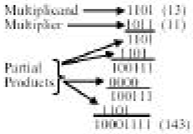
CPE/EE 422/522
Advanced Logic Design
L17

Electrical and Computer Engineering
University of Alabama in Huntsville

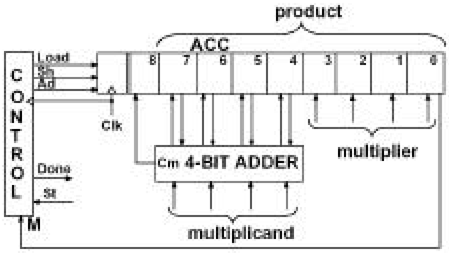
Networks for Arithmetic Operations

Case Study: Serial Parallel Multiplier



Note: we use unsigned binary numbers

Block Diagram of a Binary Multiplier

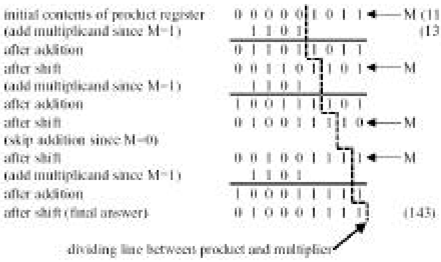


Ad – add signal // adder outputs are stored into the ACC

Sh – shift signal // shift all 9 bits to right

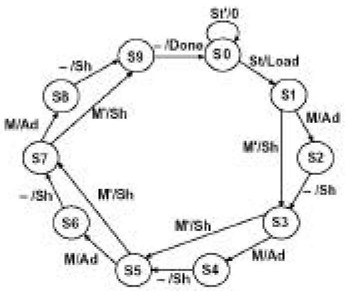
Ld – load signal // load multiplier into the 4 lower bits of the ACC and clear the upper 5 bits

Multiplication Example

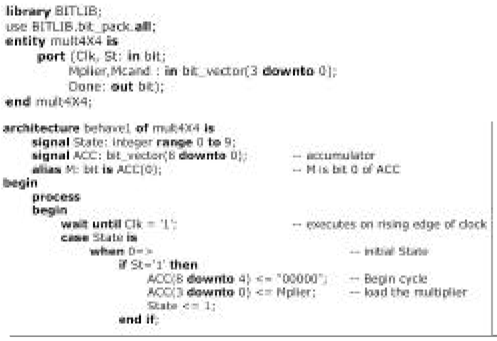


```
initial contents of product register      0 0 0 0 0 1 0 1 1   ← M (11)
(add multiplicand since M=1)                1 1 0 1                 (13)
after addition                            0 1 1 0 1 1 0 1 1
after shift                               0 0 1 1 0 1 1 0 1   ← M
(add multiplicand since M=1)                1 1 0 1
after addition                            1 0 0 1 1 1 1 0 1
after shift                               0 1 0 0 1 1 1 1 0   ← M
(skip addition since M=0)
after shift                               0 0 1 0 0 1 1 1 1   ← M
(add multiplicand since M=1)                1 1 0 1
after addition                            1 0 0 0 1 1 1 1 1
after shift (final answer)                0 1 0 0 0 1 1 1 1        (143)

dividing line between product and multiplier
```

State Graph for Binary Multiplier

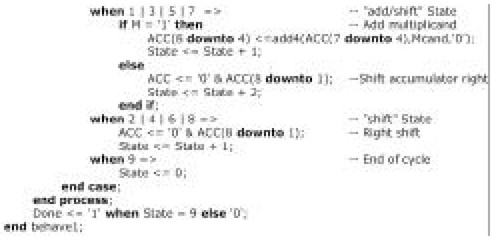


Behavioral VHDL Model



```
library BITLIB;
use BITLIB.bit_pack.all;
entity mult4X4 is
    port (Clk, St: in bit;
          Mplier,Mcand : in bit_vector(3 downto 0);
          Done: out bit);
end mult4X4;

architecture behave1 of mult4X4 is
    signal State: integer range 0 to 9;
    signal ACC: bit_vector(8 downto 0);            -- accumulator
    alias M: bit is ACC(0);                        -- M is bit 0 of ACC
begin
    process
    begin
        wait until Clk = '1';                      -- executes on rising edge of clock
        case State is
            when 0=>                                       -- initial State
                if St='1' then
                    ACC(8 downto 4) <= "00000";            -- Begin cycle
                    ACC(3 downto 0) <= Mplier;             -- load the multiplier
                    State <= 1;
                end if;
```

Behavioral VHDL Model (cont'd)

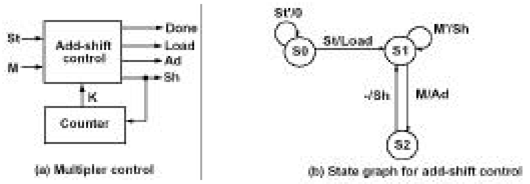


```
            when 1 | 3 | 5 | 7  =>                          -- "add/shift" State
                if M = '1' then                             -- Add multiplicand
                    ACC(8 downto 4) <=add4(ACC(7 downto 4),Mcand,'0');
                    State <= State + 1;
                else
                    ACC <= '0' & ACC(8 downto 1);           --Shift accumulator right
                    State <= State + 2;
                end if;
            when 2 | 4 | 6 | 8 =>                           -- "shift" State
                ACC <= '0' & ACC(8 downto 1);               -- Right shift
                State <= State + 1;
            when 9 =>                                       -- End of cycle
                State <= 0;
        end case;
    end process;
    Done <= '1' when State = 9 else '0';
end behave1;
```

Multiplier Control with Counter

- Current design: control part generates the control signals (shift/add) and counts the number of steps
- If the number of bits is large (e.g., 64), the control network can be divided into a counter and a shift/add control

Multiplier Control with Counter (cont'd)



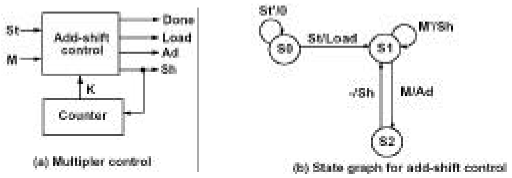
(a) Multipler control

(b) State graph for add-shift control

Add-shifts control: tests St and M and generates the proper sequence of add and shift signals

Counter control: counter generates a completion signal K that stops the multiplier after the proper number of shifts have been completed

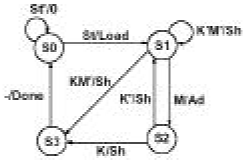
Multiplier Control with Counter (cont'd)



(a) Multipler control

(b) State graph for add-shift control

- Increment counter each time a shift signal is generated
- Generate K after n-1 shifts occured



Operation of a Multiplier Using Counter

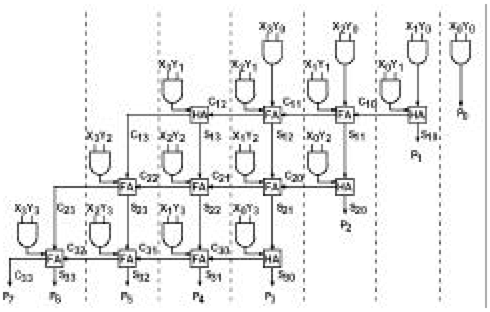
Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t0	S0	00	000000000	0	0	0	0	0	0	0
t1	S0	00	000001000	1	0	0	1	0	0	0
t2	S1	00	000001011	0	1	0	0	1	0	0
t3	S2	00	011011011	0	1	0	0	0	1	0
t4	S1	01	001101101	0	1	0	0	1	0	0
t5	S2	01	100111101	0	1	0	0	0	1	0
t6	S1	10	010011110	0	0	0	0	0	1	0
t7	S1	11	001001111	0	1	1	0	1	0	0
t8	S2	11	100011111	0	1	1	0	0	1	0
t9	S3	00	010001111	0	1	0	0	0	0	1

Array Multiplier

				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplier
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	partial product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		partial product 1
			C_{12}	C_{11}	C_{10}			1st row carries
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		1st row sums
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			partial product 2
		C_{22}	C_{21}	C_{20}				2nd row carries
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			2nd row sums
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				partial product 3
	C_{32}	C_{31}	C_{30}					3rd row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	final product

- What do we need to realize Array Multiplier?
- AND gates = ?
- FA = ?
- HA = ?

Array Multiplier (cont'd)



Array Multiplier (cont'd)

- Complexity of the N-bit array multiplier
  - – number of AND gates = ?
  - – number of HA = ?
  - – number of FA = ?
- Delay
  - – tg – longest AND gate delay
  - – tad – longest possible delay through an adder

Multiplication of Signed Binary Numbers

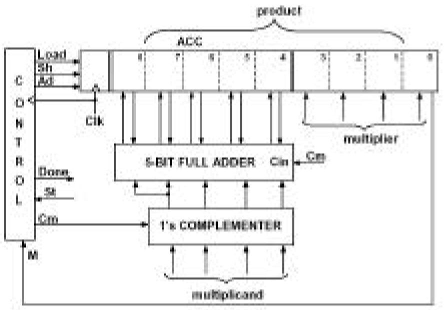
- How to multiply signed binary numbers?
- Procedure
  - – Complement the multiplier if negative
  - – Complement the multiplicand if negative
  - – Multiply two positive binary numbers
  - – Complement the product if it should be negative
- Simple but requires more hardware and time than other available methods

Multiplication of Signed Binary Numbers

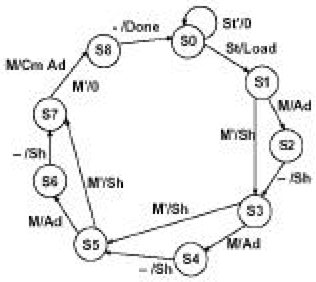
- Four cases
  - – Multiplicand is positive, multiplier is positive
  - – Multiplicand is negative, multiplier is positive
  - – Multiplicand is positive, multiplier is negative
  - – Multiplier is negative, multiplicand is negative
- Examples
  - – 0111 x 0101 = ?
  - – 1101 x 0101 = ?
  - – 0101 x 1101 = ?
  - – 1011 x 1101 = ?

- Preserve the sign of the partial product at each step
- If multiplier is negative, complement the multiplicand before adding it in at the last step

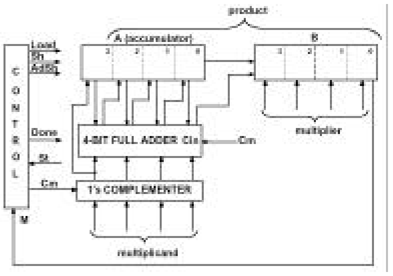
2’s Complement Multiplier



State Graph for 2’s Complement Multiplier

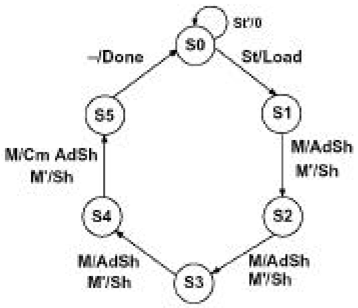


Faster Multiplier

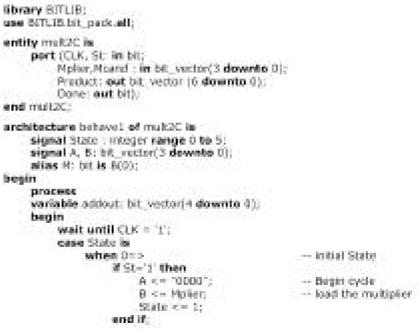


- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle

State Graph for Faster Multiplier

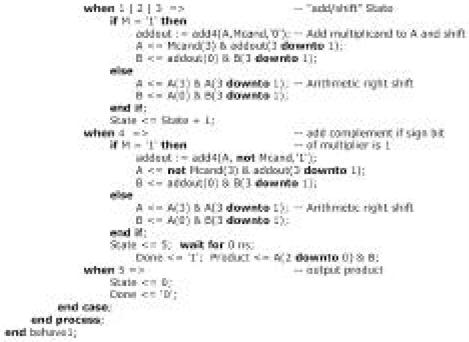


Behavioral Model for Faster Multiplier



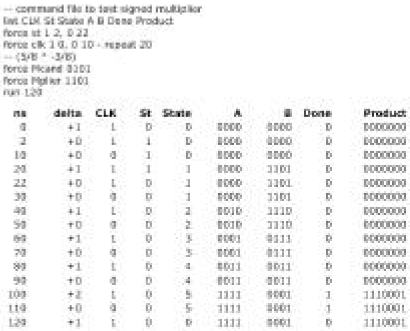
```
library BITLIB;
use BITLIB.bit_pack.all;

entity mult2C is
    port (CLK, St: in bit;
        Mplier,Mcand : in bit_vector(3 downto 0);
        Product: out bit_vector (6 downto 0);
        Done: out bit);
end mult2C;

architecture behave1 of mult2C is
    signal State : integer range 0 to 5;
    signal A, B: bit_vector(3 downto 0);
    alias M: bit is B(0);
begin
    process
    variable addout: bit_vector(4 downto 0);
    begin
        wait until CLK = '1';
        case State is
            when 0=>                            -- initial State
                if St='1' then
                    A <= "0000";                -- Begin cycle
                    B <= Mplier;                -- load the multiplier
                    State <= 1;
                end if;
```

Behavioral Model for Faster Multiplier



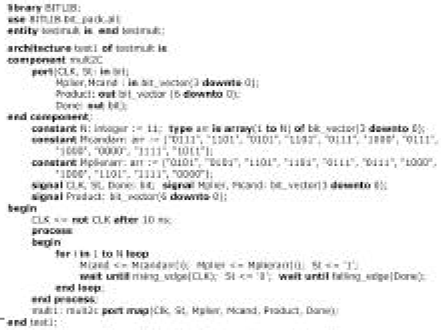
```
            when 1 | 2 | 3 =>                     -- "add/shift" State
                if M = '1' then
                    addout := add4(A,Mcand,'0'); -- Add multiplicand to A and shift
                    A <= Mcand(3) & addout(3 downto 1);
                    B <= addout(0) & B(3 downto 1);
                else
                    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
                    B <= A(0) & B(3 downto 1);
                end if;
                State <= State + 1;
            when 4 =>                             -- add complement if sign bit
                if M = '1' then                   -- of multiplier is 1
                    addout := add4(A, not Mcand,'1');
                    A <= not Mcand(3) & addout(3 downto 1);
                    B <= addout(0) & B(3 downto 1);
                else
                    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
                    B <= A(0) & B(3 downto 1);
                end if;
                State <= 5; wait for 0 ns;
                    Done <= '1'; Product <= A(2 downto 0) & B;
            when 5 =>                             -- output product
                State <= 0;
                Done <= '0';
        end case;
    end process;
end behave1;
```

Command File and Simulation



```
-- command file to test signed multiplier
list CLK St State A B Done Product
force st 1 2, 0 22
force clk 1 0, 0 10 - repeat 20
-- (5/8 * -3/8)
force Mcand 0101
force Mplier 1101
run 120

 ns   delta  CLK   St  State     A      B  Done   Product
  0     +1    1    0     0    0000   0000    0   0000000
  2     +0    1    1     0    0000   0000    0   0000000
 10     +0    0    1     0    0000   0000    0   0000000
 20     +1    1    1     1    0000   1101    0   0000000
 22     +0    1    0     1    0000   1101    0   0000000
 30     +0    0    0     1    0000   1101    0   0000000
 40     +1    1    0     2    0010   1110    0   0000000
 50     +0    0    0     2    0010   1110    0   0000000
 60     +1    1    0     3    0001   0111    0   0000000
 70     +0    0    0     3    0001   0111    0   0000000
 80     +1    1    0     4    0011   0011    0   0000000
 90     +0    0    0     4    0011   0011    0   0000000
100     +2    1    0     5    1111   0001    1   1110001
110     +0    0    0     5    1111   0001    1   1110001
120     +1    1    0     0    1111   0001    0   1110001
```

Test Bench for Signed Multiplier



```
library BITLIB;
use BITLIB.bit_pack.all;
entity testmult is end testmult;

architecture test1 of testmult is
component mult2C
    port(CLK, St: in bit;
        Mplier,Mcand : in bit_vector(3 downto 0);
        Product: out bit_vector (6 downto 0);
        Done: out bit);
end component;
    constant N: integer := 11; type arr is array(1 to N) of bit_vector(3 downto 0);
    constant Mcandarr: arr := ("0111", "1101", "0101", "1101", "0111", "1000", "0111",
        "1000", "0000", "1111", "1011");
    constant Mplierarr: arr := ("0101", "0101", "1101", "1101", "0111", "0111", "1000",
        "1000", "1101", "1111", "0000");
    signal CLK, St, Done: bit; signal Mplier, Mcand: bit_vector(3 downto 0);
    signal Product: bit_vector(6 downto 0);
begin
    CLK <= not CLK after 10 ns;
    process
    begin
        for i in 1 to N loop
            Mcand <= Mcandarr(i); Mplier <= Mplierarr(i); St <= '1';
            wait until rising_edge(CLK); St <= '0'; wait until falling_edge(Done);
        end loop;
    end process;
    mult1: mult2c port map(Clk, St, Mplier, Mcand, Product, Done);
end test1;
```

Hardware Testing and Design for Testability

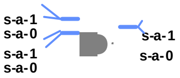
- Testing during design process
  - use VHDL test benches to verify that the overall design and algorithms used are correct
  - verify timing and logic after the synthesis
- Post-fabrication testing
  - when a digital system is manufactured, test to verify that it is free from manufacturing defects
  - today, cost of testing is major component of the manufacturing cost
  - efficient techniques are needed to test and design digital systems so that they are easy to test

Testing Combinational Logic

- Common types of errors
  - short circuit
  - open circuit
- If the input to a gate is shorted to ground, the input acts as if it is stuck at logic 0
  - s-a-0 (stuck-at-0) faults
- If the input to a gate is shorted to positive supply voltage, the input acts as if it is stuck at logic 1
  - s-a-1 (stuck-at-1) faults

Stuck-at Faults

- How many single stuck-at faults —
  - 2 (n + 1) — where n is the number of inputs

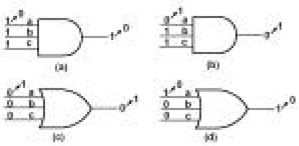


- We will assume
  - that there is only one stuck-at-fault active at a time in the whole circuit
  - "SSF" — single stuck-at fault

Stuck-at Faults for AND and OR gates

Test *a* for s-a-0

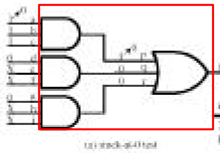
Test *a* for s-a-1



Test *a* for s-a-1

Test *a* for s-a-0

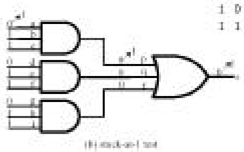
Testing an AND-OR Network



(a) stuck-at-0 test

BRUTE-FORCE testing:
apply 2^9=512 different input combinations and check the output

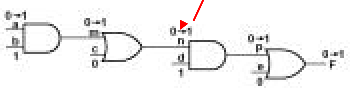
a	b	c	d	e	f	g	h	i	Faults Tested
1	1	1	0	X	X	0	X	X	a0, b0, c0, p0
0	X	X	1	1	1	0	X	X	d0, e0, f0, q0
0	X	X	0	X	X	1	1	1	g0, h0, i0, r0
0	1	1	0	1	1	0	1	1	a1, d1, g1, p1, q1, r1
1	0	1	1	0	1	1	0	1	b1, e1, h1, p1, q1, r1
1	1	0	1	1	0	1	1	0	c1, f1, i1, p1, q1, r1



(b) stuck-at-1 test

Path Detection & Sensitization: Small Example

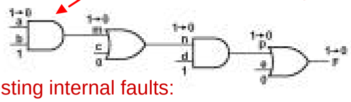
Test *n* to s-a-1



n=0 =>
m=0, c = 0 =>
a=0, b=1, c=0

d=1, e=0

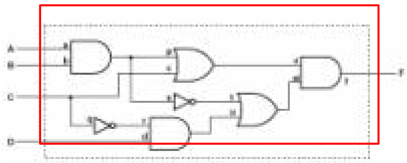
Change *a* to 1 =>
We can test *a, m, n,* or *p* to s-a-0



Testing internal faults:
choose a set of inputs that will excite the fault and then propagate the fault to the network output

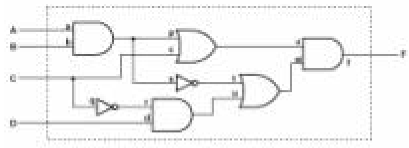
An Example

- What is a minimum set of test vectors to test the network below for all stuck-at-1 and stuck-at-0 faults?



- E.g., start with A-a-p-v-f-F path, determine the test vector to test s-a-0
- determine the list of faults covered
- select an untested fault, determine the required ABCD inputs
- determine the additional faults tested
- repeat the process until all faults are covered

An Example (cont'd)



	0	1
a	+	&
b	+	*
c	*	&
d	+	%
p	+	*
q	#	+
r	+	#
s	#	*
t	*	#
u	+	#
v	+	&
w	+	#
f	+	#

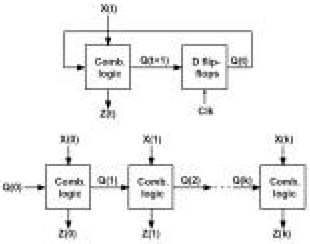
- Step 1: A-a-p-v-f-F, s-a-0
  - ABCD: 1101 (+)
- Step 2: s-a-0 for *c*
  - C=1, p=0, w=1 => ABCD=1011 (*)
- Step 3: s-a-0 for *q*
  - C=1, D=1, t=0, s=1 => ABCD=1111 (#)
- Step 4: s-a-1 for *a*
  - A=0, B=1, C=0, D=1 => ABCD=0101 (&)
- Step 5: s-a-1 for d (%)
  - D=0, C =0, t=1 => ABCD = 1100

Testing Sequential Logic

- In general, much more difficult than testing combinational logic since we must use sequences of inputs
  - typically we can observe inputs and outputs, not the state of flip-flops
  - assume the reset input, so we can reset the network to the initial state
- Test procedure
  - reset the network to the initial state
  - apply a test sequence and observe the output sequence
  - if the output is correct, repeat the test for another sequence
- How many test sequences do we have?
  - how do we test that the initial state of the network under test is equivalent to the initial state of the correct network?
  - what is the sequence length?

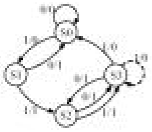
Testing Sequential Logic (cont’d)

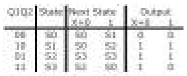
- In practice, if the network has N or fewer states, then apply only input sequences of length less than or equal 2N-1
- Example
  - consider a network which includes 5 inputs, 1 output, and 4 states
  - total number of test sequences: $(2^5)^7 = 2^{35}$ => infeasible (!)
  - derive a small set of test sequences that will adequately test a SN



Testing Sequential Logic (cont’d)

- Consider input sequence
  - X = 0 1 0 1 1 0 0 1 1
  - Output sequence Z = 0 0 1 0 1 1 1 1 0
  - If we change the network S3->S0 => S3->S3, the output sequence will be the same
- Find distinguishing sequence
  - an input sequence that will distinguish each state from the other states



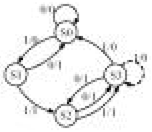


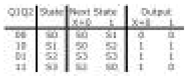
Q1Q2	State	Next State X=0	1	Output X=0	1
00	S0	S0	S1	0	0
10	S1	S0	S2	1	1
01	S2	S3	S3	1	1
11	S3	S0	S0	1	0

Input sequence: X=11

- S0: Z = 01
- S1: Z = 11
- S2: Z = 10
- S3: Z = 00

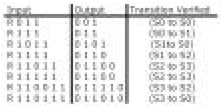
Testing Sequential Logic (cont’d)





Q1Q2	State	Next State X=0	1	Output X=0	1
00	S0	S0	S1	0	0
10	S1	S0	S2	1	1
01	S2	S3	S3	1	1
11	S3	S0	S0	1	0

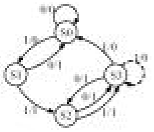
Verify each entry in the table using the following sequences:

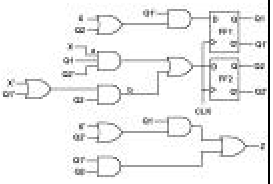


Input	Output	Transition Verified
R 0 1 1	0 0 1	(S0 to S0)
R 1 1 1	0 1 1	(S0 to S1)
R 1 0 1 1	0 1 0 1	(S1 to S0)
R 1 1 1 1	0 1 1 0	(S1 to S2)
R 1 1 0 1 1	0 1 1 0 0	(S2 to S3)
R 1 1 1 1 1	0 1 1 0 0	(S2 to S3)
R 1 1 0 0 1 1	0 1 1 1 1 0	(S3 to S2)
R 1 1 0 1 1 1	0 1 1 0 1 0	(S3 to S0)

Testing Sequential Logic (cont'd)

- Implementation of the FSM
  - S0=00, S1=10, S2=01, S3=11
- Test *a* for s-a-1
  - to do this Q1Q2 must be 10 => go to the state S1 and then set X to 0 (R10)
  - in normal operation, the next state will be S0; if a is s-a-1 then next state is S2
  - distinguish the state (S0 or S2); apply sequence 11
  - Final sequence: R1011
    Normal output: 0101
    Faulty output: 0110



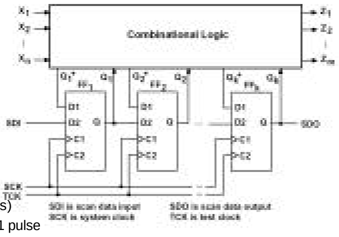


Scan Testing

- Testing of sequential networks is greatly simplified if we can observe the state of all the flip-flops instead of just observing the network outputs
  - Connect the output of each flip-flop to one of the IC pins?
  - Arrange flip-flops to form a shift register => shift out the state of flip-flops bit by bit using a single serial output pin => Scan path testing

Scan Path Testing

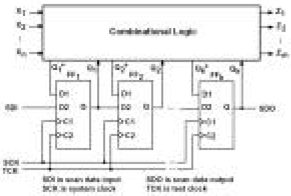
- Sequential network is separated into a combinational logic part and a state register composed of flip-flops



- Two ports FFs (2 D inputs and 2 clock inputs)
  - D1 is stored in the FF on C1 pulse
  - D2 is stored in the FF on C2 pulse
  - Q of each FF is connected to D2 of the next FF to form a shift register

Scan Path Testing

- Normal operation
  - system clock SCK = C1
  - inputs: $X_1 X_2 ... X_N$
  - outputs: $Z_1 Z_2 ... Z_N$
- Testing
  - FFs are set to a specified state using the SDI and TCK
  - test vector is applied $X_1 X_2 … X_N$
  - outputs $Z_1 Z_2 ... Z_N$ are verified
  - SCK is pulsed to take the network to the next state
  - next state is verified by pulsing the TCK to shift the state code out of the scan register via SDO

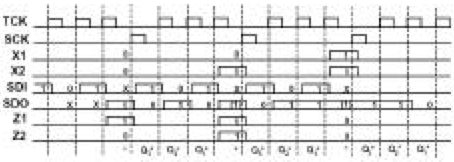


Scan Path Testing: An Example

- SQ: X_1X_2, $Q_1Q_2Q_3$, Z_1Z_2

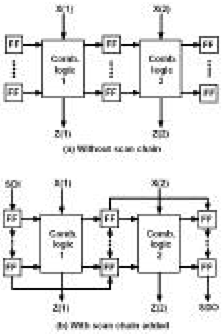
One row of the state transition table:

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$				Z_1Z_2			
	X_1X_2 = 00	01	11	10	00	01	11	10
101	010	110	011	111	10	11	00	01



* Read output (output at other times not shown)

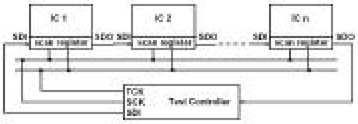
Scan Chain



(a) Without scan chain

(b) With scan chain added

Scan Test with Multiple ICs

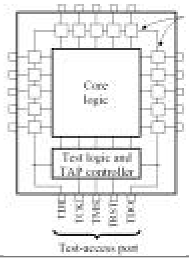


Boundary Scan

- PCB testing has become more difficult
  - ICs have become more complex, with more and more pins
  - PCBs have become more denser with multiple layers and fine traces
  - Bed-of-nails testing
    - use sharp probes to contact the traces on the board
    - test data are applied to and read from various ICs
    - => not practical for high-density PCBs with fine traces and complex ICs
- Boundary scan test methodology:
  introduced to facilitate the testing of complex PC boards
  - developed by JTAG (Joint Task Action Group)
  - adopted as ANSI/IEEE Standard 1149.1 – "Standard Test Access Port and Boundary Scan Architecture"
  - IC manufacturers make ICs that conform the standard
  - ICs can be linked together on a PCB, so that they can be tested using only a few pins on the PCB edge connector

Boundary Scan Register

- Boundary Scan Register (BSR) – cells of the BSR are placed between input or output pins and the internal core logic
- Four or five pins of the IC are devoted to the test-access-port (TAP)

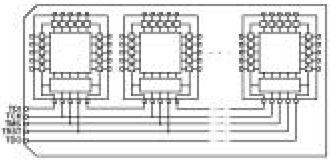


Boundary scan cells

TAP pins

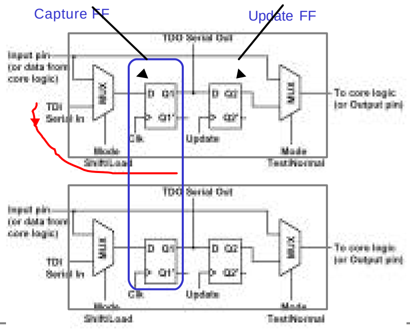
- TDI – Test data input (data are shifted serially into the BSR)
- TCK – Test clock
- TMS – Test mode select
- TDO – Test data output (serial output from BSR)
- TRST – Test reset (resets the TAP controller and test logic – optional pin)

PCB with Boundary Scan ICs

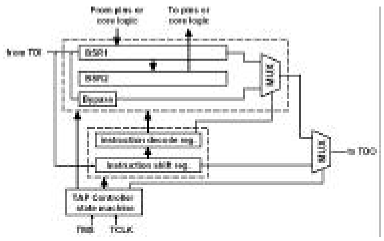


- BSRs in the ICs are linked together serially in a single chain with input TDI and output TDO.
- TCK, TMS, TRST are connected in parallel to all of the ICs.

Boundary Scan Cell



Basic Boundary Scan Architecture



- BSR1 – shift register, which consists of the Q1 flip-flops in the boundary scan cells
- BSR2 – represents the Q2 flip-flops; can be parallel loaded from BSR1 when an update signal is received
- TDI can be shifted into the BSR1, through a bypass register, or into the ISR

TAP Controller

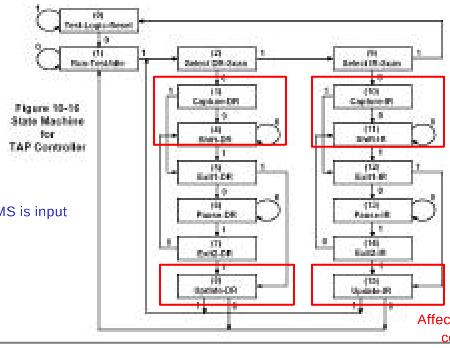


Figure 10-16 State Machine for TAP Controller

TMS is input

Affect ASIC core

TAP Controller: How it Works (I)

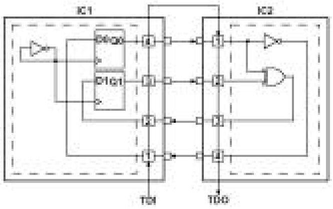
- TAP Controller
  - 16 state FSM
  - Change states depending on TMS and TCK
  - Output: signals to control the test data registers and instruction register (including serial shift clocks and update clocks)
- Test-logic-reset is the initial state;
  on a low TMS go to Run-Test/Idle state
- TMS: 1100 => Shift-IR
- In Shift-IR command is shifted in through TDI port
- …

Instructions in the IEEE Standard

- BYPASS: allows the TDI serial data to go trough 1-bit bypass register on the IC instead of through the BSR1. In this way one or more ICs on the PCB may be bypassed.
- SAMPE/RELOAD: used to scan the BSR without interfering with the normal operation of the core logic. Data is transferred to or from the core logic from or to the IC pins without interference. Samples of this data can be taken and scanned out through the BSR. Test data can be shifted into the BSR.
- EXTEST: allows board-level interconnect testing and testing of clusters of components which do not incorporate the boundary scan test features. Test data is shifted into the BSR and then it goes to the output pins. Data from the input pins is captured by the BSR.
- INTEST (optional): this instruction allows testing of the core logic by shifting test data into the boundary-scan register. Data shifted into the BSR takes the place of data from the input pins, and output data from the core logic is loaded into the BSR.
- RUNBIST (optional): this instruction causes special built-in self-test (BIST) logic within the IC to execute.

Interconnection Testing using Boundary Scan

Test PC board traces between IC1 and IC2



Assume:
IR on each IC is 3 bits long with
EXTEST coded as 000
SAMPLE/PRELOAD as 001

- Test the connections between two ICs.
- IC1: 2 input pins, 2 output pins.
- IC2: 2 input pins, 2 output pins.
- Test data is shifted into the BSRs via TDI.
- Data from the input pins is parallel-loaded into the BSRs and shifted out via TDO.

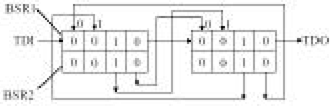
Steps Required to Test Connections

- **1.** Reset the TAP state machine to the Test-Logic-Reset state by inputting a sequence of five 1's on TMS. The TAP controller is designed so that a sequence of five 1's will always reset it regardless of the present state. Alternatively, TRST could be asserted if it is available.
- **2.** Scan in the SAMPLE/PRELOAD instruction to both ICs using the sequences for TMS and TDI given below.
  - State: 0 1 2 9 10 11 11 11 11 11 11 12 15 2
    TMS: 0 1 1 0 0 0 0 0 0 0 1 1 1
    TDI: – – – – – 1 0 0 1 0 0 – –
- The TMS sequence 01100 takes the TAP controller to the Shift-IR state. In this state, copies of the SAMPLE/PRELOAD instruction (code 001) are shifted into the instruction registers on both ICs. In the Update-IR state, the instructions are loaded into the instruction decode registers. Then the TAP controller goes back to the Select DR-scan state.

Steps Required to Test Connections (cont'd)

- **3.** Preload the first set of test data into the ICs using the sequences for TMS and TDI given below.
  State: 2 3 4 4 4 4 4 4 4 4 5 8 2
  TMS: 0 0 0 0 0 0 0 0 0 1 1 1
  TDI: – – 0 1 0 0 0 1 0 0 – –

  Data is shifted into BSR1 in the Shift-DR state, and it is transferred to BSR2 in the Update-DR state. The result is as follo



Steps Required to Test Connections (cont'd)

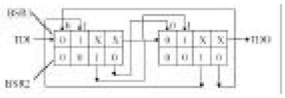
- 4. Scan in the EXTEST instruction to both ICs using the following sequences:
  State: 2 9 10 11 11 11 11 11 11 12 15 2
  TMS: 1 0 0 0 0 0 0 0 1 1 1
  TDI: – – – 0 0 0 0 0 0 – –

  The EXTEST instruction (000) is scanned into the instruction register in state Shift-IR and loaded into the instruction decode register in state Update-IR. At this point, the preloaded test data goes to the output pins, and it is transmitted to the adjacent IC input pins via the printed circuit board traces.

Steps Required to Test Connections (cont'd)

- 5. Capture the test results from the IC inputs. Scan this data out to TDO and scan the second set of test data in using the following sequences:
  State: 2 3 4 4 4 4 4 4 4 4 5 8 2
  TMS: 0 0 0 0 0 0 0 0 0 1 1 1
  TDI: – – 1 0 0 0 1 0 0 0 – –
  TDO: – – x x 1 0 x x 1 0 – –

  The data from the input pins is loaded into BSR1 in state Capture-DR. At this time, if no faults have been detected, the BSRs should be configured as shown below, where the X's indicate captured data which is not relevant to the test.



  The test results are then shifted out of BSR1 in state Shift-DR as the new test data is shifted in. The new data is loaded into BSR2 in the Update-IR state.

Steps Required to Test Connections (cont'd)

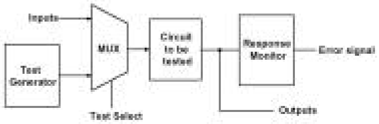
- 6. Capture the test results from the IC inputs. Scan this data out to TDO and scan all 0's in using the following sequences:
State: 2 3 4 4 4 4 4 4 4 4 4 5 8 2 9 0
TMS: 0 0 0 0 0 0 0 0 0 1 1 1 1 1
TDI: – – 0 0 0 0 0 0 0 0 – – – –
TDO: – – x x 0 1 x x 0 1 – – – –

  The data from the input pins is loaded into BSR1 in state Capture-DR. Then it is shifted out in state Shift-DR as all 0's are shifted in. The 0's are loaded into BSR2 in the Update-IR state. The controller then returns to the Test-Logic-Reset state and normal operation of the ICs can then occur. The interconnection test passes if the observed TDO sequences match the ones given above.

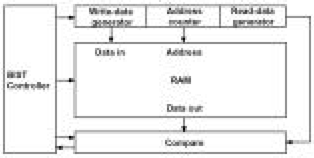
Built-In Self-Test

- Add logic to the IC so that it can test itself
  - Built-In Self-Test – BIST
- Using BIST
  - when test mode is selected by the test-select signal, an on-chip test generator applies test patterns to the circuit under test
  - the resulting outputs are observed by the response monitor, which produces an error signal if an incorrect output is detected

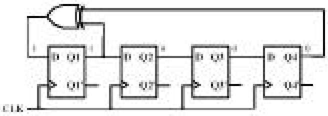
Generic BIST Scheme



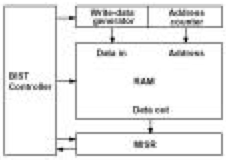
Self-Test Circuit for RAM



Linear Feedback Shift Registers (LFSR)



Self-Test Circuit for RAM with Signature Regs



MISR – Multiple Input Signature Register

E.g. for MISR –form a check-sum by adding up all data bytes stored in the RAM