JOINT INSTITUTE
交大密西根学院

ECE2700J SU24 RC6

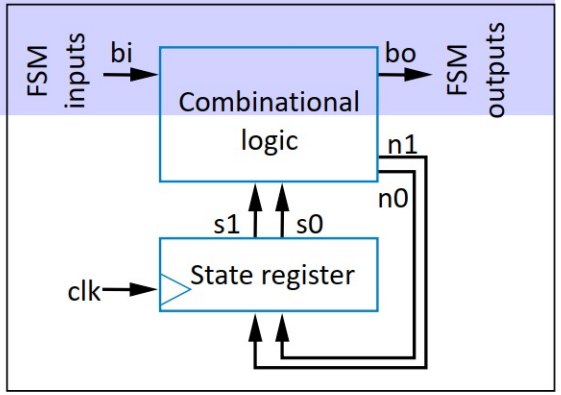
RTL Design & Arithmetic Components

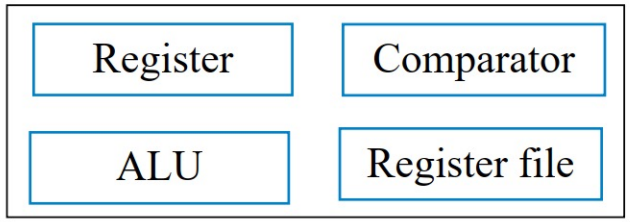
Wenyue Li
7/22/2024

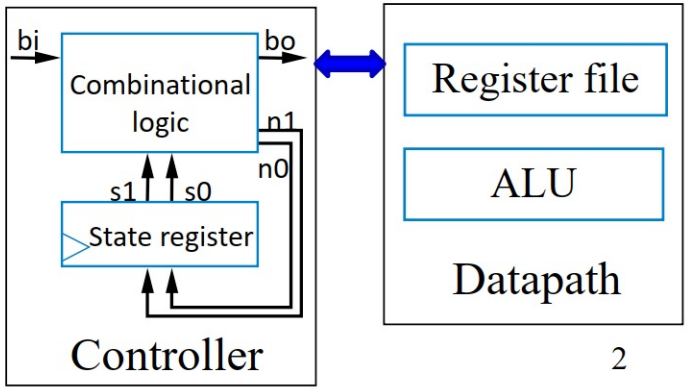
RTL Design

Introduction

- Controllers (FSM)
  - Describes behavior of circuits
  - Takes inputs, generates outputs
  - Implemented with state register and combinational logic
- Datapath components
  - Operations on data
  - Path that data flows through
  - Places data is stored
- Digital Device
  - Datapath processes data according to control signals produced by FSM
  - To implement an algorithm
  - Design on Register Transfer Level

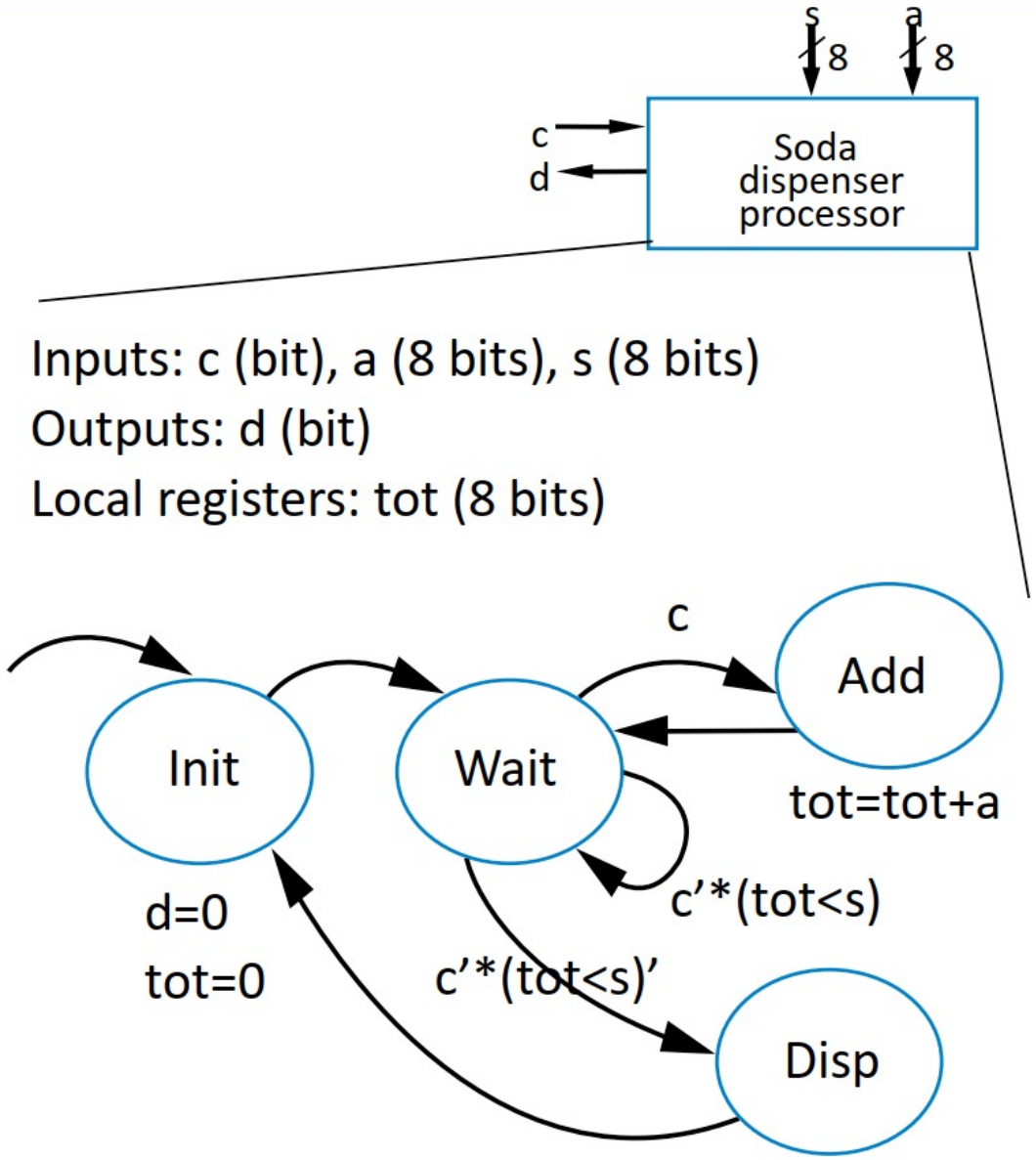






JOINT INSTITUTE 交大密西根学院

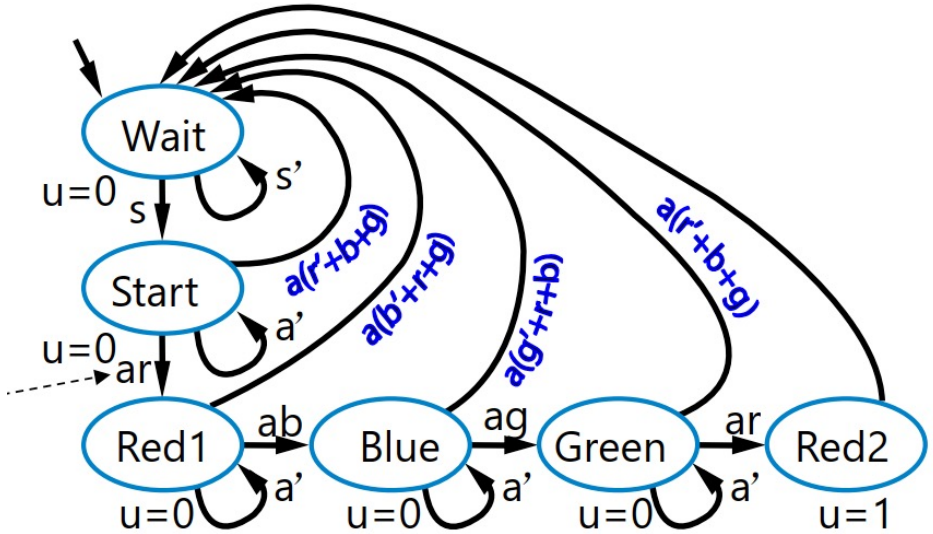
High level state machine (HLSM)



HLSM:

- Data types beyond just bits
- Arithmetic operations in states

HLSM VS. FSM



RTL Design Method

	Step	Description
Step 1	*Capture a high-level state machine*	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.
Step 2	*Create a datapath*	Create a datapath to carry out the data operations of the high-level state machine.
Step 3	*Connect the datapath to a controller*	Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.
Step 4	*Derive the controller's FSM*	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.

RTL Design Method

1. Design the following system. The system has two single-bit inputs U and D each coming from a button, and a 16-bit output C, which is initially 0. For each press of U, the system increments C. For each press of D, the system decrements C. If both buttons are pressed, the system does not change C. The system does not roll over; it goes no higher than the largest C and no lower than C=0. A press is detected as a change from 0 to 1; the duration of that 1 does not matter. Draw the HLSM, build datapath, convert the high-level state machine to an FSM, then connect the controller (FSM) and the datapath. Implementation of FSM is not required. (50 points)↵

RTL Design Method

Step 1. Capture High-Level State Machine

- Begin by declaring inputs and outputs
- Create initial state, name it **S0**

RTL Design Method

Step 2. Create a datapath

- Datapath must
  - Implement data storage
  - Implement data computations
- Look at high-level state machine, instantiate required components

RTL Design Method

Step 3. Connecting the Datapath to a Controller

- Connect all control signals between controller and datapath
- Identify the input/output signals for datapath and controller

RTL Design Method

Step 4. Deriving the Controller's FSM

- Replace data operations by bit operations using datapath

RTL Design Method

RTL Design of a Digital System

- 1. Design a digital system as two parts: controller and datapath
- 2. Capture the behavior of a digital system using HLSM
- 3. Find building blocks (datapath components) that are able to realize the data operations
- 4. Identify the input/output signals for datapath and controller
- 5. Convert HLSM to FSM

Arithmetic Components

Arithmetic-Logic Unit: ALU

- ***ALU***: Component that can perform any of various arithmetic (add, subtract, increment, etc.) and logic (AND, OR, etc.) operations, based on control inputs
- Key component in computer

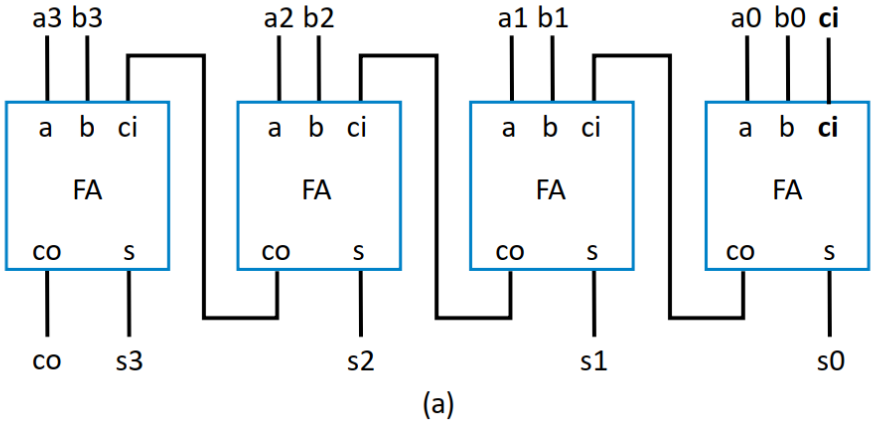
TABLE 4.2 Desired calculator operations

Inputs			Operation	Sample output if A=00001111, B=00000101
x	y	z		
0	0	0	S = A + B	S=00010100
0	0	1	S = A - B	S=00001010
0	1	0	S = A + 1	S=00010000
0	1	1	S = A	S=00001111
1	0	0	S = A AND B (bitwise AND)	S=00000101
1	0	1	S = A OR B (bitwise OR)	S=00001111
1	1	0	S = A XOR B (bitwise XOR)	S=00001010
1	1	1	S = NOT A (bitwise complement)	S=11110000

Arithmetic Components

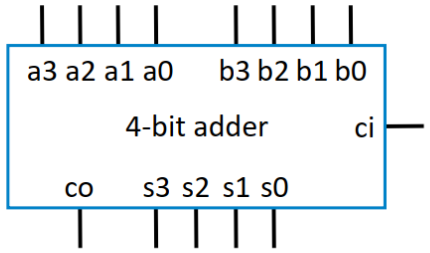
Carry-Ripple Adder

- 4-bit adder: Adds two 4-bit numbers, generates 5-bit output
  - 5-bit output can be considered 4-bit "sum" plus 1-bit "carry out"
- Can easily build any size adder



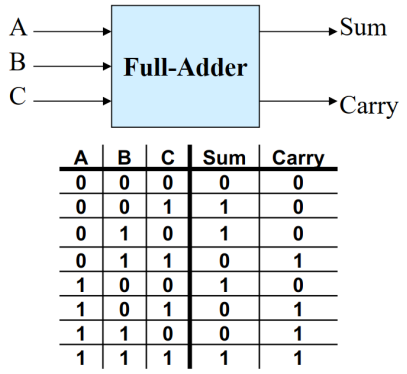
(a)

carries:	c3	c2	c1	cin
B:	b3	b2	b1	b0
A: +	a3	a2	a1	a0
cout	s3	s2	s1	s0



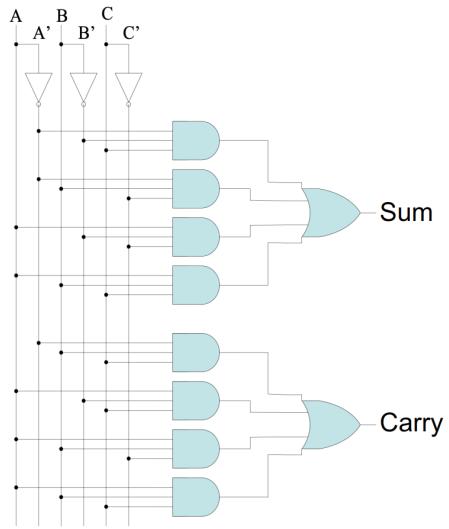
(b)

Full Adder



A	B	C	Sum	Carry
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

Sum = A'B'C + A'BC' + AB'C' + ABC
= Σ m(1, 2, 4, 7)
Carry = A'BC + AB'C + ABC' + ABC
= Σ m(3, 5, 6, 7)

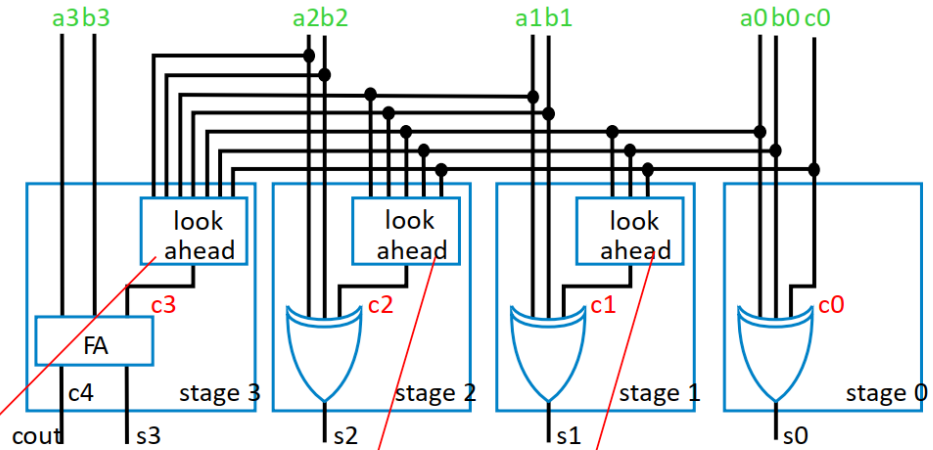


Notice the circuit draw convention

Arithmetic Components

Carry lookahead adder

- Carry lookahead logic
  - No waiting for ripple
  - 2-layer SOP logic
- Problem
  - Equations get too big
  - Not efficient
  - Need a better form of lookahead



c1 = b0c0 + a0c0 + a0b0

c2 = b1b0c0 + b1a0c0 + b1a0b0 + a1b0c0 + a1a0c0 + a1a0b0 + a1b1

c3 = b2b1b0c0 + b2b1a0c0 + b2b1a0b0 + b2a1b0c0 + b2a1a0c0 + b2a1a0b0 + b2a1b1 +
a2b1b0c0 + a2b1a0c0 + a2b1a0b0 + a2a1b0c0 + a2a1a0c0 + a2a1a0b0 + a2a1b1 + a2b2

Huge number of gates

Arithmetic Components

Carry lookahead adder

- Recall Full Adder, another equation for carry

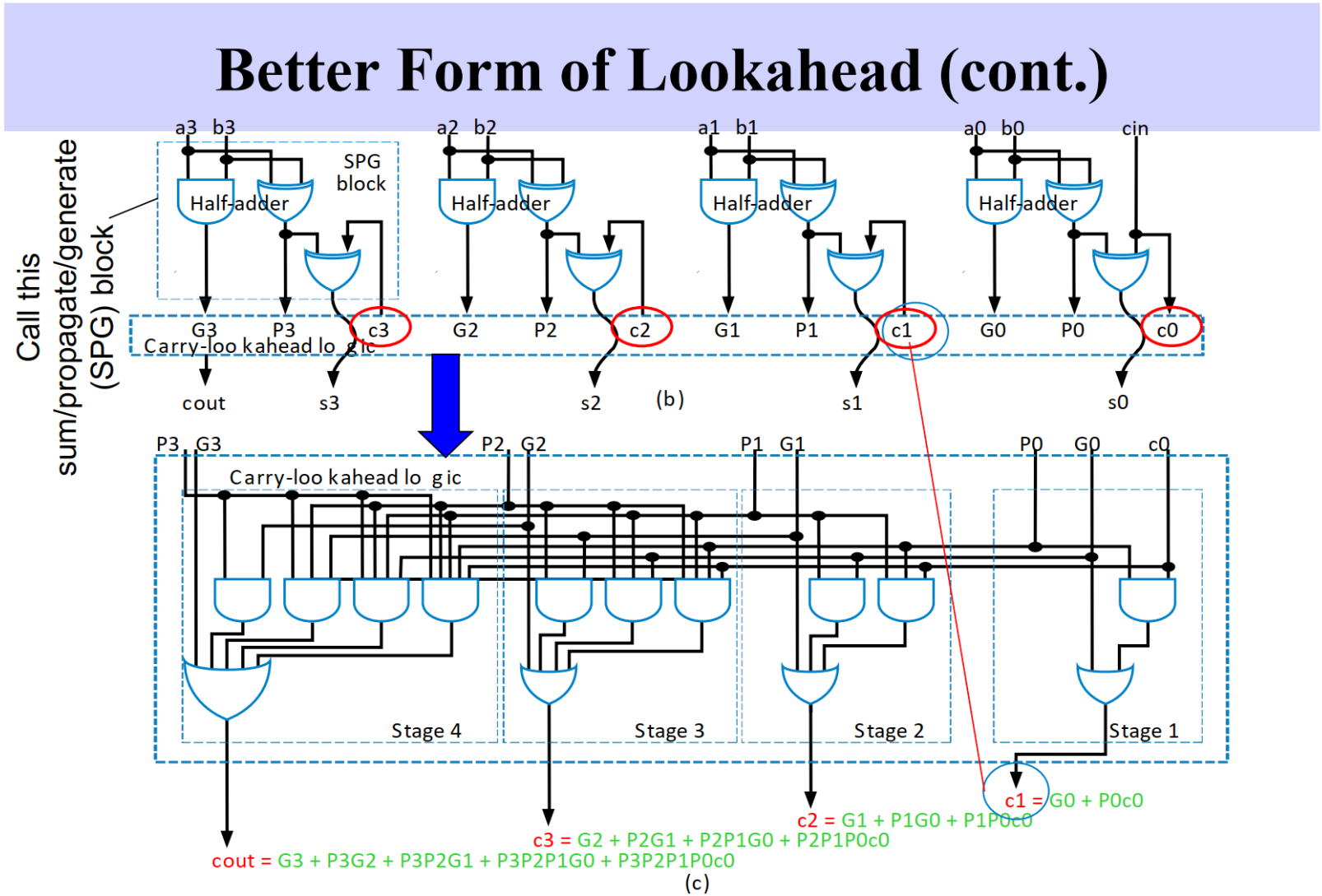
  Carry = ab + (a ⊕ b)c

  Logic statement: Carry out is **generated** when a=1 and b=1, or **propagated** from carry in when a != b.

- Define two terms
  - ***Propagate***: P = a ⊕ b
  - ***Generate***: G = ab
- Compute lookahead carries from *P* and *G* terms, *not from external inputs*
  - Cout = G + Pc
    - c1 = a0b0 + (a0⊕b0)c0 = G0 + P0c0
    - c2 = a1b1 + (a1⊕b1)c1 = G1 + P1c1
    - c3 = a2b2 + (a2⊕b2)c2 = G2 + P2c2

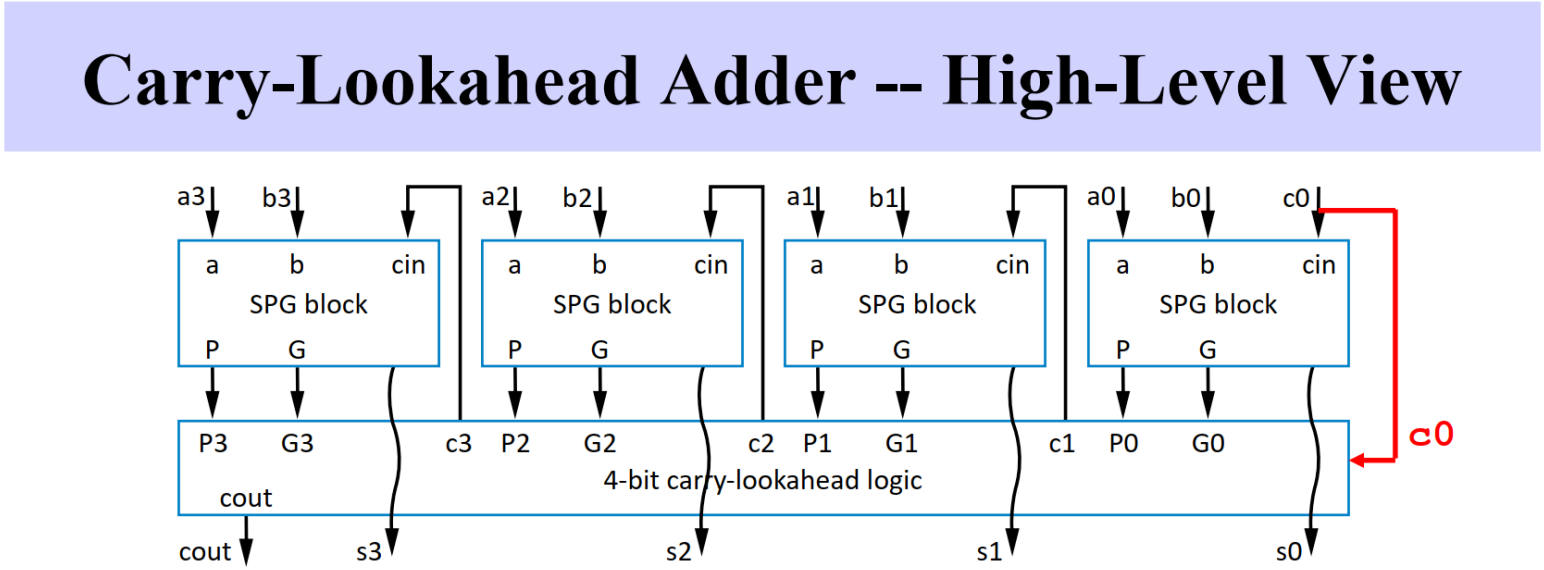
Arithmetic Components

Carry lookahead adder



Arithmetic Components

Carry lookahead adder

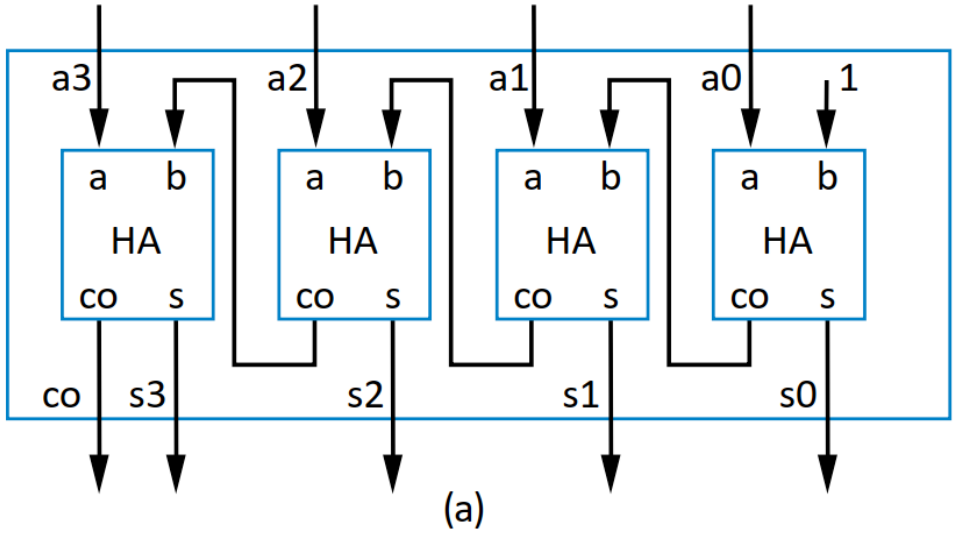


- Fast -- only 4 gate level delays
  - Each stage has SPG block with 2 gate levels
  - Carry-lookahead logic quickly computes the carry from the propagate and generate bits using 2 gate levels inside
- Reasonable number of gates
- Nice balance between intuitive lookahead and pure combinational logic

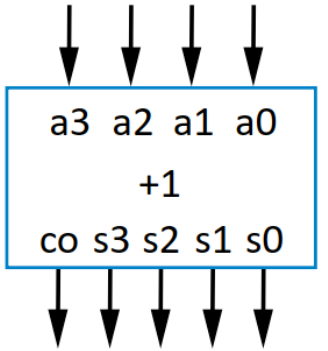
Arithmetic Components

Incrementer

Inputs				Outputs				
a3	a2	a1	a0	c0	s3	s2	s1	s0
0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	0	1	1	0	0	1	0	0
0	1	0	0	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	1	0	0	0	1	1	1
0	1	1	1	0	1	0	0	0
1	0	0	0	0	1	0	0	1
1	0	0	1	0	1	0	1	0
1	0	1	0	0	1	0	1	1
1	0	1	1	0	1	1	0	0
1	1	0	0	0	1	1	0	1
1	1	0	1	0	1	1	1	0
1	1	1	0	0	1	1	1	1
1	1	1	1	1	0	0	0	0



(a)



(b)

Arithmetic Components

Multiplier

- Can build multiplier that mimics multiplication by hand
  - Notice that multiplying multiplicand by 1 is same as ANDing with 1

```
  0110   (the top number is called the multiplicand)
  0011   (the bottom number is called the multiplier)
  ----   (each row below is called a partial product)
  0110   (because the rightmost bit of the multiplier is 1, and 0110*1=0110)
 0110    (because the second bit of the multiplier is 1, and 0110*1=0110)
0000     (because the third bit of the multiplier is 0, and 0110*0=0000)
+0000    (because the leftmost bit of the multiplier is 0, and 0110*0=0000)
--------
00010010 (the product is the sum of all the partial products: 18, which is 6*3)
```

Arithmetic Components

Multiplier