CHAPTER 6

OPTIMIZATIONS AND TRADEOFFS

6.1 EXERCISES

SECTION 6.1: INTRODUCTION

6.1) Define the terms “optimization” and “tradeof.f”

An optimization improves all criteria of interest to us, whereas a tradeoff improves certain criteria at the expense of other criteria.

6.2) A homeowner wishes to increase the amount of light inside the house during the day, with the only criteria of interest being the amount of light and the cost of electricity. Describe how to increase the light via: (a) an optimization, (b) a tradeoff.

(a) An optimization would be to add a window or sunroof (note: the initial cost of installing those items was not listed as a criteria of interest and thus can be neglected). The window or sunroof adds light without changing the cost of electricity.

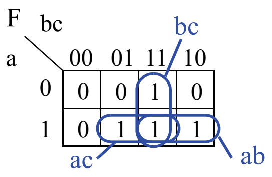
(b) A tradeoff would be to turn on a lamp during the day. The light would increase, but at the expense of higher electric cost.

SECTION 6.2: COMBINATIONAL LOGIC OPTIMIZATIONS AND TRADEOFFS

6.3) Perform two-level logic size optimization for `F(a,b,c)` = `ab'c` + `abc` + `a'bc` + `abc'` using (a) algebraic methods, (b) a K-map. Express the answers in sum-of-products form.

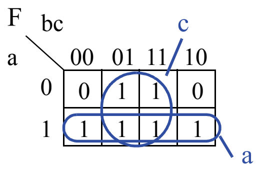
(a) F = ab'c + abc + a'bc + abc'
F = ab'c + abc + abc + a'bc + abc + abc'
F = ac(b' + b) + bc(a + a') + ab(c + c')
F = ac + bc + ab

(b)



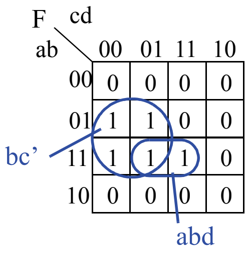
F(a,b,c) = ab + ac + bc

6.4) Perform two-level logic size optimization for `F(a,b,c)` = `a` + `a'b'c` + `a'c` using a K-map..



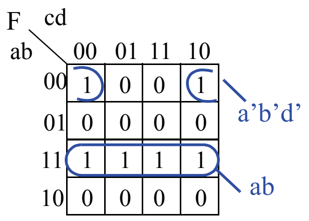
F(a,b,c) = a + c

6.5) Perform two-level logic size optimization for `F(a,b,c,d)` = `a'bc'` + `abc'd'` + `abd` using a K-map.



F(a,b,c,d) = bc' + abd

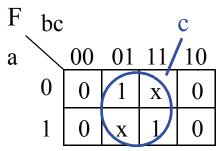
6.6) Perform two-level logic size optimization `F(a,b,c,d)` = `ab` + `a'b'd'` using a K-map.



ab \ cd	00	01	11	10
00	1	0	0	1
01	0	0	0	0
11	1	1	1	1
10	0	0	0	0

F(a,b,c,d) = ab + a'b'd'

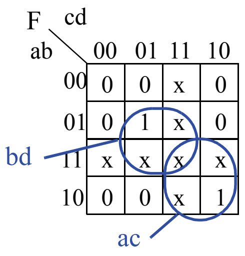
6.7) Perform two-level logic size optimization for `F(a,b,c)` = `a'b'c` + `abc`, assuming input combinations `a'bc` and `ab'c` can never occur (those two minterms represent don't cares).



a \ bc	00	01	11	10
0	0	1	x	0
1	0	x	1	0

F(a,b,c) = c

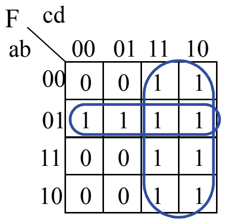
6.8) Perform two-level logic size optimization for `F(a,b,c,d)` = `a'bc'd` + `ab'cd'`, assuming that `a` and `b` can never both be `1` at the same time, and that `c` and `d` can never both be `1` at the same time (i.e., there are don't cares).



ab \ cd	00	01	11	10
00	0	0	x	0
01	0	1	x	0
11	x	x	x	x
10	0	0	x	1

F(a,b,c,d) = ac + bd

6.9) Consider the function `F(a,b,c)` = `a'c` + `ac` + `a'b`. Using a K-map: (a) Determine which of the following terms are implicants (but not necessarily prime implicants) of the equation: `a'b'c', a'b', a'bc, a'c, c, bc, a'bc', a'b`. (b) Determine which of those terms are prime implicants of the function.



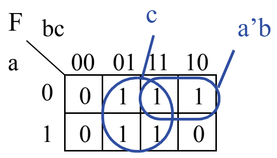
ab \ cd	00	01	11	10
00	0	0	1	1
01	1	1	1	1
11	0	0	1	1
10	0	0	1	1

Implicants listed in the question:
~~a'b'c'~~, ~~a'b'~~, a'bc, a'c, c, bc, a'bc', a'b

(b) Prime implicants: a'b, c

6.10) For the function `F(a,b,c) = a'c + ac + a'b`, determine all prime implicants and all essential prime implicants: (a) using a K-map, (b) using the tabular method.

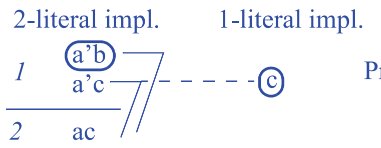
(a)



a'b and c are both prime implicants and also essential prime implicants; each is the only cover of some particular 1.

(b)

Step 1:



Prime implicants

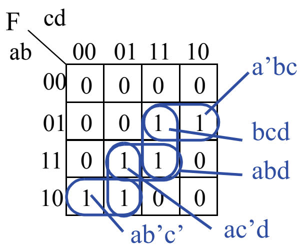
Step 2:

Minterm	a'b	c
a'b	X	
a'c		X
c		X

All prime implicants are essential; stop

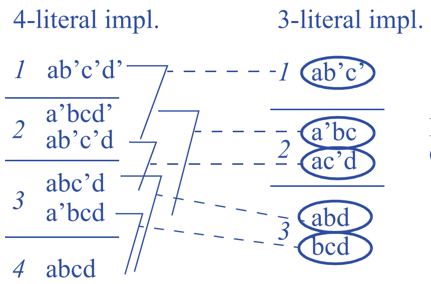
6.11) For the equation `F(a,b,c,d) = ab'c' + abc'd + abcd + a'bcd + a'bcd'`, determine all prime implicants and all essential prime implicants: (a) using a K-map, (b) using the tabular method.

(a)



Prime implicants: ab'c', ac'd, a'bc, bcd, abd
Essential prime implicants: ab'c', a'bc

Step 1:



Prime implicants are circled.
Cannot be expanded further; stop

Step 2:

Minterm	ab'c'	a'bc	ac'd	abd	bcd
ab'c'	X				
abc'd			X	X	
abcd				X	X
a'bcd		X			X
a'bcd'		X			

Essential prime implicants:
ab'c', a'bc

Step 3:
With ab'c' and a'bc, we only have abc'd and abcd left to cover. Choosing abd will cover both with only one prime implicant, so the final cover is:
F(a, b, c, d) = ab'c' + a'bc + abd

6.12) Use repeated application of the expand operation to heuristically minimize the equation `F(a,b,c) = a'b'c + a'bc + abc`. (a) Try expanding each term for each variable. (b) Instead, determine a way to randomly choose an expand operation, and then apply 5 random expands.

(a) A possible sequence of expand attempts:

F = b'c + a'bc + abc - invalid (ab'c is not in on-set)
F = a'c + a'bc + abc - valid
F = a' + a'bc + abc - invalid (a'c' is not in on-set)
F = a'c + bc + abc - valid
F = a'c + c + abc - invalid (b'c is not in on-set)
F = a'c + b + abc - invalid (bc' is not in on-set)
F = a'c + bc + bc - valid
F = a'c + bc + c - invalid (b'c is not in on-set)
F = a'c + bc + b - invalid (bc' is not in on-set)

Final equation:
F = a'c + bc + bc
(F = a'c + bc if a simple search for redundant terms is included)

(b) We may choose a heuristic which chooses a minterm to expand at random and a variable in that minterm to expand at random. One possible sequence of random

expand attempts:
F = a’b’c + a’bc + ab - invalid (abc’ is not in on-set)
F = a’b’c + bc + abc - valid
F = b’c + bc + abc - invalid (ab’c is not in on-set)
F = a’c + bc + abc - valid
F = a’c + bc + ac - invalid (ab’c is not in on-set)

6.13) Use repeated application of the expand operation to heuristically minimize the equation `F(a,b,c,d,e) = abcde + abcde' + abcd'e'`. (a) Try expanding each term for each variable. (b) Instead, determine a way to randomly choose an expand operation, and then apply 5 random expands.

(a)
One possible sequence of expand attempts:
F = bcde + abcde’ + abcd’e’ - invalid (a’bcde is not in on-set)
F = acde + abcde’ + abcd’e’ - invalid (ab’cde is not in on-set)
F = abde + abcde’ + abcd’e’ - invalid (abc’de is not in on-set)
F = abcd + abcde’ + abcd’e’ - valid
F = abcd + bcde’ + abcd’e’ - invalid (a’bcde’ is not in on-set)
F = abcd + acde’ + abcd’e’ - invalid (ab’cde’ is not in on-set)
F = abcd + abde’ + abcd’e’ - invalid (abc’de’ is not in on-set)
F = abcd + abce’ + abcd’e’ - valid
F = abcd + abc + abcd’e’ - invalid (abcd’e is not in on-set)
F = abcd + abce’ + bcd’e’ - invalid (a’bcd’e’ is not in on-set)
F = abcd + abce’ + acd’e’ - invalid (ab’cd’e’ is not in on-set)
F = abcd + abce’ + abd’e’ - invalid (abc’d’e’ is not in on-set)
F = abcd + abce’ + abce’ - valid
F = abcd + abce’ + abc - invalid (abcd’e is not in on-set)

Final equation:
F = abcd + abce’ + abce’
(F = abcd + abce’ if a simple search for redundant terms is included)

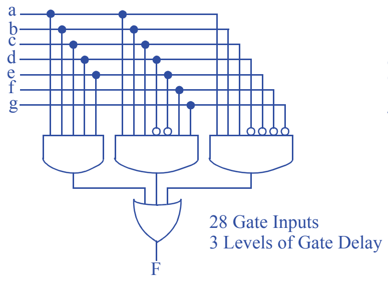
(b) We may choose a heuristic which chooses a minterm to expand at random and a variable in that minterm to expand at random. One possible sequence of random expand attempts:
F = abde + abcde’ + abcd’e’ - invalid (abc’de is not in on-set)
F = abcde + abcde’ + bcd’e’ - invalid (a’bcd’e’ is not in on-set)
F = abcde + acde’ + abcd’e’ - invalid (ab’cde’ is not in on-set)
F = abcde + abcd + abcd’e’ - valid
F = abcde + abcd + abd’e’ - invalid (abc’d’e’ is not in on-set)

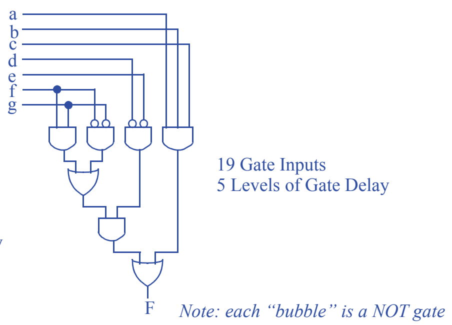
6.14) Using algebraic methods, reduce the number of gate inputs for the following equation by creating a multilevel circuit: `F(a,b,c,d,e,f,g)` = `abcde` + `abcd'e'fg` + `abcd'e'f'g'`. Assume only AND, OR, and NOT gates will be used. Draw the circuit for the original equation and for the multilevel circuit, and clearly list the delay and number of gate inputs for each circuit.

F = abcde + abcd’e’fg + abcd’e’f’g’

F = abc(de + d’e’fg + d’e’f’g’)

F = abc(de + d’e’(fg + f’g’))





SECTION 6.3: SEQUENTIAL LOGIC OPTIMIZATIONS AND TRADEOFFS

6.15) Reduce the number of states for the FSM in Figure 6.88 using the partitioning method.

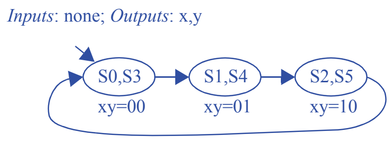
Initial groups: G1:{S0,S3}, G2:{S1,S4}, G3:{S2,S5}

G1: S0 goes to S1 (G2), S3 goes to S4 {G2} --> Next states in same group

G2: S1 goes to S2 (G3), S4 goes to S5 (G3) --> Next states in same group

G3: S2 goes to S3 (G1), S5 goes to S0 (G1) --> Next states in same group

Thus, no groups need to be partitioned further, and hence states within a group are equivalent. Replace S3 by S0, S4 by S1, and S5 by S2 to yield:



6.16) Reduce the number of states for the FSM in Figure 6.89 using the partitioning method.

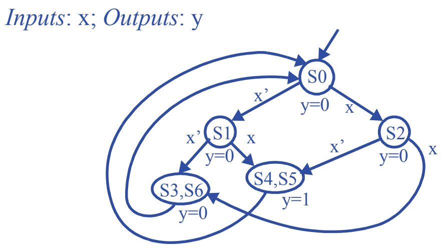
Initial groups: G1:{S0, S1, S2, S3, S6}, G2:{S4, S5}

x=0: G1: S0 -> S1 (G1), S1 -> S3 (G1), S2 -> S5 (G2), S3 -> S0 (G1), S6 -> S0 (G1)
--> Next states NOT all in same group

New groups: G1: {S0, S1, S3, S6}, G2:{S4, S5}, G3:{S2}

x=0: G1: S0 -> S1 (G1), S1 -> S3 (G1), S3 -> S0 (G1), S6 -> S0 (G1)
x=0: G2: S4 -> S0 (G1), S5 -> S0 (G1)
x=0: G3 (One state group; nothing to check)
x=1: G1: S0 -> S2 (G3), S1 -> S4 (G2), S3 -> S0 (G1), S6 -> S0 (G1)
--> Next states NOT all in same group

New groups: G1:{S0}, G2:{S4, S5}, G3:{S2}, G4:{S1}, G5:{S3, S6}
x=0: G1: (One state group; nothing to check)
x=0: G2: S4 -> S0 (G1), S5 -> S0 (G1)
x=0: G3: (One state group; nothing to check)
x=0: G4: (One state group; nothing to check)
x=0: G5: S3 -> S0 (G1), S6 -> S0 (G1)
x=1: G1: (One state group; nothing to check)
x=1: G2: S4 -> S0 (G1), S5 -> S0 (G1)
x=1: G3: (One state group; nothing to check)
x=1: G4: (One state group; nothing to check)
x=1: G5: S3 -> S0 (G1), S6 -> S0 (G1)
Thus, no groups need to be partitioned further, and hence states within a group are equivalent. Replace S6 by S3 and S5 by S4 to yield:



6.17) Reduce the number of states for the FSM in Figure 6.90 using the partitioning method.

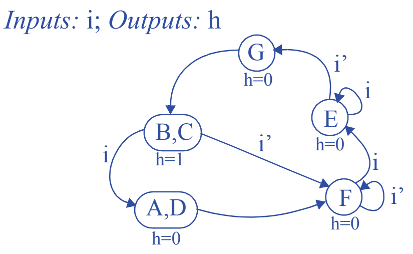
Initial groups: G1:{A, D, E, F, G}, G2:{B, C}
i=0: G1: A -> F (G1), D -> F (G1), E -> G (G1), F -> F (G1), G -> C (G2)
-->Next states NOT all in same group

New groups: G1:{A, D, E, F}, G2:{B, C}, G3:{G}
i=0: G1: A -> F (G1), D -> F (G1), E -> G (G3), F -> F (G1)
-->Next states NOT all in same group

New groups: G1:{A, D, F}, G2: {B, C}, G3:{G}, G4:{E}
i=0: G1: A -> F (G1), D -> F (G1), F -> F (G1)
i=0: G2: B -> E (G4), C -> E (G4)
i=0: G3: (One state group; nothing to check)
i=0: G4: (One state group; nothing to check)
i=1: G1: A -> F (G1), D -> F (G1), F -> E (G4)
-->Next states NOT all in same group

New groups: G1:{A, D}, G2: {B, C}, G3:{G}, G4:{E}, G5:{F}
i=0: G1: A -> F (G5), D -> F (G5)
i=0: G2: B -> E (G4), C -> E (G4)
i=0: G3: (One state group; nothing to check)
i=0: G4: (One state group; nothing to check)
i=0: G5: (One state group; nothing to check)
i=1: G1: A -> F (G5), D -> F (G5)
i=1: G2: B -> A (G1), C-> D (G1)
i=1: G3: (One state group, nothing to check)
i=1: G4: (One state group, nothing to check)
i=1: G5: (One state group, nothing to check)

Thus, no groups need to be partitioned further, and hence states within a group are-equivalent. Replace C by B and D by A to yield:



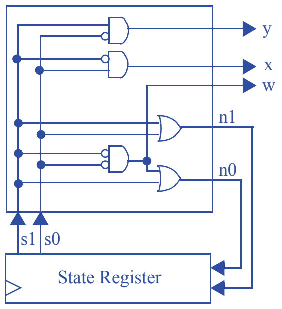
6.18) Compare the logic size (number of gate inputs) and the delay (number of gate-delays) of a straightforward 2-bit binary encoding of the FSM in Figure 6.91 using a 3-bit output encoding versus using a one-hot encoding.

2-bit binary encoding:

State encodings: S0: 00, S1: 01, S2: 10, S3: 11

Inputs		Outputs				
s1	s0	n1	n0	w	x	y
0	0	0	1	1	0	0
0	1	1	0	0	1	0
1	0	1	1	0	0	1
1	1	1	1	0	0	0

n1=s1+s0
n0=s1's0' + s1
w = s1's0'
x = s1's0
y=s1s0'



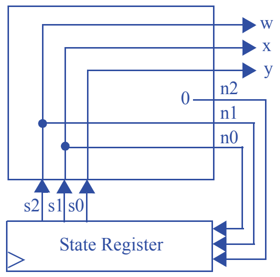
Logic size: 10 gate inputs
Delay: 2 gate delays

3-bit output encoding:

State encodings: S0: 100, S1: 010, S2: 001, S3: 000

Inputs			Outputs					
s2	s1	s0	n2	n1	n0	w	x	y
1	0	0	0	1	0	1	0	0
0	1	0	0	0	1	0	1	0
0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0

n2 = 0
n1 = s2
n0 = s1
w = s2
x = s1
y = s0



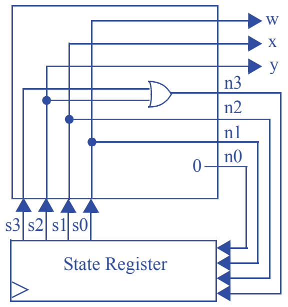
Logic size: 0 gate inputs
Delay: 0 gate delays

One-hot encoding:

State encodings: S0: 0001, S1: 0010, S2: 0100, S3: 1000

Inputs				Outputs						
s3	s2	s1	s0	n3	n2	n1	n0	w	x	y
0	0	0	1	0	0	1	0	1	0	0
0	0	1	0	0	1	0	0	0	1	0
0	1	0	0	1	0	0	0	0	0	1
1	0	0	0	1	0	0	0	0	0	0

n3 = s3 + s2
n2 = s1
n1 = s0
n0 = 0
w = s0
x = s1
y = s2



Logic size: 2 gate inputs
Delay: 1 gate delays

6.19) Compare the logic size (number of gate inputs) and the delay (number of gate-delays) of a minimal bitwidth state encoding versus an output encoding for the laser-based distance measurer FSM shown in Figure 5.26..

Minimal bit width encoding:
State encodings: S0: 000, S1: 001, S2: 010, S3: 011, S4: 100

Inputs					Outputs							
s2	s1	s0	B	S	n2	n1	n0	L	Dreg_clr	Dreg_ld	Dcnt_clr	Dcnt_cnt
0	0	0	x	x	0	0	1	0	1	0	0	0
0	0	1	0	x	0	0	1	0	0	0	1	0
0	0	1	1	x	0	1	0	0	0	0	1	0
0	1	0	x	x	0	1	1	1	0	0	0	0
0	1	1	x	0	0	1	1	0	0	0	0	1
0	1	1	x	1	1	0	0	0	0	0	0	1
1	0	0	x	x	0	0	1	0	0	1	0	0

n2 = s1s0S
n1 = s1's0B + s1s0' + s1s0S'
n0 = s1's0' + s1's0B' + s1s0' + s1s0S'
L = s1s0'
Dreg_clr = s2's1's0'
Dreg_ld = s2
Dcnt_clr = s1's0
Dcnt_cnt = s1s0

Logic size: 37 gate inputs
Delay: 2 gate delays

Output encoding:
State encodings: S0: 01000, S1: 00010, S2: 10000, S3: 00001, S4: 00100

Inputs							Outputs									
s4	s3	s2	s1	s0	B	S	n4	n3	n2	n1	n0	L	Dreg_clr	Dreg_ld	Dcnt_clr	Dcnt_cnt
0	1	0	0	0	x	x	0	0	0	1	0	0	1	0	0	0
0	0	0	1	0	0	x	0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	1	x	1	0	0	0	0	0	0	0	1	0
1	0	0	0	0	x	x	0	0	0	0	1	1	0	0	0	0
0	0	0	0	1	x	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	1	x	1	0	0	1	0	0	0	0	0	0	1
0	0	1	0	0	x	x	0	0	0	1	0	0	0	1	0	0

n4 = s1'B
n3 = 0
n2 = s0S
n1 = s3 + s1x' + s2
n0 = s4 + s0S'
L = s4
Dreg_clr = s3
Dreg_ld = s2
Dcnt_clr = s1
Dcnt_cnt = s0

Logic size: 13 gate inputs
Delay: 2 gate delays

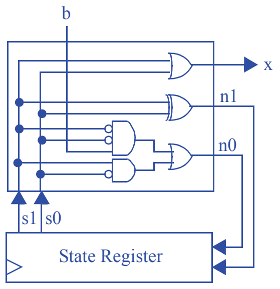
6.20) Compare the logic size (number of gate inputs) and the delay (number of gate-delays) of a minimum binary encoding, an output encoding (if it is possible; if not, indicate why not), and a one-hot encoding of the laser timer FSM in Figure 3.47..

Minimum binary encoding:

State encodings: S0: Off, On1: 01, On2: 10, On3: 11

Inputs			Outputs		
s1	s0	b	n1	n0	x
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	1	0	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	1

n1 = s1 xor s0
n0 = s1’s0’b + s1s0’
x = s1 + s0



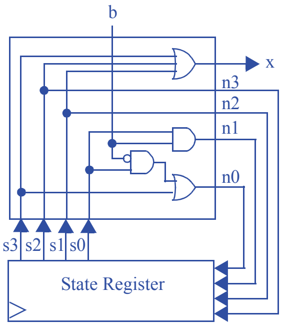
Logic size: 11 gate inputs
Delay: 2 gate delays

One-hot encoding:

State encodings: S0: 0001, S1: 0010, S2: 0100, S3: 1000

Inputs					Outputs				
s3	s2	s1	s0	b	n3	n2	n1	n0	x
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	1	0	0
0	0	1	0	x	0	1	0	0	1
0	1	0	0	x	1	0	0	0	1
1	0	0	0	x	0	0	0	1	1

n3 = s2
n2 = s1
n1 = s0b
n0 = s0b’ + s3
x = s3 + s2 + s1



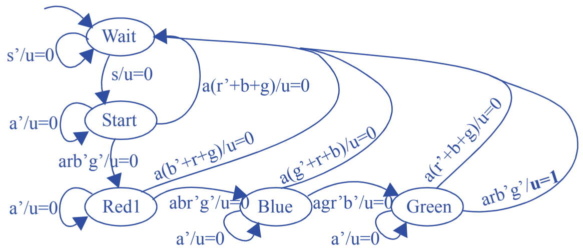
Logic size: 9 gate inputs
Delay: 2 gate delays

An output encoding is not possible since each state’s external outputs are not unique.

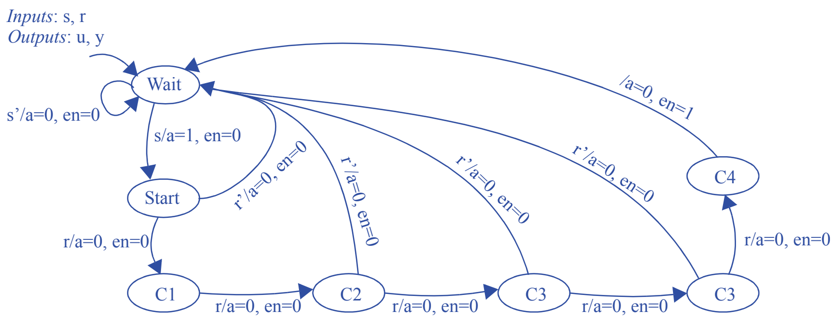
6.21) Convert the Moore FSM for the code detector circuit shown in Figure 3.58 to the nearest Mealy FSM equivalent.

.

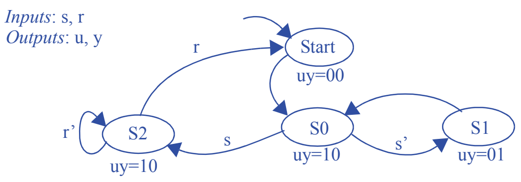
Inputs: s, r, g, b, a
Outputs: u



6.22) Convert the Moore FSM in Figure 6.92 to the nearest Mealy FSM equivalent.

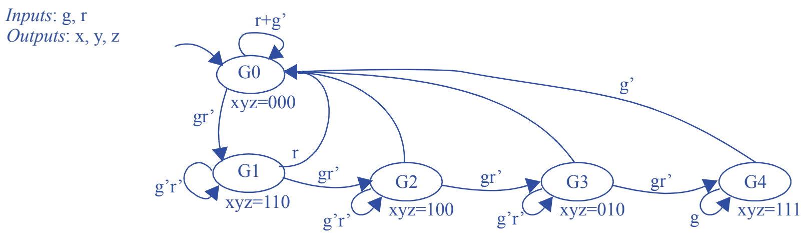


6.23) Convert the Mealy FSM in Figure 6.93 to the nearest Moore equivalent.



6.24) Convert the Mealy FSM in Figure 6.94 to the nearest Moore equivalent.

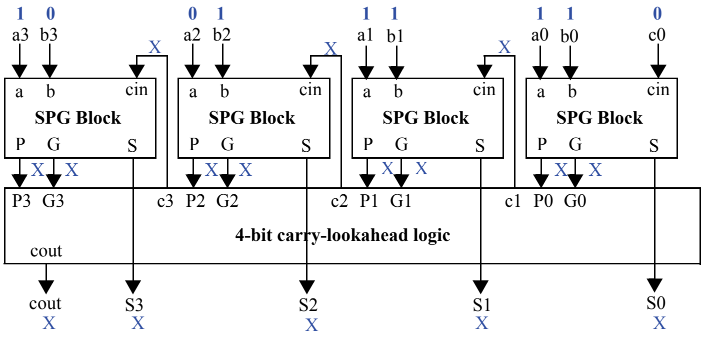
.



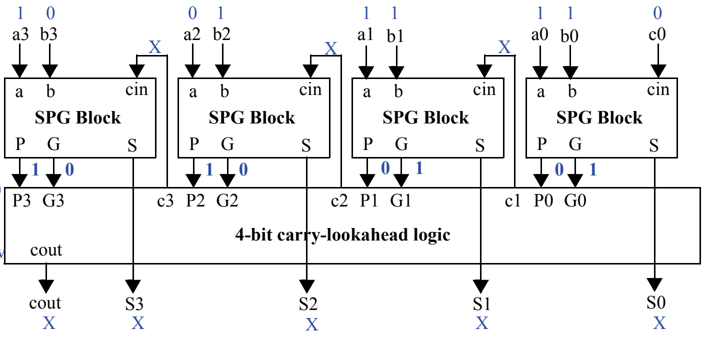
SECTION 6.4: DATAPATH COMPONENT TRADEOFFS

6.25) Trace the execution of the 4-bit carry-lookahead adder shown in Figure 6.57 when `a` = 11 (eleven) and `b` = 7. Show all the input and output values of the SPG blocks and of the carry-lookahead block initially and after each relevant number of gate delays..

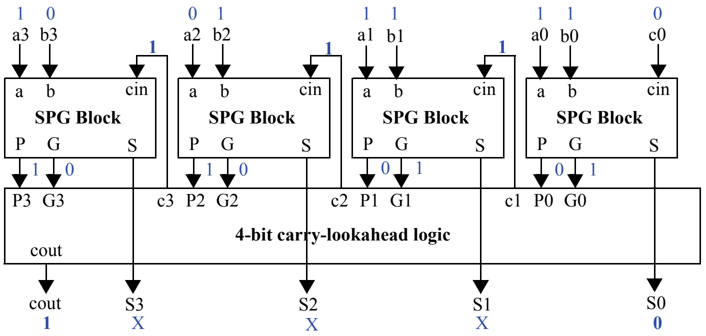
Initial values



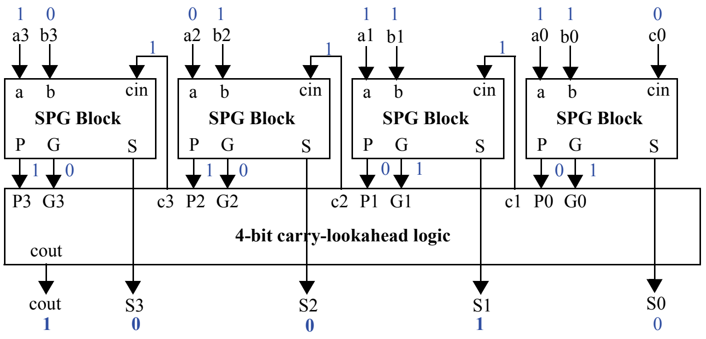
Generate/Propagate bits computed after 1 gate delay (S0 will be computed after one more gate delay; we won't show another diagram/step just for this one bit)



Carry-lookahead logic outputs computed after 2 more gate delays

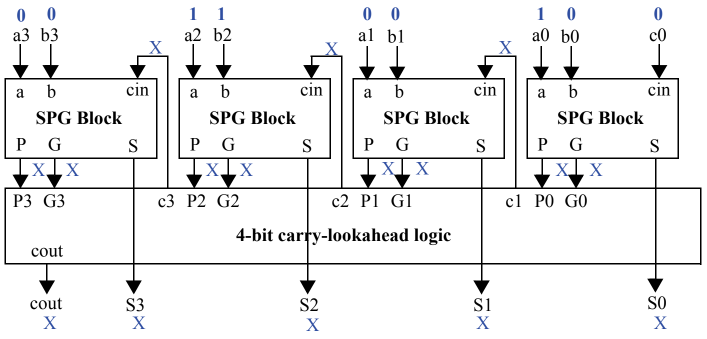


Sums computed after 1 more gate delay

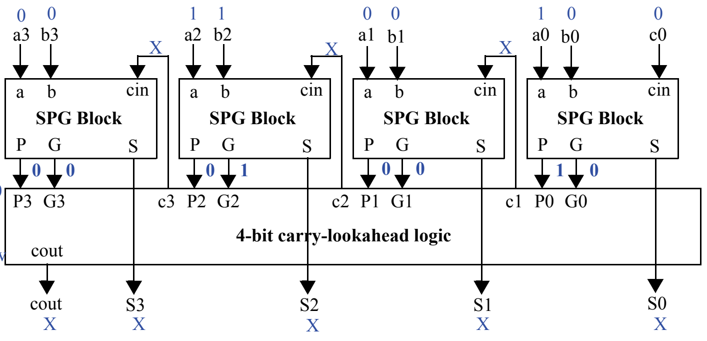


6.26) Trace the execution of the 4-bit carry-lookahead adder shown in Figure 6.57 when a = 5 and b = 4. Show all the input and output values of the SPG blocks and of the carry-lookahead block initially and after each relevant number of gate delays.

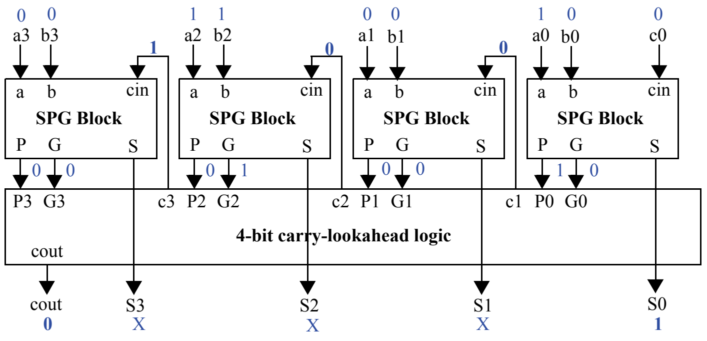
Initial values


0 0 a3 b3 X 1 1 a2 b2 X 0 0 a1 b1 X 1 0 a0 b0 0 c0
a b cin SPG Block P G S (×4)
X X X X X X X X
P3 G3 c3 P2 G2 c2 P1 G1 c1 P0 G0
4-bit carry-lookahead logic
cout
cout X S3 X S2 X S1 X S0 X

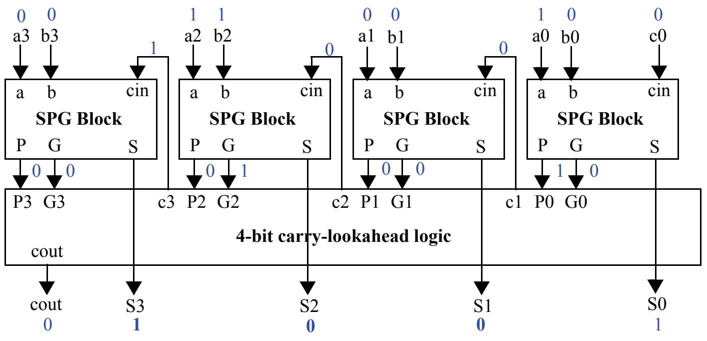
Generate/Propagate bits computed after 1 gate delay (S0 will be computed after one more gate delay; we won't show another diagram/step just for this one bit)


0 0 a3 b3 X 1 1 a2 b2 X 0 0 a1 b1 X 1 0 a0 b0 0 c0
a b cin SPG Block P G S (×4)
0 0 0 1 0 0 1 0
P3 G3 c3 P2 G2 c2 P1 G1 c1 P0 G0
4-bit carry-lookahead logic
cout
cout X S3 X S2 X S1 X S0 X

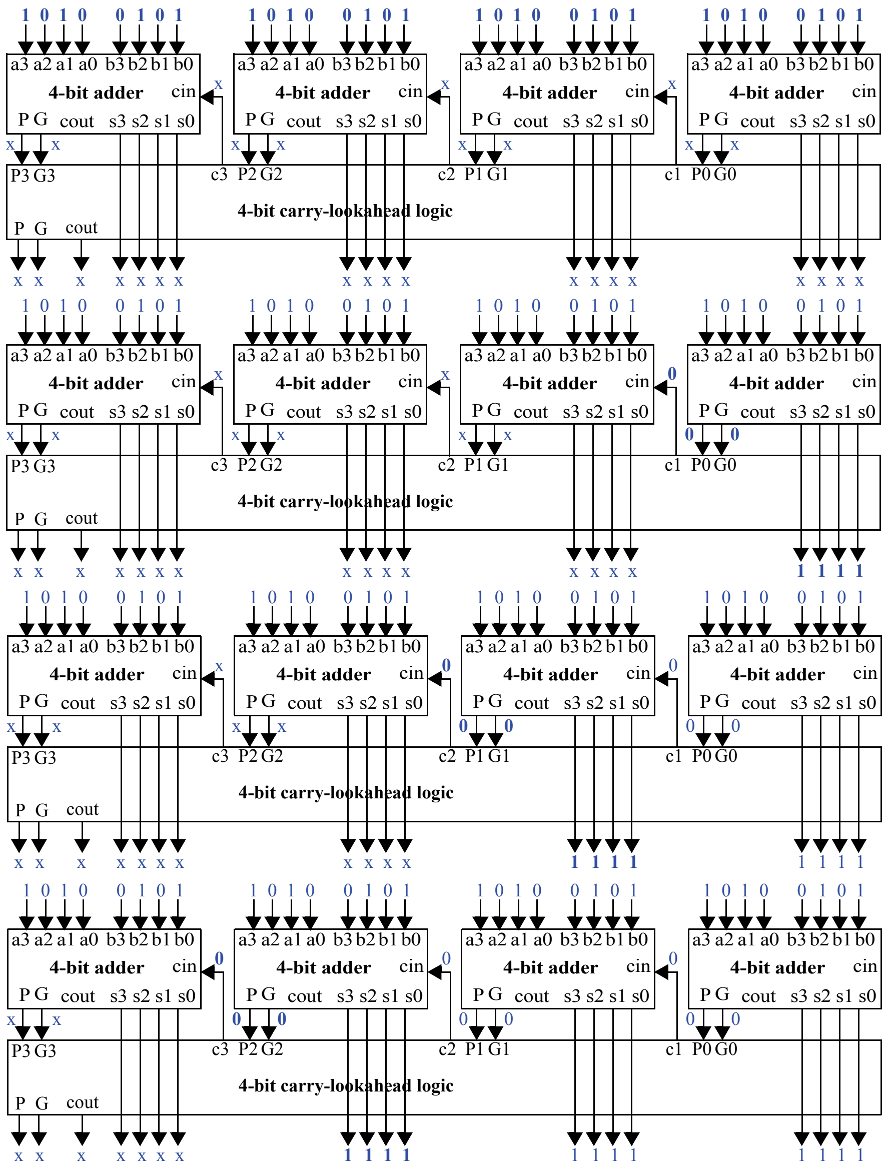
Carry-lookahead logic outputs computed after 2 more gate delays

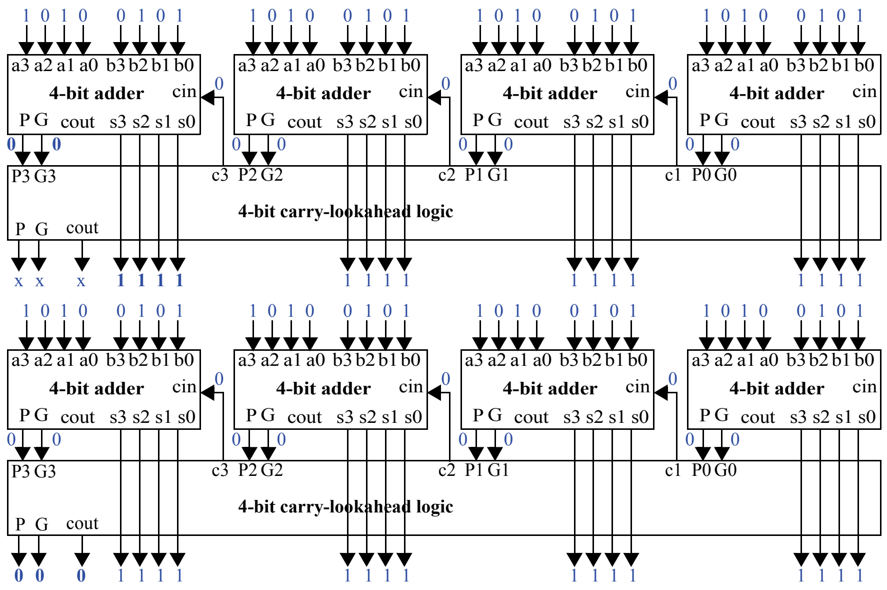

0 0 a3 b3 1 1 1 a2 b2 0 0 0 a1 b1 0 1 0 a0 b0 0 c0
a b cin SPG Block P G S (×4)
0 0 0 1 0 0 1 0
P3 G3 c3 P2 G2 c2 P1 G1 c1 P0 G0
4-bit carry-lookahead logic
cout
cout 0 S3 X S2 X S1 X S0 1

Sums computed after 1 more gate delay


0 0 a3 b3 1 1 1 a2 b2 0 0 0 a1 b1 0 1 0 a0 b0 0 c0
a b cin SPG Block P G S (×4)
0 0 0 1 0 0 1 0
P3 G3 c3 P2 G2 c2 P1 G1 c1 P0 G0
4-bit carry-lookahead logic
cout
cout 0 S3 1 S2 0 S1 0 S0 1

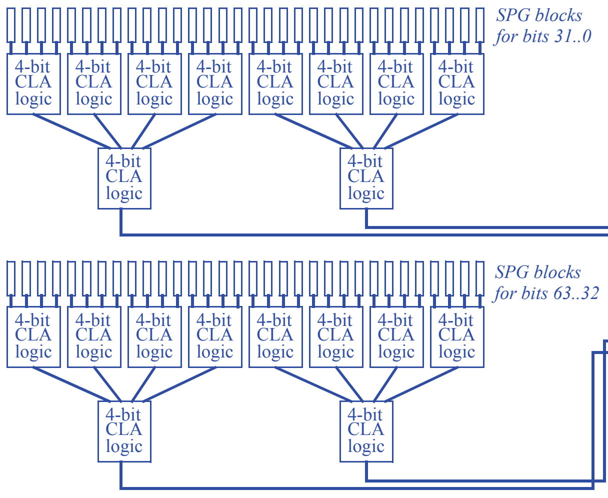
6.27) Trace the execution of the 16-bit carry-lookahead adder built from 4-bit adders as shown in Figure 6.60 when a = 43690 and b = 21845. Do not trace internal behavior of the individual 4-bit carry-lookahead adders..





6.28) (a) Design a 64-bit hierarchical carry-lookahead adder using 4-bit carry-lookahead adders. (b) What is the total delay through the 64-bit adder? (c) What is the speedup of the carry-lookahead adder compared to a 64-bit carry-ripple adder; compute speedup as (slower time)/(faster time).

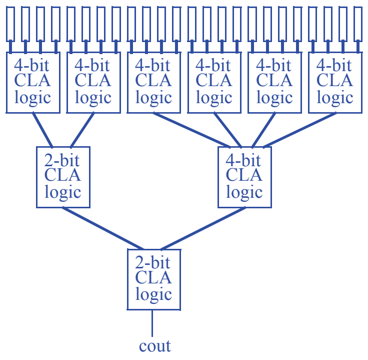
(a)



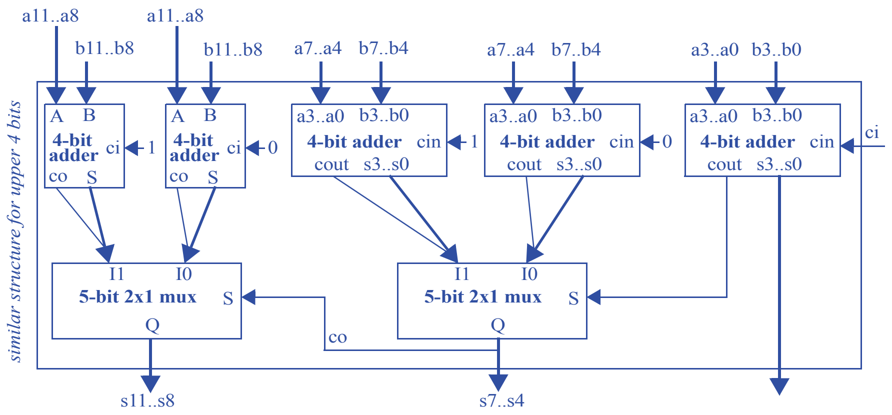
(b) The hierarchical carry-lookahead adder depicted above requires 8 gate delays (2 for the SPG blocks, and 6 for the three levels of CLA logic).

(c) Compared to a carry-ripple adder (composed of a chain of full-adders), the hierarchical carry-lookahead adder speedup is 128 gate delays/8 gate delays = 16 times faster.

6.29) Design a 24-bit hierarchical carry-lookahead adder using 4-bit carry-lookahead adders.



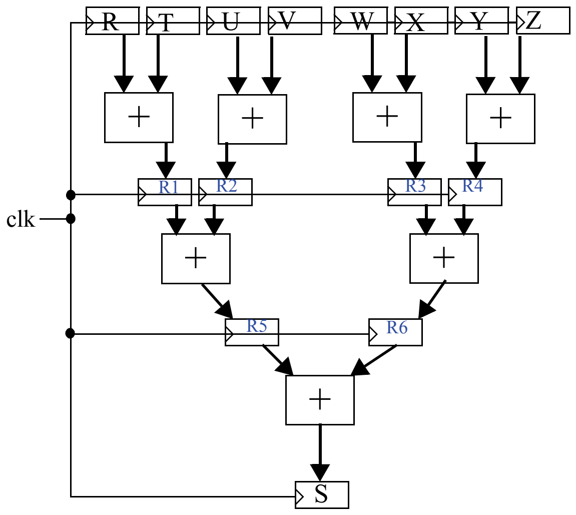
6.30) Design a 16-bit carry-select adder using 4-bit ripple carry adders.



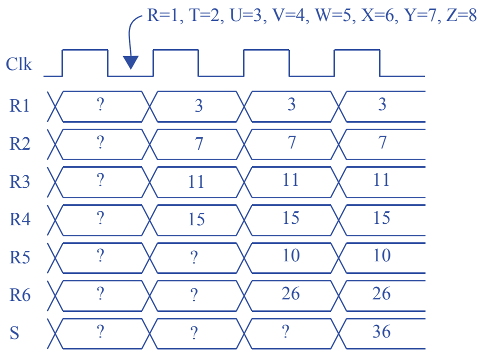
Section 6.5: RTL Design Optimizations and Tradeoffs

6.31) The adder tree shown in Figure 6.2 is used to compute the sum of eight inputs on every clock cycle, where the sum is: `S = R + T + U + V + W + X + Y + Z`. (a) Design a pipelined version of the adder tree to maximize the speed at which we can operate our clock input *clk*. (b) Create a timing diagram of the pipelined tree circuit showing the values of pipeline registers and the output register for the following input valuesL R=1, T=2, U=3, V=4, W=5, X=6, Y=7, and Z=8. (c) If the delay of an adder is 3 ns, compare the fastest clock frequency of the original circuit versus the pipelined circuit. (d) Again assuming 3 ns adders, compare the fastest latency and throughput values for the original circuit versus the pipelined circuit.

(a)



(b)



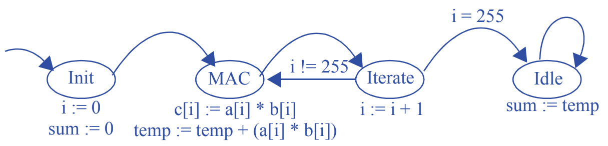
(c) The non-pipelined adder tree can be operated with a clock period of 9 ns while the pipelined adder tree can be operated with a clock period of 3 ns. The frequencies are 1/9ns = 1.11E8 or 111 MHz, versus 1/3ns = 3.33E8 or 333 MHz.

(d) Assuming the delay of an adder is 3 ns, the latency and throughput of the original circuit are 9 ns and 9 ns, and of the pipelined circuit are 9 ns and 3 ns.

6.32) (a) Convert the following C-like code to a high-level state machine. Ignore overflow. (b) Use the RTL design process shown in Table 5.1 to convert the HLSM for the C code to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only. (c) Redesign the datapath to allow for concurrency in which four multiplications and two additions can be performed concurrently. Assume memory ports can can be introduced as needed. (d) Assuming a multiplier delay is 4 ns and an adder delay is 2 ns, list the fastest clock period, latency, and throughput for the original design and for the more concurrent design, assuming the critical path is in the datapath. (e) Introduce more multipliers or adders and pipeline registers as needed to further improve the speed of the design, and compare the clock period, throughput, and latency with the previous two designs.

(a)

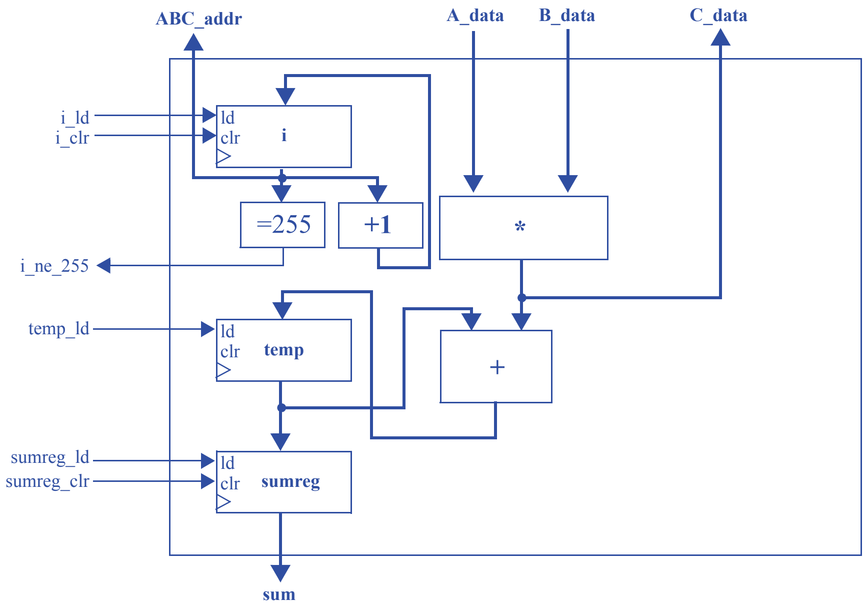
Inputs: byte a[256], byte b[256]
Outputs: byte sum, byte c[256]
Local Storage: byte temp, byte i



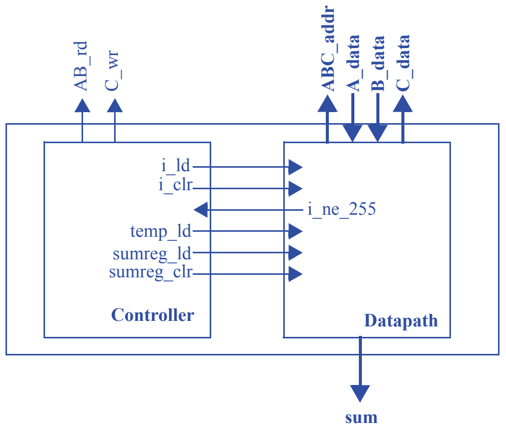
(b)

Step 1 - Capture a high-level state machine - (completed above)

Step 2 - Create a datapath

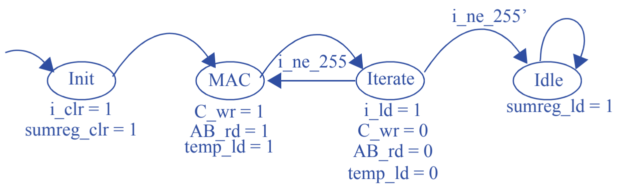


Step 3 - Connect the datapath to a controller

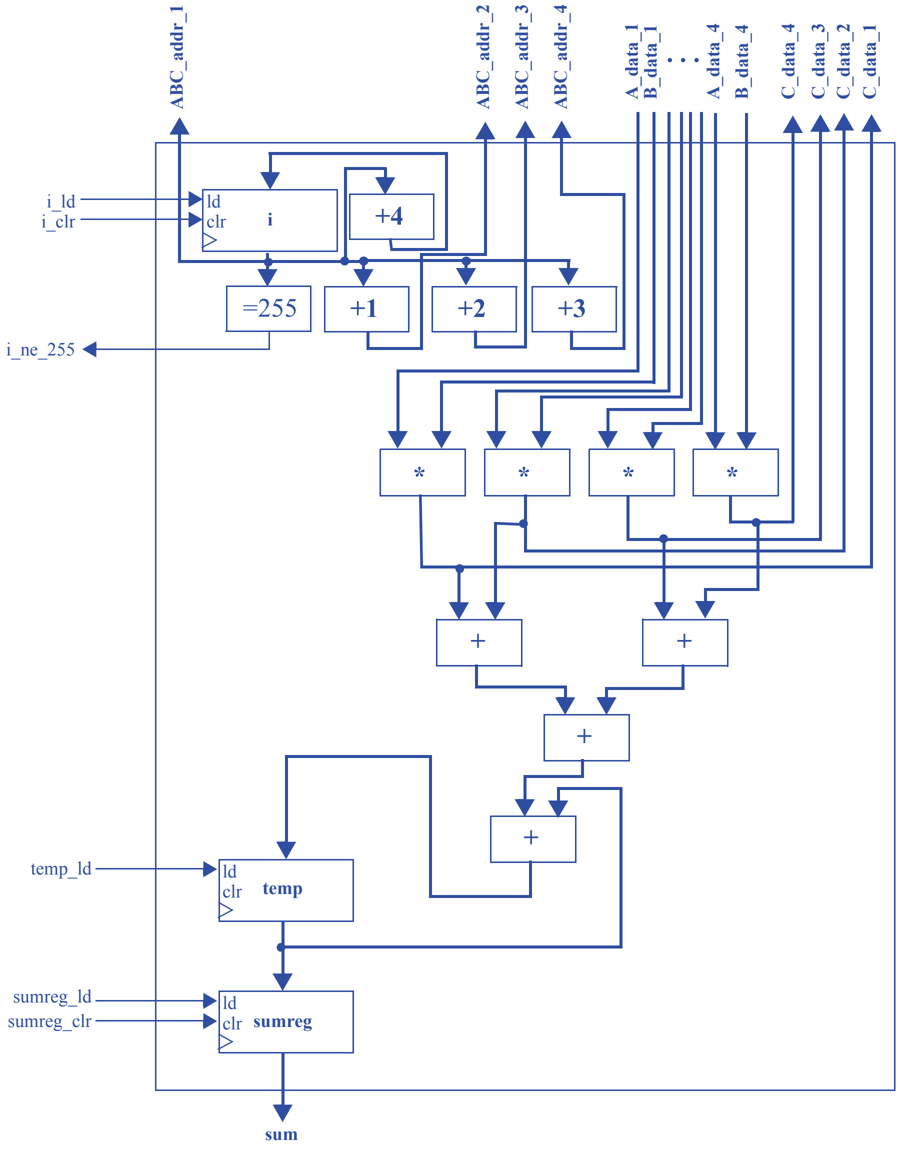


Step 4 - Derive the controller's FSM

Inputs: i_ne_255
Outputs: i_ld, i_clr, temp_ld, sumreg_ld, sumreg_clr, AB_rd, C_wr



(c)


ABC_addr_1
ABC_addr_2
ABC_addr_3
ABC_addr_4
A_data_1
B_data_1
. . .
A_data_4
B_data_4
C_data_4
C_data_3
C_data_2
C_data_1
i_ld
i_clr
ld
clr
i
+4
=255
+1
+2
+3
i_ne_255
*
*
*
*
+
+
+
+
temp_ld
ld
clr
temp
sumreg_ld
sumreg_clr
ld
clr
sumreg
sum

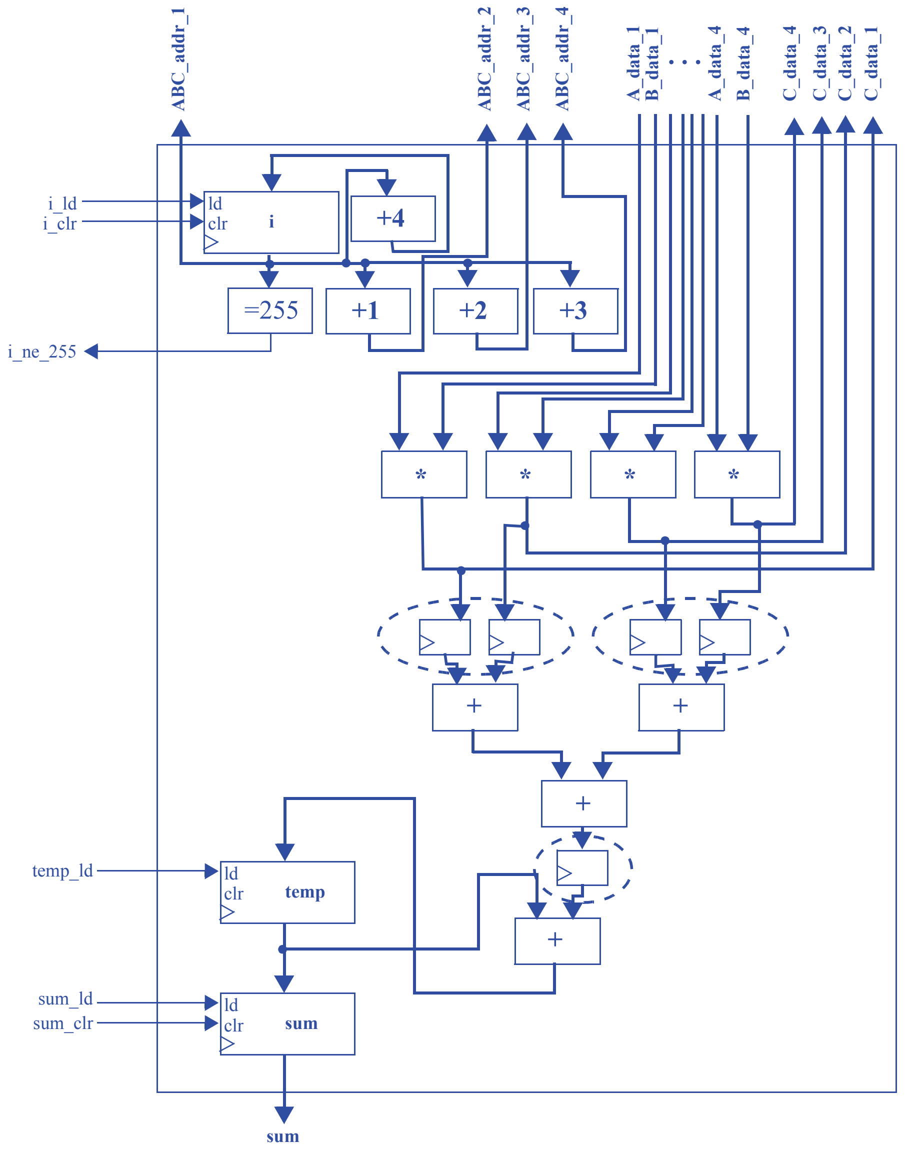
(d)
Original Design: 4ns + 2ns = 6ns critical path, so 6ns clock period. Latency is 6 ns, and throughput is 1 multiply-accumulates per 6ns -- 166.6 million multiply-accumulates per second.

Concurrent Design: 4ns + 2ns + 2ns + 2ns = 10ns critical path, so 10ns clock period. Latency is also 10ns, and throughput is 4 multiply-accumulates per 10ns -- 400 million multiply-accumulates per second.

(e) We have a range of area-performance tradeoffs available to us. For instance, we could theoretically include 128 multipliers and a full adder tree (assuming we can either reorganize the memory or create a 256 port memory). With pipeline registering, we could have a 4ns clock period. Our latency would be 5 clock cycles, or 20ns.

We would, however, complete the entire operation in ‘one go’, for a throughput of 256 MACs in 20ns = 12.80 billion MACs / second.

A more likely scenario, though, would be to pipeline the datapath in (c):

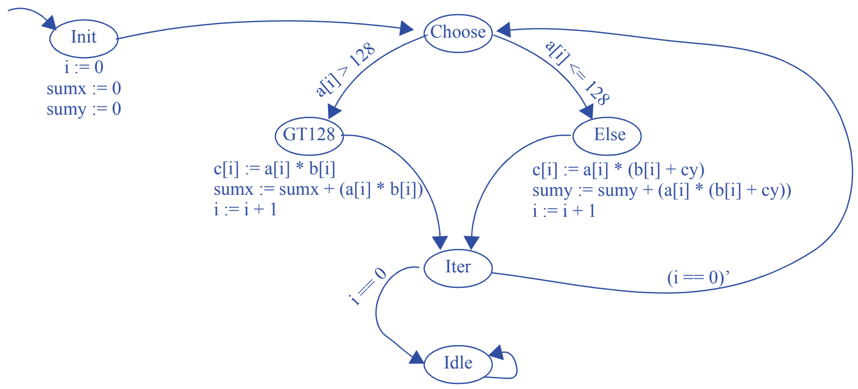


With the circuit above, we would see a clock period of 4ns, a latency of (4ns + 4ns + 4ns) = 12ns, and a throughput of 4 MACs per cycle, or 1 billion MACs / second.

6.33) (a) Convert the following C-like code to a high-level state machine. Ignore overflow. (b) Use the RTL design process shown in Table 5.1 to convert the high-level state machine for the C code to a controller and a datapath. Design the datapath to structure, but design the controller to the point of an FSM only. (c) Redesign your datapath to allow for concurrency in which three comparisons, three additions, and three multiplications can be performed concurrently.

(a)

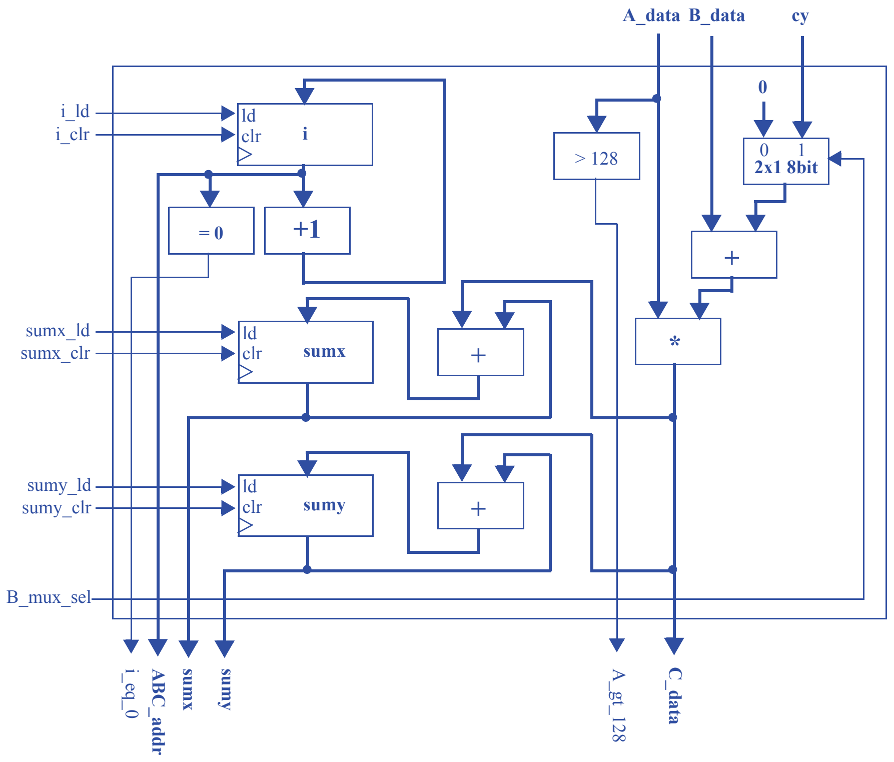
Inputs: byte a[256], byte b[256], byte cy
Outputs: byte sumx, byte sumy, byte c[256]
Local Storage: byte i



(b)

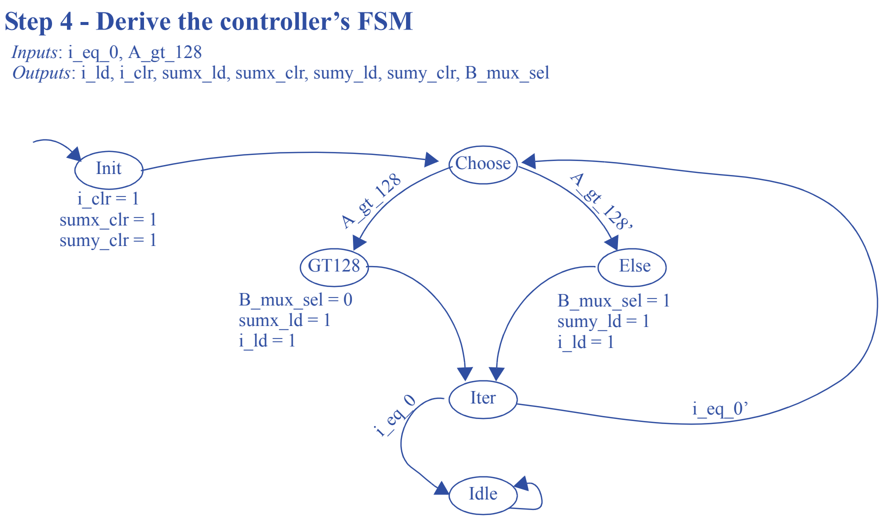
Step 1 - Capture a high-level state machine - (completed above)

Step 2 - Create a datapath

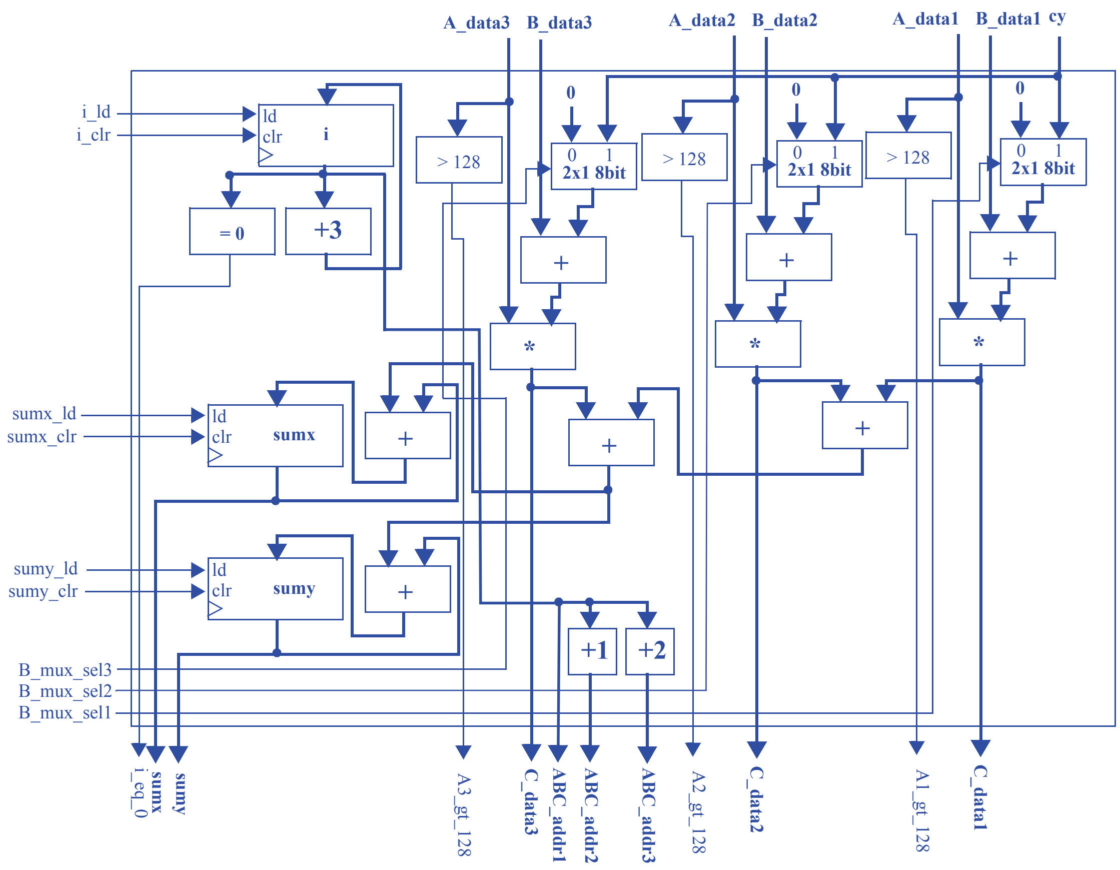


Step 3 - Connect the datapath to a controller

Omitted. Datapath and controller are connected in the same manner as 6.32. The controller's signals to the datapath are i_ld, i_clr, sumx_ld, sumx_clr, sumy_ld, sumy_clr, and B_mux_sel. The datapath's signals to the controller are i_eq_0 and A_gt_128.


Step 4 - Derive the controller's FSM
Inputs: i_eq_0, A_gt_128
Outputs: i_ld, i_clr, sumx_ld, sumx_clr, sumy_ld, sumy_clr, B_mux_sel
Init
i_clr = 1
sumx_clr = 1
sumy_clr = 1
Choose
A_gt_128
A_gt_128'
GT128
B_mux_sel = 0
sumx_ld = 1
i_ld = 1
Else
B_mux_sel = 1
sumy_ld = 1
i_ld = 1
Iter
i_eq_0
i_eq_0'
Idle

(c)


A_data3
B_data3
A_data2
B_data2
A_data1
B_data1
cy
i_ld
i_clr
ld
clr
i
= 0
+3
> 128
0
0 1
2x1 8bit
+
*
sumx_ld
sumx_clr
sumx
sumy_ld
sumy_clr
sumy
+1
+2
B_mux_sel3
B_mux_sel2
B_mux_sel1
i_eq_0
sumx
sumy
A3_gt_128
C_data3
ABC_addr1
ABC_addr2
ABC_addr3
A2_gt_128
C_data2
A1_gt_128
C_data1

6.34) Redesign the datapath and controller designed in Exercise 6.33 by allowing up to nine concurrent additions and inserting pipeline registers, updating the controller as necessary. Assuming a comparator has a delay of 4 ns, an adder has a delay of 3 ns, and a multiplier has a delay of 20 ns, how long will the circuit take to finish its computation?

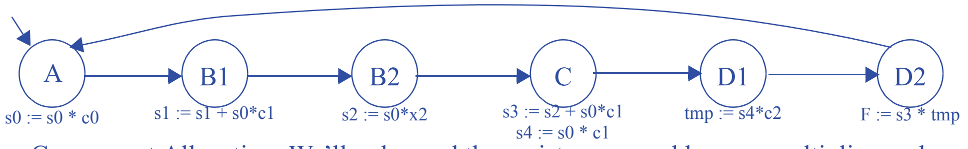
Note that if we choose the maximum number of operations (9), then we will have a few units at the end adding erroneous data, and so the results must be gated off on the last cycle. If we choose 8 operations, we have a similar problem -- we end up adding an element from address 0. While entirely possible, these are likely not the best design choices. Thus, we will use the maximum number of concurrent additions which allow an easy design (i.e. the remainder of 255 divided by this number is zero). Thus, we will use 5 concurrent additions in this solution.

The solution is very similar to 6.33(c), but with 5 separate (mux, comparator, adder, multiplier) units instead of 3. The most obvious pipeline register insertion would be before and after each multiplier, to give us a clock period of 20 ns.

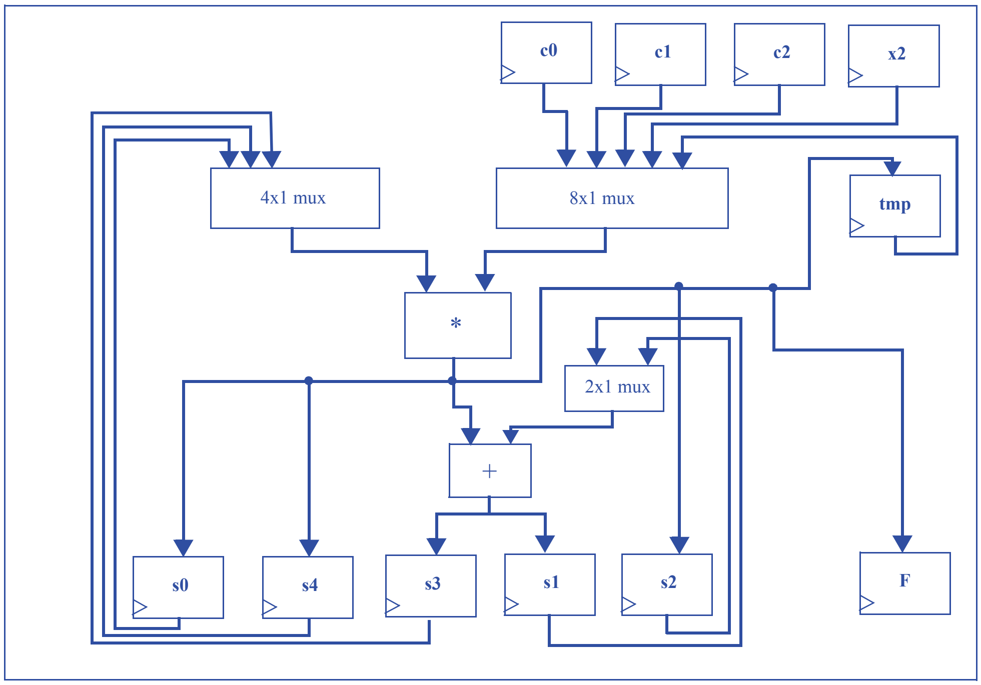
6.35) Given the HLSM in Figure 6.98, create two different designs: one optimized for minimum circuit speed and the other optimized for minimum circuit size. Be sure to clearly indicate the component allocation, operator binding, and operator scheduling used to design the two circuits.

Design 1: Optimize For Size

New Schedule: (an extra register is definitely smaller than an extra multiplier)



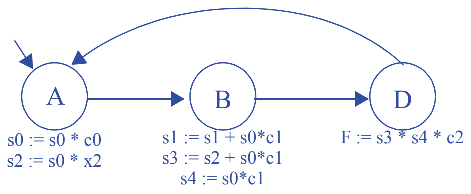
Component Allocation: We'll only need the registers, one adder, one multiplier, and three muxes (one with two inputs, one with at least three inputs and one with at least 5 inputs)



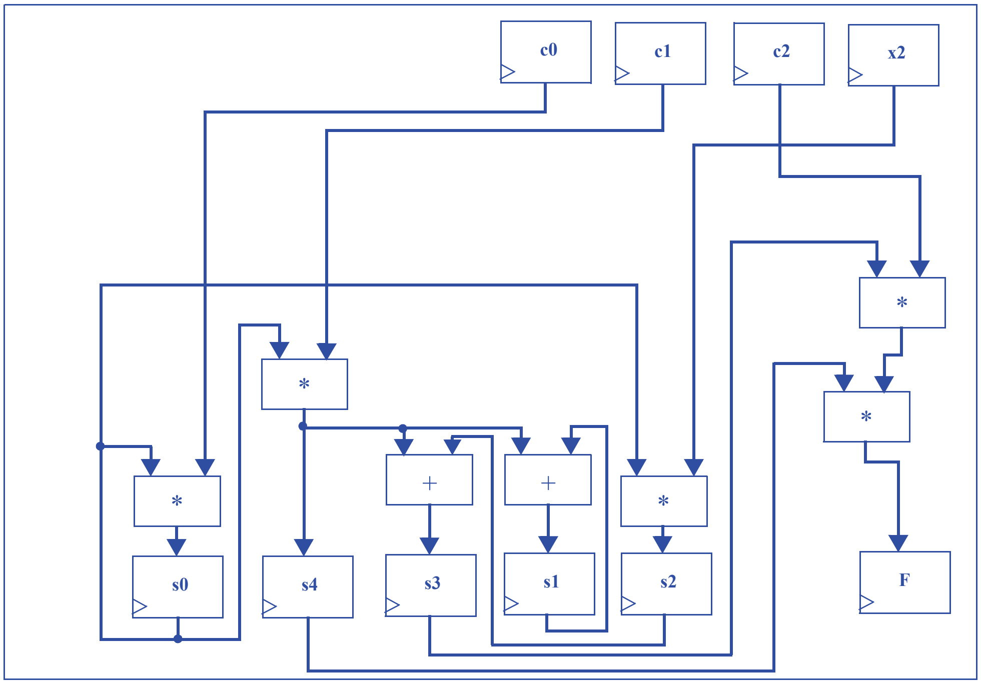
Note: control signals are omitted for simplicity

Design 2: Optimize For Speed

New Schedule:



Component Allocation: We can use two multipliers if we are OK with using muxes. However, for the best performance possible, we will use dedicated multipliers (albeit at a huge cost in area). We will also use dedicated adders.



Note: control signals are omitted for simplicity

SECTION 6.6: MORE ON OPTIMIZATIONS AND TRADEOFFS

6.36) Trace through the execution of the binary search algorithm when searching for the number 86 in the following sorted list of 15 numbers: 1, 10, 25, 62, 74, 75, 80, 84, 85, 86, 87, 100, 106, 111, 121. How many comparisons were required to find the number using the binary search and how many comparisons would have been required using a linear search?

Assume that the 15 numbers are indexed from 0 to 14.

1. We compare the middle number (number[7]: 84) with 86 and determine that 86 might be between number[8] and number[14], inclusive

2. We compare the middle number (number[11]: 100) to 86 and determine that 86 might be between number[8] and number[10], inclusive

3. We compare the middle number (number[9]: 86) to 86 and conclude the search

A binary search requires 3 comparisons to find number 86, while a linear search (assuming we start from number[0]) requires 9 comparisons to find number 86.

6.37) Trace through the execution of the binary search algorithm when searching for the number 99 in the following list of 15 numbers: 1, 10, 25, 62, 74, 75, 80, 84, 85, 87, 99, 100, 106, 111, 121. How many comparisons were required to look for the number using the binary search and how many comparisons are required using a linear search?

Assume that the 15 numbers are indexed from 0 to 14.

1. We compare the middle number (number[7]: 84) with 99 and determine that 99 might be between number[8] and number[14], inclusive

2. We compare the middle number (number[11]: 100) to 99 and determine that 99 might be between number[8] and number[10], inclusive

3. We compare the middle number (number[9]: 86 to 99) and determine that 99 might be number[10].

4. We compare number[10] (87) and conclude the search (99 was not found).

Using a binary search required 4 comparisons, while a linear search would require 12 comparisons.

6.38) Trace through the execution of the binary search algorithm when searching for the number 121 in the list of numbers from the previous example. How many comparisons were required to find the number using the binary search and how many comparisons are required using a linear search?

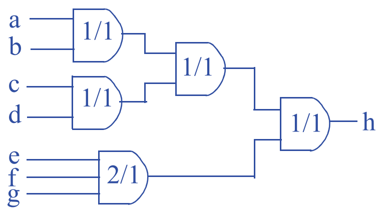
> A binary search requires 4 or 5 comparisons (depending on how the middle number is chosen for even-sized ranges) to find 121, while a linear search takes 14 comparisons to find 121.

6.39) Using the list of 15 numbers from Exercise 6.37, how many numbers can be found faster using a linear search algorithm compared with the binary search algorithm?

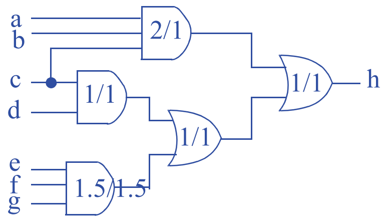
> Depending on how the middle number is chosen for even-sized ranges, we can find the first 2 or first 3 numbers in the list faster using linear search instead of binary search.

Section : Power Optimization

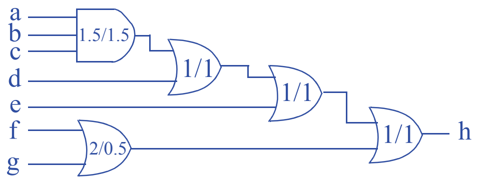
6.40) Given the logic gate library in Figure 6.99, optimize the circuit in Figure 6.100 by reducing power consumption without increasing the circuit's delay..



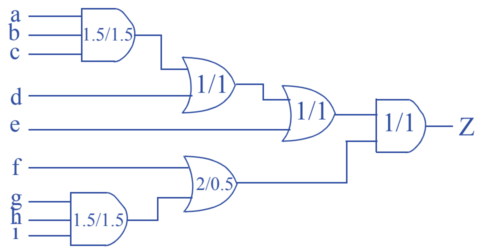
.6.41) Given the logic gates shown in Figure 6.99, optimize the circuit in Figure 6.101 by reducing power consumption without increasing the circuit's delay.



6.42) Given the logic gates shown in Figure 6.99, optimize the circuit in Figure 6.102 by reducing power consumption without increasing the circuit's delay..



6.43) Given the logic gates shown in Figure 6.99, optimize the circuit in Figure 6.103by reducing power consumption without increasing the circuit's delay.


a
b
c
d
e
f
g
h
i
1.5/1.5
1/1
1/1
1/1
2/0.5
1.5/1.5
Z