

209K

- Polar Fire FPGA /
- Implement scan to chain / single DES on fpga
- how 2 design scan chain
- how 2 attack
- scan chain for DES \leftrightarrow practical attack
- complex adder

DR CHEF DID
NOT SEEM IMPRESSED
W MY PRELIMINARY
PRESENTATION :)

X8
COMPLEX
MULTIPLIER
IN SOFT

- steps
- ① FIGURE OUT HOW TO GET COMPLEX MULTIPLIER IMPLEMENTED ON FPGA LIBERO
 - ② FIGURE OUT SCAN CHAIN INSERTION + ACTIVATION → how BPF flows?
 - ③ FIGURE OUT HOW TO IMPLEMENT A STANDARD DES MUX



S.W:

where \rightarrow SPT b0
AMD Xilinx \rightarrow Vivado limited edition (+ series)

Vivado / VHDL \rightarrow schematic
 \hookrightarrow simple design

① Simple design + latch test

\downarrow

② Replicate on FPGA \rightarrow the final Xilinx FPGA

③ Develop attack

Cow 2

- \rightarrow summarize learning pts + presentation
 \rightarrow getting started \rightarrow play w/ simple gates (NOT, NAND, OR)
 \rightarrow define technical requirement
 ↳ generate bitstream
 ↳ read bitstream
 \rightarrow start emulating simpler scan chain

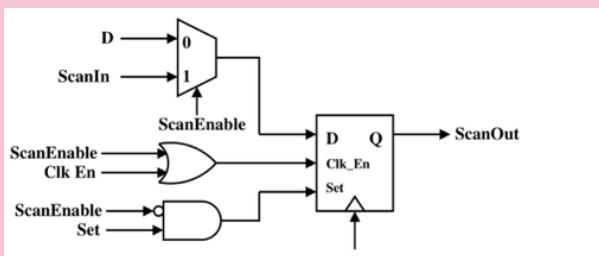
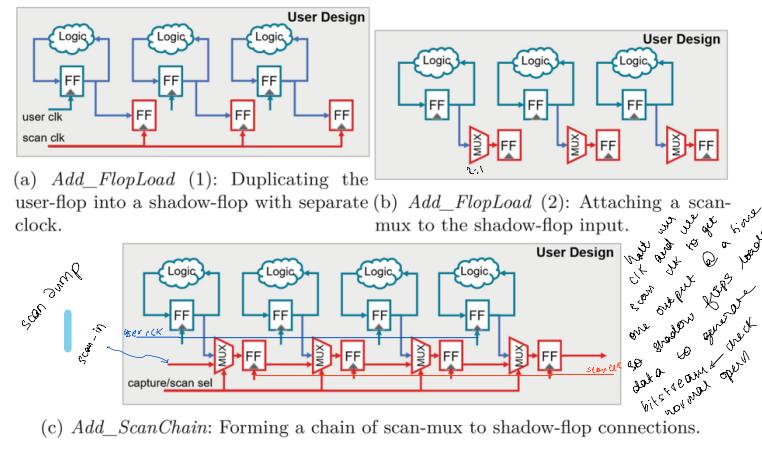


Fig. 1. Instrumenting a Flip-Flop for Scan

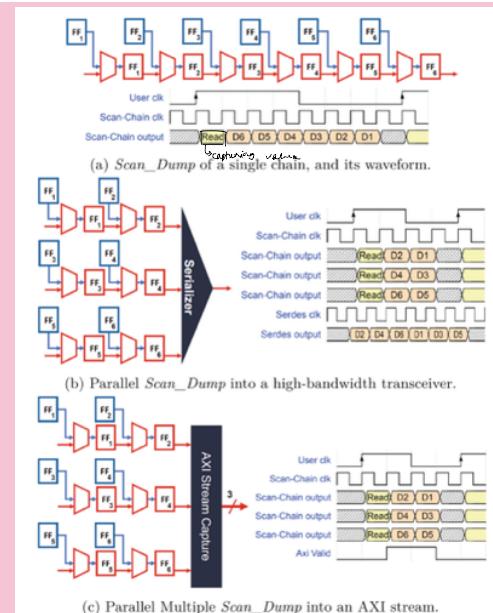
22/01/2024 AIMS

- Finish reading 4 research papers
- Setup for Xilinx Kinetix 7 series or capstone
- Start simulating simple verilog / VHDL circuits to learn these :)

SOFT SCAN CHAIN METHODOLOGY

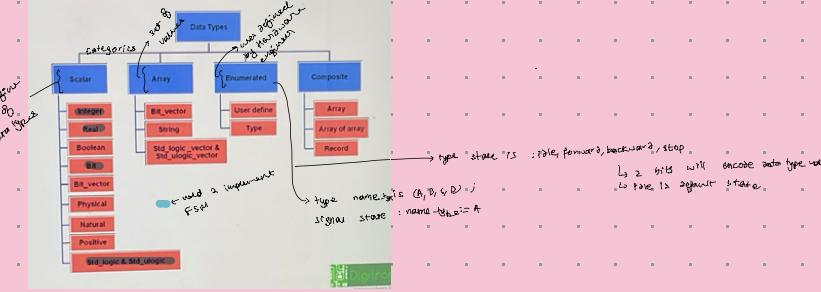


(c) Add_ScanChain: Forming a chain of scan-mux to shadow-flop connections.



VHDL Data types and operators
 integer, bit, std_logic, std_logic_vector
 IEEE 1076
 IEEE 1076 is being defined in more

use ieee... → defines data types



bit → 0 / 1 bit vector → group of multi-bit signals → bus

SIGNAL x: BIT;

x ← 1

SIGNAL y : BIT_VECTOR(3 DOWNTO 0); y ← 0111

SIGNAL w: BIT_VECTOR(0 TO 7); w ← 01010101

signal assignment operator

Boolean

BOOLEAN (TRUE, FALSE)

variable VAR1 : boolean := FALSE

integer (32 bit, -2147483648 to 2147483647)

Real [-1.0e-38 to 1.0e+38]

by default

VHDL Data Types

- The IEEE Standard 1164
- Introduce Multivalue Logic (std_logic_1164) Packages
- The primary data type std_ulogic (standard unresolved logic) consists of nine character literals in the following order:

 - U - uninitialized (default value)
 - X - strong drive, unknown logic value
 - Z - strong drive, logic zero
 - T - strong drive, logic one
 - - don't care
 - W - weak drive, unknown logic value
 - L - weak drive, logic zero
 - H - weak drive, logic one
 - - high impedance (for tri-state logic)

- std_ulogic and its subtype (std_logic, std_logic_vector, std_logic_vector) values can be categorized in terms of their state and strength (forcing, weak and high impedance.)
- Weak strength is used for multi-driver inputs catering for

VHDL Reserved words

abs	file	of	then
after	generic	open	transport
all	others	or	type
and	out	others	until
architecture	package	port	use
array	in	process	
begin	internal	procedure	
case	inout	rem	
component	library	report	
configuration	linkage	rol	
constant	loop	when	
downto	mod	ror	
else	nand	with	
elsif	nor		
end	nor		
entity	not		

Adding Operators

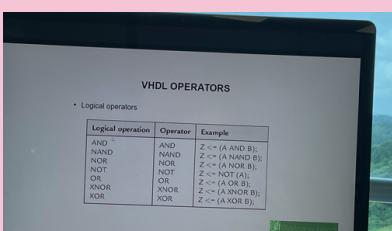
- + addition
- subtraction
- & concatenation
puts two bits or bit_vectors into a bit_vector

example:

signal A: bit_vector(5 downto 0);
 signal B,C: bit_vector(2 downto 0);
 B <= '0' & '1' & '0';
 C <= '1' & '1' & '0';
 A <= B & C; -- A now has "010110"

Type Conversion

- Data conversion defined in STD_LOGIC_ARITH
 - conv_integer(p): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an INTEGER value.
- Note that STD_LOGIC_VECTOR is not included.
 - conv_unsigned(p): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to an UNSIGNED value with size b bits.
 - conv_signed(p): Converts a parameter p of type INTEGER, UNSIGNED, SIGNED, or STD_ULOGIC to a SIGNED value with size b bits.



VHDL OPERATORS		
Logical operators		
Logical operation		
AND	AND	Z <= (A AND B);
NAND	NAND	Z <= (A NAND B);
NOR	NOR	Z <= (A NOR B);
NOT	NOT	Z <= NOT(A);
OR	OR	Z <= (A OR B);
XNOR	XNOR	Z <= (A XNOR B);
XOR	XOR	Z <= (A XOR B);

VHDL OPERATORS

Arithmetic operators

Arithmetic operation	Operator	Example
Addition	+	Z <= A + B;
Subtraction	-	Z <= A - B;
Multiplication	*	Z <= A * B;
Division	/	Z <= A / B;
Exponentiating	**	Z <= 4 ** 2;
Modulus	MOD	Z <= A MOD B;
Remainder	REM	Z <= A REM B;
Absolute value	ABS	Z <= ABS A;

TO DO:
 - summarize learning vs from 1st revision
 - infer initial requirements
 - which circuit?
 - fast paths
 - slow paths
 - carry merging techniques
 - start remaining circuits
 - add train chain