- Bridging Faults: Short circuits in the interconnects between transistors in a logic cell are called bridging faults.
- Bridging faults are detected by measuring the quiescent current through the CMOS logic circuit. It takes more time to detect Bridging faults whereas **Stuck-At faults** are easier to locate.
- Problem: The fault sites are typically located in the middle of the logic circuit and their inputs or outputs are not directly accessible from i/o pins.
- There are maximum 100s of i/o pins vs the number of gates and their interconnects is in millions.

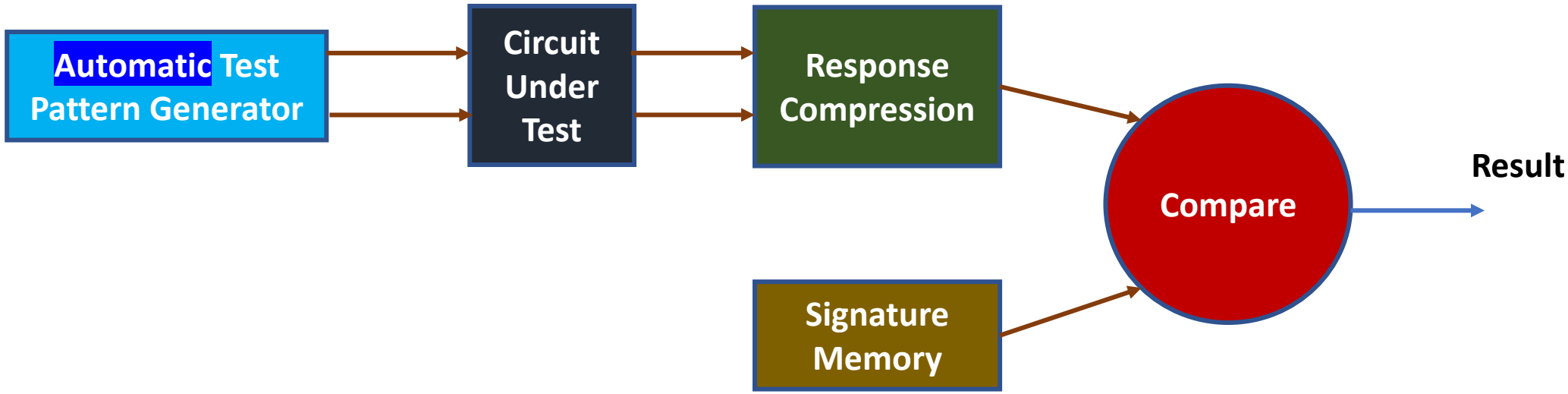
Un-Testable Faults

- Redundant Logic
- Un-Controllable Nets
- Un-Observable Nets that cannot be sensitized through I/O pins

Typical Test Setup



Automated Test Setup



Exhaustive Testing

Requires 2^N test vectors to exhaustively test all input combinations

Exhaustive Testing compares correct and faulty outputs for each input combination

This is a very slow approach

Single Stuck-At Fault Models

Stuck-At Fault Model: Assumes that there is just one stuck-at fault in circuit under test. Hope that single fault removal will remove multiple faults as well.

Stuck at 0 / Stuck at 1 (SA0/SA1) faults: Only two types of logical faults assumed in the model at gate level.

Observability: The degree to which one can observe a node at the output pins of an IC package.

Given that only a limited number of nodes could be directly observed, alternative methods such as JTAG are used to observe all outputs with some delays.

Controllability: Measure of the ease of setting the node to '1' or '0' state. Easiest would be directly settable by an input pin on IC package.

Fault Coverage

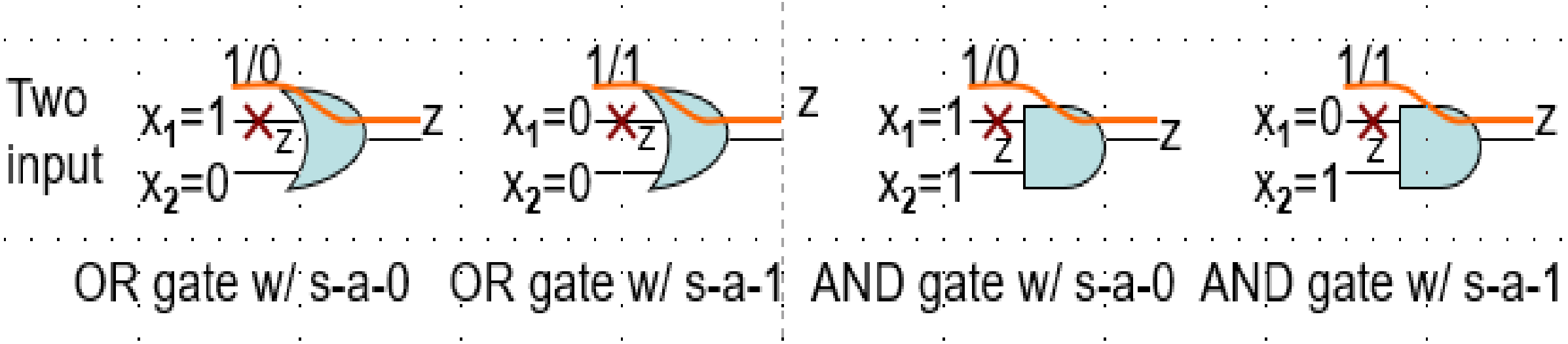
- % Fault Coverage = $$\frac{No.\ of\ nodes\ when\ set\ to\ 1\ or\ 0\ result\ in\ detection\ of\ fault}{total\ number\ of\ nodes\ in\ the\ circuit}$$

- KN Cycles are needed; K = no. of nodes in the circuit
- N/2 cycles are needed to detect each fault
- N = length of test sequence
- In turn, every node is tested for SA0 and SA1 sequentially i.e. Sequential Fault Grading

Fault Representation

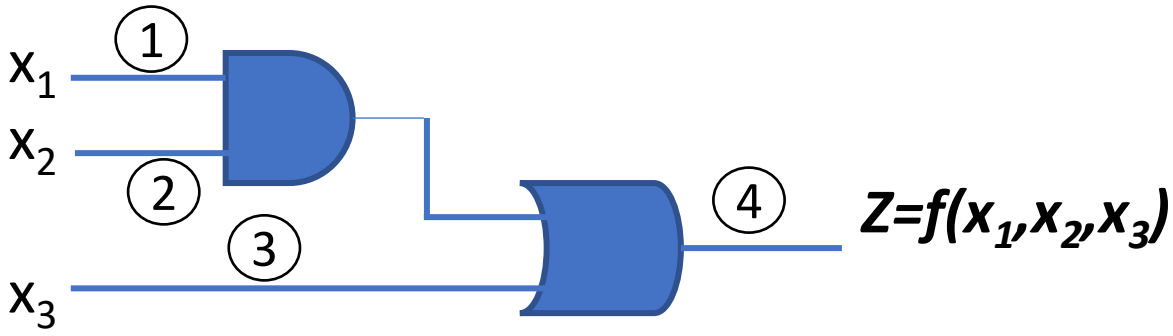
- $f(x_n) = f(x_1, x_2, x_3, ..., x_n)$ represents a fault-free circuit
- $f^{p/d}(x_n)$ represents the same circuit with fault p/d;
- Where p is a wire label, d is '0' or '1' representing SA0 or SA1 respectively
- n is the no. of input variables

Different Faults and Input vectors



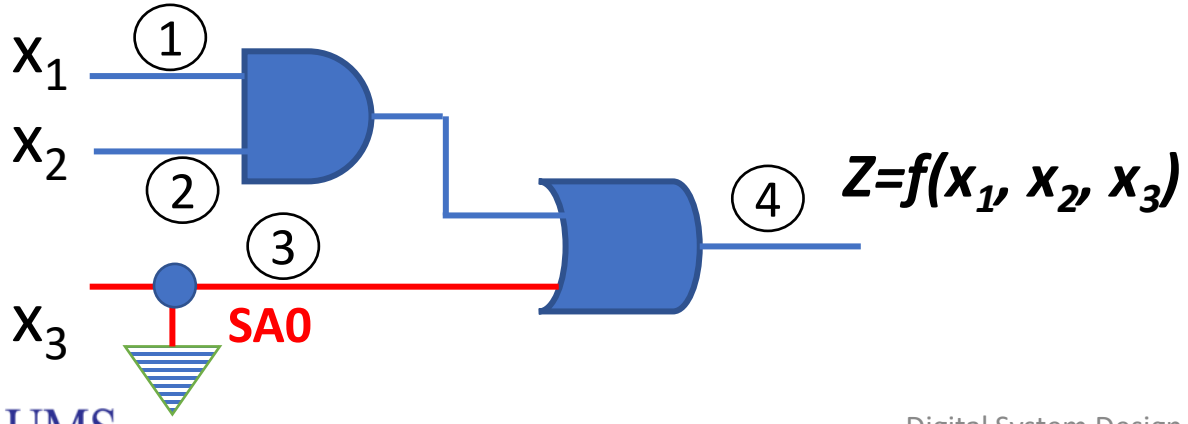
How to observe different faults at outputs of gates?

Example of fault representation



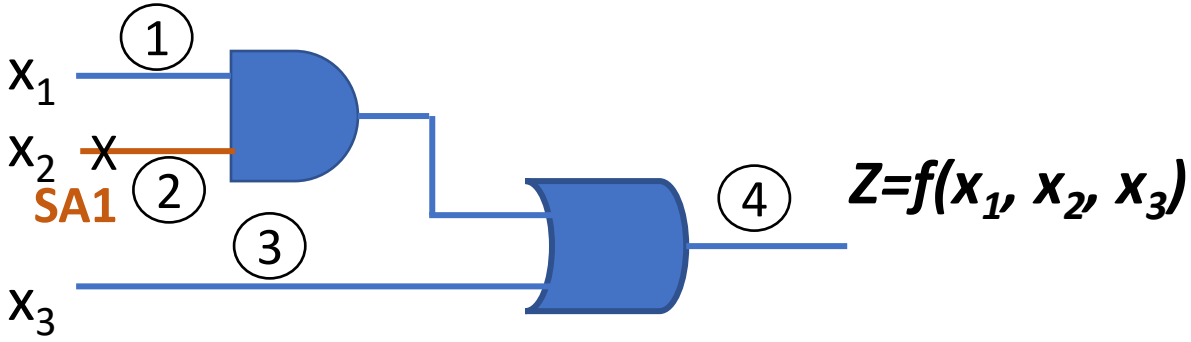
Stuck at 0 (SA0) fault at node 3:

$f^{3/0}(x_3) = x_1.x_2$



Stuck at 1 (SA1) fault at node 2:

$f^{2/1}(x_2) = x_1 + x_3$



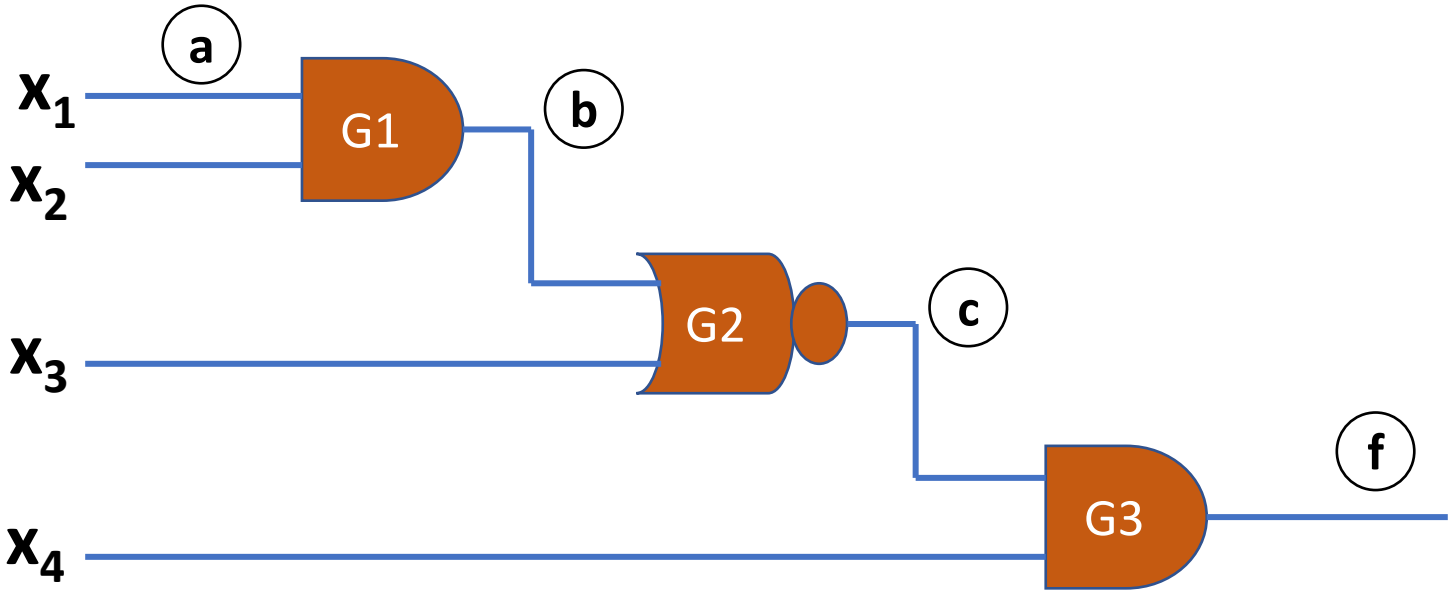
Path Sensitization

- Purpose is to sensitize the path so that inputs can help observe effects of **SA0** or **SA1** faults at the outputs
- In multi-level circuits, one set of test vector can act as test for faults in several paths

Three Steps in Path Sensitization Method

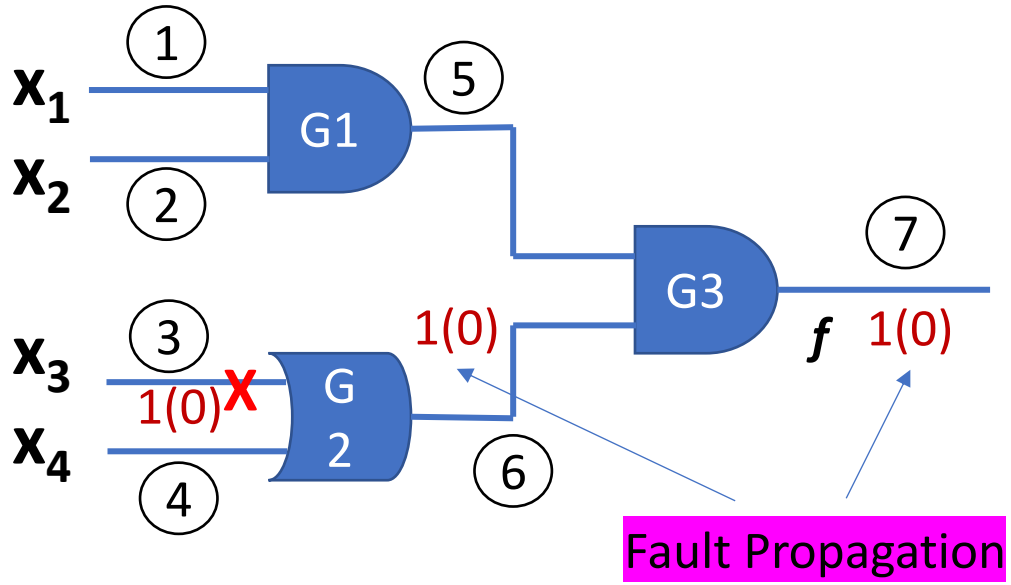
- **Fault Excitation:** Which vector to be induced to detect the suspected SA0 or SA1 fault at the suspicious path
- **Fault Propagation:** Identify path/s through which fault can be propagated to the observable output
- **Back tracking:** Move back from output towards all inputs and assign appropriate test values

Path Sensitization – how to



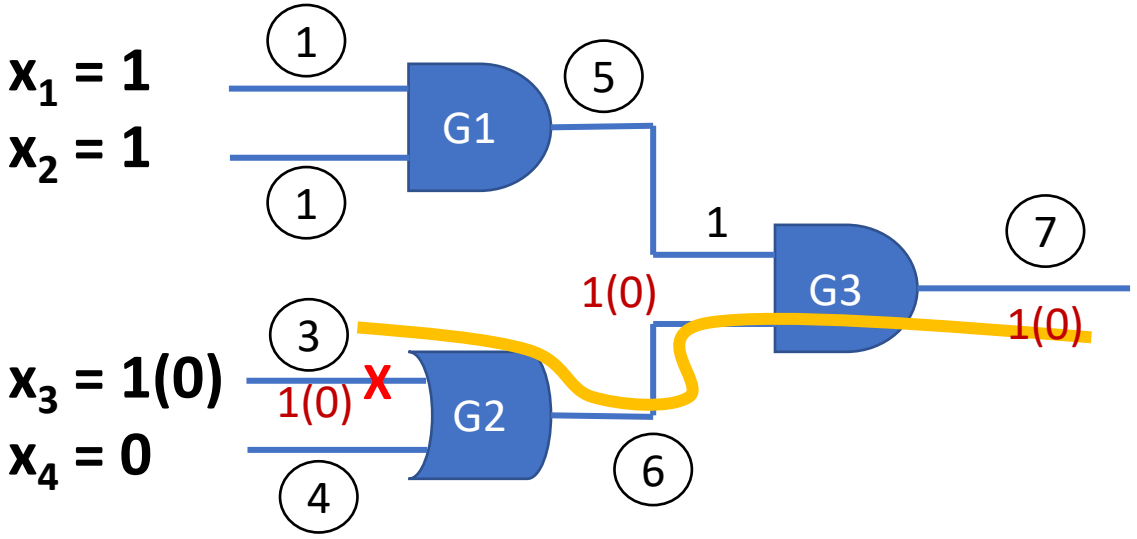
- To sensitize a path through an input of AND gate or NAND gate, all other inputs must be set to '1'
- To sensitize a path through an input of OR gate or NOR gate, all other inputs must be set to '0'

Path Sensitization – Example 1



Purpose: To detect **SA0** fault at wire 3 connected to input of OR gate
Input x_3 is selected opposite to Stuck-At fault (eg. SA0), written as **x_3=1(0)**
This is fault generation or excitation

Test Vectors for Example 1



Sensitized path = 3 → 6→ 7

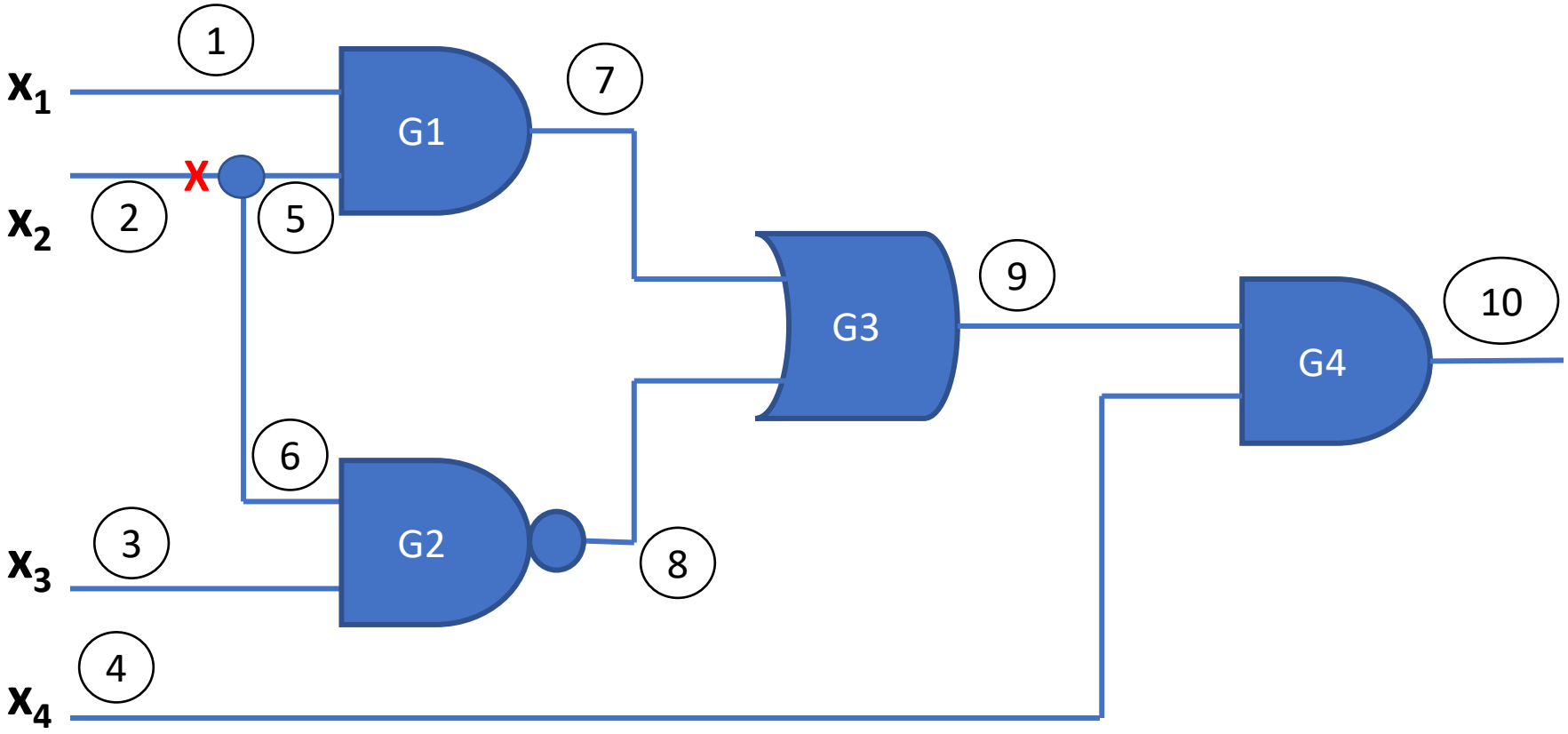
Back tracking reveals inputs to all gates to ensure fault propagation

Required test vector to detect SA0 at wire 3 is **"1110"**

Path Sensitization – Example 2

To detect SA1 fault at wire 2

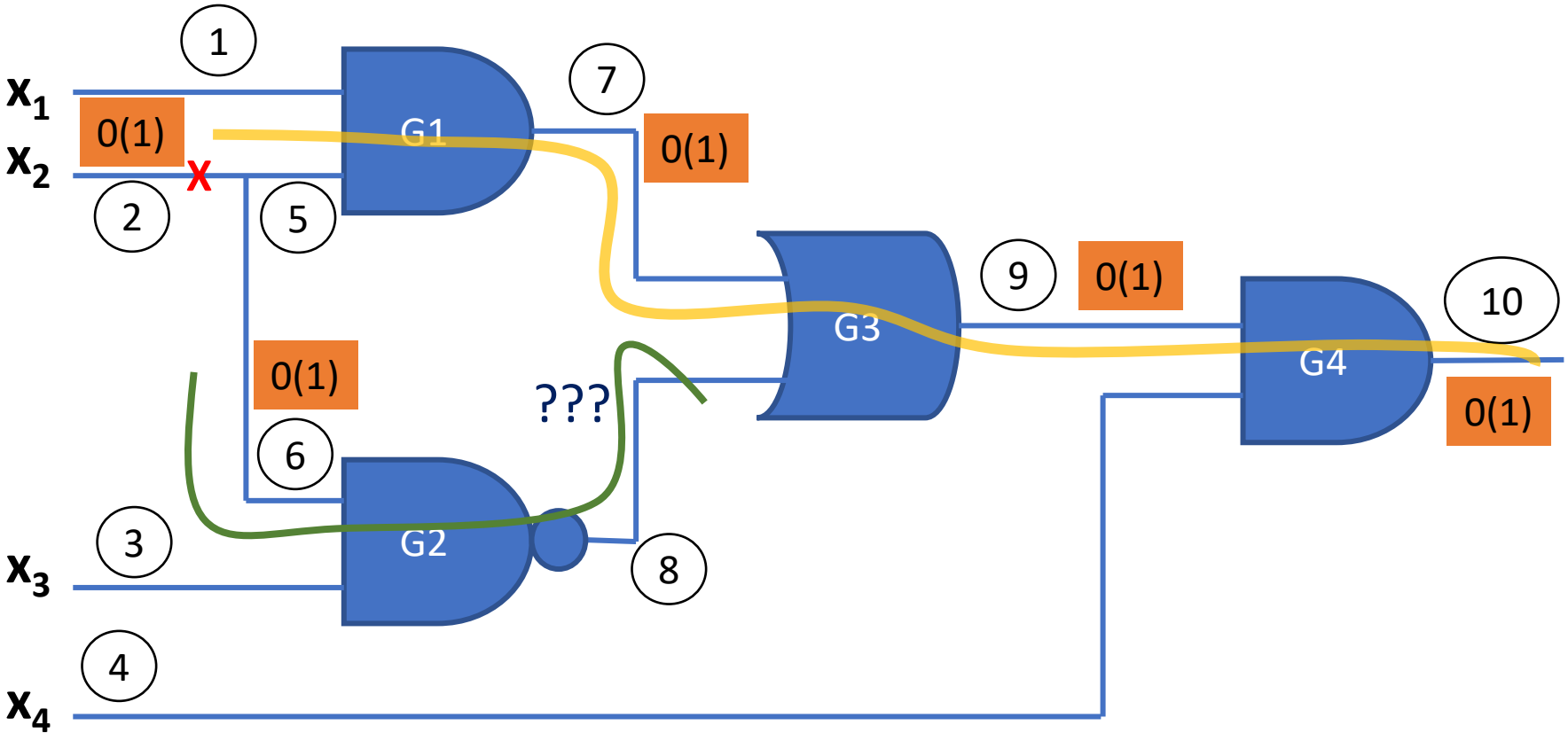
Two possible paths can be excited



… continued

Detect SA1 at path 2

Fault propagation by supplying input **x_2=0(1)**



Case 1:

Back tracking reveals:
First Selected path = 2→7 → 9 → 10

But path 8=1 due to path 2 input x_2
This is not correct to have '1' at Path 9. Hence '0' cannot be justified at line 8 and line 2 simultaneously

This situation is 'Inconsistent' hence Some other path is thus required

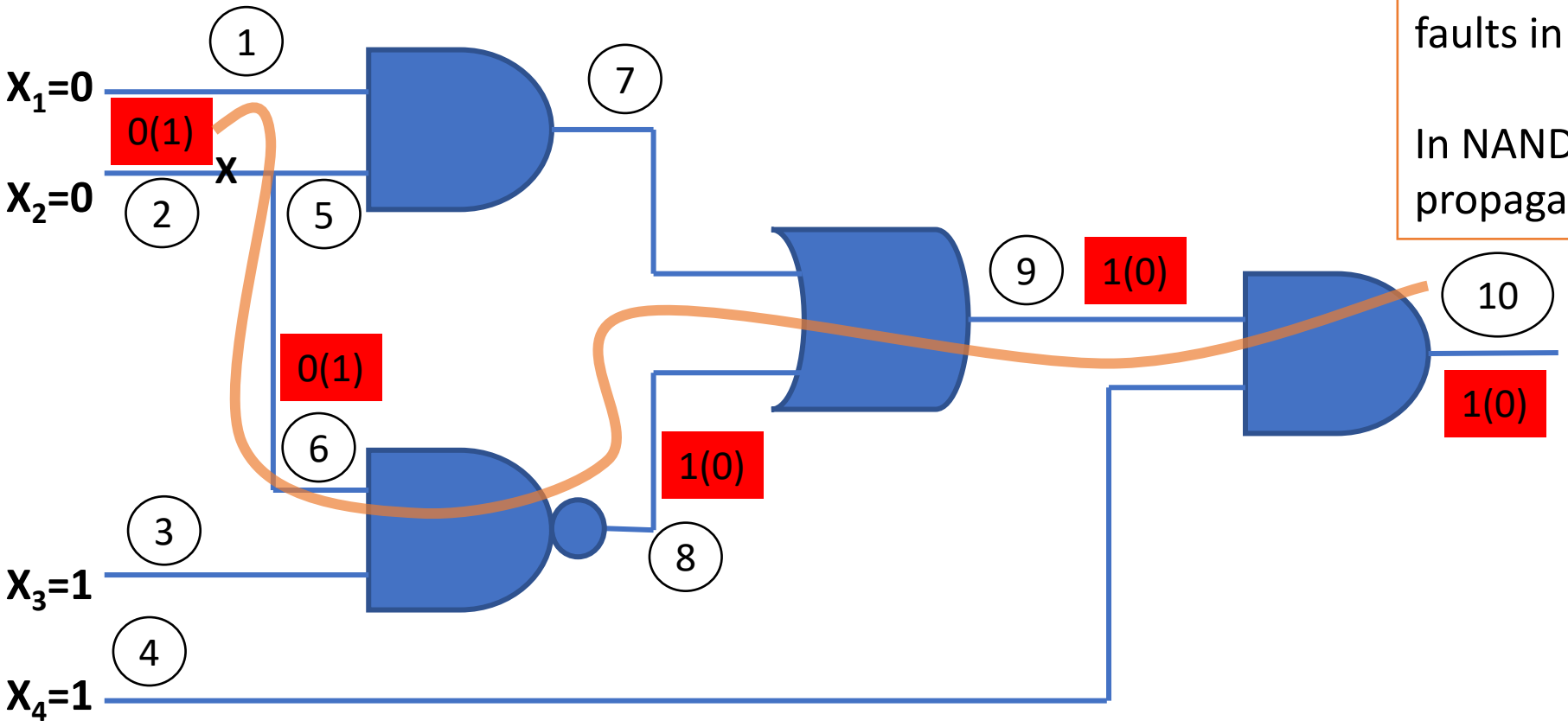
Continued – final test vectors

Selected path = 2 → 6 → 8 → 9 → 10

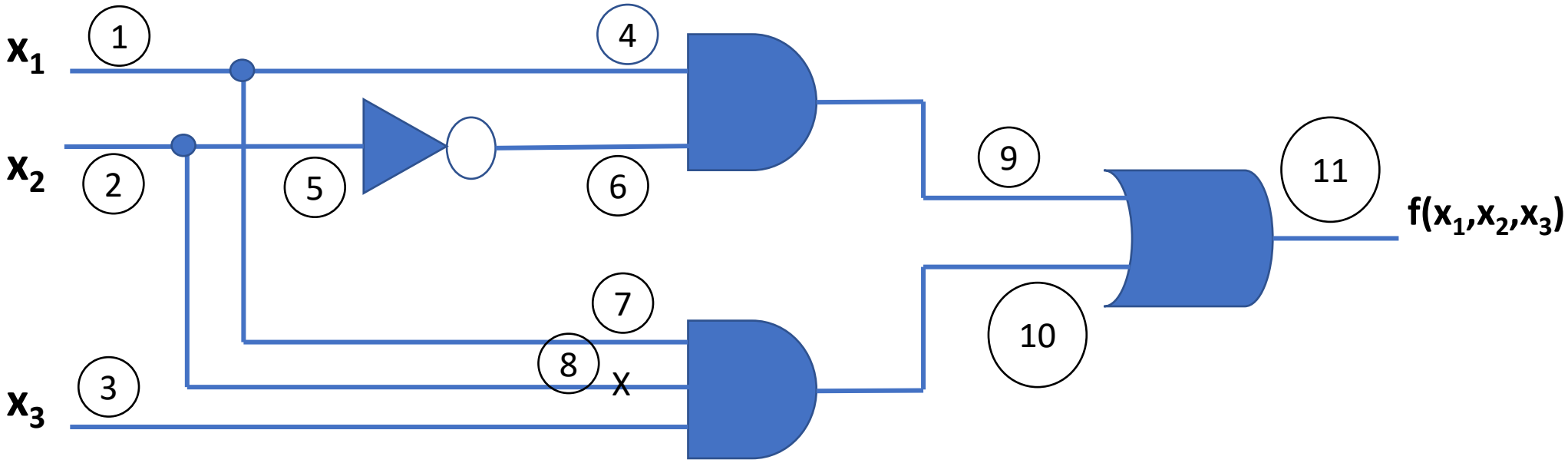
Test Vector = “0011”

This test vector can also reveal other faults in wires 6, 8 and 9

In NAND and NOR, reverse fault is propagated



Untestable Fault



Look at SA1 fault on path 8

This fault cannot be distinguished (sensitized) by changing inputs x1 to x3

Mathematically:

An untestable fault exists when $f^{8/1} \oplus f^{8} = 0$

This condition means it is not possible to test this path