

Combinational logic design case studies

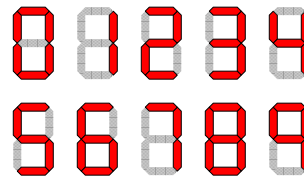
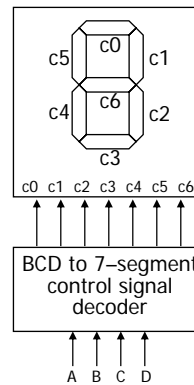
- General design procedure
- Case studies
 - BCD to 7-segment display controller
 - calendar subsystem
- Arithmetic circuits
 - integer representations
 - addition/subtraction
 - arithmetic/logic units

General design procedure for combinational logic

- 1. Understand the problem
 - what is the circuit supposed to do?
 - write down inputs (data, control) and outputs
 - draw block diagram or other picture
- 2. Formulate the problem using a suitable design representation
 - truth table or waveform diagram are typical
 - may require encoding of symbolic inputs and outputs
- 3. Choose implementation target
 - ROM, PAL, PLA
 - mux, decoder and OR-gate
 - discrete gates
- 4. Follow implementation procedure
 - K-maps for two-level, multi-level
 - design tools and hardware description language (e.g., Verilog)

BCD to 7-segment display controller

- Understanding the problem
 - input is a 4 bit bcd digit (A, B, C, D)
 - output is the control signals for the display (7 outputs C0 – C6)
- Block diagram



Formalize the problem

- Truth table
 - show don't cares
- Choose implementation target
 - if ROM, we are done
 - don't cares imply PAL/PLA may be attractive
- Follow implementation procedure
 - minimization using K-maps

A	B	C	D	C0	C1	C2	C3	C4	C5	C6
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	-	-	-	-	-	-	-	-
1	1	-	-	-	-	-	-	-	-	-

Implementation as minimized sum-of-products

- 15 unique product terms when minimized individually

$C0 = A + B D + C + B' D'$
 $C1 = C' D' + C D + B'$
 $C2 = B + C' + D$
 $C3 = B' D' + C D' + B C' D + B' C$
 $C4 = B' D' + C D'$
 $C5 = A + C' D' + B D' + B C'$
 $C6 = A + C D' + B C' + B' C$

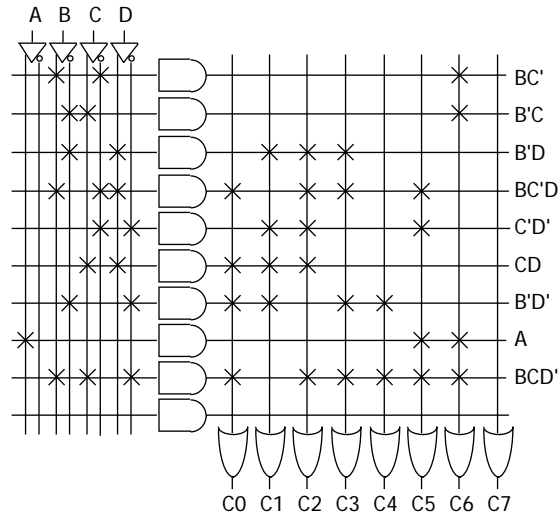
Implementation as minimized S-o-P (cont'd)

- Can do better
 - 9 unique product terms (instead of 15)
 - share terms among outputs
 - each output not necessarily in minimized form

$C0 = A + B D + C + B' D'$
 $C1 = C' D' + C D + B'$
 $C2 = B + C' + D$
 $C3 = B' D' + C D' + B C' D + B' C$
 $C4 = B' D' + C D'$
 $C5 = A + C' D' + B D' + B C'$
 $C6 = A + C D' + B C' + B' C$

$C0 = B C' D + C D + B' D' + B C D' + A$
 $C1 = B' D + C' D' + C D + B' D'$
 $C2 = B' D + B C' D + C' D' + C D + B C D'$
 $C3 = B C' D + B' D + B' D' + B C D'$
 $C4 = B' D' + B C D'$
 $C5 = B C' D + C' D' + A + B C D'$
 $C6 = B' C + B C' + B C D' + A$

PLA implementation



PAL implementation vs. Discrete gate implementation

- Limit of 4 product terms per output
 - decomposition of functions with larger number of terms
 - do not share terms in PAL anyway (although there are some with some shared terms)

$$C2 = B + C + D$$

$$C2 = B'D + B C'D + C'D' + CD + B C D'$$

$$C2 = B'D + B C'D + C'D' + W$$

$$W = C D + B C D'$$
← need another input and another output
- decompose into multi-level logic (hopefully with CAD support)
 - find common sub-expressions among functions

$$C0 = C3 + A' B X' + A D Y$$

$$C1 = Y + A' C5' + C' D' C6$$

$$C2 = C5 + A' B' D + A' C D$$

$$C3 = C4 + B D C5 + A' B' X'$$

$$C4 = D' Y + A' C D'$$

$$C5 = C' C4 + A Y + A' B X$$

$$C6 = A C4 + C C5 + C4' C5 + A' B' C$$

$$X = C' + D'$$

$$Y = B' C'$$

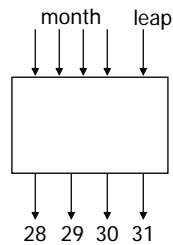
Calendar subsystem

- Determine number of days in a month (to control watch display)
 - used in controlling the display of a wrist-watch LCD screen
 - inputs: month, leap year flag
 - outputs: number of days
- Use software implementation to help understand the problem

```
integer number_of_days ( month, leap_year_flag) {
    switch (month) {
        case 1: return (31);
        case 2: if (leap_year_flag == 1)
                then return (29)
                else return (28);
        case 3: return (31);
        case 4: return (30);
        case 5: return (31);
        case 6: return (30);
        case 7: return (31);
        case 8: return (31);
        case 9: return (30);
        case 10: return (31);
        case 11: return (30);
        case 12: return (31);
        default: return (0);
    }
}
```

Formalize the problem

- Encoding:
 - binary number for month: 4 bits
 - 4 wires for 28, 29, 30, and 31
 - one-hot – only one true at any time
- Block diagram:



month	leap	28	29	30	31
0000	-	-	-	-	-
0001	-	0	0	0	1
0010	0	1	0	0	0
0011	1	0	1	0	0
0100	-	0	0	0	1
0101	-	0	0	1	0
0110	-	0	0	0	1
0111	-	0	0	0	1
1000	-	0	0	0	1
1001	-	0	0	1	0
1010	-	0	0	0	1
1011	-	0	0	1	0
1100	-	0	0	0	1
1101	-	-	-	-	-
111-	-	-	-	-	-

Choose implementation target and perform mapping

- Discrete gates

- 28 = $m8' m4' m2 m1' \text{ leap}'$

- 29 = $m8' m4' m2 m1' \text{ leap}$

- 30 = $m8' m4 m1' + m8 m1$

- 31 = $m8' m1 + m8 m1'$

- Can translate to S-o-P or P-o-S

month	leap	28	29	30	31
0000	-	-	-	-	-
0001	-	0	0	0	1
0010	0	1	0	0	0
0011	1	0	1	0	0
0011	-	0	0	0	1
0100	-	0	0	1	0
0101	-	0	0	0	1
0110	-	0	0	1	0
0111	-	0	0	0	1
1000	-	0	0	0	1
1001	-	0	0	1	0
1010	-	0	0	0	1
1011	-	0	0	1	0
1100	-	0	0	0	1
1101	-	-	-	-	-
111-	-	-	-	-	-

Leap year flag

- Determine value of leap year flag given the year

- For years after 1582 (Gregorian calendar reformation),
 - leap years are all the years divisible by 4,
 - except that years divisible by 100 are not leap years,
 - but years divisible by 400 are leap years.

- Encoding the year:

- binary – easy for divisible by 4,
but difficult for 100 and 400 (not powers of 2)
 - BCD – easy for 100,
but more difficult for 4, what about 400?

- Parts:

- construct a circuit that determines if the year is divisible by 4
 - construct a circuit that determines if the year is divisible by 100
 - construct a circuit that determines if the year is divisible by 400
 - combine the results of the previous three steps to yield the leap year flag

Activity: divisible-by-4 circuit

Activity: divisible-by-4 circuit

- BCD coded year
 - YM8 YM4 YM2 YM1 , YH8 YH4 YH2 YH1 , YT8 YT4 YT2 YT1 , YO8 YO4 YO2 YO1
- Only need to look at low-order two digits of the year
all years ending in 00, 04, 08, 12, 16, 20, etc. are divisible by 4
 - if tens digit is even, then divisible by 4 if ones digit is 0, 4, or 8
 - if tens digit is odd, then divisible by 4 if the ones digit is 2 or 6.

- Translates into the following Boolean expression
(where YT1 is the year's tens digit low-order bit,
YO8 is the high-order bit of year's ones digit, etc.):

$$YT1' (YO8' YO4' YO2' YO1' + YO8' YO4 YO2' YO1' + YO8 YO4' YO2' YO1') \\ + YT1 (YO8' YO4' YO2 YO1' + YO8' YO4 YO2 YO1')$$

- Digits with values of 10 to 15 will never occur, simplify further to yield:

$$YT1' YO2' YO1' + YT1 YO2 YO1'$$

Divisible-by-100 and divisible-by-400 circuits

- Divisible-by-100 just requires checking that all bits of two low-order digits are all 0:

$$YT8' YT4' YT2' YT1' \cdot YO8' YO4' YO2' YO1'$$

- Divisible-by-400 combines the divisible-by-4 (applied to the thousands and hundreds digits) and divisible-by-100 circuits

$$(YM1' YH2' YH1' + YM1 YH2 YH1')$$

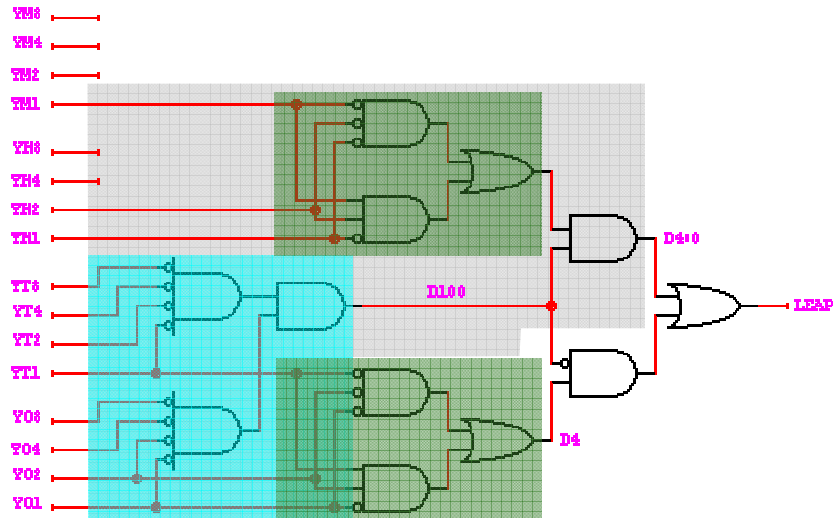
$$\cdot (YT8' YT4' YT2' YT1' \cdot YO8' YO4' YO2' YO1')$$

Combining to determine leap year flag

- Label results of previous three circuits: D4, D100, and D400

$$\text{leap_year_flag} = D4 \cdot D100' + D400$$

Implementation of leap year flag



Arithmetic circuits

- Excellent examples of combinational logic design
- Time vs. space trade-offs
 - doing things fast may require more logic and thus more space
 - example: carry lookahead logic
- Arithmetic and logic units
 - general-purpose building blocks
 - critical components of processor datapaths
 - used within most computer instructions

Number systems

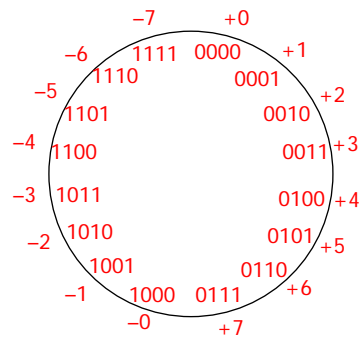
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
 - sign and magnitude
 - 1s complement
 - 2s complement
- Assumptions
 - we'll assume a 4 bit machine word
 - 16 different values can be represented
 - roughly half are positive, half are negative

Sign and magnitude

- One bit dedicate to sign (positive or negative)
 - sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
 - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
 - $\pm 2^{n-1} - 1$ (two representations for 0)
- Cumbersome addition/subtraction
 - must compare magnitudes to determine sign of result

$$0\ 100 = +4$$

$$1\ 100 = -4$$

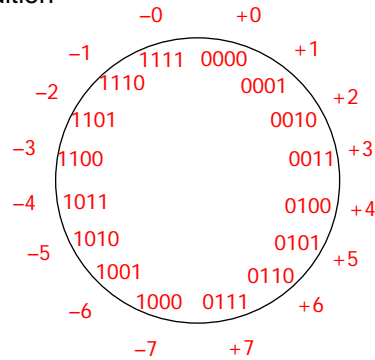


1s complement

- Subtraction: first form 1s complement and then add
- Two representations of 0
 - causes some complexities in addition
- High-order bit can act as sign bit

$$0\ 100 = +4$$

$$1\ 011 = -4$$

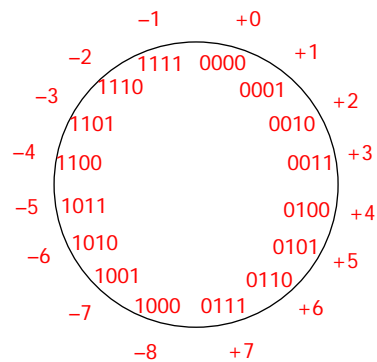


2s complement

- Same as 1s complement but with negative numbers shifted one position towards 0 (merge two 0s into a single representations)
 - only one representation for 0
 - one more negative number than positive numbers
 - high-order bit can act as sign bit

$$0\ 100 = +4$$

$$1\ 100 = -4$$



2s complement (cont'd)

- If N is a positive number, then the negative of N (its 2s complement or N^*) is $N^* = 2^n - N$

- example: 2s complement of 7

$$\begin{array}{r} 2^4 = 10000 \\ \text{subtract } 7 = \underline{0111} \end{array}$$

$$1001 = \text{repr. of } -7$$

- example: 2s complement of -7

$$\begin{array}{r} 2^4 = 10000 \\ \text{subtract } -7 = \underline{1001} \end{array}$$

$$0111 = \text{repr. of } 7$$

- shortcut: 2s complement = bit-wise complement + 1
 - 0111 -> 1000 + 1 -> 1001 (representation of -7)
 - 1001 -> 0110 + 1 -> 0111 (representation of 7)

2s complement addition and subtraction

- Simple addition and subtraction
 - simple scheme makes 2s complement the unanimous choice for integer number systems in computers

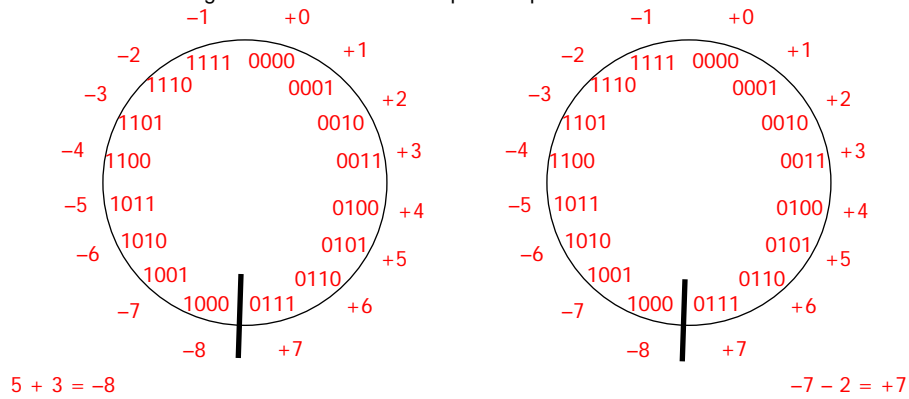
$$\begin{array}{r} 4 \quad 0100 \\ + 3 \quad 0011 \\ \hline 7 \quad 0111 \end{array} \qquad \begin{array}{r} -4 \quad 1100 \\ + (-3) \quad 1101 \\ \hline -7 \quad 11001 \end{array}$$

$$\begin{array}{r} 4 \quad 0100 \\ - 3 \quad 1101 \\ \hline 1 \quad 10001 \end{array} \qquad \begin{array}{r} -4 \quad 1100 \\ + 3 \quad 0011 \\ \hline -1 \quad 1111 \end{array}$$

Overflow in 2s complement addition/subtraction

- Overflow conditions

- add two positive numbers and end up with a negative number
- add two negative numbers and end up with a positive number



Overflow conditions

- Overflow

- $A_3' B_3' S_3 + A_3 B_3 S_3'$

- Another way to say same thing

- When carry into sign bit position is not equal to carry-out

0 1 1 1	0 0 0 0	1 0 0 0	1 1 1 1
5 0 1 0 1	5 0 1 0 1	-7 1 0 0 1	-3 1 1 0 1
<u>3 0 0 1 1</u>	<u>2 0 0 1 0</u>	<u>-2 1 1 1 0</u>	<u>-5 1 0 1 1</u>
-8 1 0 0 0	7 0 1 1 1	7 1 0 1 1 1	-8 1 1 0 0 0
overflow	no overflow	overflow	no overflow

Circuits for binary addition

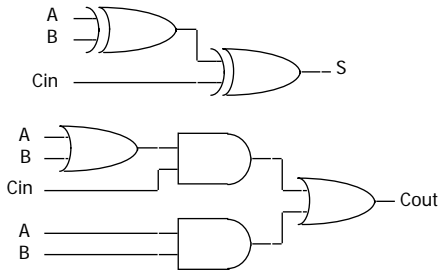
- Half adder (add 2 1-bit numbers)
 - $\text{Sum} = A_i' B_i + A_i B_i' = A_i \text{ xor } B_i$
 - $\text{Cout} = A_i B_i$
- Full adder (carry-in to cascade for multi-bit adders)
 - $\text{Sum} = C_i \text{ xor } A \text{ xor } B$
 - $\text{Cout} = B C_i + A C_i + A B = C_i (A + B) + A B$

A _i	B _i	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

A _i	B _i	C _{in}	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

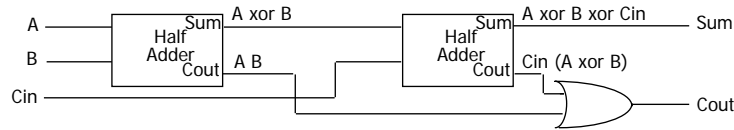
Full adder implementations

- Standard approach
 - 6 gates
 - 2 XORs, 2 ANDs, 2 ORs



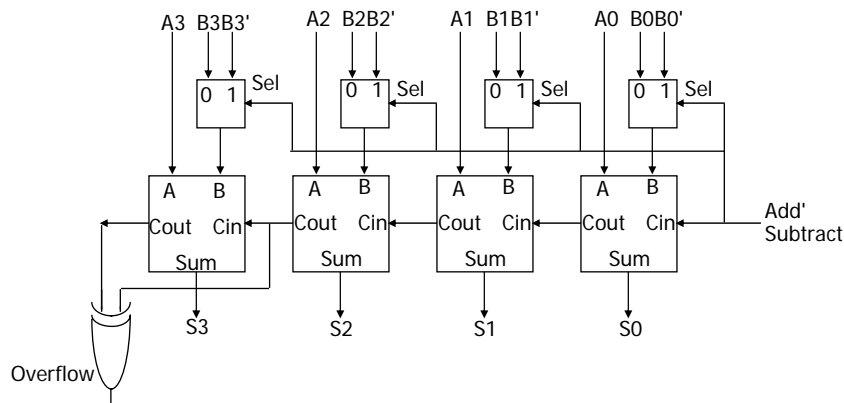
- Alternative implementation
 - 5 gates
 - half adder is an XOR gate and AND gate
 - 2 XORs, 2 ANDs, 1 OR

$$\text{Cout} = A B + C_{in} (A \text{ xor } B) = A B + B C_{in} + A C_{in}$$



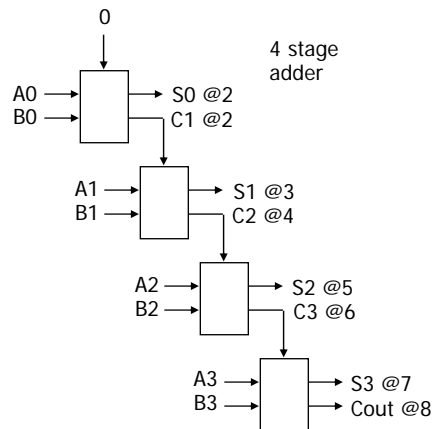
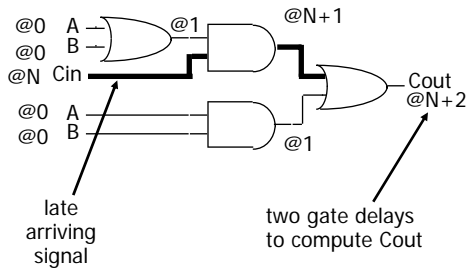
Adder/subtractor

- Use an adder to do subtraction thanks to 2s complement representation
 - $A - B = A + (-B) = A + B' + 1$
 - control signal selects B or 2s complement of B



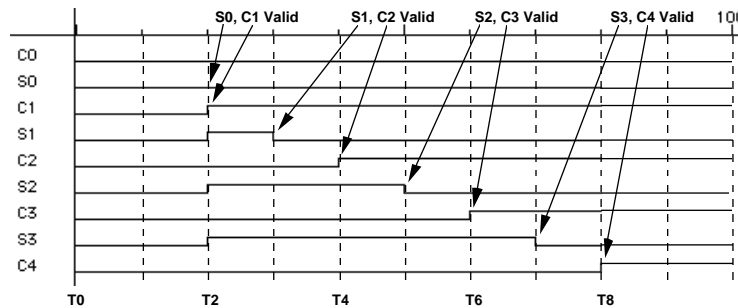
Ripple-carry adders

- Critical delay
 - the propagation of carry from low to high order stages



Ripple-carry adders (cont'd)

- Critical delay
 - the propagation of carry from low to high order stages
 - 1111 + 0001 is the worst case addition
 - carry must propagate through all bits



Carry-lookahead logic

- Carry generate: $G_i = A_i B_i$
 - must generate carry when $A = B = 1$
- Carry propagate: $P_i = A_i \text{ xor } B_i$
 - carry-in will equal carry-out here
- Carry kill: $K_i = A_i' B_i'$
 - carry-out will be zero no matter what carry-in when $A = B = 0$
- $G_i + P_i + K_i = 1$
- Sum and Cout can be re-expressed in terms of generate/propagate (or in terms of generate/kill):
 - $S_i = A_i \text{ xor } B_i \text{ xor } C_i$
 $= P_i \text{ xor } C_i$
 - $C_{i+1} = A_i B_i + A_i C_i + B_i C_i$
 $= A_i B_i + C_i (A_i + B_i)$
 $= A_i B_i + C_i (A_i \text{ xor } B_i)$
 $= G_i + P_i C_i$

$$C_{i+1}' = K_i + P_i C_i'$$

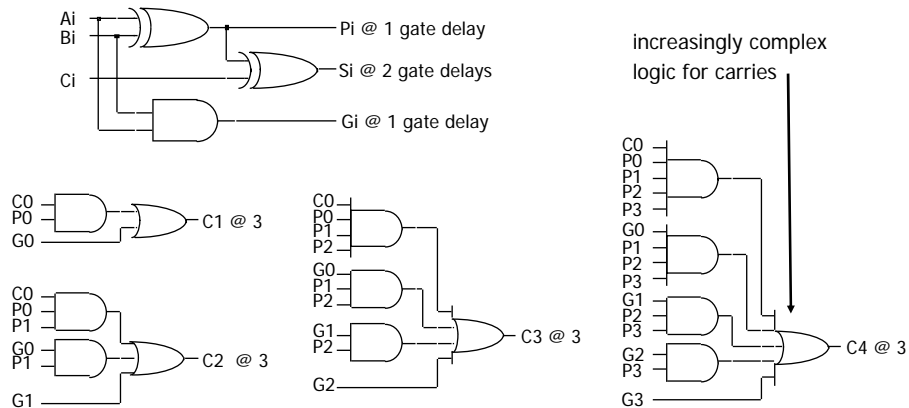
$$C_{i+1} = K_i' (P_i' + C_i)$$

Carry-lookahead logic (cont'd)

- Re-express the carry logic as follows:
 - $C_1 = G_0 + P_0 C_0$
 - $C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0$
 - $C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$
 - $C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0$
- Each of the carry equations can be implemented with two-level logic
 - all inputs are now directly derived from data inputs and not from intermediate carries
 - this allows computation of all sum outputs to proceed in parallel

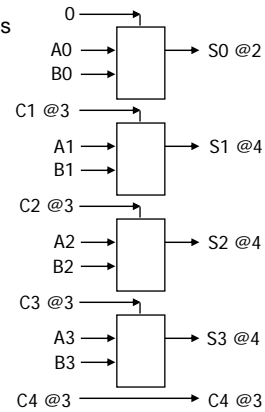
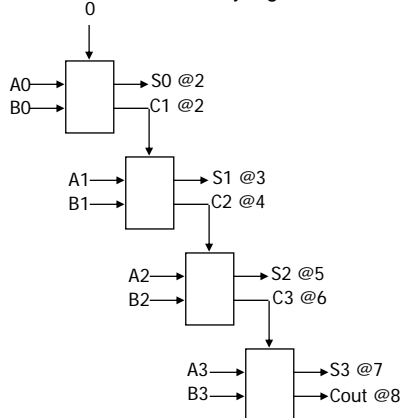
Carry-lookahead implementation

- Adder with propagate and generate outputs



Carry-lookahead implementation (cont'd)

- Carry-lookahead logic generates individual carries
 - sums computed much more quickly in parallel
 - however, cost of carry logic increases with more stages

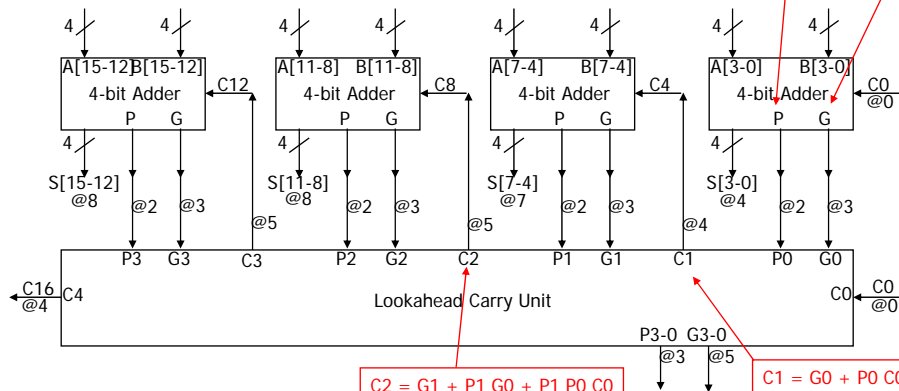


Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
 - 4 four-bit adders with internal carry lookahead
 - second level carry lookahead unit extends lookahead to 16 bits

$$G = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

$$P = P_3 P_2 P_1 P_0$$

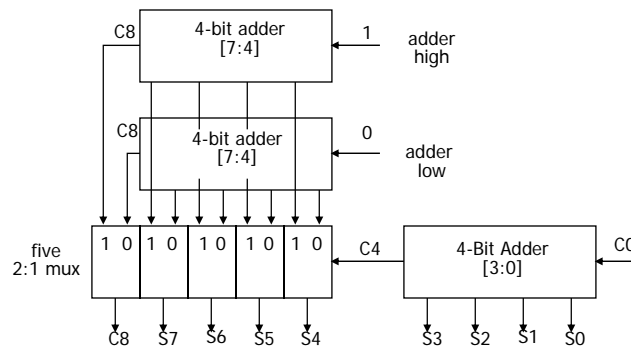


$$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_1 = G_0 + P_0 C_0$$

Carry-select adder

- Redundant hardware to make carry calculation go faster
 - compute two high-order sums in parallel while waiting for carry-in
 - one assuming carry-in is 0 and another assuming carry-in is 1
 - select correct result once carry-in is finally computed



Arithmetic logic unit design specification

M = 0, logical bitwise operations

S1	S0	Function	Comment
0	0	$F_i = A_i$	input A_i transferred to output
0	1	$F_i = \text{not } A_i$	complement of A_i transferred to output
1	0	$F_i = A_i \text{ xor } B_i$	compute XOR of A_i, B_i
1	1	$F_i = A_i \text{ xnor } B_i$	compute XNOR of A_i, B_i

M = 1, C0 = 0, arithmetic operations

0	0	$F = A$	input A passed to output
0	1	$F = \text{not } A$	complement of A passed to output
1	0	$F = A \text{ plus } B$	sum of A and B
1	1	$F = (\text{not } A) \text{ plus } B$	sum of B and complement of A

M = 1, C0 = 1, arithmetic operations

0	0	$F = A \text{ plus } 1$	increment A
0	1	$F = (\text{not } A) \text{ plus } 1$	twos complement of A
1	0	$F = A \text{ plus } B \text{ plus } 1$	increment sum of A and B
1	1	$F = (\text{not } A) \text{ plus } B \text{ plus } 1$	B minus A

logical and arithmetic operations
not all operations appear useful, but "fall out" of internal logic

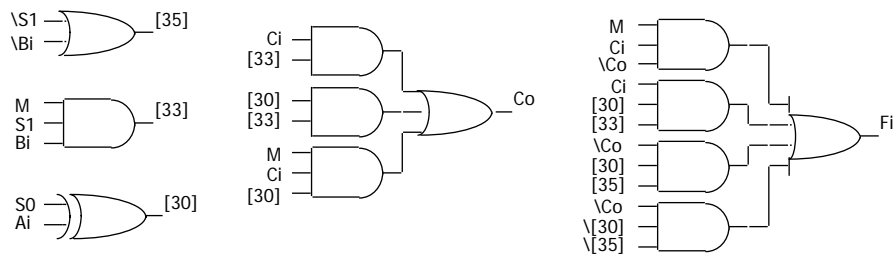
Arithmetic logic unit design (cont'd)

- Sample ALU – truth table

M	S1	S0	Ci	Ai	Bi	Fi	Ci+1		
0	0	0	X	0	X	0	X		
		1	X	1	X	1	X		
		X	X	X	X	X	X		
	1	0	1	X	0	X	0	X	
		0	0	X	1	X	1	X	
		1	0	X	0	0	0	X	
	1	1	0	X	0	1	1	X	
			1	0	1	0	1	X	
			X	X	X	X	X	X	
		1	1	1	X	0	0	0	X
			0	1	X	0	1	0	X
			0	0	X	1	0	0	X
1	0	0	0	0	X	0	X		
		1	0	1	X	1	X		
		X	X	X	X	X	X		
	1	0	1	0	0	X	0	X	
		0	0	0	1	X	1	X	
		1	0	0	0	0	0	0	
	1	1	0	0	1	0	1	0	
			1	0	1	1	1	0	
			X	X	X	X	X	X	
		1	1	1	0	0	1	0	1
			0	1	0	1	0	1	0
			0	0	1	0	1	0	0
1	0	0	1	0	X	1	0		
		1	1	0	X	0	1		
		X	X	X	X	X	X		
	1	0	1	1	0	X	1	0	
		0	0	1	1	X	0	1	
		1	0	1	0	0	0	1	
	1	1	1	1	0	1	0	1	
			1	1	1	0	1	1	
			X	X	X	X	X	X	
		1	1	1	0	0	1	1	
			0	1	0	1	0	1	
			0	0	1	0	1	0	

Arithmetic logic unit design (cont'd)

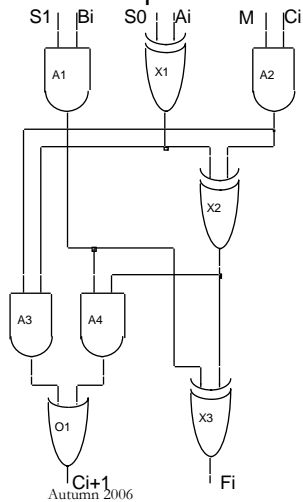
- Sample ALU – multi-level discrete gate logic implementation



12 gates

Arithmetic logic unit design (cont'd)

■ Sample ALU – clever multi-level implementation



first-level gates
 use S0 to complement Ai
 S0 = 0 causes gate X1 to pass Ai
 S0 = 1 causes gate X1 to pass Ai'
 use S1 to block Bi
 S1 = 0 causes gate A1 to make Bi go forward as 0
 (don't want Bi for operations with just A)
 S1 = 1 causes gate A1 to pass Bi
 use M to block Ci
 M = 0 causes gate A2 to make Ci go forward as 0
 (don't want Ci for logical operations)
 M = 1 causes gate A2 to pass Ci

other gates
 for M=0 (logical operations, Ci is ignored)
 $F_i = S_1 B_i \text{ xor } (S_0 \text{ xor } A_i)$
 $= S_1 S_0' (A_i) + S_1 S_0 (A_i') +$
 $S_1 S_0' (A_i B_i' + A_i' B_i) + S_1 S_0 (A_i' B_i' + A_i B_i)$
 for M=1 (arithmetic operations)
 $F_i = S_1 B_i \text{ xor } ((S_0 \text{ xor } A_i) \text{ xor } C_i) =$
 $C_{i+1} = C_i (S_0 \text{ xor } A_i) + S_1 B_i ((S_0 \text{ xor } A_i) \text{ xor } C_i) =$
 just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

CSE370 - VI - Combinational Logic Case Studies

41

Summary for examples of combinational logic

- Combinational logic design process
 - formalize problem: encodings, truth-table, equations
 - choose implementation technology (ROM, PAL, PLA, discrete gates)
 - implement by following the design procedure for that technology
- Binary number representation
 - positive numbers the same
 - difference is in how negative numbers are represented
 - 2s complement easiest to handle: one representation for zero, slightly complicated complementation, simple addition
- Circuits for binary addition
 - basic half-adder and full-adder
 - carry lookahead logic
 - carry-select
- ALU Design
 - specification, implementation

Autumn 2006

CSE370 - VI - Combinational Logic Case Studies

42