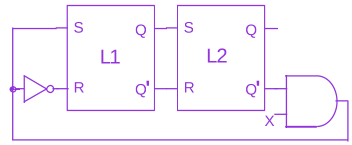
Outline

- ❍ Last time:
  - ➜ **Race-free State Assignment**
  - ➜ **Excitation Equations**
  - ➜ **Design Example: Implementation**
  - ➜ **Summary of Asynchronous Design Process**
- ❍ This lecture:
  - ➜ **Asynchronous Examples**
  - ➜ **State Assignment in Asynchronous**
  - ➜ **Error Detection & Correction**
  - ➜ **Review: Bandwidth and Latency**
  - ➜ **Binary Decision Diagrams**

Race Conditions and Cycles



Q_1Q_2 \ X	0	1
00	(00)	10
01	00	00
11	01	01
10	01	11

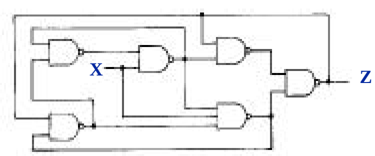
Q_1Q_2 \ X	0	1
00	(00)	
01	00	
11	01	
10	01	11

Race Conditions

Q_1Q_2 \ X	0	1
00	(00)	11
01	00	(01)
11	01	(11)
10	(10)	(10)

Example 3: Asynchronous Analysis

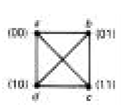
(a) Analyze the following asynchronous network using a flow table. Starting in the stable total state state for which X=Z=0, determine the state and the output sequences when the input sequence is X=0, 1, 0, 1, 0,...
(b) Find any critical races which are present in the table.



Race-Free State Assignment

- ❍ A race-free state assignment for any 4-row table can be found using three state variables
- ❍ Example:

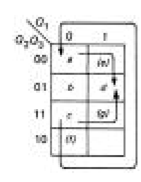
	00	01	11	10
a	(a)	c	d	(a)
b	a	d	(b)	(b)
c	d	(c)	b	a
d	(d)	(d)	(d)	b



column 00	$b \to a,\ c \to d$
column 01	$a \to c,\ b \to d$
column 11	$a \to d,\ c \to b$
column 10	$c \to a,\ d \to b$

Race-Free State Assignment

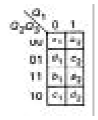
- ❍ The assignment below is a universal state-assignment map that will work for any 4-row table
- ❍ The example can then be expanded as shown, with the additional rows added

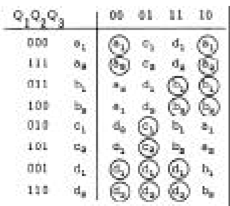


$Q_1Q_2Q_3$		00	01	11	10
000	a	(a)	f	e	(a)
001	b	a	d	(b)	(b)
011	c	g	(c)	b	f
101	d	(d)	(d)	(d)	b
100	e	-	-	d	-
010	f	-	c	-	a
111	g	d	-	-	-

Race-Free State Assignment

- Another universal assignment for 4-row tables is shown below (M. Mano, *Logic and Computer Design*, Prentice Hall)
- Using this approach, the expanded example from before becomes as shown below right
- This approach faster than previous general approach above. Why?

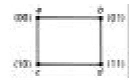




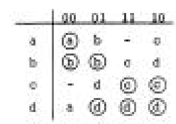
Race-Free State Assignment

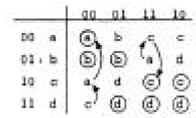
- Don't cares can be used to make race-free assignment
- In example below-left, we need:

col_{00} d→a; col_{01} a→b, c→d; col_{11} b→c; col_{10} a→c, b→d

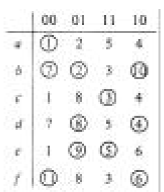


- Looks like we need extra rows, but use the don't cares:





Shared-Row Assignments



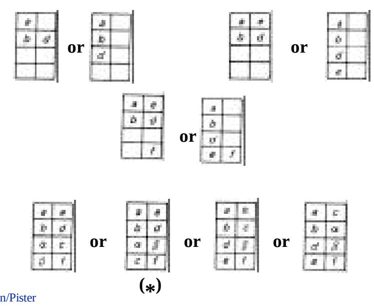
column 00: e, c → a d → b
column 01: a → b c, f → d
column 11: b, f → c a, d → e
column 10: a, c → d e → f

Consider the example above. We need at least three state variables and maybe even four.

e→a, c→a; e→c→a; c→e→a
or any of these going through an intermediate state

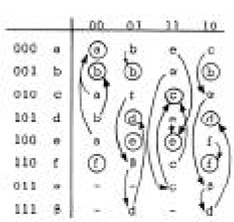
Shared-Row Assignment

- How do we get there? Trial and error...



Shared-Row Assignment

- Result of using second option from previous slide:



Error Detection & Correction

- As transistors & wires become smaller, the number of electrons used to represent a stored 1 or 0 decreases
- The probability that a cosmic ray, alpha particle, local electric field (crosstalk) or some other disturbance will change a stored value (or even the value on a wire) increases
- Two aspects:
  - Error detection (e.g. parity)
  - Error correction (e.g. Hamming Codes)
    - 2-bit detect, 1-bit correct

Hamming Codes

- ❍ Most common types of error-correcting codes used in RAM
- ❍ Based on work of R. W. Hamming
- ❍ k parity bits are added to an n-bit word, forming a new n+k bit word
- ❍ The positions numbered with powers of two are reserved for the parity bits
- ❍ Can be used with words of any length
- ❍ Example: 8-bit data word 11000100

1	2	3	4	5	6	7	8	9	10	11	12
P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

Hamming Codes

- ❍ Calculate parity bits as follows:

P_1= xor(3,5,7,9,11) = $1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 0 = 0$
P_2= xor(3,6,7,10,11) = 0
P_4= xor(5,6,7,12) = 1
P_8= xor(9,10,11,12) = 1

1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	1	0	0	1	0	1	0	0

- ❍ When bits are read from memory, compute check bits:

C_1= xor(1,3,5,7,9,11)
C_2= xor(2,3,6,7,10,11)
C_4= xor(4,5,6,7,12)
C_8= xor(8,9,10,11,12)

Hamming Codes

- ❍ $C=C_8C_4C_2C_1$=0000 indicates no error has occurred
- ❍ Examples:

1	2	3	4	5	6	7	8	9	10	11	12	
0	0	1	1	1	0	0	1	0	1	0	0	no error
1	0	1	1	1	0	0	1	0	1	0	0	error in bit 1
0	0	1	1	0	0	0	1	0	1	0	0	error in bit 5

C_8	C_4	C_2	C_1	
0	0	0	0	no error
0	0	0	1	error bit 1
0	1	0	1	error bit 5

Hamming Codes

- ❍ For n data bits and k check bits, $n+k \le 2^k-1$
- ❍ Grouping of bits for parity generation can be observed from listing of binary numbers:

	B_1	B_2	B_3
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

B_1=1 for (1,3,5,7)
B_2=1 for (2,3,6,7)
B_3=1 for (4,5,6,7)

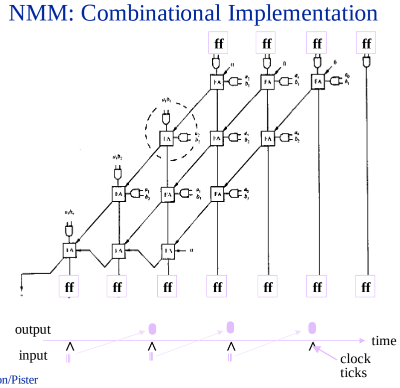
Hamming Codes

- ❍ Add P_{13} as an overall parity bit: 001110010100P_{13}
- ❍ Detect/Correct Rules as follows:

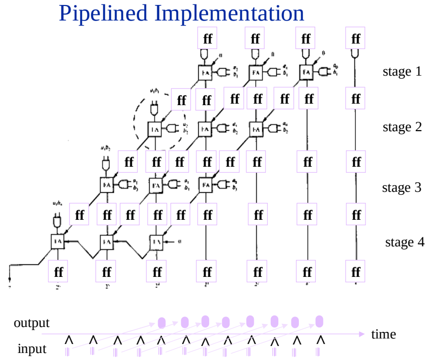
If C=0, P=0 no error
If C≠ 0, P=1 single error, correctable
If C ≠ 0, P=0 double error, uncorrectable
If C=0, P=1 error occurred in P_{13}

- ❍ Can also have far more sophisticated schemes
- ❍ Covered in detail in communication & information theory courses

NMM: Combinational Implementation



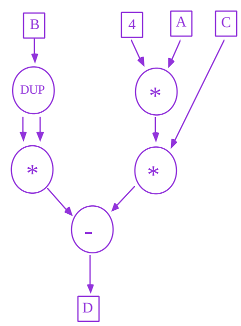
Pipelined Implementation



Bandwidth and Latency

- *Latency*: The minimum time to get the first result (sec.) - the one-time cost.
- *Bandwidth*: The maximum rate at which results can be produced in the steady-state (values/sec.) - the incremental cost
- For example, disc drives, RAM (normal, video), networks, signal-processors (e.g. HDTV, radar)

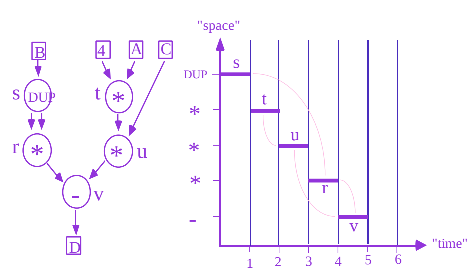
Simple Dataflow Description: B^2 - 4AC



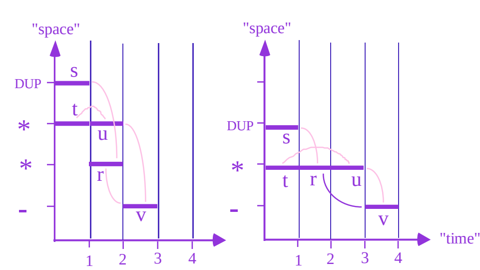
Scheduling of Functional Units

- Given a certain number of functional units (e.g. ALU's, RAM's), schedule a particular computation (from the HDL or software) onto a particular functional unit at a particular time relative to other operations.
- Analogous problem scheduling "wires"
- What about the values on the wires?

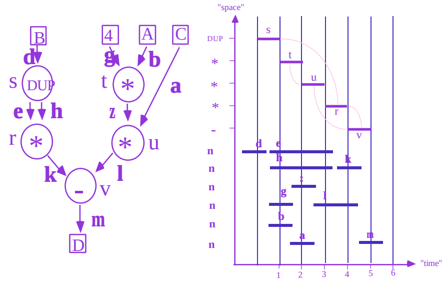
Allocation of Functional Units: Data Dependencies



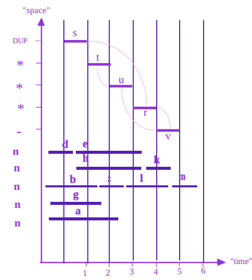
Allocation of Functional Units: Time-Space Tradeoffs



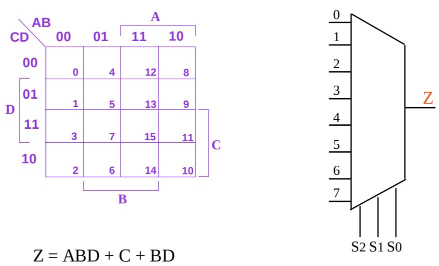
Allocation of Functional Units: Sharing of Signal Links



Allocation of Functional Units: More Efficient Use of Links



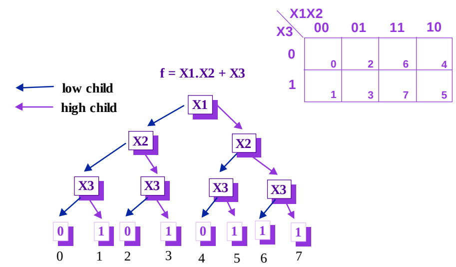
Design Using Multiplexers



Z = ABD + C + BD

Binary Decision Diagrams

- In general: fv = Xv' flow + Xv fhigh



Implementation of Logic Using Switches

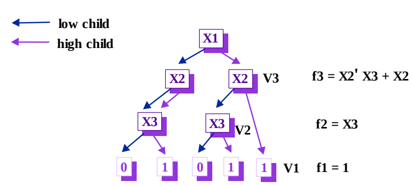
Shannon Expansion:

(T15) F(X,Y,Z) = X•F(1,Y,Z) + X'•F(0,Y,Z)

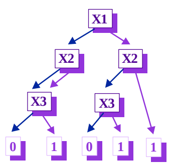
(T15') F(X,Y,Z) = (X+F(0,Y,Z))•(X'+F(1,Y,Z))

Binary Decision Diagrams

f = X1.X2 + X3



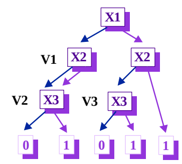
Use of BDDs for Verification



- ❍ **Condition on edges forces order on variables.**
- ❍ **Order must be consistent with all edges.**
- ❍ **For each edge, parent before child in the order.**
  - ➜ X1 X2 X3

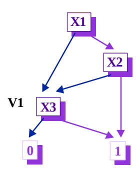
Use of BDDs for Verification

- ❍ **V1 is a redundant vertex**
- ❍ **V2, V3 represent the same function**
- ❍ **A BDD is a reduced binary decision graph**
  - ➜ **Reduction is O(N*log(N)) for N verticies**

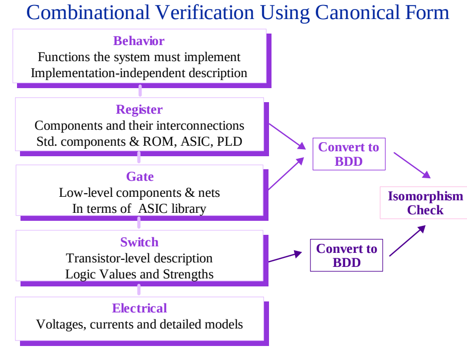


Use of BDDs for Verification

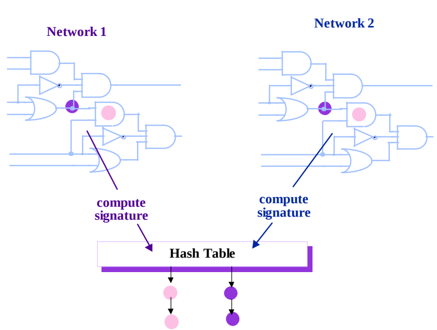
- ❍ **Each vertex corresponds to a partial assignment of inputs.**
  - ➜ **V1: 0-, 10**
- ❍ **Result of reduction is a canonical form.**



Combinational Verification Using Canonical Form



Connectivity Verification



Connectivity Verification

(1) Read Network1 and Network2 into separate graph data structures (usually, node = transistor or gate, edge = connection).

(2) Compute signatures for nodes or edges or both.

- ➔ **Type-specific: Gate type, #inputs, #outputs**
- ➔ **Network-specific (local): Fanin types, #fanouts, fanout types**
- ➔ **Network-specific (global): Distance from primary inputs, primary outputs, nodes that are known to be the same in each network (e.g. named, primary inputs or outputs ("seeds")).**

(3) Hash signatures from both networks into single hash table.

(4) If a hash table cell has:

- ➔ **> 2 nodes: ignore for now**
- ➔ **= 2 nodes: a match has been found**
- ➔ **= 1 node: an (easy!) error has been found**

(5) Add links between networks for nodes that have been matched.

(6) Recompute hash functions for un-bound nodes and repeat until done or no change.