Parallel Adders

Introduction

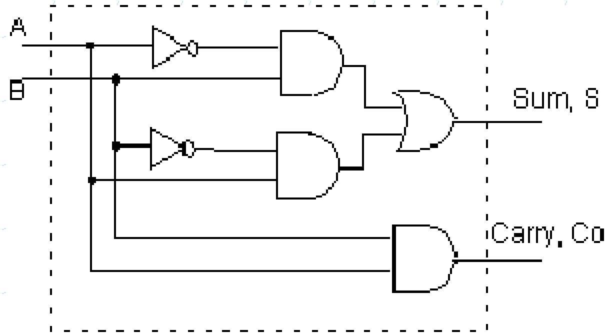
- Binary addition is a fundamental operation in most digital circuits
- There are a variety of adders, each has certain performance.
- Each type of adder is selected depending on where the adder is to be used.

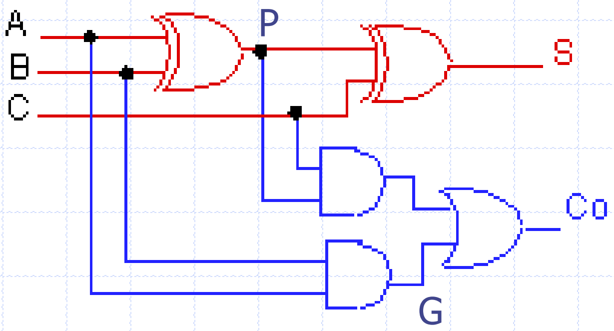
Adders

- Basic Adder Unit
- Ripple Carry Adder
- Carry Skip Adders
- Carry Look Ahead Adder
- Carry Select Adder
- Pipelined Adder
- Manchester carry chain adder
- Multi-operand Adders
- Pipelined and Carry save adders

Basic Adder Unit

- A combinational circuit that adds two bits is called a half adder
- A full adder is one that adds three bits, the third produced from a previous addition operation





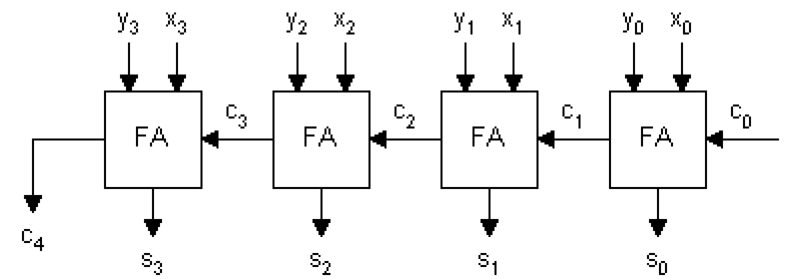
2. A brief introduction to Ripple Carry Adder

- Reuse carry term to implement full adder

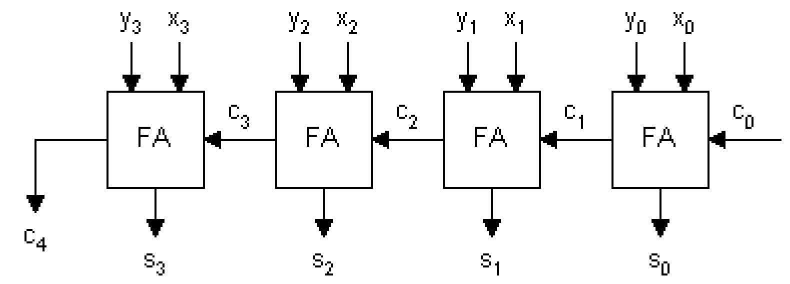
Figure 2.2 1bit full adder CMOS complementary implementation

Ripple Carry Adder

- The ripple carry adder is constructed by cascading full adder blocks in series
- The carryout of one stage is fed directly to the carry-in of the next stage
- For an n-bit parallel adder, it requires n full adders



Ripple Carry Drawbacks



- Not very efficient when large bit numbers are used
- Delay increases linearly with the bit length

•Delay

Critical path in a 4-bit ripple-carry adder

Note: delay from carry-in to carry-out is more important than from A to carry-out or from carry-in to SUM, because the carry-propagation chain will determine the latency of the whole circuit for a Ripple-Carry adder.

Delay

The latency of a 4-bit ripple carry adder can be derived by considering the above worst-case signal propagation path. We can thus write the following expression:

$T_{RCA\text{-}4bit} = T_{FA}(A0,B0 \rightarrow Co) + T_{FA}(Cin \rightarrow C1) + T_{FA}(Cin \rightarrow C2) + T_{FA}(Cin \rightarrow S3)$

And, it is easy to extend to k-bit RCA:

$T_{RCA\text{-}4bit} = T_{FA}(A0,B0 \rightarrow Co) + (K\text{-}2) * T_{FA}(Cin \rightarrow Ci) + T_{FA}(Cin \rightarrow S_{k-1})$

Comparison of CMOS and TG Logic

- Simulation result

CCT Logic Struture	Area (μm^2)	Total# of Transistor	Input tr,tf (ps)	Tp(max) (ns)	Power (mW) Average	Power (mW) Max	AT	AT^2	DP
CMOS (Normal)	305.76	112	10	1.3	0.695	19.5	397.49	516.73	0.9035
			250	1.3	0.784	9.06	397.49	516.73	1.0192
CMOS (Optimized)	262.08	108	10	0.9	0.33	13.3	235.87	212.28	0.297
			250	0.9	0.372	4.94	235.87	212.28	0.3348
TG (Normal)	280.8	104	10	1.7	0.624	22.2	477.36	811.51	1.0608
			250	1.8	0.749	7.98	505.44	909.79	1.3482
TG (Optimized)	212.16	100	10	1.4	0.452	17.3	297.02	415.83	0.6328
			250	1.5	0.504	5.91	318.24	477.36	0.756

4-bit RCA performance comparison of CMOS and TG logic (min size)

Comparison of CMOS and TG Logic

- Simulation result

CCT Logic Struture	Area (μm^2)	Transistor	Input tr,tf (ps)	Tp(max) (ns)	Power (mW) Average	Power (mW) Max	AT	AT^2	DP
CMOS (2/1)	393.12	108	10	0.8	0.695	19.5	314.50	251.60	0.556
			250	0.8	0.784	9.06	314.50	251.60	0.6272
TG (2/1)	280.8	100	10	0.9	0.452	17.3	252.72	227.45	0.4068
			250	1	0.504	5.91	280.80	280.80	0.504

4-bit RCA performance comparison of CMOS and TG logic (Wp/Wn=2/1)

Carry Look-Ahead Adder

- Calculates the carry signals in advance, based on the input signals

Boolean Equations

$P_i = A_i \oplus B_i$	Carry propagate
$G_i = A_i B_i$	Carry generate
$S_i = P_i \oplus C_i$	Sum
$C_{i+1} = G_i + P_i C$	Carry out

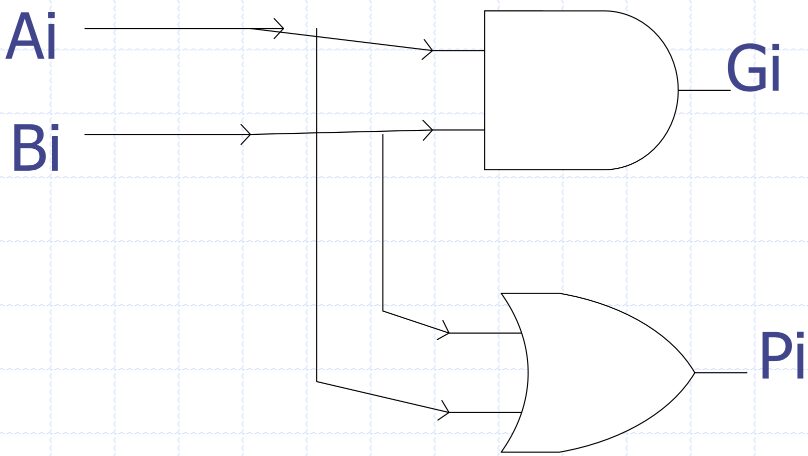
- Signals P and G only depend on the input bits

Carry Look-Ahead Adder

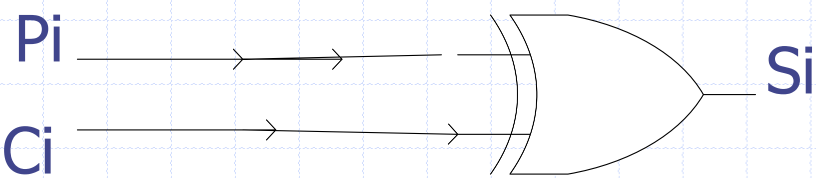
- Applying these equations for a 4-bit adder:

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1C_1 = G_1 + P_1(G_0 + P_0C_0) = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2C_2 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$C_4 = G_3 + P_3C_3 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$$

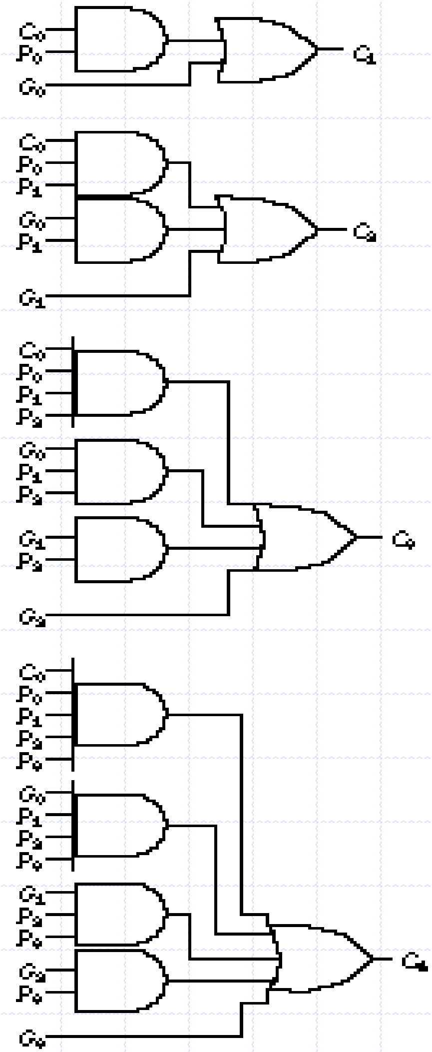
Carry Look-Ahead Structure



Propagate/Generate Generator

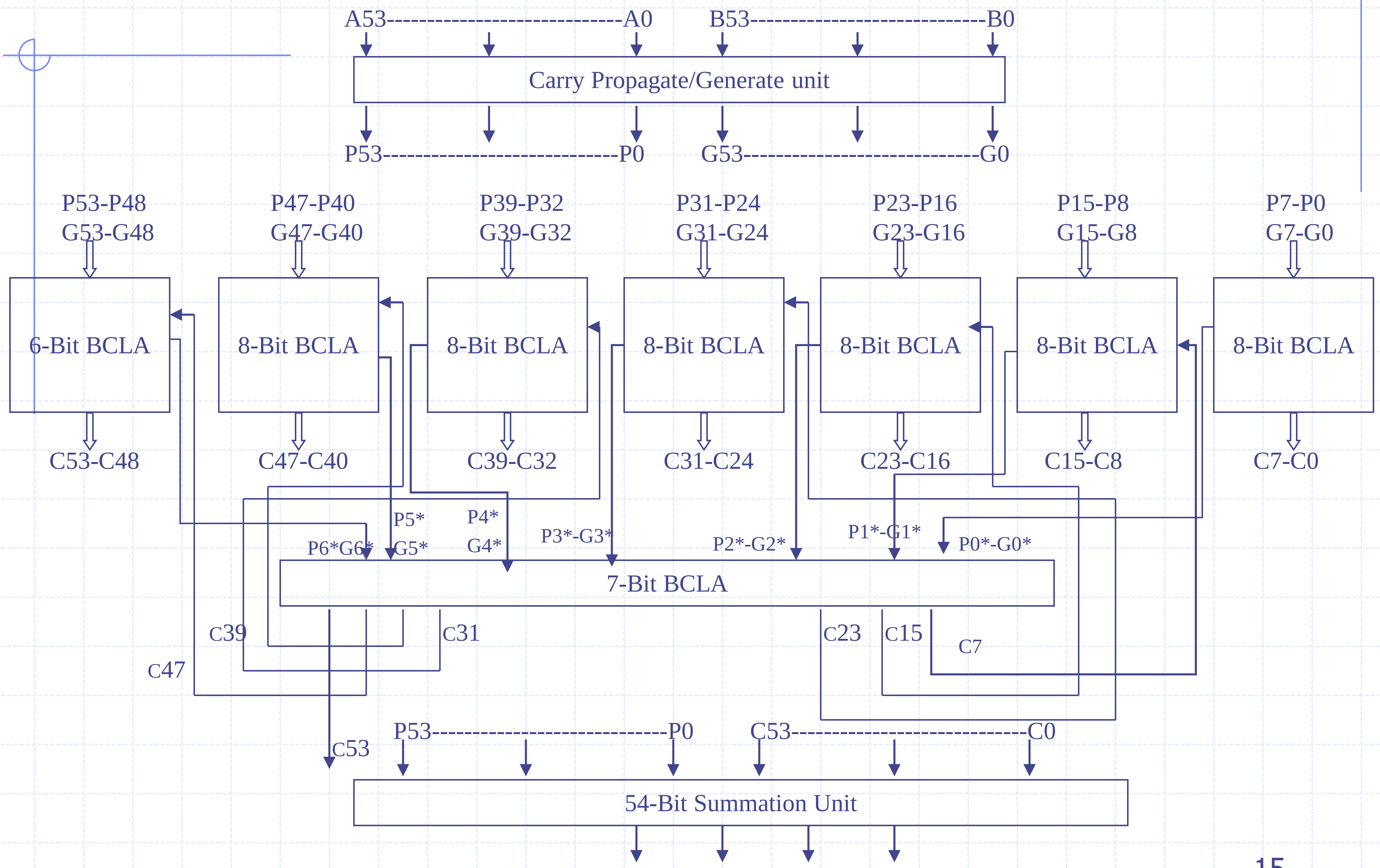


Sum generator

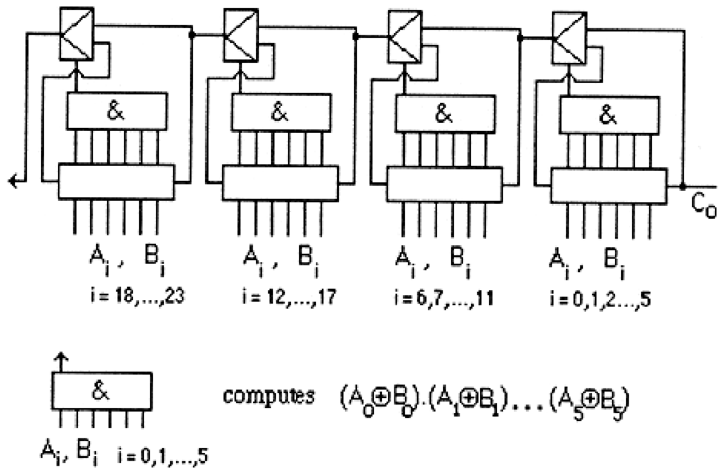


Look-Ahead Carry generator

Example Design of a large Carry Look-ahead Adder
Equations are in the Notes



Carry Skip Adders



- Are <u>composed of ripple carry adder blocks</u> of fixed size* and a carry skip chain
- The size of the blocks are chosen so as to minimize the longest <u>life of a carry</u>

Carry Skip Mechanics

<u>Boolean Equations</u>

Carry Propagate: $P_i = A_i \oplus B_i$

Sum: $S_i = P_i \oplus C_i$

Carry Out: $C_{i+1} = A_i B_i + P_i C_i$

<u>Worthwhile to note:</u>

If $\underline{A_i = B_i}$ then $P_i = 0$, making the carry out, C_{i+1}, depend only on A_i and B_i → $\underline{C_{i+1} = A_i B_i}$

- $C_{i+1} = 0$ if $A_i = B_i = 0$
- $C_{i+1} = 1$ if $A_i = B_i = 1$

Alternatively if $\underline{A_i \neq B_i}$ then $P_i = 1$ → $\underline{C_{i+1} = C_i}$

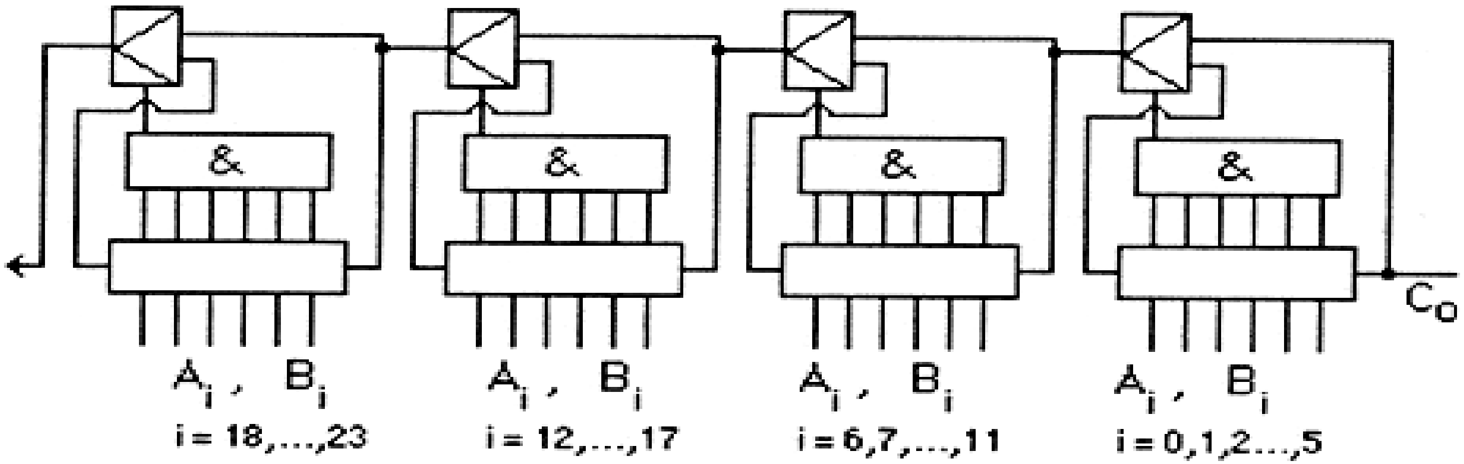
Carry Skip (example)

Two Random Bit Strings:

A	10100	01011	10100	01011
B	01101	10100	01010	01100
	block 3	block 2	block 1	block 0

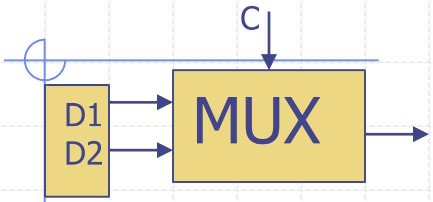
- compare the two binary strings inside each block
- If all the bits inside are unequal, block 2, then the carry in from block 1 is propagated to block 3
- Carry-ins from block 2 receive the carry in from block 1
- If there exists a pair of bits that is equal carry skip mechanism fails

Carry Skip Chain

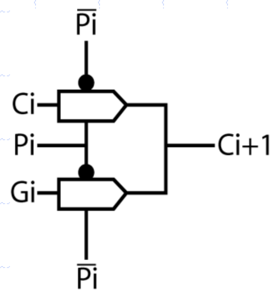


computes $(A_0 \oplus B_0).(A_1 \oplus B_1)...(A_5 \oplus B_5)$

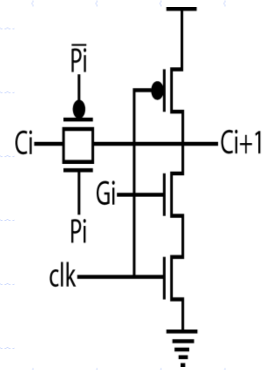
Various Implementations of Multiplexer (MUX)



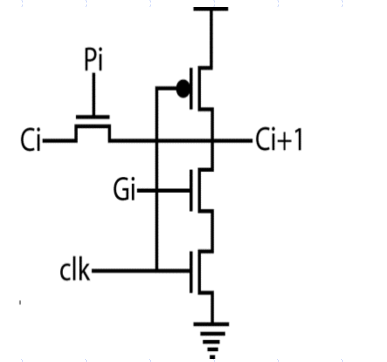
MUX



MUX with Ci and Gi



MUX, Implementation

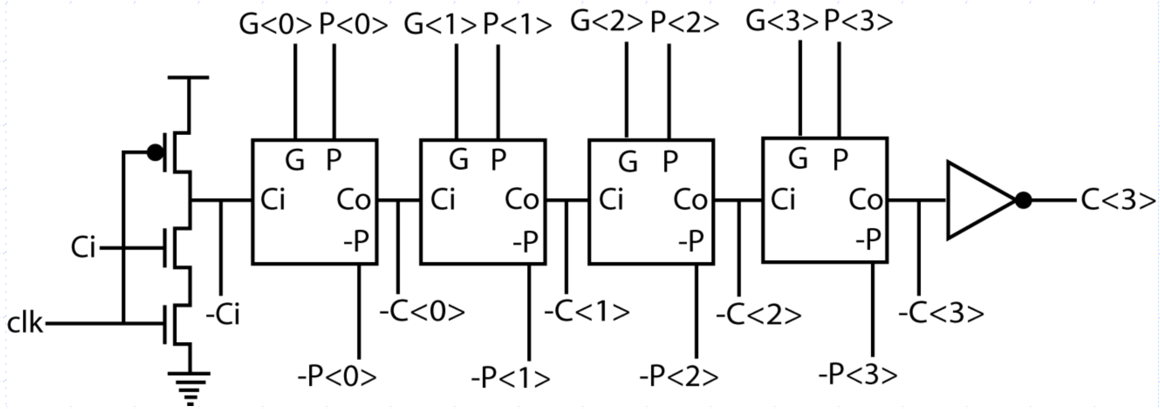


MUX Implementation

Boolean Equations:

1) $G_i = A_i B_i$ --carry generate of i^{th} stage
2) $P_i = A_i \oplus B_i$ --carry propagate of i^{th} stage
3) $S_i = P_i \oplus C_i$ --sum of i^{th} stage
4) $C_{i+1} = G_i + P_i C_i$ --carry out of i^{th} stage

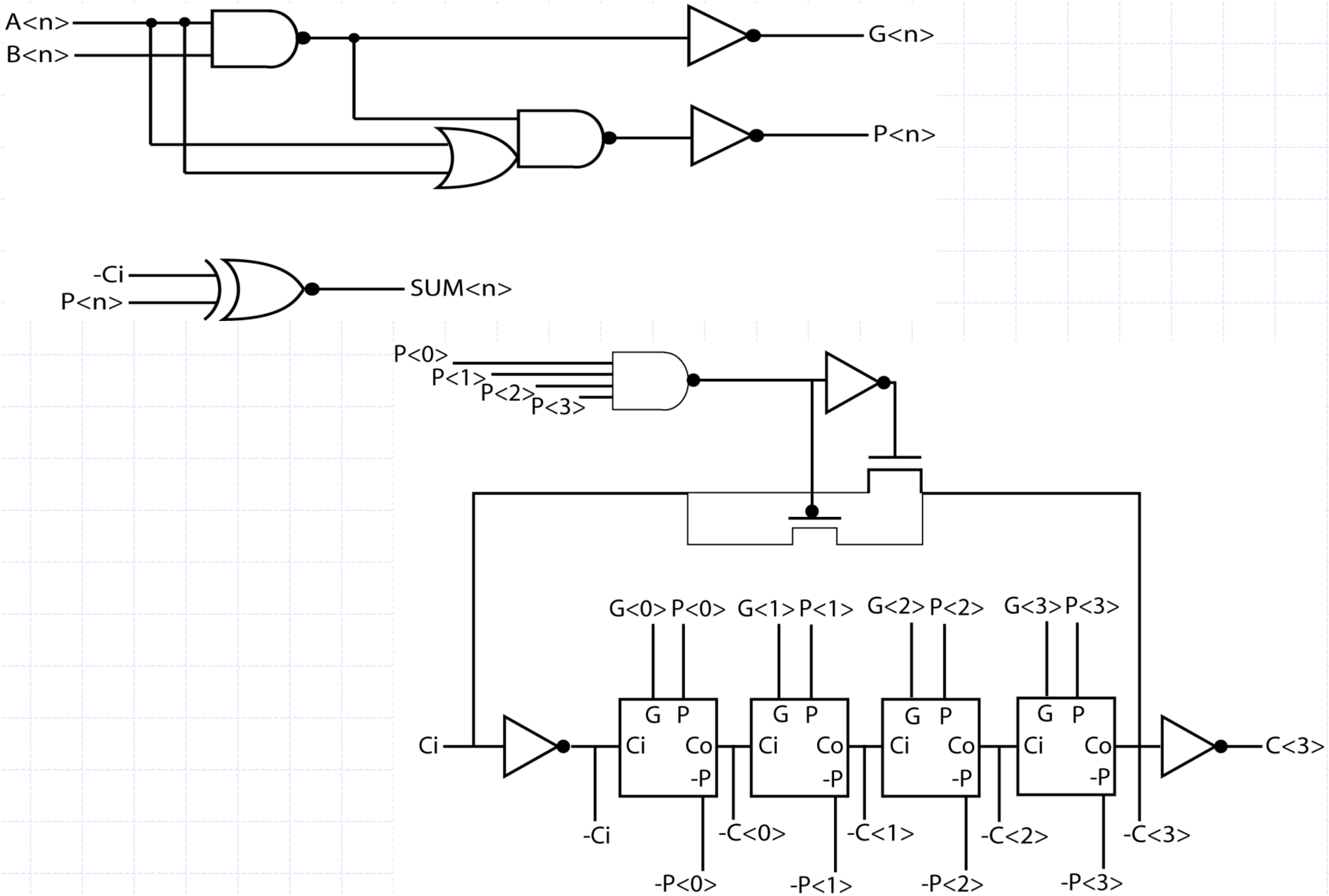
Manchester Carry Adder



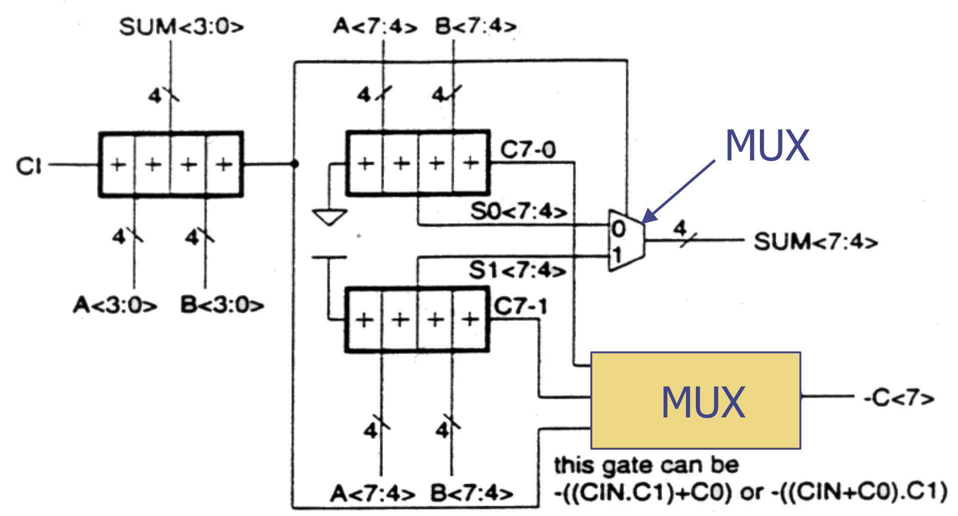
Boolean Equations:

1) $G_i = A_i B_i$ --carry generate of i^{th} stage
2) $P_i = A_i \oplus B_i$ --carry propagate of i^{th} stage
3) $S_i = P_i \oplus C_i$ --sum of i^{th} stage
4) $C_{i+1} = G_i + P_i C_i$ --carry out of i^{th} stage

Manchester Carry Adder with Skip Mechanism

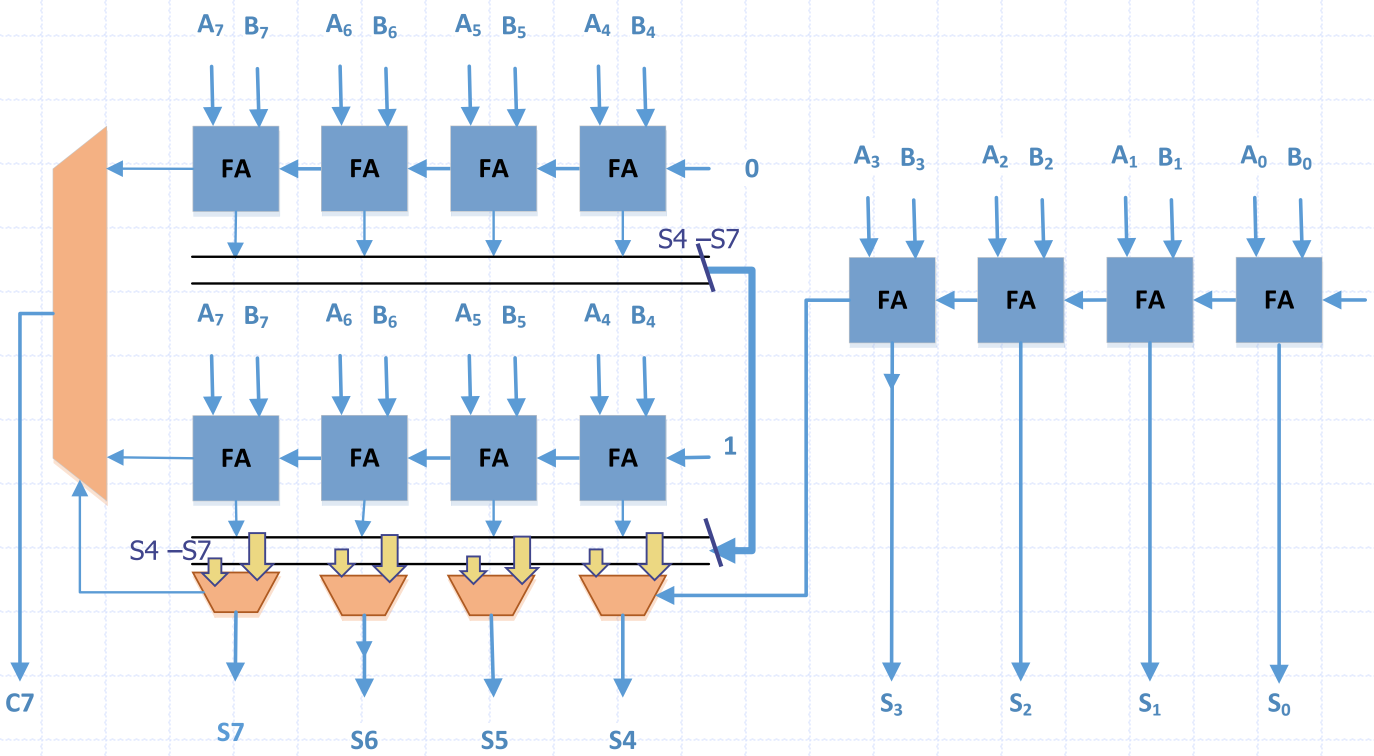


Carry Select Adder Example 8-bit Adder



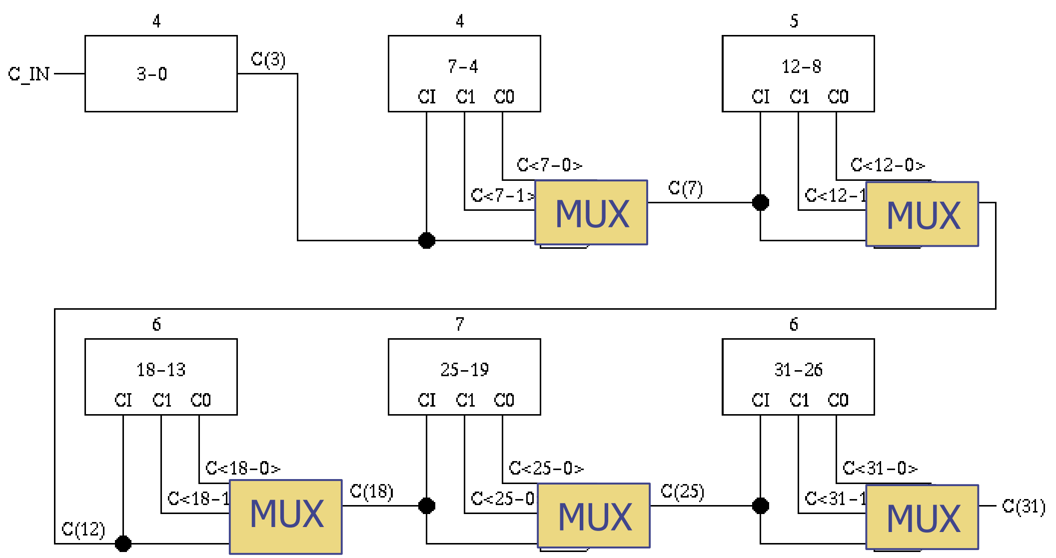
- It is <u>composed of 3 sections of one 4-bit and two four-bit ripple carry adders</u>.
- <u>Both sum and carry bits are calculated</u> for the two alternatives of the input carry, "0" and "1"

8-Bit Carry Select Adder



32 bit Carry Select (Mechanics)

- The carry out of each section determines the carry in of the next section, which then selects the appropriate ripple carry adder
- The very first section has a carry in of zero
- Time delay: time to compute first section + time to select sum from subsequent sections



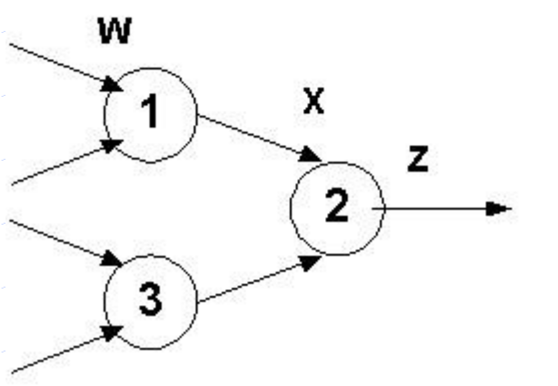
Carry Select Adder Design

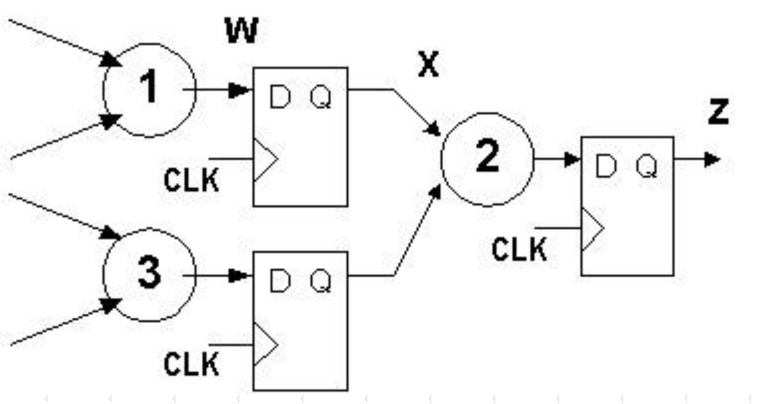
Linear Carry Select and Non_Linear Adders

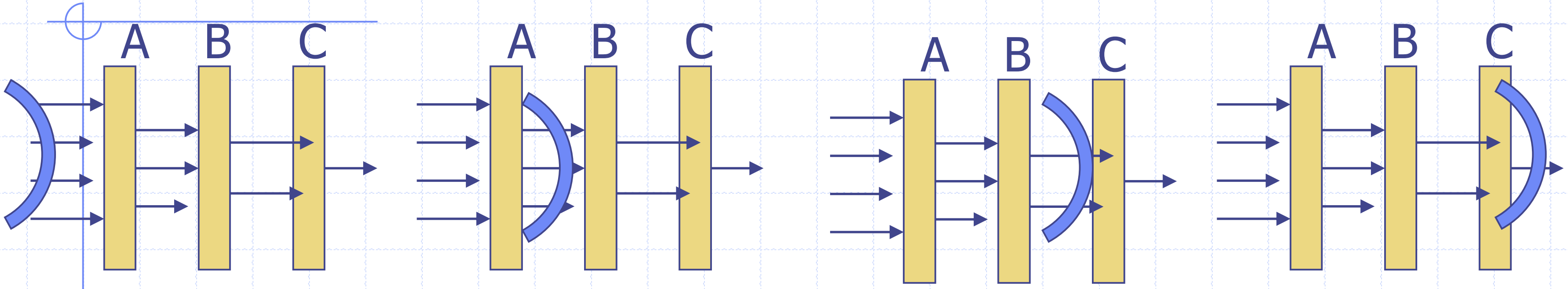
The linear carry-select adder is constructed by chaining a number of equal-length adder stages

The Non-Linear Adder is constructed according to the delay of the MUX and the Adder.

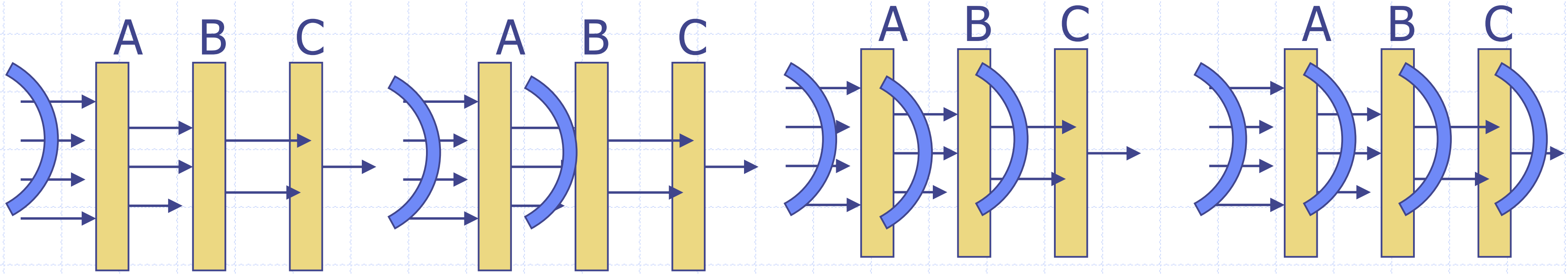
Multi-Operand and Pipelining





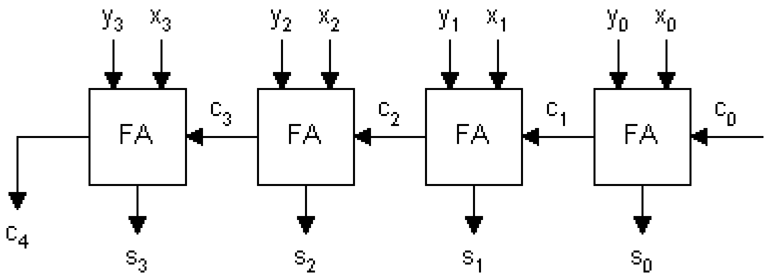


Signal propagation in serial blocks

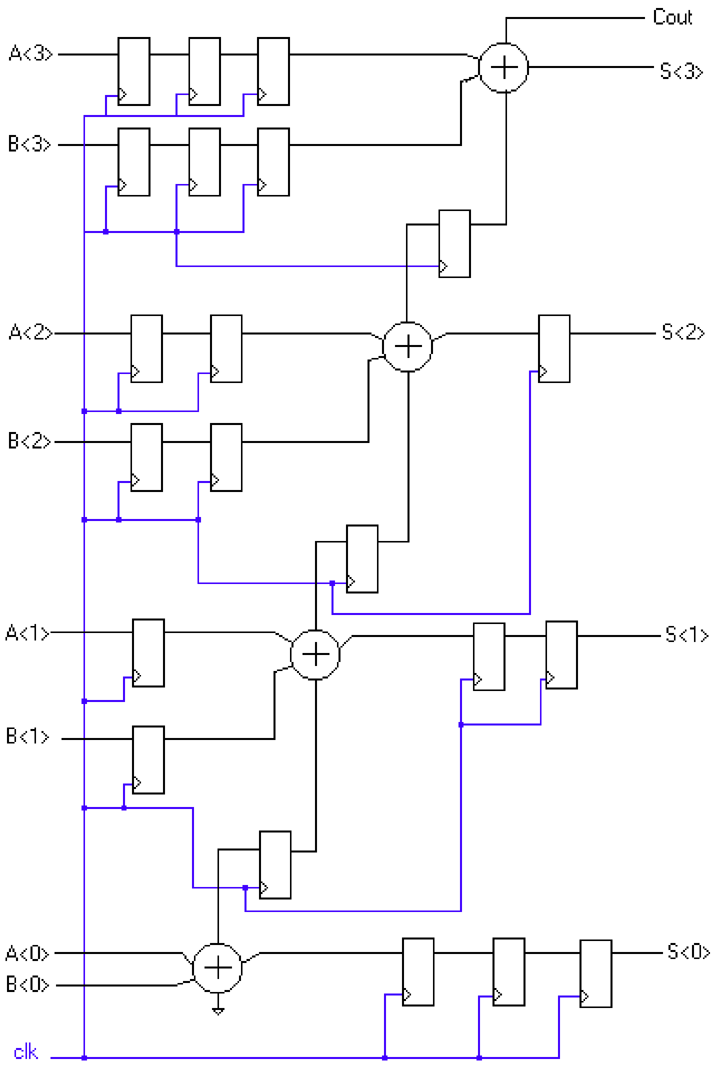


Signal Propagation in Pipelined serial Blocks

Pipelined Adder



- The added complexity of such a pipelined adder pays off if long sequences of numbers are being added.



Pipelined Adder

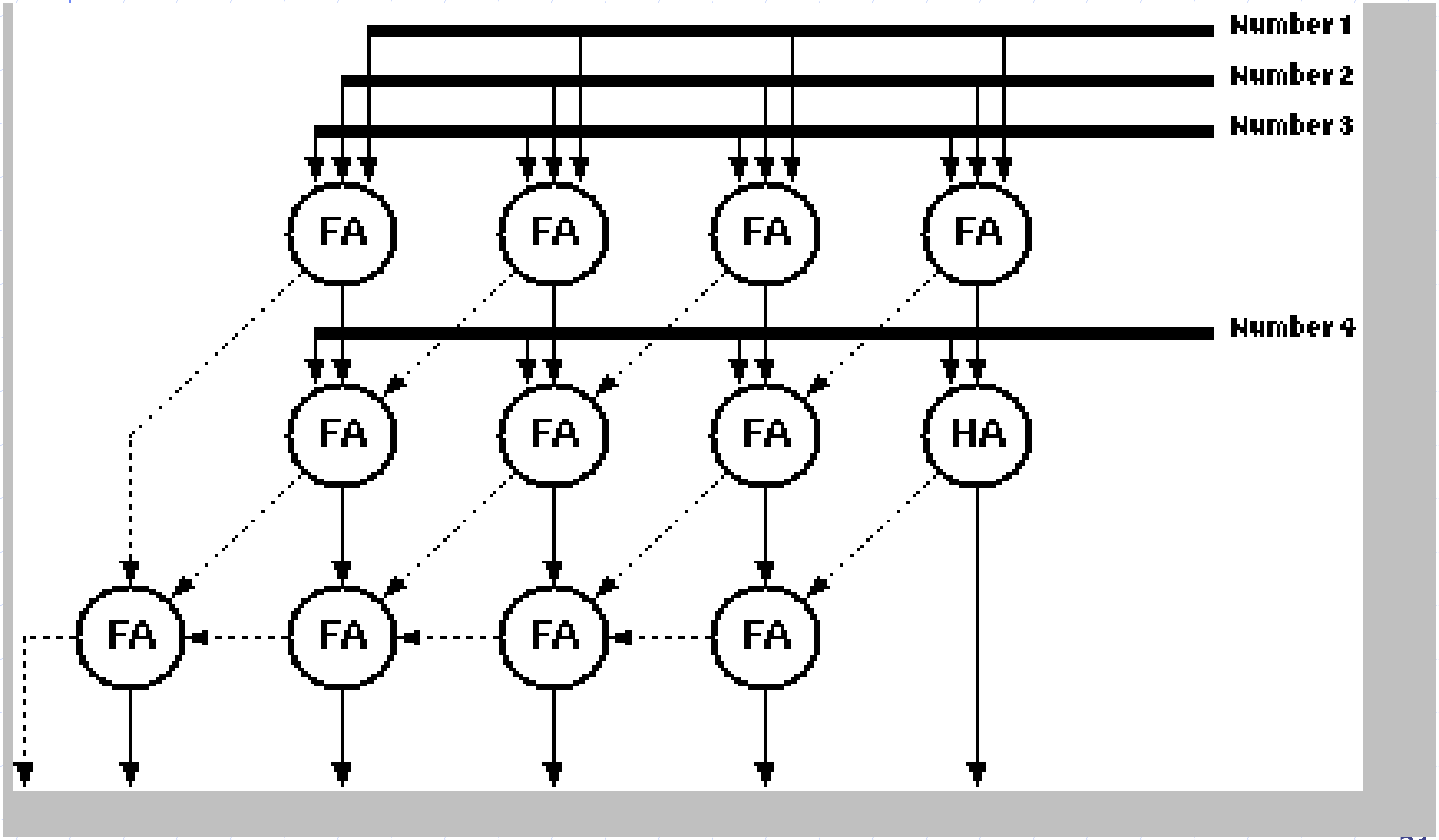
- Pipelining a design will increase its throughput
- The trade-off is the use of registers
- If pipelining is to be useful these three points has to be present:

  -It repeatedly executes a basic function.

  -The basic function must be divisible into independent *stages* having minimal overlap with each other.

  -The stages must be of similar complexity

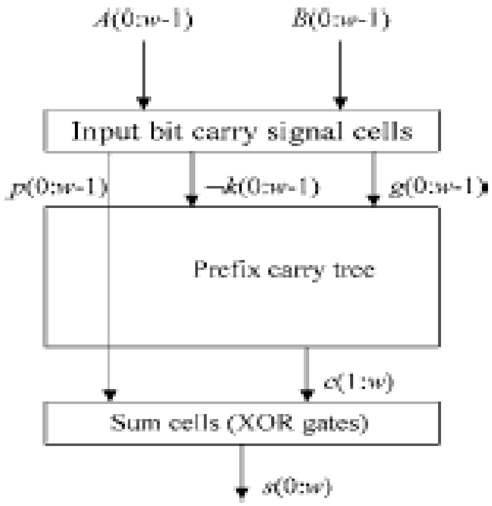
Carry Save adder



The rest of these slides are for information only

Parallel Prefix Adder[13,15,2]

The parallel prefix adder is a kind of carry look-ahead adders that accelerates a n-bit addition by means of a parallel prefix carry tree.



A block diagram of a prefix adder

Input bit propagate, generate, and not kill cells

$$p(i) = a(i) \oplus b(i)$$
$$g(i) = a(i) \wedge b(i)$$
$$\neg k(i) = a(i) \vee b(i)$$

Output sum cells

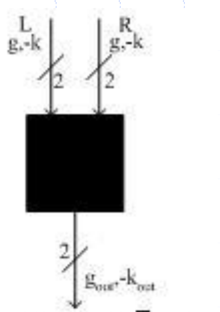
$$s(i) = p(i) \oplus c(i).$$

The prefix carry tree

$$G_z^x = G_z^y \vee \neg K_z^y \wedge G_{y-1}^x$$ G_z^x "group generate" signal across the bits from x up to z

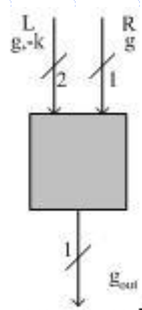
$$\neg K_z^x = \neg K_z^y \wedge \neg K_{y-1}^x$$ $\neg K_z^x$ "group not kill" signal across the bits from x up to z

$$c(i) = G_{i-1}^0 = G_{i-1}^y \vee \neg K_{i-1}^y \wedge G_{y-1}^0$$



$$g_{out} = g_L + \bar{k}_L \cdot g_R$$
$$\bar{k}_{out} = \bar{k}_L \cdot \bar{k}_R$$

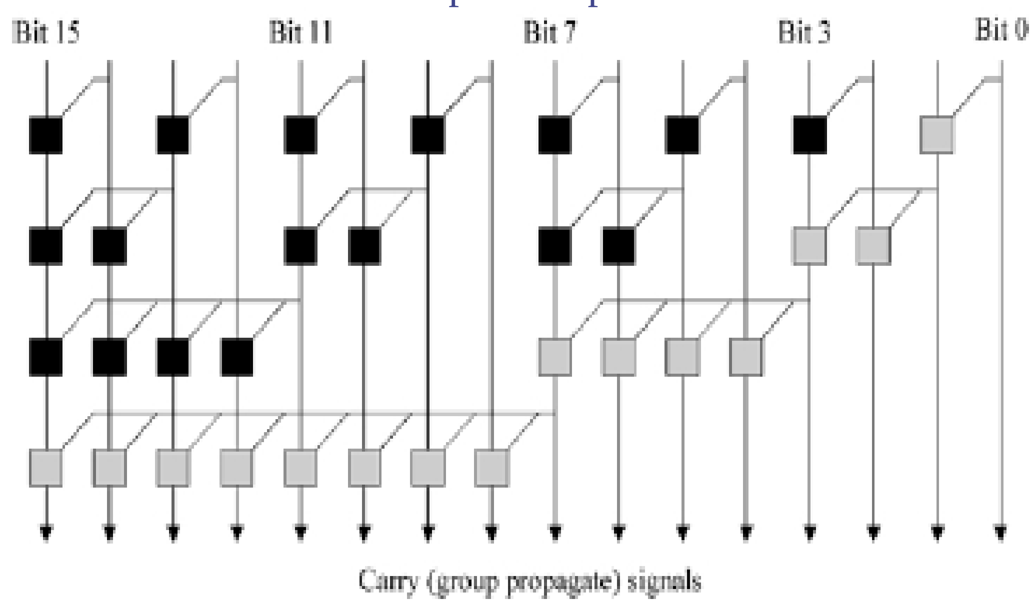
black cell



$$g_{out} = g_L + \bar{k}_L \cdot g_R$$

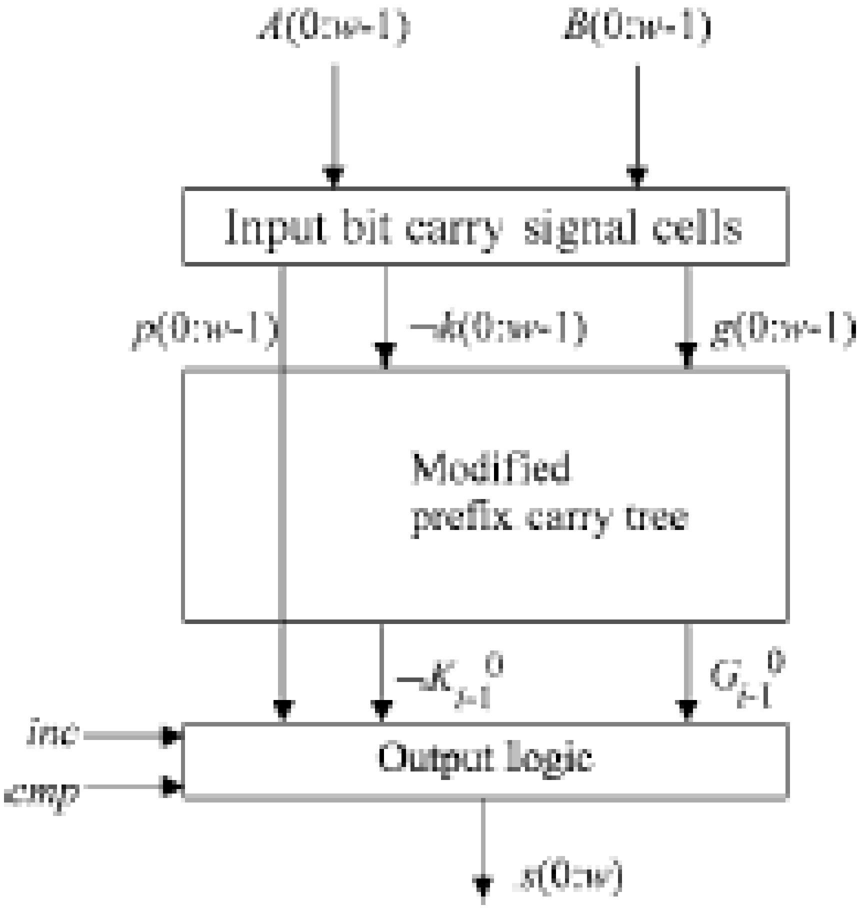
grey cell

16-bit Ladner-Fiacher parallel prefix tree



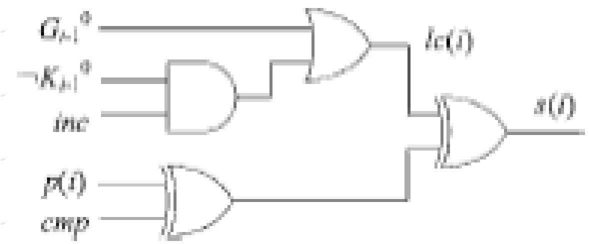
Flagged Prefix Adder[13,15]

Block diagram of a flagged prefix adder



The parallel prefix adder may be modified slightly to support late increment operations. If the output grey cells are replaced by black cells so that both G_{i-1}^0 and $\neg K_{i-1}^0$ signals are returned, a sum may be incremented readily.

$$lc(i) = G_{i-1}^0 \vee \neg K_{i-1}^0 \wedge inc$$



Output cell logic for the flagged prefix adder.

Output logic coding scheme.

cmp	*inc*	Result $(A + B)$	Result $(A + \neg B)$
0	0	$A + B$	$A - B - 1$
0	1	$A + B + 1$	$A - B$
1	0	$-(A + B + 1)$	$B - A$
1	1	$-(A + B + 2)$	$B - A - 1$

Reference List

[1] Reduced latency IEEE floating-point standard adder architectures. *Beaumont-Smith, A.; Burgess, N.; Lefrere, S.; Lim, C.C.;* Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on , 14-16 April 1999

[2] *M.D. Ercegovac and T. Lang,* "**Digital Arithmetic.**" San Francisco: Morgan Daufmann, 2004.

[3] Using the reverse-carry approach for double datapath floating-point addition. *J.D. Bruguera and T. Lang*. In Proceedings of the 15th IEEE Symposium on Computer Arithmetic, pages 203-10.

[4] A low power approach to floating point adder design. *Pillai, R.V.K.; Al-Khalili, D.; Al-Khalili, A.J.;* Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings. 1997 IEEE International Conference on, 12-15 Oct. 1997 Pages:178 – 185

[5] An IEEE compliant floating-point adder that conforms with the pipeline packet-forwarding paradigm. *Nielsen, A.M.; Matula, D.W.; Lyu, C.N.; Even, G.;* Computers, IEEE Transactions on, Volume: 49 , Issue: 1, Jan. 2000 Pages:33 - 47

[6] Design and implementation of the snap floating-point adder. *N. Quach and M. Flynn.* Technical Report CSL-TR-91-501, Stanford University, Dec. 1991.

[7] On the design of fast IEEE floating-point adders. *Seidel, P.-M.; Even, G.* Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on , 11-13 June 2001 Pages:184 – 194

[8] Low cost floating point arithmetic unit design. *Seungchul Kim; Yongjoo Lee; Wookyeong Jeong; Yongsurk Lee;* ASIC, 2002. Proceedings. 2002 IEEE Asia-Pacific Conference on, 6-8 Aug. 2002 Pages:217 - 220

[9] Rounding in Floating-Point Addition using a Compound Adder. *J.D. Bruguera and T. Lang.* Technical Report. University of Santiago de Compostela. (2000)

[10] Floating point adder/subtractor performing ieee rounding and addition/subtraction in parallel. *W.-C. Park, S.-W. Lee, O.-Y. Kown, T.-D. Han, and S.-D. Kim.* IEICE Transactions on Information and Systems, E79-D(4):297–305, Apr. 1996.

[11] Efficient simultaneous rounding method removing sticky-bit from critical path for floating point addition. *Woo-Chan Park; Tack-Don Han; Shin-Dug Kim;* ASICs, 2000. AP-ASIC 2000. Proceedings of the Second IEEE Asia Pacific Conference on , 28-30 Aug. 2000 Pages:223 – 226

[12] Efficient implementation of rounding units Burgess. *N.; Knowles, S.; Signals*, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on, Volume: 2, 24-27 Oct. 1999 Pages: 1489 - 1493 vol.2

[13] The Flagged Prefix Adder and its Applications in Integer Arithmetic. *Neil Burgess*. Journal of VLSI Signal Processing 31, 263–271, 2002

[14] A family of adders. *Knowles, S.;* Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on , 11-13 June 2001 Pages:277 – 281

[15] PAPA - packed arithmetic on a prefix adder for multimedia applications. *Burgess, N.;* Application-Specific Systems, Architectures and Processors, 2002. Proceedings. The IEEE International Conference on, 17-19 July 2002 Pages:197 – 207

[16] Nonheuristic optimization and synthesis of parallelprefix adders. *R. Zimmermann,* in Proc. Int.Workshop on Logic and Architecture Synthesis, Grenoble, France, Dec. 1996, pp. 123–132.

[17] Leading-One Prediction with Concurrent Position Correction. *J.D. Bruguera and T. Lang.* IEEE Transactions on Computers. Vol. 48. No. 10. pp. 1083-1097. (1999)

[18] Leading-zero anticipatory logic for high-speed floating point addition. *Suzuki, H.; Morinaka, H.; Makino, H.; Nakase, Y.; Mashiko, K.; Sumi, T.;* Solid-State Circuits, IEEE Journal of , Volume: 31 , Issue: 8 , Aug. 1996 Pages:1157 – 1164

[19] An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis. *Oklobdzija, V.G.;* Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, Volume: 2 , Issue: 1 , March 1994 Pages:124 – 128

[20] Design and Comparison of Standard Adder Schemes. *Haru Yamamoto, Shane Erickson,* CS252A, Winter 2004, UCLA

Comparisons

Adder	Number of CLBs	Delay (ns)	Area	Power Consumption (W)
Ripple-Carry	16	212.79	40.00	1.7318
Carry Look-Ahead	34	143.69	51.00	1.9668
Carry-Select	44	102.74	108.00	3.3595

- Which one should we choose?

Comparison of 64 bit Adders Using FPGA

For this comparison Synopsys tools were used to perform logic synthesis.

- The implemented VHDL codes for all the 64-bit adders are translated into net list files.
- The virtex2 series library, XC2V250-4_avg, is used in those 64-bit adders synthesis and targeting
- After synthesizing, the related power consumption, area, and propagation delay are reported.

Synthesis result parameter comparison listings:

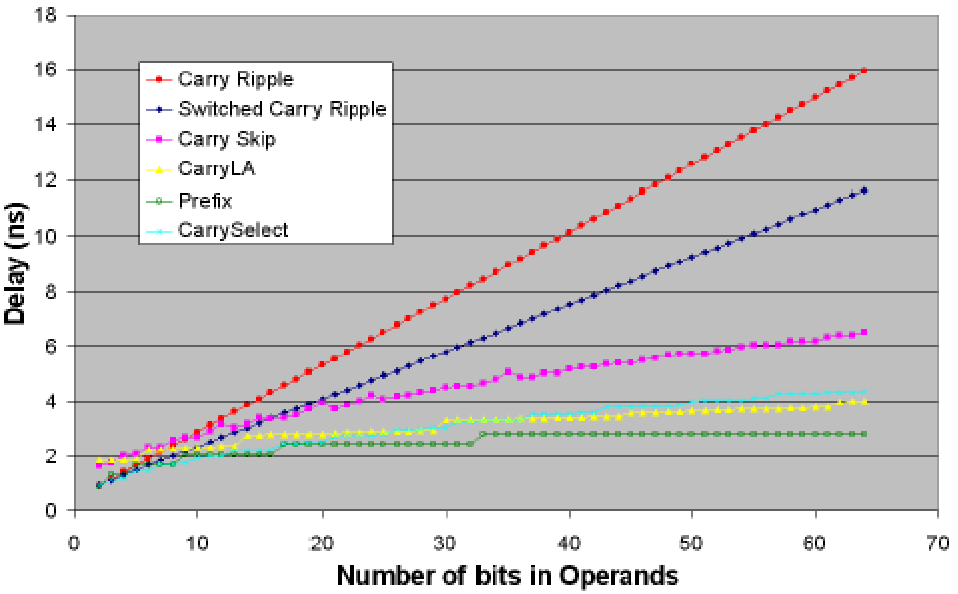
Primitive Component	Delay (ns)	Area	Power (W)	AT	AT^2	PD
4-bit carry ripple adder	72.1	160	0.8745784	11536	831745.6	63.058
8-bit carry ripple adder	72.1	160	0.8745784	11536	831745.6	63.058
16-bit carry ripple adder	72.1	160	0.8745784	11536	831745.6	63.058
4-bit carry look-ahead adder	93.54	288	1.049	26939.52	2519922	98.12346
8-bit carry look-ahead adder	118.9	302	1.1627	35907.8	4269437	138.25
16-bit carry look-ahead adder	124.3	310	1.1757	38533	4789651	146.14
two-level 8-bit carry look-ahead adder	31.57	434	1.348	13701.38	432552	42.56
4-bit carry select adder	24.72	422.5	1.6351	10444.2	258180	40.42
8-bit carry select adder	20.48	394.5	1.5757	8079.36	165465	32.27
16-bit carry select adder	26	356.5	1.4792	9269	240994	38.4592
Nonlinear Carry select adder	17.94	412	1.6267	7391.28	132599	29.183
4-bit Manchester adder	27.58	256	1.0857	7060.48	194728	29.9436
8-bit Manchester adder	27.58	256	1.0857	7060.48	194728	29.9436
16-bit Manchester adder	27.58	256	1.0857	7060.48	194728	29.9436
16-bit Ladner-Fischer prefix adder	24.79	326	1.23	8081.54	200341	30.4917
16-bit Brent-Kung prefix adder	26.94	290	1.15	7812.6	210471	30.981
16-bit Han-Carlson prefix adder	25.43	326	1.2758	8290.18	210819	32.4436
16-bit Kogge-Stone prefix adder	25.59	428	1.5546	10952.52	280274	39.78
64-bit Kogge-Stone adder	11.97	611	1.919	7313.67	87544	22.97

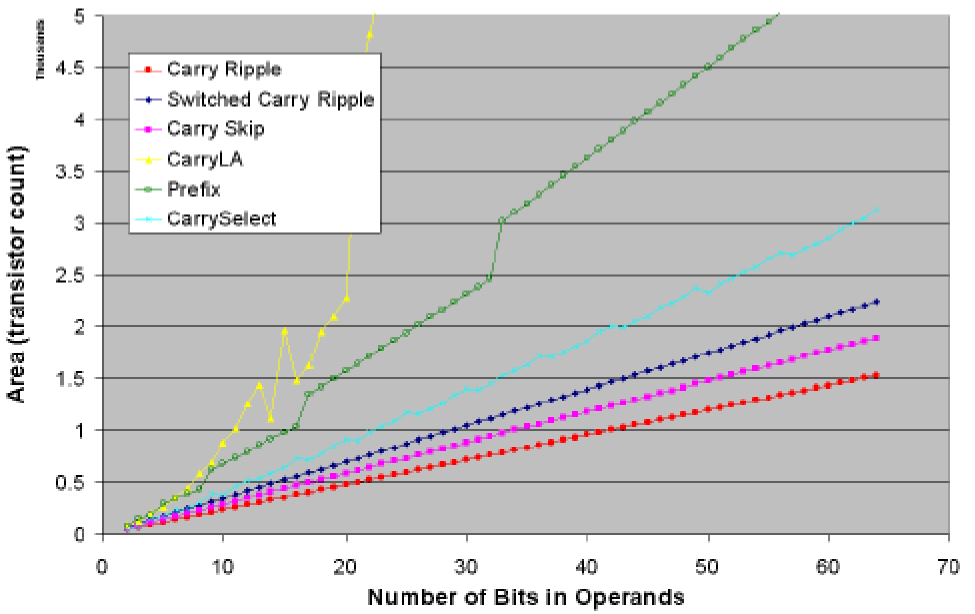
Compound Adder Design[2,13-16,20]

The Prefix Adder Scheme is chosen.

Advantages:
Simple and regular structure
Well-performance
A wide range of area-delay trade-offs

Moreover, the Flagged Prefix Adder is particular useful in compound adder implementation because, unlike other adder schemes which need a pair of adders to obtain sum and sum+1 simultaneously, it only use one adder.

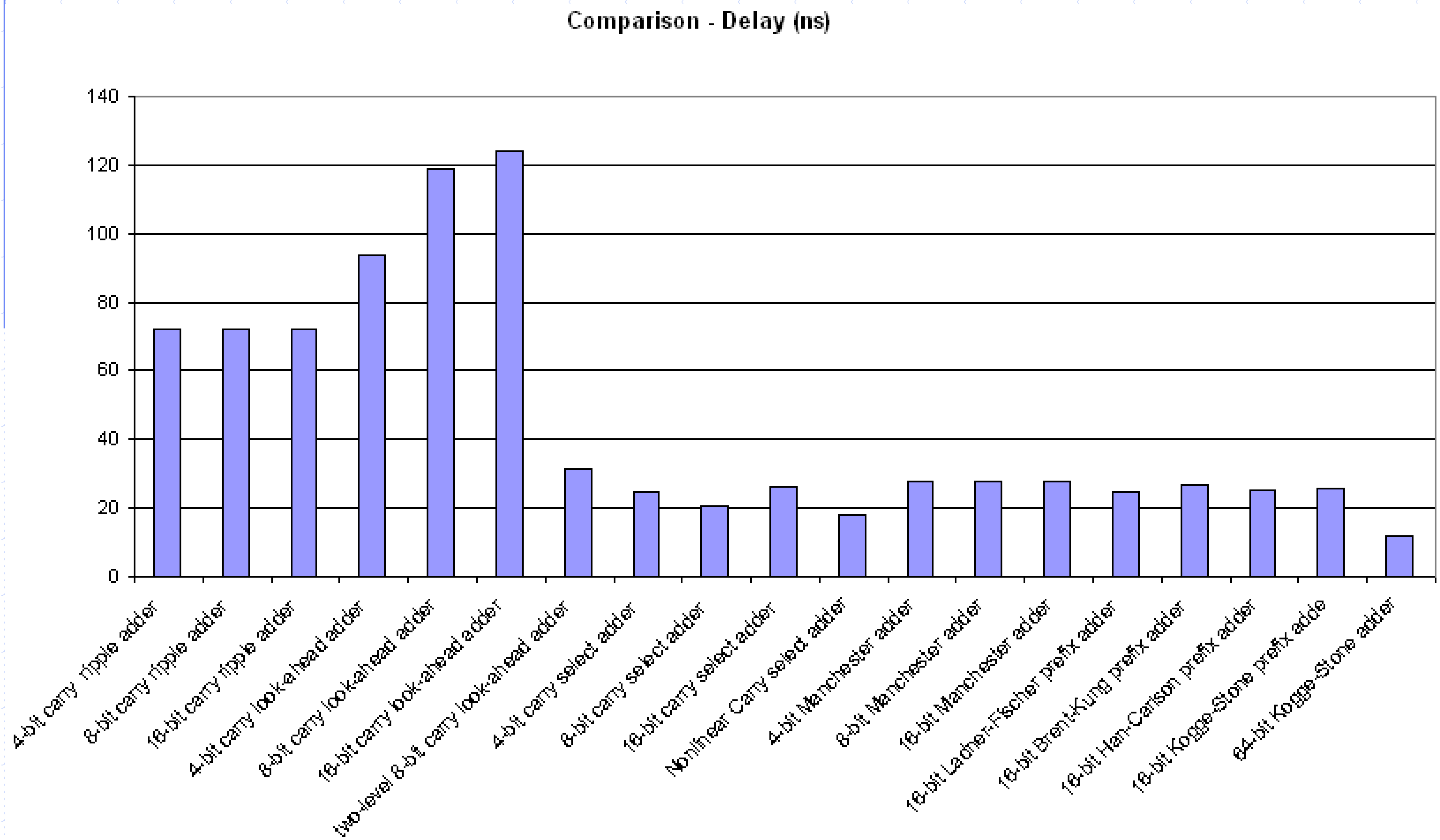


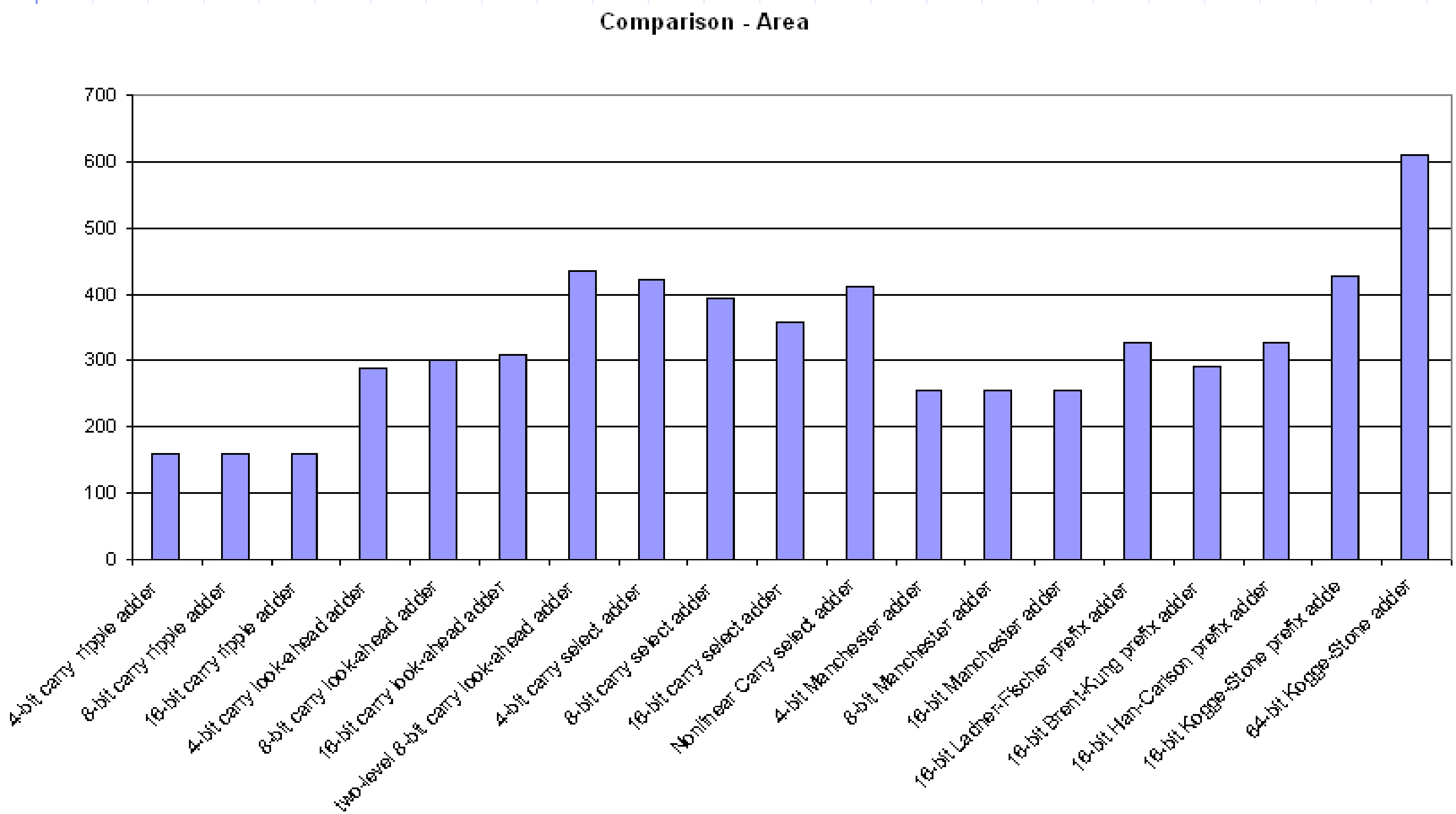


synthesis and targeting

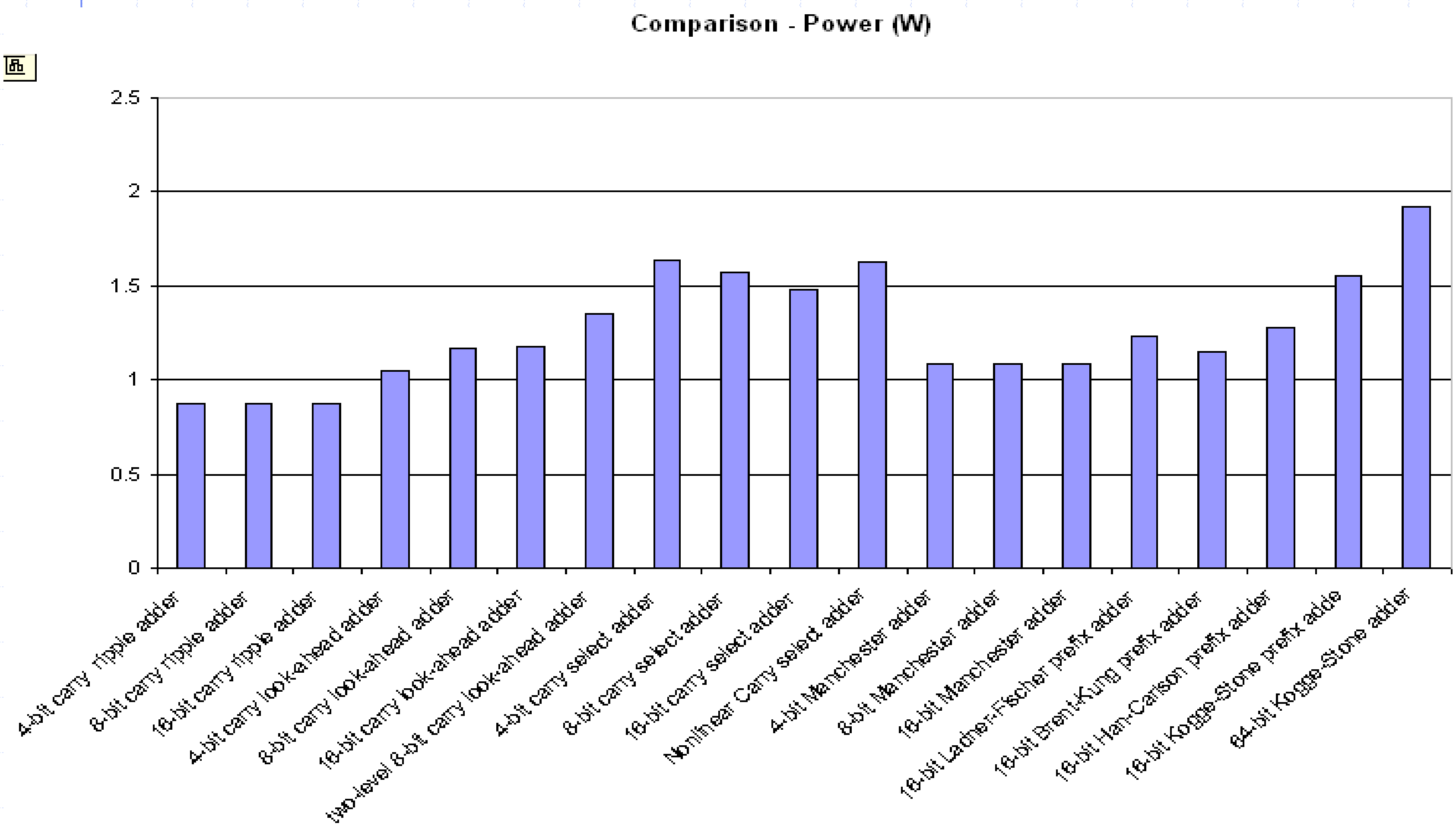
- Synopsys tools are used to perform logic synthesis.
- the implemented VHDL codes for all the 64-bit adders are translated into net list files.
- The virtex2 series library, XC2V250-4_avg, is used in those 64-bit adders synthesis and targeting because the area and the propagation delay is suitable for these adders.
- After synthesizing, the related power consumption, area, and propagation delay are reported.
- From the synthesis, the related FPGA layout schematic is reported.

64-bit adders comparison



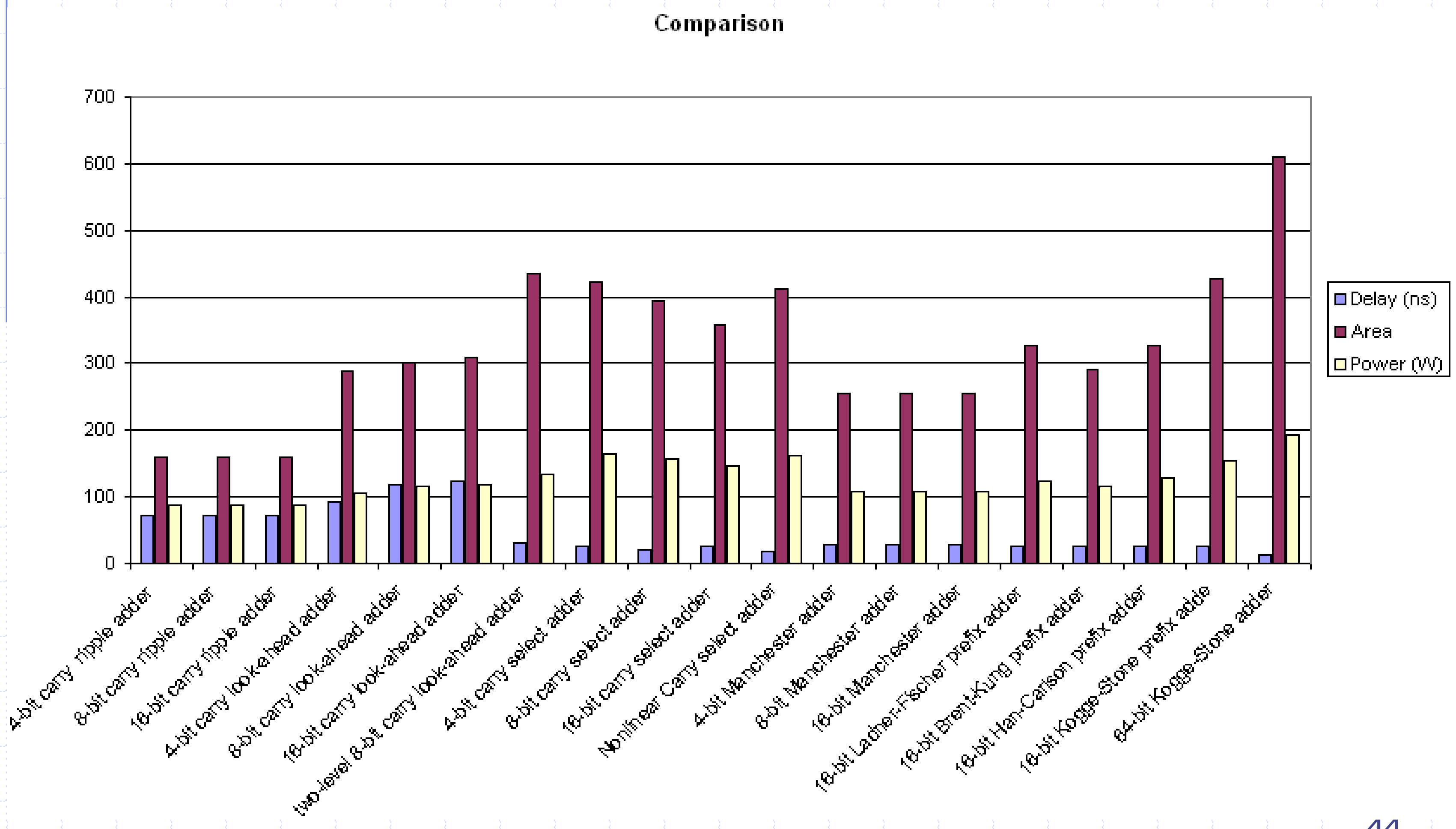


Comparison - Area
700
600
500
400
300
200
100
0
4-bit carry ripple adder
8-bit carry ripple adder
16-bit carry ripple adder
4-bit carry look-ahead adder
8-bit carry look-ahead adder
16-bit carry look-ahead adder
two-level 8-bit carry look-ahead adder
4-bit carry select adder
8-bit carry select adder
16-bit carry select adder
Nonlinear Carry select adder
4-bit Manchester adder
8-bit Manchester adder
16-bit Manchester adder
16-bit Ladner-Fischer prefix adder
16-bit Brent-Kung prefix adder
16-bit Han-Carlson prefix adder
16-bit Kogge-Stone prefix adde
64-bit Kogge-Stone adder



Comparison - Power (W)
2.5
2
1.5
1
0.5
0
4-bit carry ripple adder
8-bit carry ripple adder
16-bit carry ripple adder
4-bit carry look-ahead adder
8-bit carry look-ahead adder
16-bit carry look-ahead adder
two-level 8-bit carry look-ahead adder
4-bit carry select adder
8-bit carry select adder
16-bit carry select adder
Nonlinear Carry select adder
4-bit Manchester adder
8-bit Manchester adder
16-bit Manchester adder
16-bit Ladner-Fischer prefix adder
16-bit Brent-Kung prefix adder
16-bit Han-Carlson prefix adder
16-bit Kogge-Stone prefix adde
64-bit Kogge-Stone adder

The power is not in scale(*100).

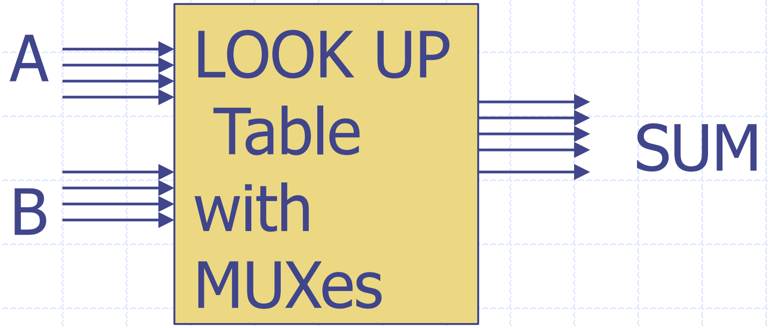


64-bit adders conclusion

- Adders can be implemented in different methods according to the different requirements.
- Each kind of adder has different properties in area, propagation delay, and power consumption.
- There is no absolute advantages or disadvantages for an adder, and usually, one advantage compensates with another disadvantage.
- A ripple carry adder is easy to implemented, and for short bit length, the performances are good.
- For long bit length, a carry look-ahead adder is not practical, but a hierarchical structure one can improve much.

- A carry select adder has good performance in propagation delay especially the nonlinear one; however, it compensates with large area.
- In these 64-bit adders, the Manchester carry adder has the best performance when considered all of the propagation delay, area, and power consumption.
- The parallel prefix adder has good performance in propagation delay, but the area becomes large.
- The 64-bit Kogge-Stone prefix adder has the shortest propagation delay, but it has the largest area and power consumption as well.

Adders Using Tables (FPGAs)

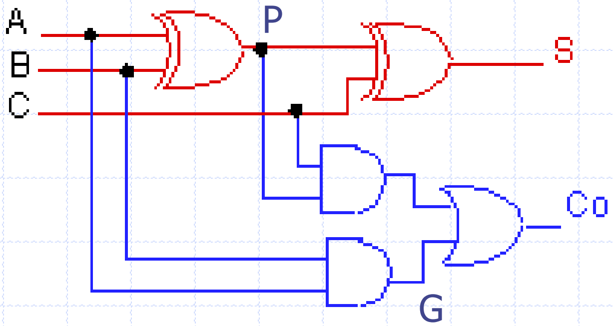


Ripple Carry’s VHDL

```
library IEEE;
use ieee.std_logic_1164.all;

entity ripple_carry is
        port( A, B    : in std_logic_vector( 15 downto 0);
              C_in    : in std_logic;
              S       : out std_logic_vector( 15 downto 0);
              C_out   : out std_logic);
end ripple_carry;

architecture RTL of ripple_carry is

begin

process(A, B, C_in)

        variable tempC      : std_logic_vector( 16 downto 0 );
        variable P         : std_logic_vector( 15 downto 0 );
        variable G         : std_logic_vector( 15 downto 0 );

        begin
```

Ripple Carry's VHDL

```
tempC(0) := C_in;

        for i in 0 to 15 loop
                P(i):=A(i) xor B(i);
                G(i):=A(i) and B(i);

                S(i)<= P(i) xor tempC(i);
                tempC(i+1):=G(i) or (tempC(i) and P(i));
        end loop;

        C_out <= tempC(16);

end process;


end;
```



Carry Select's VHDL (ripple4)

- Two four-bit ripple carry adders were used to build a carry select section of the same size
- Four 4-bit carry select sections were used as components in building our 16 bit adders

ripple_carry4

```
library IEEE;
use ieee.std_logic_1164.all;

entity ripple_carry4 is
    port( e, f      : in std_logic_vector( 3 downto 0);
          carry_in      : in std_logic;
          S         : out std_logic_vector( 3 downto 0);
          carry_out     : out std_logic);
end ripple_carry4;
```

Carry Select's VHDL (ripple4)

```
architecture RTL of ripple_carry4 is

begin

process(e, f, carry_in)

    variable tempC      : std_logic_vector( 4 downto 0 );
    variable P         : std_logic_vector( 3 downto 0 );
    variable G         : std_logic_vector( 3 downto 0 );

    begin

    tempC(0) := carry_in;

    for i in 0 to 3 loop
        P(i):=e(i) xor f(i);
        G(i):=e(i) and f(i);

        S(i)<= P(i) xor tempC(i);
        tempC(i+1):=G(i) or (tempC(i) and P(i));
    end loop;
    carry_out <= tempC(4);

end process;
end;
```

Carry Select's VHDL (select4)

carry_select4

```
library IEEE;
use ieee.std_logic_1164.all;

entity carry_select4 is
    port( c, d      : in std_logic_vector( 3 downto 0);
          C_input     : in std_logic;
          Result        : out std_logic_vector( 3 downto 0);
          C_output    : out std_logic);
end carry_select4;

architecture RTL of carry_select4 is

component ripple_carry4

port(         e, f     : in std_logic_vector( 3 downto 0);
          carry_in     : in std_logic;
          S        : out std_logic_vector( 3 downto 0);
          carry_out    : out std_logic);

end component;
```

Carry Select's VHDL (select4)

```
For S0: ripple_carry4 Use entity work.ripple_carry4(RTL);
For S1: ripple_carry4 Use entity work.ripple_carry4(RTL);

signal SUM0, SUM1      : std_logic_vector( 3 downto 0 );
signal carry0, carry1 : std_logic;
signal zero, one      : std_logic;

begin

zero<='0';
one<='1';

S0: ripple_carry4 port map( e=>c, f=>d, carry_in=>zero, S=>SUM0,
carry_out=>carry0 );
S1: ripple_carry4 port map( e=>c, f=>d, carry_in=>one, S=>SUM1,
carry_out=>carry1 );

Result<=SUM0 when C_input='0' else
     SUM1 when C_input='1' else
     "ZZZZ";

C_output<= (C_input and carry1) or carry0;

end;
```

Carry Select's VHDL (select16)

carry_select16

```
library IEEE;
use ieee.std_logic_1164.all;

entity carry_select16 is
        port( A, B    : in std_logic_vector( 15 downto 0);
              C_in    : in std_logic;
              SUM     : out std_logic_vector( 15 downto 0);
              C_out   : out std_logic);
end carry_select16;

architecture RTL of carry_select16 is

component carry_select4

port( c, d    : in std_logic_vector( 3 downto 0);
      C_input    : in std_logic;
      Result       : out std_logic_vector( 3 downto 0);
      C_output    : out std_logic);

end component;
```

Carry Select's VHDL (select16)

```
For S0: carry_select4 Use entity work.carry_select4(RTL);
For S1: carry_select4 Use entity work.carry_select4(RTL);
For S2: carry_select4 Use entity work.carry_select4(RTL);
For S3: carry_select4 Use entity work.carry_select4(RTL);

signal tempc1, tempc2, tempc3 : std_logic;

begin

S0: carry_select4 port map( c=>A ( 3 downto 0 ), d =>B ( 3 downto 0 ),
C_input=>C_in, Result=>SUM ( 3 downto 0 ), C_output=>tempc1 );
S1: carry_select4 port map( c=>A ( 7 downto 4 ), d =>B ( 7 downto 4 ),
C_input=>tempc1, Result=>SUM ( 7 downto 4 ), C_output=>tempc2 );
S2: carry_select4 port map( c=>A ( 11 downto 8 ), d =>B ( 11 downto 8 ),
C_input=>tempc2, Result=>SUM ( 11 downto 8 ), C_output=>tempc3 );
S3: carry_select4 port map( c=>A ( 15 downto 12 ), d =>B ( 15 downto 12
), C_input=>tempc3, Result=>SUM ( 15 downto 12 ), C_output=>C_out );

end;
```

Carry Look-Ahead's VHDL

half_adder

```
library IEEE;
use ieee.std_logic_1164.all;

entity half_adder is
        port( A, B : in std_logic_vector( 16 downto 1 );
                   P, G : out std_logic_vector( 16 downto 1 ) );
end half_adder;

architecture RTL of half_adder is

begin

P <= A xor B;
G <= A and B;

end;
```

Carry Look-Ahead's VHDL

carry_generator

```
library IEEE;
use ieee.std_logic_1164.all;

entity carry_generator is
        port(    P , G : in std_logic_vector(16 downto 1);
                 C1    : in std_logic;
                 C     : out std_logic_vector(17 downto 1));
end carry_generator;

architecture RTL of carry_generator is
begin
        process(P, G, C1)
        variable tempC   : std_logic_vector(17 downto 1);

        begin
                tempC(1) := C1;
                for i in 1 to 16 loop
                        tempC(i+1) := G(i) or (P(i) and tempC(i));
                end loop;
        C <= tempC;
        end process;
end;
```

Carry Look-Ahead's VHDL

Look_Ahead_Adder

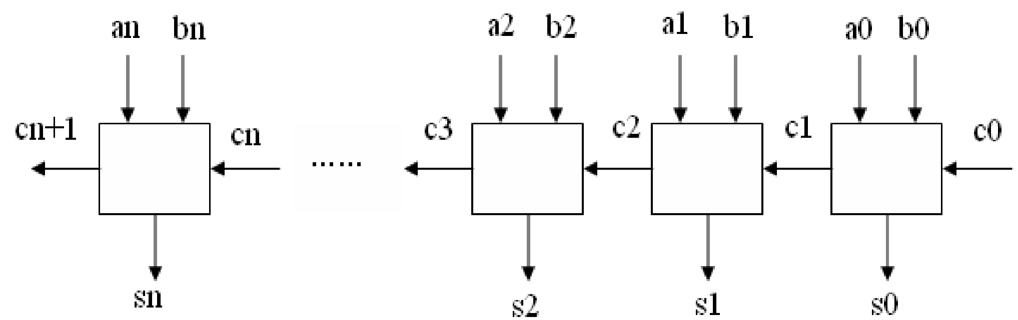
```
library IEEE;
use ieee.std_logic_1164.all;

entity Look_Ahead_Adder is

port( A, B : in std_logic_vector( 16 downto 1 );
  carry_in : in std_logic;
 carry_out : out std_logic;
         S :  out std_logic_vector( 16 downto 1 ) );

end Look_Ahead_Adder;

architecture RTL of Look_Ahead_Adder is

component carry_generator

 port(   P , G : in std_logic_vector(16 downto 1);
                C1    : in std_logic;
                C     : out std_logic_vector(17 downto 1));
end component;
```

Carry Look-Ahead's VHDL

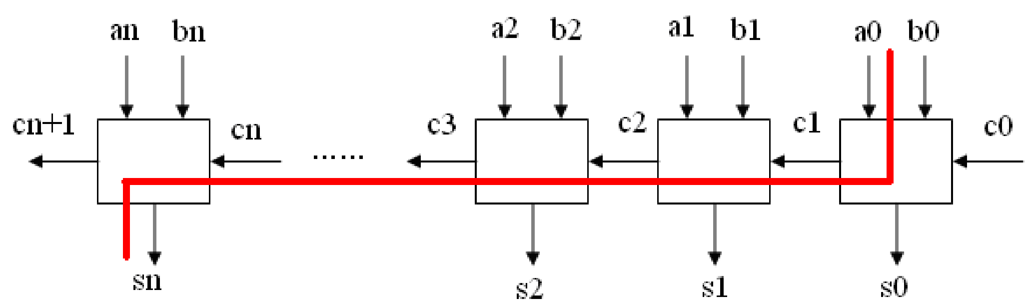
```
component half_adder

 port( A, B : in std_logic_vector( 16 downto 1 );
             P, G : out std_logic_vector( 16 downto 1) );

end component;

For CG: carry_generator Use entity work.carry_generator(RTL);
For HA: half_adder Use entity work.half_adder(RTL);

signal tempG, tempP : std_logic_vector( 16 downto 1 );
signal tempC : std_logic_vector( 17 downto 1 );

begin

HA: half_adder port map( A=>A, B=>B, P =>tempP, G=>tempG );
CG: carry_generator port map( P=>tempP, G=>tempG, C1=>carry_in, C=>tempC );
S <= tempC( 16 downto 1 ) xor tempP;
carry_out <= tempC(17);


end;
```

Ripple carry adder

Block diagram:



Critical path:



Carry look-ahead adder

$P_i = A_i \oplus B_i$ Carry propagate

$G_i = A_i \cdot B_i$ Carry generate

$S_i = P_i \oplus C_i$ Summation

$C_{i+1} = G_i + P_iC_i$ Carryout

C0 = Cin;

C1 = G (0) + (P(0)C0);

C2 = G (1) + (P (1)G (0)) + (P(1) P(0)C0);

C3 = G (2) + (P(2) G(1)) + (P(2)P(1)G(0)) + (P(2)P(1)P(0) C0);

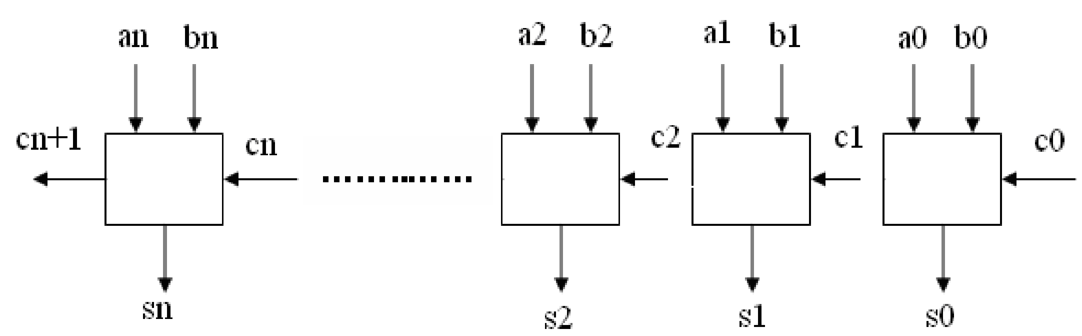
C4 = G(3) + (P(3) G(2)) + (P(3) P(2) G(1)) + (P(3) P(2) P(1) G(0)) + (P(3)P(2) P(1) P(0)C0);

....

$C_{i+1} = G_i + P_iG_{i-1} + P_iP_{i-1}G_{i-2} + \ldots P_iP_{i-1} \ldots P_2P_1G_0 + P_iP_{i-1} \ldots P_1P_0C_0$.

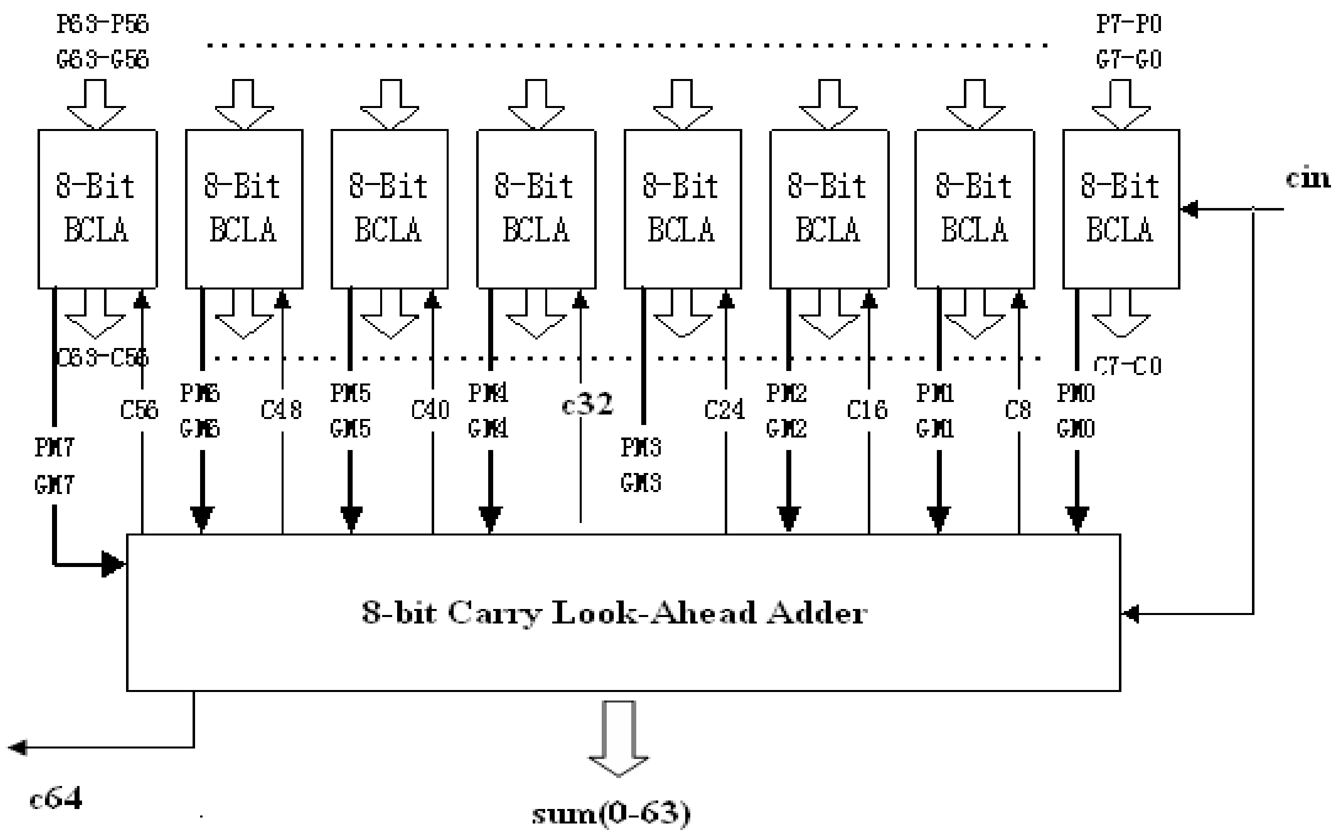
Carry look-ahead adder

Block diagram



- When n increases, it is not practical to use standard carry look-ahead adder since the fan-out of carry calculation becomes very large.
- A hierarchical carry look-ahead adder structure could be implemented.

Hierarchical 2- level 8-bit carry look-ahead adder

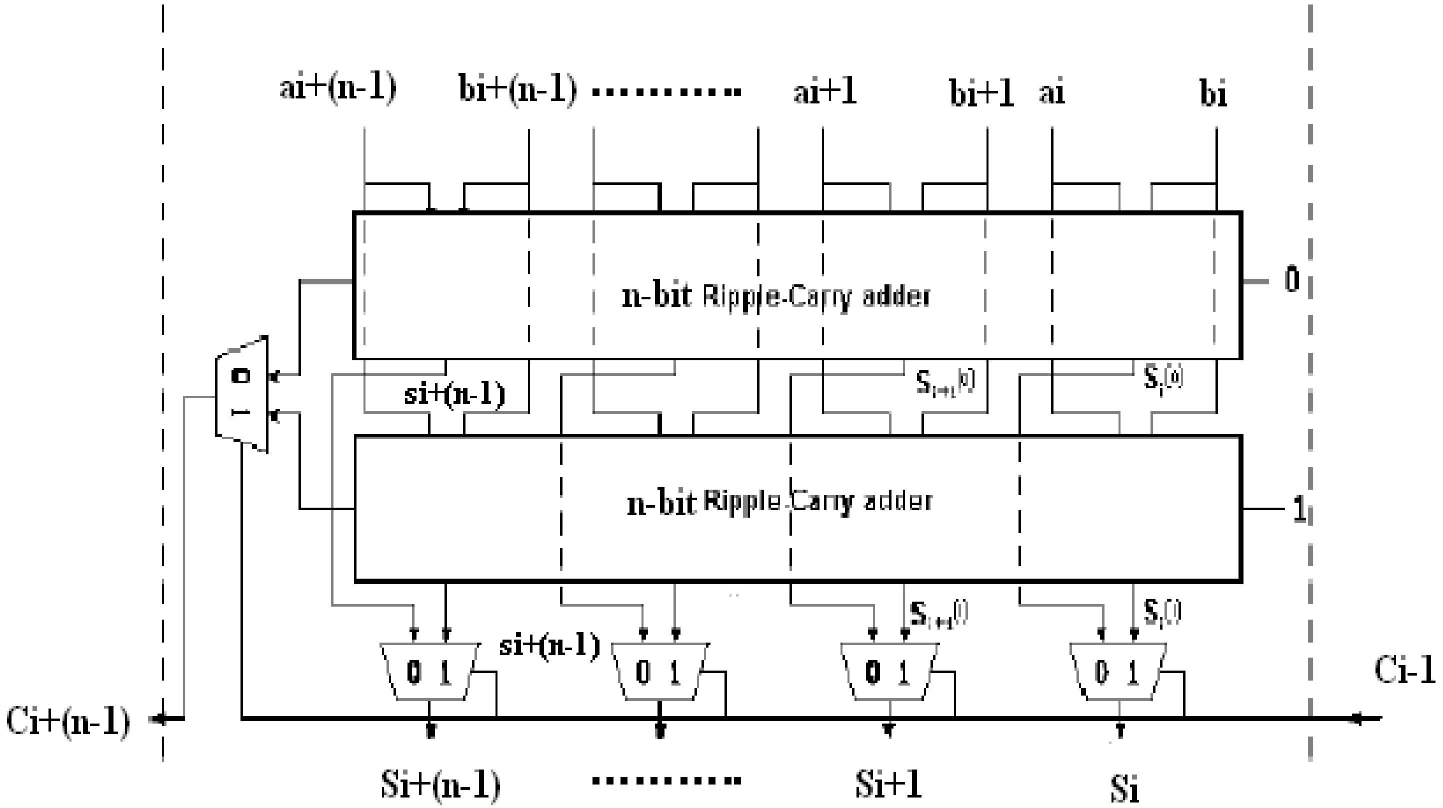


Carry select adder

- compute alternative results in parallel and subsequently select the carry input which is calculated from the previous stage.
- compensate with an extra circuit to calculate the alternative carry input and summation result.
- need multiplexer to select the carry input for the next stage and the summation result.
- the drawback is that the area increases.
- time delay=time to compute the first section + time to select sum from subsequent section.
- The summation part could be implemented by ripple carry adder, Manchester adder, carry look-ahead adder as well as prefix adder……

Carry select adder

block diagram

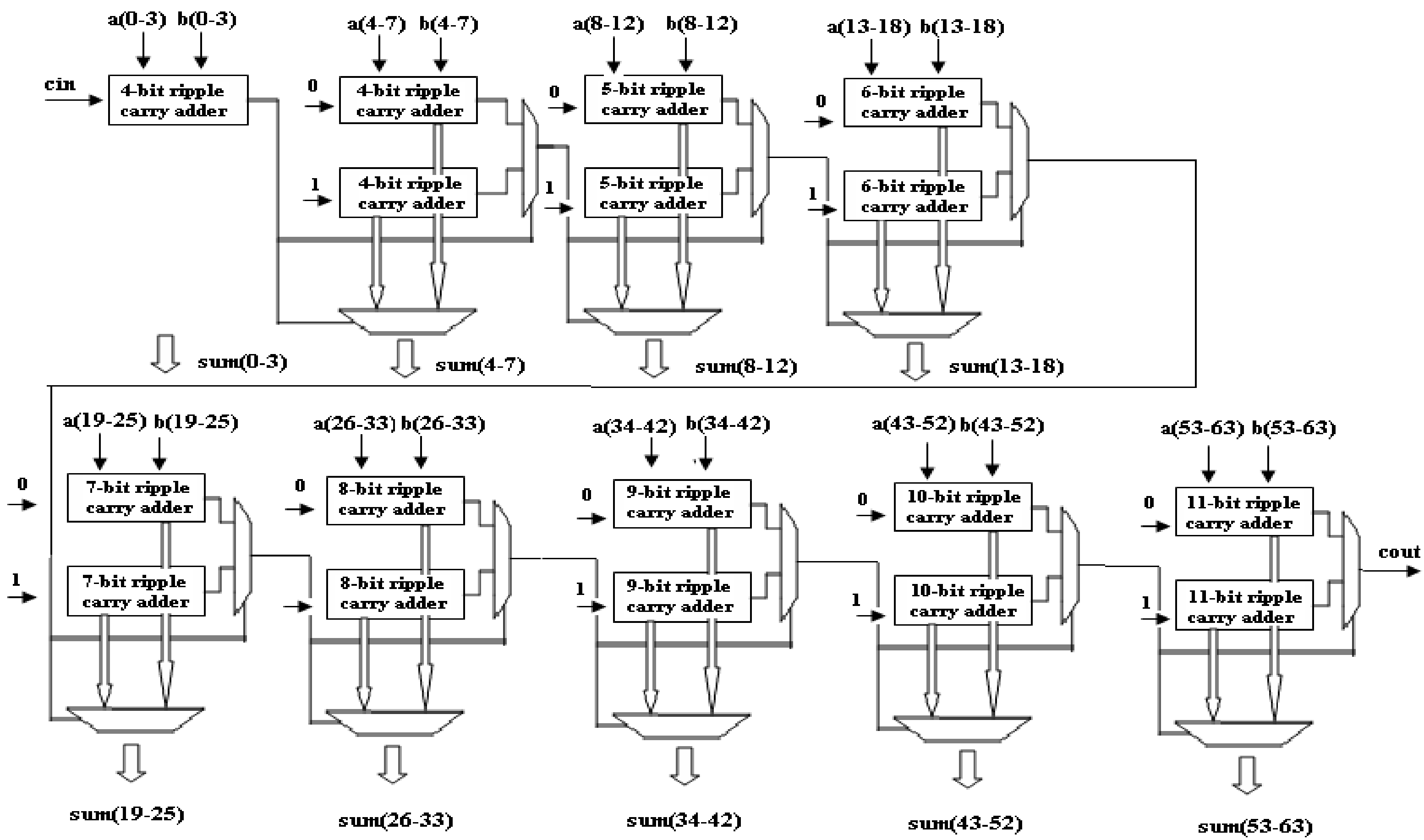


Carry select adder

- For an n bit adder, it could be implemented with equal length of carry select adder, and this is called linear carry select adder.
- However. the linear carry select adder does not always have the best performance.
- A carry select adder can be implemented in different length, and this is called nonlinear carry select adder.
- A 64-bit adder can be implemented in 4, 4, 5, 6, 7, 8, 9, 10,11 bit nonlinear structure.
- The performance of 64-bit nonlinear carry select adder is better than linear one in propagation delay.

64-bit nonlinear carry select adder

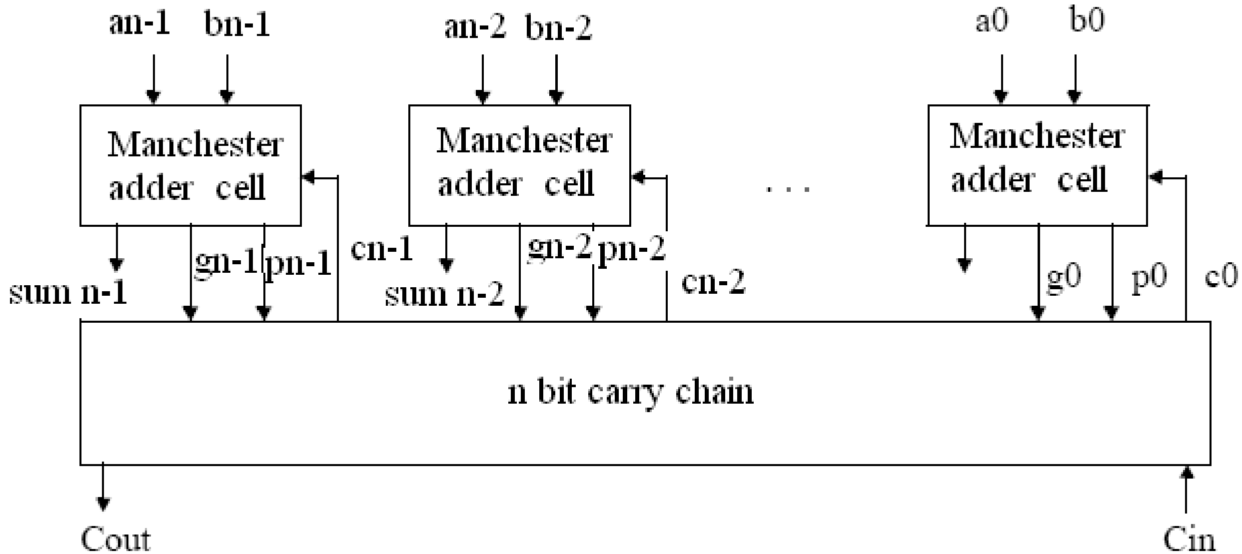
Block diagram



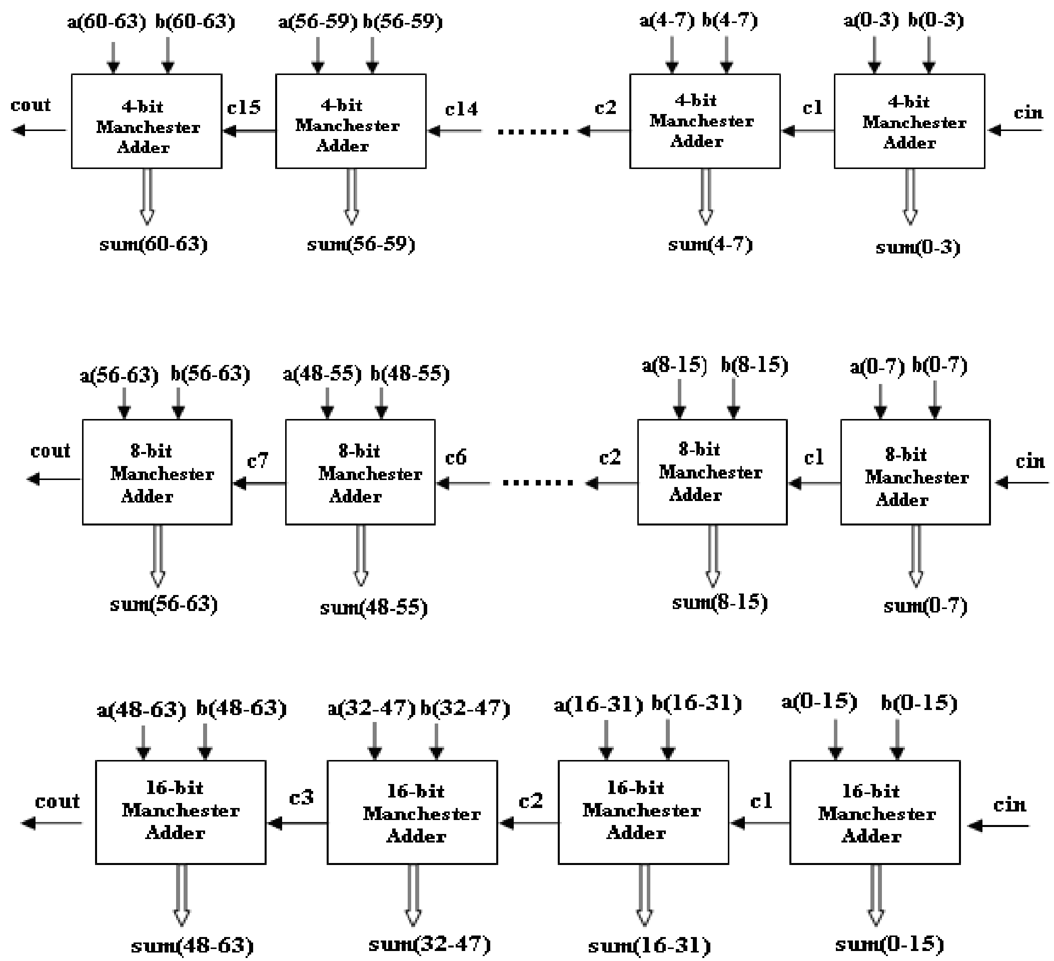
Manchester carry adder

- A Manchester adder could be constructed in dynamic stage, static stage, and multiplexer stage structure.
- A Manchester adder, based on multiplexer, is called a conflict free Manchester Adder.

Block diagram:



64-bit adders implemented in Manchester carry adder



Parallel prefix adder

- like a carry look-ahead adder, the prefix adder accelerates addition by the parallel prefix carry tree.
- the production of the carries in the prefix adder can be designed in many different ways based on the different requirements.
- the main disadvantage of prefix adder is the large fan-out of some cells as well as the long interconnection wires.
- the large fan-out can be eliminated by increasing the number of levels or cells; as a result, there are different structure.
- the long inter-connections produce an increase in delay which can be reduced by including buffers.

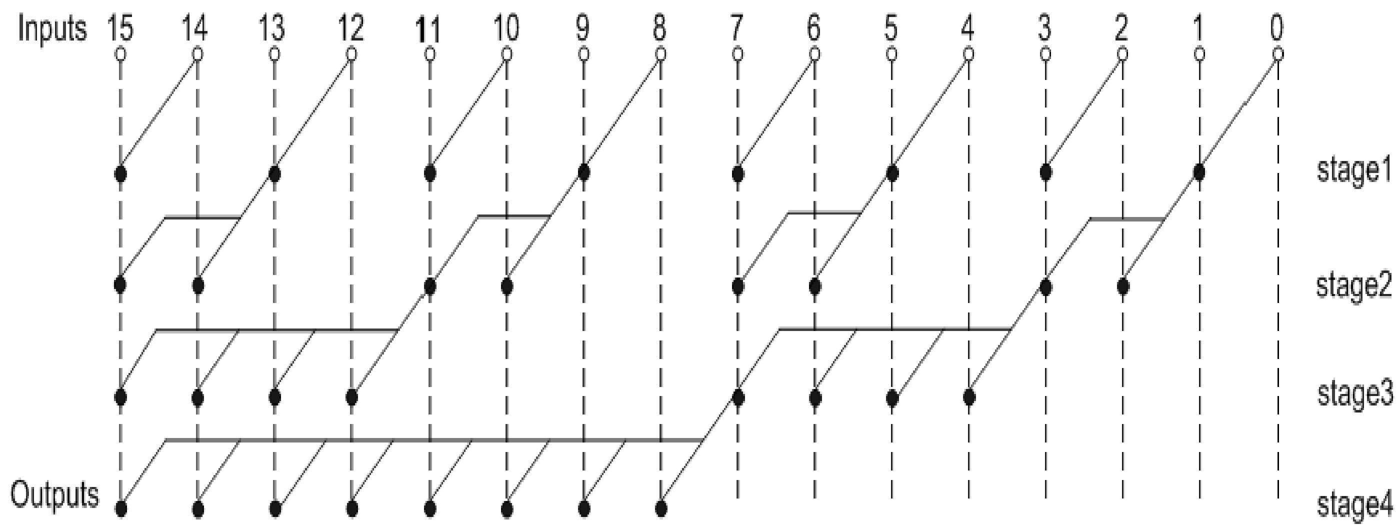
Ladner-Fischer parallel prefix adder

Carry stages: $\log 2^n$

The number of cells: (n/2) * $\log 2^n$

Maximum fan-out: n/2.

Block diagram(16 bits):



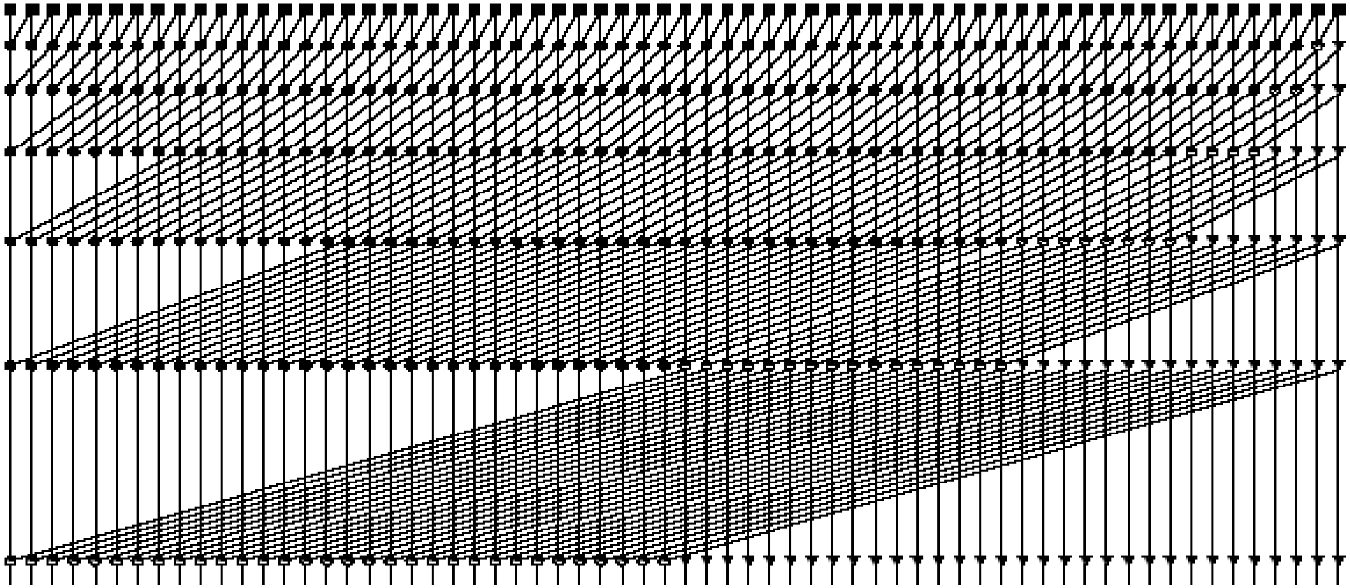
Kogge-Stone parallel prefix adder

Carry stages: $\log 2^n$

The number of cells: n ($\log 2^n$ -1) +1.

Maximum fan-out: 2

Block diagram(64 bits):



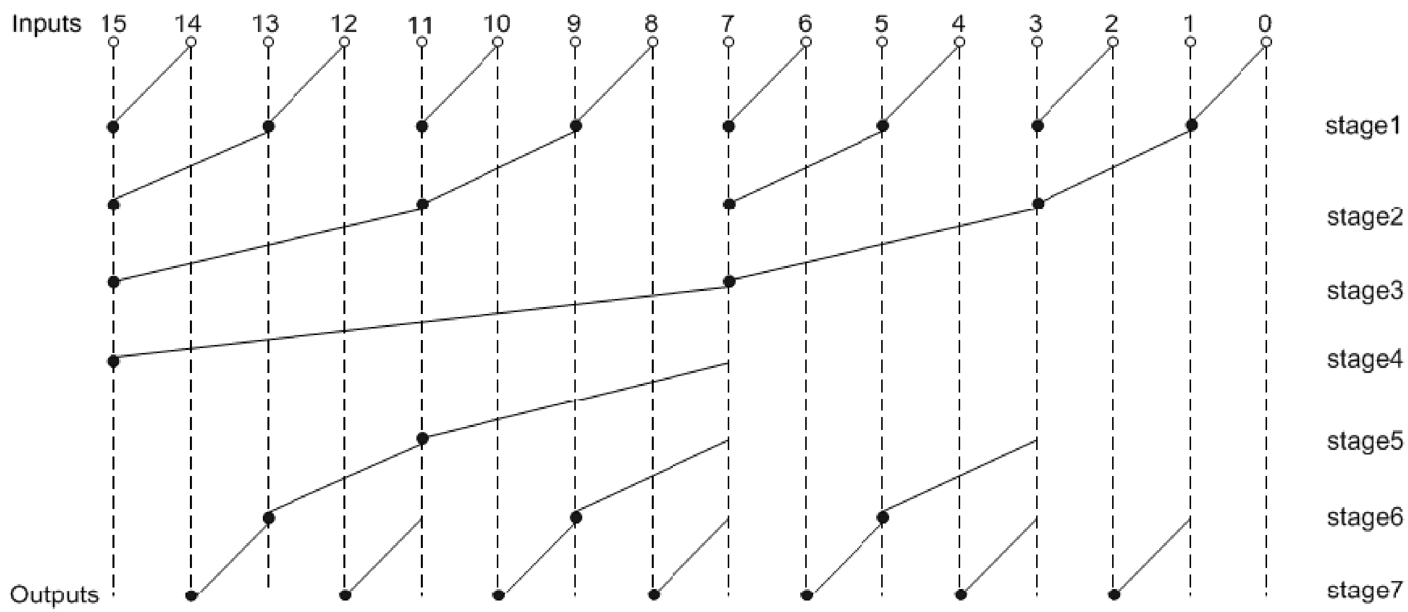
Brent-kung parallel prefix adder

Carry stages: $2\log 2^n$-1;

The number of cells: 2(n-1) - $\log 2^n$;

Maximum fan-out: 2

Block diagram(16 bits):

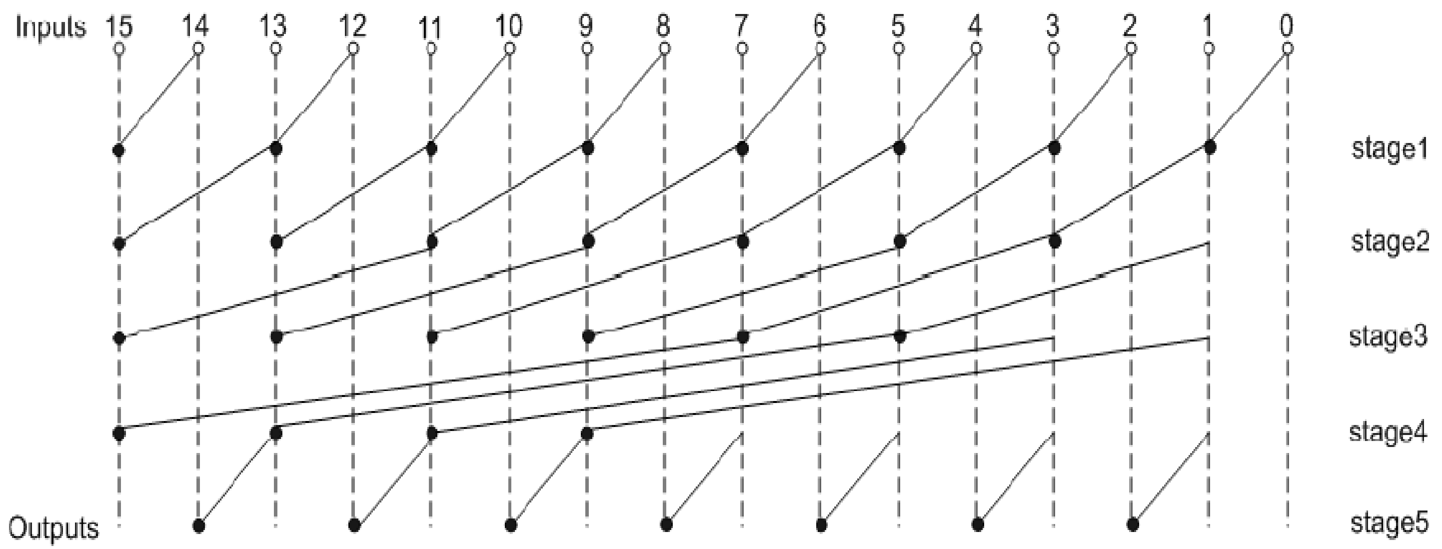


Han-Carlson parallel prefix adder

It is a hybrid structure combining from the Brent-Kung and Kogge-Stone prefix adder.

Carry stages: $\log 2^n + 1$.

Maximum fan-out: 2.

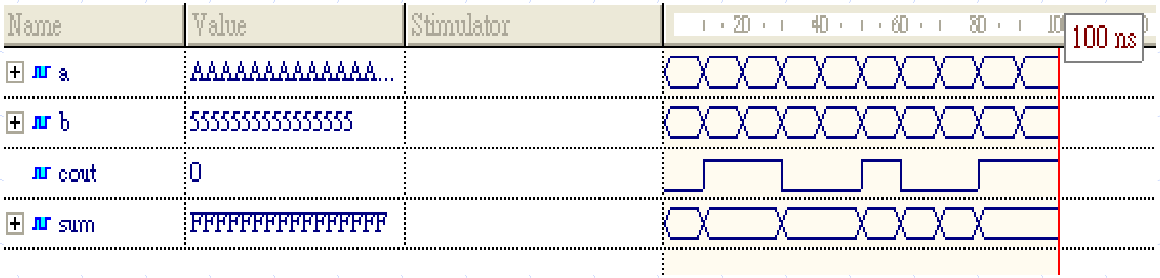


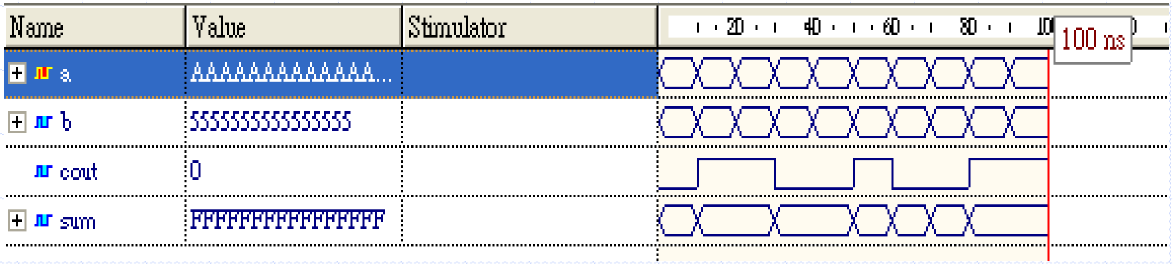
64-bit adders implementations and simulations

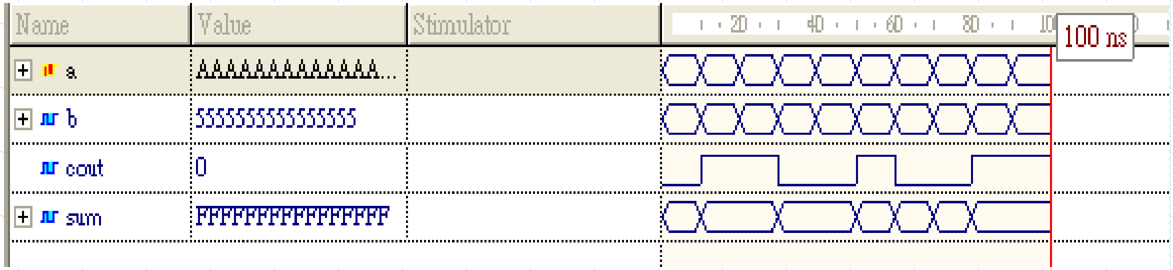
- 18 kinds of adders are implemented, including ripple carry adders, carry look-ahead adders, carry select adders, Manchester carry adders, and parallel prefix adders.
- Each 64 bits adder might be consisted of 4 bits, 8 bits, and 16 bits adder component as well as different prefix adder component.
- Hierarchical carry look-ahead adder and nonlinear carry select adder are also implemented.
- A test bench is written to test the simulation result.
- In the test bench, each bit of the 64-bit adder should be verified in carry propagation and summation.

Test bench simulation result

carry ripple adder, carry look-head adder, hierarchical carry look-ahead adder.

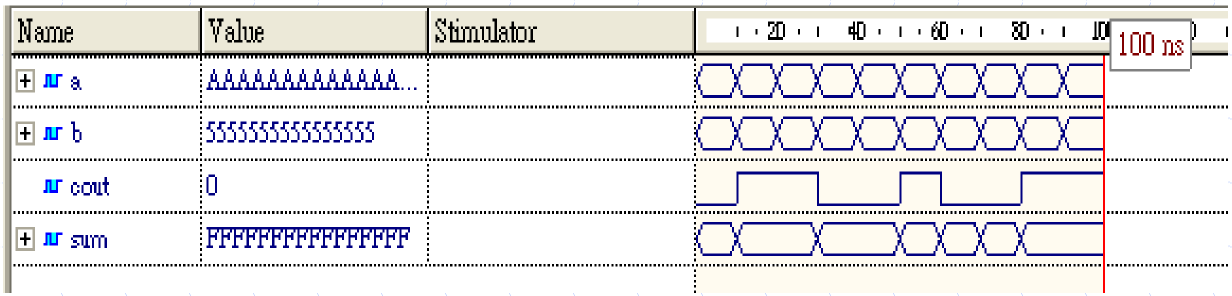


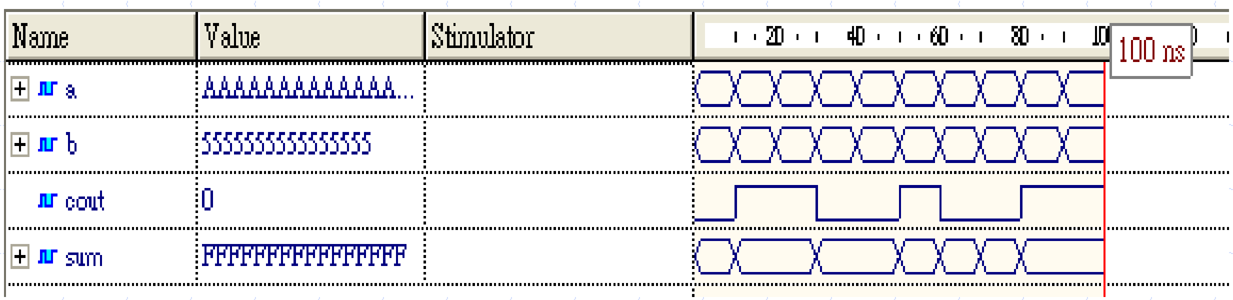


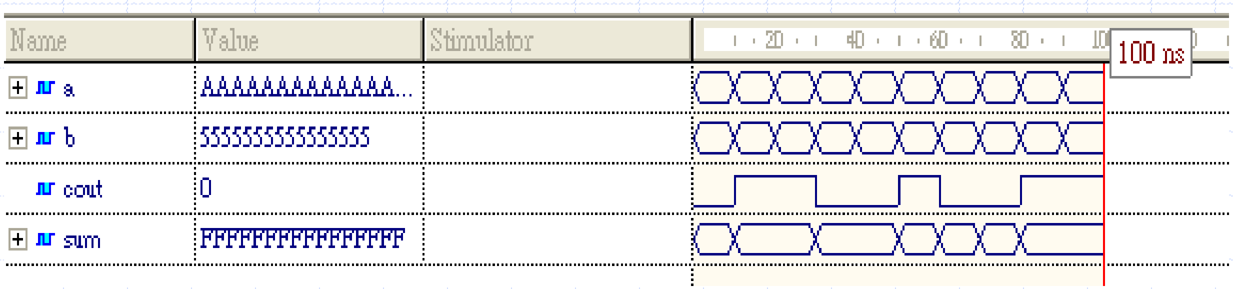


Test bench simulation result- continued

carry select adder, nonlinear carry select adder, Manchester carry adder.







Test bench simulation result- continued

Ladner-Fischer, Brent-Kung , Han-Carlson . Kogge-Stone prefix adders