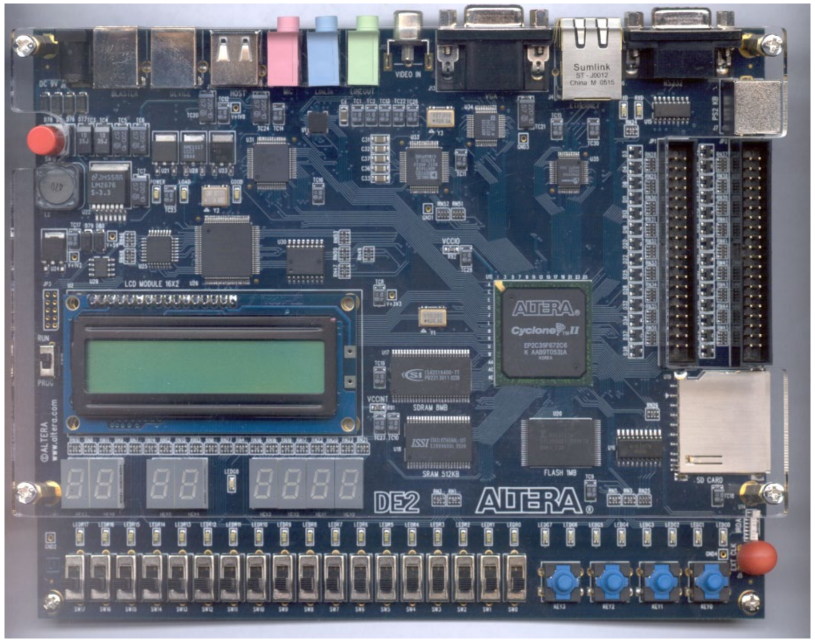
CPE 322 Fundamentals of Hardware Design

Electrical and Computer Engineering
University of Alabama in Huntsville

Hardware Testing and Design For Testability Issues

Hardware/Software Co-Design and General Trade-offs



Hardware Testing and Design for Testability

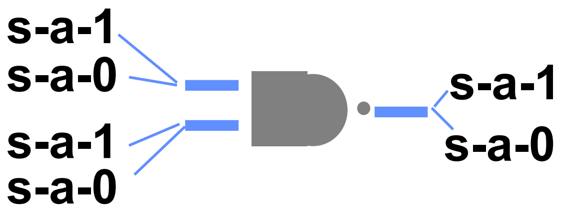
- Testing during design process
  - use Verilog HDL test benches to verify that the overall design and algorithms used are correct
  - verify timing and logic after the synthesis
- Post-fabrication testing
  - when a digital system is manufactured, test to verify that it is free from manufacturing defects
  - 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

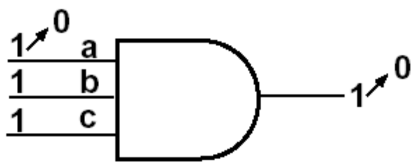
- How many single stuck-at faults —
  - 2 (n + 1) — where n is the number of inputs



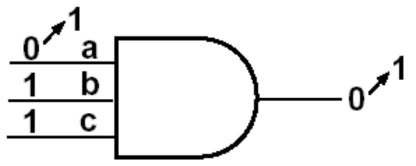
- We will assume
  - that there is only one stuck-at-fault active at a time in the whole circuit
  - "SSF" — single stuck-at fault

Stuck-at Faults for AND and OR gates

Test *a* for s-a-0



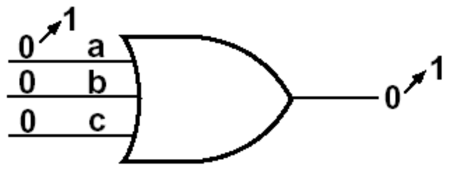
(a)



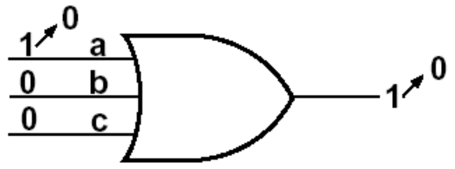
(b)

Test *a* for s-a-1

Test *a* for s-a-1



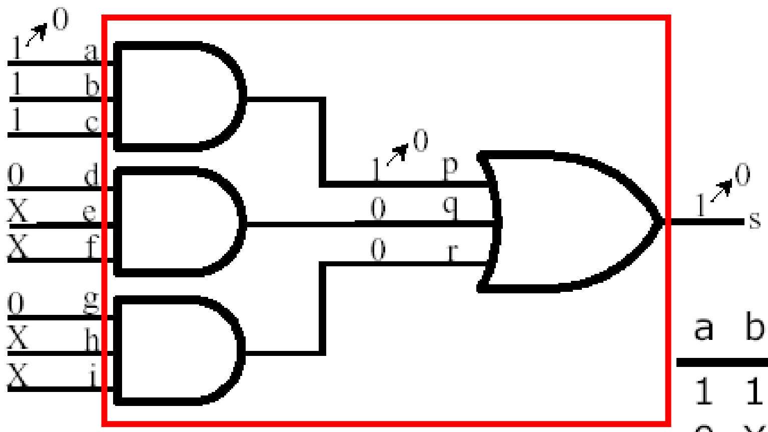
(c)



(d)

Test *a* for s-a-0

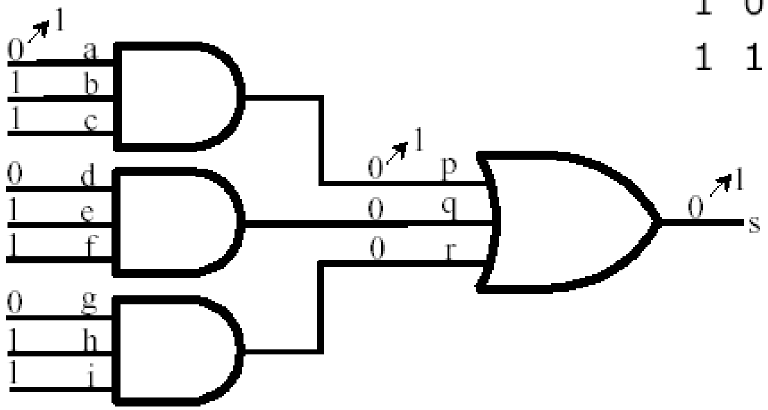
Testing an AND-OR Network



(a) stuck-at-0 test

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

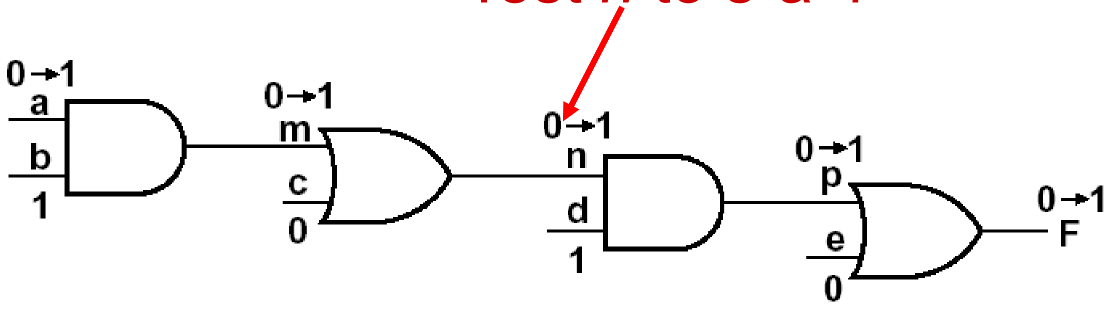
a	b	c	d	e	f	g	h	i	Faults Tested
1	1	1	0	X	X	0	X	X	a0, b0, c0, p0
0	X	X	1	1	1	0	X	X	d0, e0, f0, q0
0	X	X	0	X	X	1	1	1	g0, h0, i0, r0
0	1	1	0	1	1	0	1	1	a1, d1, g1, p1, q1, r1
1	0	1	1	0	1	1	0	1	b1, e1, h1, p1, q1, r1
1	1	0	1	1	0	1	1	0	c1, f1, i1, p1, q1, r1



(b) stuck-at-1 test

Path Detection & Sensitization: Small Example

Test *n* to s-a-1



n=0 =>

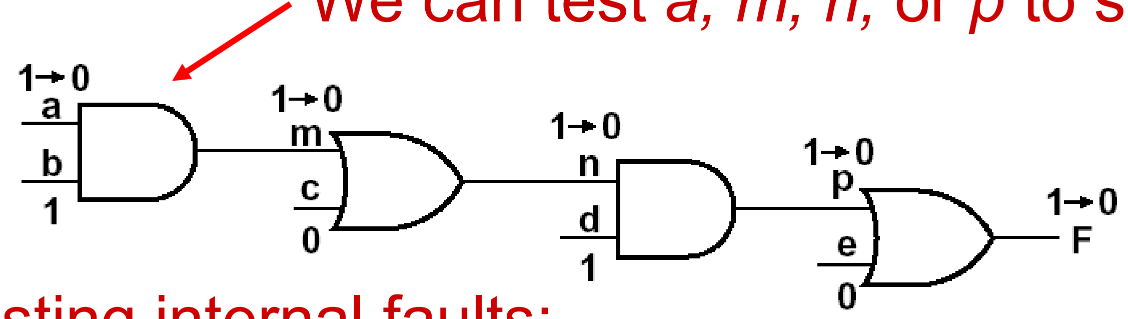
m=0, c = 0 =>

a=0, b=1, c=0

d=1, e=0

Change *a* to 1 =>

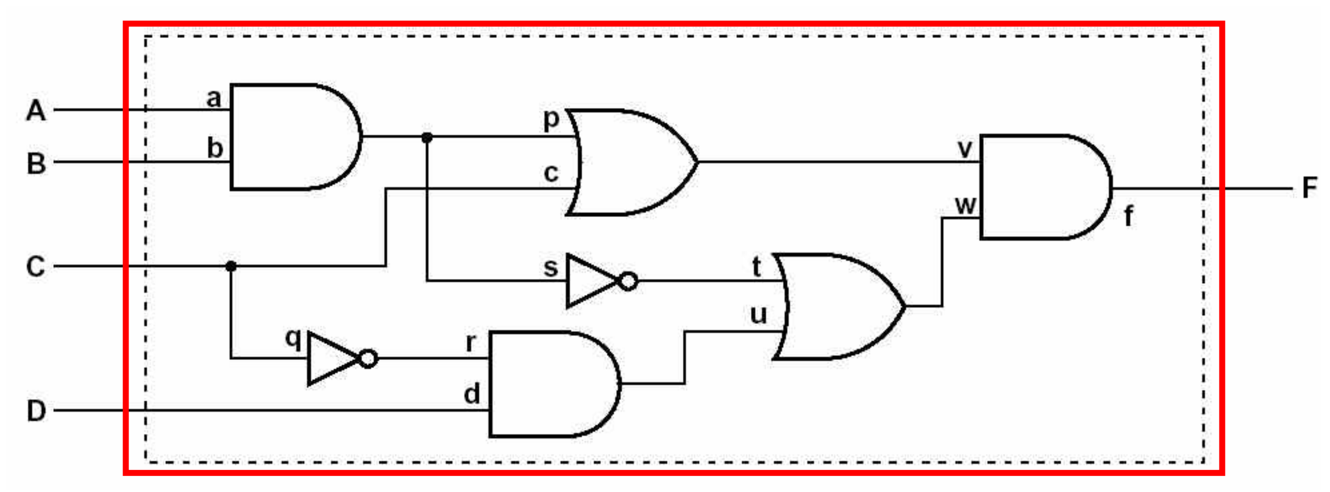
We can test *a, m, n,* or *p* to s-a-0



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

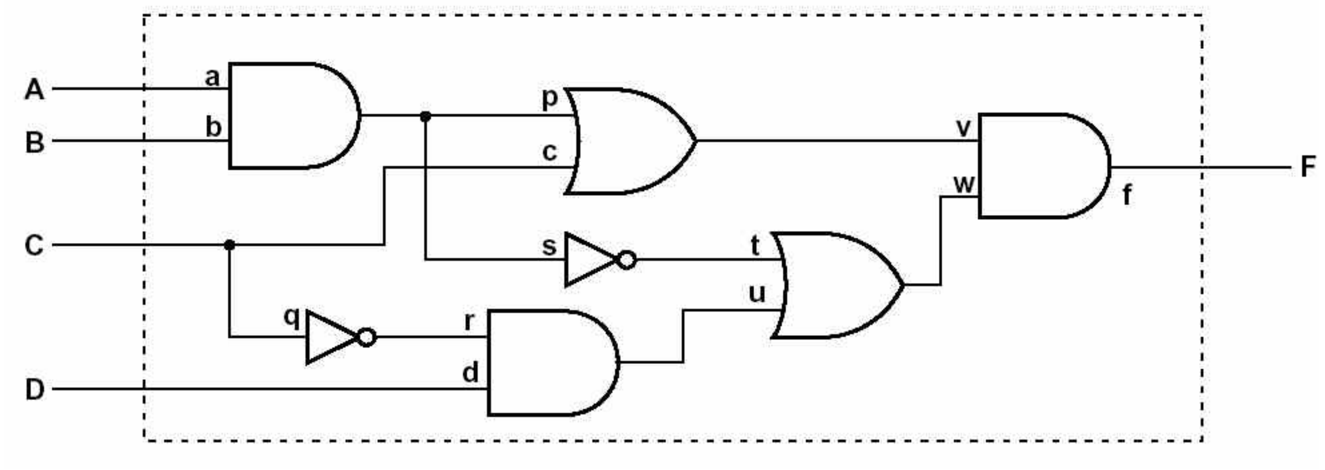
An Example

- 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)



- 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

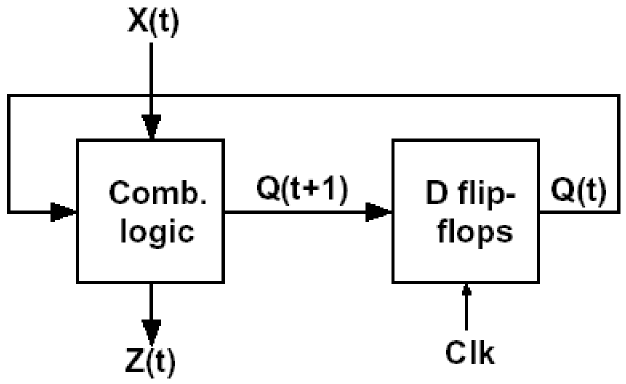
	0	1
a	+	&
b	+	*
c	*	&
d	+	%
p	+	*
q	#	+
r	+	#
s	#	*
t	*	#
u	+	#
v	+	&
w	+	#
f	+	#

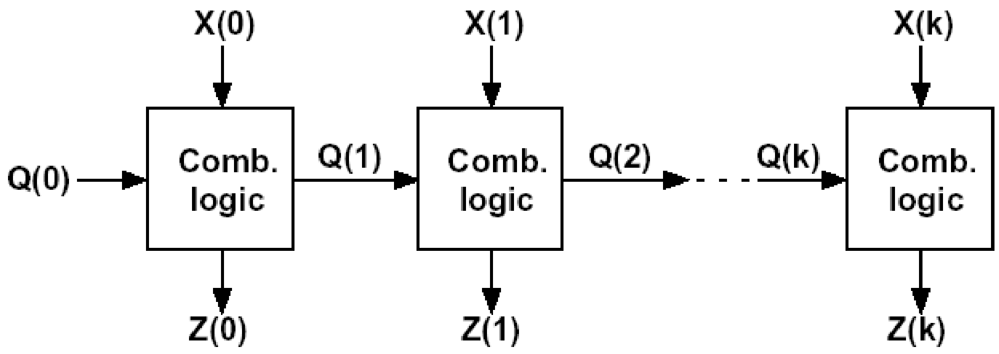
Testing Sequential Logic

- 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)

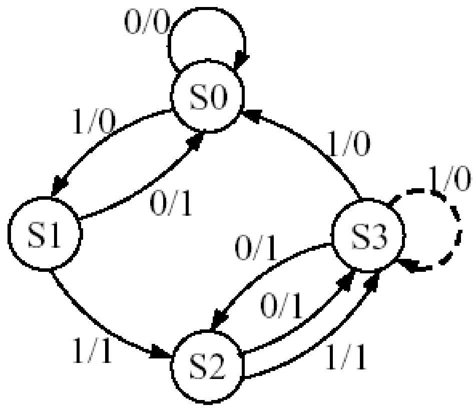
- In practice, if the network has N or fewer states, then apply only input sequences of length less than or equal 2N-1
- Example
  - consider a network which includes 5 inputs, 1 output, and 4 states
  - total number of test sequences: $(2^5)^7 = 2^{35}$ => infeasible (!)
  - derive a small set of test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





Testing Sequential Logic (cont'd)

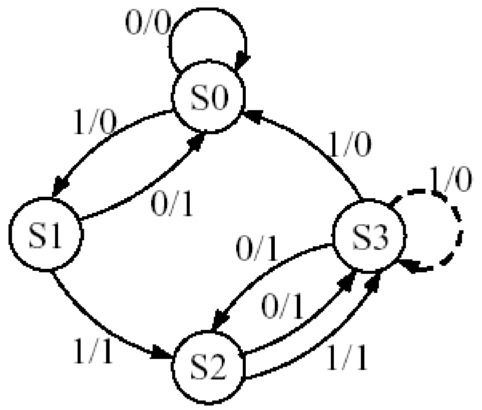
- Consider input sequence
  - X = 0 1 0 1 1 0 0 1 1
  - Output sequence
    Z = 0 0 1 0 1 1 1 1 0
  - If we change the network S3->S0 => S3->S3, the output sequence will be the same
- Find distinguishing sequence
  - an input sequence that will distinguish each state from the other states



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	S2	S0	1	0

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

Testing Sequential Logic (cont'd)



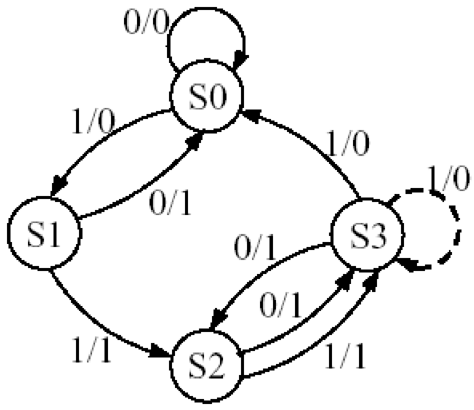
Q1Q2	State	Next State X=0	Next State 1	Output X=0	Output 1
00	S0	S0	S1	0	0
10	S1	S0	S2	1	1
01	S2	S3	S3	1	1
11	S3	S2	S0	1	0

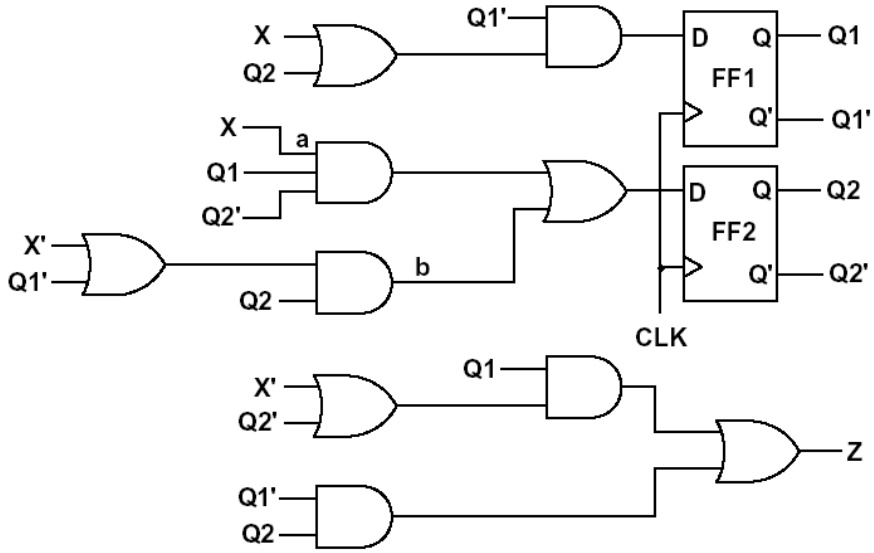
Verify each entry in the table using the following sequences:

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	(S1to 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



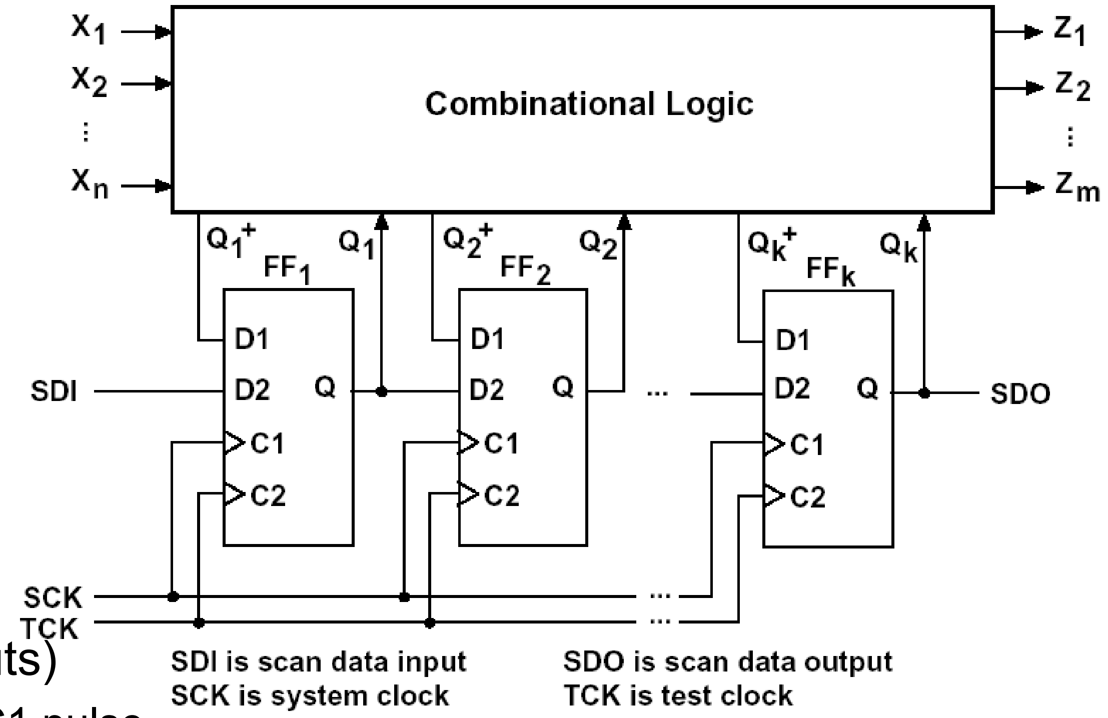


Scan Testing

- 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 => <u>Scan path testing</u>

Scan Path Testing

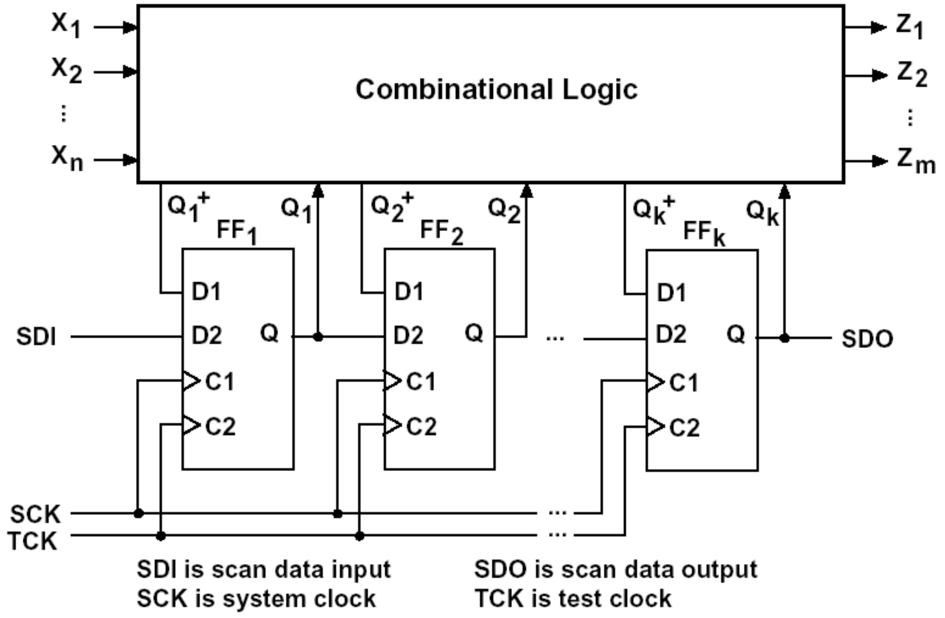
- Sequential network is separated into a combinational logic part and a state register composed of flip-flops



- 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: $X_1X_2...X_N$
  - outputs: $Z_1Z_2...Z_N$
- Testing
  - FFs are set to a specified state using the SDI and TCK
  - test vector is applied $X_1X_2...X_N$
  - outputs $Z_1Z_2...Z_N$ 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

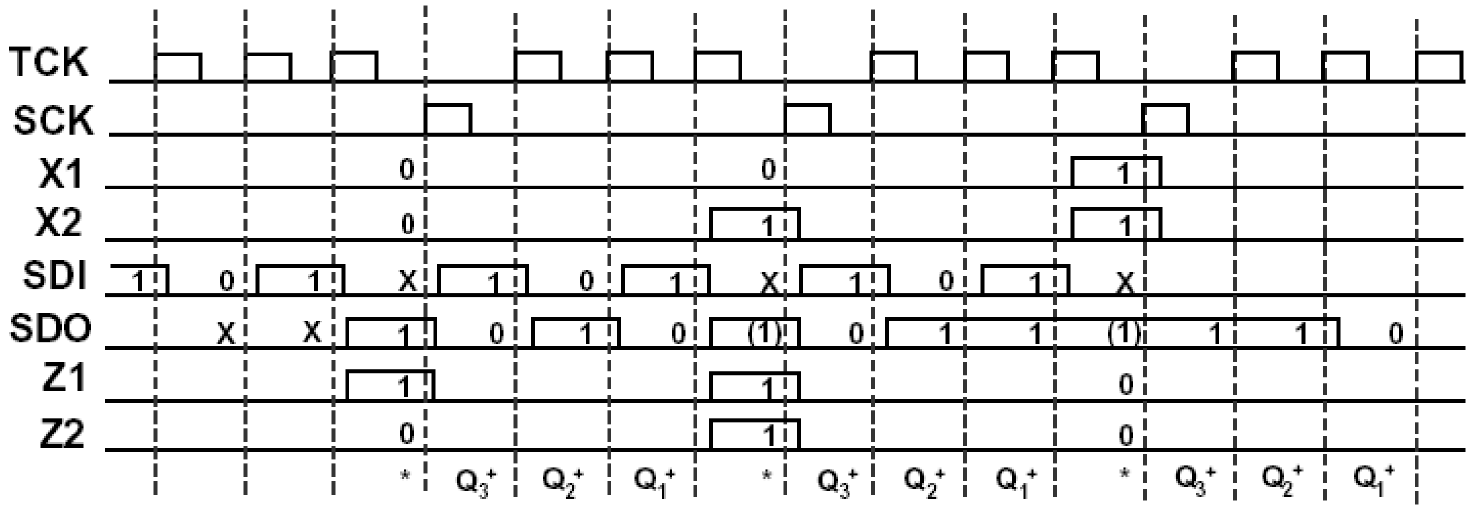


Scan Path Testing: An Example

- SQ: X_1X_2, $Q_1Q_2Q_3$, Z_1Z_2

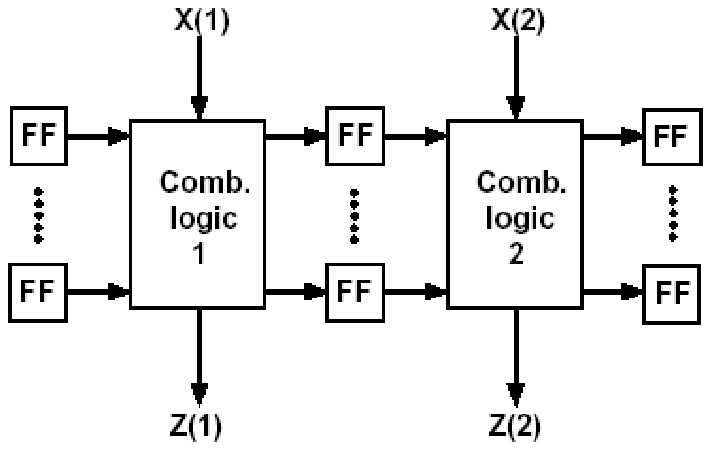
One row of the state transition table:

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

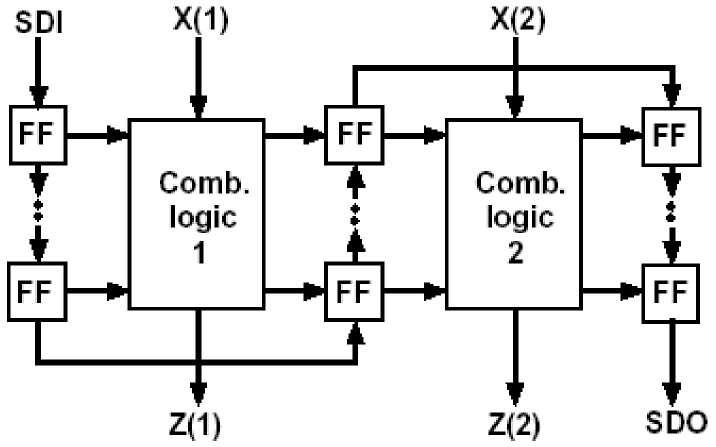


* Read output (output at other times not shown)

Scan Chain

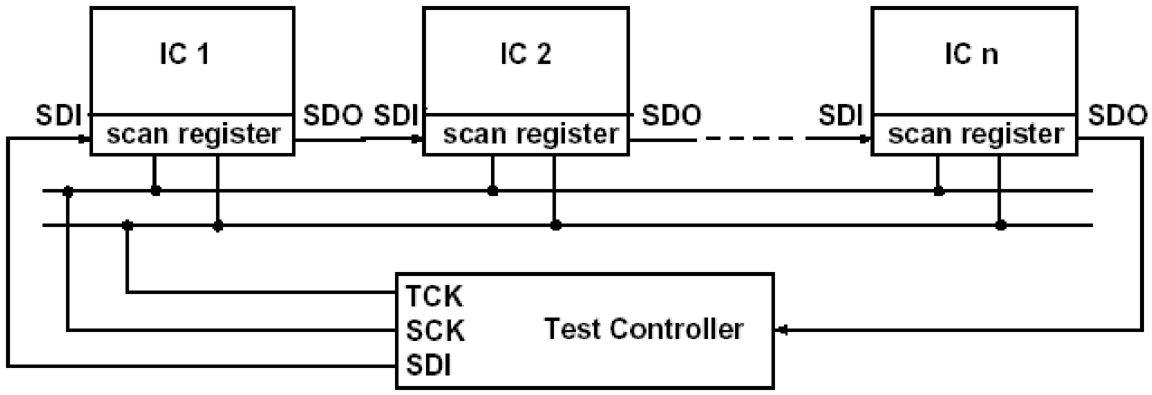


(a) Without scan chain



(b) With scan chain added

Scan Test with Multiple ICs

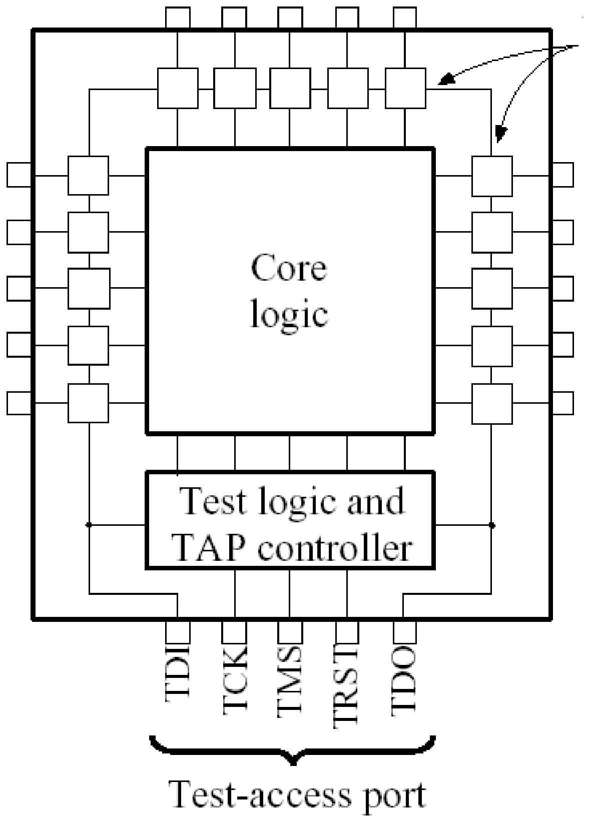


Boundary Scan

- 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
- <u>Boundary scan test methodology:</u>
  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)

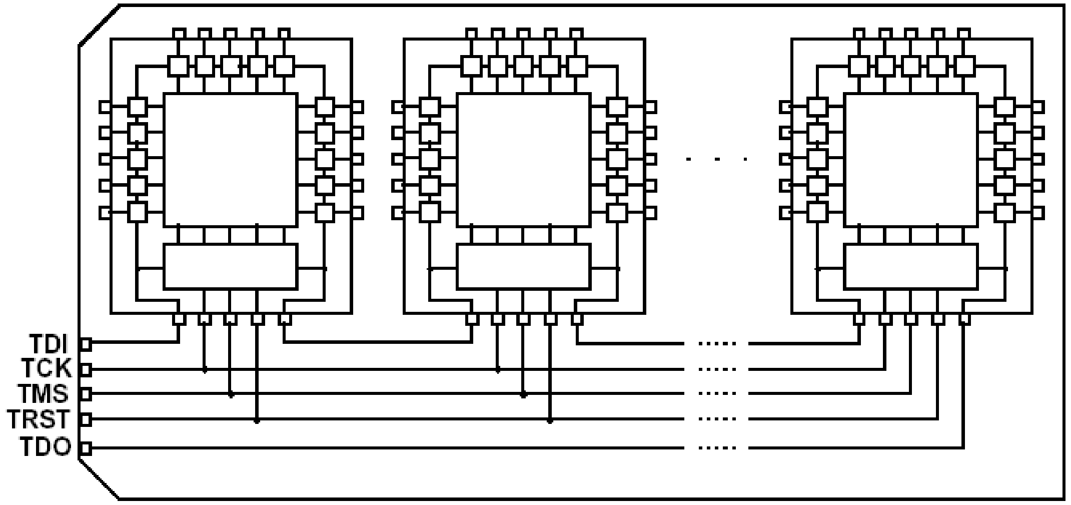


Boundary scan cells

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

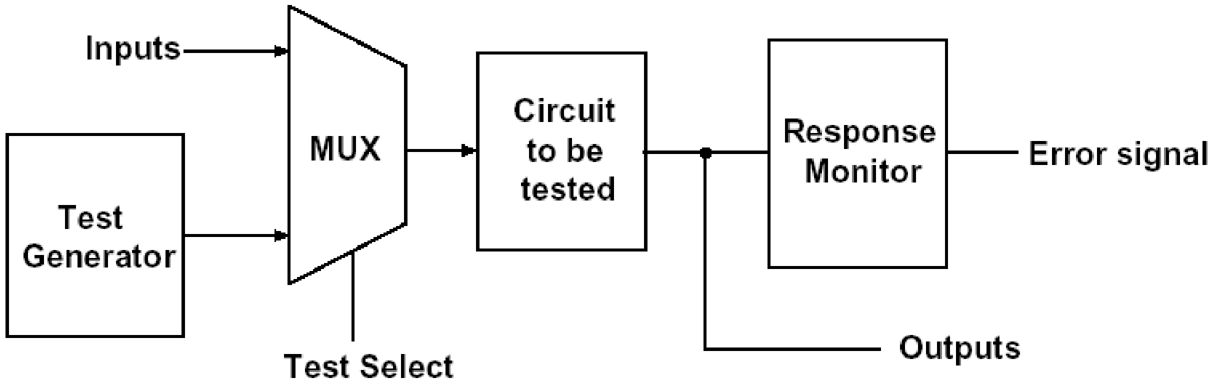


- BSRs in the ICs are linked together serially in a single chain with input TDI and output TDO.
- TCK, TMS, TRST are connected in parallel to all of the ICs.

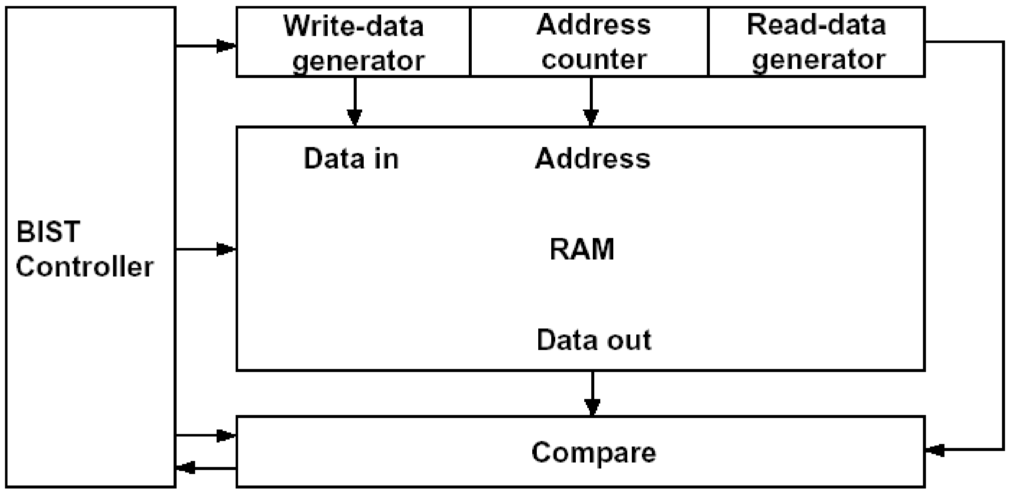
Built-In Self-Test

- 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

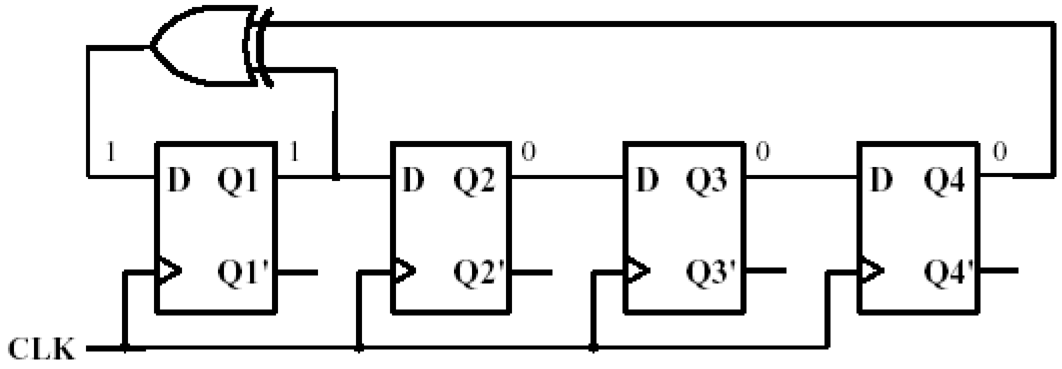
Generic BIST Scheme



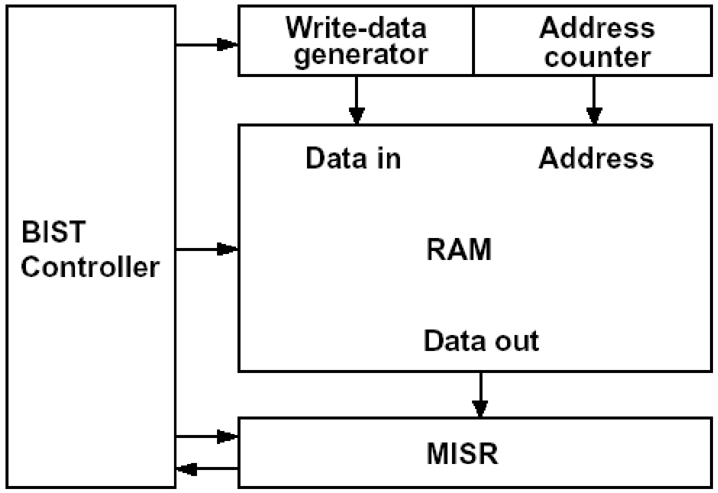
Self-Test Circuit for RAM



Linear Feedback Shift Registers (LFSR)



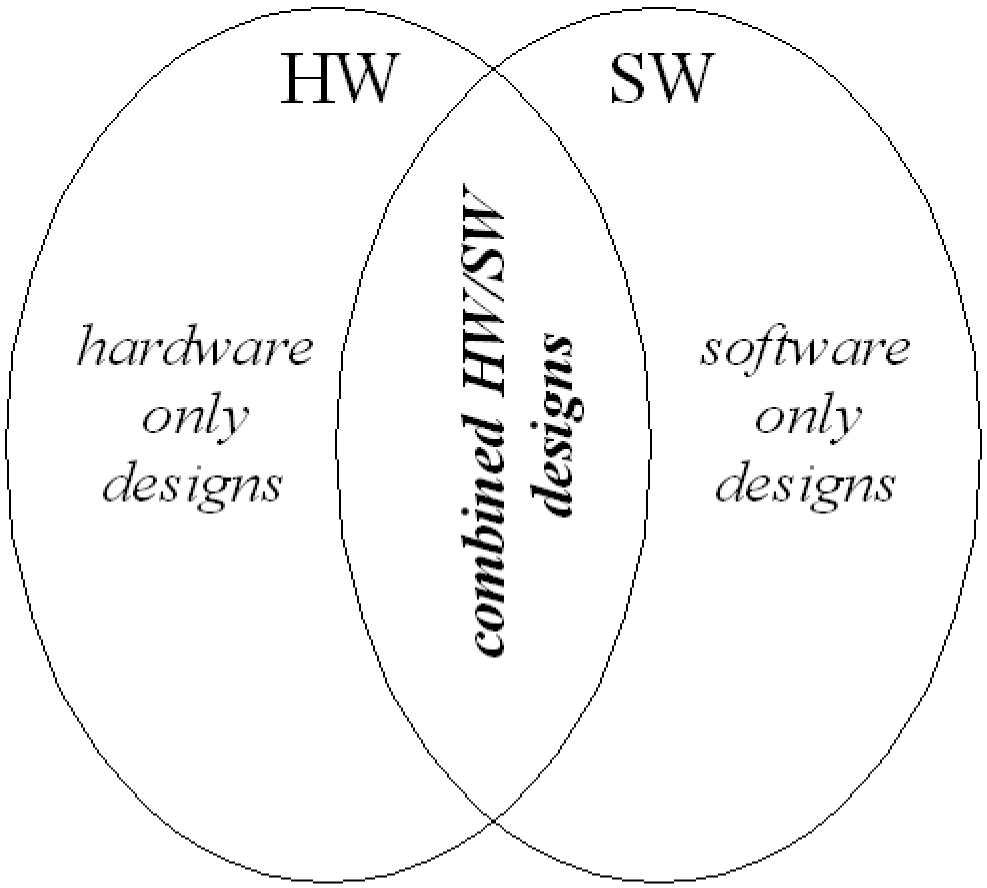
Self-Test Circuit for RAM with Signature Regs



MISR – Multiple Input Signature Register

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

Hardware/Software Design Space Continuum



Hardware/Software Trade-offs

- Performance of hardware for a given function can be much faster than a software implementation because
  - There is little or no instruction fetch overhead like there is in a traditional instruction set processor
  - It can make use of the available fine grain concurrency and execute many operations in parallel
- Software is better suited for irregularly-structured control structures
- Traditionally software was considered to be more flexible and field upgradable when compared to hardware
  - FPGA's blures this distinction though

Hardware/Software Trade-offs

- Cost of implementing a function in hardware is usually much greater than implementing the same function in software
  - Requires additional chip area in an Application Specific Integrated Circuit or board area in a PC board or reconfigurable logic area in an FPGA.
  - Usually takes more time to create a hardware design than a software one
- Both hardware and software resources are limited but hardware resources tend to have tighter limits and should be reserved for elements that have the greatest effect on the performance (or possibly other important constraints such as reliability)

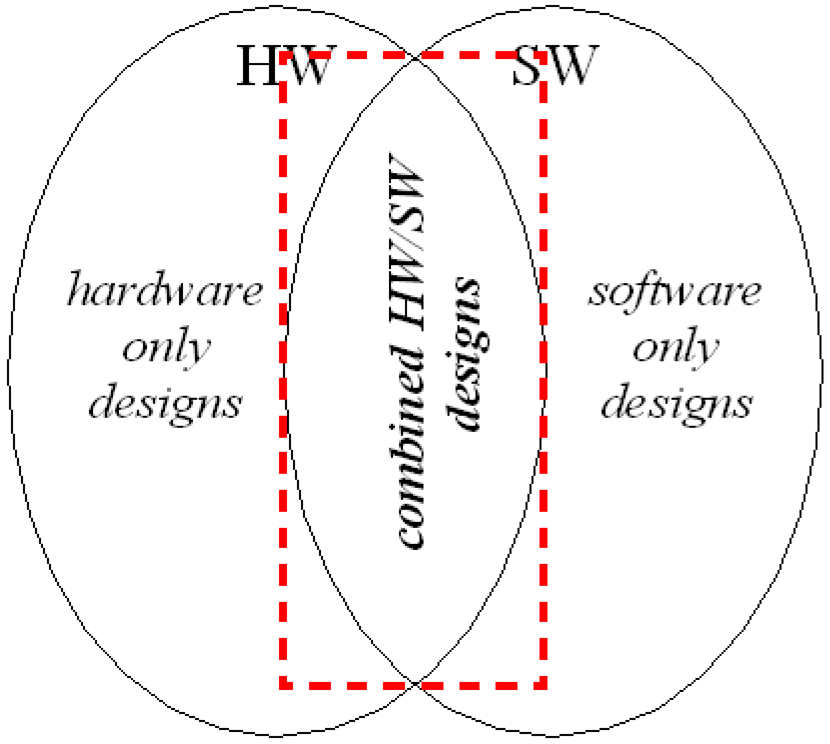
Hardware/Software Trade-offs

- Both hardware and software solutions allow for reduced energy consumption
  - Dynamic and Static Voltage Frequency Scaling, DVFS
  - Dynamic power management
  - Power gating with retention modes
  - Clock Gating (single and multi-level)
- Specific hardware energy consumption strategies can be applied at a finer grain which can result in extremely low power consumption for custom hardware.
- These features are also often dynamically controllable by software at a courser grain using commercially available hardware for much less cost.

Differences between ISP Programming and FPGA Reconfiguration

- Instruction Set Processors, ISPs, such as microcontrollers, allow you to change the program or programs that execute in one or more thread control units.
- FPGAs and other reconfigurable hardware devices allow you to reconfigure both the control units and the data paths of the device.
- In both cases this allows one to increase the usability of the devices involved.
- The allowable number of configurations is much smaller for an ISP when compared to an FPGA but the size of the program (reconfiguration information) is also much smaller!

Embedded Systems



- Application-specific systems which contain hardware and software tailored for a particular task
- Generally part of a larger system
- Responds to external inputs and drives external outputs
- Often must respond in real-time
- Must adhere to strict area (footprint), performance, and power consumption constraints

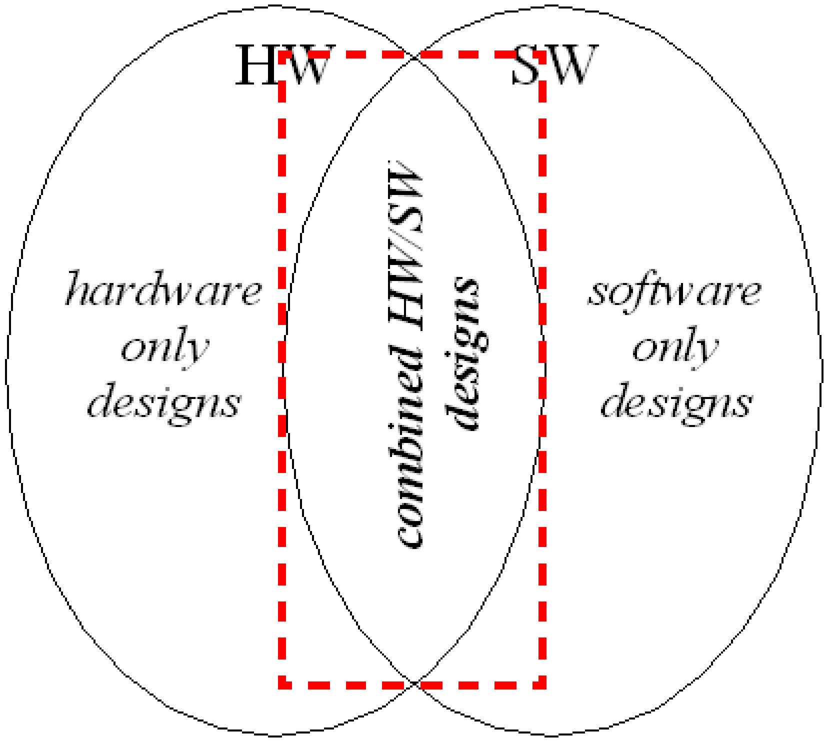
Embedded Systems Examples

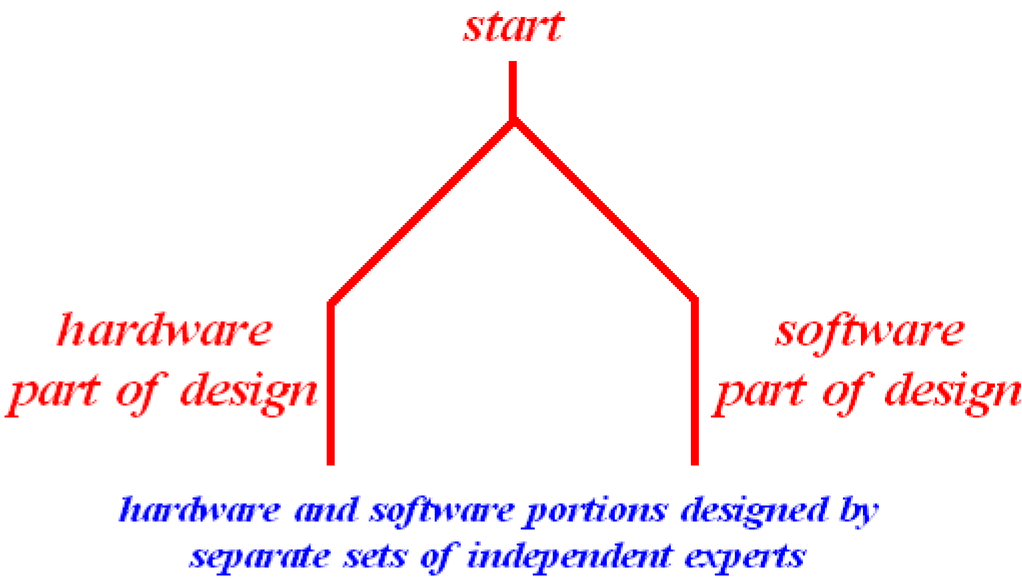
- Automotive: Anti-lock breaks, engine control, dynamic ride control, engine diagnostics
- Consumer Electronics: cameras, DVD Controllers, Microwave Ovens, Toasters, Refrigerators, Washers, Smart Card Controllers, RFID systems, Stand Alone GPS systems, TV remote controller, landline telephones, smart scales,
- Industrial Process Controllers: motor control systems , electronic data acquisition and supervisory control, automated laboratory instrument control
- Computer Peripherals: printers, external disk controllers, FAX controllers
- Cell Phones and Peripherals: wireless headsets, wifi bridging devices
- Medical Applications: ECG display and diagnostics, blood cell recorder/ analyzer, patient monitor system
- Network Systems: Routers, network switches, external firewalls

Traditional HW/SW Design Process

- System requirements are developed and then analyzed to determine the “level” of technology required to fulfill these needs.
- Hardware and software teams then independently develop their designs
  - combining of design efforts occur late in the development cycle during the first prototype testing.
- Often leads to a generalized hardware platform being selected and all specialization occurring in the software.

Traditional HW/SW Design Process



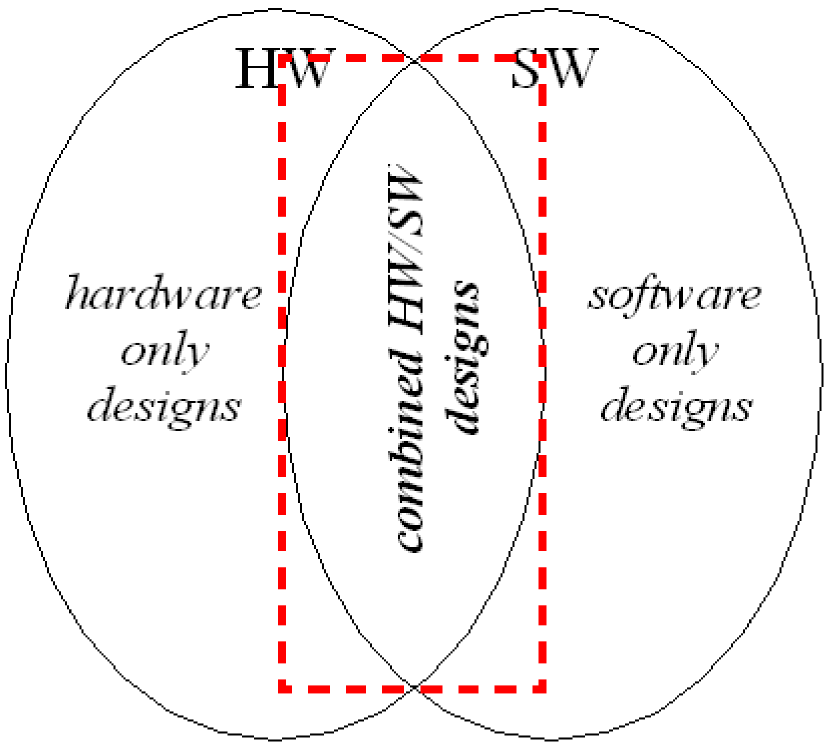

start
hardware part of design
software part of design
hardware and software portions designed by separate sets of independent experts

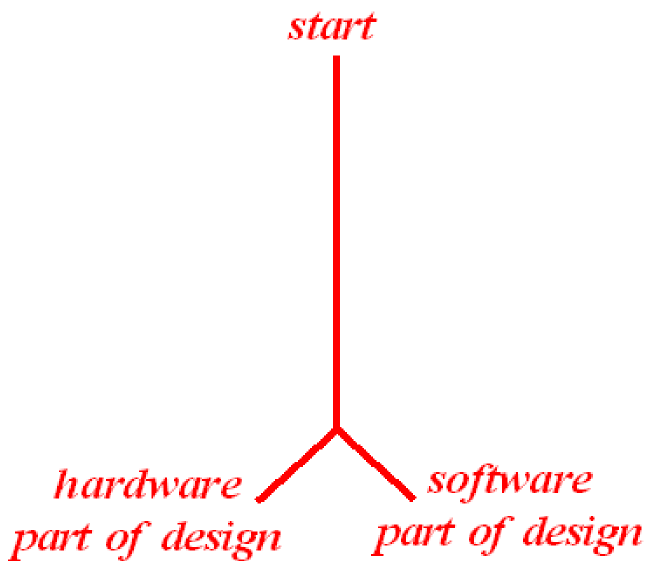
Hardware/Software Co-Design

Meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design.

"Hardware/Software Co-Design", Giovanni De Micheli, and Rajesh K. Gupta, Readings in Hardware/Software Codesign Academic Press, 2002

Hardware/Software Codesign





Hardware and software portions designed by the same group of experts with cooperation with one another. Placement of functionality can occur later in the design cycle

Hardware/Software Co-Design

Meeting system level objectives by exploiting the synergism of hardware and software through their concurrent design.

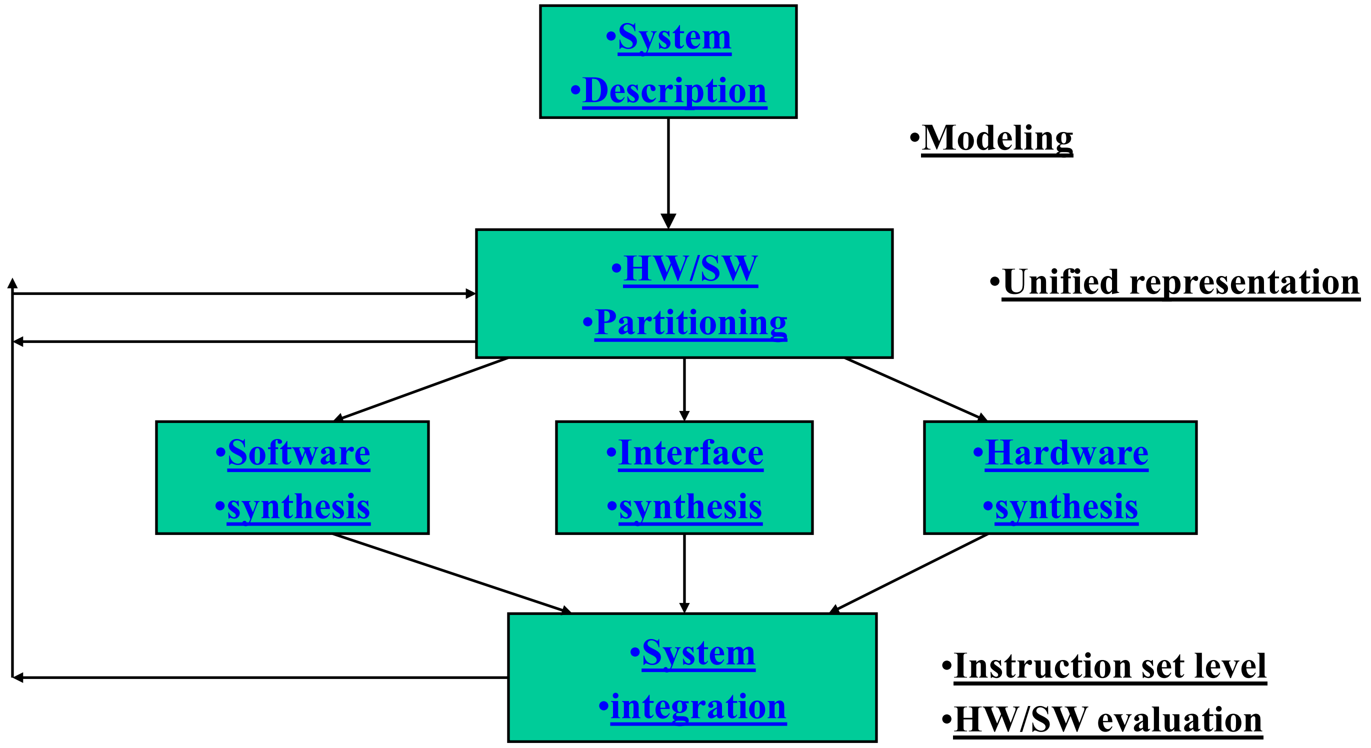
"Hardware/Software Co-Design", Giovanni De Micheli, and Rajesh K. Gupta, Readings in Hardware/Software Codesign Academic Press, 2002

Hardware/Software Co-Design

Design Flow for HW/SW Co-Design

- specification
- modeling
- design space exploration and partitioning
- synthesis and optimization
- validation
- implementation

Hardware/Software Co-Design



•System
•Description
•Modeling
•HW/SW
•Partitioning
•Unified representation
•Software
•synthesis
•Interface
•synthesis
•Hardware
•synthesis
•System
•integration
•Instruction set level
•HW/SW evaluation