Combinational logic design case studies

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

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 look-ahead logic
- Arithmetic and logic units
  - general-purpose building blocks
  - critical components of processor data-paths
  - used within most computer instructions

Number systems

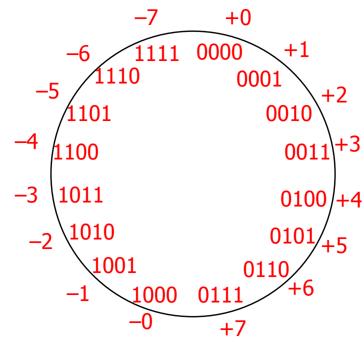
- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
  - sign and magnitude
  - 1s complement
  - 2s complement
- Assumptions
  - we'll assume a 4 bit machine word
  - 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
  - +/– 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

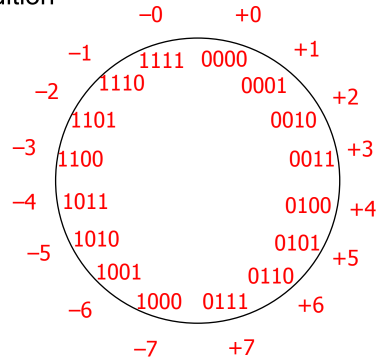


1s complement

- 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

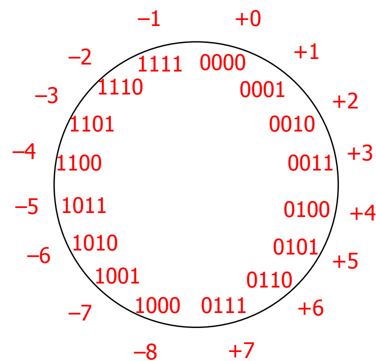


2s complement

- 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

$$\begin{array}{lrcr} & 2^4 & = & 10000 \\ \text{subtract} & 7 & = & \underline{0111} \\ & & & 1001 \end{array} = \text{repr. of } -7$$

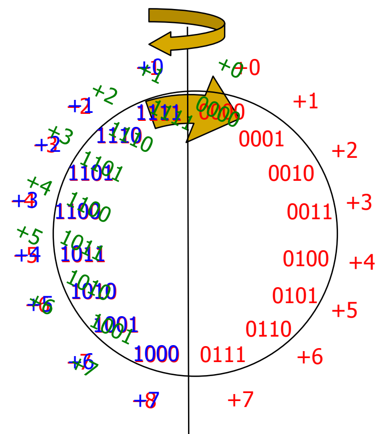
  - example: 2s complement of –7

$$\begin{array}{lrcr} & 2^4 & = & 10000 \\ \text{subtract} & -7 & = & \underline{1001} \\ & & & 0111 \end{array} = \text{repr. of } 7$$

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

2s complement (cont’d)

- Why does bit-wise complement + 1 work?



2s complement addition and subtraction

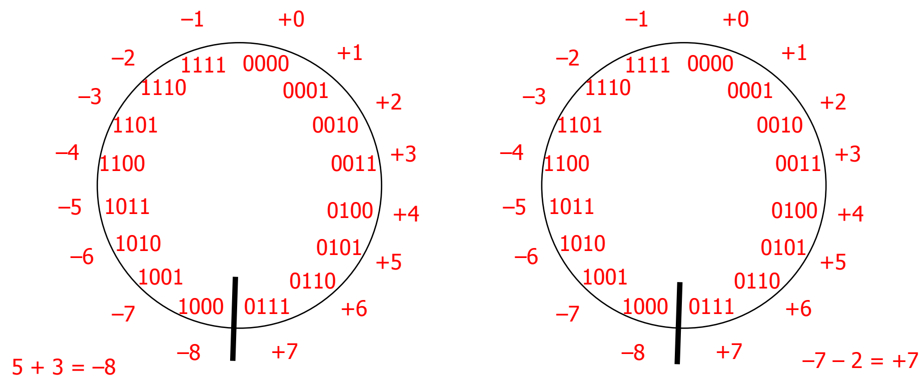
- Simple addition and subtraction
  - simple scheme makes 2s complement the unanimous choice for integer number systems in computers

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

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

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
  - A3’ B3’ S3 + A3 B3 S3’
- 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
3	0 0 1 1	2	0 0 1 0	– 2	1 1 1 0	– 5	1 0 1 1
– 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

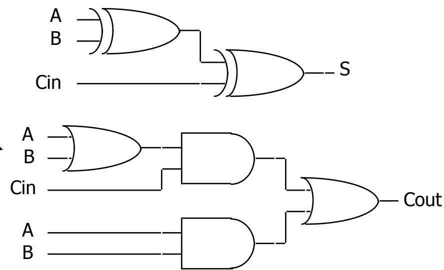
- Half adder (add 2 1-bit numbers)
  - Sum = Ai' Bi + Ai Bi' = Ai xor Bi
  - Cout = Ai Bi
- Full adder (carry-in to cascade for multi-bit adders)
  - Sum = Cin xor A xor B
  - Cout = B Cin + A Cin + A B = Cin (A + B) + A B

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

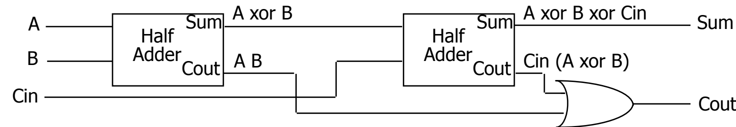
Full adder implementations

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



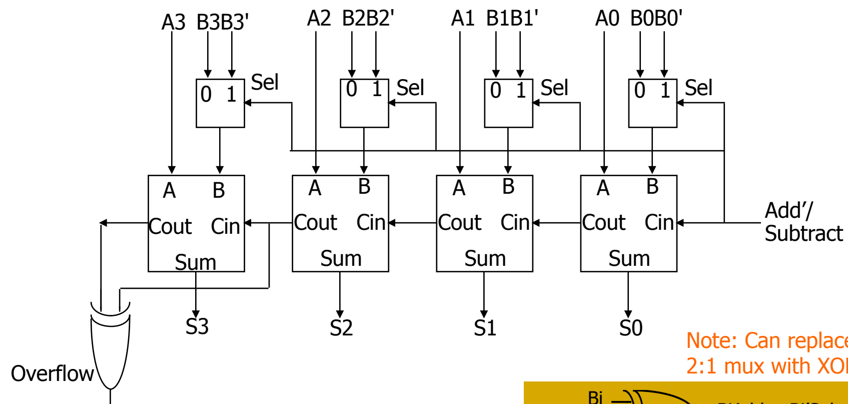
- Alternative implementation
  - 5 gates
  - half adder is an XOR gate and AND gate
  - 2 XORs, 2 ANDs, 1 OR

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



Adder/subtractor

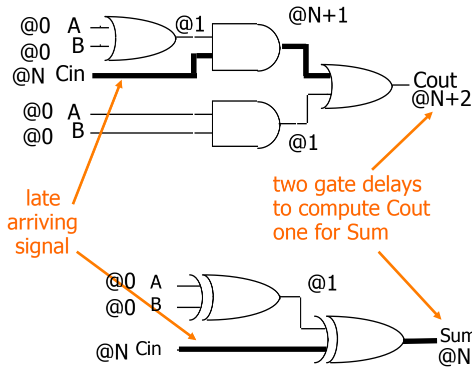
- Use an adder to do subtraction thanks to 2s complement representation
  - A – B = A + (– B) = A + B' + 1
  - control signal selects B or 2s complement of B

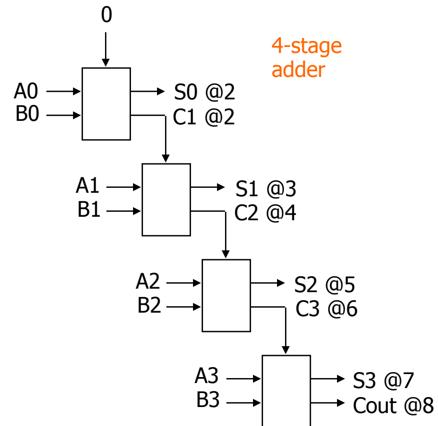


Note: Can replace 2:1 mux with XOR gate

Ripple-carry adders

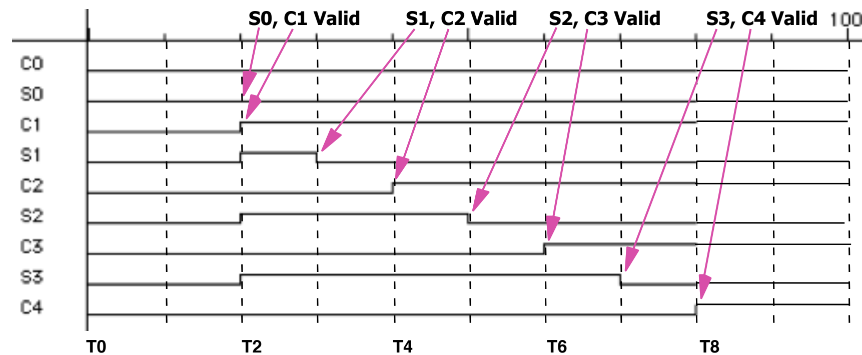
- Critical delay
  - the propagation of carry from low to high order stages





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: Gi = Ai Bi
  - must generate carry when A = B = 1
- Carry propagate: Pi = Ai xor Bi
  - carry-in will equal carry-out in these two cases: A = 0, B = 1 or A = 1, B = 0
- Carry kill: Ki = Ai'Bi'
  - carry-out will be zero no matter what carry-in when A = B = 0
- Gi + Pi + Ki = 1
- Sum and Cout can be re-expressed in terms of generate/propagate (or in terms of generate/kill):
  - Si = Ai xor Bi xor Ci
    = Pi xor Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai + Bi)
    = Ai Bi + Ci (Ai xor Bi)
    = Gi + Pi Ci

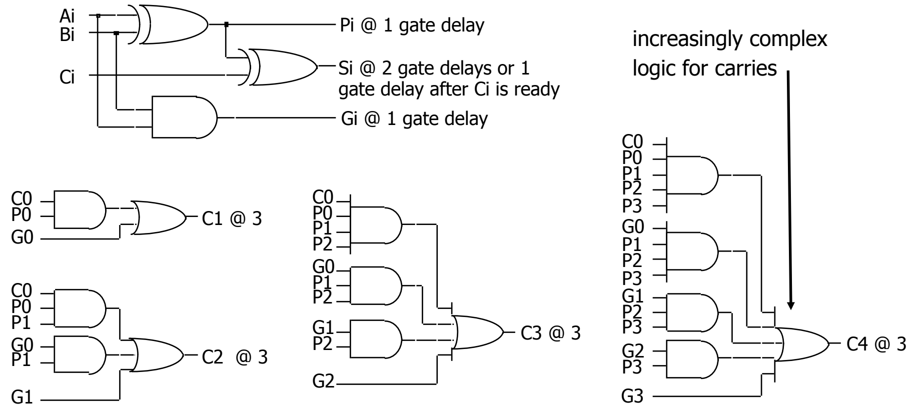
    Ci+1' = Ki + Pi Ci'
    Ci+1 = Ki' (Pi' + Ci)

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

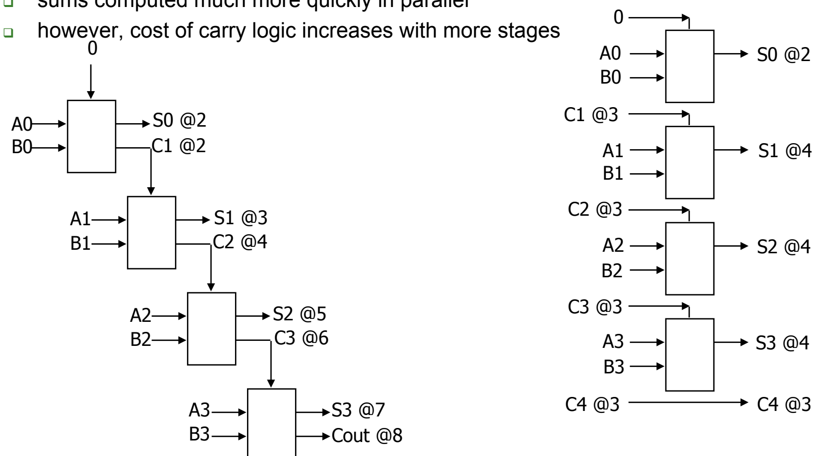
Carry-lookahead implementation

- Adder with propagate and generate outputs



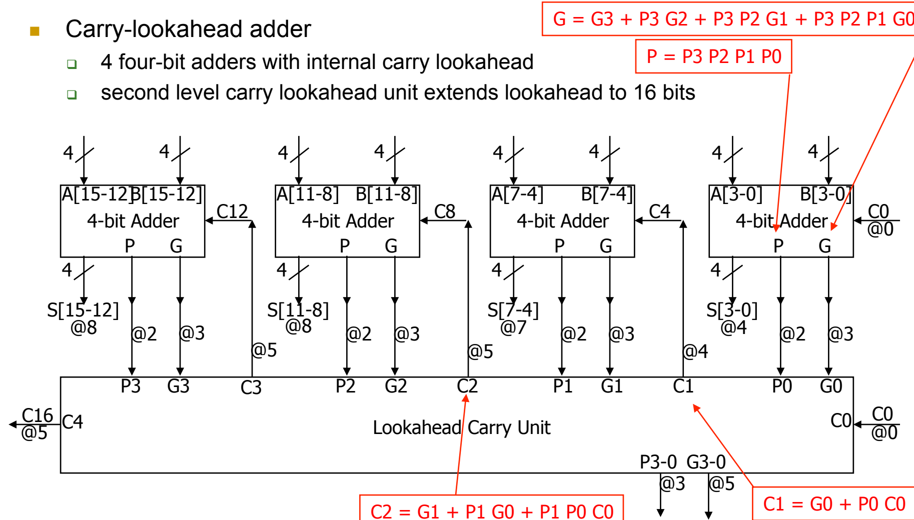
Carry-lookahead implementation (cont'd)

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



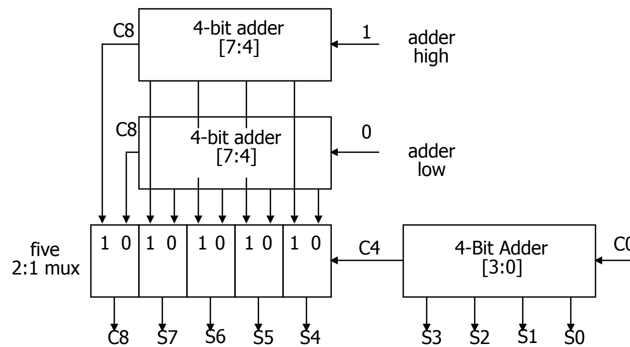
16-bit carry-lookahead adder with cascaded carry-lookahead logic

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



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



Scaling of carry-select adders

- Size: roughly twice the size of a ripple-carry
  Delay: delay through a 4-bit ripple-carry plus the multiplexor path highlighted in blue (3 2-1 multiplexors, in this example)
- We can do better – how?