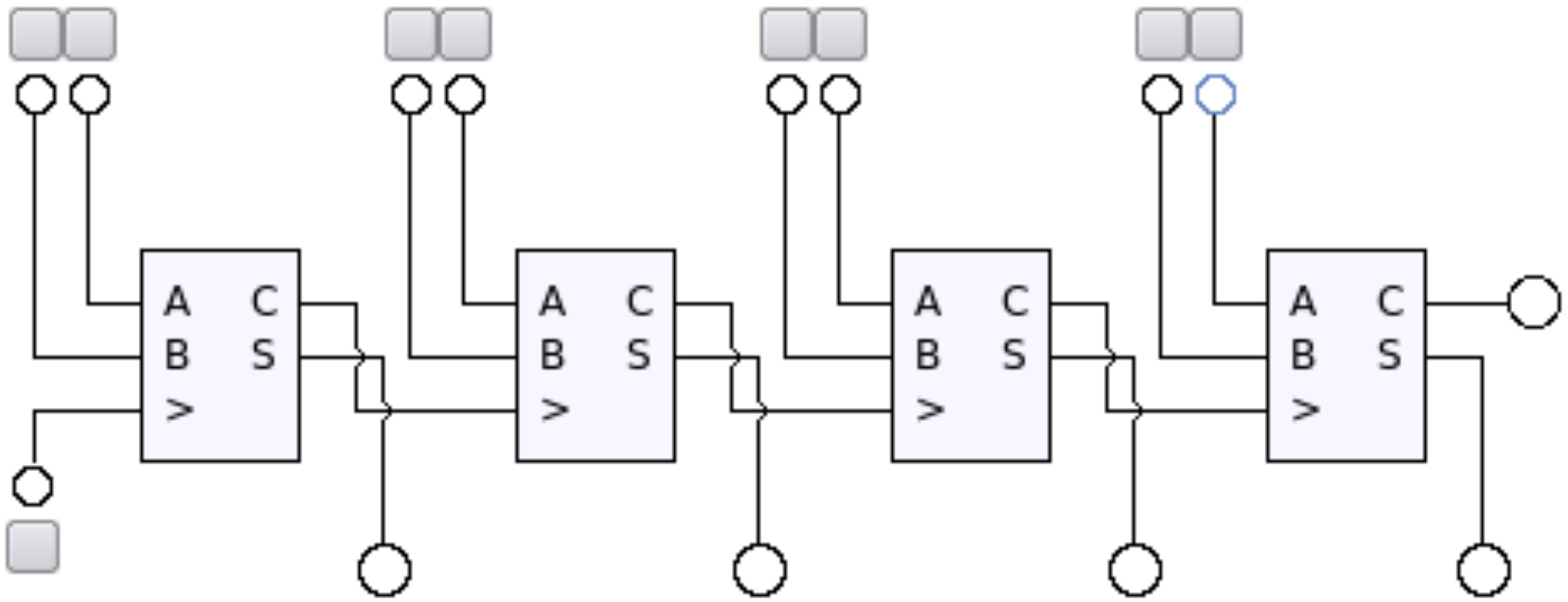
Ripple Adder

- e.g., 4-bit adder (MSB is right)



Signal Delay

- Ripple Adder slow – carry needs to propagate
- linear in number of bits

- speed up: Carry Lookahead Adder
  - extra (more complex) circuits to determine carry
  - gates can switch in parallel
  - hierarchical application

Carry Lookahead

- determine carry bits in parallel to main addition
- unroll sequential computation
- AND and OR can have more than 2 inputs

- basic observation about bit pair A,B:
  - Carry Generate: $G(A,B) = A \wedge B$
  - Carry Propagate: $P(A,B) = A \vee B$

•4-Bit Carry Computation

- input: $A_0 \ldots A_3$, $B_0 \ldots B_3$, C_0
- output: $C_1 \ldots C_4$
- intermediate: $G_0 \ldots G_3$, $P_0 \ldots P_3$

$$C_1 = G_0 \vee P_0 \wedge C_0$$

$$C_2 = G_1 \vee P_1 \wedge C_1 = G_1 \vee P_1 \wedge G_0 \vee P_1 \wedge P_0 \wedge C_0$$

$$C_3 = G_2 \vee P_2 \wedge C_2 = \ldots$$

$$C_4 = G_3 \vee P_3 \wedge C_3 = \ldots$$

4-Bit Carry Computation

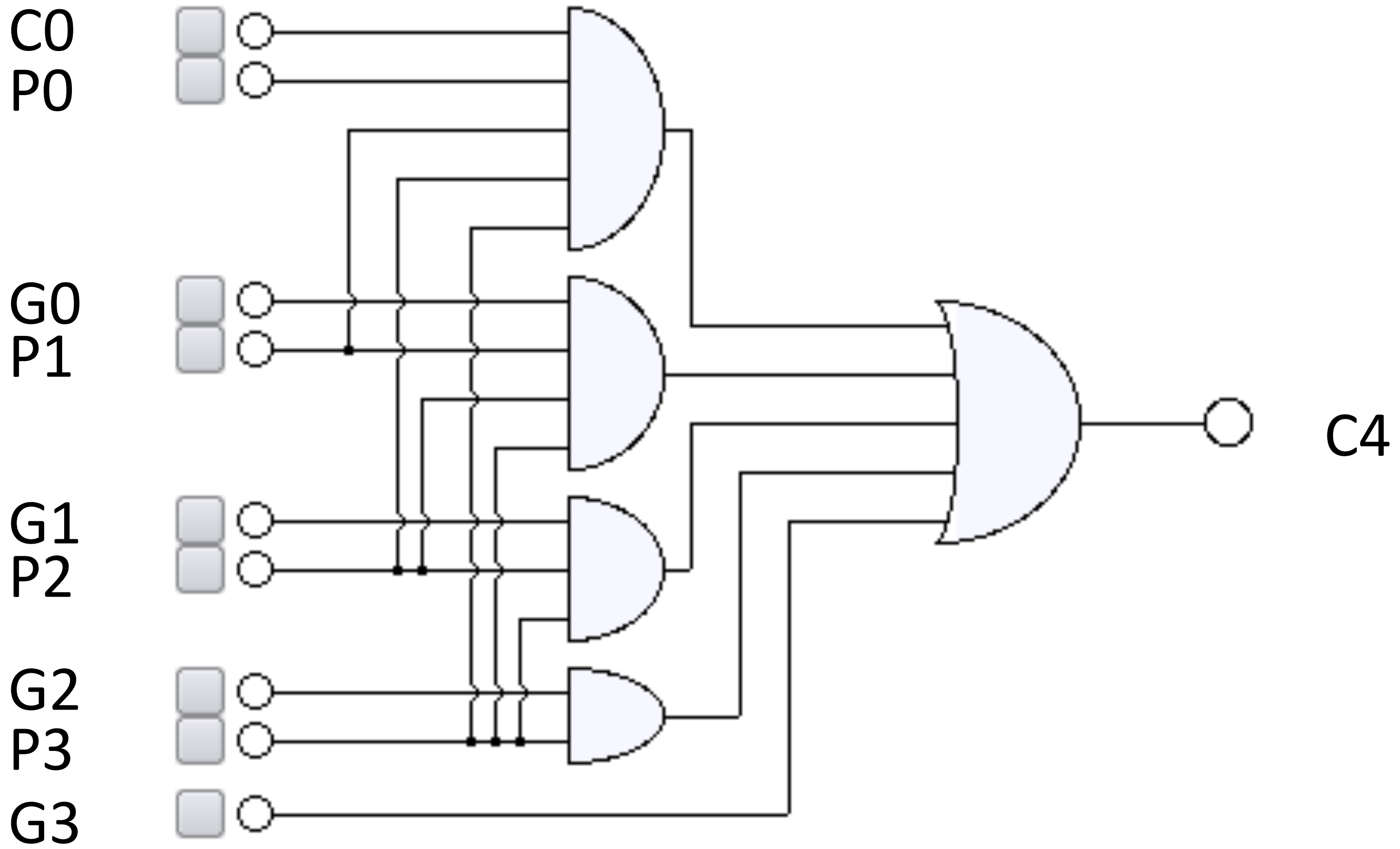
- $C_4 = G_3 \vee$
  $P_3 \wedge G_2 \vee$
  $P_3 \wedge P_2 \wedge G_1 \vee$
  $P_3 \wedge P_2 \wedge P_1 \wedge G_0 \vee$
  $P_3 \wedge P_2 \wedge P_1 \wedge P_1 \wedge C_0$

- with

- $G_x = A_x \wedge B_x$

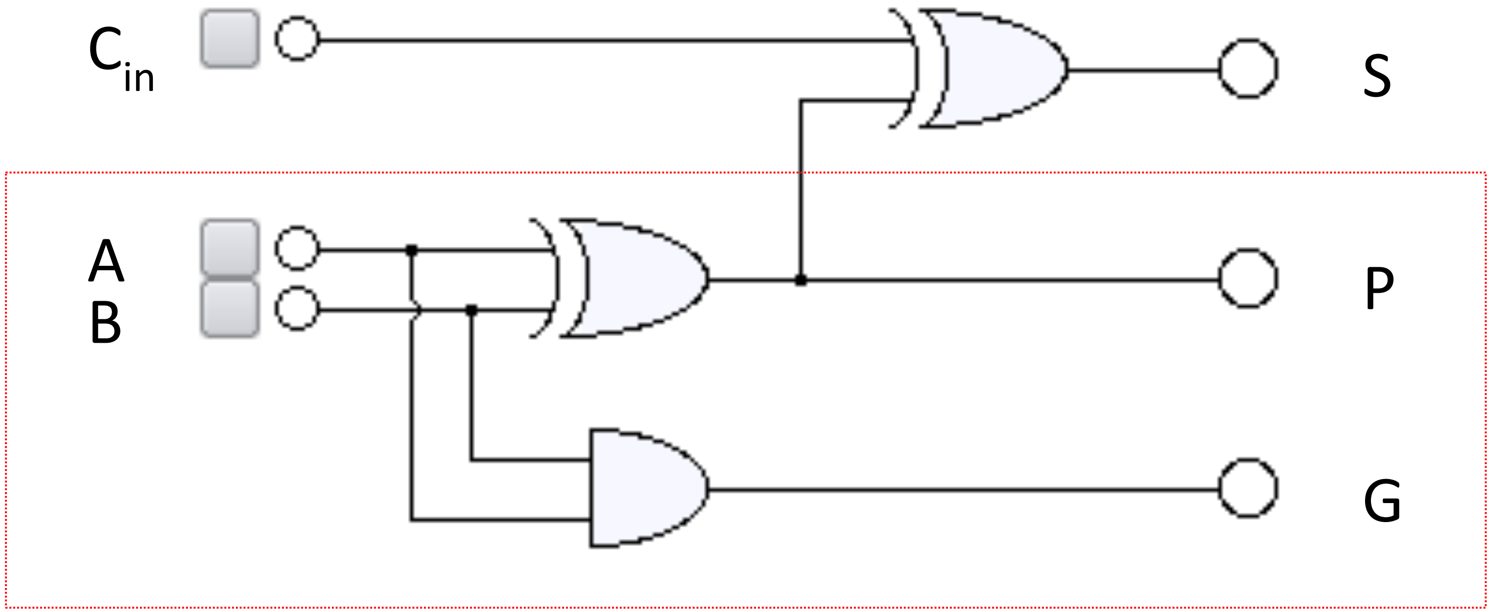
- $P_x = A_x \vee B_x$

Computing C_4



Partial Full Adder

- Note: $(A \wedge B) \vee (A \vee B) = (A \wedge B) \vee (A \oplus B)$
- $\Rightarrow$ Can use $P = A \oplus B$



Carry Lookahead Adder

- Step 1: Compute all P_i, G_i
- Step 2: Compute all C_i
- Step 3: Compute all S_i

- in practice: limited to 4 bits
  - scheme can be used recursively/hierarchical

Principles

- sequential execution
  - turned into parallel execution
  - trade-off: number of gates vs. speed

Basic Memory Circuit – Flip-flop gate

- stored bit is at Q;

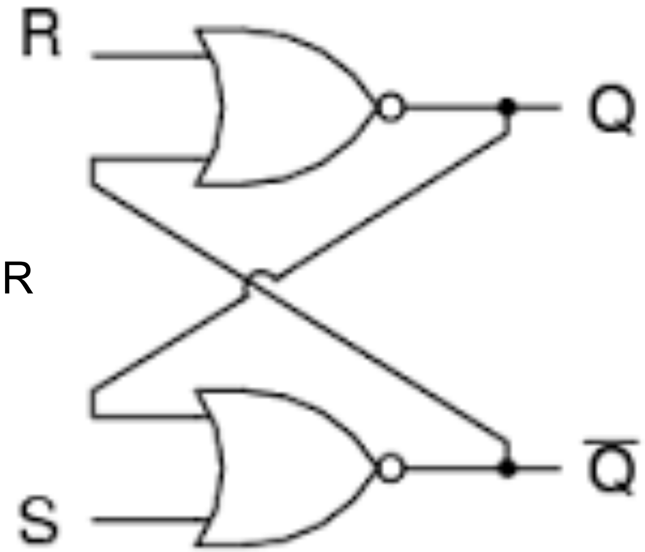
One of the fastest ways that we can store 1-bit (and more) of data

- Output depends on the inputs *and* the previous output (stored bit)

Input: R and NOT Q

1. Start from S NOR Q to get NOT Q
2. Use the value of NOT Q to get Q_next = R NOR (NOT Q)

Input: S and Q



We want a way to:
- hold the value at Q
- reset the value at Q to 0
- set the value at Q to 1

Q_next is the next value of Q, i.e. the output of the circuit in the next instance after something has been changed (set, reset, hold)

- S – set, R – reset

Revisit Binary Addition

- If you are building circuits to handle the data, then you have a limit to the number of bits available to represent values
- fixed width n-bit representation: *overflow*
  - modular arithmetic
  - 4 bits: 14 + 4 = 2

Exceeding the n-bit representation,
e.g 100 + 111 + 011

Sign Representation

- fixed width n-bit representation
  - most significant bit: left-most (highest value)
  - least significant bit: right-most (lowest value)
- sign extension: treat MSB as sign
  - 0 means positive, 1 means negative

e.g. 110 -> -2, 010 -> +2

- two zeros: 0000 and 1000
- cannot use basic addition

Sign extension can lead to this

  - e.g. 3-1 = -4 ??

Ones' Complement

- negative number: invert bits
- still two zeros: 0000 and 1111
- addition possible
  - add carry-over to sum

Arithmetic

```
•    00001101      13
• +  11111011    -  4
• = 100001000       8 ?
• +         1
• =  00001001       9
```

Two's Complement

- negative number: invert bits and add 1
- single zero: 0000
- range: $-2^{n-1} \ldots 2^{n-1} - 1$
- straightforward addition

Arithmetic

- `   00001101      13`
- `+ 11111100    - 4`
- `=100001001       9`

- ignore carry over

Overflow

- assume 8-bit integers in two's complement
- 100 + 50 - 25 = ?
- 100 + (50 - 25) = 100 + 25 = 125
- (100 + 50) - 25 = -106 - 25 = -131 = 125

More Arithmetic

- addition, subtraction – done
- multiplication, division? – *Not In Course*
- integer vs. fraction?

Shift Operations

- shift bitstring to left or right
  - with or without carry-over
  - simplest: no carry-over
- equivalent to multiplication/division by 2
- very fast machine instructions
- programming languages, operators << and >>
  - a << b $a * 2^{b}$
  - a >> b $a / 2^{b}$