Chapter 5
Case studies in combinational logic design

Now we have gone through every part of combinational logic system design.

In this chapter, the final chapter of combination logic, we will look at some examples and elaborate on the whole design process.

Combinational logic design case studies

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

I will talk about the steps of a general procedure to design a combinational logic system first and then take some examples.

General design procedure for combi. 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 (fixed logic)
4. **Follow implementation procedure**
   - K-maps for two-level, multi-level
   - design CAD tools and hardware description language (e.g., Verilog)

Typically, we have to separate the I/O variables and system internals to understand the problem. Step 2 is abstract representation while steps 3 and 4 are H/W implementation dependent

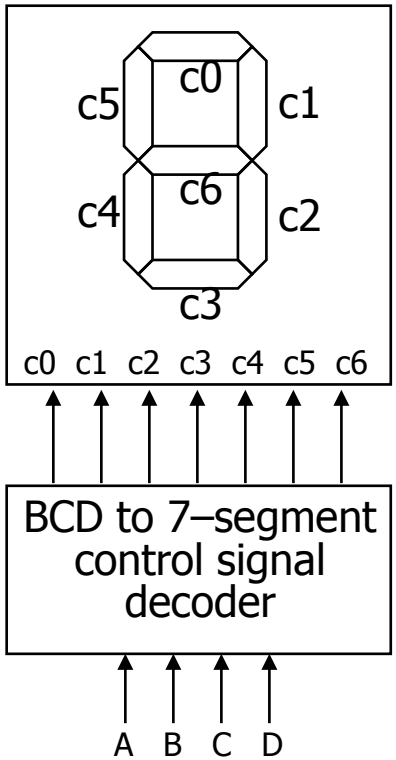
Tip: CAD tool's typical functionalities

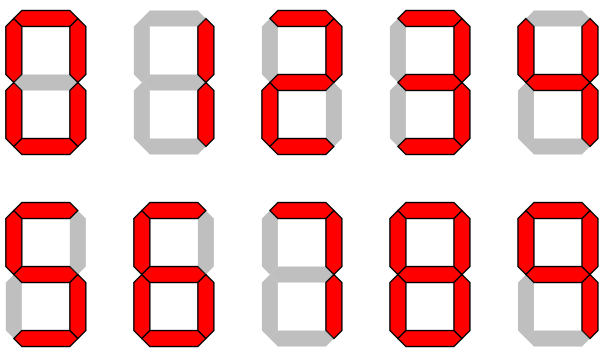
- **Design entry**
  - Truth table, schematic capture, HDL
- **Synthesis and optimization**
- **Simulation**
- **Physical design**

First of all, a user of a CAD tool should be able to specify the requirements of a logic circuit to be designed. The requirements can be expressed by a truth table or a graphical drawing, or a language.
Synthesis usually refers to the process of translating designer's requirements into a circuit graph.
Physical design transforms the circuit graph into a layout (or blueprint) for fabrication

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**





The first case is to control or display one digit system whose input follows BCD coding.

A digit can be represented by a combination of 7 segments. Depending on input variables, we have to turn on relevant pieces or segments.

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	–	–	–	–	–	–	–	–	–

How many input variables? And output variables?

If we use ROM for the implementation technology, then the game is over. What we need to do is just to store 7 bit values for each minterm, total 2**4 cases. Even though we need to use only 10 minterms, we have to use 2**4 AND gates, whose size is much bigger than other technologies. So we choose PAL or PLA, where minimum # of product terms are desirable. As there are DC terms, we may be able to reduce the size of the system by minimization in two-level logic design.

Implementation as minimized sum-of-products

- **15 unique product terms when minimized individually**

C0

C \ AB	00	01	11	10
00	1	0	X	1
01	0	1	X	1
11	1	1	X	X
10	1	1	X	X

C1

C \ AB	00	01	11	10
00	1	1	X	1
01	1	0	X	1
11	1	1	X	X
10	1	0	X	X

C2

C \ AB	00	01	11	10
00	1	1	X	1
01	1	1	X	1
11	1	1	X	X
10	0	1	X	X

C3

C \ AB	00	01	11	10
00	1	0	X	1
01	0	1	X	0
11	1	0	X	X
10	1	1	X	X

C4

C \ AB	00	01	11	10
00	1	0	X	1
01	0	0	X	0
11	0	0	X	X
10	1	1	X	X

C5

C \ AB	00	01	11	10
00	1	1	X	1
01	0	1	X	1
11	0	0	X	X
10	0	1	X	X

C6

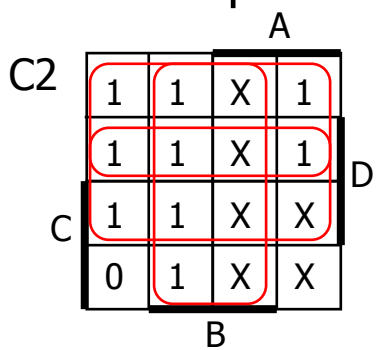
C \ AB	00	01	11	10
00	0	1	X	1
01	0	1	X	1
11	1	0	X	X
10	1	1	X	X

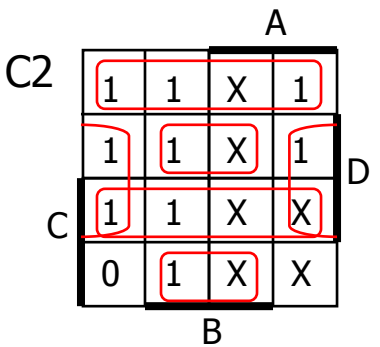
C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

Here are the final two-level s-o-p forms for 7 output functions. After removing duplicate product terms, we have 15 different product terms to represent 7 outputs. We should note that the minimized form of each output may not lead to the globally minimum number of product terms. In this case, we have 15 product terms for 7 outputs

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



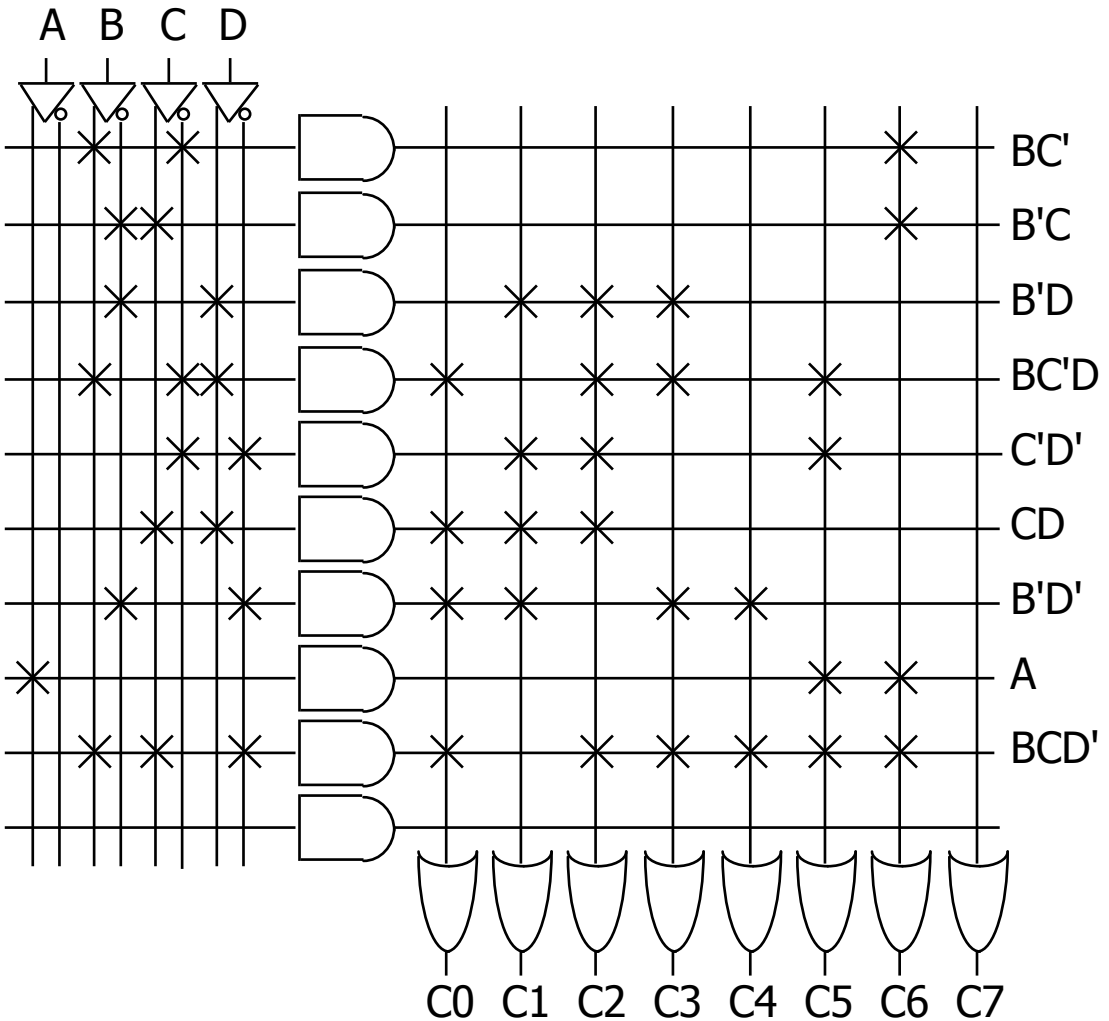


C0 = A + B D + C + B' D'
C1 = C' D' + C D + B'
C2 = B + C' + D
C3 = B' D' + C D' + B C' D + B' C
C4 = B' D' + C D'
C5 = A + C' D' + B D' + B C'
C6 = A + C D' + B C' + B' C

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

Let's look at C2 output variable. If we relax the minimization process of C2, we have more terms for C2 output. However, this less efficient minimization achieves sharing more product terms in the overall system (7 outputs). Note that the number of product terms of each output increases. Can we figure out this globally optimal minimization?

PLA implementation



Since there are up to 5 product terms for outputs, PAL may not be attractive. There are many common terms; so PLA is a good choice.

PAL implem. 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 shared terms)

    C2 = B + C' + D

    C2 = B' D + B C' D + C' D' + C D + 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'

Let's consider PAL implementation where up to 4 product terms can be ORed. Unfortunately, there are outputs that have 5 product terms. In that case, we have to resort to multi-level logic. You don't need to do that; the CAD tool will do the job.

Logical function unit

- **Multi-purpose function block**
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

C0	C1	C2	Function	Comments
0	0	0	1	always 1
0	0	1	A + B	logical OR
0	1	0	(A • B)'	logical NAND
0	1	1	A xor B	logical xor
1	0	0	A xnor B	logical xnor
1	0	1	A • B	logical AND
1	1	0	(A + B)'	logical NOR
1	1	1	0	always 0

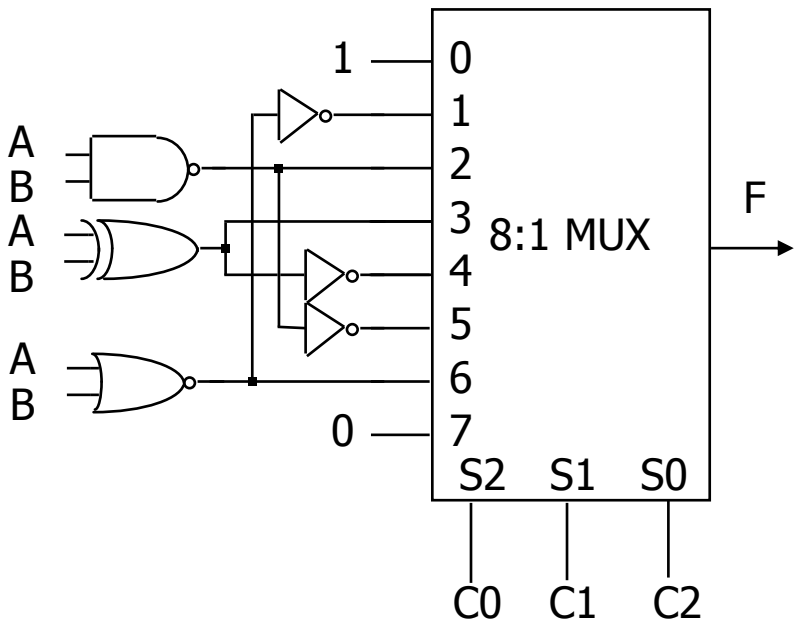
3 control inputs: C0, C1, C2
2 data inputs: A, B
1 output: F

The next example is the very versatile functional block, which performs various functions. There are three control variables whose values determine the output of the function of two data input variables.

Formalize the problem

C0	C1	C2	A	B	F
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	0	1	0
1	1	1	1	0	0
1	1	1	1	1	0

choose implementation technology
5-variable K-map to discrete gates
multiplexer implementation



In order to realize the multi-purpose function, we combine discrete gates and an 8:1 MUX.

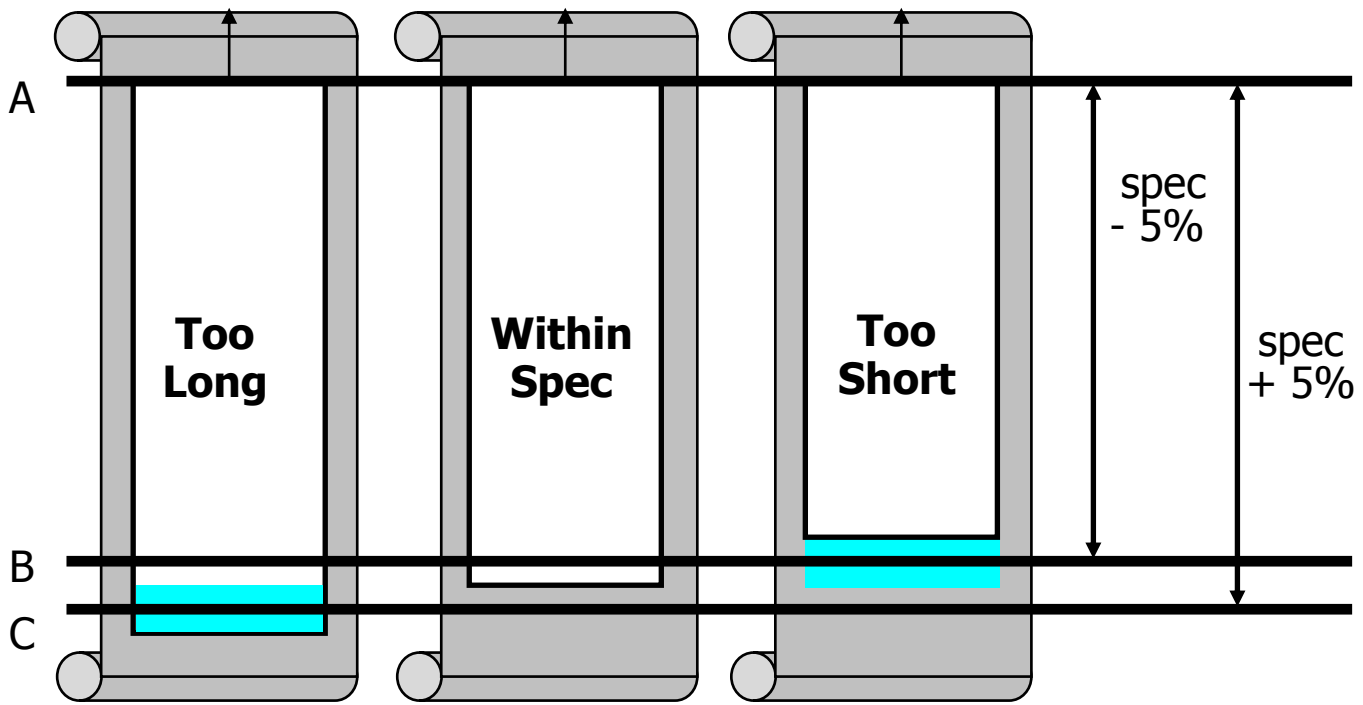
Production line control

- **Rods of varying length (+/-10%) travel on conveyor belt**
  - mechanical arm pushes rods within spec (+/-5%) to one side
  - second arm pushes rods too long to other side
  - rods that are too short stay on belt
  - 3 light barriers (light source + photocell) as sensors
  - design combinational logic to activate the arms
- **Understanding the problem**
  - inputs are three sensors
  - outputs are two arm control signals
  - assume sensor reads "1" when tripped
    - "0" otherwise (if nothing to detect)

The next case study to look at is a product line control (PLC) system that manufactures a rod. What we need to do is to check whether each rod's length is within a certain bound (± 5% of the spec). In order to examine the length of each rod, we use three sensors to measure the rod. If the rod is within spec, the rod will be pushed to one side. If the rod is too long, it will be pushed to the other side. Otherwise, it will stay on the belt.

Sketch of problem

- **Position of sensors**
  - A to B distance = specification – 5%
  - A to C distance = specification + 5%



What if we don't have sensor A? We cannot know the reference point or time at which we can measure the length of each rod. So when sensor A becomes 1, we should check sensors B and C as well to measure the length of the moving rod.

Formalize the problem

- **Truth table**
  - show don't cares

A	B	C	Function
0	0	0	do nothing
0	0	1	do nothing
0	1	0	do nothing
0	1	1	do nothing
1	0	0	too short
1	0	1	don't care
1	1	0	in spec
1	1	1	too long

logic implementation now straightforward
just use three 3-input AND gates

"too short" = AB'C'
(only first sensor tripped)

"in spec" = A B C'
(first two sensors tripped)

"too long" = A B C
(all three sensors tripped)

If A is zero, that means there is no rod to inspect. So all the outputs are 0.

Otherwise, there is a rod. Then we have to check sensors B and C. Note that 101 is a DC term. 110 means that the rod is of standard length. The textbook says the top 4 minterms (000 – 011) are also DC terms, but that may cause energy waste.

Steps 3 and 4

- **Choose implementation technology**
  - The actual logic is so simple; use fixed logic gates
- **Follow implementation procedure**
  - One minterm for each output

So the truth table is done. As the overall logic is so simple. Maybe the random logic approach is the best. There are three outputs, each of which has only one minterm

Calendar subsystem

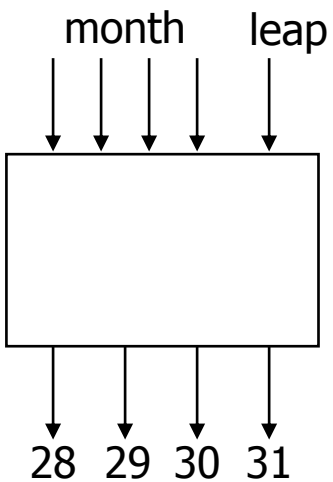
- **Determine number of days in a month (to control watch display)**
  - used in controlling the display of a wrist-watch LCD screen
  - inputs: month, leap year flag
  - outputs: number of days
- **Use software implementation to help understand the problem**

```
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);
     }
}
```

For the next example, we revisit the number of days per month problem. But this time, we will follow the general procedure to design the system. Then we will look at the details of the leap year flag.

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
0010	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–	–	–	–	–	–

Again 4 wires for the input of month and one wire for leap year and 4 wires for the output (note that only two wires are needed for output at minimum)

Choose implem. target and perform mapping

- **Discrete gates**
  - 28 = m8′ m4′ m2 m1′ leap′
  - 29 = m8′ m4′ m2 m1′ 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
0010	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–	–	–	–	–	–

If we are to use discrete logic gates such as AND or OR, SoP or PoS will be enough.

Just investigate all the elements of the On-set, and make product terms and then combine them by OR gates in SoP case. From now on we will look at the leap year flag. How can we know that a year is a leap year or not?

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, 2006:1111101010
    but difficult for 100 and 400 (not powers of 2)
  - BCD – easy for 100, 2006: 0010 0000 0000 0110
    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

Pope Gregory I set up the current leap year system in 1582. There are only three simple rules that decides the number of days in Feb. Note that an encoding scheme can affect the system design fundamentally.

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:**

  D4 = YT1' YO2' YO1' + YT1 YO2 YO1'

YM stands for year millennium and the each digit of the year will be represented as the binary value by BCD encoding. So we need 4 wires for each digit of the year, total 16 wires. D4 is the flag that indicates whether the year of input is divisible by 4

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:**

  D100 = YT8' YT4' YT2' YT1' • YO8' YO4' YO2' YO1'

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

  D400 = (YM1' YH2' YH1' + YM1 YH2 YH1')

  • (YT8' YT4' YT2' YT1' • YO8' YO4' YO2' YO1')

With BCD encoding, it is easy to check whether the year of input is divisible by 100 and 400. A year is divided by 100 without remainder if the lowest two digits are 00

A year is divided by 400 without residue if the lowest two digits are 00 and the highest two digits are divided by 4.

Combining to determine leap year flag

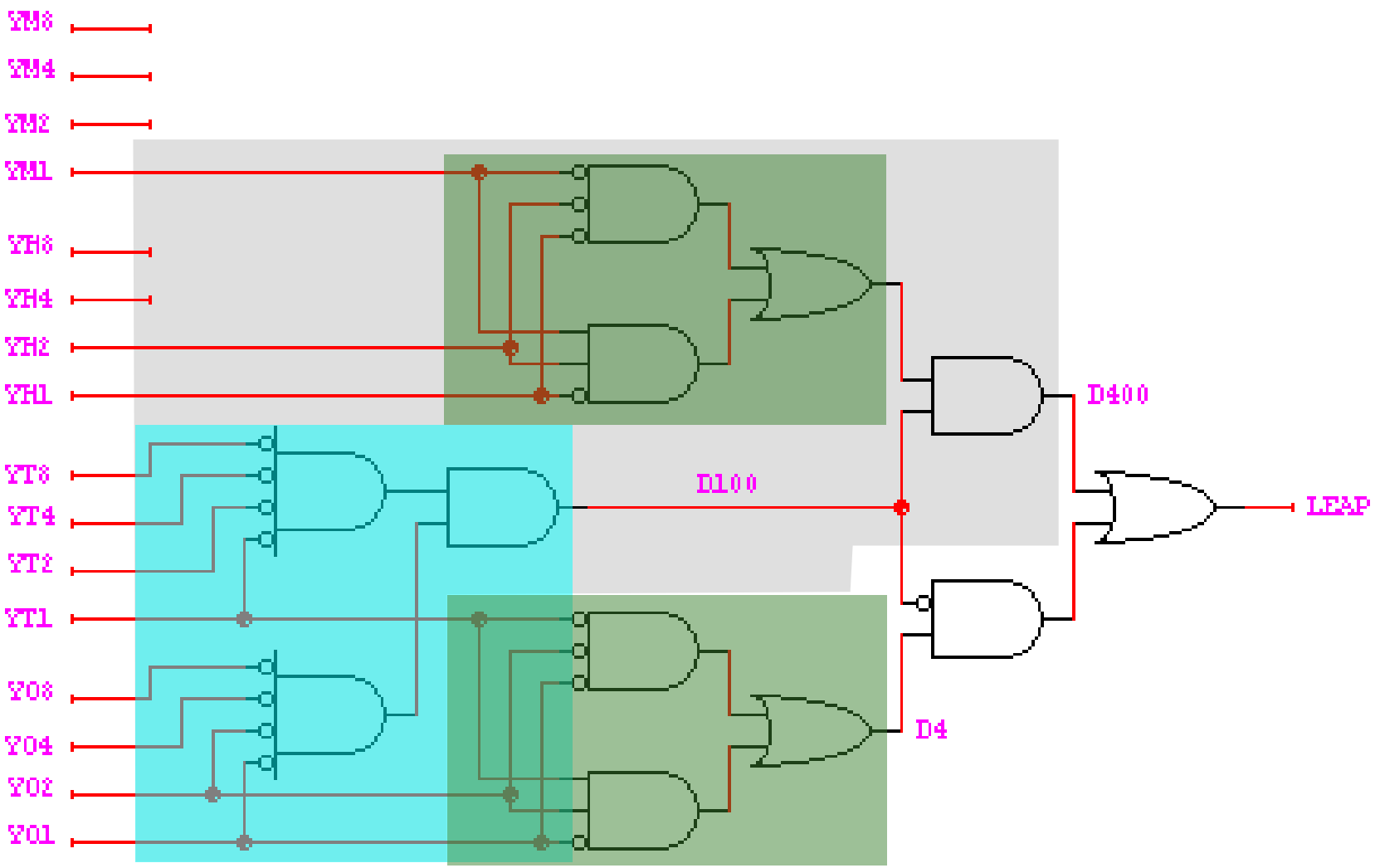
- **Label results of previous three circuits: D4, D100, and D400**

$$\text{leap\_year\_flag} = D4 \cdot D100' + D4 \cdot D400$$

$$= D4 \cdot D100' + D400$$

Now we have three (internal) variables, each of which indicates whether the year is divisible by 4, 100, 400, respectively. The final leap year flag should be true when D4 is true. But if D100 is true than the flag is false. Note that the flag is also true when D4 and D400 is true

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

Now we are going to see a little bit different logic system, whose purpose is to perform a mathematical function. We will look at adders mostly.

In most of engineering systems, there is always a trade-off. We cannot have all the best features in a single system. As mentioned earlier, if we use a number of gates in parallel (say, two-level SoP canonical forms), we can minimize the delay of the system. If we want to reduce the number of gates, there will be multiple levels, which increases delay. We will revisit this issue by going over multiple variations of adder systems

Number systems

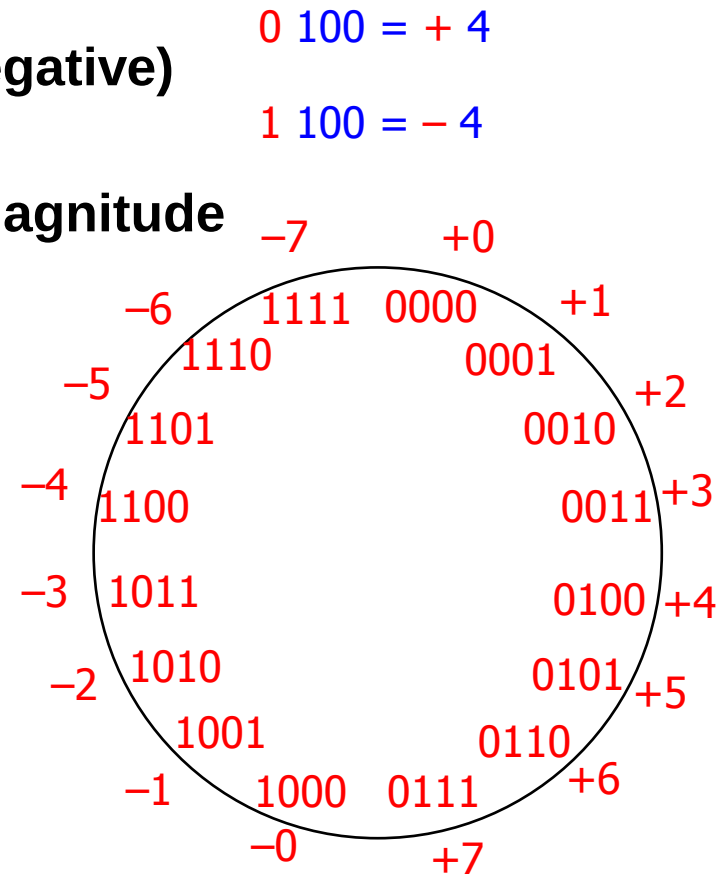
- **Representation of positive numbers is the same in most systems**
- **Major differences are in how negative numbers are represented**
- **Representation of negative numbers come in three major schemes**
  - sign and magnitude
  - 1s complement
  - 2s complement
- **Assumptions**
  - we'll assume a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

To design arithmetic functions, we have to understand how a number is represented in computers. We talked about binary coding, BCD coding and so on. Here we choose the binary coding as the basic representation system but will focus on how negative numbers are expressed.

For the purpose of illustration, we consider the case of using 4 wires to represent a digit (positive and negative and zero).

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



The first encoding scheme is sign and magnitude; one bit is dedicated to the sign of the number and the rest of the bits represents the absolute value of the number |x|. If we use 4 bits total, the MSB is the sign + or -, and the other three bits represent the magnitude from 0 to 7. The problem is that when adding or subtracting numbers, we have to check the sign of the results.

1s complement

- **If N is a positive number, then the negative of N (or its 1s complement) is N' = (2^n –1) – N**
  - example: 1s complement of 7

2^4 = 10000

1 = 00001

2^4–1 = 1111

7 = 0111

1000 = –7 in 1s complement form

  - shortcut: simply compute bit-wise complement (0111 -> 1000)

To remedy the sign-magnitude encoding problem, 1s complement is introduced. Here, a negative of N (which is –N) is denoted by N' and is expressed by (2^n-1)-N.

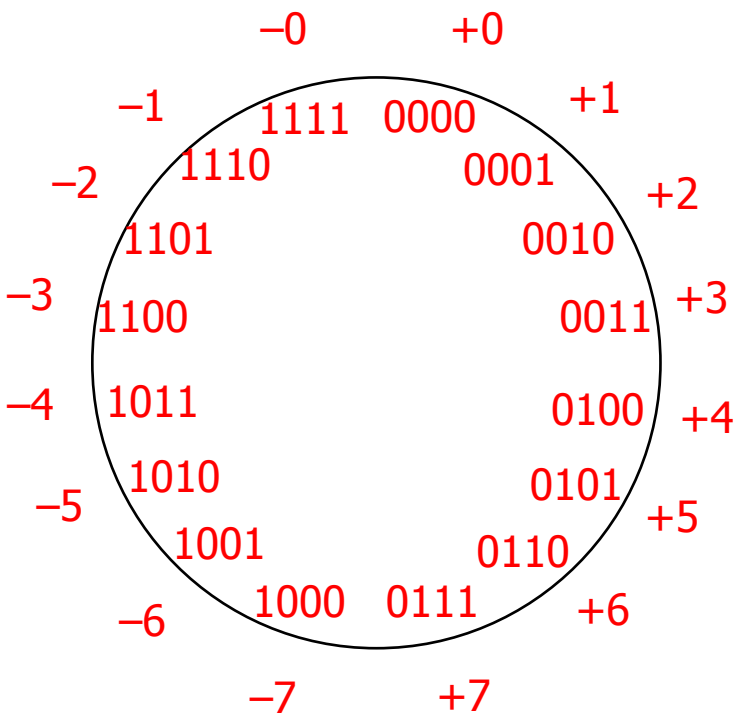
The 1st bit implicitly says the sign of the number. The merit of this encoding scheme is that subtraction becomes easier.

1s complement (cont'd)

- **Subtraction implemented by 1s complement and then addition**
- **Two representations of 0**
  - causes some complexities in addition
- **High-order bit can act as sign bit**
- **Carry should be added to the sum**

0 100 = + 4

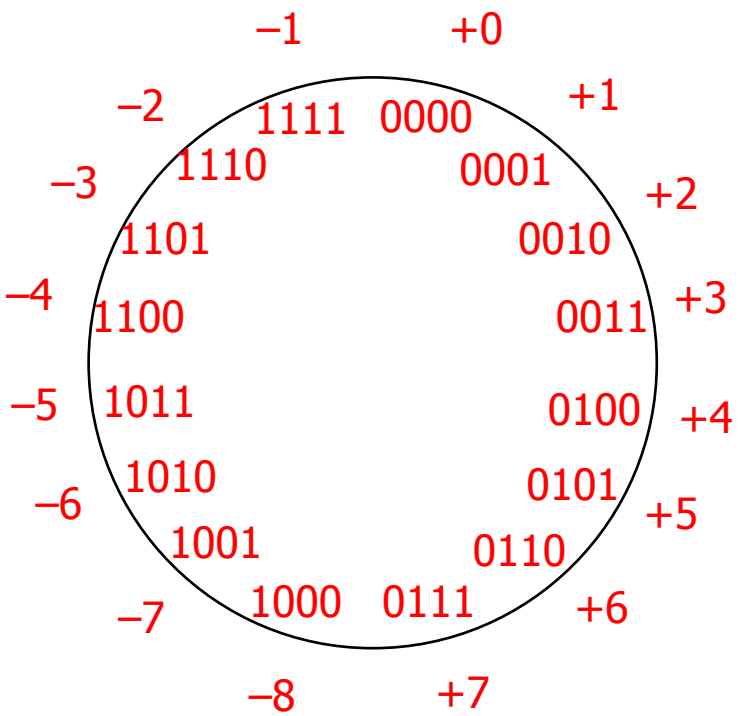
1 011 = – 4



Then, subtraction can be easily performed. To compute A-B, just calculate 1s complement of B and then add it to A. Actually this works only when B >= A. Look at the range from -7 to +7 and note that there are two kinds of 0s.

2s complement

- **1s complement with negative numbers shifted one position clockwise**
  - 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 compliment enhances the 1s complement encoding, it can represent up to -8 as shown in the above.

2s complement (cont'd)

- **If N is a positive number, then the negative of N (or its 2s complement) is N* = 2^n – N**
  - example: 2s complement of 7

$$2^4 = 10000$$
$$\text{subtract } 7 = 0111$$
$$1001 = \text{repr. of } -7$$

  - example: 2s complement of –7

$$2^4 = 10000$$
$$\text{subtract } -7 = 1001$$
$$0111 = \text{repr. of } 7$$

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

Likewise, the 2s compliment of a positive number N is denoted by N*. If you want to make a 2s complement in case of 4 wires, just subtract the number from 2**4. Another simple way to calculate 2s complements is to invert all the bits and add 1.

2s complement addition and subtraction

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

4	0100	− 4	1100
+ 3	0011	+ (− 3)	1101
7	0111	− 7	11001

4	0100	− 4	1100
− 3	1101	+ 3	0011
1	10001	− 1	1111

Let's see how we can add or subtract the numbers encoded by 2s complement. Now subtraction is not so different from addition. Just make 2s complement and add it.

Why can the carry-out be ignored?

- **Can't ignore it completely**
  - needed to check for overflow (see next two slides)
- **When there is no overflow, carry-out may be true but can be ignored**

  – M + N when N > M:

  $M^* + N = (2^n - M) + N = 2^n + (N - M)$

  ignoring carry-out is just like subtracting 2^n

  – M + – N where $N + M \leq 2^{n-1}$

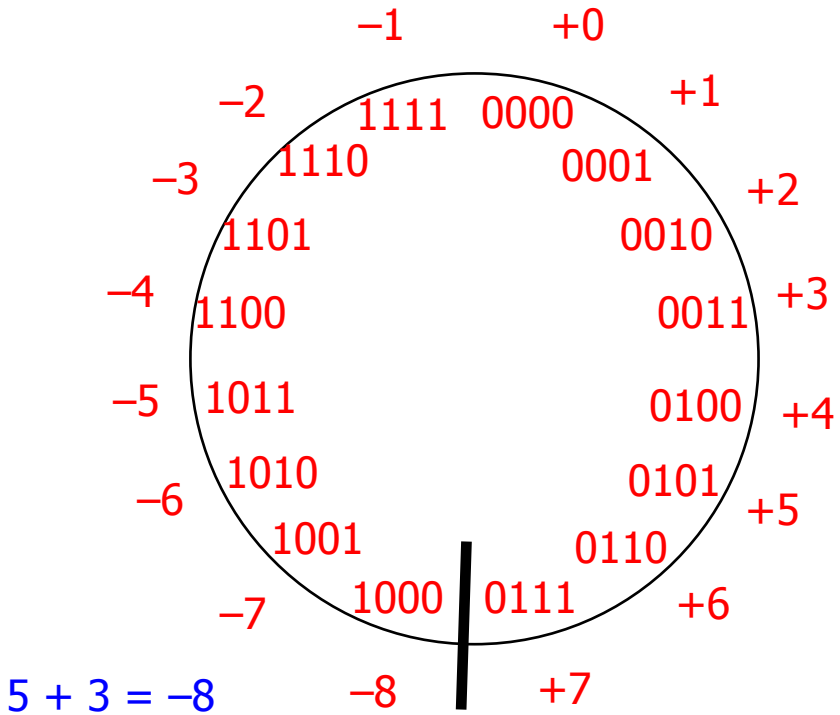
  $(-M) + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2^n$

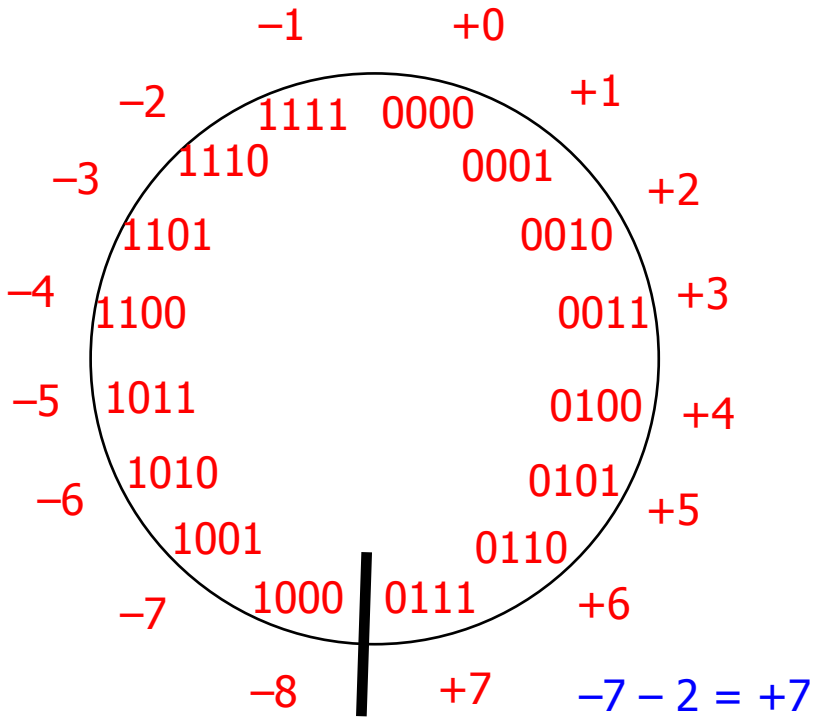
  ignoring the carry, it is just the 2s complement representation for – (M + N)

So far, we didn't pay attention to the carry. We will first look at the cases where carry-outs can be ignored. Let M be the positive number to be converted by 2s complement. M* denotes 2s complement of M, which is 2^n-M.

Overflow in 2s complement addition / subtraction

- **Overflow conditions**
  - add two positive numbers to get a negative number
  - add two negative numbers to get a positive number





There are two kinds of overflows in 2s complement arithmetic. In this case, the carry-out should not be ignored. The upper bound in 4 bits is +7; so when the sum exceeds this bound, there will be an overflow. Likewise, the lower bound is -8; so when the sum goes below -8, there is an overflow, which is also called underflow. Anyway, in these overflow cases, the adding system should report the error.

Overflow conditions

- **Overflow when carry into sign bit position is not equal to carry-out**

	0 1 1 1	
5	0 1 0 1	
3	0 0 1 1	
– 8	1 0 0 0	

overflow

	1 0 0 0	
– 7	1 0 0 1	
– 2	1 1 1 0	
7	1 0 1 1 1	

overflow

	0 0 0 0	
5	0 1 0 1	
2	0 0 1 0	
7	0 1 1 1	

no overflow

	1 1 1 1	
– 3	1 1 0 1	
– 5	1 0 1 1	
– 8	1 1 0 0 0	

no overflow

The numbers in blue indicate carries into the next higher-order bits in calculation.

The condition to check overflow is to compare two highest carries in addition. If they are different, then it is an overflow.

Circuits for binary addition

- **Half adder (add 2 1-bit numbers)**
  - Sum = Ai' Bi + Ai Bi' = Ai xor Bi
  - Cout = Ai Bi
- **Full adder (carry-in to cascade for multi-bit adders)**
  - Sum = Ci xor A xor B
  - Cout = B Ci + A Ci + A B = Ci (A + B) + A B

Ai	Bi	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Ai	Bi	Cin	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

Now we get familiar with 2s complement. Let's design a circuit that adds two binary numbers. First of all, let's start with a 1 bit adder. Ai and Bi are i-th bits of two binary numbers, A and B, respectively. On the left, there is a truth table for a half adder; on the right, a truth table for a full adder which also considers the carry-in from the lower-order bits.

(A or B)C vs. (A xor B)C

- **They are not equivalent but AB+(A+B)C = AB+(A⊕B)C**

A	B	C	A+B	(A+B)C
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

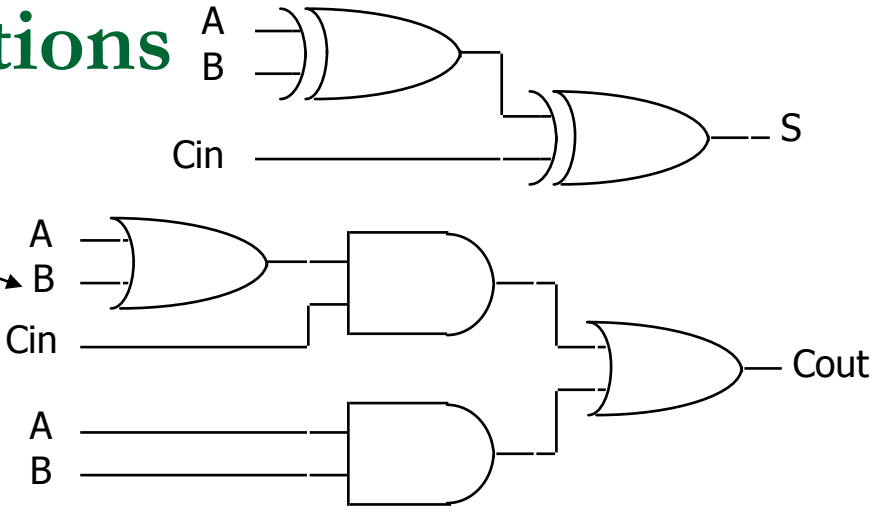
A	B	C	A⊕B	(A⊕B)C
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

Before going to the next step, let's see the relation between two boolean expressions.

(A or B)C is not equal to (A xor B)C. but when we add AB product term, they are equivalent.

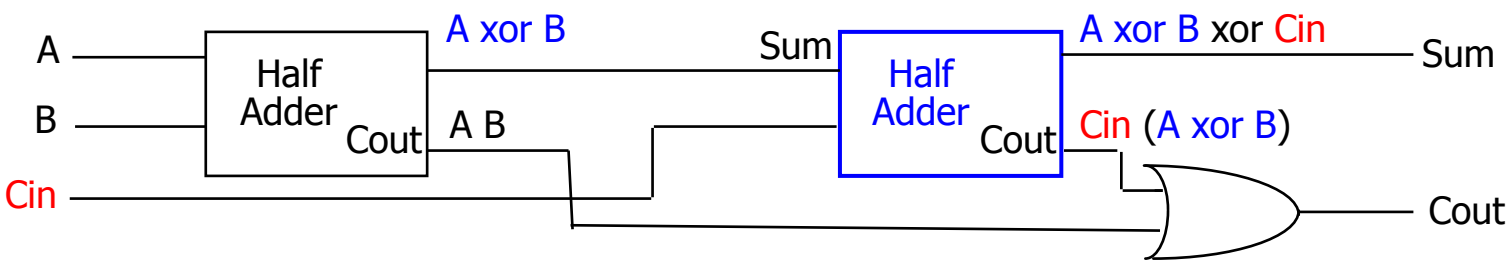
Full adder implementations

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



Cout = A B + Cin (A xor B) = A B + Cin (A or B)

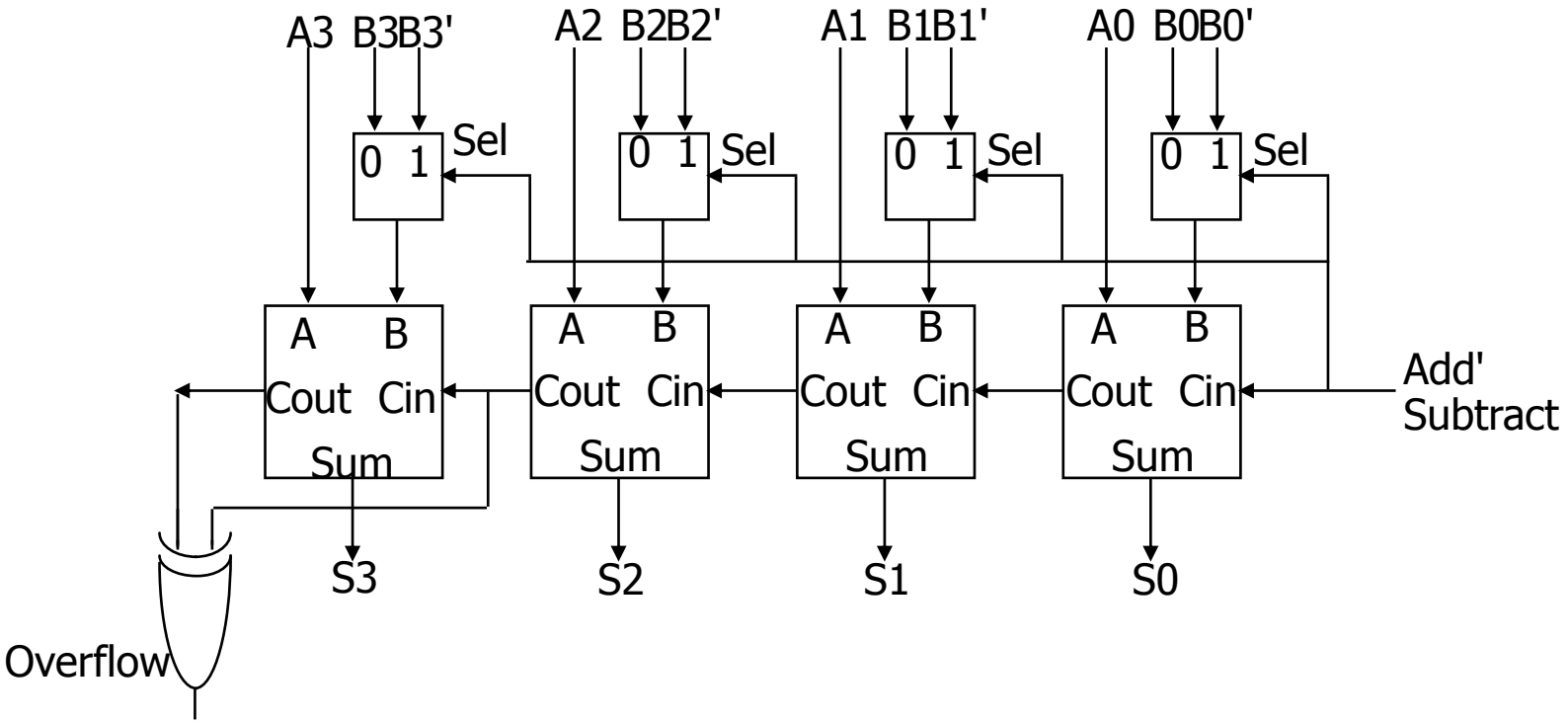
- **Alternative implementation**
  - 5 gates
  - half adder is an XOR gate and AND gate
  - 2 XORs, 2 ANDs, 1 OR



At the top, there is a fixed logic implementation of a full adder. If we use two half-adder modules to construct the full-adder system instead of using random logic, we can reduce the number of gates.

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

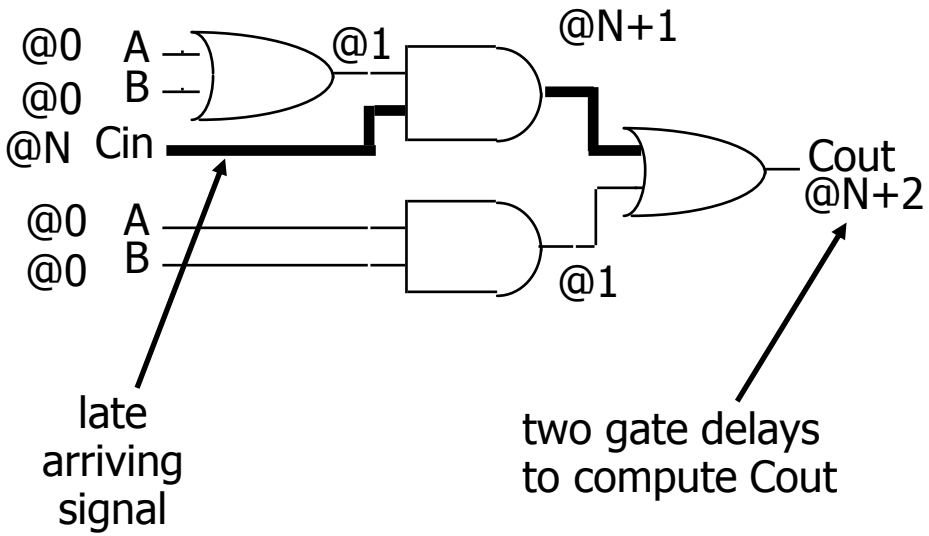


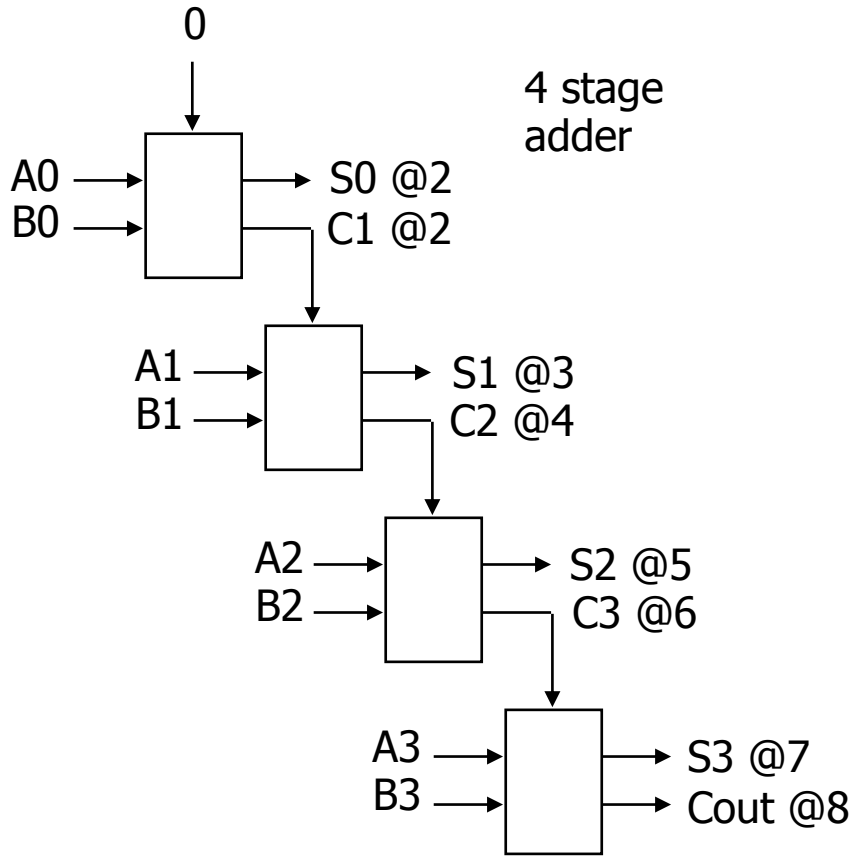
Now we have a full adder for one bit addition. Then what we want to do is addition/subtraction of two 4bit-numbers. Addition is easy. Recall that 2s complement of a number is its inverted form+1. So we can use the same adding function to perform subtraction too. For subtraction, we just enable the control input.

Ripple-carry adders

1111+0001

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

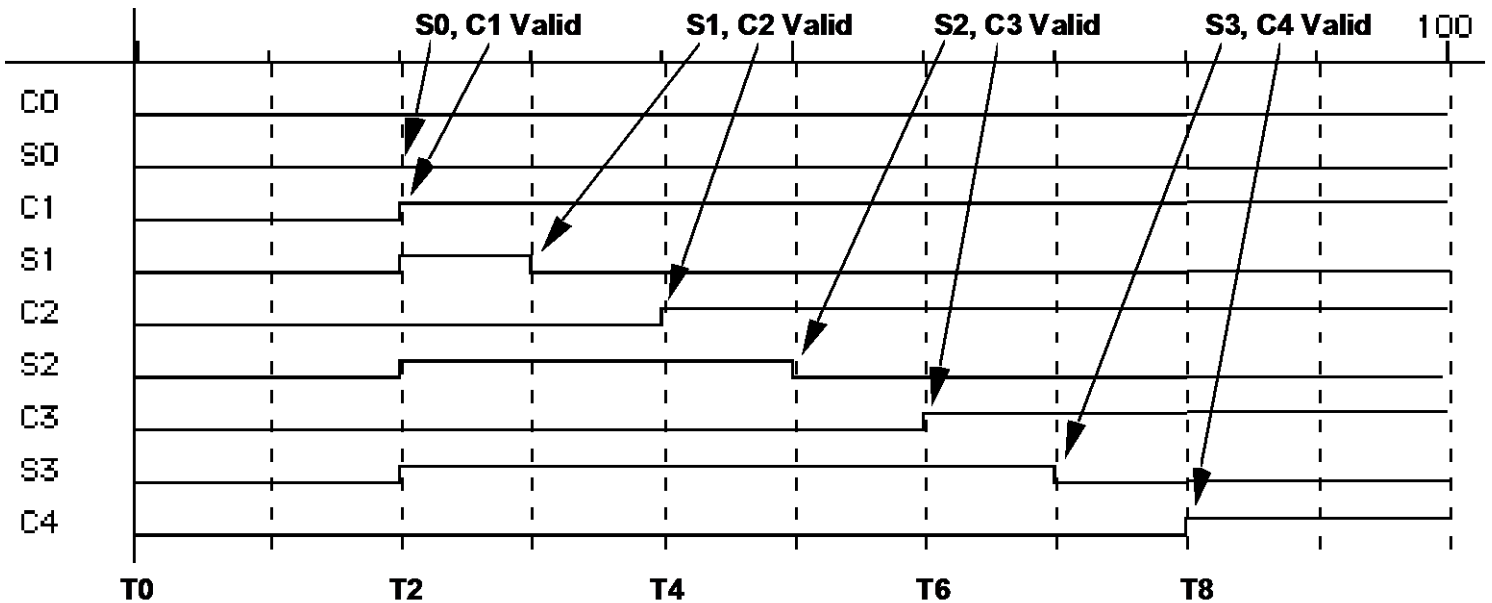




The next advanced adder is a ripple-carry adder. What is the problem of the full adder in the previous slide? When we consider data bits, they are coming in parallel at the same time. However, carries should be cascaded with delays proportional to the bit positions. On the left, Cin comes at time N, then Cout will be valid at time N+2 (each one-bit adder incurs 2 gate delays)

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



This slide shows a waveform of a 4 bit adder. Suppose we add 1111 and 0001, the carry will propagate from the LSB all the way through to the final carry-out.

Carry-lookahead logic

- **Carry generate: Gi = Ai Bi**
  - must generate carry when A = B = 1
- **Carry propagate: Pi = Ai xor Bi**
  - carry-in will equal carry-out here
- **Sum and Cout can be re-expressed in terms of generate/propagate:**
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - C_{i+1} = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Ci Pi

Instead of awaiting the carry from the lower-order bits, we can process the carry in advance. How?

In order to do so, the carry should be derived from the lower-order data bits directly. First of all, let's look at two new functions: Gi and Pi. Then Si and C_{i+1} can be rewritten as shown in the above. Actually, those imply two cases. When Gi is true, there is always a carry-out. If Pi is true, a carry-out depends on a carry-in.

Carry-lookahead logic (cont'd)

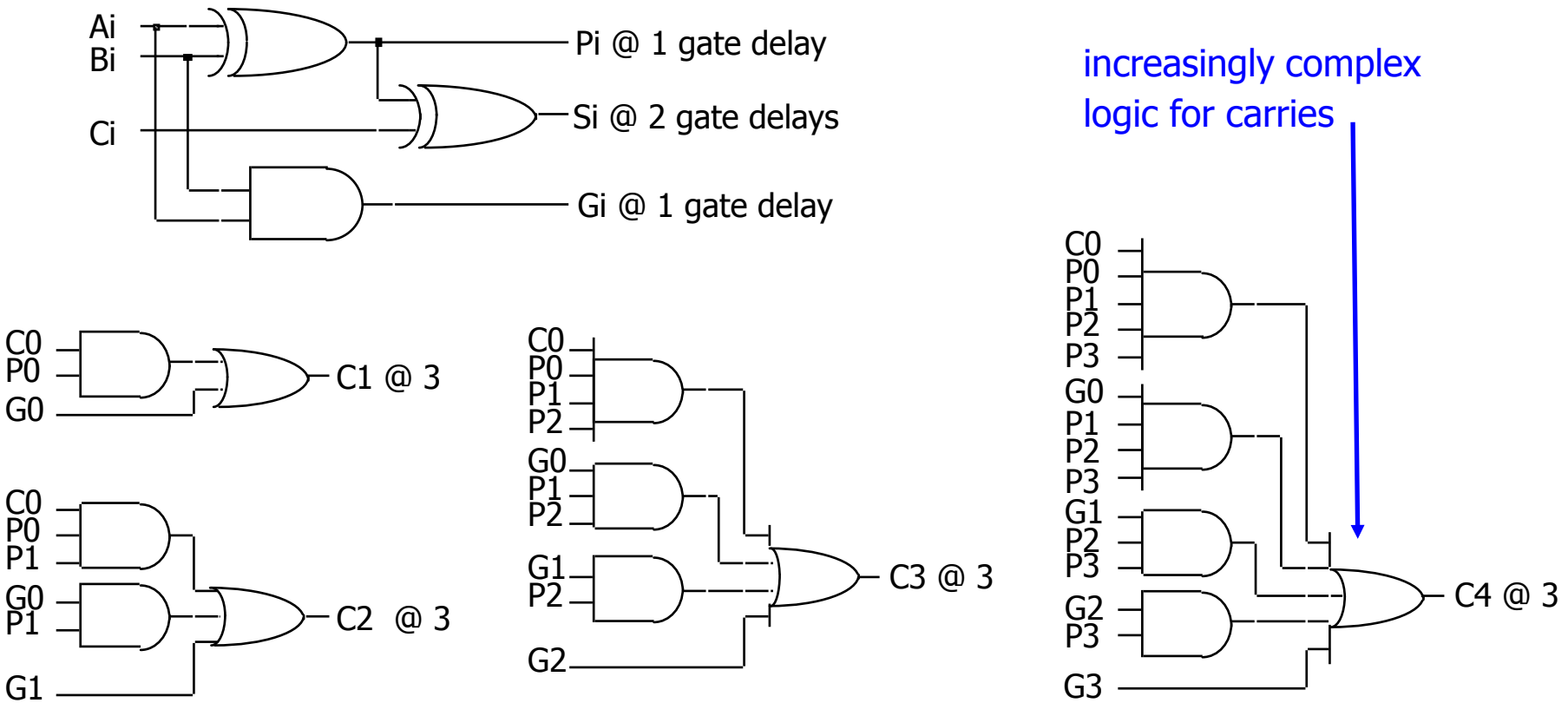
- **Re-express the carry logic as follows:**
  - C1 = G0 + P0 C0
  - C2 = G1 + P1 C1 = G1 + P1 G0 + P1 P0 C0
  - C3 = G2 + P2 C2 = G2 + P2 G1 + P2 P1 G0 + P2 P1 P0 C0
  - C4 = G3 + P3 C3 = G3 + P3 G2 + P3 P2 G1 + P3 P2 P1 G0 + P3 P2 P1 P0 C0

- **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

Then all the carries are calculated by the expanding Pi and Gi as shown in the above, which is called the carry-lookahead logic. Note that all carries are now just two-level logic functions. So here is the bottom line: there is a tradeoff between the number of gates and the delay.

Carry-lookahead (CLA) implementation

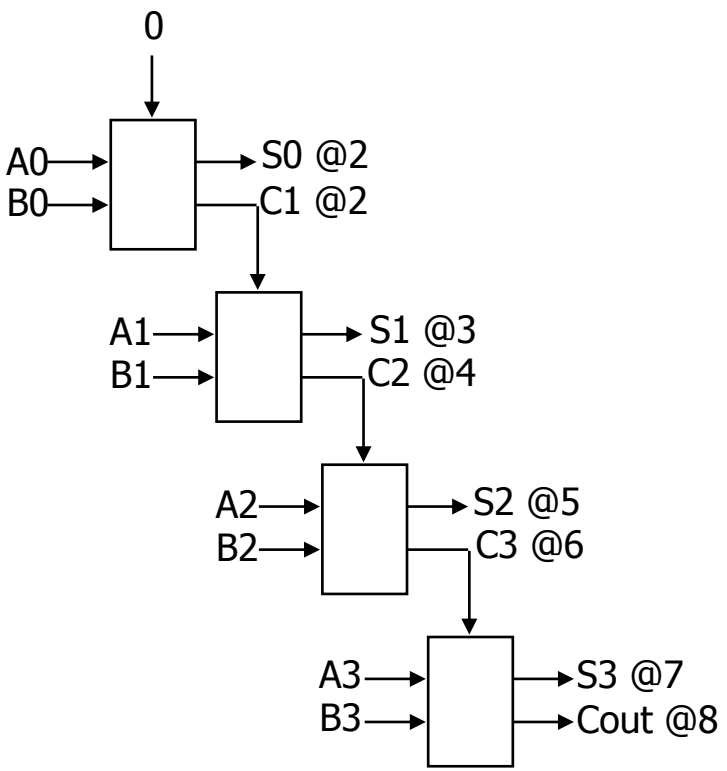
- **Adder with propagate and generate outputs**

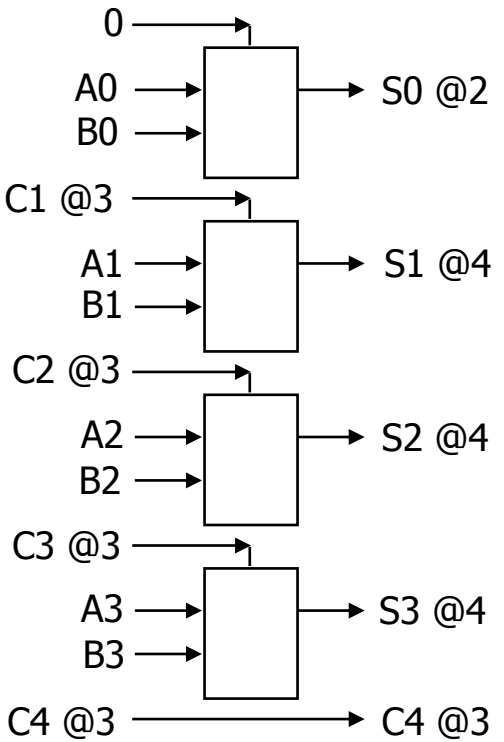


This slide shows how 4 carries are implemented in two level fixed logic based on Pi and Gi. As Pi and Gi take only one gate delay, the final carries take maximum 3 gate delays.

CLA 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

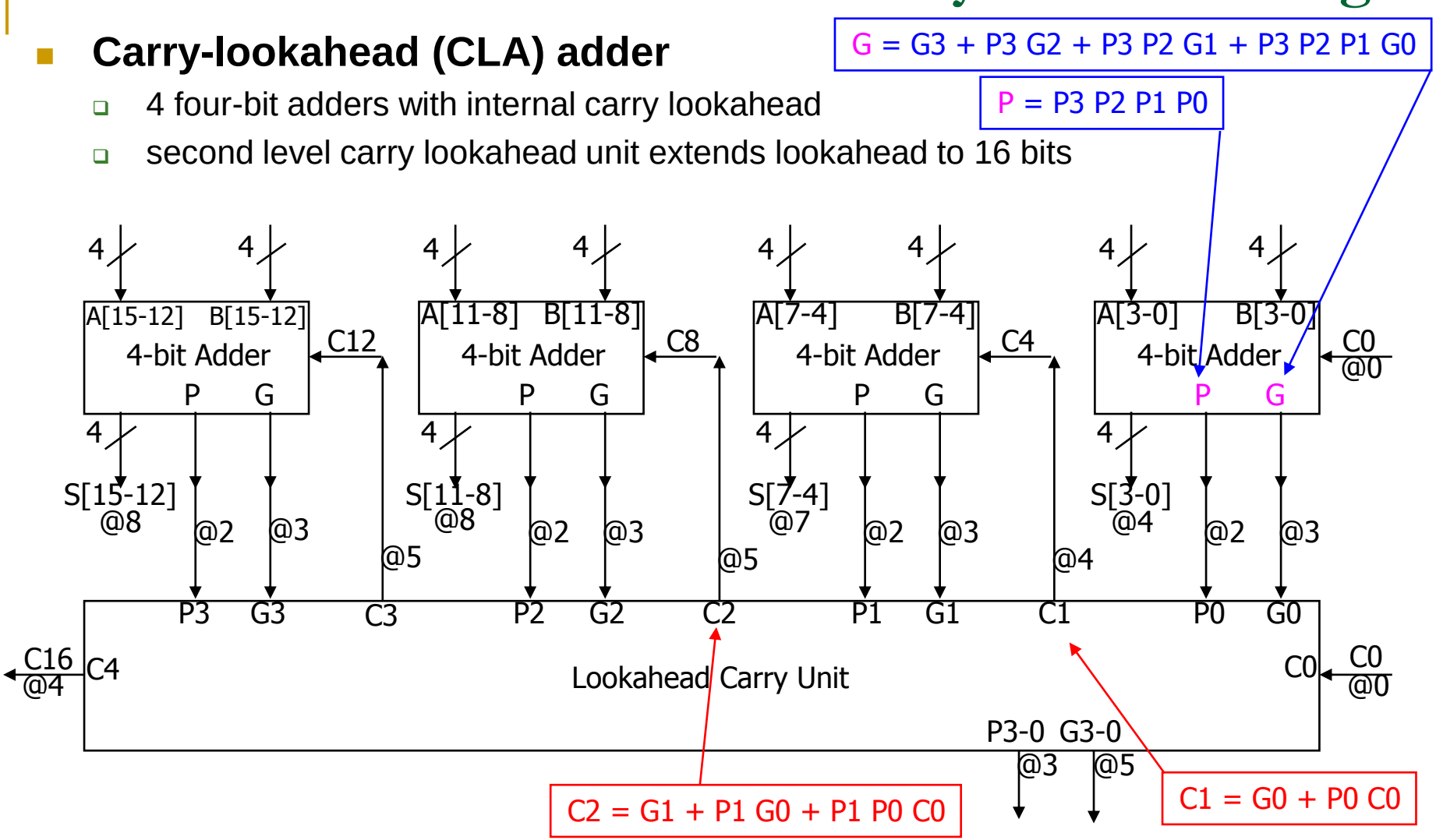




on the left, there is the original full adder, which takes a long time to finish addition. On the right, there is the carry lookahead adder. The two level logic functions for carries are not shown. Each box is the one-bit adder module. Overall, by using a lot of gates for CLA in the previous slide, we can reduce the delay of the addition process.

16bit CLA adder with cascaded carry-lookahead logic

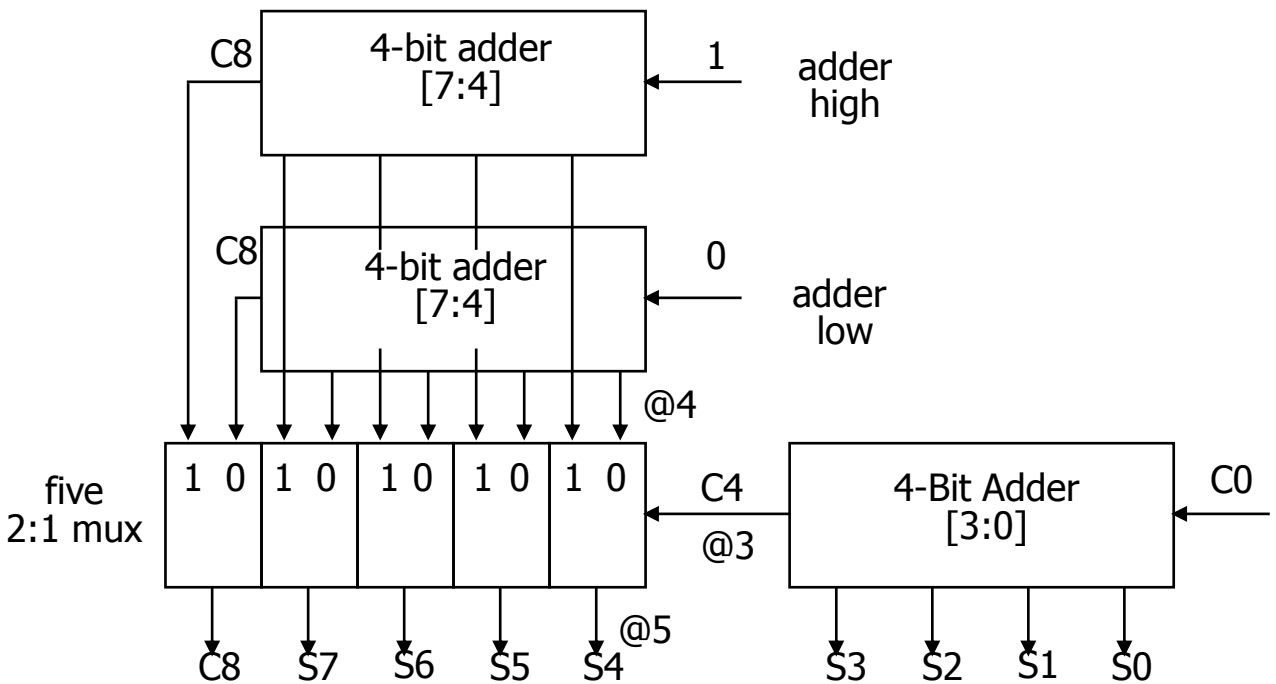
- **Carry-lookahead (CLA) adder**
  - 4 four-bit adders with internal carry lookahead
  - second level carry lookahead unit extends lookahead to 16 bits



If we add 16bit long numbers, we have to use 4 4bit CLA adder modules. Each module adds 4 bits. Here Ci at the bottom box is the carry from the i-th 4bit CLA adder module. Pi and Gi in blue color are the same functions in the previous example. While Pi and Gi at the bottom unit give info about the carries between the 4bit CLA modules, which is more complicated.

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



The next version, a carry-select adder, achieves even lower delay by redundant h/w. Note that the carry between 4bit CLA adders is either 0 or 1. We use two adder modules for high 4 bit of the 8-bit adder system. So we perform addition for both cases and then the carry from the lower 4bit CLA module will perform selection (MUX).

Arithmetic logic unit (ALU) design specification

M = 0, logical bitwise operations

S1	S0	Function	Comment
0	0	Fi = Ai	input Ai transferred to output
0	1	Fi = not Ai	complement of Ai transferred to output
1	0	Fi = Ai xor Bi	compute XOR of Ai, Bi
1	1	Fi = Ai xnor Bi	compute XNOR of Ai, Bi

M = 1, C0 = 0, arithmetic operations

0	0	F = A	input A passed to output
0	1	F = not A	complement of A passed to output
1	0	F = A plus B	sum of A and B
1	1	F = (not A) plus B	sum of B and complement of A

M = 1, C0 = 1, arithmetic operations

0	0	F = A plus 1	increment A
0	1	F = (not A) plus 1	twos complement of A
1	0	F = A plus B plus 1	increment sum of A and B
1	1	F = (not A) plus B plus 1	B minus A

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

An ALU is a key subsystem of computers that performs logic and arithmetic functions. There are one high level control input M, and two lower level selection inputs S1 and S0. Here Ci is the carry value which is useful only for some arithmetic functions.

Arithmetic logic unit (ALU) design (cont'd)

- **Sample ALU – truth table**

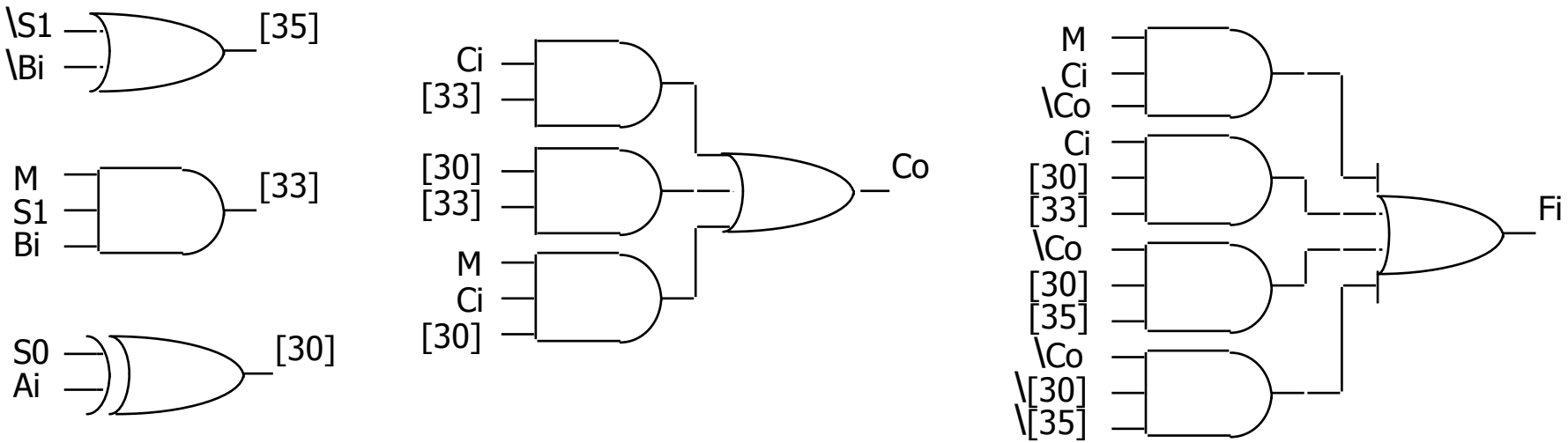
M	S1	S0	Ci	Ai	Bi	Fi	Ci+1
0	0	0	X	0	X	0	X
			X	1	X	1	X
	0	1	X	0	X	1	X
			X	1	X	0	X
	1	0	X	0	0	0	X
			X	0	1	1	X
			X	1	0	1	X
			X	1	1	0	X
	1	1	X	0	0	1	X
			X	0	1	0	X
			X	1	0	0	X
			X	1	1	1	X
1	0	0	0	0	X	0	X
			0	1	X	1	X
	0	1	0	0	X	1	X
			0	1	X	0	X
	1	0	0	0	0	0	0
			0	0	1	1	0
			0	1	0	1	0
			0	1	1	0	1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

First of all, there are three control inputs, three data inputs, and two data outputs.

If M is 0, Ci is a DC term. Sometimes, Bi is also a DC term when only the 1st input, Ai, matters

ALU design (cont'd)

- **Sample ALU – multi-level discrete gate logic implementation**



Total 12 gates + 5 inverters

If we implement the truth table in the previous slide by a random logic with some minimization techniques, we can get the above multi-level logic system. Don't worry; there are six variables. Humans are not supposed to do that. This result comes from a CAD tool. Numbers in [] are internal wires.

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

We looked at the design process of several combinational logic circuits and then reviewed how binary numbers are represented. Arithmetic functions are also discussed.