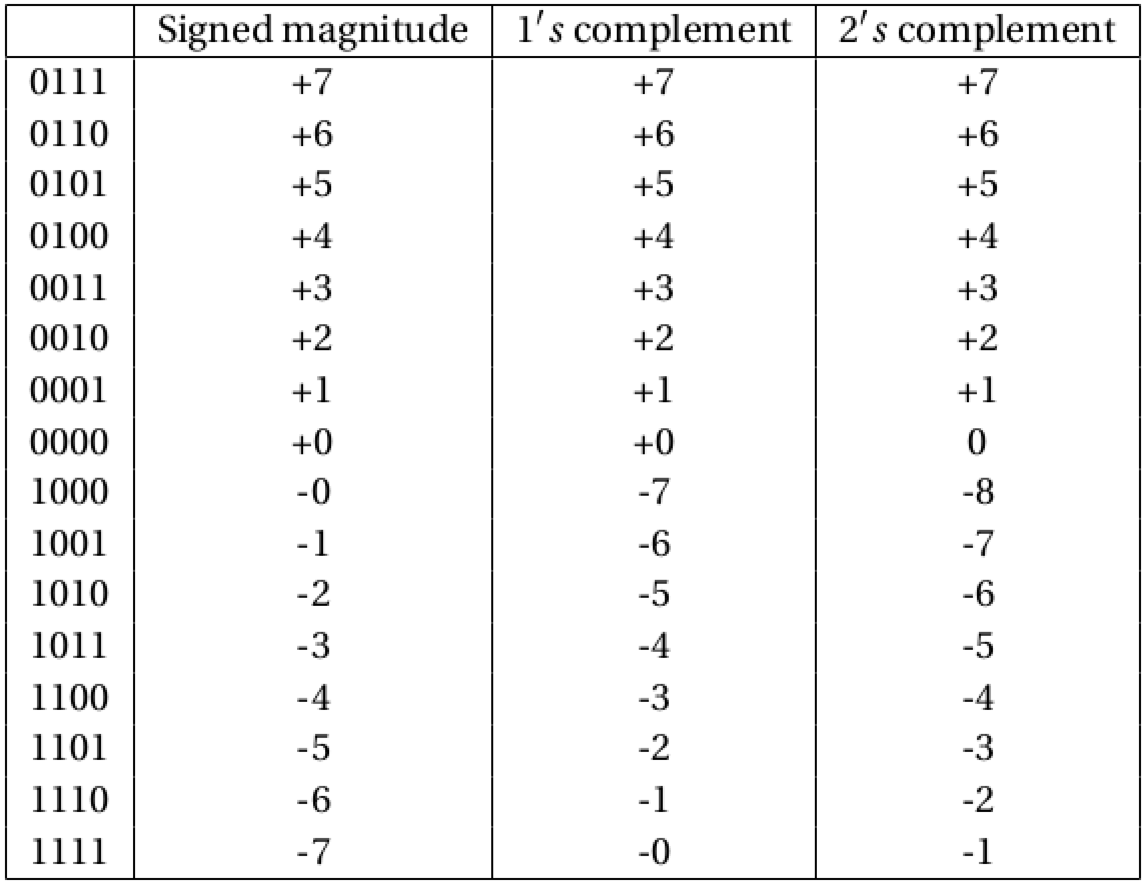
Fixed Point Adder and Multiplier Circuits

Representation of Fixed Point Numbers

The range of representation for n-bit binary number using 2's complement is from $-(2^{n-1})$ to $+(2^{n-1} - 1)$. In case of signed magnitude and 1's complement of *n*-bit binary number, the range is from $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$.



	Signed magnitude	1′s complement	2′s complement
0111	+7	+7	+7
0110	+6	+6	+6
0101	+5	+5	+5
0100	+4	+4	+4
0011	+3	+3	+3
0010	+2	+2	+2
0001	+1	+1	+1
0000	+0	+0	0
1000	-0	-7	-8
1001	-1	-6	-7
1010	-2	-5	-6
1011	-3	-4	-5
1100	-4	-3	-4
1101	-5	-2	-3
1110	-6	-1	-2
1111	-7	-0	-1

Table . 4-bit fixed point number representation

High Performance Circuit Design

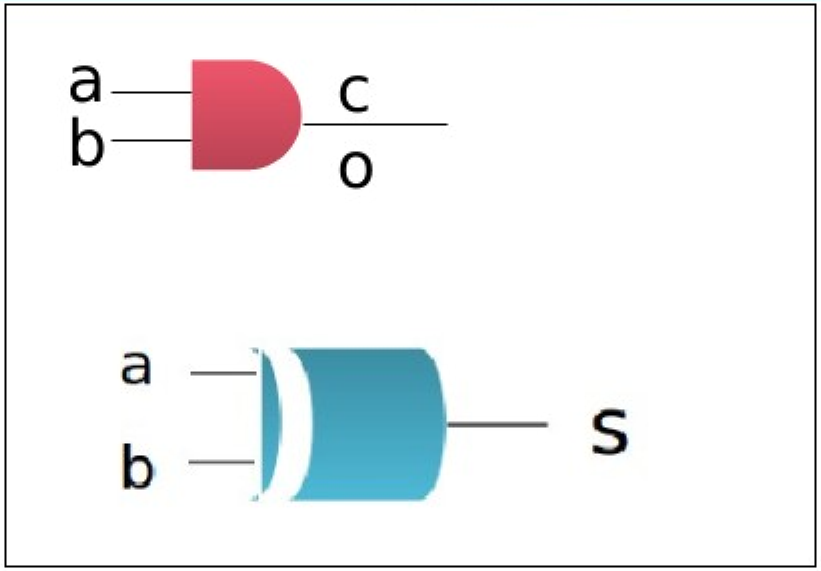
- High speed Adder Circuits
  - Carry Ripple – Inherently Sequential
  - Carry Look ahead – Parallel version

- High Speed Multiplier Circuits
  - Carry save array multiplier
  - Wallace tree multiplier
  - Divide & Conquer based multiplication
  - Booth Encoding

High Performance Circuit Design

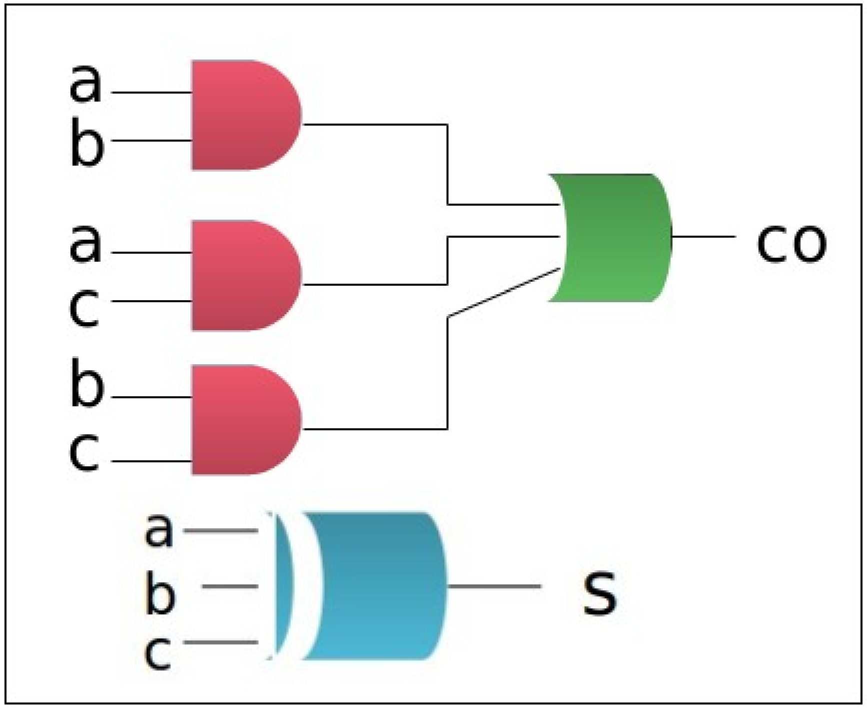
- Performance of a circuit
  - Circuit depth – maximum level in the topological sort.
  - Circuit Size – Number of combinational elements.
- Optimize both for high performance.
- Both are inversely proportional – so a balance to be arrived.

Half Adder



a	b	s	co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

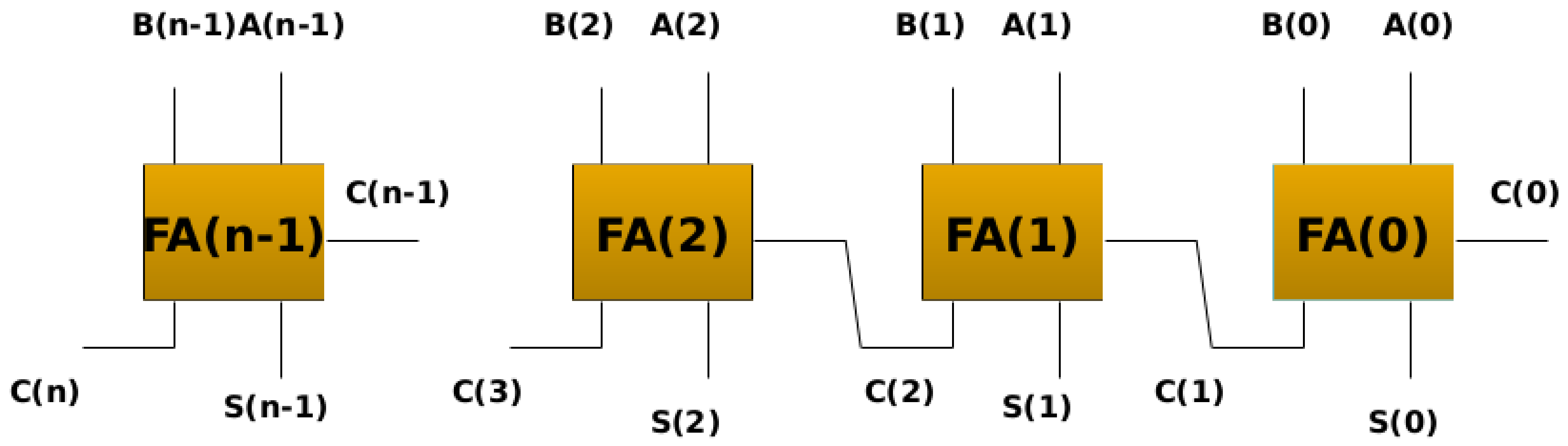
Full Adder



a	b	c	s	co
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

Ripple Carry Adder

- Circuit Depth is ‘*n*’.
- Circuit area is ‘*n*’ times size of a Full Adder
- Given two *n*-bit numbers *A* and *B*.
- Time complexity *O(n)* and circuit complexity *O(n)*



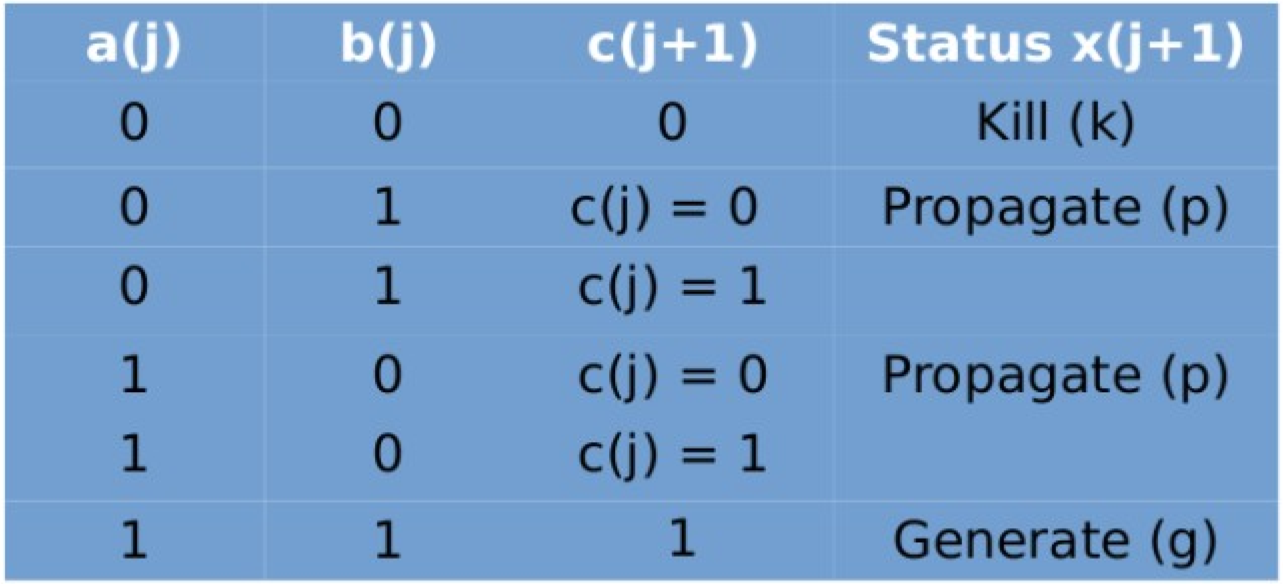
Carry Lookahead Adder

- The depth is ‘n’ because of the carry.
- Some interesting facts about carry

a(j)	b(j)	c(j)	c(j+1)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

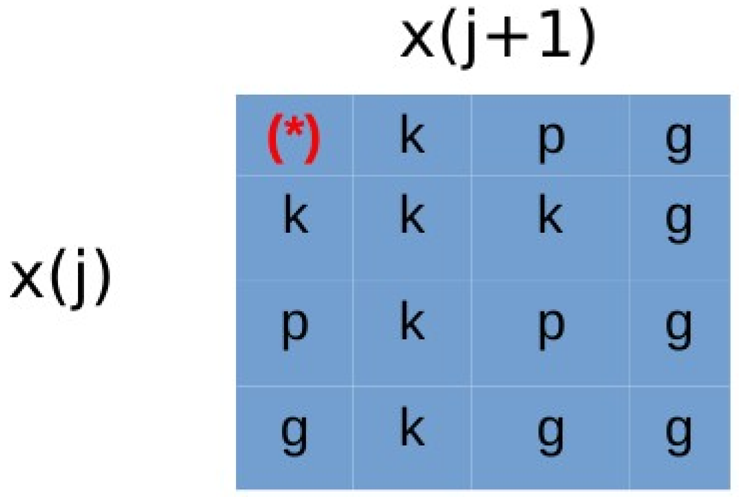
If a(j) = b(j) then
    c(j+1) = a(j) = b(j)
If a(j) <> b(j) then
    c(j+1) = c(j)

Carry Lookahead Circuit



a(j)	b(j)	c(j+1)	Status x(j+1)
0	0	0	Kill (k)
0	1	c(j) = 0	Propagate (p)
0	1	c(j) = 1	
1	0	c(j) = 0	Propagate (p)
1	0	c(j) = 1	
1	1	1	Generate (g)

Carry Lookahead Circuit



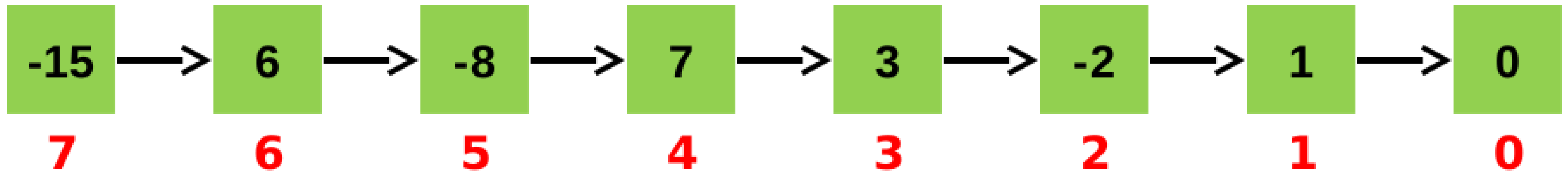
	x(j+1)			
	(*)	k	p	g
x(j)	k	k	k	g
	p	k	p	g
	g	k	g	g

← The carry status of jth and (j+1)th full adders and the resultant is the carry status of the combination of both the full adders

If x(j+1) = k then output carry status = k
If x(j+1) = g then output carry status = g
If x(j+1) = p then output carry status = status of x(j)

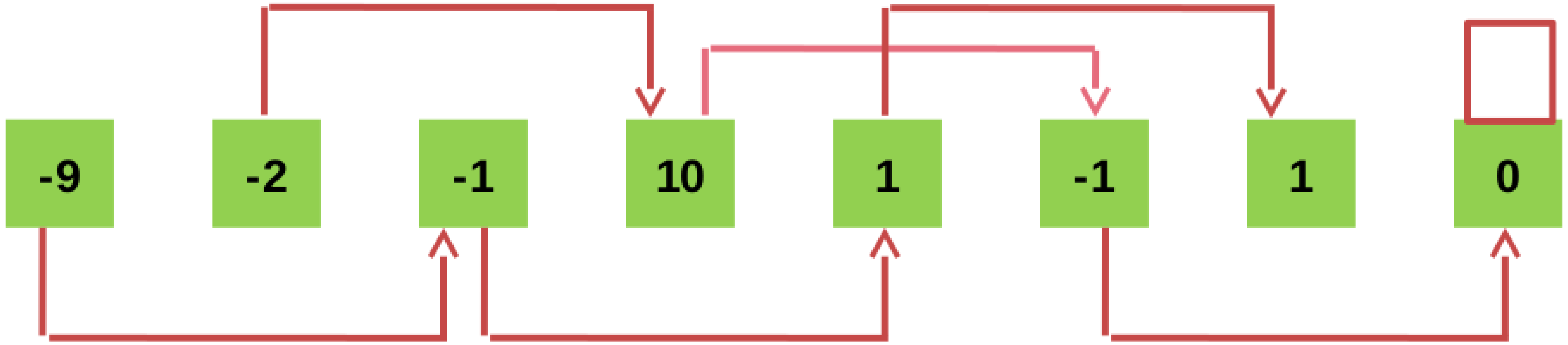
Parallel Techniques: Recursive Doubling

- Finding Prefix sum of ‘8’ numbers



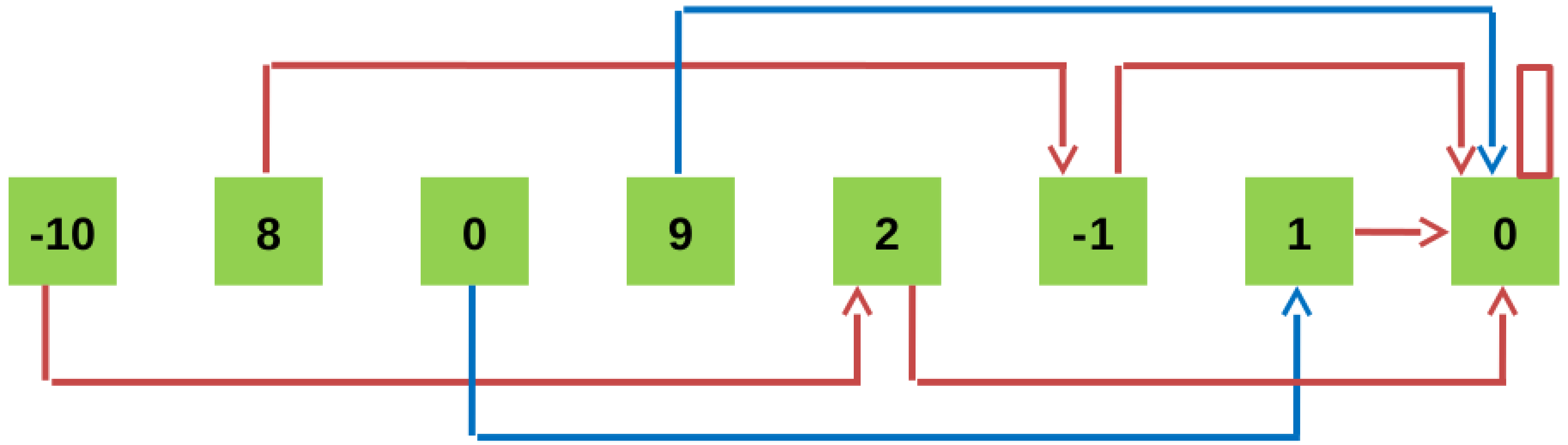
Recursive Doubling (Step 1)

- Finding Prefix sum of ‘8’ numbers



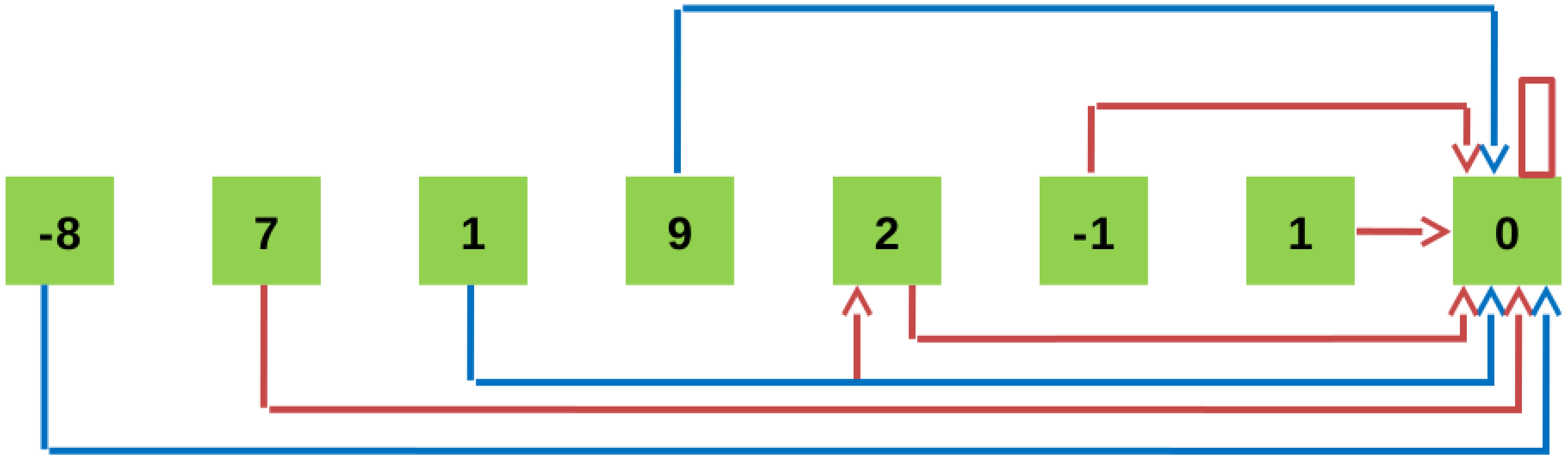
STEP 1

Recursive Doubling (Step 2)



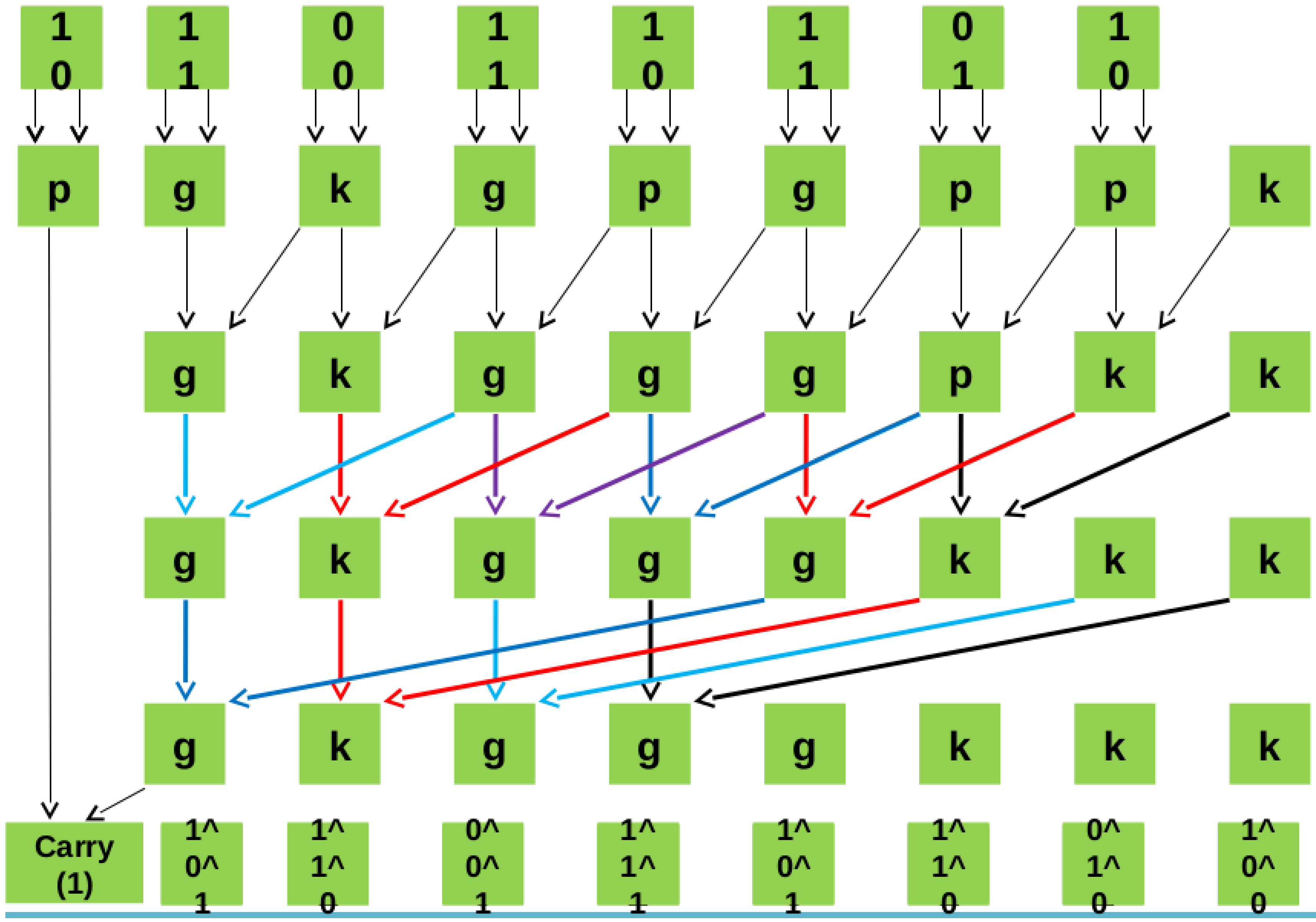
STEP 2

Recursive Doubling (Step 3)



STEP 3

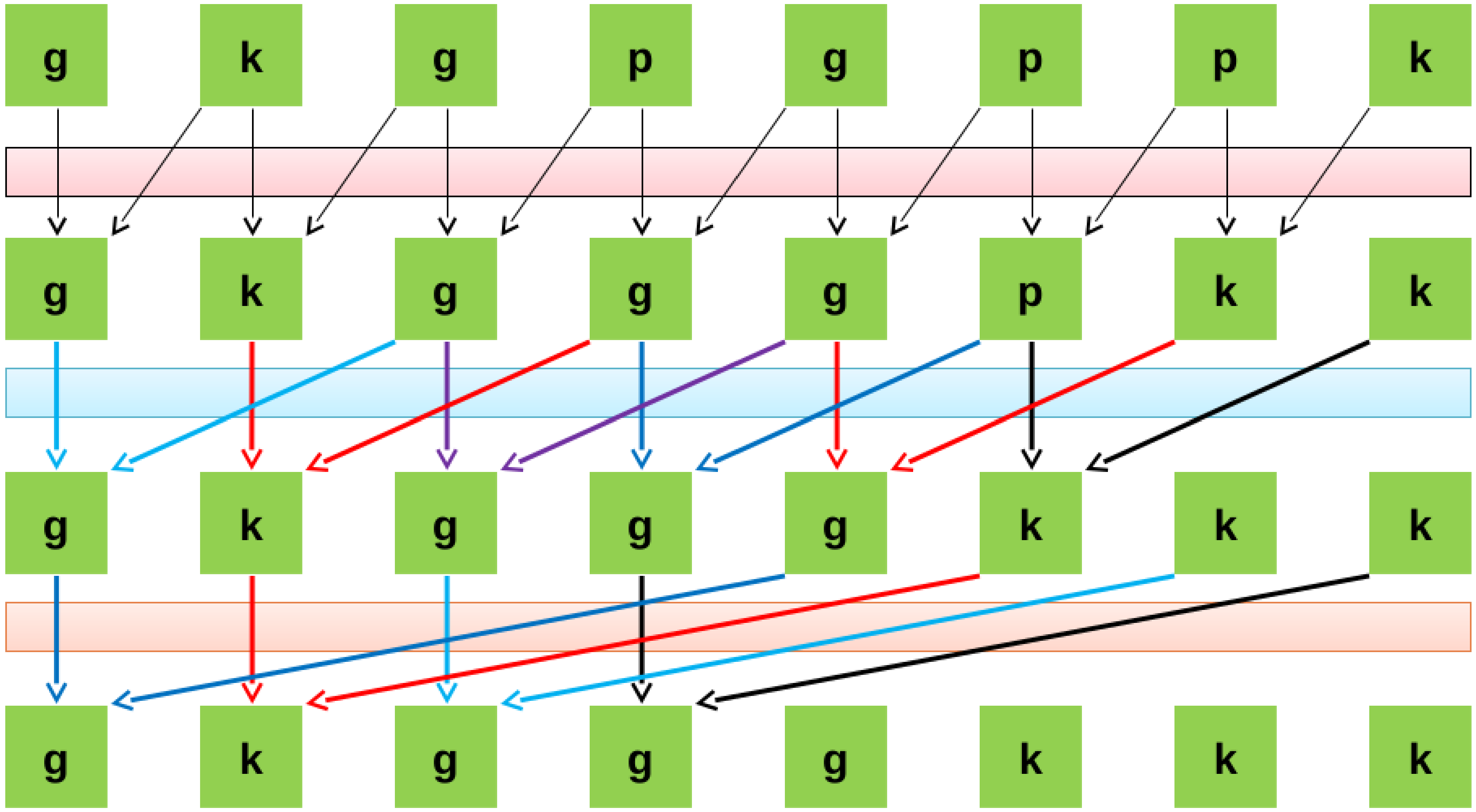
Prefix Calculation Based on Recursive Doubling Technique



Outcome of Above Method

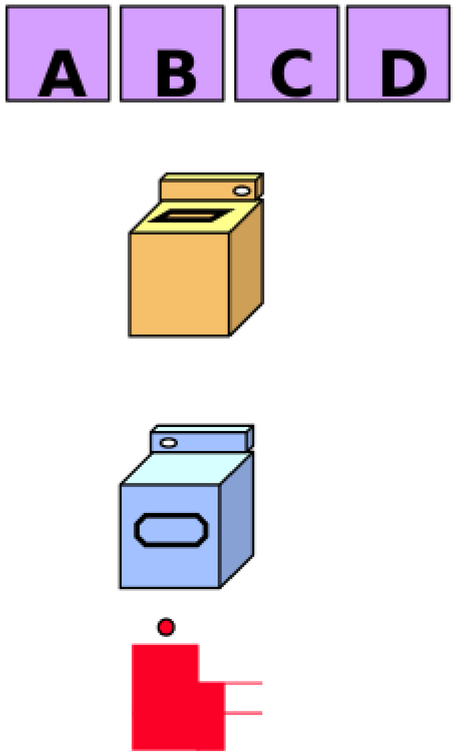
- Depth of the circuit reduced from $O(n)$ to $O(\log_2 n)$
- This results in a Fast Adder
- The circuit complexity of the CLA is $O(n.\log_2 n)$, because it includes $\log_2 n$ stages and each requires $O(n)$ prefix units.

Pipelined Prefix Calculation Based on Recursive Doubling Technique



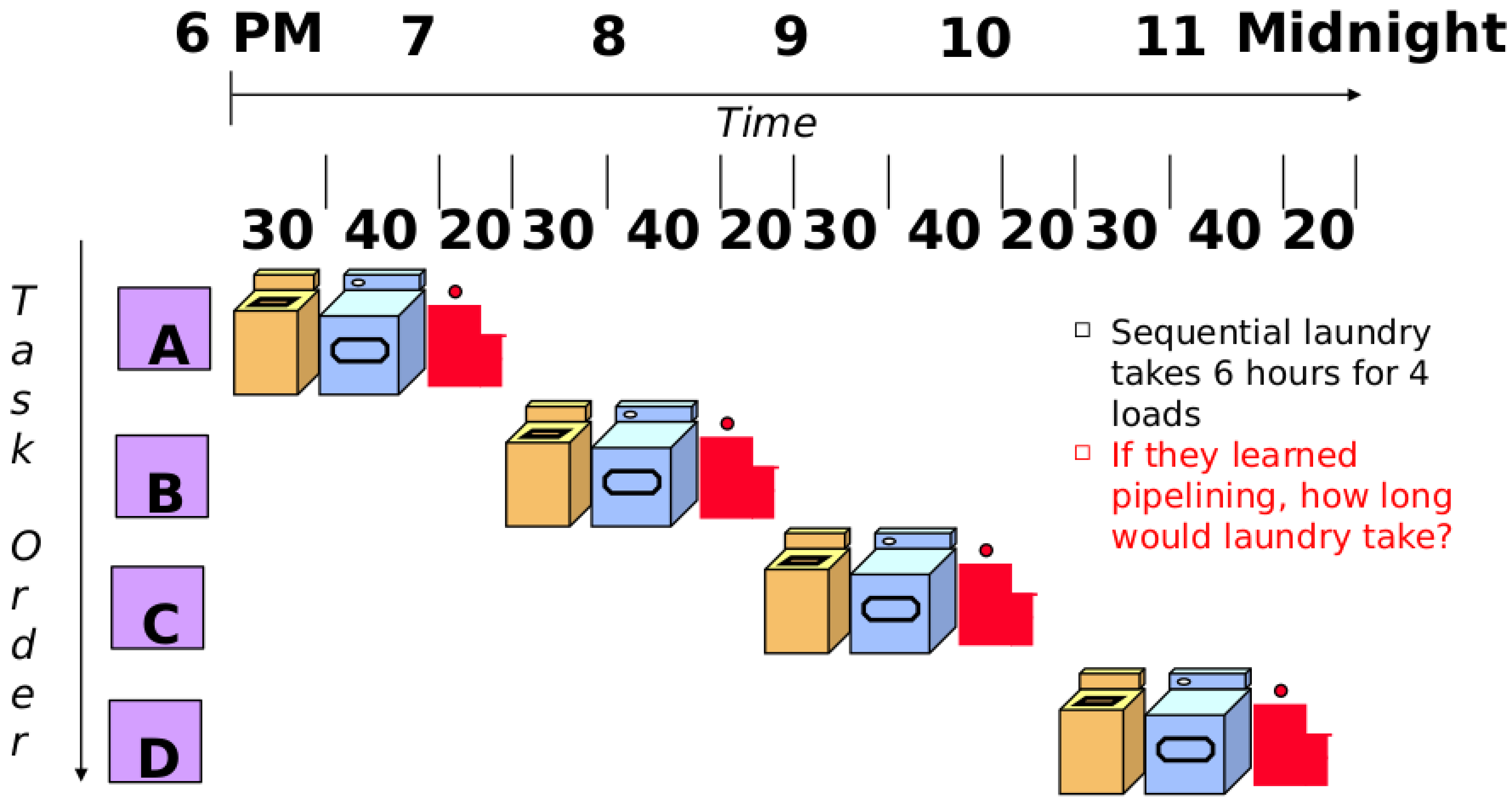
Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave
- each have one load of clothes
- to wash, dry, and fold
- Washer takes 30 minutes



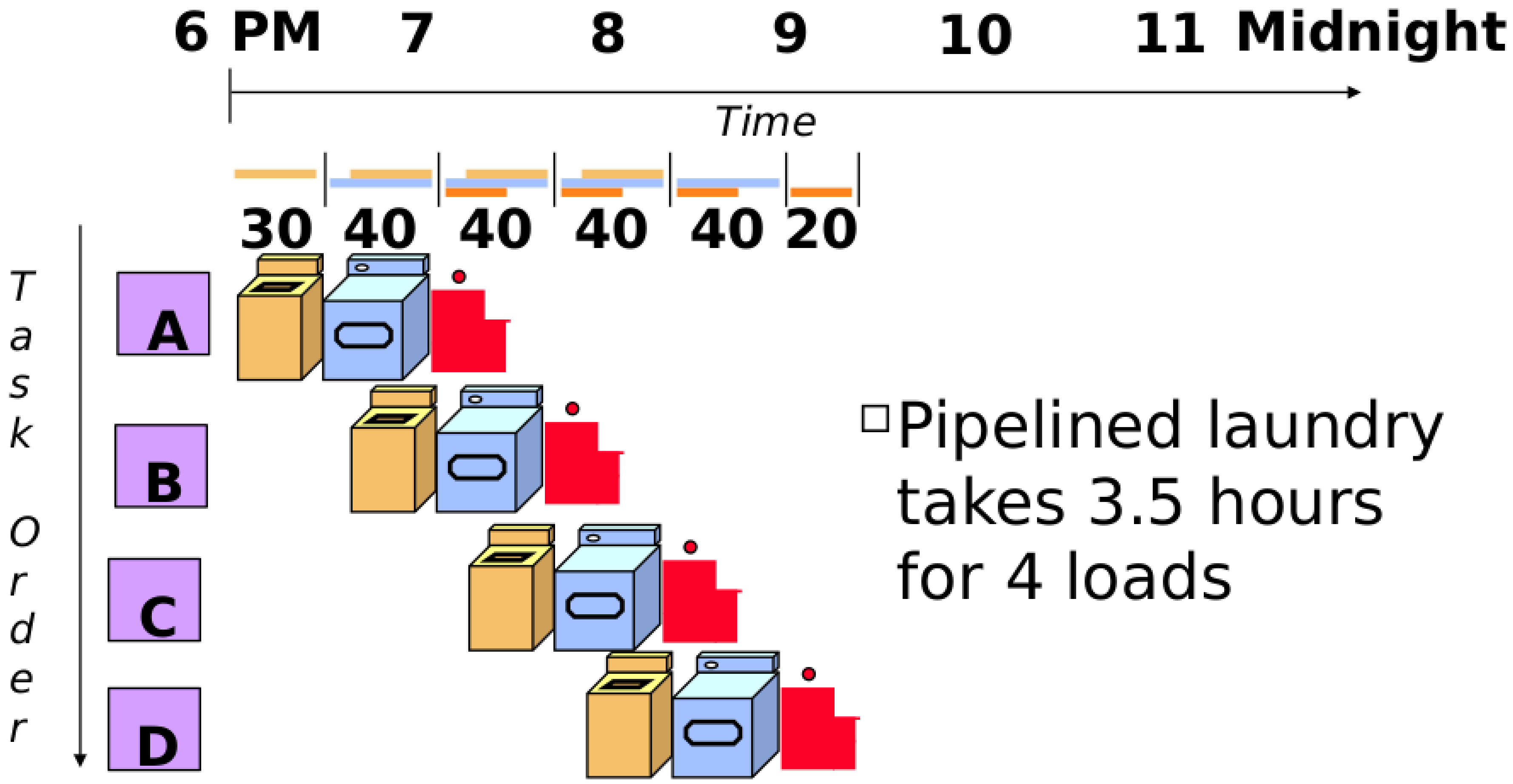
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry

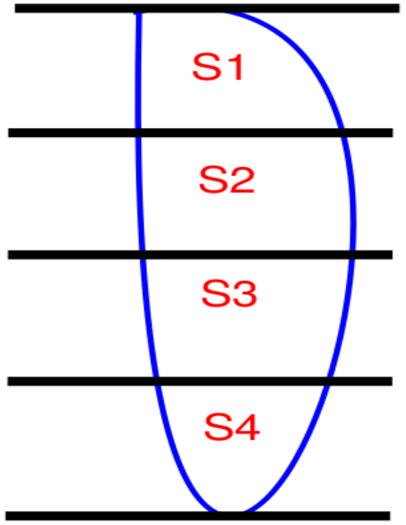


- Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons

- Pipelining does not help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup

Pipelining in Digital Circuits

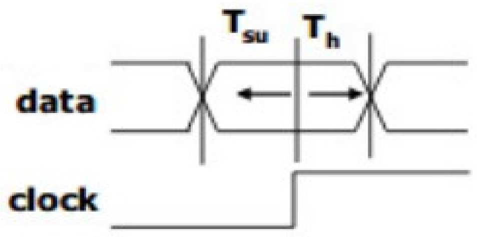


Clock	1	2	3	4	5	6	7
Task1	S1	S2	S3	S4			
Task2		S1	S2	S3	S4		
Task3			S1	S2	S3	S4	
Task4				S1	S2	S3	S4

Total time $= (K + n - 1)T_c$
K= No. of segments
n= No. of tasks
T_c = Clock period

Slack, Skew, Setup time, and Hold time

- ***Slack*** is the difference between the expected data arrival time
- and the actual data arrival time.
- If a clock edge does not arrive at different flip-flops at same time, then
- the clock is said to be ***skewed***.
- The difference between the times of arrival at flip-flops is known
- as the amount of ***clock skew.***
- ***Positive skew*** occurs when the transmitting register receives
- the clock tick earlier than the receiving register.
- ***Negative skew*** is the opposite: the receiving register gets the clock tick earlier than the sending register. It is similar to ***hold violation***.
- ***Zero clock skew*** refers to the arrival of the clock tick simultaneously at transmitting and receiving register. Here, the data is moved from the trasmitting register to receiving register.
- ***Setup time (T_{su})*** : The minimum time before the clock triggering by which the data must be stable.
- ***Hold time (T_h)*** : The minimum time after the clock triggering in which the data must be stable.



Setup Violation

- In a positive edge triggered flip-flop, input signal is captured on the positive edge
- of the clock and corresponding output is generated after a small delay called
- the T_{c2q}.
- T_{c2q} is known as a propagation delay between the clocking event and change in
- output. The flip flop can only do the job correctly if the data at its input does not
- change for some time before the clock edge (T_{setup}) and some time after the clock
- edge (T_{hold}).
- If the setup check is violated, data will not be captured properly at the next clock edge. Here, $T_{c2q} + T_{comb} + T_{setup} \leq T_{clk} + T_{skew}$.
- The difference between left and right sides is represented by a parameter known as setup slack, $T_{setup\ slack} = T_{clk} - T_{c2q} - T_{comb} - T_{setup} + T_{skew}$.
- If setup slack is positive, it means the timing path meets setup requirement.
- On the other hand, a negative setup slack means setup violating timing path. We can try one or more of the following to bring the setup slack back to a positive value by:

(a) Decreasing data path delay,
(b) Choosing a flip-flop with less setup time requirement,
(c) Increasing clock skew.

Hold Violation

- If hold check is violated, data intended to get captured at the next edge will get
- captured at the same edge. Here, the short data path delay occurs (delay of
- clock path > delay of data path).
- It can be removed in some cases by clock reversing and Clocking on Alternate Edges. Here, $T_{c2q} + T_{comb} \geq T_{hold} + T_{skew}$ and $T_{hold\ slack} = T_{c2q} + T_{comb} - T_{hold} - T_{skew}$.
- If a timing path violates for hold, we can do either of the following:
- (a) Increase data path delay;
- (b) Decrease clock skew
- (c) Choose a flip-flop with less hold requirement

In the Clock Reversing technique, clock is applied in the reverse direction with respect to data so that clock skew (hold violation) is automatically eliminated at the cost of positive skew. So, the destination register will clock in the source register (current) value before the source register receives it's clock edge.In the Clocking on Alternate Edges techniques sequentially adjacent Flops are clocked on the opposite edges of the clock.

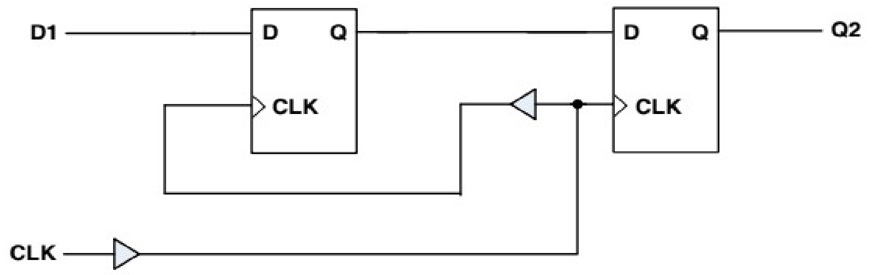


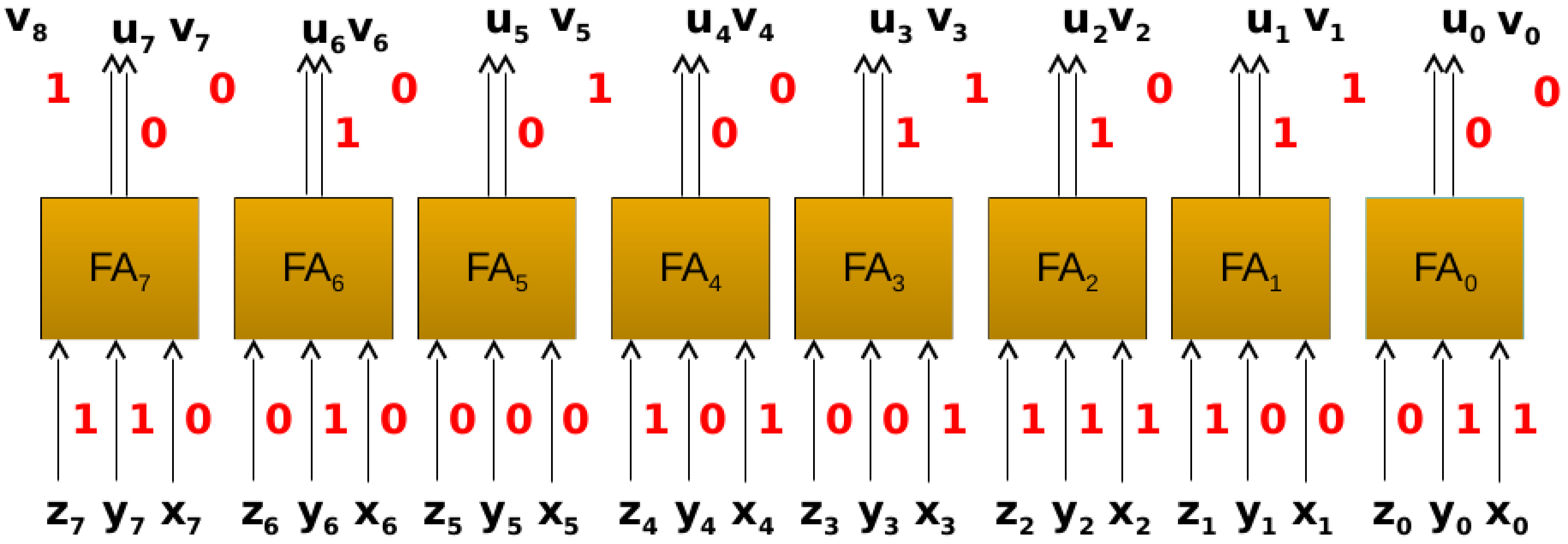
Fig. Clock reversing methodology

Carry Save Addition Example

8	7	6	5	4	3	2	1	0		i
	0	0	0	1	1	1	0	1	=	x
	1	1	0	0	0	1	0	1	=	y
	1	0	0	1	0	1	1	0	=	z
	0	1	0	0	1	1	1	0	=	u
1	0	0	1	0	1	0	1	0	=	v

(1) Time complexity is *O(1)* because sum *(u)* and carry *(v)* can be generated in a single step.
(2) Circuit complexity is *O(n)* because it requires *n* full adders.

Carry Save Adder Circuit



v_8 u_7 v_7 u_6 v_6 u_5 v_5 u_4 v_4 u_3 v_3 u_2 v_2 u_1 v_1 u_0 v_0
1 0 0 1 0 0 1 0 0 1 1 0 1 1 1 0 0
FA_7 FA_6 FA_5 FA_4 FA_3 FA_2 FA_1 FA_0
1 1 0 0 1 0 0 0 0 1 0 1 0 0 1 1 1 1 1 0 0 0 1 1
z_7 y_7 x_7 z_6 y_6 x_6 z_5 y_5 x_5 z_4 y_4 x_4 z_3 y_3 x_3 z_2 y_2 x_2 z_1 y_1 x_1 z_0 y_0 x_0

Multiplication of two 5-bit numbers

$$
\begin{array}{l|cccccccccc}
 & & & & & & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & \times & x_4 & x_3 & x_2 & x_1 & x_0 \\
\cline{6-11}
ax_0\,2^0 & & & & & & a_4x_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
ax_1\,2^1 & & + & & & a_4x_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
ax_2\,2^2 & & & & a_4x_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
ax_3\,2^3 & & & a_4x_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
ax_4\,2^4 & & a_4x_4 & a_3x_4 & a_2x_4 & a_1x_4 & a_0x_4 & & & & \\
\hline
 & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$

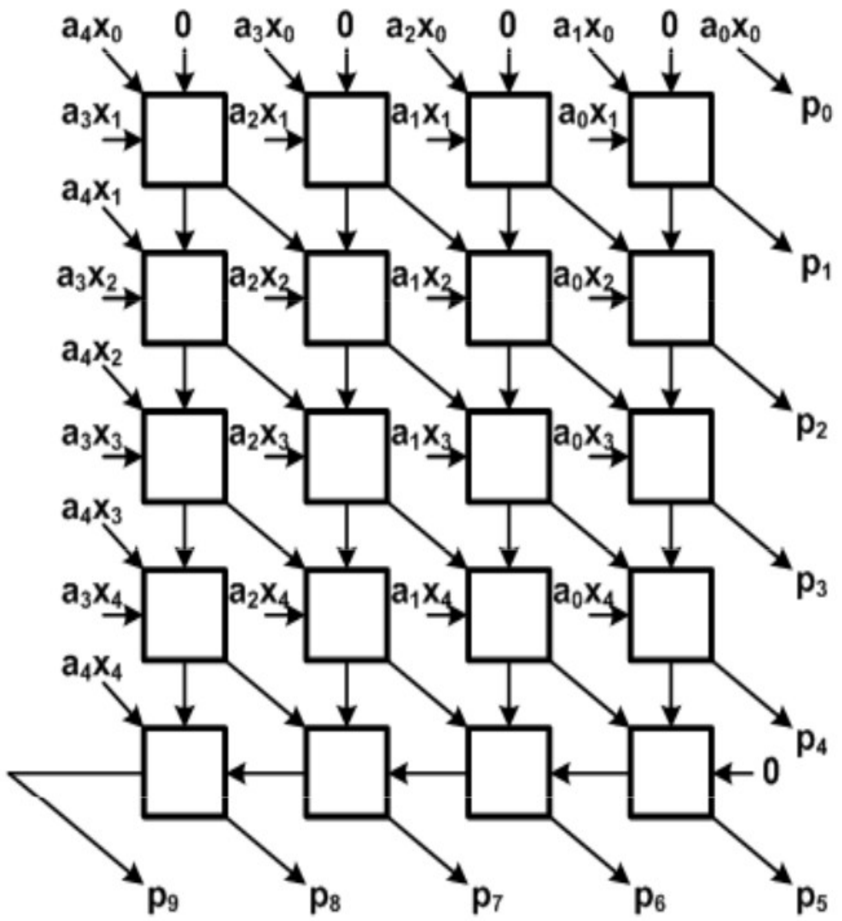
Squaring the 5-bit number

Multiply x by x									
					x_4	x_3	x_2	x_1	x_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					x_4x_0	x_3x_0	x_2x_0	$\boxed{x_1x_0}$	$\boxed{x_0x_0}$
				x_4x_1	x_3x_1	x_2x_1	x_1x_1	$\boxed{x_0x_1}$	Reduce
			x_4x_2	x_3x_2	x_2x_2	x_1x_2	x_0x_2	Move	to x_0
		x_4x_3	x_3x_3	x_2x_3	x_1x_3	x_0x_3		to next	
	x_4x_4	x_3x_4	x_2x_4	x_1x_4	x_0x_4			column	
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Reduce the bit matrix									
					x_4	x_3	x_2	x_1	x_0
				$\times$	x_4	x_3	x_2	x_1	x_0
	x_4x_3	x_4x_2	x_4x_1	x_4x_0	x_3x_0	x_2x_0	x_1x_0	-	x_0
	x_4		x_3x_2	x_3x_1	x_2x_1		x_1		
			x_3		x_2				
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	0	x_0

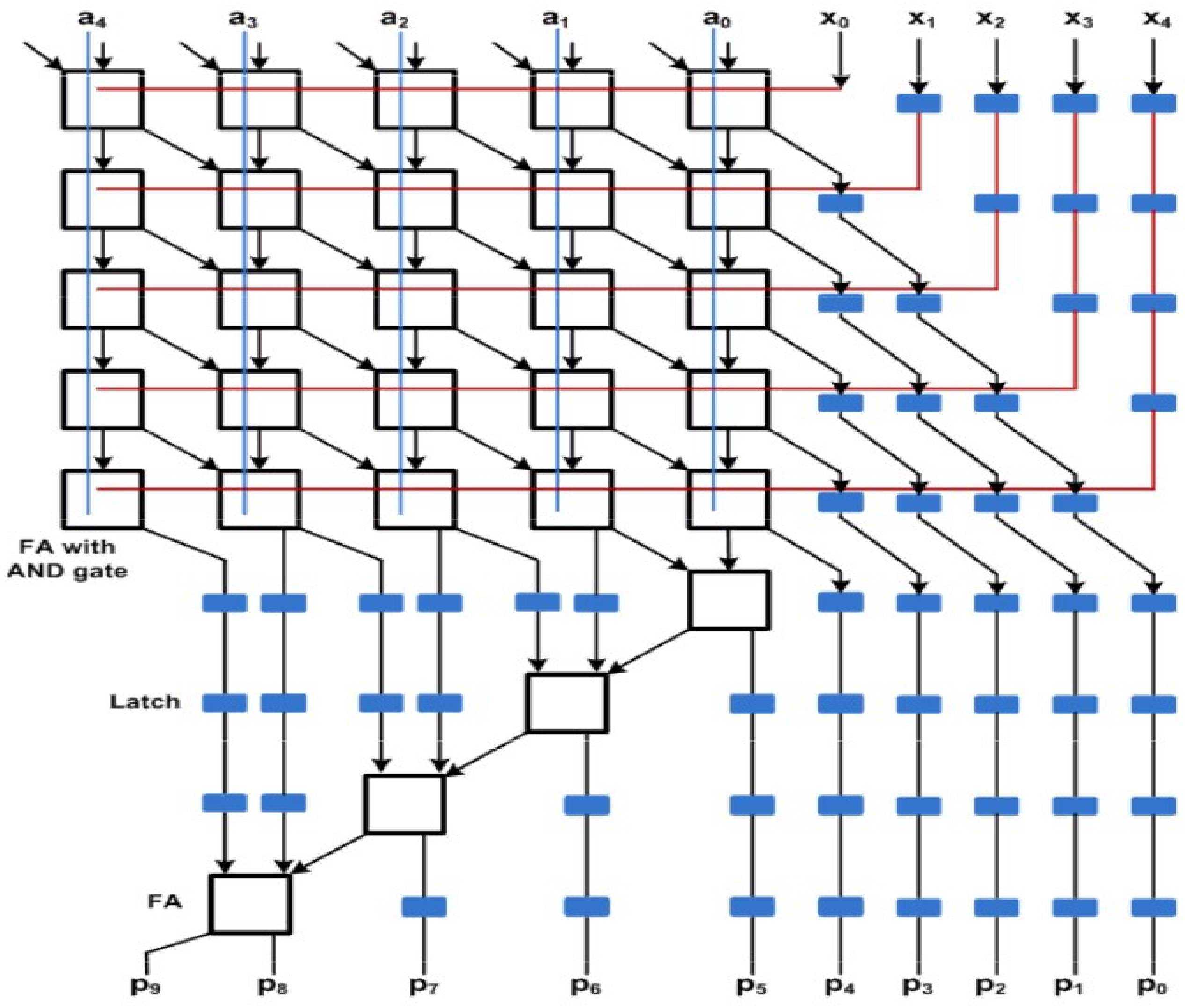
Carry Save Array Multiplier

- Sum outputs are connected diagonally, while the carry outputs are linked vertically, except in the last row, where they are chained from right to left



(1) Time complexity is *O(n)*.
(2) Circuit complexity is *O(n^2)* because it requires *n* rows, first row requires *n* *HAs and rests require n FAs*.

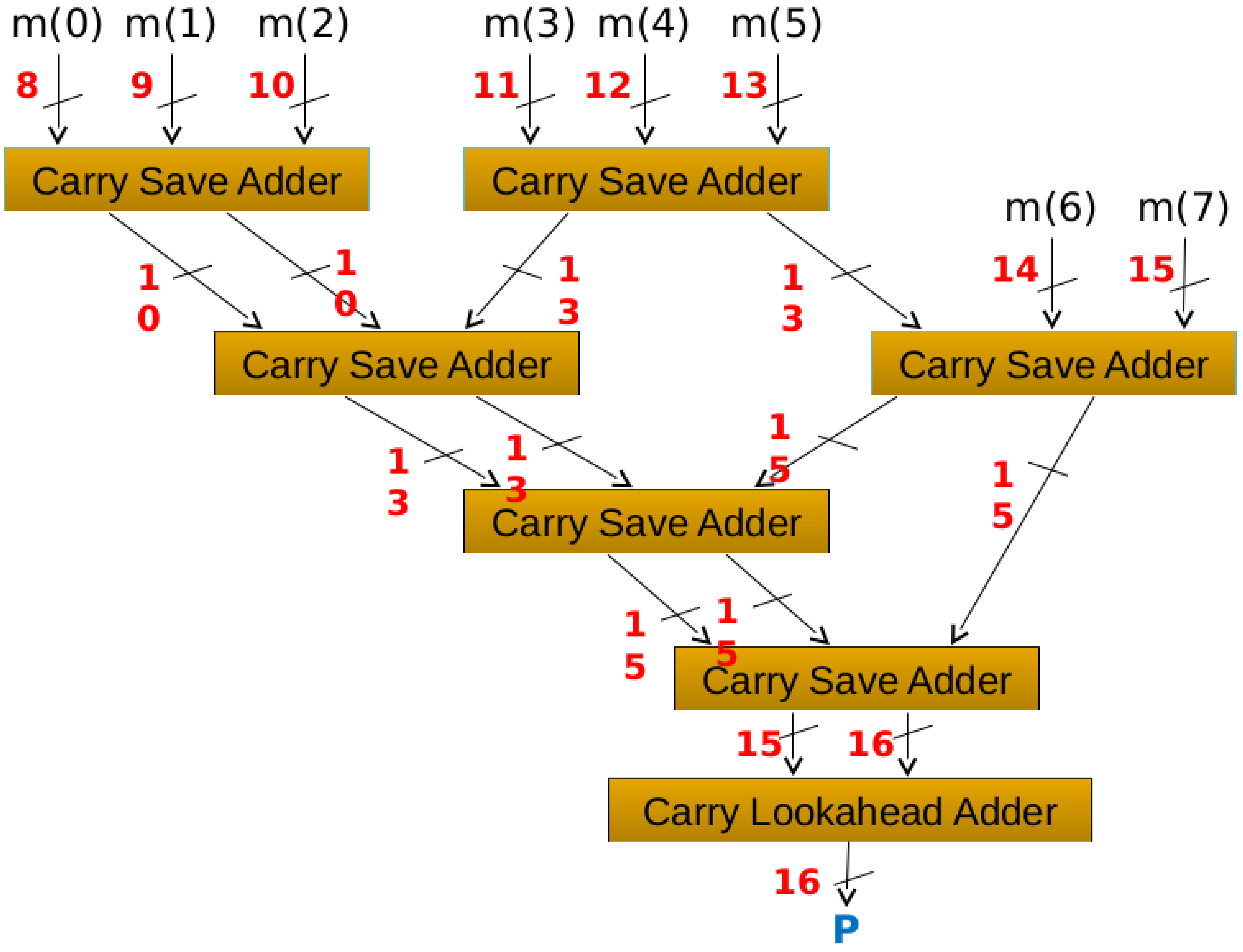
Carry Save Array Multiplier with pipeline



Wallace Tree Multipliers

- While multiplying two n-bit numbers a total of n partial products have to be added.
- In Wallace structure, the number of outputs at each level is equal to $(2/3)^{rd}$ of the outputs in the previous level.
- The number of outputs at 1st, 2nd, 3rd, and last CSA levels are *(2n/3), (2.2n/3.3), (2.2.2n/3.3.3),* and *2* respectively.
- The last level of CSA tree *h* is $(2/3)^h.n=2$
- Therefore, $h=-1.71+1.71\log_2 n$

8-bit Wallace Tree Multiplication



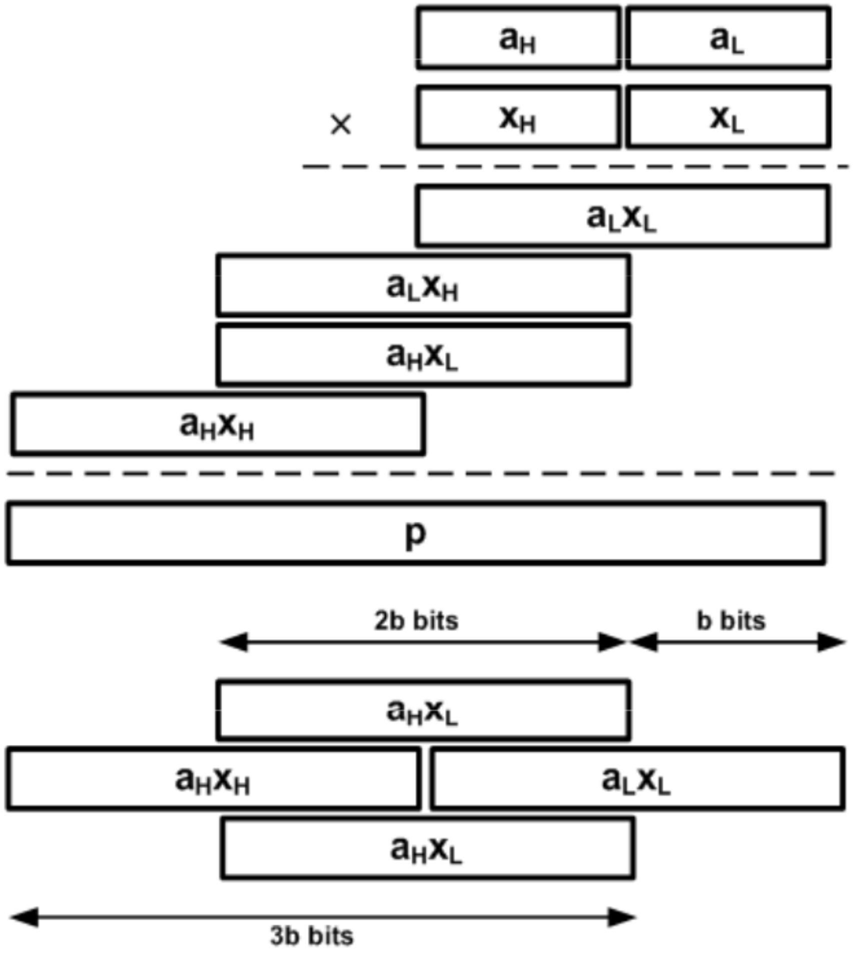
Wallace Tree – Circuit Complexity

- ◆First row requires *(n/3)* CSAs and each with *n* FAs
- ◆Second row requires *(2n/3.3)* CSAs and each with *n* Fas
- ◆Third row requires *(2.2n/3.3.3)* CSAs and each with *n* FAs
- ◆Last row requires *1* CSA with *n* FAs.
- ◆Therefore, the circuit complexity is $O(n^2)$

Multiplication Using Divide & Conquer

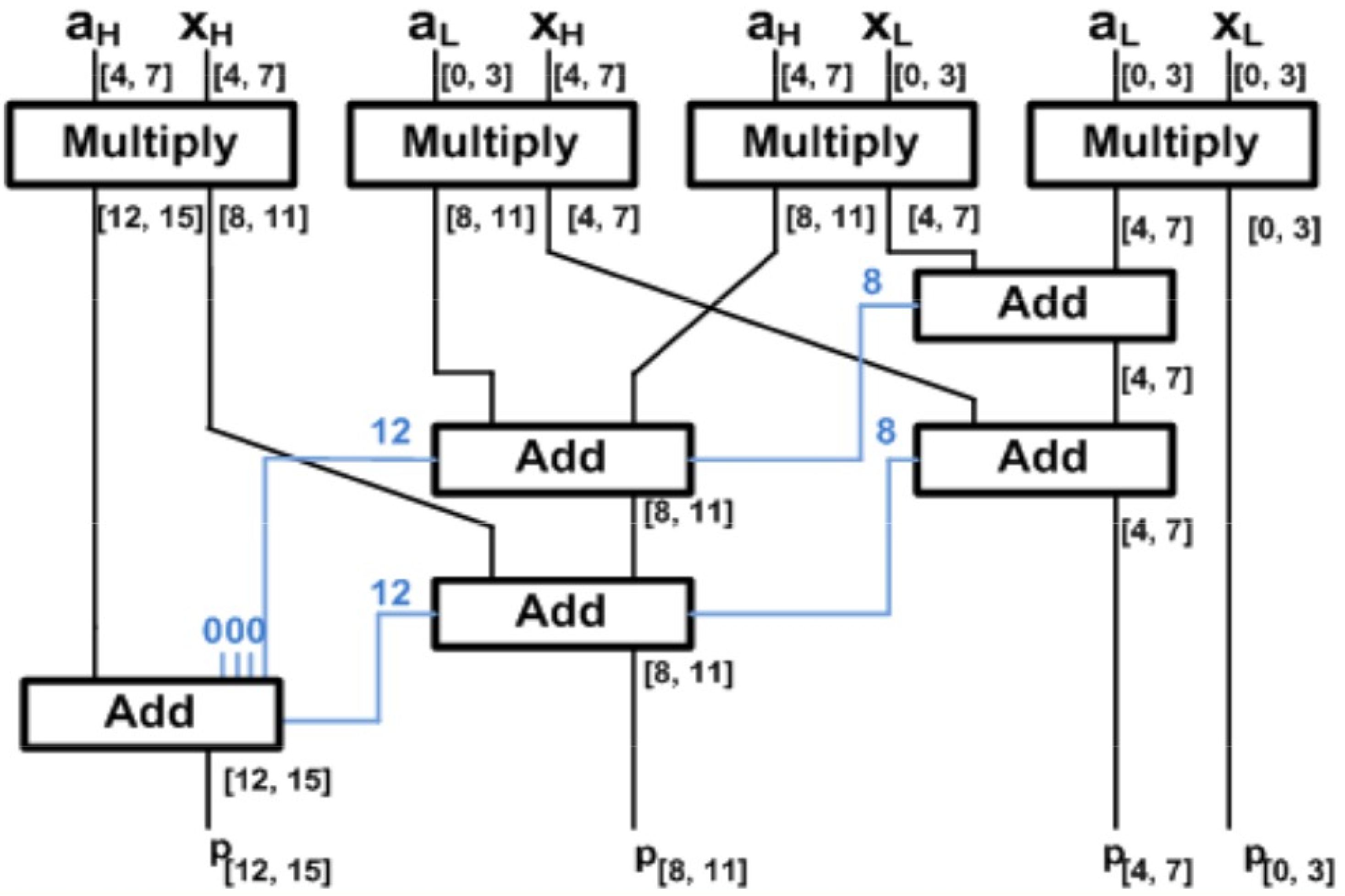
In general, the multiplication of two *n*-bit numbers $a = (10^{n/2}.a_H)+a_L$ and $x = (10^{n/2}.x_H)+x_L$ can be represented as $a.x = (2^n.a_H .x_H) + (2^{n/2}.a_H.x_L) + (2^{n/2}.x_H.a_L) + (a_L.x_L)$, where a_H, a_L, x_H, and x_L are $n/2$-bit numbers. ax is $2n$-bit number.

- A 2b×2b multiplier can be synthesized using b×b multiplier
- Although there are four partial products, only three values need to be added
- 2b×2b multiplication has been reduced to 4 b×b multiplications and a three-operand addition



Multiplication Using Divide & Conquer

- For 2b×2b multiplication one can use b-bit adders exclusively to accumulate the partial products



Modified Booth Algorithm

- MBA is used to reduce the number of partial products of the multiplier.
- In general, MBA with radix *r* can be represented as $b=1+\log_2 r$, where *b* represents the number of bits grouped.
- Partial product generation involves the multiplication of $X=\{x_{n-1}, \ldots x_1, x_0\}$ with

$d_i \in \{-r/2, \ldots 0, \ldots r/2\}$, where *X* is *n*-bit multiplicand and d_i is an integer obtained from the *n*-bit multiplier $Y=\{y_{n-1}, \ldots y_1, y_0\}$ by radix-*r* Booth encoding technique.

Modified Booth Algorithm Contd...

- The value of $X.d_i$ can be obtained by shift/2's complement/addition of X with respect to d_i .

- For example, $7X$ can be obtained by $(4X + 2X + X)$. If r is increased, then the largest odd value of d_i will be increased, which leads to increase the number of additions to get $X.d_i$.

- The following equations show d_i generated by Booth encoder for different radix values.

Radix 2 : $d_i = -y_i+y_{i-1}$

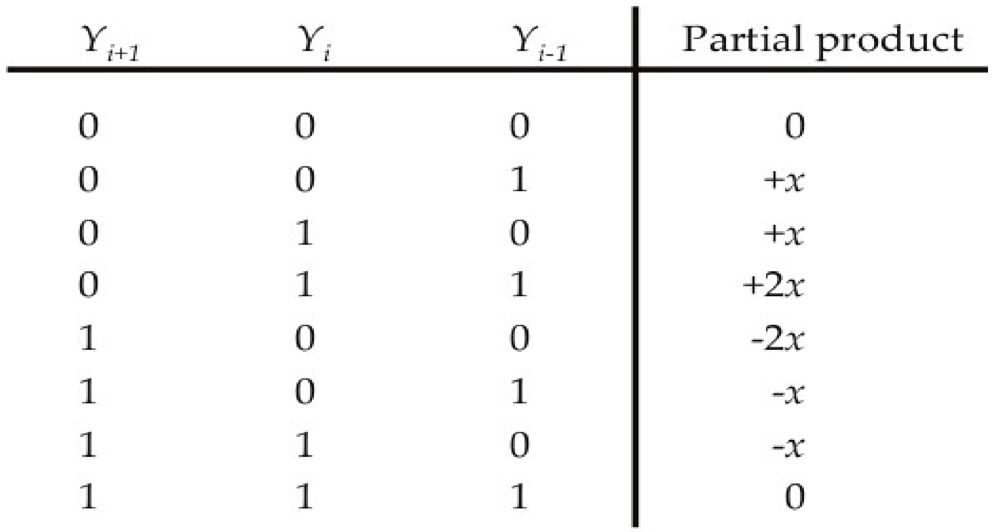
Radix 4: $d_i = -2y_{2i+1}+y_{2i}+y_{2i-1}$

Radix 8: $d_i = -4y_{3i+2}+2y_{3i+1}+y_{3i}+y_{3i-1}$

Radix 16: $d_i= -8y_{4i+3}+4y_{4i+2}+2y_{4i+1}+y_{4i}+y_{4i-1}$

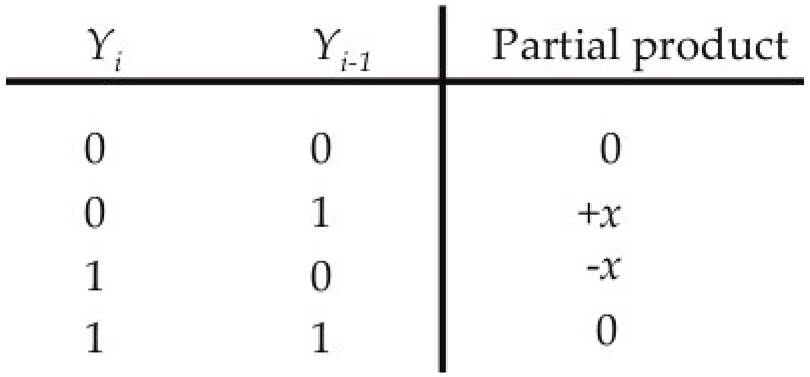
Modified Booth Algorithm Contd...

Here, the multiplicand and multiplier are *x* and *y* respectively.



Y_{i+1}	Y_i	Y_{i-1}	Partial product
0	0	0	0
0	0	1	$+x$
0	1	0	$+x$
0	1	1	$+2x$
1	0	0	$-2x$
1	0	1	$-x$
1	1	0	$-x$
1	1	1	0

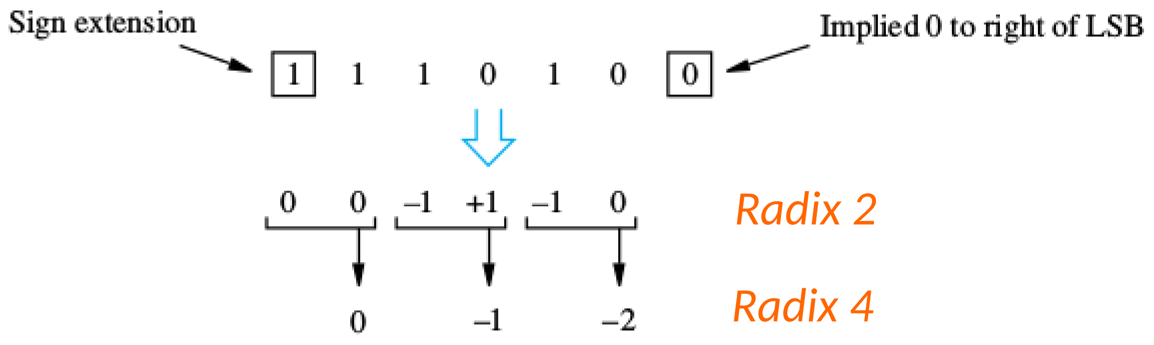
Radix-4



Y_i	Y_{i-1}	Partial product
0	0	0
0	1	$+x$
1	0	$-x$
1	1	0

Radix-2

Booth Algorithm based Multiplication - Example



Example of bit-pair recoding derived from Booth recoding

13x(-1)=?

13=01101

1's complement=10010

2's complement=10010+1

=10011

=1110011

13x(-2)=?

26=011010

1's complement=100101

2's complement=100101+1

=100110=11100110

```
              0  1  1  0  1   (+13)
           ×  1  1  0  1  0   (–6)
           ----------------
Radix 2       0  1  1  0  1
              0 –1 +1 –1  0
           ----------------
0  0  0  0  0  0  0  0  0  0
1  1  1  1  1  0  0  1  1
0  0  0  0  1  1  0  1
1  1  1  0  0  1  1
0  0  0  0  0  0
----------------------------
1  1  1  0  1  1  0  0  1  0  (–78)

Radix 4       0  1  1  0  1
              0    –1    –2
           ----------------
1  1  1  1  1  0  0  1  1  0
1  1  1  1  0  0  1  1
0  0  0  0  0  0
----------------------------
1  1  1  0  1  1  0  0  1  0
```

Thank You

Queries?