

Lecture 7

Arithmetic Circuits



Outline

- Synthesis from Cadence Genus (**)
- Arithmetic Circuits (***)
 - Adder, Subtractor
 - Arithmetic Logic Units (ALU)
 - Shifter
 - Multiplier
 - Textbook Chapter 8.1, 8.2, 8.3



Synthesis Flow

- Commercial EDA tool can automatically convert RTL into gate circuits
 - For better performance, we still need some manual design
- But you need to provide information
 - RTL: *.v
 - Physical information: *.lef
 - Timing information: *.lib
 - Design constraints: *.sdc
- This step is also referred as “technology mapping”, i.e. mapping to silicon technology, e.g., 65nm or 28nm process of a certain foundry
- Type “cdsnhelp” in genus to start Cadence online documents



Library File

set_db library /vol/ece303/genus_tutorial/NangateOpenCellLibrary_typical.lib

Contain standard cells' information

area, power, etc.

```
cell (INV_X1) {
  area
  pin (A) {
    : 0.532000;
    direction
    related_power_pin
    related_ground_pin
    capacitance
    fall_capacitance : 1.549360;
    rise_capacitance : 1.700230;
  }
  pin (ZN) {
    direction
    related_power_pin
    related_ground_pin
    max_capacitance
    function
    timing () {
      related_pin
      timing_sense
      cell_rise(Timing_7_7) {
        index_1 ("0.00117378,0.00472397,0.0171859,0.0409838,0.0780596,0.130081,0.198535");
        index_2 ("0.365616,1.897810,3.795620,7.591250,15.182500,30.365000,60.730000");
        values ("0.00558495,0.00952547,0.0142069,0.0234111,0.0416815,0.0781322,0.150988", \
          "0.00726612,0.0110313,0.0156990,0.0249513,0.0432921,0.0797973,0.152683", \
          "0.0117593,0.0172024,0.0222810,0.0312450,0.0494226,0.0858830,0.158767", \
          "0.0169697,0.0245178,0.0319657,0.0437440,0.0621260,0.0981372,0.170748", \
          "0.0234502,0.0327927,0.0422113,0.0575993,0.0814250,0.118167,0.190083", \
          "0.0313821,0.0424084,0.0535919,0.0721230,0.101629,0.146333,0.218093", \
          "0.0409686,0.0535508,0.0664252,0.0878376,0.122423,0.176146,0.255965");
      }
    }
  }
}
```

Delay (2D look up table) based on load and input transition time



LEF File

set_db lef_library /vol/ece303/genus_tutorial/NangateOpenCellLibrary.lef

```
MACRO INV_X1
CLASS core ;
FOREIGN INV_X1 0.0 0.0 ;
ORIGIN 0 0 ;
SYMMETRY X Y ;
SITE FreePDK45_38x28_10R_NP_162NW_34O ;
SIZE 0.38 BY 1.4 ;
PIN A
  DIRECTION INPUT ;
  ANTENNAPARTIALMETALAREA 0.018375 LAYER metal1 ;
  ANTENNAPARTIALMETALSIDEAREA 0.0728 LAYER metal1 ;
  ANTENNAGATEAREA 0.05225 ;
  PORT
    LAYER metal1 ;
    POLYGON 0.06 0.525 0.165 0.525 0.165 0.7 0.06 0.7 ;
  END
END A
PIN ZN
  DIRECTION OUTPUT ;
  ANTENNAPARTIALMETALAREA 0.1045 LAYER metal1 ;
  ANTENNAPARTIALMETALSIDEAREA 0.3107 LAYER metal1 ;
  ANTENNADIFFAREA 0.109725 ;
  PORT
    LAYER metal1 ;
    POLYGON 0.23 0.15 0.325 0.15 0.325 1.25 0.23 1.25 ;
  END
END ZN
PIN VDD
  DIRECTION INOUT ;
  USE power ;
  SHAPE ABUTMENT ;
  PORT
    LAYER metal1 ;
    POLYGON 0 1.315 0.04 1.315 0.04 0.975 0.11 0.975 0.11 1.315 0.38 1.315 0.38 1.485 0 1.485 ;
  END
END VDD
END INV_X1
```

LEF shows abstract of cell's layout

- Geometry information: X, Y size
- Metal layers
- Port locations
- Power wires
- Allow tools to make connections



SDC File

```
read_sdc ../alu_conv.sdc
```

- The most important supporting file
- Synopsis Design Constraint (SDC) file
 - Constrain total speed of the design
 - Constrain timing relationships between ports
 - Constrain clock information
 - Constrain cell usages



SDC File

```
#create_clock -name clk -period 0.6 -waveform { 0 0.3 } [get_ports clk]
# ----- Input constraints -----
#set_input_delay 0.3 -clock clk [get_ports [list din, start, rstb, wr_ctrl_test_crtl]]
# ----- Output constraints -----
#set_output_delay -clock clk_out_mf -max 0.2 [get_ports [list addr_out*]]
set_max_delay 1 -from [all_inputs] -to [all_outputs]
# Assume 50fF load capacitances everywhere:
set_load 0.050 [all_outputs]
# Set 10fF maximum capacitance on all inputs
set_max_capacitance 0.010 [all_inputs]
# set clock uncertainty of the system clock (skew and jitter)
#set_clock_uncertainty -setup 0.03 [get_clocks clk*]
#set_clock_uncertainty -hold 0.03 [get_clocks clk*]
# set maximum transition at output ports
set_max_transition 0.07 [current_design]
set_attr use_scan_seqs_for_non_dft false
```

Define clock waveform

Define max circuit delay, 1ns

Define max output port load, 0.05pF

Define max input port load, 0.01pF

Define timing margins for clock, 0.03ns

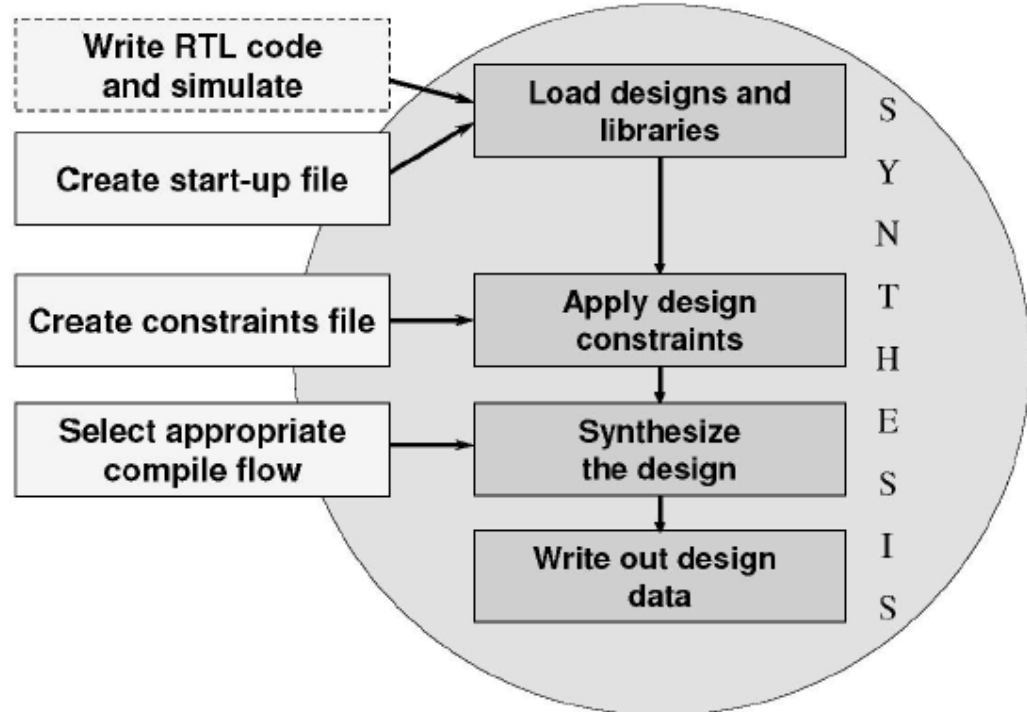
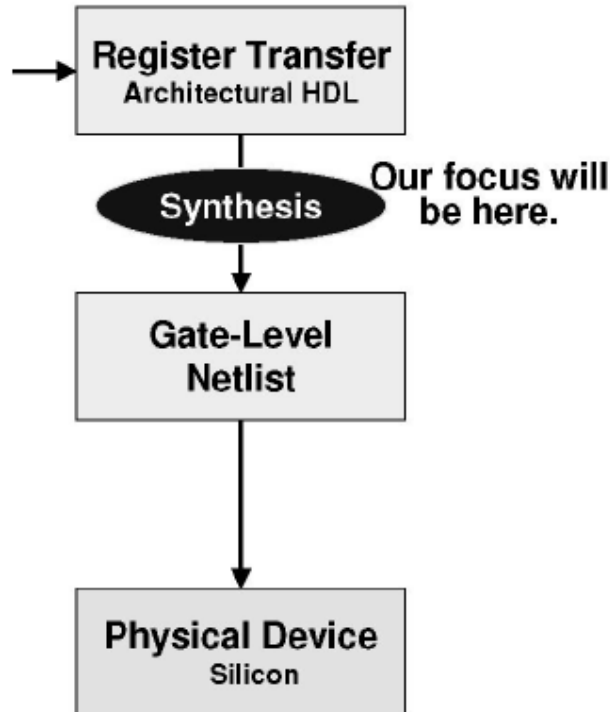
Define signal transition speed, 0.07ns

Disable/enable usage of special cells

- Note many lines are commented out in this example but will be used later in sequential design



Synthesis Flow





Synthesis Flow

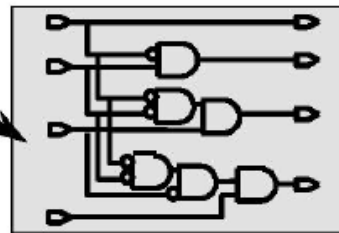
Synthesis = Translation + Logic Optimization + Gate Mapping

```
residue = 16'h0000; RTL Source  
if (high_bits == 2'b10)  
    residue = state_table[index];  
else  
    state_table[index] = 16'h0000;
```

1 **Translate** (read_verilog
read_vhdl)

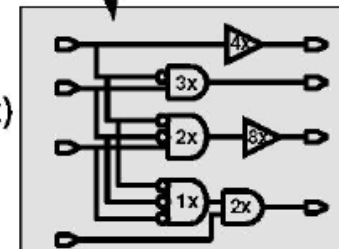
Constraints
set_max_area ...
create_clock ...
set_input_delay ...

2 **Constrain** (source)



Generic Boolean Gates
(GTECH or unmapped *ddc* format)

3 **Optimize + Map**
(compile)



Technology-specific Gates

The verb “to compile” is used
synonymously with “to synthesize”



Analyze and Elaborate

```
read_hdl ../alu_conv.v  
elaborate
```

- Analyze: read_hdl
 - Check syntax
 - Convert file into binary format and generate design folders
- Elaborate: elaborate
 - Set current design
 - Link modules and libraries
 - Elaborate "parameters"



Synthesis

syn_generic: generic mapping and RTL opt; tech independent

Synthesizes the design to generic gates. It takes a list of top-level designs and synthesizes the RTL blocks to generic gates using the given constraints and performs RTL optimization.

Use “syn_generic_effort” to control level of efforts [high | low | medium | express]

syn_map: technology dependent mapping and opt

Maps the design to the cells described in the supplied technology library and performs logic optimization.

Use “syn_map_effort” to control level of efforts

syn_opt: gate level optimization

Performs gate level optimization to improve timing on critical paths and recover area on non-critical paths. Optimizations can be done either on placed or mapped gates depending on the command options.

Use “syn_opt_effort” to control level of efforts



Timing Analysis

Path 1: MET (556 ps) Path Delay Check

Startpoint: (F) SEL[0]

Endpoint: (R) OUT4[7]

Capture Launch

Path Delay:+ 1000 -

Arrival:= 1000

Required Time:= 1000

Data Path:- 444

Slack:= 556

- Use "report_timing"
- List detailed path delay
- Follow constraint from SDC
- Should have positive "Slack"
 - Negative slack means "violation", require fix

```
#-----
# Timing Point  Flags  Arc  Edge  Cell  Fanout Load Trans Delay Arrival
#                                     (fF) (ps) (ps) (ps)
#-----
SEL[0]          -  -  F  (arrival)  1 1.0  0  0  0
drc_buf_sp2041/Z -  A->Z  F  CLKBUF_X1  9 14.4  35  60  60
drc_bufs1987/ZN -  A->ZN  R  INV_X1    8 14.9  37  58  118
g1703__1297/ZN -  C2->ZN  F  OAI211_X1  4 5.6  26  44  162
g1688__7654/ZN -  A2->ZN  R  NAND2_X1  2 4.8  18  33  195
g1670__7547/ZN -  B1->ZN  F  OAI21_X1  2 4.1  20  22  217
g1668__2006/ZN -  B1->ZN  R  AOI21_X1  2 4.4  34  44  261
g1666__1237/ZN -  B1->ZN  F  OAI21_X1  2 4.1  20  26  288
g1664__3779/ZN -  B1->ZN  R  AOI21_X1  2 4.4  34  44  332
g1662__1377/ZN -  B1->ZN  F  OAI21_X1  2 4.1  20  26  358
g1660__8867/ZN -  B1->ZN  R  AOI21_X1  2 4.4  34  44  402
g1659__7557/ZN -  A->ZN  R  XNOR2_X1  1 0.3  13  41  444
OUT4[7]        <<<  -  R  (port)   -  -  -  0  444
#-----
```



Area Report

- Use "report_area"
- Report estimated area (both cell area and wire area) and cell counts

Instance	Module	Cell Count	Cell Area	Net Area	Total Area
alu_conv		220	225.834	322.806	548.640



Generate gate level netlist

- "write_hdl"
- Generate gate level netlist in *.v
 - Structural Verilog netlist
- Use this generated Verilog netlist for:
 - Backend layout generation (placement)
 - Post-synthesis gate level simulation in Xcelium



Generated Verilog Netlist

```
// Generated by Cadence Genus(TM) Synthesis Solution 16.24-s065_1
// Generated on: Sep  2 2019 23:35:33 CDT (Sep  3 2019 04:35:33 UTC)

// Verification Directory fv/mini_alu

module mini_alu(A, B, SEL, OUT1, OUT2, OUT3, OUT4);
  input [7:0] A, B;
  input [1:0] SEL;
  output [7:0] OUT1, OUT2, OUT3;
  output [8:0] OUT4;
  wire [7:0] A, B;
  wire [1:0] SEL;
  wire [7:0] OUT1, OUT2, OUT3;
  wire [8:0] OUT4;
  wire n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7;
  wire n_8, n_9, n_10, n_11, n_12, n_13, n_14, n_15;
  wire n_16, n_17, n_37, n_38, n_39, n_40, n_42, n_43;
  wire n_44, n_45, n_47, n_48, n_49, n_50, n_52, n_53;
  wire n_54, n_57, n_59, n_62, n_64, n_67, n_69, n_95;
  wire n_100, n_105, n_110, n_115, n_120, n_125, n_130, n_135;
  wire n_140, n_145, n_150, n_155, n_160, n_165, n_170, n_175;
  wire n_180;
  OAI21_X1 g1658__7837(.A (n_47), .B1 (n_69), .B2 (n_12), .ZN
    (OUT4[8]));
  XNOR2_X1 g1659__7557(.A (n_69), .B (n_48), .ZN (OUT4[7]));
  XOR2_X1 g1661__7654(.A (n_67), .B (n_50), .Z (OUT4[6]));
  AOI21_X1 g1660__8867(.A (n_49), .B1 (n_67), .B2 (OUT3[6]), .ZN
    (n_69));
  OAI21_X1 g1662__1377(.A (n_52), .B1 (n_64), .B2 (n_14), .ZN (n_67));
  XNOR2_X1 g1663__3717(.A (n_64), .B (n_53), .ZN (OUT4[5]));
  XOR2_X1 g1665__4599(.A (n_62), .B (n_45), .Z (OUT4[4]));
  AOI21_X1 g1664__3779(.A (n_44), .B1 (n_62), .B2 (OUT3[4]), .ZN
    (n_64));
  XNOR2_X1 g1667__2007(.A (n_59), .B (n_43), .ZN (OUT4[3]));
  OAI21_X1 g1666__1237(.A (n_42), .B1 (n_59), .B2 (n_13), .ZN (n_62));
  XOR2_X1 g1669__1297(.A (n_57), .B (n_40), .Z (OUT4[2]));
  AOI21_X1 g1668__2006(.A (n_39), .B1 (n_57), .B2 (OUT3[2]), .ZN
    (n_59));
  XNOR2_X1 g1671__2833(.A (n_38), .B (n_54), .ZN (OUT4[1]));
  OAI21_X1 g1670__7547(.A (n_37), .B1 (n_54), .B2 (n_11), .ZN (n_57));
```

- Should function the same as the original RTL
- Please watch the “genus.log” files for warnings about issues



Simulate generated netlist

```
xrun -64bit -gui -access r -xmelab_args "-warnmax 0 -  
delay_mode zero -maxdelays" ./Synthesis/alu_conv_syn.v  
alu_conv_test.v  
/vol/ece303/genus_tutorial/NangateOpenCellLibrary.v
```

- Use the same testbench as RTL
- Results should be the same if RTL is written properly and synthesis ran effectively
- We can now also annotate 'real' delay into logic gates to simulate actual speed of the design

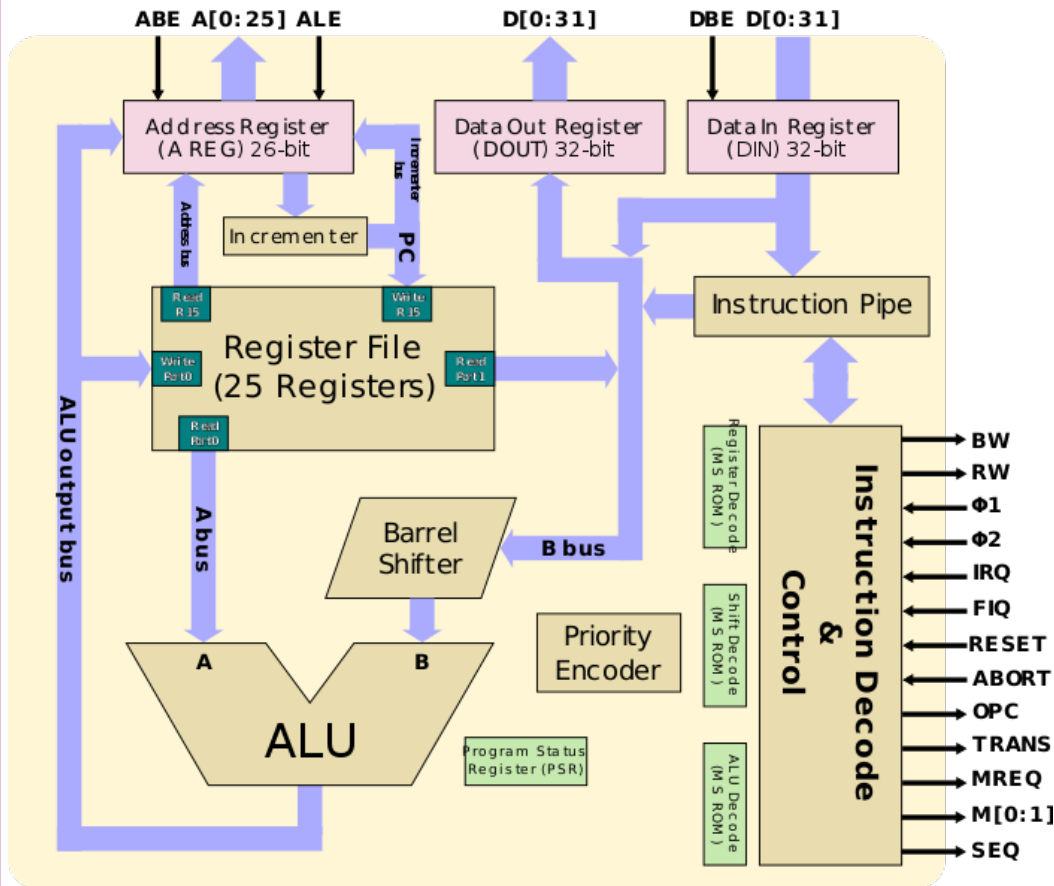


Arithmetic Circuits

- Circuits perform arithmetic operations
- Important building blocks of digital systems
 - Datapaths of processors
 - Computation kernels of all application specific integrated circuits
 - Image, Video encoding/decoding
 - DSP, Filtering
 - Cryptography
 - Networking



Example of CPU Core



Pipeline Operation Sequence

- Instructions are decoded
 - E.g. ADD or SUB
- Operands are loaded from Register File
- ALU performs arithmetic calculation
- Data saved back into Register File or Memory

ARM1 Architecture



Basic Arithmetic Operations

- Logic operations: AND, XOR, etc
- Addition/Subtraction
- Multiplication
- Division
- Multiply-Accumulate (MAC)

Note we will only briefly cover the concepts of these blocks in this class. There are many more techniques developed to improve each of the arithmetic modules in advanced VLSI design techniques.



1-bit Half Adder and Full Adder

- Adder operations:

$$\begin{array}{rclcl} A & B & CO & HS \\ 1 & + 1 & = & 1 & 0 \end{array}$$

Half Adder:
(2 inputs)

$$\begin{aligned} \text{Half Sum: } HS &= A \oplus B \\ &= A \cdot B' + A' \cdot B \end{aligned}$$

$$\text{Carry-out: } CO = A \cdot B$$

Full Adder:
(3 inputs)

Carry Input

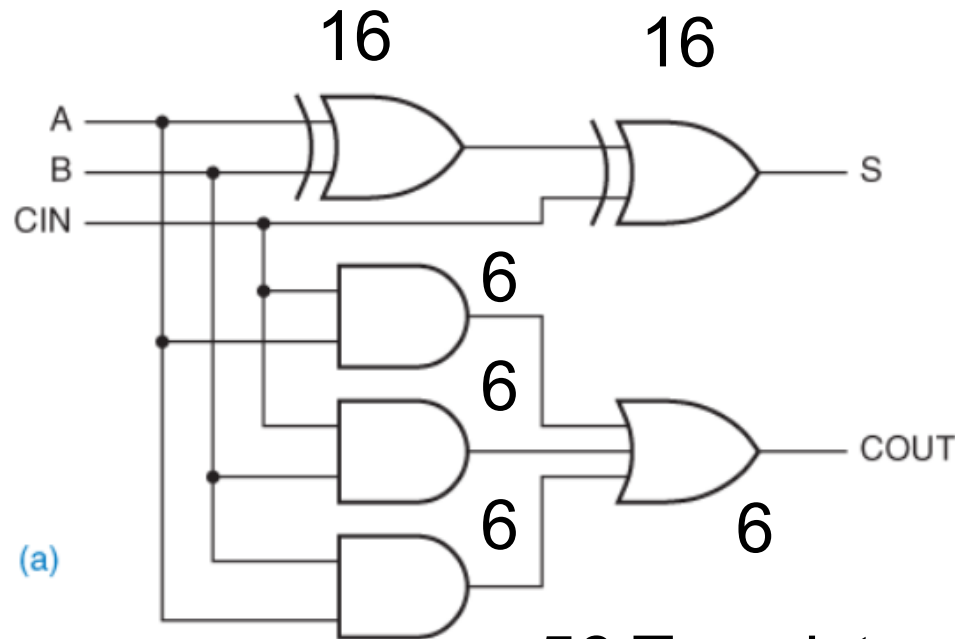
$$\begin{array}{rclcl} A & B & CIN & COUT & S \\ 1 & + 1 & + 1 & = & 1 & 1 \end{array}$$

$$\begin{aligned} \text{Sum: } S &= A \oplus B \oplus CIN \\ &= A \cdot B' \cdot CIN' + A' \cdot B \cdot CIN' + A' \cdot B' \cdot CIN + A \cdot B \cdot CIN \end{aligned}$$

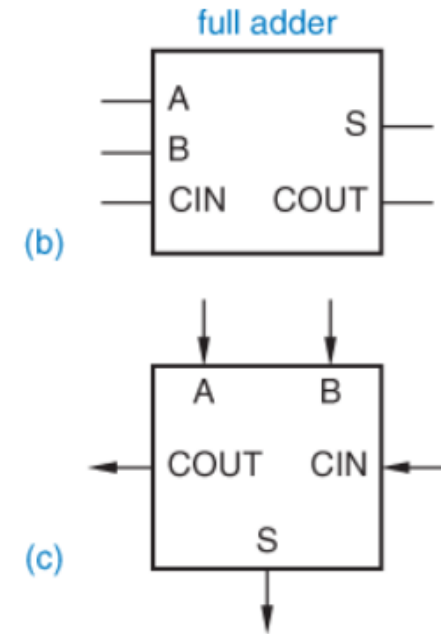
$$\text{Carry: } COUT = A \cdot B + A \cdot CIN + B \cdot CIN$$



Full Adder



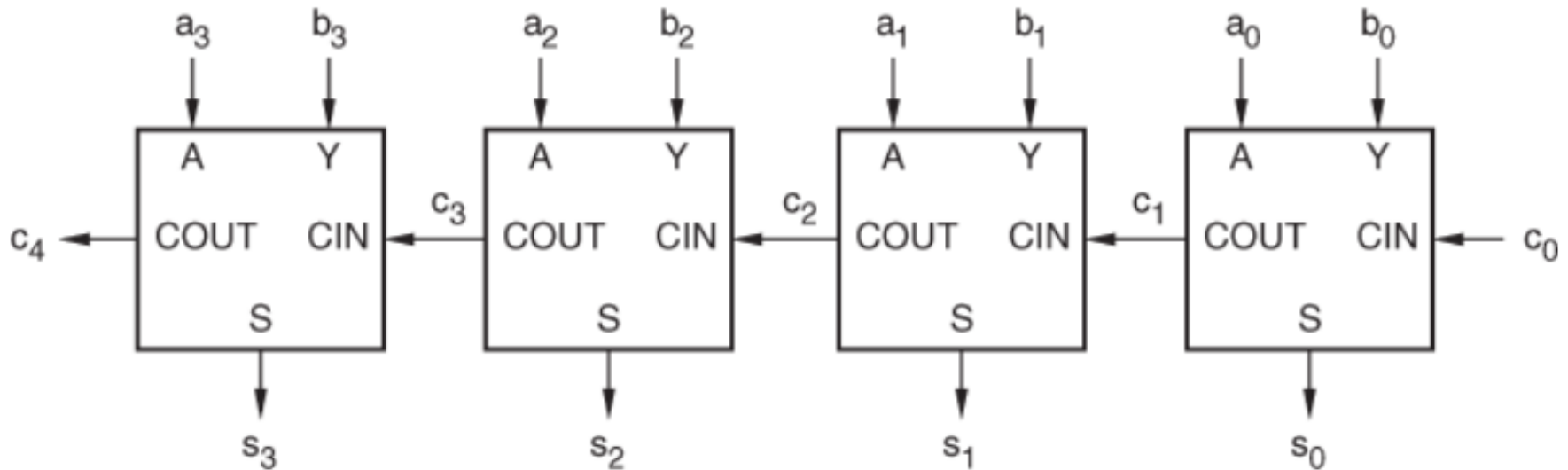
56 Transistors



- Can be built as schematic above
- Transistor level optimization improves adder
 - I can use fewer transistors to realize full adder above. How?



Ripple Adder



- We often need 32 or 64 bit adders to perform arithmetic with a wide range of values
- Ripple Adder: Connect full adder in series



Subtractors

Difference
Borrow

$$D = A \oplus B' \oplus \text{BIN}$$
$$\text{BOUT} = A' \cdot B + A' \cdot \text{BIN} + B \cdot \text{BIN}$$

Borrow In (=1)

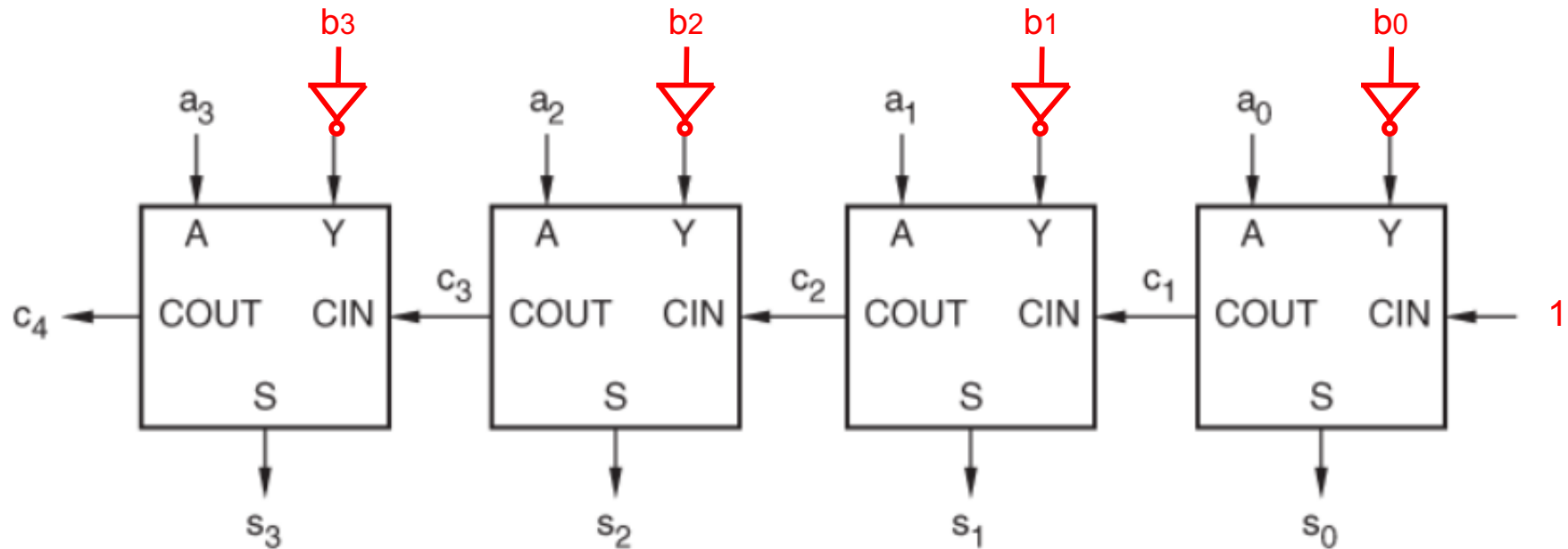
Full Adder

$$S = A \oplus B \oplus \text{CIN}$$
$$= A \cdot B' \cdot \text{CIN}' + A' \cdot B \cdot \text{CIN}' + A' \cdot B' \cdot \text{CIN} + A \cdot B \cdot \text{CIN}$$
$$\text{COUT} = A \cdot B + A \cdot \text{CIN} + B \cdot \text{CIN}$$

- Almost identical to the full adder
 - Except for one inverted input
- So we can use full adder to design subtractor
 - $A - B = A + (\sim B + 1)$



Subtractor



- $A - B = A + (\sim B + 1)$



Adder delay analysis

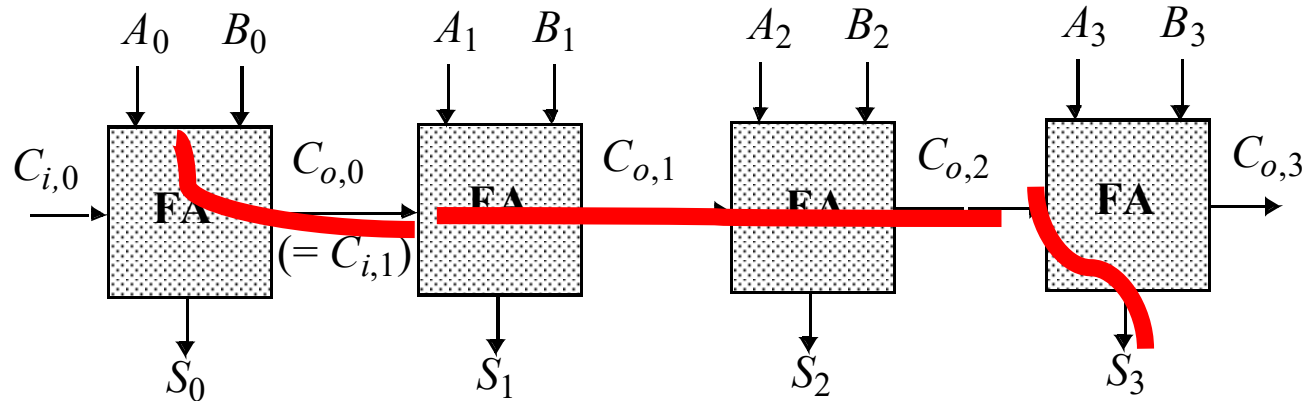
- Carry signal is the bottleneck of the ripple adder, i.e. propagation path too long for 64 bits
- So we need to break the carry chain to speed up the operation
- Refer to textbook 8.1.4



The Ripple-Carry Adder



The Ripple-Carry Adder



Total delay through this chain correlates with the size of the adder

$$t_{\text{ADD}} = t_{(A_0B_0)\text{To}C_{\text{out}0}} + (N-2) t_{C_{\text{in}i}\text{To}C_{\text{out}i+1}} + t_{C_{\text{in}(n-1)}\text{To}S_{n-1}}$$

$$t_{\text{ADD4Bit}} = t_{(A_0B_0)\text{to}C_{\text{out}0}} + 2 t_{C_{\text{in}i}\text{To}C_{\text{out}i+1}} + t_{C_{\text{in}3}\text{To}S_3}$$



The Carry Lookahead Adder



Basic Signals

Generate signal:

$$g_i = x_i y_i$$

Describes conditions where a carry out '1' is created

Propagate signal:

$$p_i = x_i \oplus y_i$$

Carry recurrence

$$c_{i+1} = g_i + c_i p_i = g_i + c_i t_i$$



Unrolling Carry Recurrence

$$\begin{aligned}c_i &= g_{i-1} + c_{i-1}p_{i-1} = \\&= g_{i-1} + (g_{i-2} + c_{i-2}p_{i-2})p_{i-1} = g_{i-1} + g_{i-2}p_{i-1} + c_{i-2}p_{i-2}p_{i-1} = \\&= g_{i-1} + g_{i-2}p_{i-1} + (g_{i-3} + c_{i-3}p_{i-3})p_{i-2}p_{i-1} = \\&= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + c_{i-3}p_{i-3}p_{i-2}p_{i-1} = \\&= \dots = \\&= g_{i-1} + g_{i-2}p_{i-1} + g_{i-3}p_{i-2}p_{i-1} + g_{i-4}p_{i-3}p_{i-2}p_{i-1} + \dots + \\&\quad + g_0p_1p_2\dots p_{i-2}p_{i-1} + c_0p_0p_1p_2\dots p_{i-2}p_{i-1} =\end{aligned}$$

$$= \boxed{g_{i-1} + \sum_{k=0}^{i-2} g_k \prod_{j=k+1}^{i-1} p_j + c_0 \prod_{j=0}^{i-1} p_j}$$



4-bit Carry-Lookahead Adder

$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$

$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$

$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$

$$c_1 = g_0 + c_0 p_0$$

$$s_0 = x_0 \oplus y_0 \oplus c_0 = p_0 \oplus c_0$$

$$s_1 = p_1 \oplus c_1$$

$$s_2 = p_2 \oplus c_2$$

$$s_3 = p_3 \oplus c_3$$



4-bit Carry-Lookahead Adder : Resource Optimized

$$c_4 = g_3 + c_3 p_3$$

$$c_3 = g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$

$$c_2 = g_1 + g_0 p_1 + c_0 p_0 p_1$$

$$c_1 = g_0 + c_0 p_0$$

**3 gates less
But
introduces
dependency
between c_3
and c_4**

$$s_0 = x_0 \oplus y_0 \oplus c_0 = p_0 \oplus c_0$$

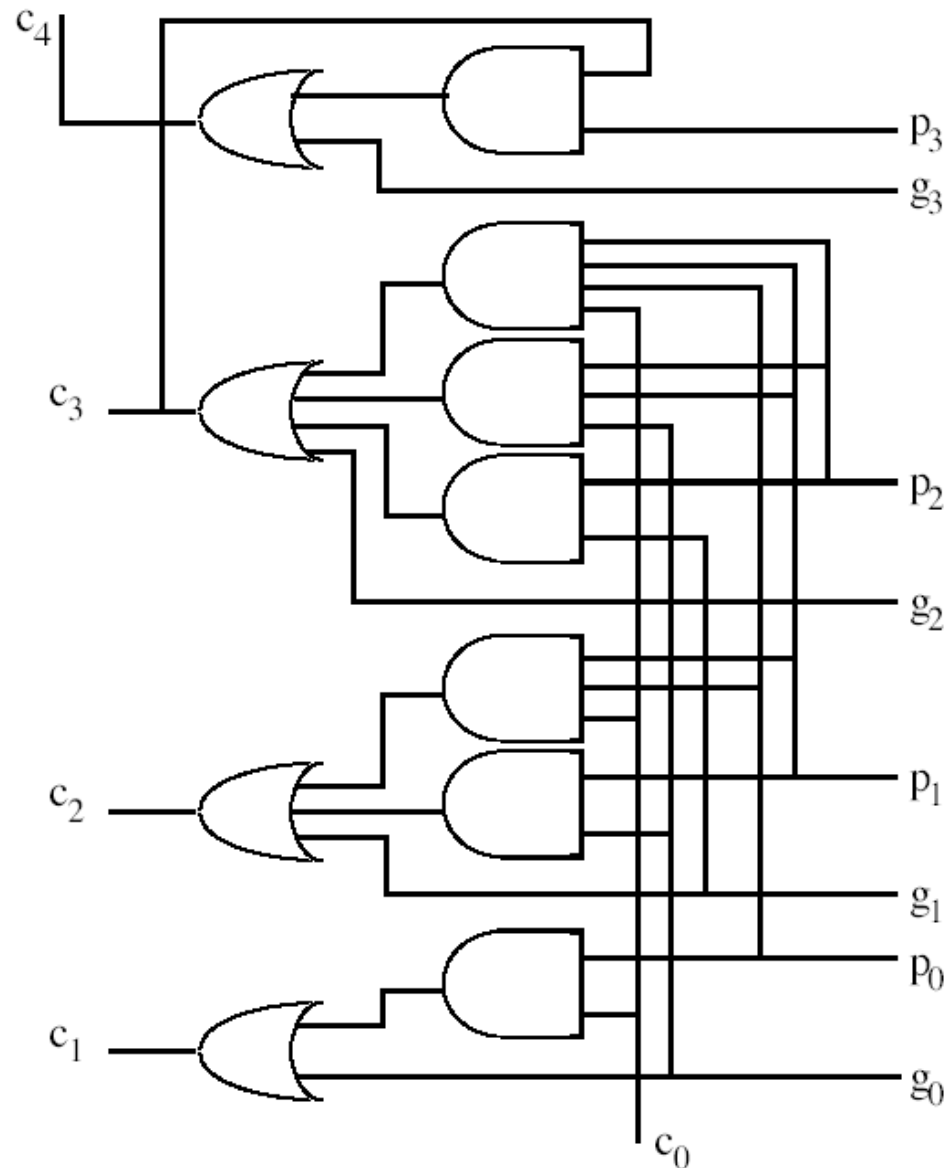
$$s_1 = p_1 \oplus c_1$$

$$s_2 = p_2 \oplus c_2$$

$$s_3 = p_3 \oplus c_3$$



Resource Optimized 4-bit Carry Network with Full Lookahead



Without this dependency all carry values are independent at the expense of more gates to calculate the p and g for c_4



4-bit Lookahead Carry Generator

Equations

$$c_{i+3} = g_{i+2} + g_{i+1} p_{i+2} + g_i p_{i+1} p_{i+2} + c_i p_i p_{i+1} p_{i+2}$$

$$c_{i+2} = g_{i+1} + g_i p_{i+1} + c_i p_i p_{i+1}$$

Generators are usually built
in blocks of 4 in larger adders

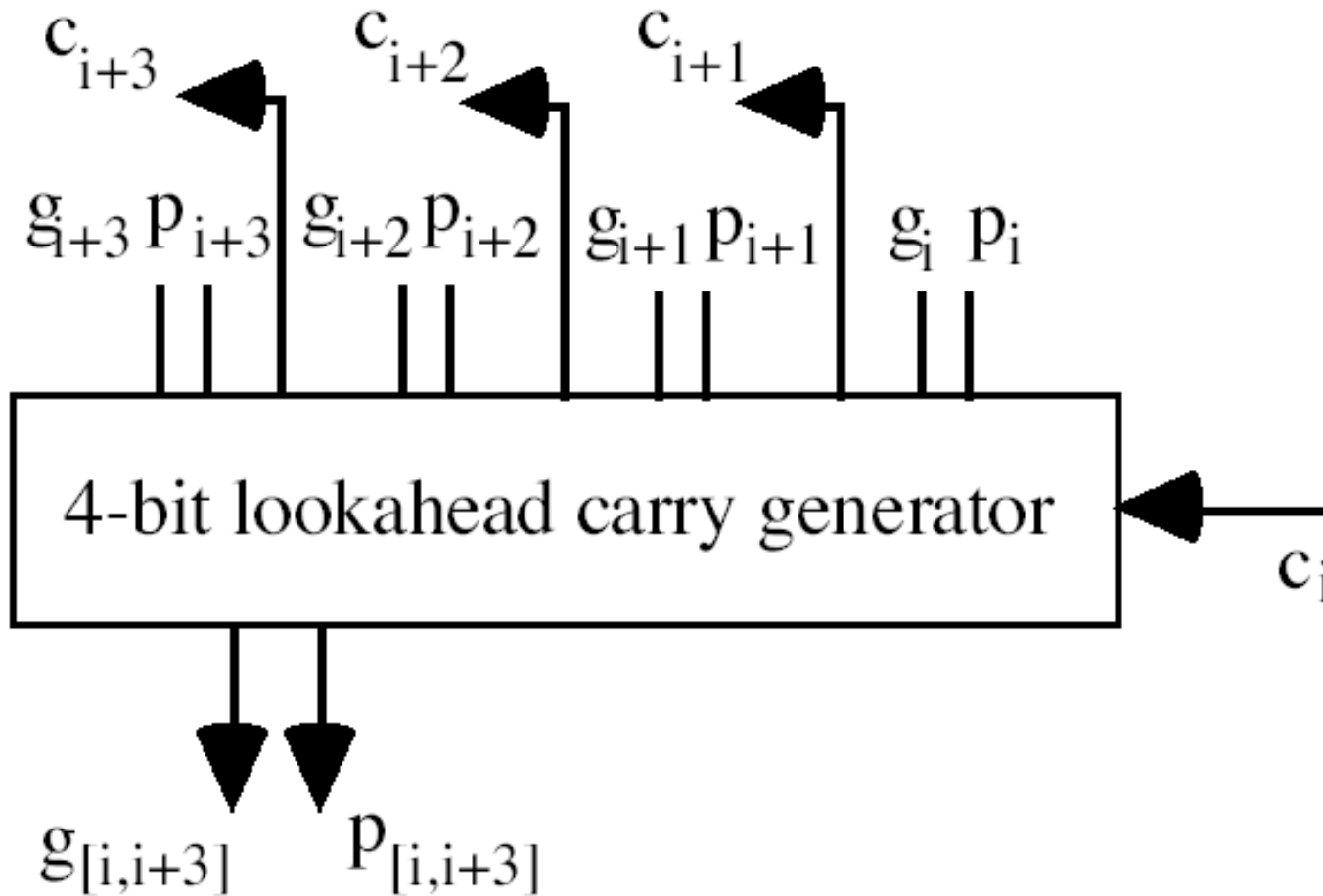
$$c_{i+1} = g_i + c_i p_i$$

$$g_{[i..i+3]} = g_{i+3} + g_{i+2} p_{i+3} + g_{i+1} p_{i+2} p_{i+3} + g_i p_{i+1} p_{i+2} p_{i+3}$$

$$p_{[i..i+3]} = p_i p_{i+1} p_{i+2} p_{i+3}$$



4-bit Lookahead Carry Generator: Block Diagram





Delay of a k-bit Carry-Lookahead Adder

$$T_{\text{lookahead-adder}} = 4 \lceil \log_4 k \rceil$$

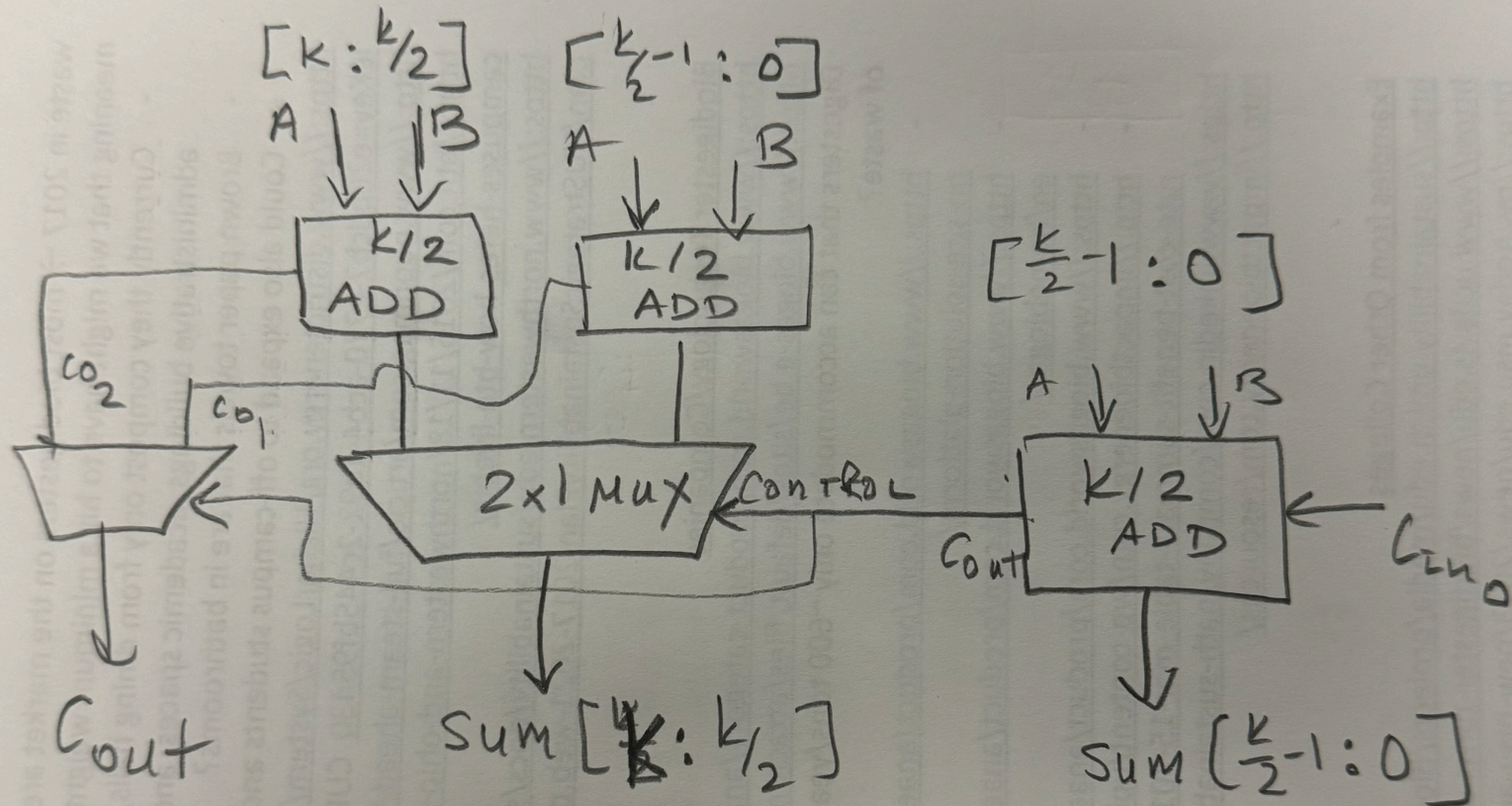
k	$T_{\text{lookahead-adder}}$	$T_{\text{ripple-carry-adder}}$
4	4	8
16	8	32
32	12	64
64	12	128
128	16	256
256	16	512



Carry-Select Adders



One-level k-bit Carry-Select Adder





Prefix (Parallel) Adders



Parallel Prefix Operation

Terminology background:

- Prefix: The outcome of the operation depends on the initial inputs.
- Parallel: Involves the execution of an operation in parallel. This is done by segmentation into smaller pieces that are computed in parallel.
- Operation: Any arbitrary primitive operator “ \circ ” that is associative is parallelizable
 - it is fast because the processing is accomplished in a parallel fashion.



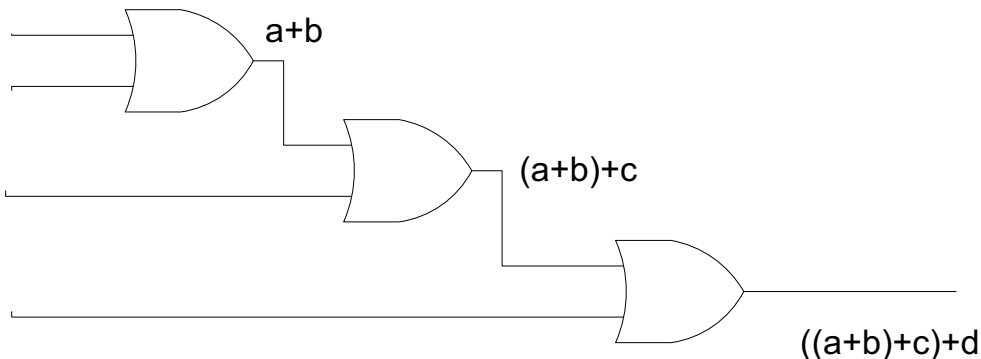
Example: Associative operations are parallelizable

Consider the logical OR operation: $a + b$

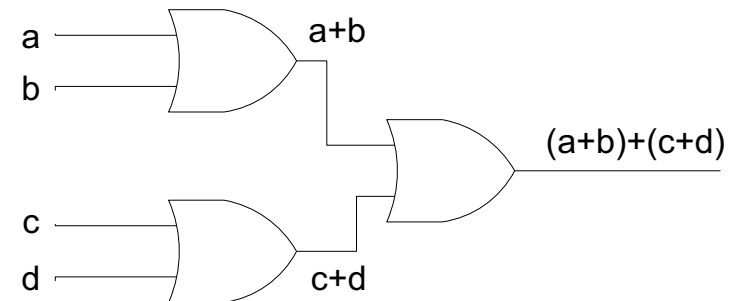
The operation is associative:

$$a + b + c + d = (((a + b) + c) + d) = ((a + b) + (c + d))$$

Serial implementation:



Parallel implementation:





Mathematical Formulation: Prefix Sum

- Operator: “ \circ ”
← this is the unary operator known as “scan” or “prefix sum”
- Input is a vector:
 $A = A_n A_{n-1} \dots A_1$
- Output is another vector:
 $B = B_n B_{n-1} \dots B_1$
where
 $B_1 = A_1$
 $B_2 = A_1 \circ A_2$
...
 $B_n = A_1 \circ A_2 \dots \circ A_n$
← B_n represents the operator being applied to all terms of the vector.



Example of prefix sum

Consider the vector: $\mathbf{A} = A_n A_{n-1} \dots A_1$ where element A_i is an integer

The “*” unary operator, defined as:

$$*A = B$$

With

$$B = B_n B_{n-1} \dots B_1$$

$$B_1 = A_1$$

$$B_2 = A_1 * A_2$$

$$B_3 = A_1 * A_1 * A_3$$

...

and “*” here is the integer addition operation.



Example of prefix sum

Calculation of $*A$, where $A = 6\ 5\ 4\ 3\ 2\ 1$ yields:

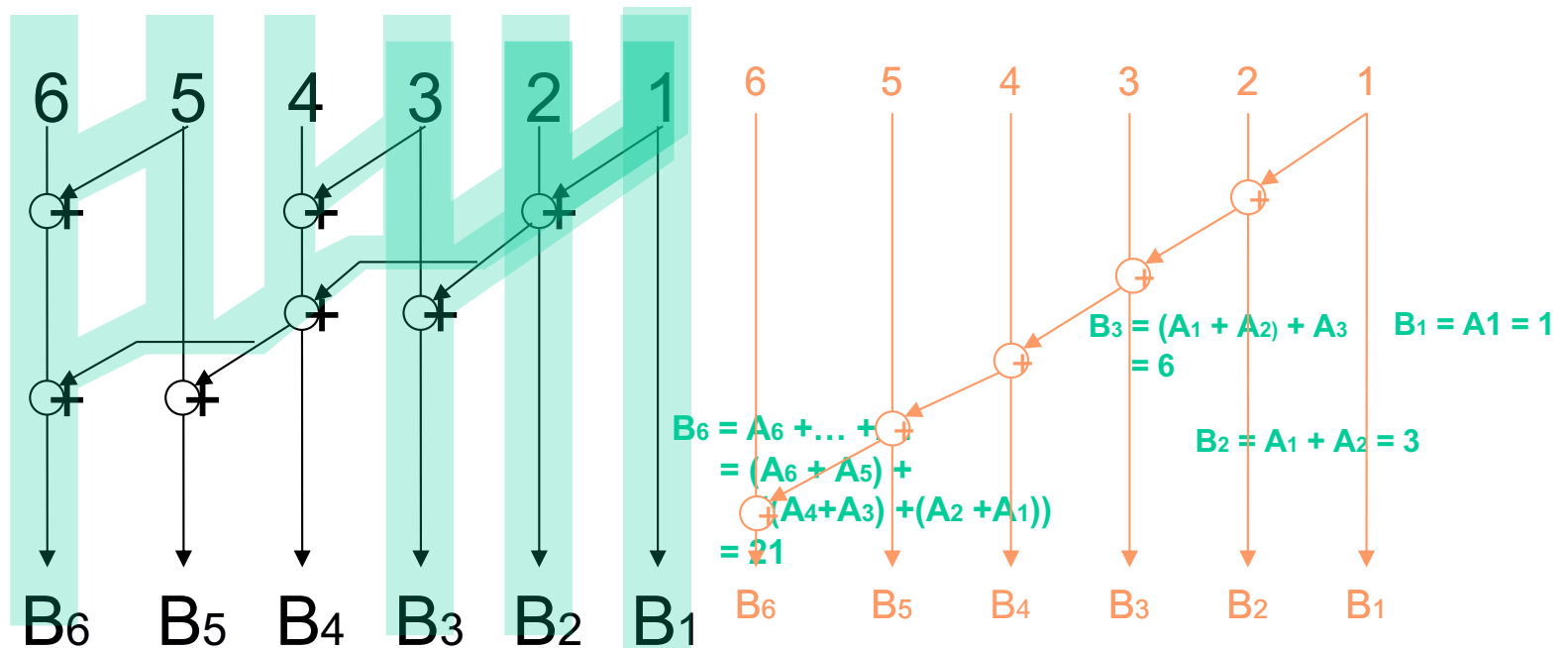
$$\mathbf{B} = *A = \mathbf{21\ 15\ 10\ 6\ 3\ 1}$$

Because the summation is associative the calculation can be done in parallel in the following manner:

Parallel implementation

versus

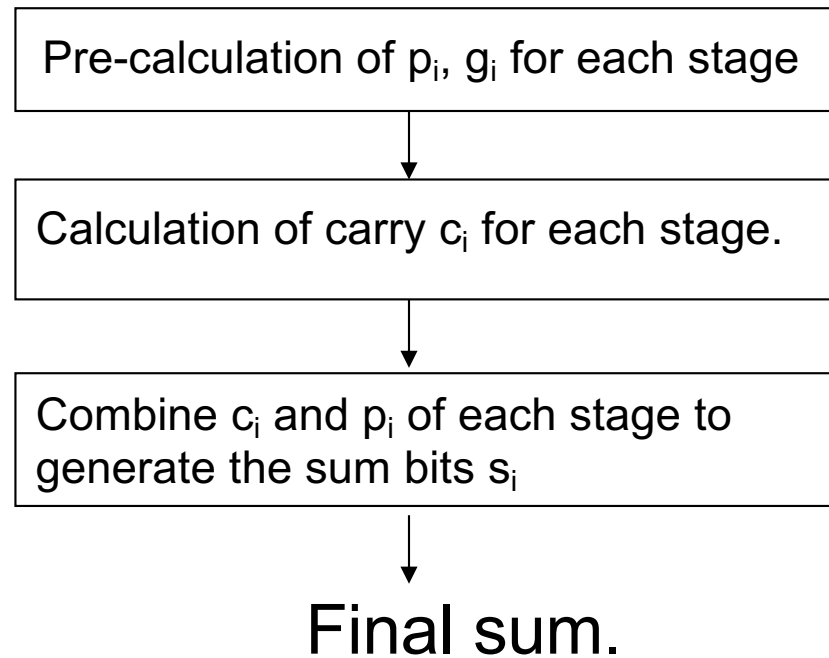
Serial implementation





Remember Carry Look Ahead adders

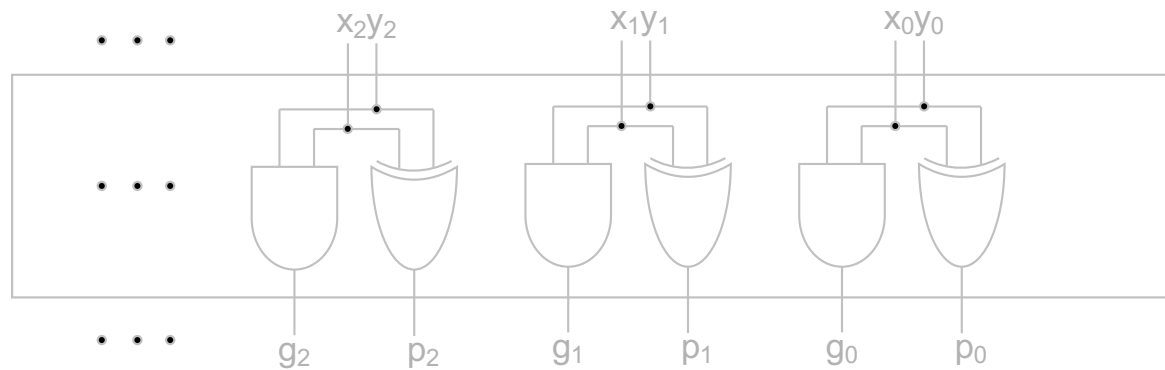
The CLA adder has the following 3-stage structure:





Carry Look Ahead adders

- The pre-calculation stage is implemented using the equations for p_i , g_i shown at a previous slide:





Carry Look Ahead adders

- The carry calculation stage is implemented using the equations produced when unfolding the recursive equation:

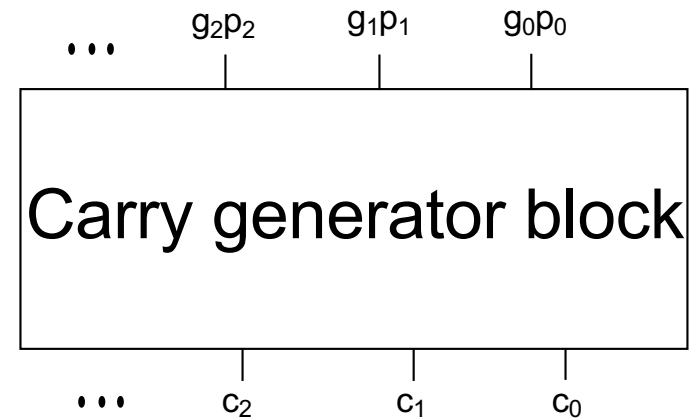
$$c_i = g_i + p_i \cdot c_{i-1} = g_i + a_i \cdot c_{i-1}$$

$$c_0 = g_0$$

$$c_1 = g_1 + p_1 \cdot g_0$$

$$\begin{aligned} c_2 &= g_2 + p_2 \cdot c_1 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0) \\ &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 \end{aligned}$$

etc ...





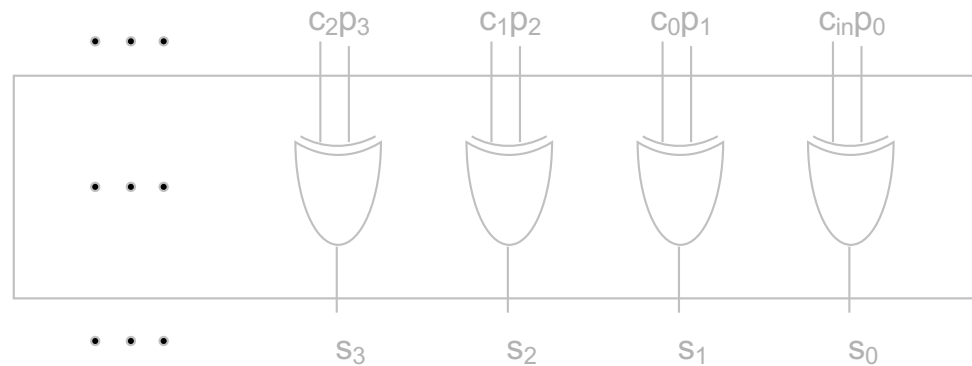
Carry Look Ahead adders

- The final sum calculation stage is implemented using the carry and propagate bits c_i, p_i :

$$s_i = p_i \oplus c_{i-1}, \quad \text{with } p_i = x_i \oplus y_i$$

Note:

$$s_i = g_i + a_i \cdot c_{i-1}, \quad \text{with } a_i = x_i + y_i$$





Addition as a prefix sum problem.

The equations of the well known CLA adder can be formulated as a parallel prefix problem by employing a special operator “ \circ ”.

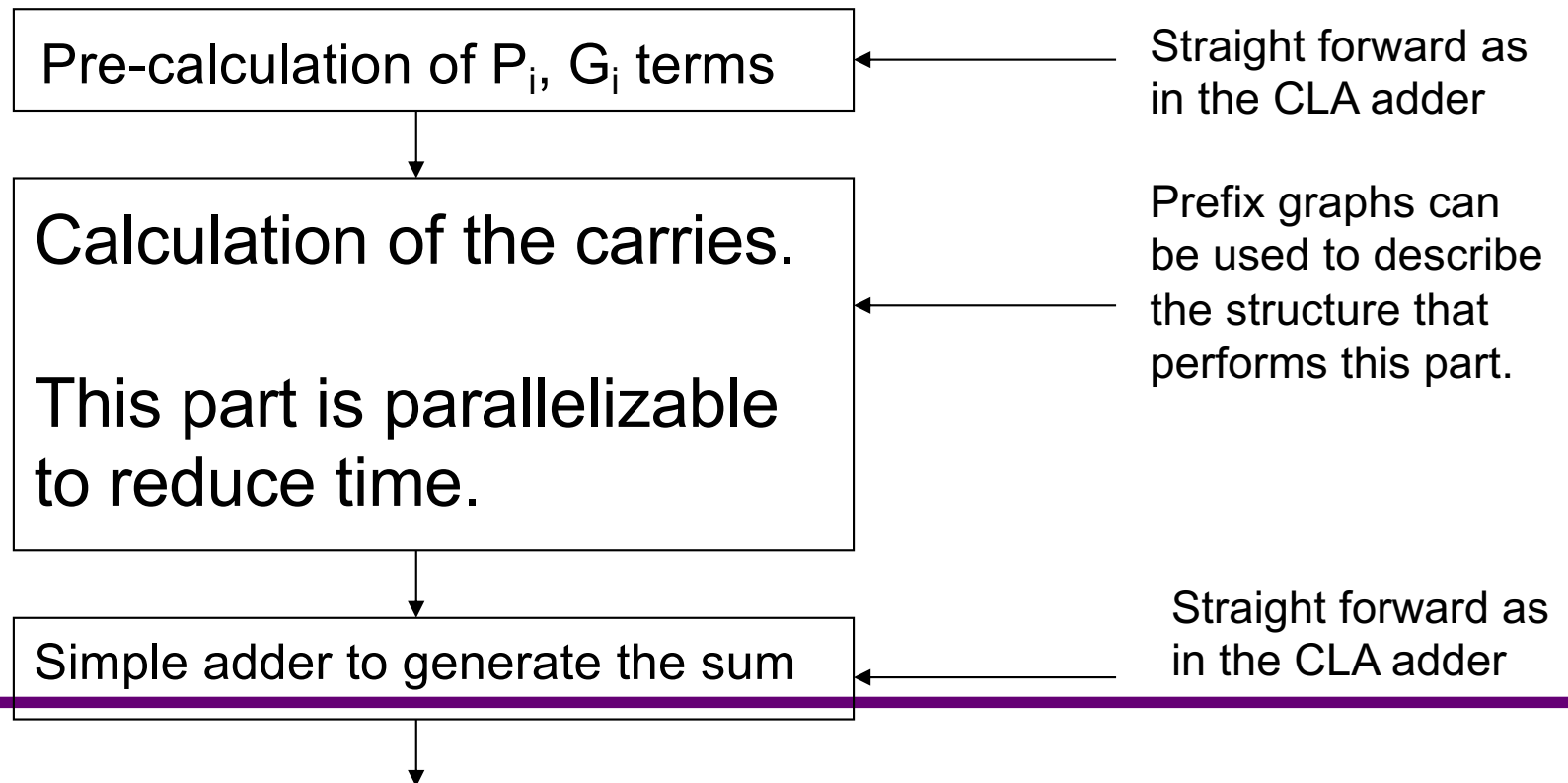
This operator is associative hence it can be implemented in a parallel fashion.

A Parallel Prefix Adder (PPA) is equivalent to the CLA adder... The two differ in the way their carry generation block is implemented.



Parallel Prefix Adders

- The parallel prefix adder employs the 3-stage structure of the CLA adder. The improvement is in the carry generation stage which is the most intensive one:

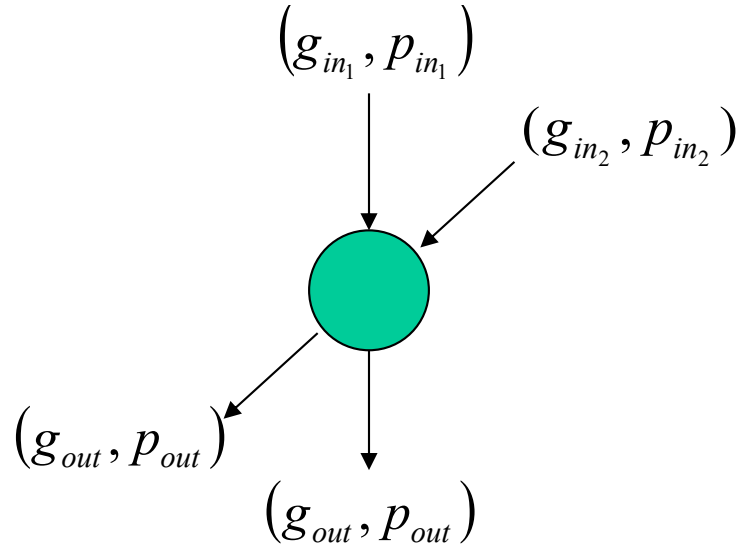




Calculation of carries – Prefix Graphs

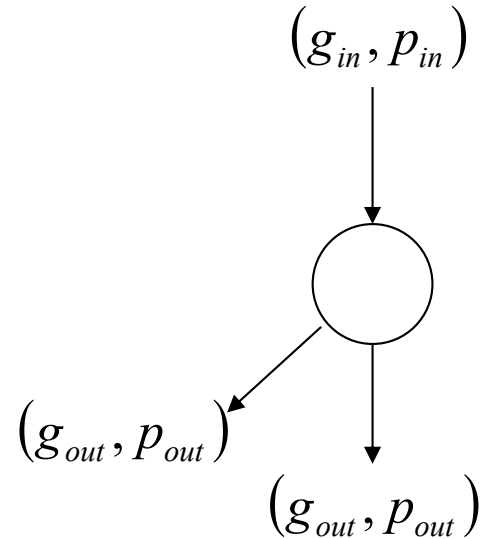
The components usually seen in a prefix graph are the following:

processing component:



$$(g_{out}, p_{out}) = (g_{in_1} + p_{in_1} \cdot g_{in_2}, p_{in_1} \cdot p_{in_2})$$

buffer component:

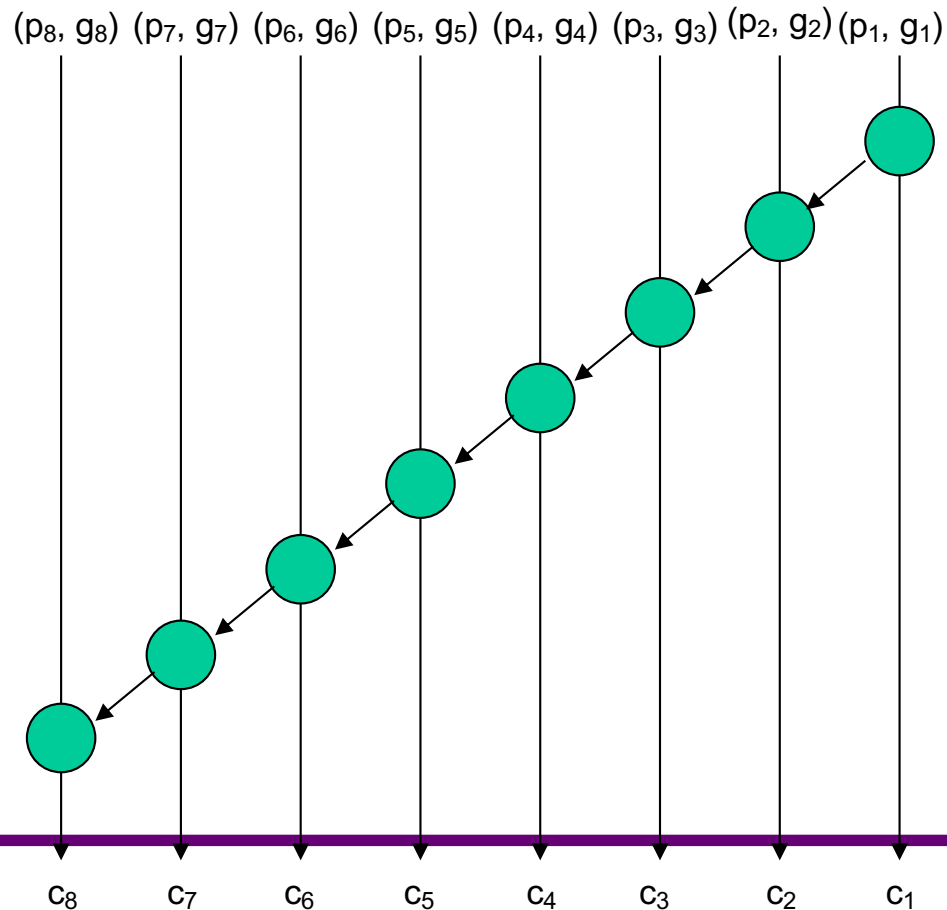


$$(g_{out}, p_{out}) = (g_{in}, p_{in})$$



Prefix graphs for representation of Prefix addition

- Example: serial adder carry generation represented by prefix graphs





Key architectures for carry calculation:

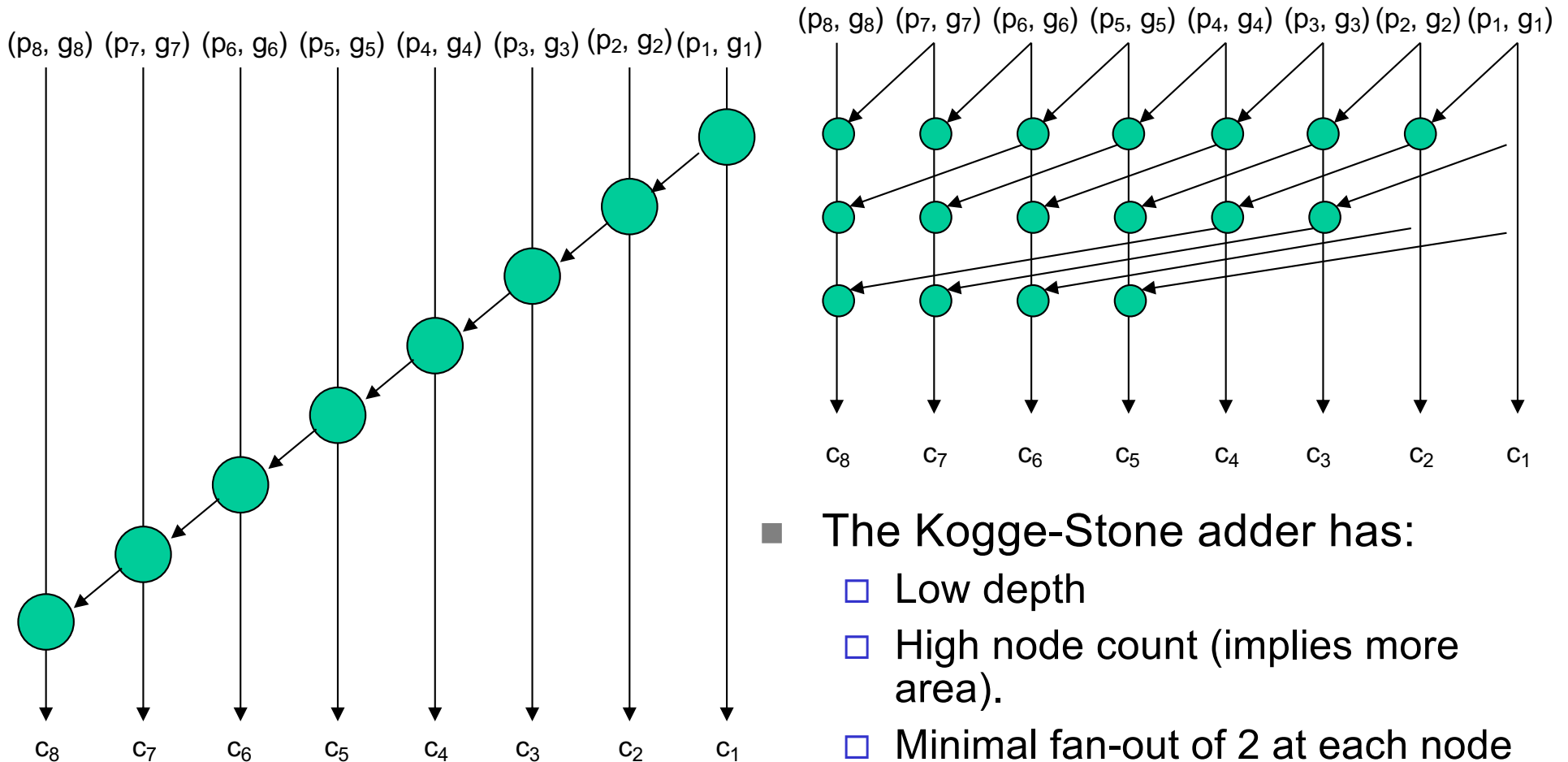
- 1960: J. Sklansky – conditional adder
- 1973: Kogge-Stone adder
- 1980: Ladner-Fisher adder
- 1982: Brent-Kung adder
- 1987: Han Carlson adder
- 1999: S. Knowles

Other parallel adder architectures:

- 1981: H. Ling adder
- 2001: Beaumont-Smith

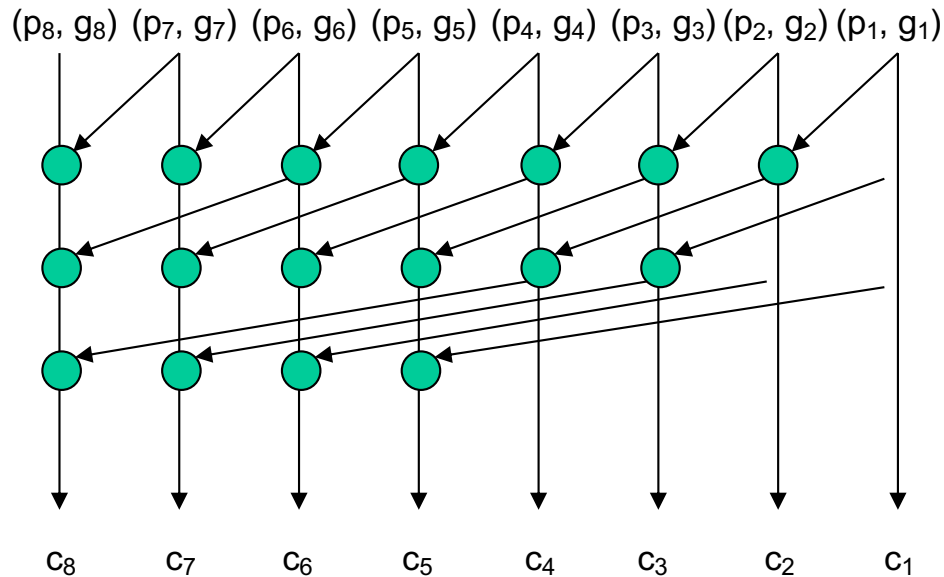


Kogge-Stone adder

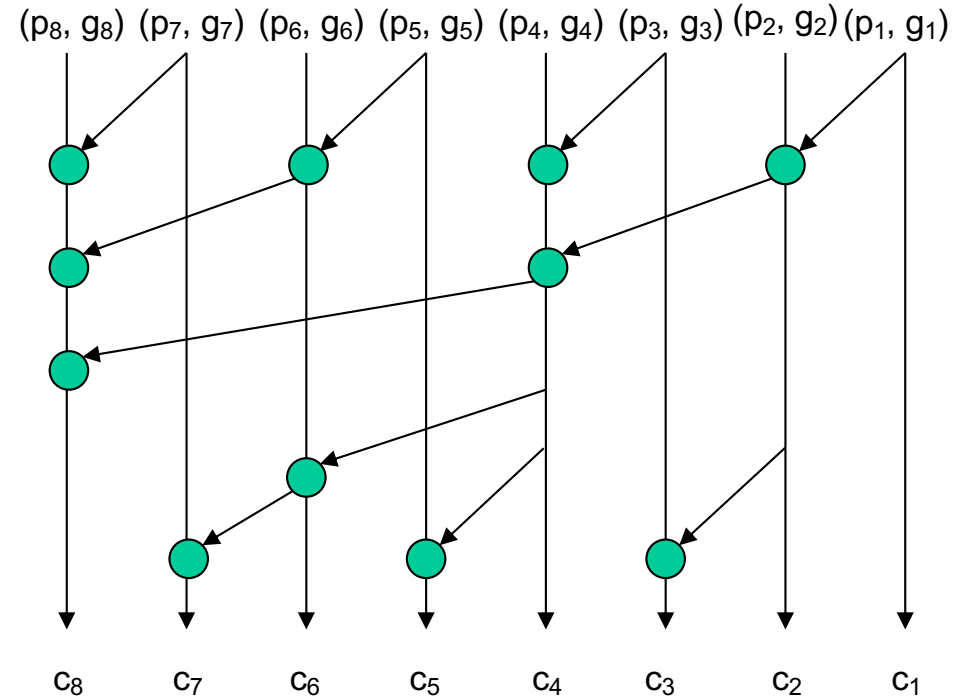




Brent-Kung adder



Kogge-Stone



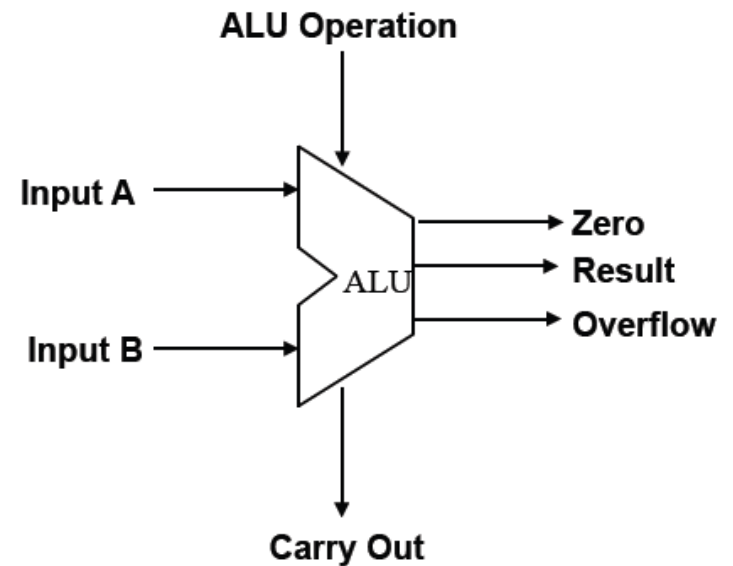
Brent-Kung

- The Brent-Kung adder is the *extreme* boundary case of:
 - Maximum logic depth in PP adders (implies longer calculation time).
 - Minimum number of nodes (implies minimum area).



Arithmetic Logic Unit (ALU)

- Performs a number of common arithmetic and logic operations
- Most fundamental and critical building block of the Central Processing Unit (CPU) of a computer
- A combinational logic circuit





Examples of ALU Functionality

Selection Bits	Main Function
0	ADD
1	SUB
2	NOT
3	OR
4	AND

- Realize basic logic and arithmetic operation
 - ADD, SUB, NOT, OR, AND
 - Also sometimes complex operation, e.g. set on less
- Handle inputs as signed or unsigned numbers
 - 2's complement number: $[-2^{N-1}, 2^{N-1}-1]$
- Also has zero detector and overflow detector





Shifting Operations

- Commonly used for division and other logical operations
 - For example, divide-by-2: right shift 1 bit, multiply-by-2: left shift by 1 bit
 - But you need to watch out the sign bit

- Logical Shifting: adding zero to vacated bits

011011 $\xrightarrow{\text{Left Shift}}$ 110110 011011 $\xrightarrow{\text{Right Shift}}$ 001101

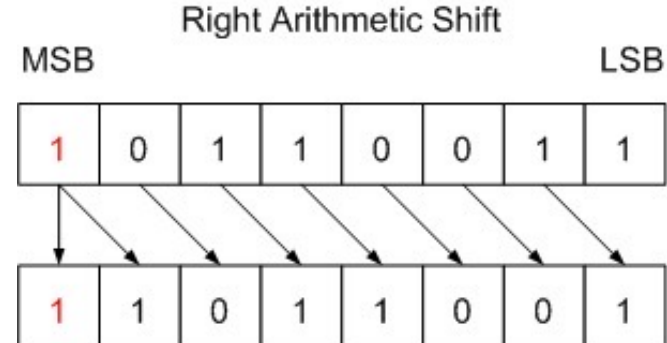
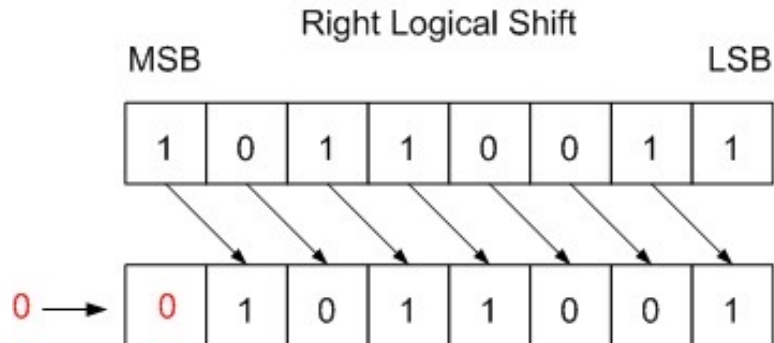
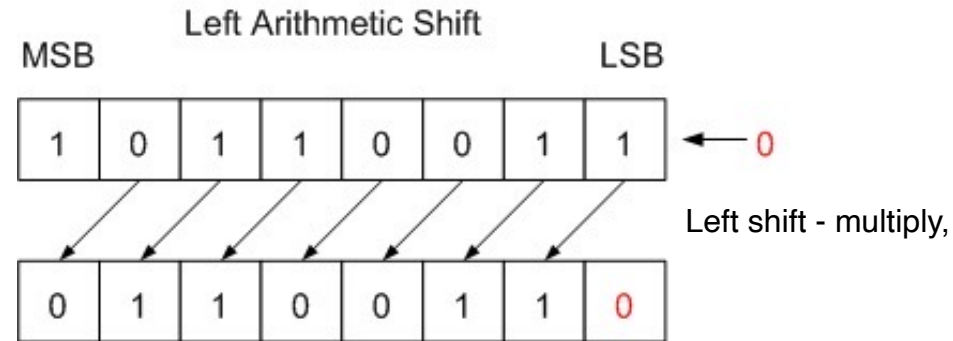
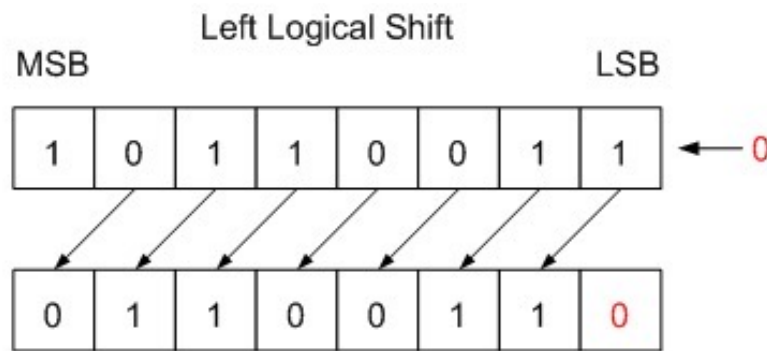
- Arithmetic Shifting: preserving sign bit

011011 $\xrightarrow{\text{Left Shift}}$ 110110 101011 $\xrightarrow{\text{Right Shift}}$ 110101



Shifting Operations

- Logical Shifting: adding zero to vacated bits
- Arithmetic Shifting:
 - Same as logical shift for left shift
 - Preserve sign bit for right shift





Barrel Shifters

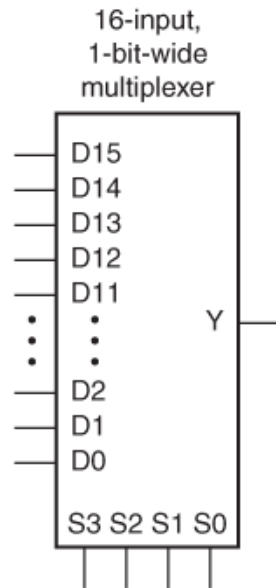
- Can choose shifting bits, direction, and functions (logical or arithmetic)

<i>Shift Type</i>	<i>Name</i>	<i>Code</i>	<i>Function</i>	<i>Note</i>
Left rotate	Lrotate	000	Vrol	Wrap-around
Right rotate	Rrotate	001	Vror	Wrap-around
Left logical	Llogical	010	Vsll	0 into LSB
Right logical	Rlogical	011	Vslr	0 into MSB
Left arithmetic	Larith	100	Vsla	0 into LSB
Right arithmetic	Rarith	101	Vsra	Replicate MSB

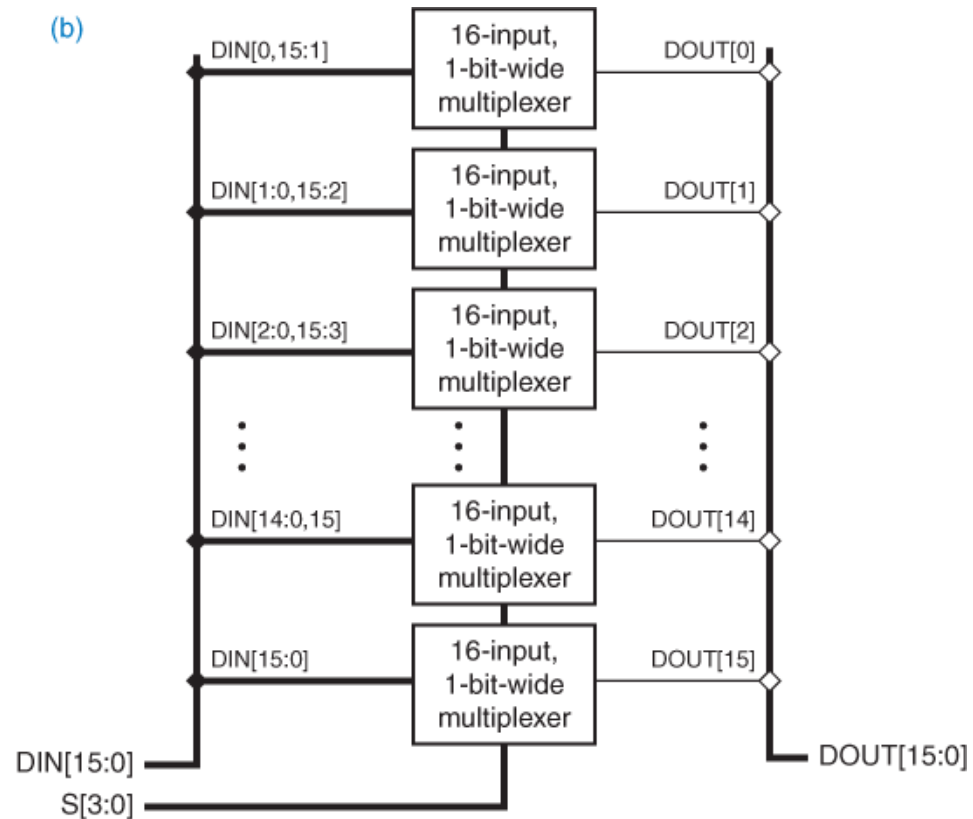


Barrel Shifters

(a)



(b)



- Use Multiplexer to choose which input bit to send out



Example Verilog Code

- S determines shift # of bits
 - DIR determines direction
 - Have not included “sign” consideration for arithmetic shift
- If {S[0], DIR} (this is two bits) == 2'b00 or 2'b01, do the following,
Then, if the two bits made up 2'b01, then do this line
Etc.
This is “don't care”

```
module Vrrorl16 (DIN, S, DIR, DOUT);
input [15:0] DIN; // Data inputs
input [3:0] S; // Shift amount, 0-15
input DIR; // Shift direction, 0=>L, 1=>R
output [15:0] DOUT; // Data bus output
reg [15:0] DOUT, X, Y, Z;
always @ (*) begin
    case ( {S[0], DIR} )
        2'b00, 2'b01 : X = DIN;
        2'b10 : X = {DIN[14:0], DIN[15]};
        2'b11 : X = {DIN[0], DIN[15:1]};
        default : X = 16'bx; This is “don't care”
    endcase
    case ( {S[1], DIR} )
        2'b00, 2'b01 : Y = X;
        2'b10 : Y = {X[13:0], X[15:14]};
        2'b11 : Y = {X[1:0], X[15:2]};
        default : Y = 16'bx;
    endcase
    case ( {S[2], DIR} )
        2'b00, 2'b01 : Z = Y;
        2'b10 : Z = {Y[11:0], Y[15:12]};
        2'b11 : Z = {Y[3:0], Y[15:4]};
        default : Z = 16'bx;
    endcase
    case ( {S[3], DIR} )
        2'b00, 2'b01 : DOUT = Z;
        2'b10, 2'b11 : DOUT = {Z[7:0], Z[15:8]};
        default : DOUT = 16'bx;
    endcase
end endmodule
```

This is a cascade/relay
Reg: DOUT is protected by orchestration

- Refer to 8.2 for more details



Multiplier

- Multiplier is another critical building block for arithmetic operation.
- It is usually a separate module outside of ALU due to its complexity
- Becomes a bottleneck of speed and power consumption
 - A lot of different architectures have been explored

Multiplier usually sequential



Combinational Multiplier

Basic Concept

product of 2 4-bit numbers
is an 8-bit number

multiplicand 1101 (13)

multiplier 1011 (11)

$$\begin{array}{r} \text{Partial products} \quad * \\ \quad \quad \quad 1101 \\ \quad \quad 1101 \\ \quad 0000 \\ \quad 1101 \\ \hline 10001111 \quad (143) \end{array}$$



Combinational Multiplier

Partial Product Accumulation

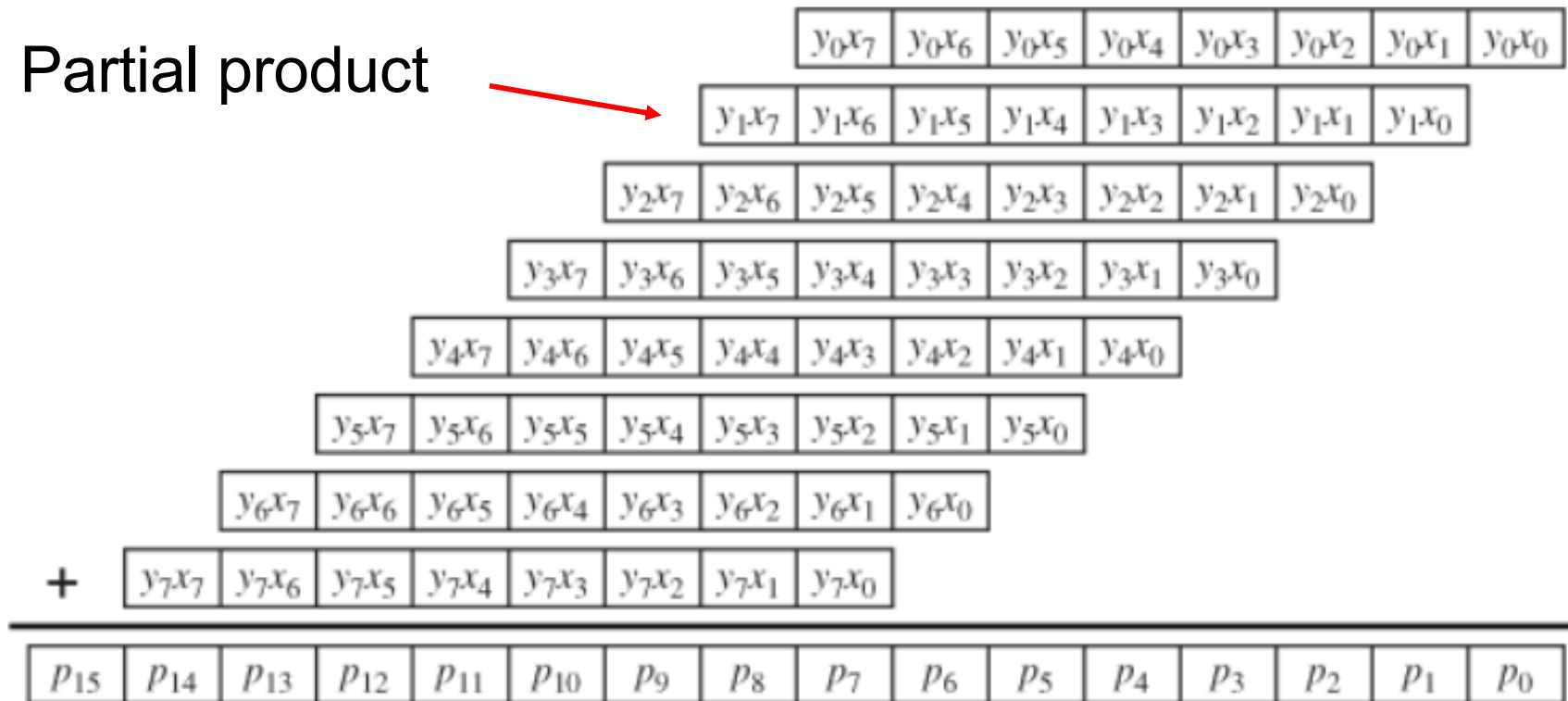
				A3	A2	A1	A0
				B3	B2	B1	B0
				<hr/>			
				A2 B0	A2 B0	A1 B0	A0 B0
			A3 B1	A2 B1	A1 B1	A0 B1	
		A3 B2	A2 B2	A1 B2	A0 B2		
	A3 B3	A2 B3	A1 B3	A0 B3			
<hr/>							
S7	S6	S5	S4	S3	S2	S1	S0
<hr/>							

Each term is an and
Then sum over all the terms



Multiplication Calculation

Partial product

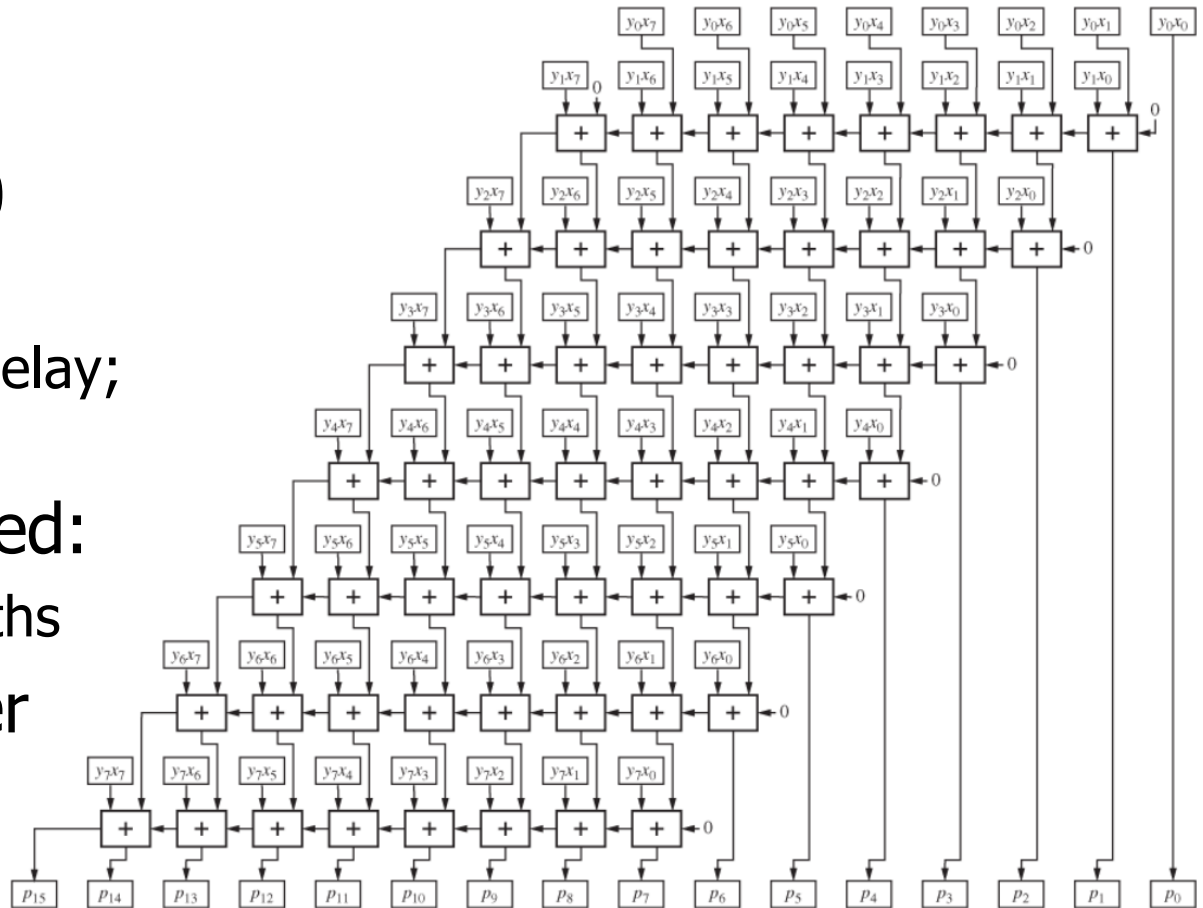


- Array of “AND” to general “partial product”
- Addition of multiple inputs



Circuit Implementation

- Worst delay path:
traverses through 20
adder blocks
 - Two types of adder delay;
what's the caveat?
- Hard to improve speed:
 - A lot of long logic paths
- Huge area and power
- Also called “array
multiplier” or
combinational multiplier



8 bit x 8 bit multiplier



Multiplier Optimization

- Hard to improve the speed due to large number of critical paths
- Sequential Multiplier:
 - Gatekeeper elements break the full summation into 3 parts,
 - Break into multiple stages of pipelined operations
- Braun Multiplier:
 - Change Carry signal to next stage (Textbook 8.3.1)
- Booth Encoding Multiplier (ECE 391):
 - Faster but much more costly
- Not a focus on this class



Verilog Code of Multiplier

```
module Vrmul8x8i(X, Y, P);  
    input [7:0] X, Y;  
    output [15:0] P;  
    assign P = X * Y; Combinational, no hinting of orchestration/time  
endmodule
```

- This simple code will work most of time
 - But not highly optimized
 - For high performance design, you should define the operations line by line
- Synthesis tool recognizes "*" and uses built-in functions to generate multiplier