دانشگاه کردستان
University of Kurdistan
زانکۆی کوردستان

Department of Computer Engineering
University of Kurdistan

Computer Architecture

Computer Arithmetic

By: Dr. Alireza Abdollahpouri

پیش نیاز

- به یاد آورید که:
  - با n بیت 2^n نمایش مختلف و یکتا می‌توان داشت.
- نحوه ی نوشتن: $b_{31}\ b_{30}\ \ldots\ b_3\ b_2\ b_1\ b_0$
- معنای ذاتی و خاصی ندارد:
  - می تواند یک عدد طبیعی باشد.
  - می تواند یک عدد اعشاری باشد.
  - می تواند سیگنالهای کنترلی باشد.
  - می تواند یک دستورالعمل باشد.

پیش نیاز

- با ۳۲ بیت می توان:
  - اعداد طبیعی بدون علامت را نمایش داد.
  - اعداد طبیعی علامت دار را مدل کرد.
  - اعداد اعشاری با دقت معمولی را نمایش داد.
  - دستورالعملهای MIPS را مدل نمود.

اعداد طبیعی بدون علامت

- اعداد طبیعی در قالب باینری

- $f(b_{31}\ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$

  E.g. 0…11011001

  $= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^1 + 1 \times 2^0$

  $= 128 + 64 + 16 + 4 + 1 = 213$

- Max $f(111\ldots11) = 2^{32} - 1 = 4{,}294{,}967{,}295$
- Min $f(000\ldots00) = 0$
- Range $[0, 2^{32}-1]$ => # values $(2^{32}-1) - 0 + 1 = 2^{32}$

اعداد صحیح

- مکمل ۲

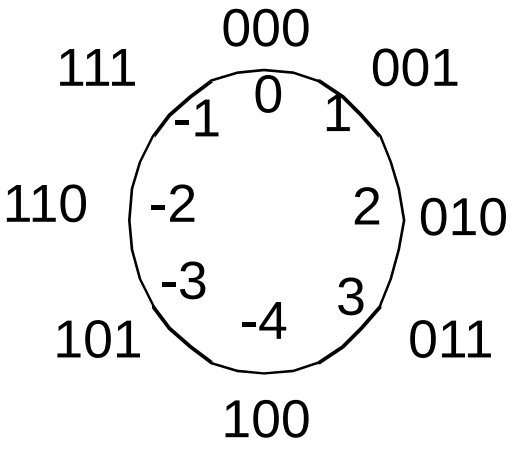
$f(b_{31}\ldots b_0) = -b_{31} \times 2^{31} + \ldots b_1 \times 2^1 + b_0 \times 2^0$

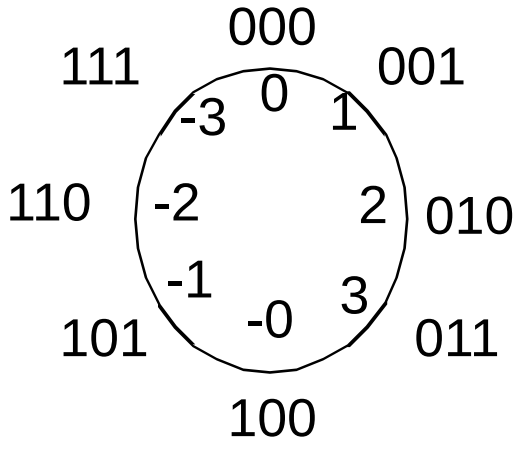
- Max f(0111…11) = $2^{31} - 1$ = 2147483647
- Min f(100…00) = -2^{31} = -2147483648 (نامتقارن)
- Range[-2^{31},$2^{31}-1$] => # values($2^{31}-1 - -2^{31}$) = 2^{32}

- بیتها را معکوس کنید و با عدد یک جمع بزنید: 6–
- 000…0110 => 111…1001 + 1 => 111…1010

چرا مکمل ۲ ؟

- چرا از روش علامت-عدد استفاده نمی کنیم؟
- طراحی سخت افزار مکمل ۲ راحتتر است.
- درست مثل انسانها که کار با اعداد رومی برای آنها سخت است.
- شیوه نمایش باعث تسهیل محاسبات می شود ولی روی درستی محاسبات تاثیری ندارد.





جمع و تفریق

- اعداد ۴ بیتی بدون علامت

0	0	1	1		3
1	0	1	0		10
1	1	0	1		13

- اعداد ۴ بیتی مکمل دو - صرفنظر کردن از سرریز

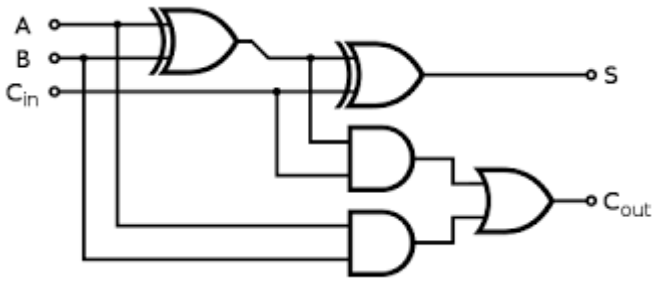
0	0	1	1		3
1	0	1	0		-6
1	1	0	1		-3

تفریق

- A – B = A + 2’s complement of B
- مثال: 3 – 2 =1

0	0	1	1		3
1	1	1	0		-2
0	0	0	1		1

جمع کننده کامل(Full Adder)

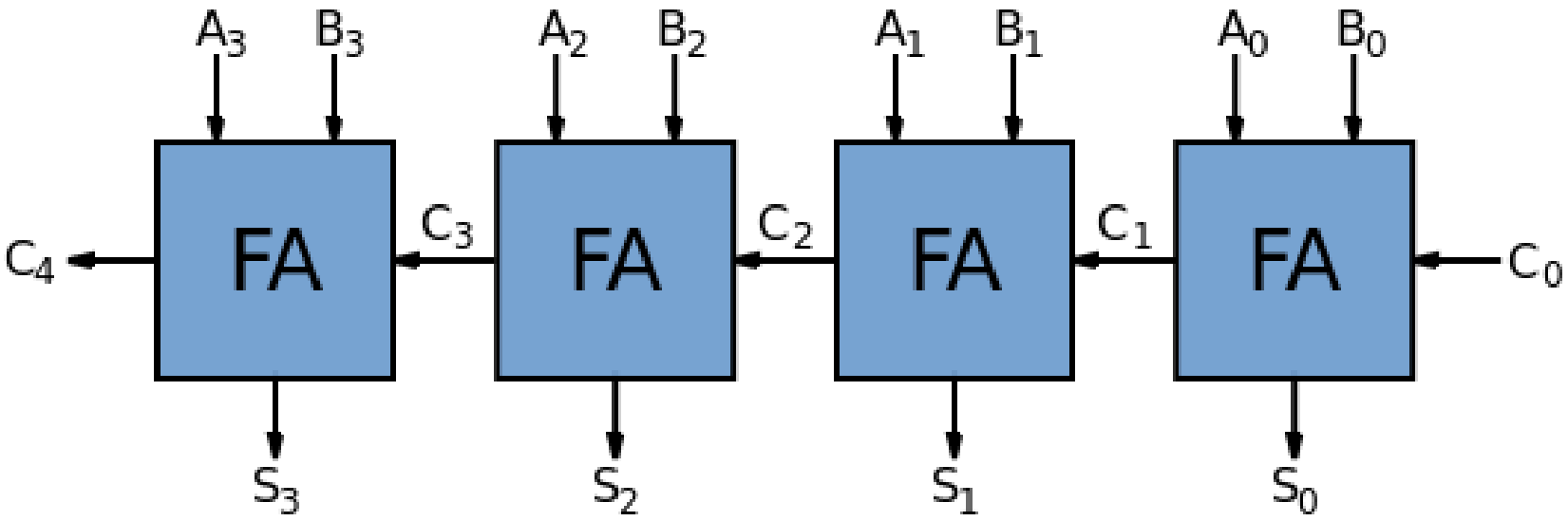


- Full adder (a,b,c_{in}) => (c_{out}, s)
- c_{out} = two or more of (a, b, c_{in})
- s = exactly one or three of (a,b,c_{in})

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

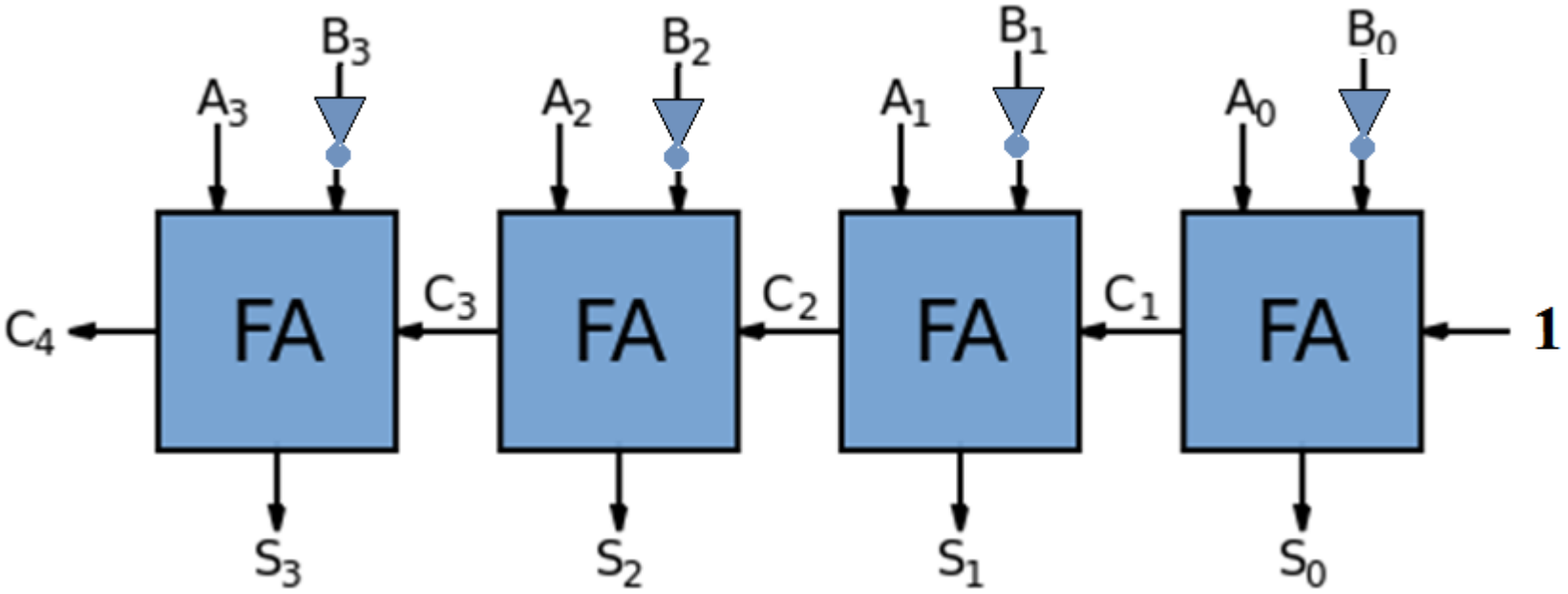
جمع کننده ریپل

- جمع کننده ها را پشت سر هم به هم وصل کنید.



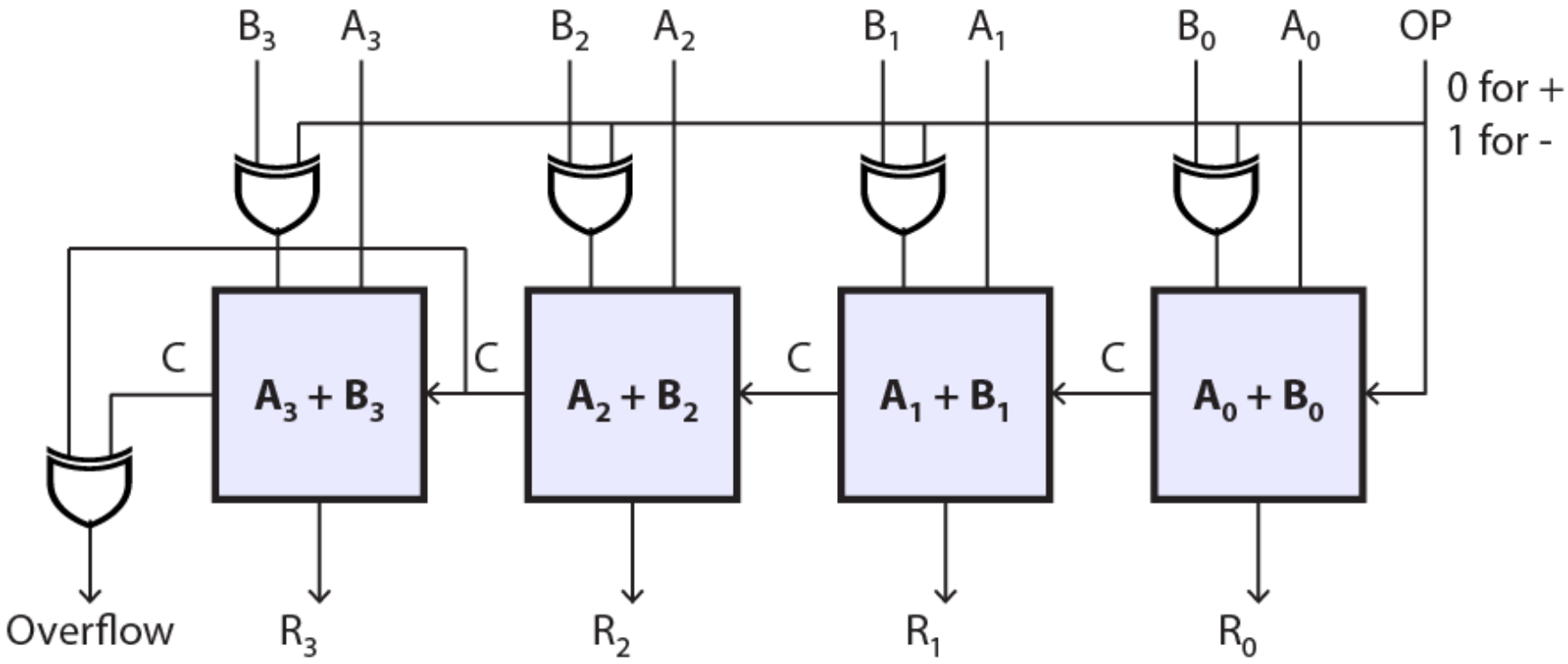
تفریق کننده ریپل

- A – B = A + (-B) => invert B and set c_{in} to 1



تجمیع جمع کننده و تفریق کننده

- Control = 1 => subtract
- XOR B with control and set c_{in0} to control



پیش بینی رقم نقلی

- ALU قبلی خیلی کند است.
  - تاخیر گیتها برای عمل جمع: 32 x FA + XOR

- باید مصالحه کنیم:
  - درخت را طوری بسازیم که تاخیر برای n بیت برابر $O(\log_2 n)$ باشد.
  - مثلا برای ۳۲ بیت تاخیر در حدود 2 x 5 گیت باشد.

- ما از حالت ۴ بیتی برای تشریح این مفهوم استفاده می کنیم.

پیش بینی رقم نقلی

برای تولید رقم نقلی در هر سطح به دو ۱ نیاز داریم. برای انتشار رقم نقلی مرحله جاری به مرحله بعد حداقل به یک ۱ نیاز داریم.

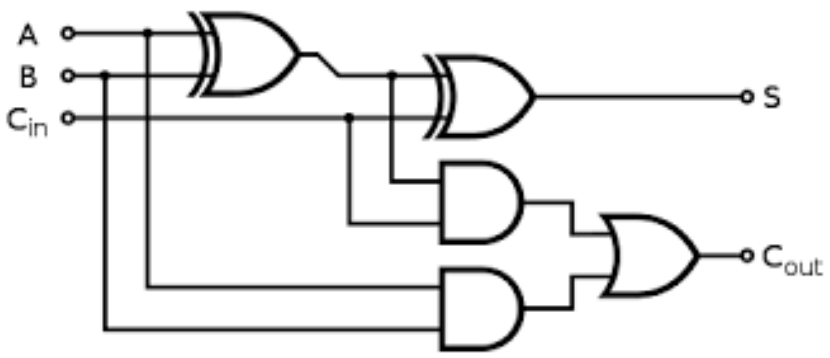
تعریف می کنیم:

$g_i = a_i * b_i$ **# carry generate**

$p_i = a_i \oplus b_i$ **# carry propagate**

به یاد بیاورید:

$$c_{i+1} = a_i * b_i + (a_i \oplus b_i) * c_i$$
$$= g_i + p_i * c_i$$



پیش بینی رقم نقلی

- بنابراین:

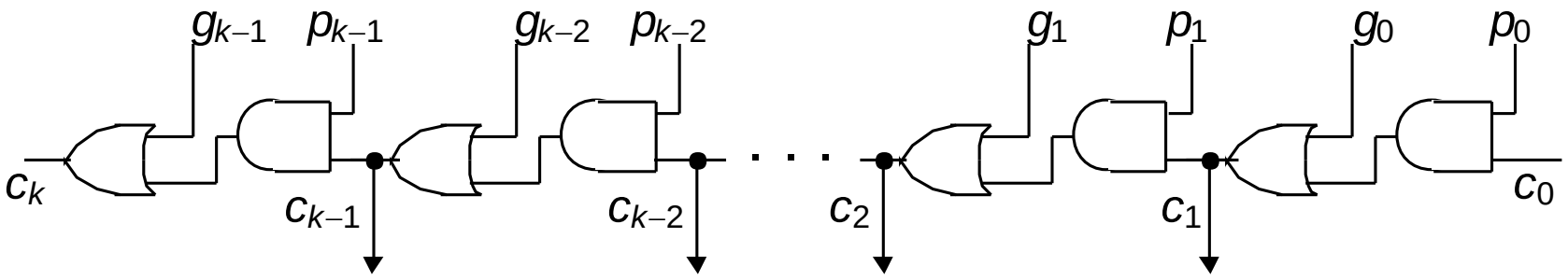
$c_1 = g_0 + p_0 * c_0$

$c_2 = g_1 + p_1 * c_1 = g_1 + p_1 * (g_0 + p_0 * c_0)$

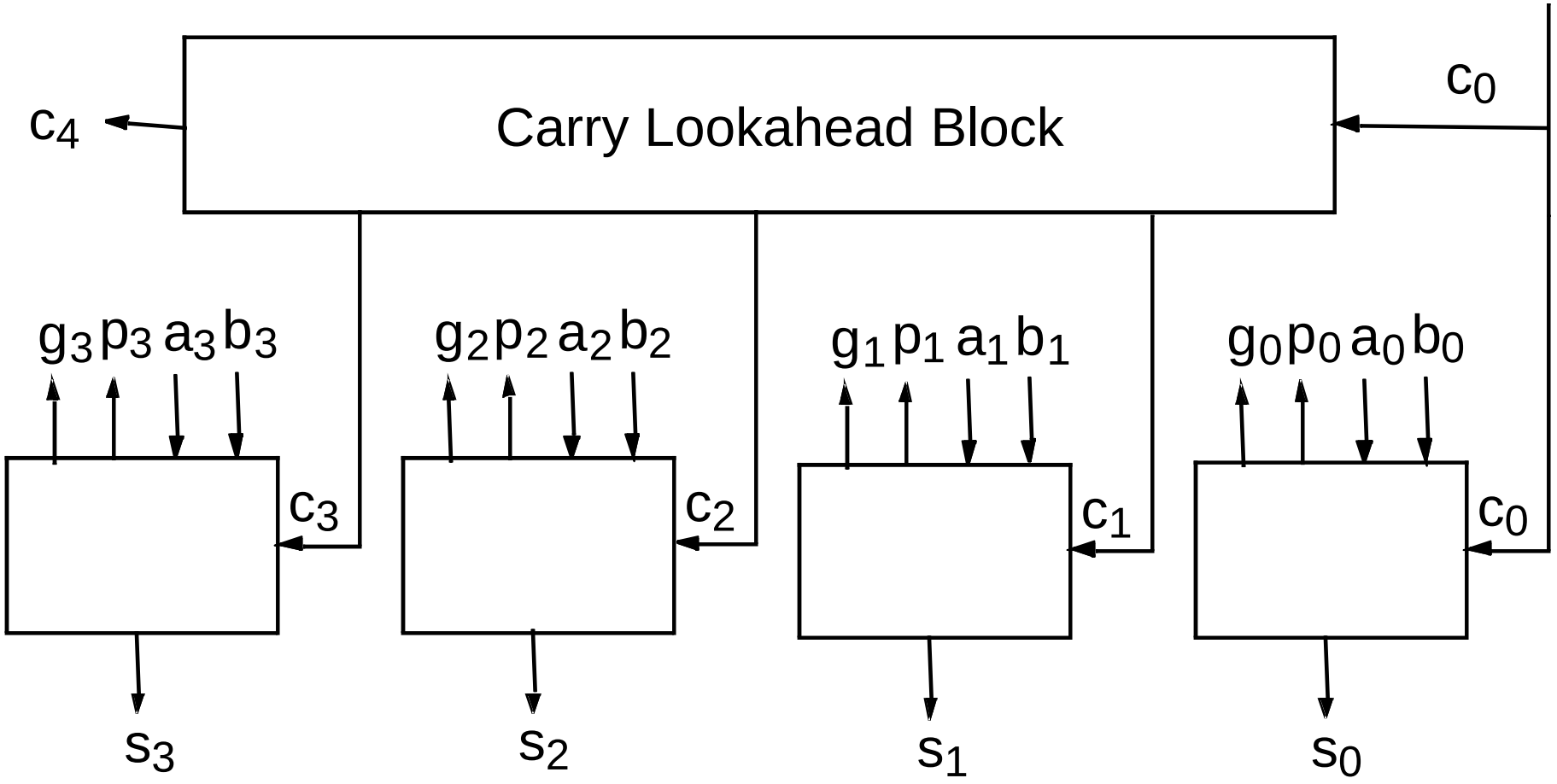
$= g_1 + p_1 * g_0 + p_1 * p_0 * c_0$

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

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



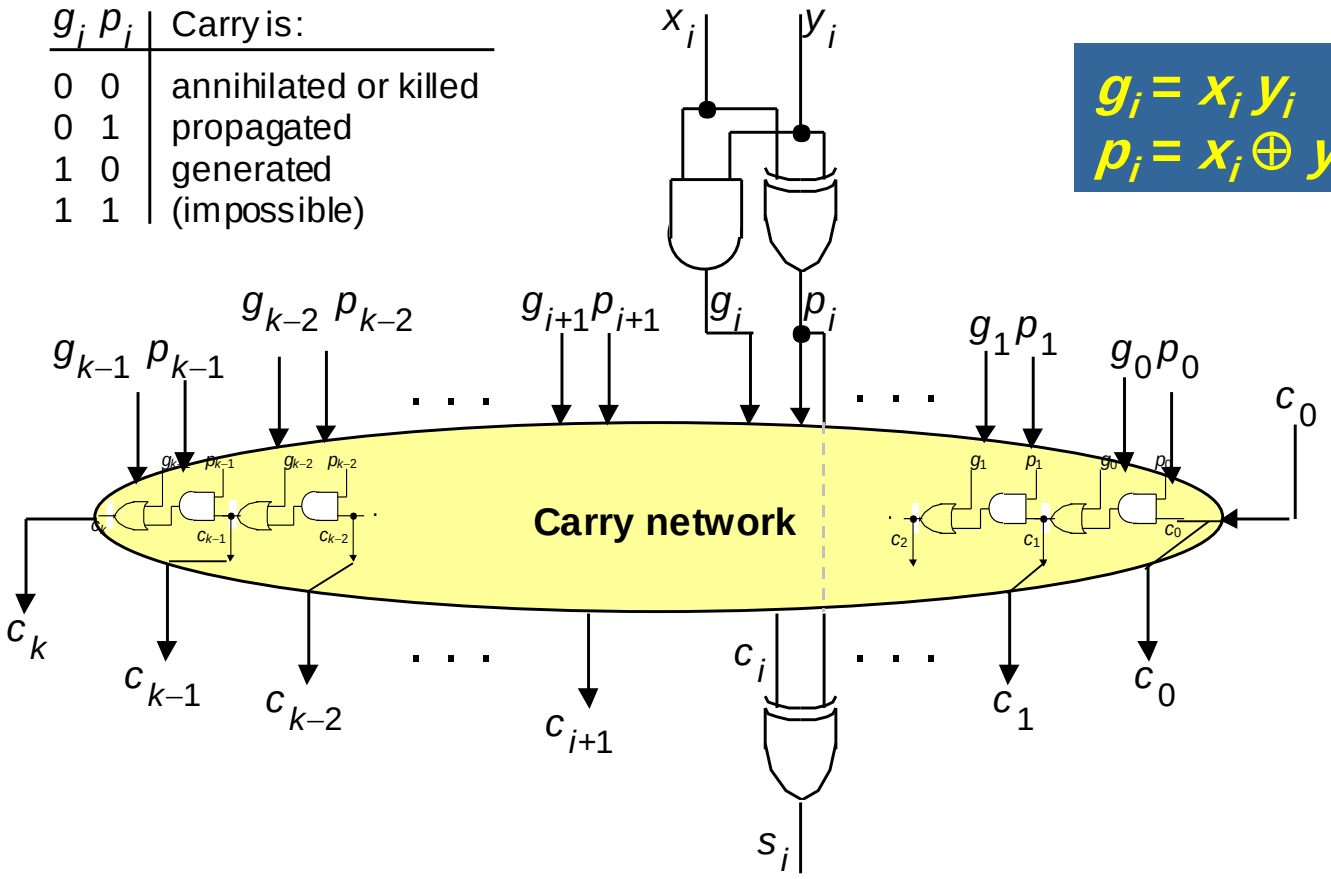
جمع کننده ۴ بیتی



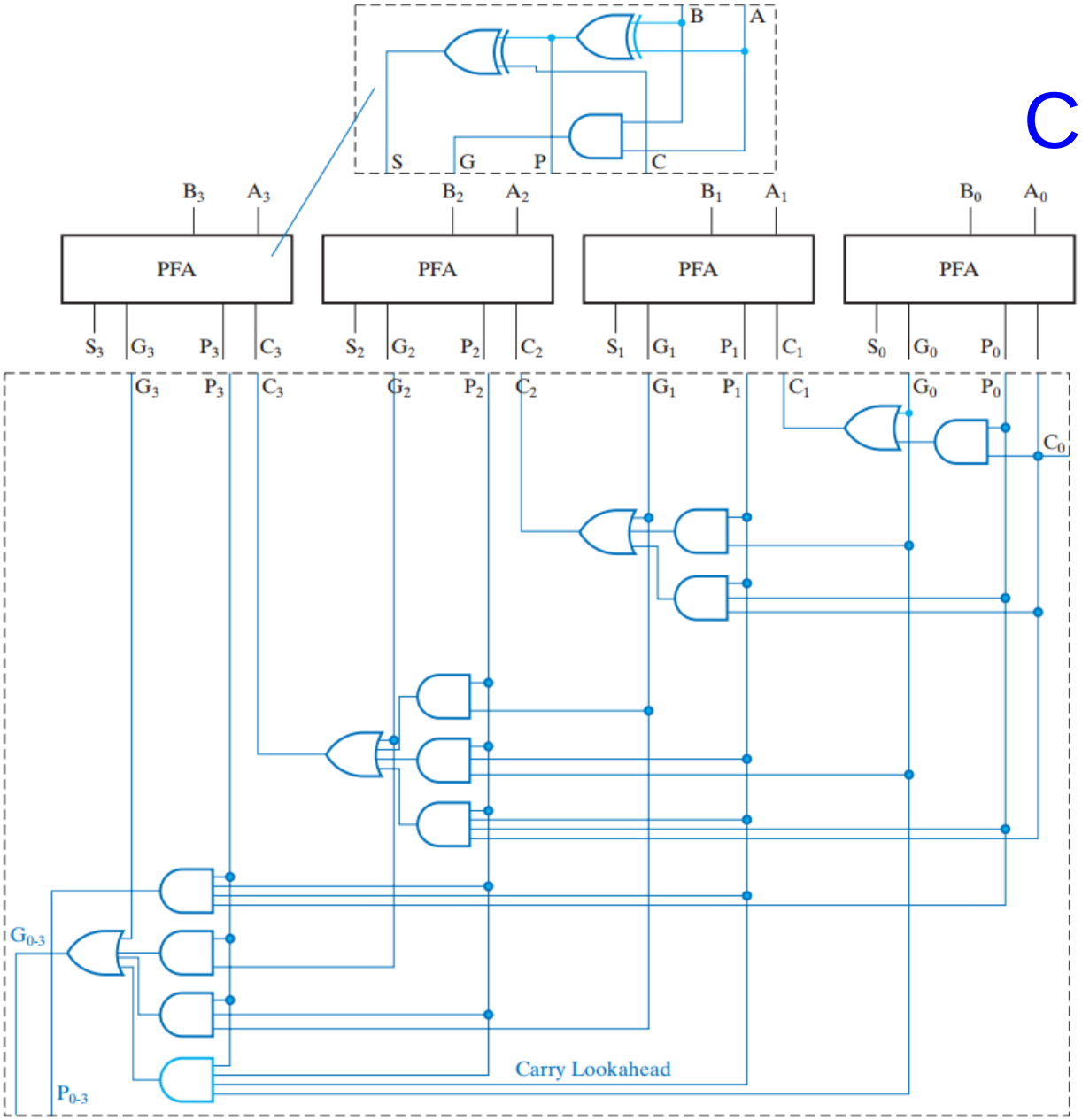
طرح کامل CLA

g_i	p_i	Carry is:
0	0	annihilated or killed
0	1	propagated
1	0	generated
1	1	(impossible)

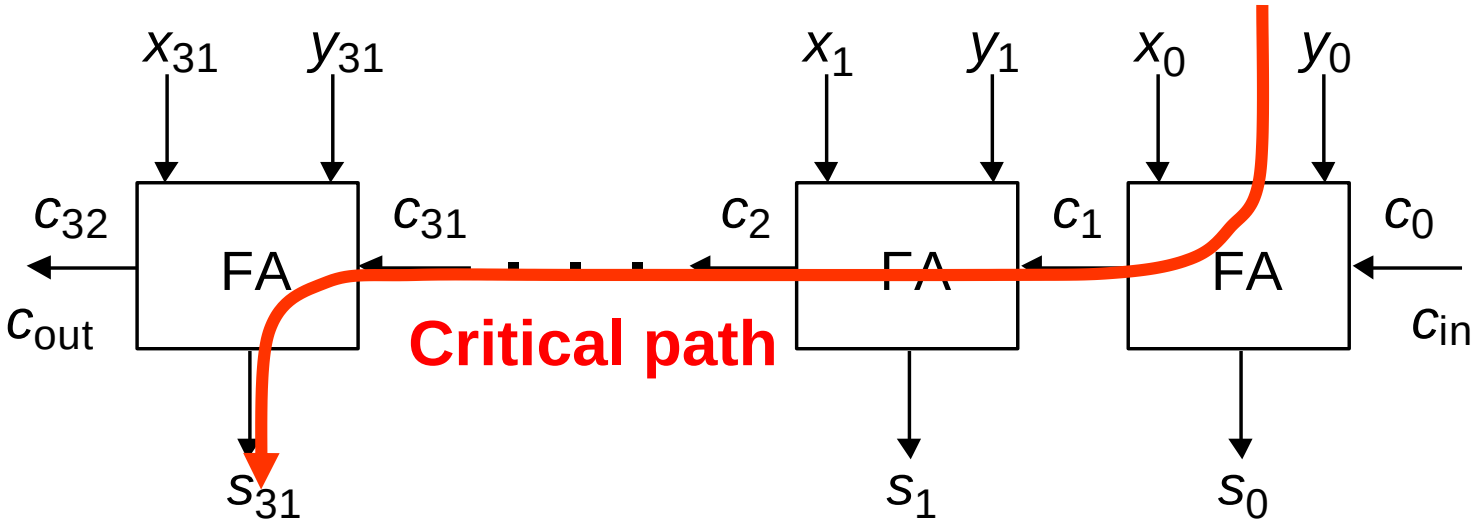
$$g_i = x_i y_i$$
$$p_i = x_i \oplus y_i$$

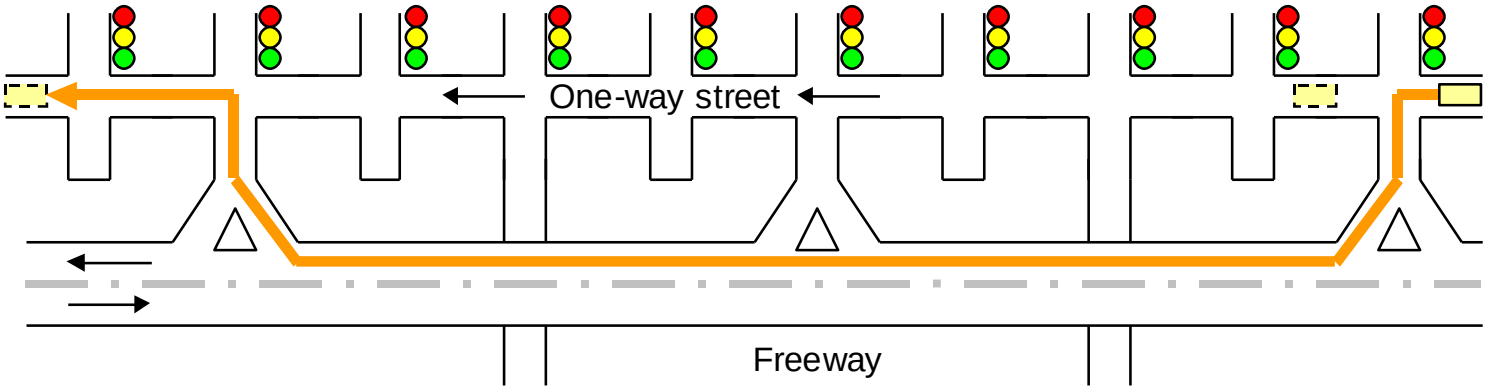


طرح کامل CLA


B
A
S
G
P
C
B_3 A_3
B_2 A_2
B_1 A_1
B_0 A_0
PFA
PFA
PFA
PFA
S_3 G_3 P_3 C_3
S_2 G_2 P_2 C_2
S_1 G_1 P_1 C_1
S_0 G_0 P_0
C_0
G_{0-3}
P_{0-3}
Carry Lookahead

تاثیر پیش بینی رقم نقلی

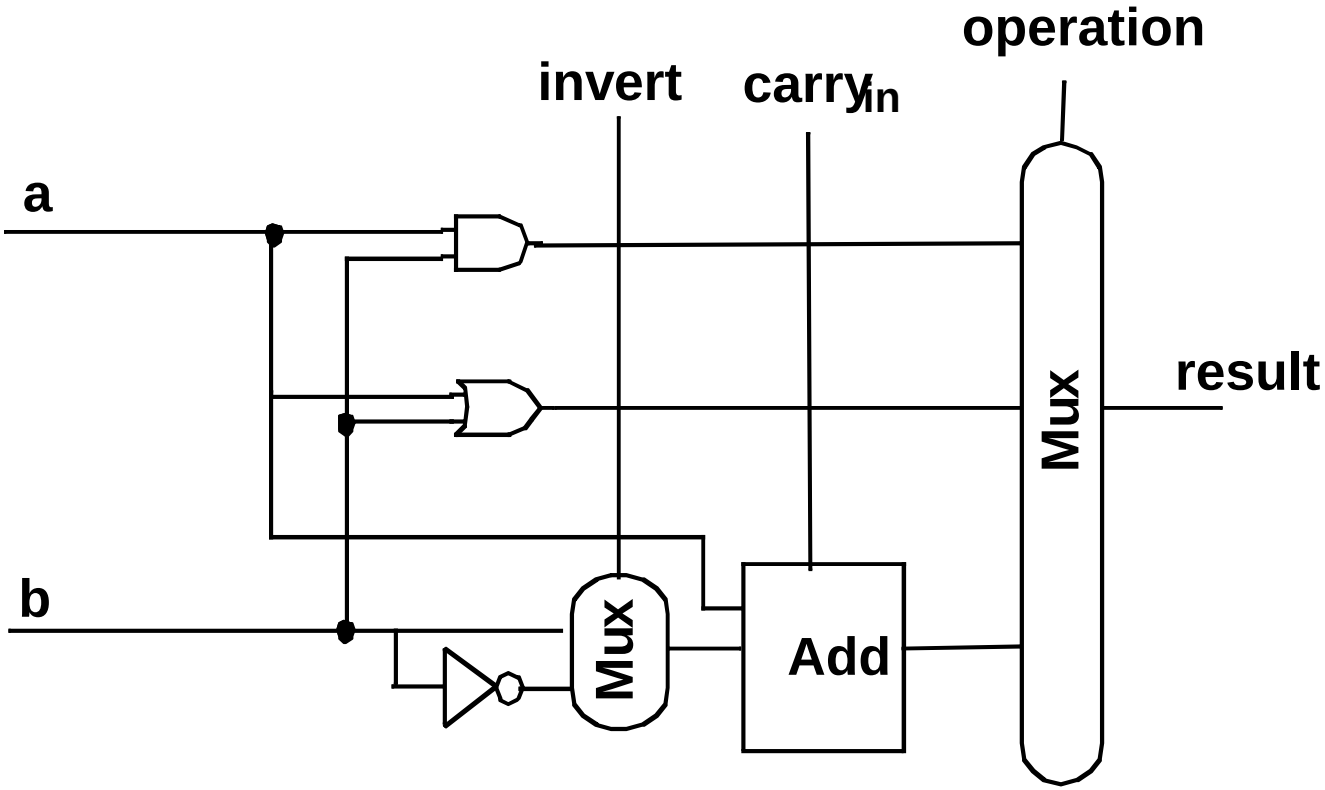




اعمال منطقی

- AND, OR, XOR, NOR بیتی
  - برای پیاده سازی هر کدام به ۳۲ گیت موازی نیاز داریم.
- شیفت و چرخش
  - rol => rotate left (MSB->LSB)
  - ror => rotate right (LSB->MSB)
  - sll -> shift left logical (0->LSB)
  - srl -> shift right logical (0->MSB)
  - sra -> shift right arithmetic (old MSB->new MSB)

واحد محاسبه و منطق

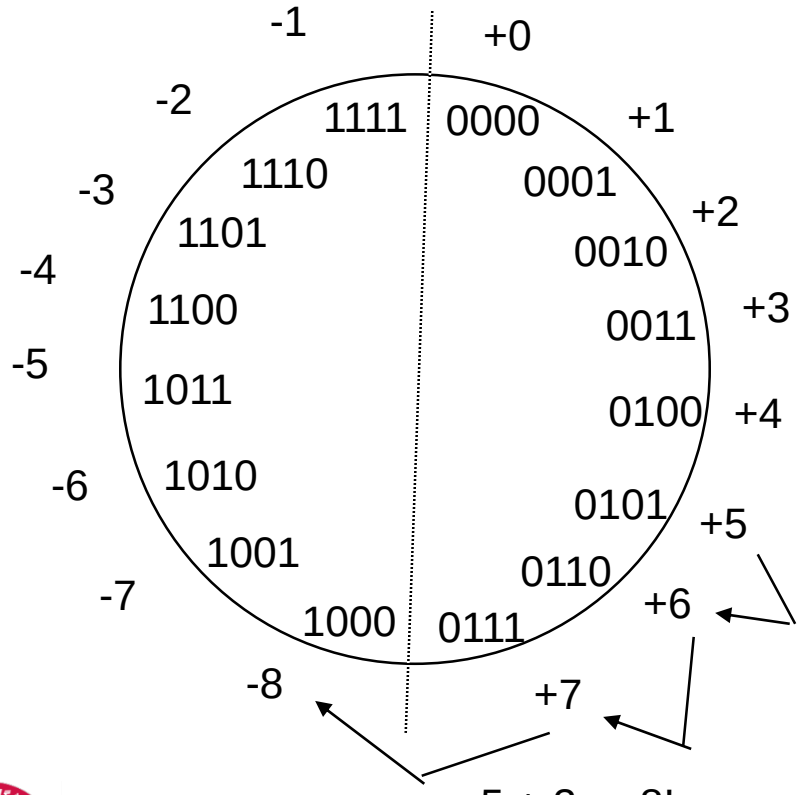


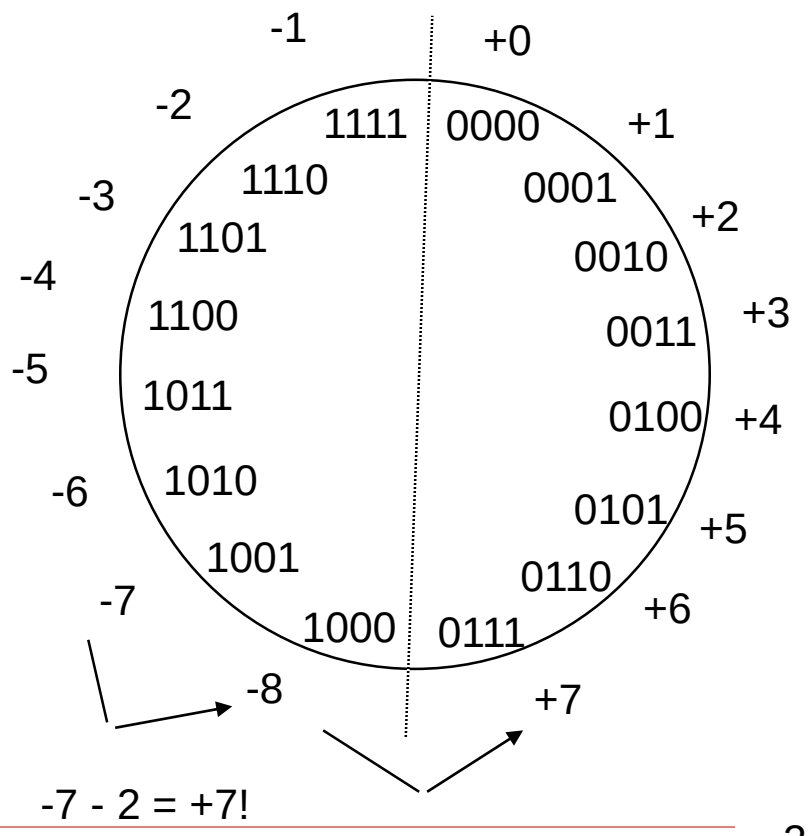
سرریز (Overflow)

How can you tell an overflow occurred?

Add two positive numbers to get a negative number

or two negative numbers to get a positive number





سرریز

	0 1 1 1
5	0 1 0 1
3	0 0 1 1
-8	1 0 0 0

Overflow

	1 0 0 0
-7	1 0 0 1
-2	1 1 0 0
7	1 0 1 1 1

Overflow

	0 0 0 0
5	0 1 0 1
2	0 0 1 0
7	0 1 1 1

No overflow

	1 1 1 1
-3	1 1 0 1
-5	1 0 1 1
-8	1 1 0 0 0

No overflow

Overflow occurs when carry in to sign does not equal carry out

ضرب (دبستان)

			1	0	0	0
		x	1	0	0	1
			1	0	0	0
		0	0	0	0	
	0	0	0	0		
1	0	0	0			
1	0	0	1	0	0	0

- مبنای ده: 8 x 9 = 72
  - PP: 8 + 0 + 0 + 64 = 72
- نتیجه چند بیتی است؟
  - log(n x m) = log(n) + log(m)
  - 32b x 32b = 64b result

ضرب ترکیبی (موریس مانو)

- تولید مضارب جزیی
- استفاده از تسهیم کننده های ۲ به ۱ برای انتخاب یک بیت از مضرب جزیی مورد نظر یا انتخاب 0x0
- ۳۲ مضرب جزیی داریم.

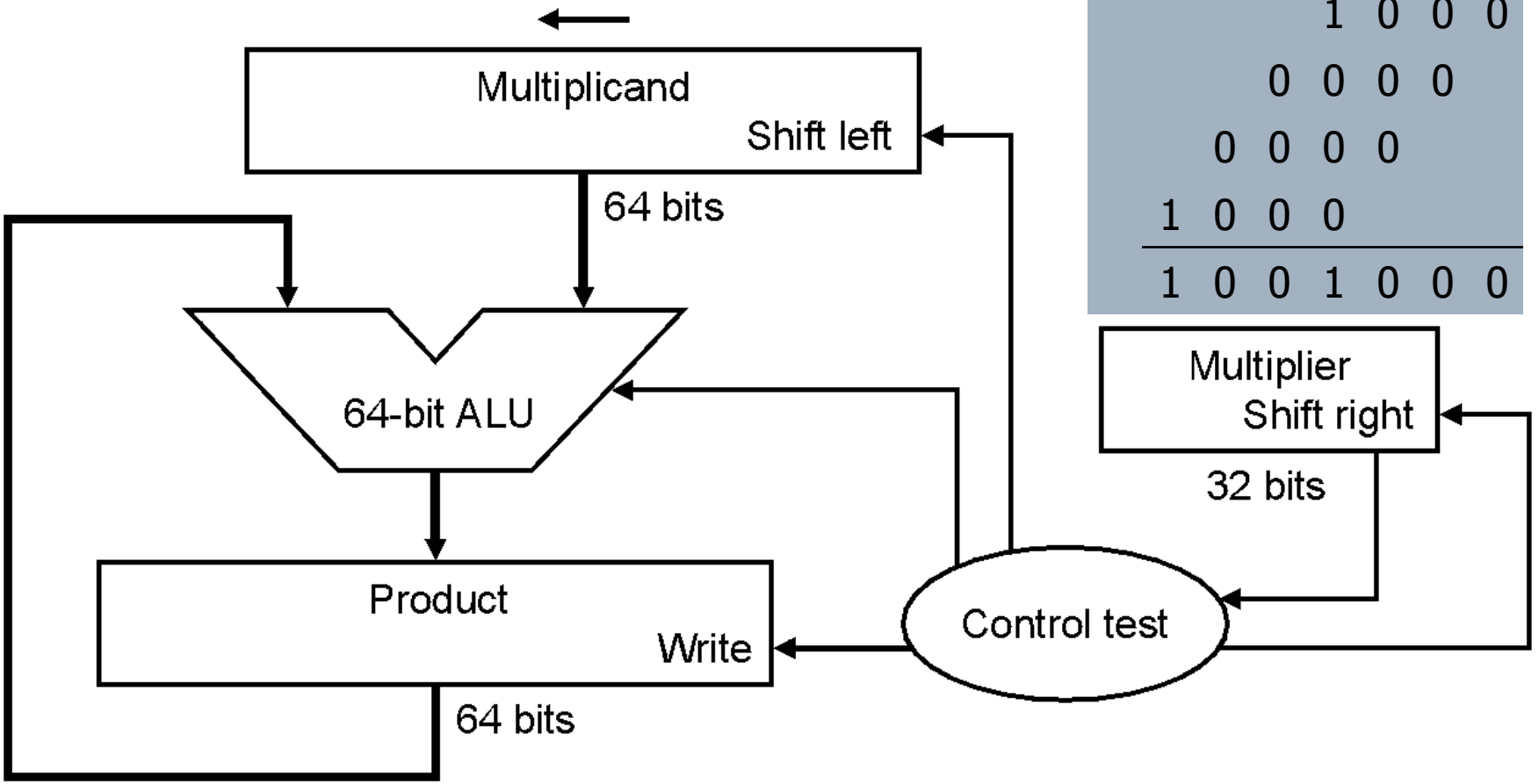
- جمع مضارب جزیی

ضرب کننده چند مرحله ای

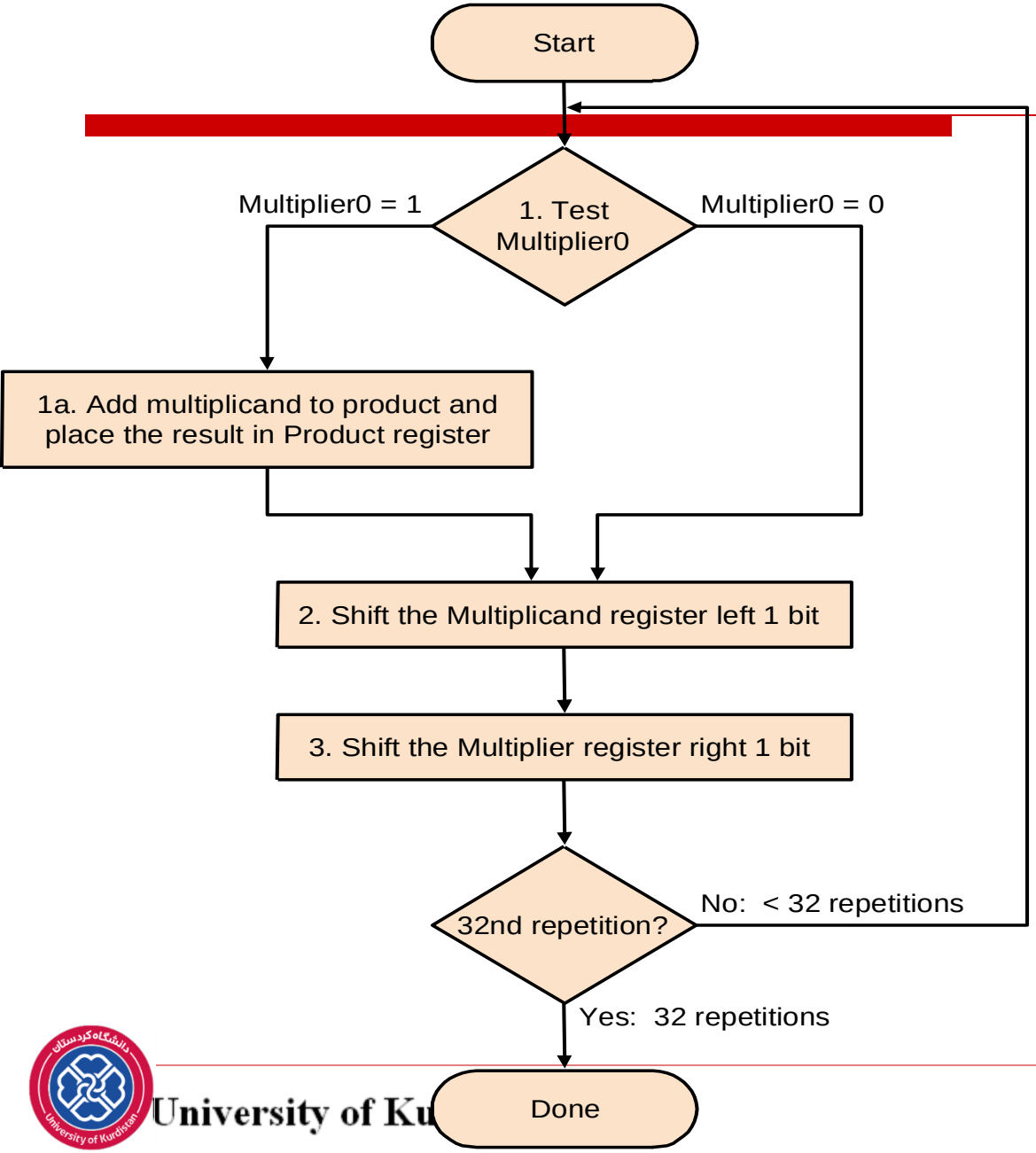
- ضرب کننده ترکیبی
  - مصرف سخت افزار بالا
  - ضرب طبیعی تقریبا پرکاربرد نیست
  - منطقی نیست منابع را اینجا مصرف کنیم.
- ضرب چند مرحله ای
- بیتهای مضروب را چک می کنیم و بر اساس آن (اگر برابر یک بود) شیفت یافته مضروب فیه را با حاصلضرب جمع می کنیم.

ضرب کننده

			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	



الگوریتم ضرب



			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	

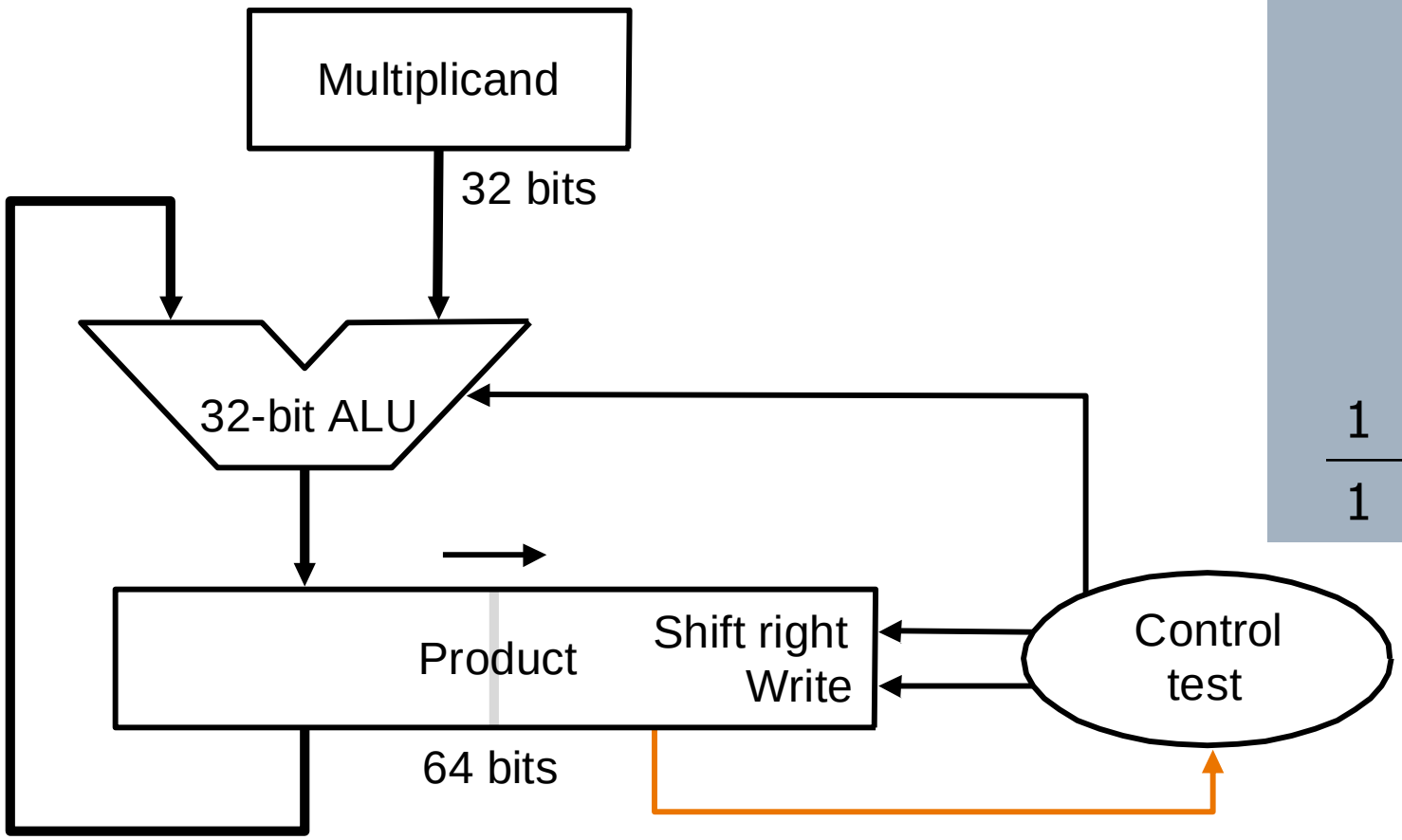
مثال

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

بهبود ضرب کننده

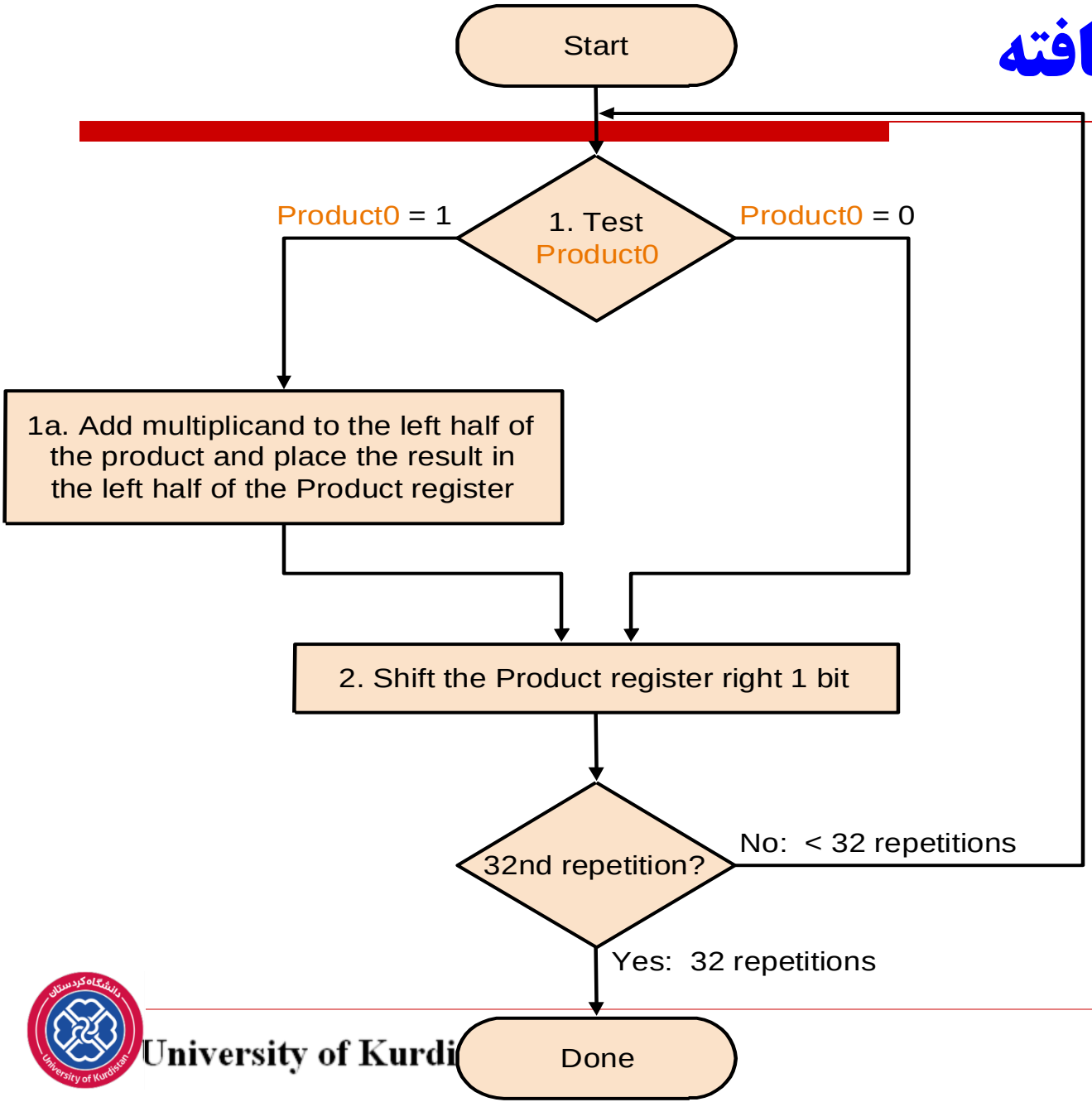
- آیا ما واقعاً به جمع کننده ۶۴ بیتی نیاز داریم؟
- خیر، ما همیشه از ۳۲ بیت استفاده می کنیم.
- لذا از یک جمع کننده ۳۲ بیتی استفاده می کنیم.
  - نتیجه ضرب را در هر مرحله به راست شیفت می دهیم.
- آیا به یک رجیستر مجزا برای مضروب فیه نیاز داریم؟
  - خیر، بیتهای کم ارزش رجیستر ۶۴ بیتی حاصل ضرب در ابتدا استفاده نمی شوند.
  - لذا، مضروب فیه را آنجا ذخیره می کنیم.

مدار ضرب کننده بهبود یافته



			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	

ضرب کننده بهبود یافته



			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	

ضرب اعداد دارای علامت

- یادآوری
  - در ضرب اگر هر دو عدد مثبت یا منفی باشند نتیجه مثبت است.
  - اما اگر علامت دو عدد متفاوت باشد نتیجه منفی است.
  - لذا، بیت علامت با xor علامت دو عدد برابر است.
    - sign(p) = sign(a) xor sign(b)
- بنابر این:
  - هر دو عدد را به دو عدد مثبت n-1 بیتی تبدیل کنید.
  - اعداد مثبت را در هم ضرب کنید.
  - علامت را طبق فرمول فوق محاسبه و نتیجه را طبق آن تنظیم کنید.

کدگذاری بوث

- تکنیک ضرب اعداد در ۹

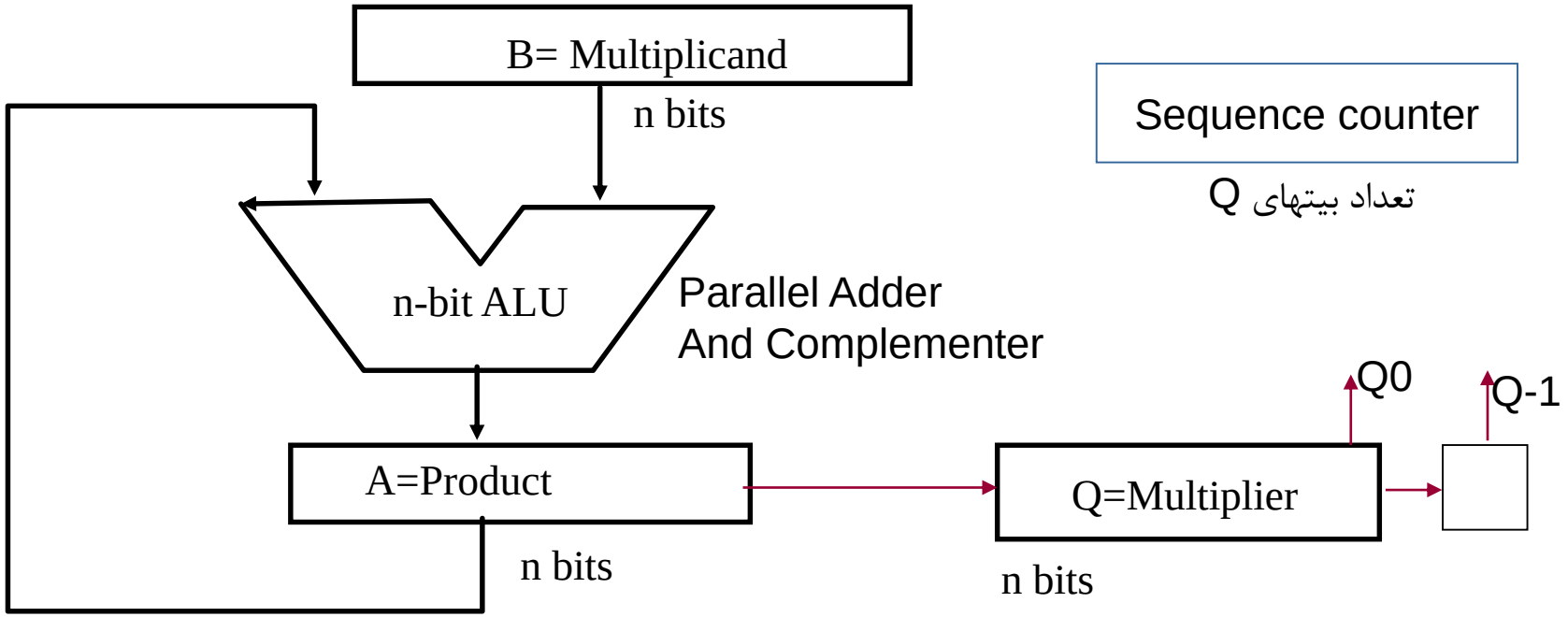
- $123454 \times 9 = 123454 \times (10 - 1) = 1234540 - 123454$
  - به جای انجام ۶ جمع جزیی، یک شیفت و یک تفریق انجام می دهیم

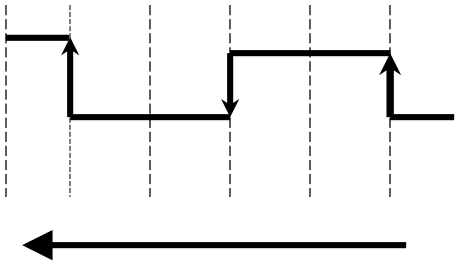
- در الگوریتم بوث از این تکنیک در مبنای دو استفاده می شود.

کدگذاری بوث

- به دنبال تعدادی ۱ پشت سرهم بگردید.
  - مثلا ۰۱۱۰ دو ۱ پشت سر هم دارد.
  - ضرب در ۰۱۱۰ به معنای ضرب در ۸ و تفریق نتیجه از دوبرابر عدد است:
- $6 \times m = (8 - 2) \times m = 8m - 2m$
- لذا از سمت راست شروع کنید:
  - اولین ۱را که دیدید یک تفریق متناسب با محل آن ۱ انجام دهید.
  - به آخرین ۱ که رسیدید یک جمع متناسب با محل بعد از آن ۱انجام دهید.
  - کاری به ۱ های میانی نداشته باشید.

سخت افزار ضرب به روش Booth





Q=10011

نکته:

تعداد اعمال شیفت = تعداد بیتهای Q = ۵

تعداد اعمال جمع: به تعداد لبه های پایین رونده = ۱

تعداد اعمال تفریق: به تعداد لبه های بالا رونده = ۲

مثال

Current bit	Bit to right	Explanation	Example	Operation
1	0	Begins run of '1'	00001111000	Subtract
1	1	Middle of run of '1'	00001111000	Nothing
0	1	End of a run of '1'	00001111000	Add
0	0	Middle of a run of '0'	00001111000	Nothing

Booth Algorithm: Example 1

- $7 \times 3 = 21$

0111	multiplicand	= 7
$\times$0011(0)	multiplier	= 3
11111001	bit-pair 10, add -7 in two's com.	
	bit-pair 11, do nothing	
000111	bit-pair 01, add 7	
	bit-pair 00, do nothing	
00010101	21	

Booth Algorithm: Example 2

- $7 \times (-3) = -21$

0111	multiplicand	= 7
$\times$1101(0)	multiplier	= -3
11111001	bit-pair 10, add -7 in two's com.	
0000111	bit-pair 01, add 7	
111001	bit-pair 10, add -7 in two's com.	
	bit-pair 11, do nothing	
11101011	- 21	

تقسیم اعداد طبیعی

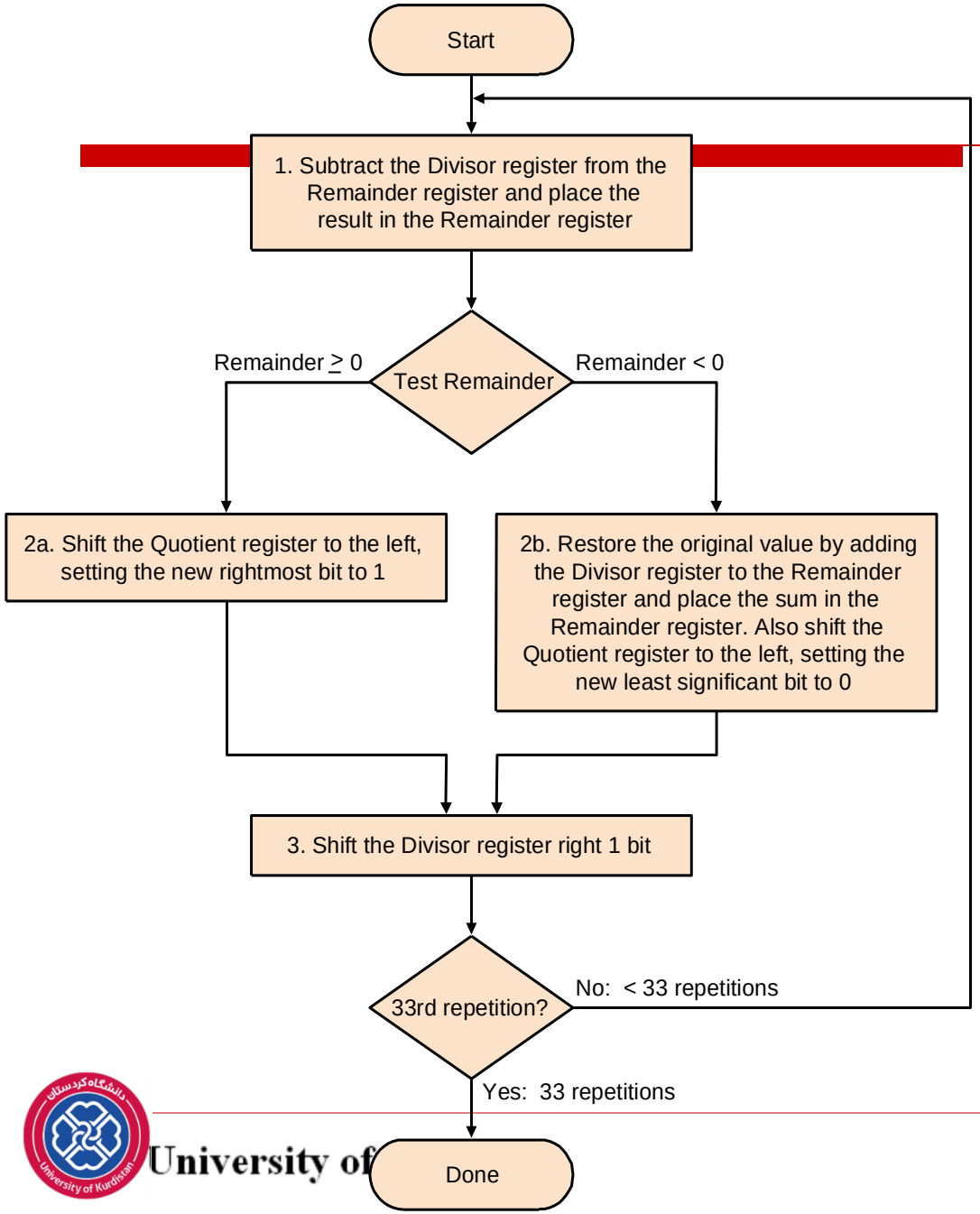
- می خواهیم ۷۴ را بر ۸ تقسیم کنیم:

							1	0	0	1		Quotient
Divisor	1	0	0	0	1	0	0	1	0	1	0	Dividend
				-	1	0	0	0				
							1	0				
							1	0	1			
							1	0	1	0		
						-	1	0	0	0		
									1	0		Remainder

تقسیم اعداد طبیعی

- سخت افزار از کجا می داند که عدد بخش پذیر است یا نه؟
  - شرط: اگر باقیمانده از مقسوم علیه بزرگتر باشد.
  - استفاده از تفریق: $(\text{remainder} - \text{divisor}) \geq 0$
- اگر بخش پذیر بود چکار کنیم؟
  - $\text{Remainder}_{n+1} = \text{Remainder}_n - \text{divisor}$
- اگر بخش پذیر نبود چکار کنیم؟
  - باید مقدار قبلی را بازیابی کنیم.
- Called restoring division

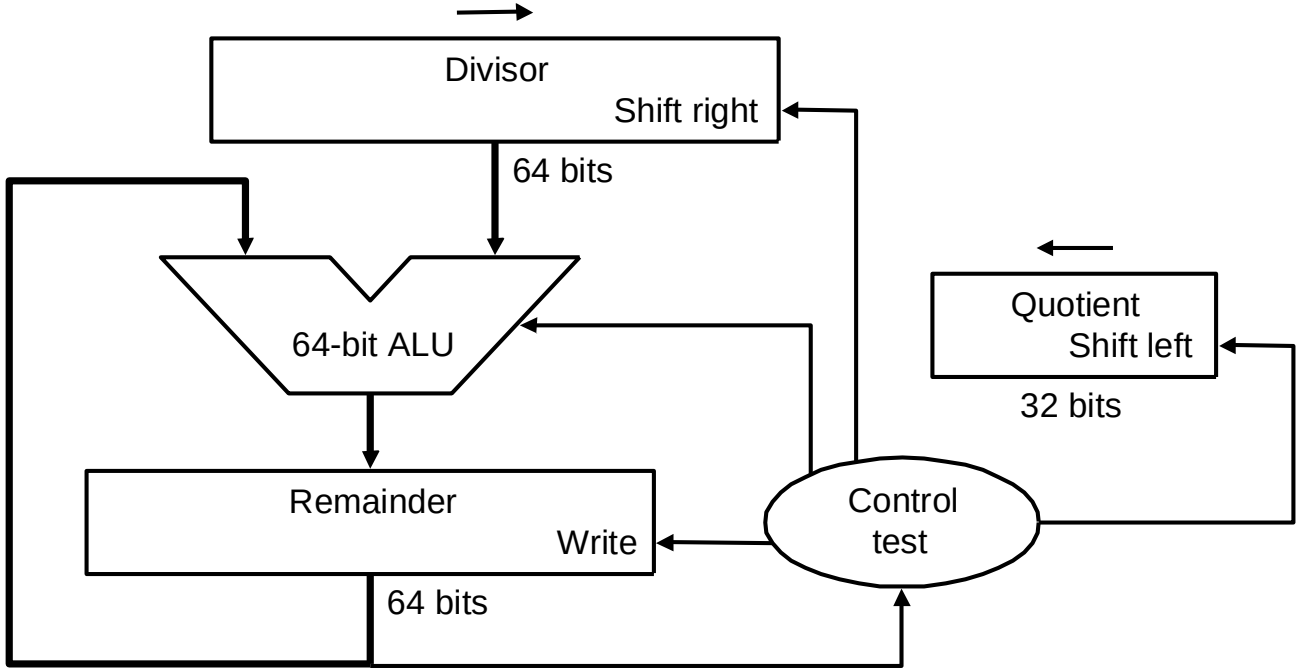
تقسیم اعداد طبیعی



Divisor					1	0	0	1	Quotient
1	0	0	0	1	0	0	1	0	Dividend
			-	1	0	0	0		
					1	0			
					1	0	1		
					1	0	1	0	
				-	1	0	0	0	
							1	0	Remainder

تقسیم اعداد طبیعی

```
                        1 0 0 1   Quotient
Divisor  1 0 0 0 | 1 0 0 1 0 1 0   Dividend
               - 1 0 0 0
                 -------
                    1 0
                    1 0 1
                    1 0 1 0
                  - 1 0 0 0
                    -------
                        1 0   Remainder
```



مثال

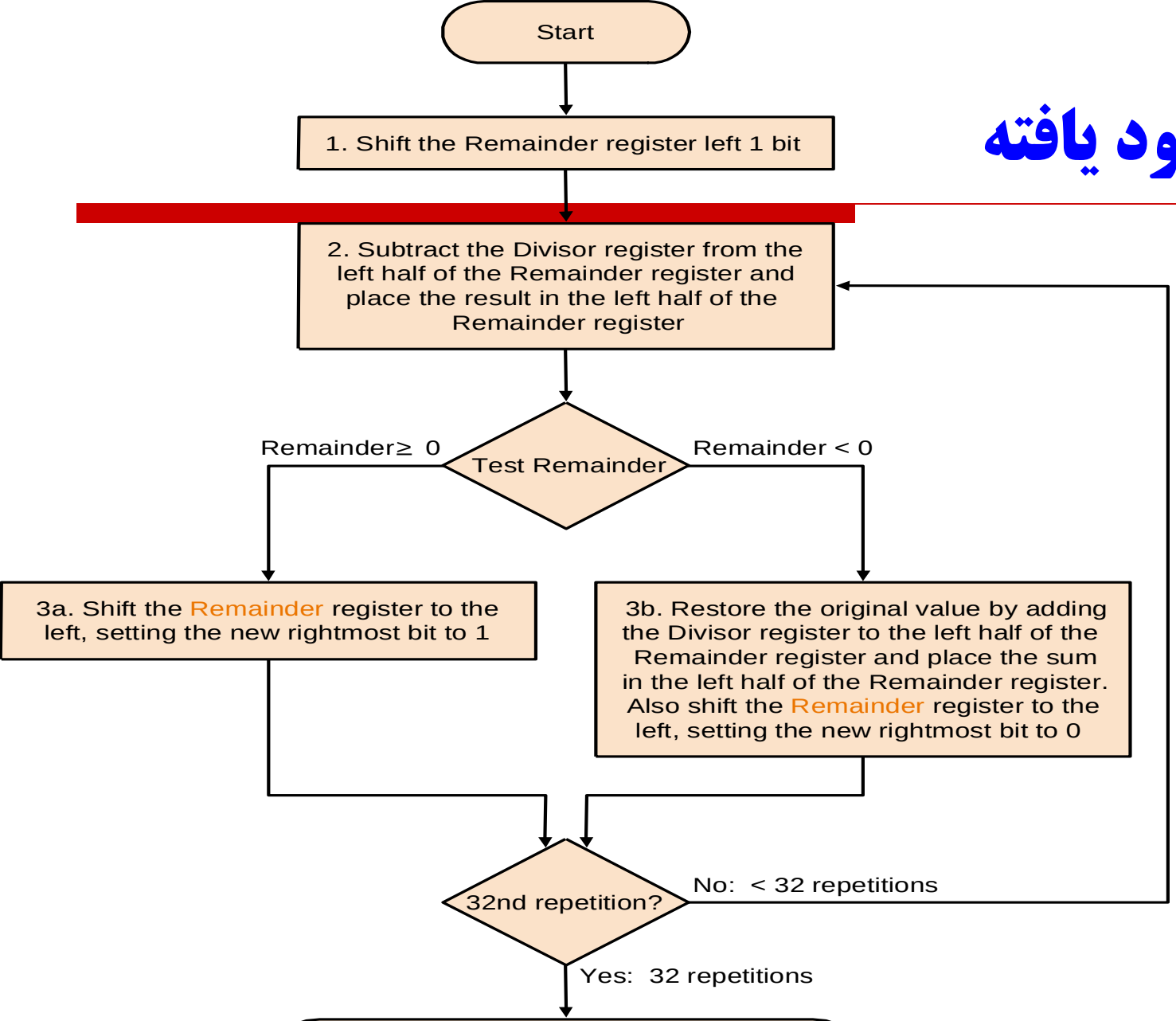
Divisor=0010, dividend= 0111 ➔
quotient = 0011, reminder = 0001

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

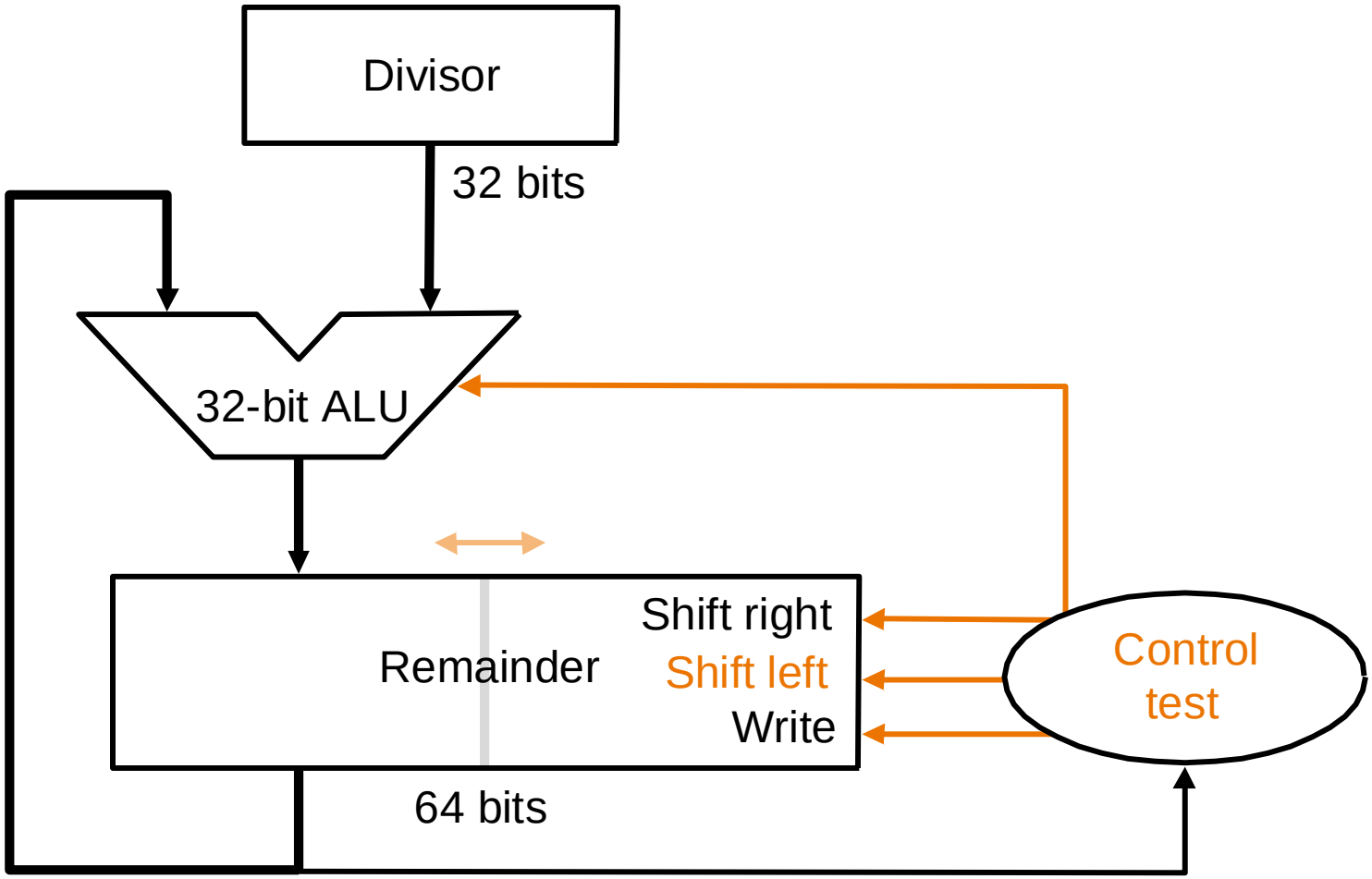
بهبودهای عمل تقسیم

- از تفریق اول صرفنظر کنید.
  - چون در هر صورت نمی توان ۱ را وارد خارج قسمت کرد.
  - لذا اول شیفت می دهیم، بعد کم می کنیم.
    - باید این شیفت اضافی را در مرحله آخر خنثی کنیم.
- می توان از سخت افزار مشابهی استفاده کرد.
  - خارج قسمت را در رجیستر باقیمانده ذخیره می کنیم.
  - از ALU ۳۲ بیتی استفاده می کنیم.
    - به جای شیفت دادن مقسوم علیه به راست، باقیمانده را به چپ شیفت می دهیم.

تقسیم بهبود یافته



تقسیم بهبود یافته



مثال

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0, restore, sll 0	0010	0001 1100
2	2: Rem = Rem - Div	0010	1111 1100
	3b: Rem < 0, restore, sll 0	0010	0011 1000
3	2: Rem = Rem - Div	0010	0001 1000
	3a: Rem > 0 , sll 1	0010	0011 0001
4	Rem = Rem – Div	0010	0001 0001
	3a: Rem > 0 , sll 1	0010	0010 0011
	Shift Rem right by 1	0010	0001 0011

یک بهبود دیگر

- هنوز برای تقسیم، به ازای هر بیت به دو سیکل ALU نیاز داریم:
  - تشخیص امکان بخش پذیری (عمل تفریق)
  - بازیابی (در صورت نیاز)
- می توان به ازای هر بیت یک سیکل صرفه جویی کرد.
- Called non-restoring division
- در این روش اگر آزمایش بخش پذیری ناموفق بود، لازم نیست که بازیابی انجام دهیم.

تقسیم بدون بازیابی

- Restoring: Each time subtraction fails,
  1. the divisor is added
  2. the dividend is shifted left
  3. the divisor is subtracted
- the result equals $2(d + v) - v$
- v can be added *after* shifting, thus replacing the subtraction in the next cycle
- this equals $2d + v$
- both variants deliver identical results, but non-restoring saves one subtraction step!

بجای بازیابی کردن ابتدا شیفت میدهیم و سپس جمع میکنیم. این کار باعث می شود که دیگر نیازی به بازیابی نداشته باشیم.

تقسیم بدون بازیابی

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0 (add next), sll 0	0010	1101 1100
2	2: Rem = Rem + Div	0010	1111 1100
	3b: Rem < 0 (add next), sll 0	0010	1111 1000
3	2: Rem = Rem + Div	0010	0001 1000
	3a: Rem > 0 (sub next), sll 1	0010	0011 0001
4	Rem = Rem – Div	0010	0001 0001
	Rem > 0 (sub next), sll 1	0010	0010 0011
	Shift Rem right by 1	0010	0001 0011

اعداد اعشاری

- برای نمایش عدد اعشاری از تعداد محدودی بیت استفاده می کنیم.

استاندارد IEEE 754:

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

$$x = (-1)^S \times (1 + Fraction) \times 2^{(Exponent - Bias)}$$

اعداد اعشاری

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Representation of Floating Point Numbers

- **IEEE 754 single precision**

Sign **Biased exponent** **Normalized Mantissa (implicit 24th bit = 1)**

$$(-1)^s \times 1.F \times 2^{E-127}$$

Exponent	Mantissa	Object Represented
0	0	0
0	non-zero	denormalized
1-254	anything	FP number
255	0	pm infinity
255	non-zero	NaN

Floating-Point Example

- Represent –0.75 (- 0.11 two)
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 10111111101000…00
- Double: 10111111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  1100000010100000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

توان اعداد اعشاری

- برای نمایش توان آنرا بایاس می کنیم که به آن *excess* می گوییم.
- چرا؟
  - برای ساده سازی عمل مرتب کردن اعداد اعشاری
  - بیت MSB نشان‌دهنده علامت است که باعث تسهیل مرتب سازی می‌گردد.
  - اگر توان مکمل ۲ باشد:
    - اعداد بزرگ توان مثبت دارند.
    - اعداد کوچک توان منفی دارند.
  - چون در مکمل ۲ توان منفی از توان مثبت بزرگتر است، لذا نمی توان اعداد اعشاری را راحت مرتب سازی کرد.

توان بایاس شده

توان	مکمل دو	Excess-127
-127	1000 0001	0000 0000
-126	1000 0010	0000 0001
…	…	…
+127	0111 1111	1111 1110

- Value: $(-1)^S \times F \times 2^{(E-bias)}$
  - SP: bias is 127
  - DP: bias is 1023

نرمال سازی اعداد اعشاری

- نمایش S، E و F به ما اجازه می دهد که برای یک مقدار بیش از چند نمایش داشته باشیم. مثال: $1.0 \times 10^5 = 0.1 \times 10^6 = 10.0 \times 10^4$
  - لذا اعمال مقایسه مشکل می شوند.
  - ترجیح می دهیم که یک نمایش یکسان داشته باشیم.
- بنابراین، برای نرمال سازی یک قرارداد وضع می کنیم.
  - فقط یک رقم در سمت چپ اعشار باشد.
  - در دودویی: آن رقم باید ۱ باشد.
- چون که عدد 1 همیشه به طور ضمنی حضور دارد، نیازی به ذخیره آن نیست.
- لذا یک بیت دقت اضافی داریم که مفت است.

سرریز FP

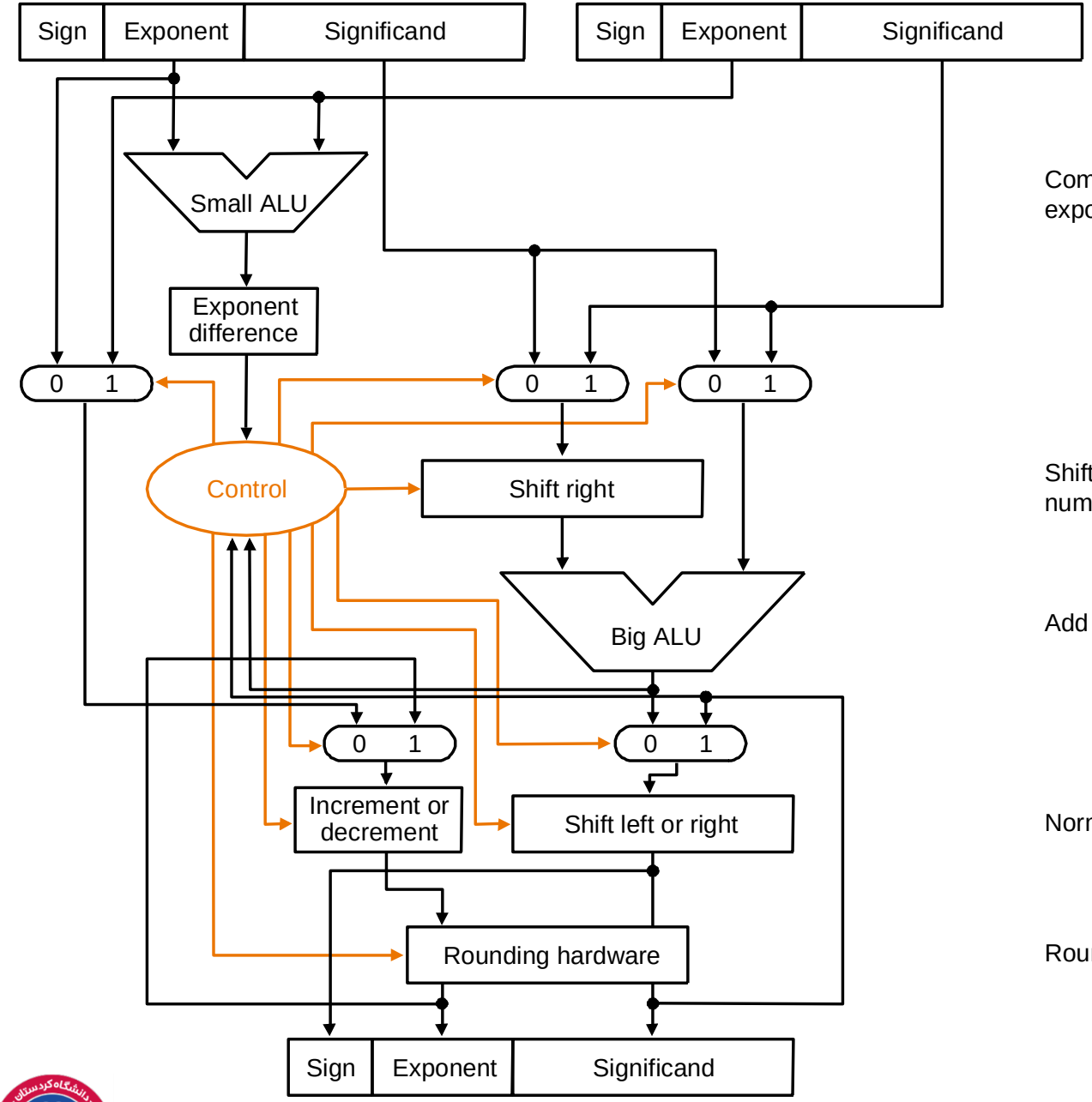
- سرریز FP
  - مثل سرریز اعداد طبیعی
  - عدد خیلی بزرگ باشد که قابل نمایش نباشد.
  - یعنی توان خیلی بزرگ باشد.
- FP Underflow
  - نتیجه خیلی کوچک باشد که قابل نمایش نباشد.
  - یعنی توان خیلی منفی باشد.
- هر دو در استاندارد IEEE754 منجر به تولید استثناء خواهد شد.

جمع اعشاری

- مثل دبستان:
  - توان دو عدد را با شیفت دادن اعشار شبیه هم نمایید.
  - مانتیس ها را جمع کنید.
  - نتیجه را نرمال کنید.
  - در نهایت جمع حاصل را گرد کنید.
  - مثال:

9.997×10^{2}	9.997000×10^{2}
4.631×10^{-1}	0.004631×10^{2}
Sum	10.001631×10^{2}
Normalized	1.0001631×10^{3}
Rounding	1.00016×10^{3}

جمع اعشاری



Compare exponents

Shift smaller number right

Add

Normalize

Round

ضرب اعشاری

- علامت: $P_s = A_s \text{ xor } B_s$
- توان: $P_E = A_E + B_E$
  - بدلیل اینکه توان بایاس شده باید مقدار بایاس را از نتیجه کم کرد:

$e = e1 + e2$

$E = e + 1023 = e1 + e2 + 1023$

$E = (E1 - 1023) + (E2 - 1023) + 1023$

$E = E1 + E2 - 1023$

- مانتیس: $P_F = A_F \times B_F$

- ضرب استاندارد طبیعی (23b or 52b + g/r/s bits)
- استفاده از جمع کننده والاس برای جمع مضارب جزیی

ضرب اعشاری

- محاسبه علامت، مانتیس و توان مثل اسلاید قبلی
- نرمال کردن:
  - شیفت به چپ و شیفت به راست.
    - در شیفت به راست از خارج ۱ وارد می شود. چرا؟
  - کنترل سرریز و underflow
  - گرد کردن
  - نرمال کردن دوباره (در صورت لزوم)


Questions