ECE 385 – Digital Systems Laboratory

Lecture 5 – Lab 4 Introduction

Zuofu Cheng

Spring 2019

Link to Course Website

ECE ILLINOIS

ILLINOIS

Procedures

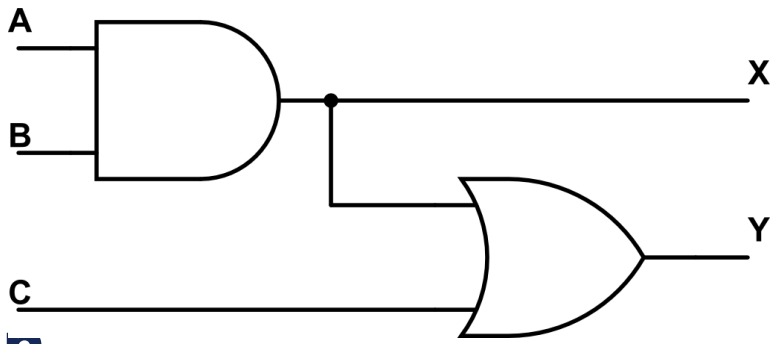
- **Procedure blocks** describe the behavior of an entity or its structure (gates, wires, etc.) using SystemVerilog constructs. There are three types of procedure blocks in SystemVerilog, but we will generally only use the two from Verilog (initial and always, optionally look at fork-join)
- **`initial`:** an initial block is used to initialize testbench assignments and is carried out only once at the beginning of the execution at time zero. It executes all the statements within the "begin" and the "end" statement without waiting. **Non-synthesizable (simulation ONLY)**
- **`always`:** an always block described hardware which operates **under certain conditions (e.g. a clock edge)**. Typically we will use two specialized always blocks:
  - **`always_comb`** : forces synthesizable combinational logic behavior
  - **`always_ff`** : forces synthesizable sequential logic behavior
  - **`always_latch`** : forces synthesizable latch behavior (avoid this!)
- Always think of and partition your modules in terms of combinational and sequential logic (note that this is the underlying model for the FPGA)
- You might encounter "sensitivity list" in Verilog code, this is necessary in Verilog to limit scope of simulator (to reduce CPU power required for simulation). This is unnecessary in SV if you use `always_comb` and `always_ff` blocks (introduced in SV)

Assignment Inside `Always` Procedure

- Two types of assign function inside `always` procedure
  - = blocking assignment (a = b) (sequential assignments)
  - <= non-blocking assignment (a <= b) (simultaneous assignments)
- General rules
  - Always use blocking for combinational logic
  - Always use non-blocking for sequential logic
- Using wrong assignment sometimes creates same result in synthesis but different results in simulation (different behavior between synthesis and simulation = bad)
- Why? Let's take a look at a couple of examples
- For detailed explanation, see: Cummings, C. E. (2000). Nonblocking assignments in Verilog synthesis, coding styles that kill!. *SNUG (Synopsys Users Group) 2000 User Papers*.

Combinational `Always` Procedure

- Consider function to right
  - X = A•B
  - Y = X + C



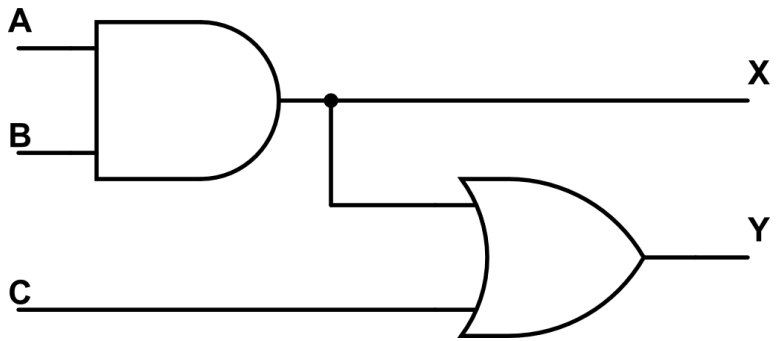
- Note: Combinational function
- Correct way to write (inside `always_comb` procedure)

```
X = A & B;
Y = X | C;
```

- Suppose A = 1, B = 1, C = 0 (initially) and A changes to 0
  - What is expected value for X and Y

Combinational `Always` Procedure

A	B	C	X	Y
1	1	0	1	1
0	1	0	1	1



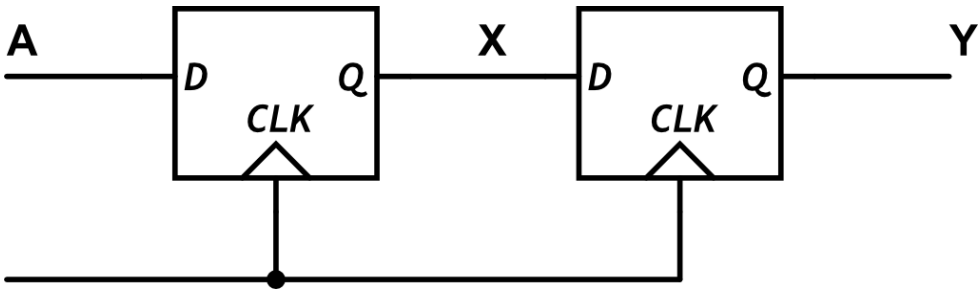
- **Incorrect way to write**

```
X <= A & B;
Y <= X | C;
```

- **Suppose A = 1, B = 1, C = 0 (initially) and A changes to 0**
  - **What is value for X and Y if we use non-blocking assignments?**
- **Simulation will have different result than synthesis**

Sequential `Always` Procedure

- **Sequential Circuit Example**
  - **Chained flip-flops**
  - **Shift Register or digital filter**



- **Correct way to write (with non-blocking assignment)**

```
always_ff @ (posedge clk) begin
    X <= A;
    Y <= X;
end
```

- **What happens (in simulation) if we try to use blocking assignment?**

Assign Primitive

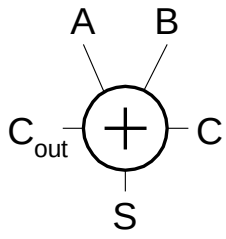
- For the most part all circuit behavior is defined by code inside `always` procedure
- Exception: Assign primitive can be used to describe simple combinational logic **outside of `always` procedure**
- `logic x; assign x = a & b;`
- Useful for simple modules or debugging (for example, fix a given signal to logic '0' to test
- Anything described using `assign` can also be done using **`always_comb`**

Full Adder

- Basic building block from ECE 120
- Inputs (A, B, C)
- Outputs (S, C_{out})
- Fully combinational logic
- Note: MAJ means majority function
  - OR of pairwise AND combinations
  - MAJ(A, B, C) = (AB)+(AC)+(BC)

$$S = A \oplus B \oplus C$$

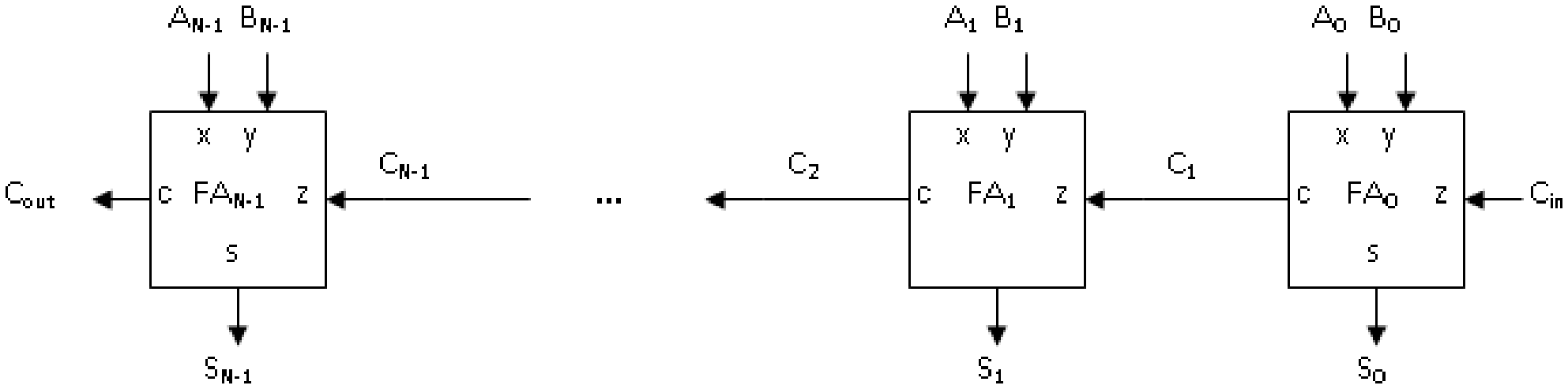
$$C_{out} = MAJ(A, B, C)$$



A	B	C	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

N-bit Carry Ripple Adder

- Simplest design: cascade full adders
  - Critical path goes from C_{in} to C_{out}
  - This critical path limits maximum operating frequency
  - Design full adder to have fast carry delay (fewest gate levels for carry)

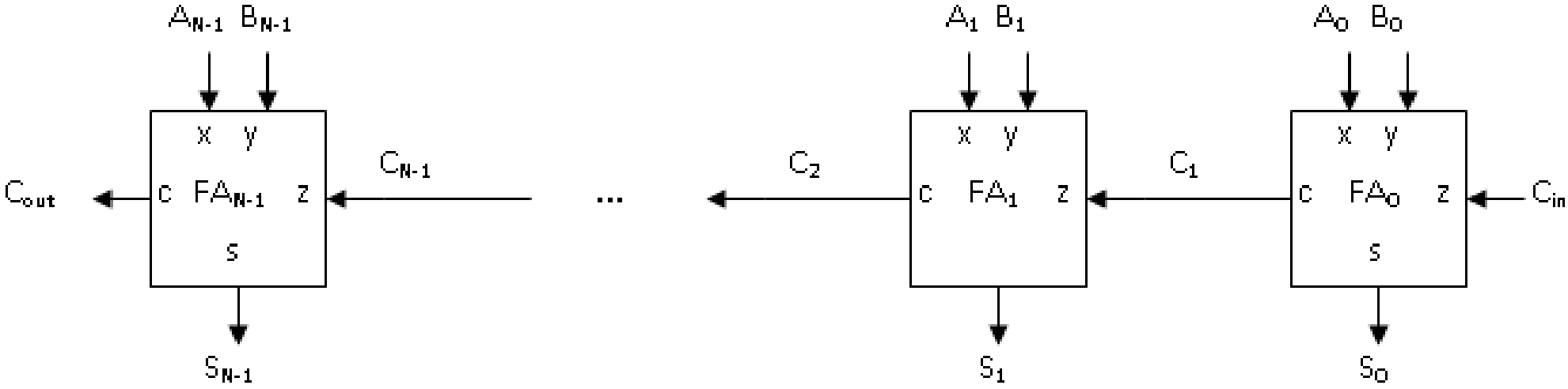


4-bit Carry Ripple Adder with SystemVerilog

- Wire the carry out of each bit into previous carry in

```
module full_adder  (input  x, y, z,
                    output s, c     );
     assign s = x^y^z;
     assign c = (x&y)|(y&z)|(x&z);
endmodule

module ADDER4 (input   [3:0] A, B,
               input         c_in,
               output  [3:0] S,
               output        c_out);

logic       c1, c2, c3;

full_adder  FA0(.x(A[0]), .y(B[0]), .z(c_in), .s(S[0]), .c(c1));
full_adder  FA1(.x(A[1]), .y(B[1]), .z(c1), .s(S[1]), .c(c2));
full_adder  FA2(.x(A[2]), .y(B[2]), .z(c2), .s(S[2]), .c(c3));
full_adder  FA3(.x(A[3]), .y(B[3]), .z(c3), .s(S[3]), .c(c_out));

endmodule
```

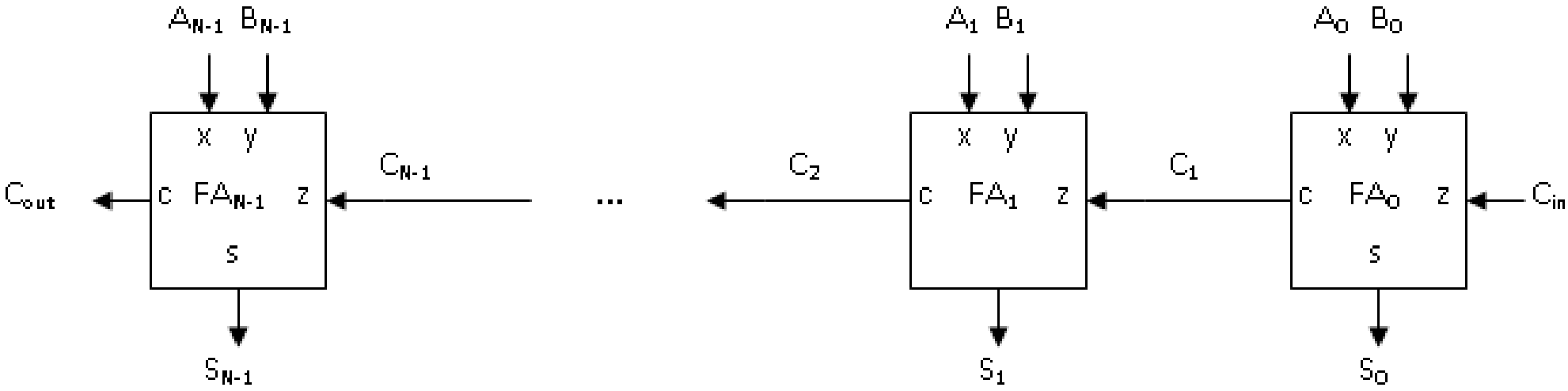
Carry Ripple Adder Speed



Carry Look-ahead Adder – P/G Bits

- For a full adder, define what happens to carries
- There are two possibilities at each bit (depending on A & B)
  - Generate : C_{out} = 1 independent of C
    - This happens when both A & B are 1
    - G = A • B
  - Propagate: C_{out} = C
    - This happens when either A or B are 1 (both not both)
    - P = A ⊕ B
  - Note: P_i, G_i may be calculated for each bit independent of previous bits
  - This implies all P_i, G_i may be calculated in parallel

Carry Look-ahead Generator



Carry Look-ahead Generator

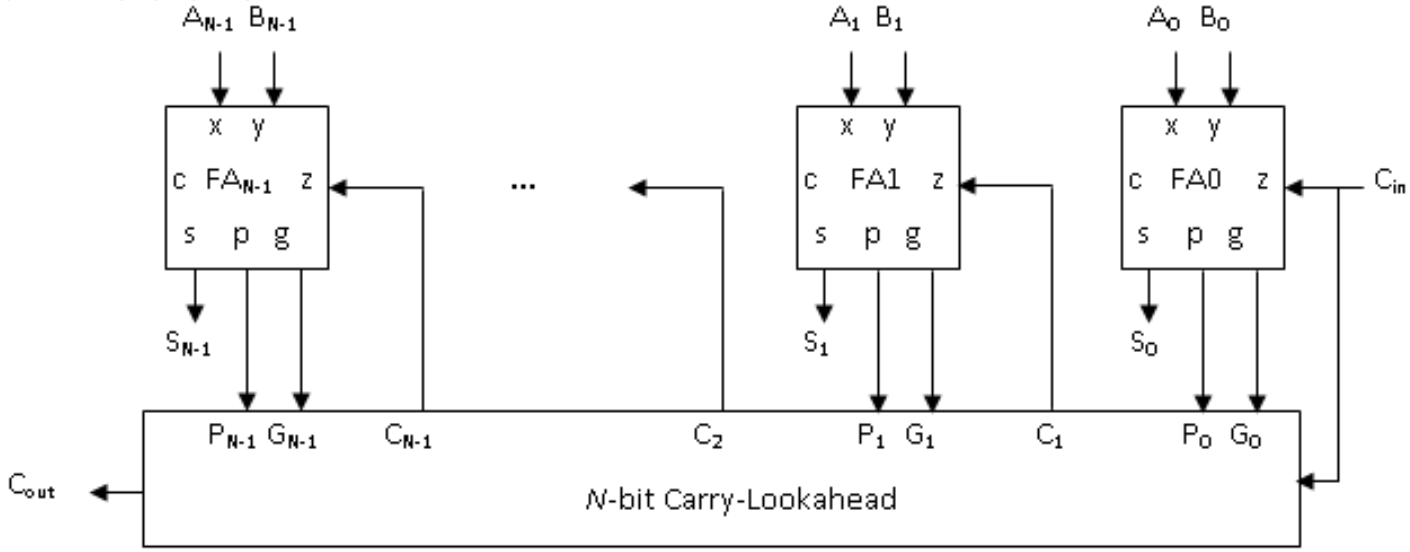
- Carry look-ahead generator computes C_i using **only** $P_{i-1...0}$ and $G_{i-1...-1}$ bits
- Since P_i and G_i bits are computed in parallel, this is faster than computing C_i from C_{i-1}
- G_{-1} is equivalent to C_{in}
- Formula for computing C_i in look-ahead block:
- For a 4-bit CLA block:
  - $C_0 = G_{-1}$
  - $C_1 = G_{-1} \bullet P_0 + G_0$
  - $C_2 = G_{-1} \bullet P_0 \bullet P_1 + G_0 \bullet P_1 + G_1$
  - $C_3 = \ldots$

$$C_i = \sum_{j=-1}^{i-1} G_j \prod_{k=j+1}^{i-1} P_k$$

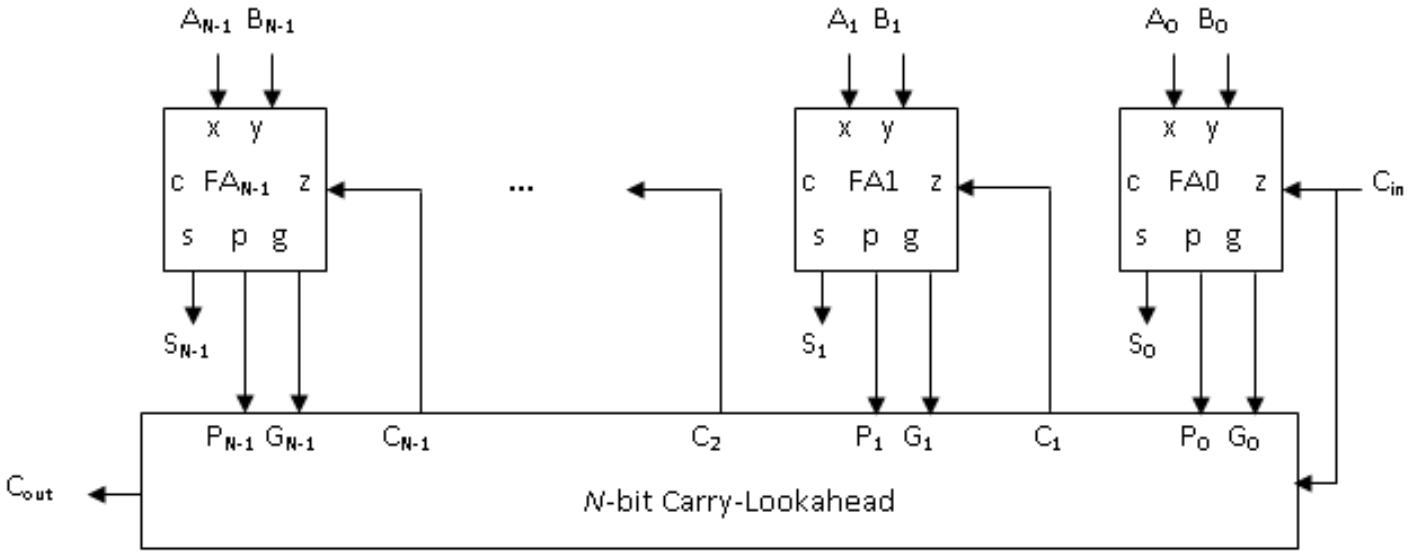
- **Very important note**: Lab manual gives iterative equation, but in implementation $C_{3...1}$ must be function of (G_x, P_x, C_0) only!
  - Must not ripple carry inside carry generation block (otherwise no point)!
  - Acceptable to cascade carry in between 4-bit CLA blocks

Carry Look-ahead Adder

- Critical path between C_{in} and C_{out} is long for large number of bits
- Note that carry calculation is strictly serial (due to dependency)
- Limits maximum frequency of operation
- Would like to compute some bits in parallel
- Modify FA to add P/G bit generation, feed P/G into carry look-ahead module

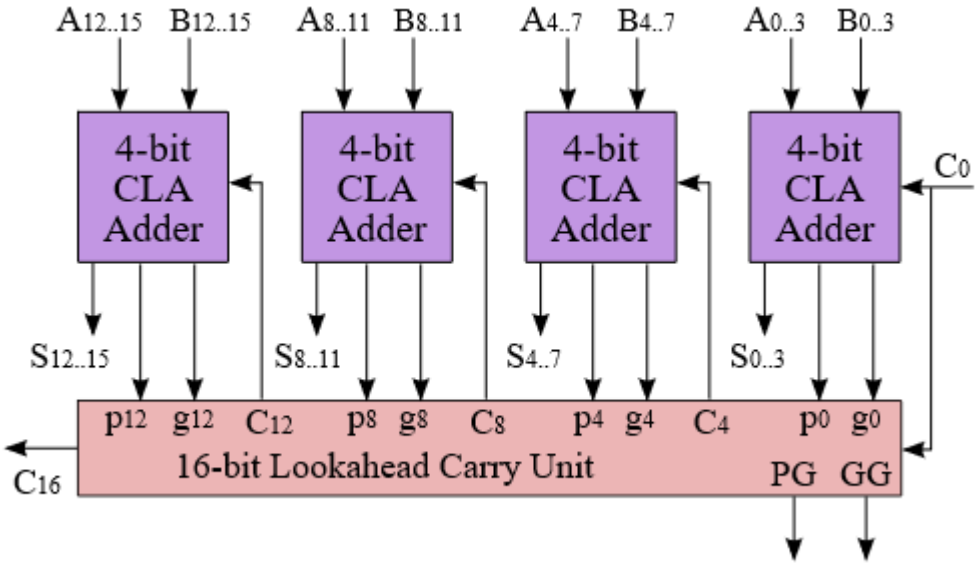


Carry Look-ahead Adder Speed



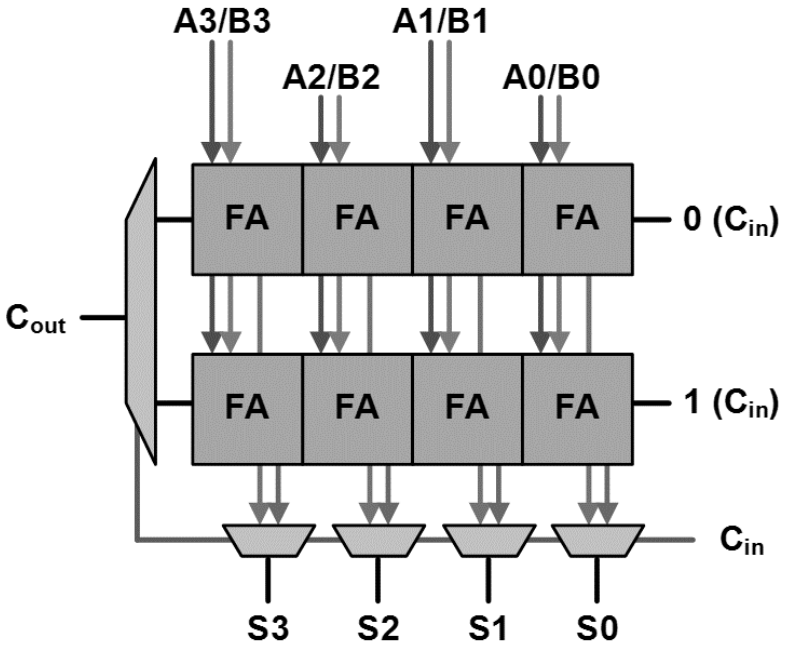
Carry Look-ahead Adder Cascading

- CLA Generator gets prohibitively expensive with larger number of bits
- Flat design becomes very expensive (and slow, because FPGA LUT is only 4 input)
- Solution: Cascade 4-bit CLAs to make larger adders
  - Option 1: Cascade C_{in} to C_{out} (as with ripple adder – fewest resources)
  - Option 2: Use another 4-bit CLA generator (hierarchical design – faster, this is what you will need to do in Lab 4)



Carry Select Adder

- An alternative approach to dealing with carry delay is to pre-compute the result for both C = 0 and C = 1 for each n bit block
- How does this speed up the ripple propagation?
- This requires twice as many full adders (and additional multiplexors) but reduced carry propagation delay
- 4-bit version show on right:
- Create 16-bit version from 4-bit
  - For this lab, can ripple 4 x 4-bit blocks
  - Why might this not be ideal?
  - Can first 4-bit block be simplified?



Timing Analysis

- Report should contain simulation traces
  - Relevant output e.g. P/G bits for CLA
  - Note: simulation does not take into account timing (all combinational logic is assumed to take Δt time (minimum simulation time unit)
  - Possible to do simulation with timing (post place & route) but in practice simulation performance is very slow
- Include timing analysis in report
  - Demo setup (top-level) takes inputs from register and stores output to register
  - Why is this important?