Disclaimer

The slides presented here are a combination of the CS251 course notes from previous terms, the work of Xiao-Bo Li, and material from the required textbook "Computer Organization and Design, ARM Edition," by David A. Patterson and John L. Hennessy. It is being used here with explicit permission from the authors.

CS251 course policy requires students to delete all course files after the term. Therefore, please do not post these slides to any website or share them.

CS251 - Computer Organization and Design

Data Representation and Manipulation and 1-Bit ALU

Instructor: Zille Huma Kamal

University of Waterloo

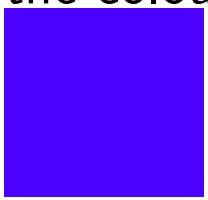
Spring 2023

Data Representation and Manipulation

- How characters and numbers are represented in a typical computer
- Hardware designs which implement arithmetic operations

Data representation

- ARMv8 is a 64-bit architecture,
  1 byte = 8 bits; 4 bytes = 1 word; 8 bytes = 1 double-word
- Bits numbered 63, 62, ..., 0
- Most significant bit (MSB) is bit 63
- Least significant bit (LSB) is bit 0
- In many examples, we will use only 4 bits for illustration purposes
- Sometimes, numbers are written in *hexadecimal*:
  `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F`
- 64-bit binary becomes 16 character hexadecimal
  1 hexadecimal digit is 4 binary digits
- Hexadecimal numbers are useful when addressing memory
- Can also be used in other applications, such as representing pixel color (RGB) specifications. For instance, `0x4c00ff = (76, 0, 255)` resulting in the colour



Characters

- ASCII (American Standard Code for Information Interchange)
- Uses 7 bits to represent 128 different characters
- 8th bit (topmost) used as parity check (error detection)
- 4 characters fit into 32-bit word, 128 possibilities include upper and lower case Roman letters, punctuation marks, some computer control characters

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	123	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.16 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string.

- Unicode: 16 bits per character
  - Many more languages, other than just English

Numbers - Unsigned Binary Numbers

- With 4 bits, can represent 0 through 15

$$\begin{aligned} 1101_2 &= (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) \\ &= 13_{10} \end{aligned}$$

- With 32 bits, can represent 0 through $2^{32} - 1 = 4,294,967,295$
- With 64 bits, 0 through $2^{64} - 1 = 18,446,774,073,709,551,615$
- Useful powers of 2 for CS 251:

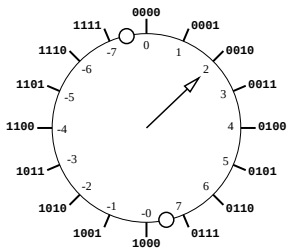
2^5	32
2^{10}	1K
2^{20}	1M
2^{30}	1G

Think About It

How can we represent negative numbers?

Numbers - Signed Binary Numbers

- First idea: use MSB to represent sign of the number, 0 for positive numbers and 1 for negative numebrs.
- This approach is called *sign and magnitude*, may also be referred to as signed magnitude or sign-magnitude
- 4-bit example: 1110 is -6
- With 4 bits, can represent -7 (1111) to $+7$ (0111)
- Problems:
  - two different versions of zero, $+0$ and -0
  - addition operation requires extra steps



- Second (and better) idea: two's complement representation

Signed Binary Numbers - Two's Complement Representation

- Let MSB represent the negative of a power of 2 the smallest number that will be represented in the binary number
- With 4 bits, bit 3 (MSB) represents -2^3
- $1110 = -2^3 \times 1 + 2^2 \times 1 + 2^1 \times 1 + 2^0 \times 0 = -2$
- With 4 bits, we can represent -8 (1000) to $+7$ (0111)
- With 32 bits, can represent $-2,147,483,648$ to 2,147,483,647
- With 64 bits, -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- Usefulness becomes apparent when we have arithmetic operations on signed numbers

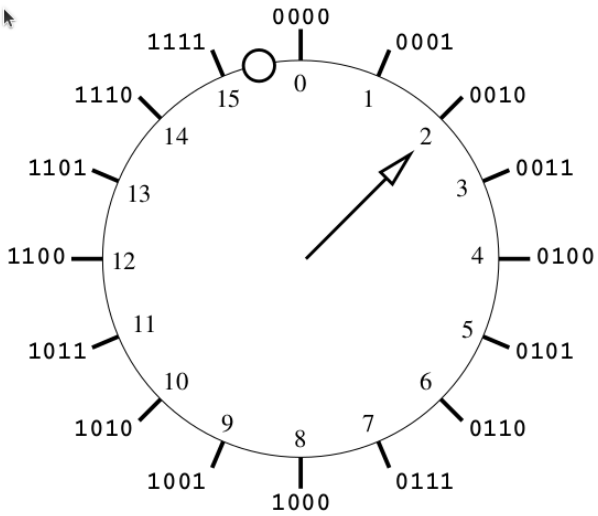
Think About It

Using 2's complement representation of a signed binary number, what is the range of number you can represent in base 10 with an N-bit binary number?

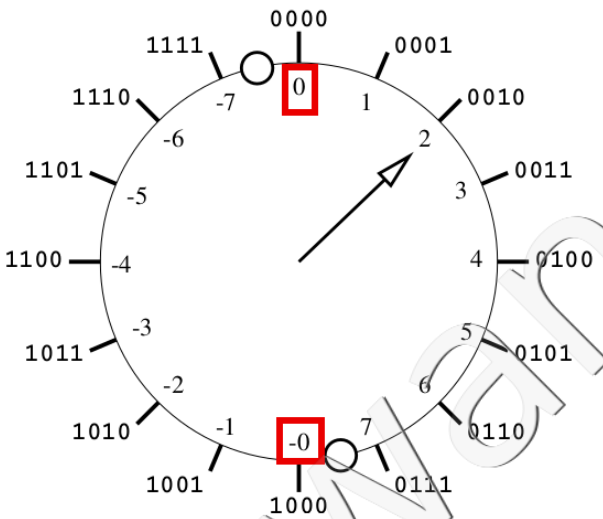
Solution: Range of numbers using 2's complement

- Using 2's complement, an N-bit binary number can represent: $-2^{(N-1)}$ to $+2^{(N-1)} - 1$
- For example, a 4-bit binary number can represent: $-2^{(4-1)}$ to $+2^{(4-1)} - 1 = -2^3$ to $+2^3 - 1 = -8$ to $+7$

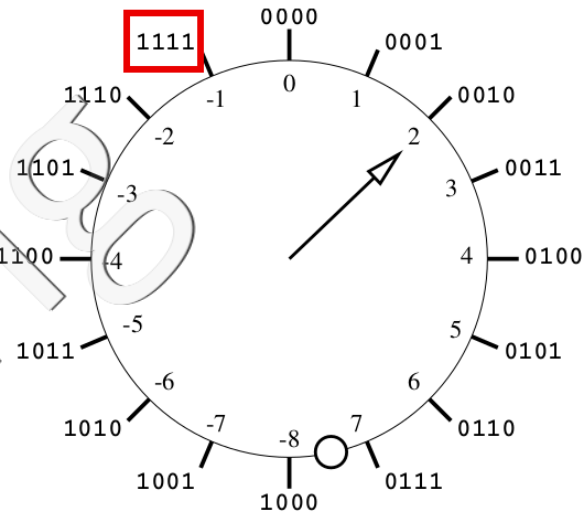
4 Bit Numbers Comparison



Unsigned



Signed Magnitude



Two's Complement

Note:

- 0, ..., 7 all look the same, all positive numbers look the same.
- Signed magnitude has 2 zeros. This increases hardware needed for test for zero.
- Bit pattern with all 1's represents -1,
- For two's complement, $-8 + 1 = -7$ works out nicely.

Representing a Negative Number using 2's Complement

Negating a Two's Complement Number:

- Step 1: Begin with the representation of its absolute value, treat it as an unsigned binary number.
- Step 2: Invert x.
  - Suppose, the bit pattern from Step 1 is x
  - $\bar{x}$ is the result of inverting each bit
  - Example: $x = 0110$, $\bar{x} = 1001$
- Step 3: Add 1.
  - Observe $x + \bar{x} = -1$, then $-x = \bar{x} + 1$

In short, to negate a number in two's complement representation, invert every bit and add 1 to the result.

Sign Extension

- With 4 bits, 0110 is $+6$.

Try this

With 8 bits, what is $+6$?

- With 4 bits, 1010 is -6.

Try this

With 8 bits, what is -6?

- **To expand number of bits used, copy old MSB into new bit positions.**
- Suppose, j is MSB in 4-bit digit and i is MSB in 8-bit digit, then

$$-2^i + 2^{i-1} + 2^{i-2} + \cdots + 2^{j+1} + 2 \cdot 2^j = 0$$

Addition

- To add two two's complement numbers, simply use the "elementary school algorithm", throwing away any carry out of the MSB position
- To subtract, simply negate and add

Think About It

What if the answer cannot be represented in the number of bits used in the hardware?

- This is an **overflow**
- Overflow in addition **cannot** occur if one number is positive and the other negative
- If both addends have same sign but answer has different sign, overflow has occurred

Two's Complement Arithmetic Examples

$3 + 3 = 6$

$$\begin{array}{rcccc} & 0 & 0 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & 0 & 1 & 1 & 0 \end{array}$$

The result is correct, $(0110)_2 = (6)_{10}$.

$6 - 3 = 3$

$$\begin{array}{rcccc} & 0 & 1 & 1 & 0 \\ + & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 1 & 1 \end{array}$$

There is a carry out of 1, which we ignore.

The result is correct, $(0011)_2 = (3)_{10}$.

$6 + 3 = 9$

$$\begin{array}{rcccc} & 0 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 1 \end{array}$$

The result is wrong, $(1001)_2 = (-7)_{10} \neq 9$.

Since sum of two positive numbers is negative we have an **overflow**.

Remember: Overflow Conditions

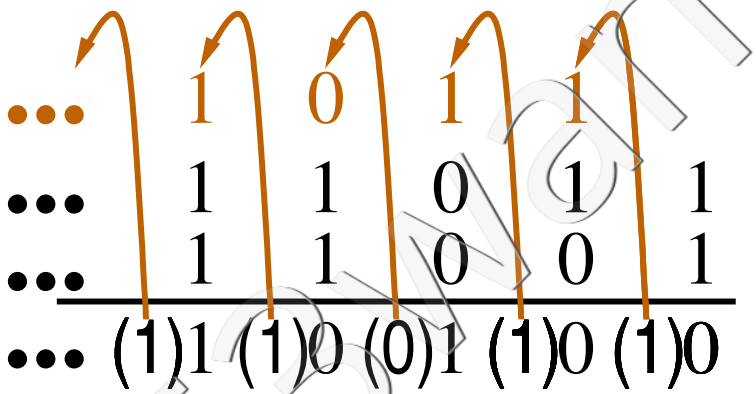
For three numbers in 2's complement and arithmetic addition:
$a_1 + a_2 = a_3$,

- If the sign of a_1, a_2 and a_3 are all the same, there is no overflow.
- The sign of a_1 and a_2 are the same, but a_3 has a different sign, **there is overflow**.
- The sign of a_1 and a_2 are different, we can never have overflow.

Building An Addition Circuit

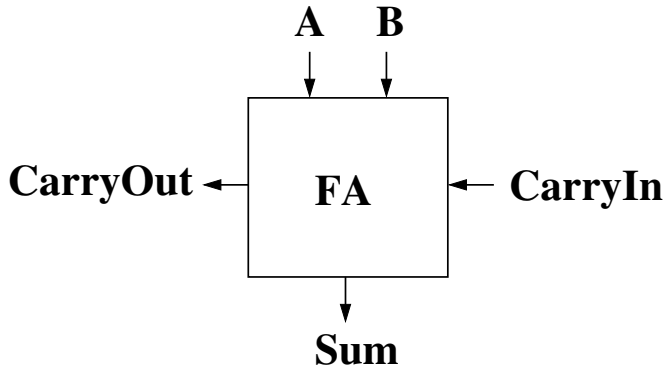
Recall adding one bit has three inputs:

- the two numbers we add, and
- a carry in.



Basic building block is a Full Adder:

- 3 bits as inputs: `A`, `B` and `CarryIn`
- 2 bits as outputs: `CarryOut` and `Sum`



Implementing a Full Adder

Truth table to describe the Full Adder

CarryIn	A	B	CarryOut	Sum
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

$$\begin{aligned} \texttt{CarryOut} &= A \cdot B \\ &+ A \cdot CarryIn \\ &+ B \cdot CarryIn \end{aligned}$$

$$\begin{aligned} Sum &= \overline{CarryIn} \cdot \bar{A} \cdot B \\ &+ \overline{CarryIn} \cdot A \cdot \bar{B} \\ &+ CarryIn \cdot \bar{A} \cdot \bar{B} \\ &+ CarryIn \cdot A \cdot B \end{aligned}$$

Upon closer inspection:

$\texttt{Sum} = A \oplus B \oplus$ CarryIn

Combinational Circuit of Full Adder

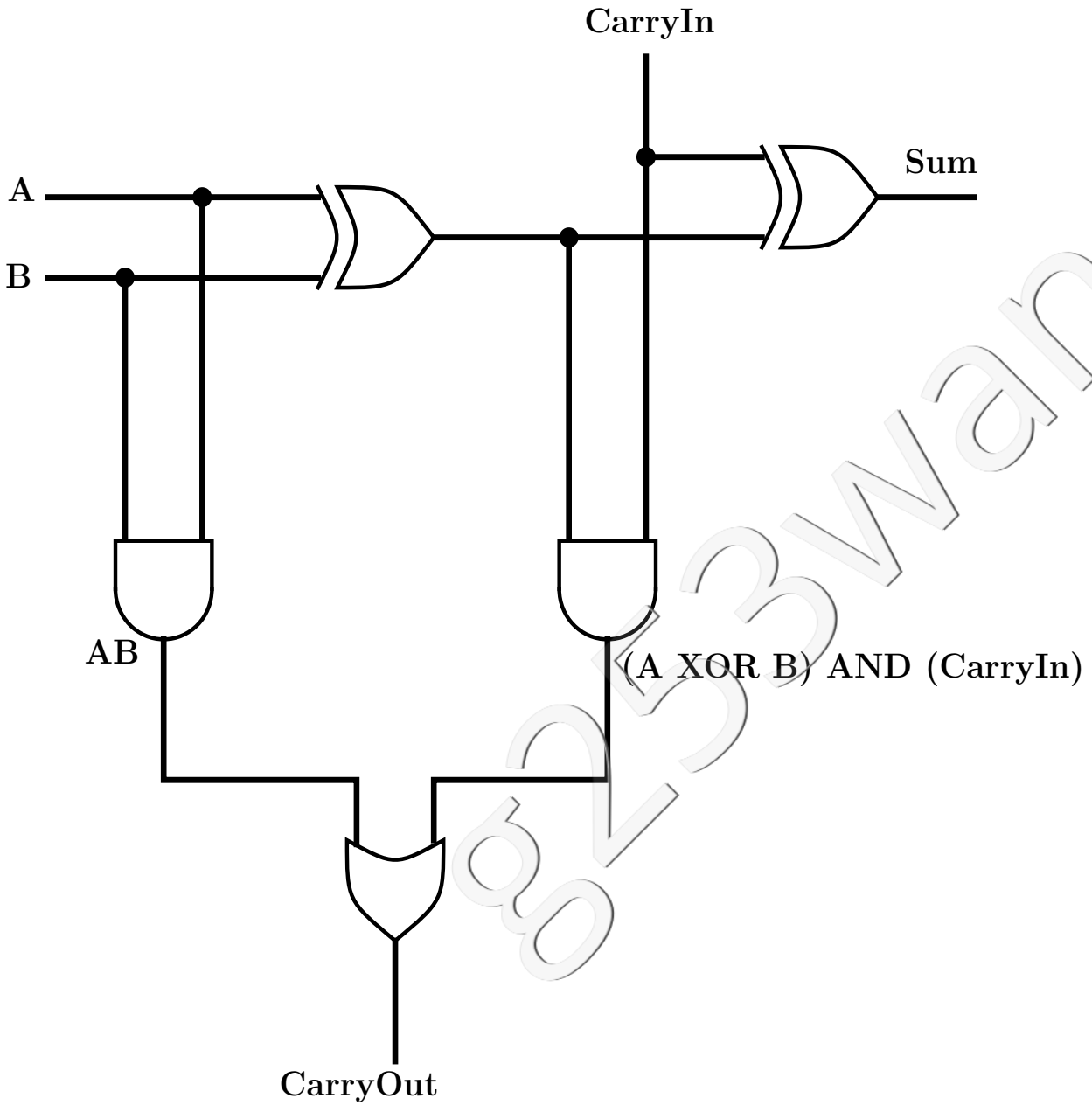
Try this

Can you implement the combinational circuit of a full adder using only AND and OR gates?

Try this

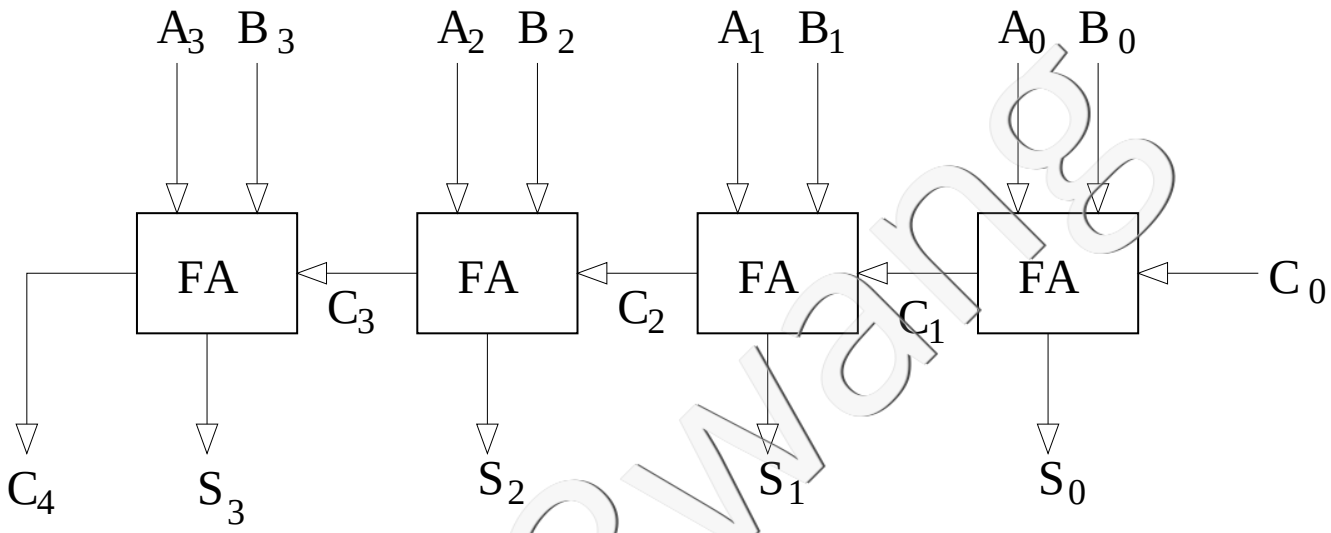
Can you implement the combinational circuit of a full adder using as few logic gates as possible?

Could this circuit be a solution for the full adder?



Ripple-Carry Adder

- 4-bit example



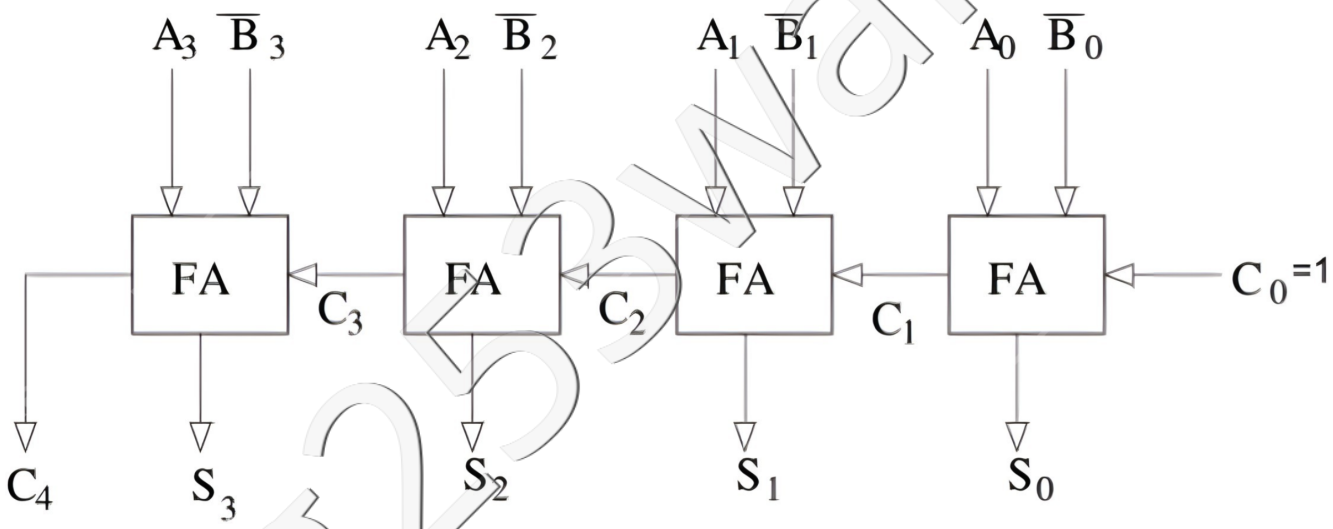
Think About It

What should C_0 be?

- A carry out from the least significant bit $C_1 = 1$ can "ripple" all the way to make $C_4 = 1$.
- Easy to extend to 64 bits.
- Can be slow; "carry-lookahead" idea improves speed (not in the scope of our course)

Ripple-Carry Adder for Subtraction?

Recall, $A - B = A + \bar{B} + 1$.
Therefore, if we can invert every bit in B and set $C_0 = 1$, we can use full adder to accomplish subtraction, such that, $A - B = A + \bar{B} + 1$.



Therefore, for subtraction $C_0 = 1$, otherwise $C_0 = 0$ for addition operation

Logical Operations

Logical Shift

- ARMv8 uses `LSL` (Logic Shift Left) `LSR` (Logic Shift Right) the `shamt` field to specify how many bits to shift.
- shift bits in word left or right, fill in with 0's
- Shifting left i bits is multiplication by 2^i. For e.g.left shift $(0001)_2$ by 1 digit $= 1 \times 2^1 = 2_{10} = (0010)_2$

Arithmetic Shift Right

- ARMv8 uses `ASR`
- shift right, copy sign bit in the empty bits to the left

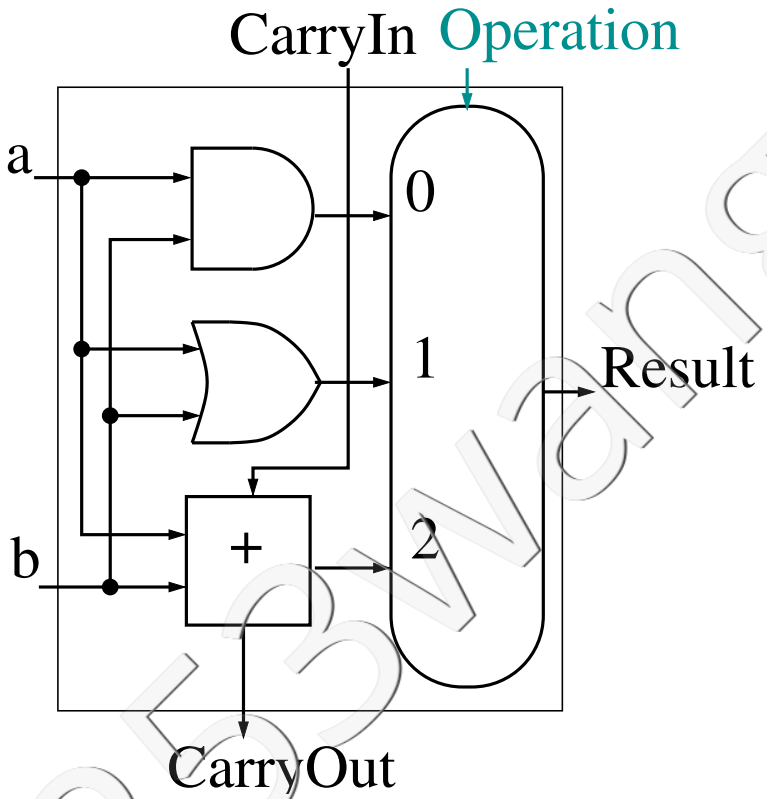
Other shifts include: rotate left or right (moving bit rotated out to other side)

Bitwise AND, OR and EOR. EOR is exclusive OR. ORR is inclusive OR.

Logical Operation	A	B	Result
Bitwise AND	0000 1111	1011 0111	0000 0111
Bitwise OR	1011 1111	0000 0111	1011 1111
Bitwise EOR	1011 0110	1111 1111	0100 1001

Bitwise NOT is just EOR with all-ones

A 1-Bit ALU

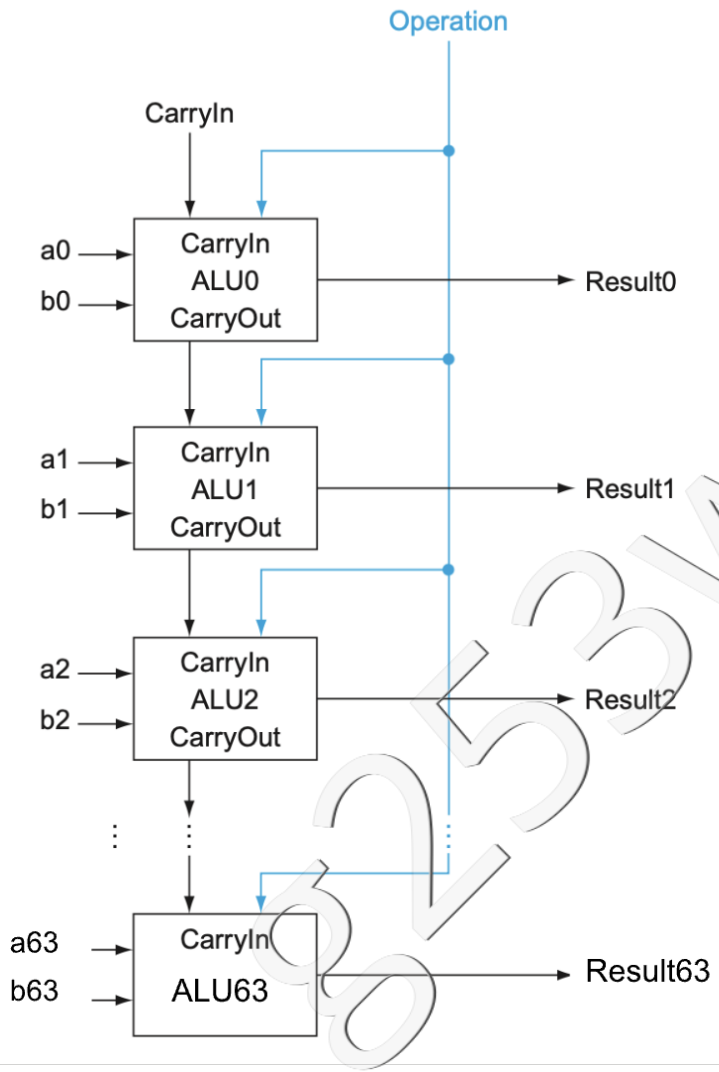


- Extends functionality of full adder
- Performs AND, OR, addition

Think About It

How to build a 64-bit ALU?

A 64-bit ALU from 64 1-Bit ALUs



[1]

Connect 64 1-Bit ALU as with ripple-carry adder to perform 64-bit operations

[1] image modified for correctness

Improving the 1-Bit ALU

- How to implement subtraction, $a - b$?
- $a + \bar{b} + 1$.
- How to achieve this using the 1-Bit ALU:
  - for $\bar{b}$: add input `Binvert` to 1-Bit ALU and set `Binvert` $= 1$
  - to add 1 to $\bar{b}$, set `CarryIn = 1` at the top of the chain when ALU operation is subtraction
  - add $\bar{b} + 1$ to a, set `Operation` $= (10)_2$ to use result of adder by selecting input 2 in MUX to be `Result` of 1-Bit ALU
- Result $= a + \bar{b} + 1$. This is equal to $a - b$

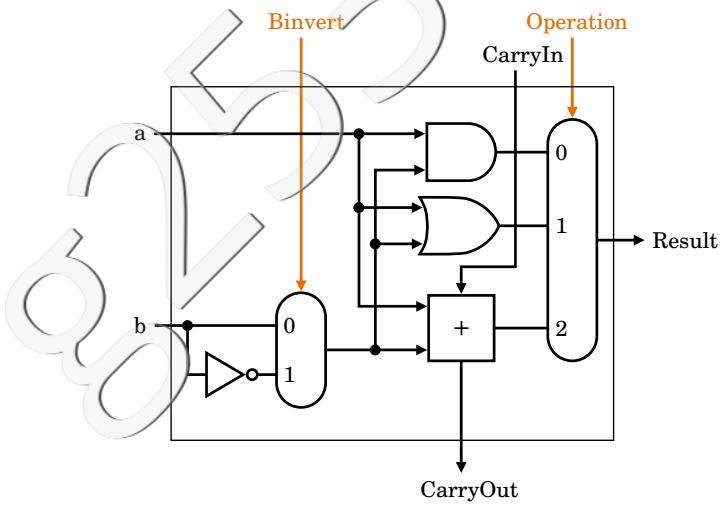


Figure A.5.8 from the text

1-Bit ALU for NOR

Notice: a NOR b $= \overline{a + b} = \bar{a} \cdot \bar{b}$

- Compute $\bar{a}$ with Ainvert=1
- Compute $\bar{b}$ with Binvert=1
- Operation $= (00)_2$
- Result $= \bar{a} \cdot \bar{b}$

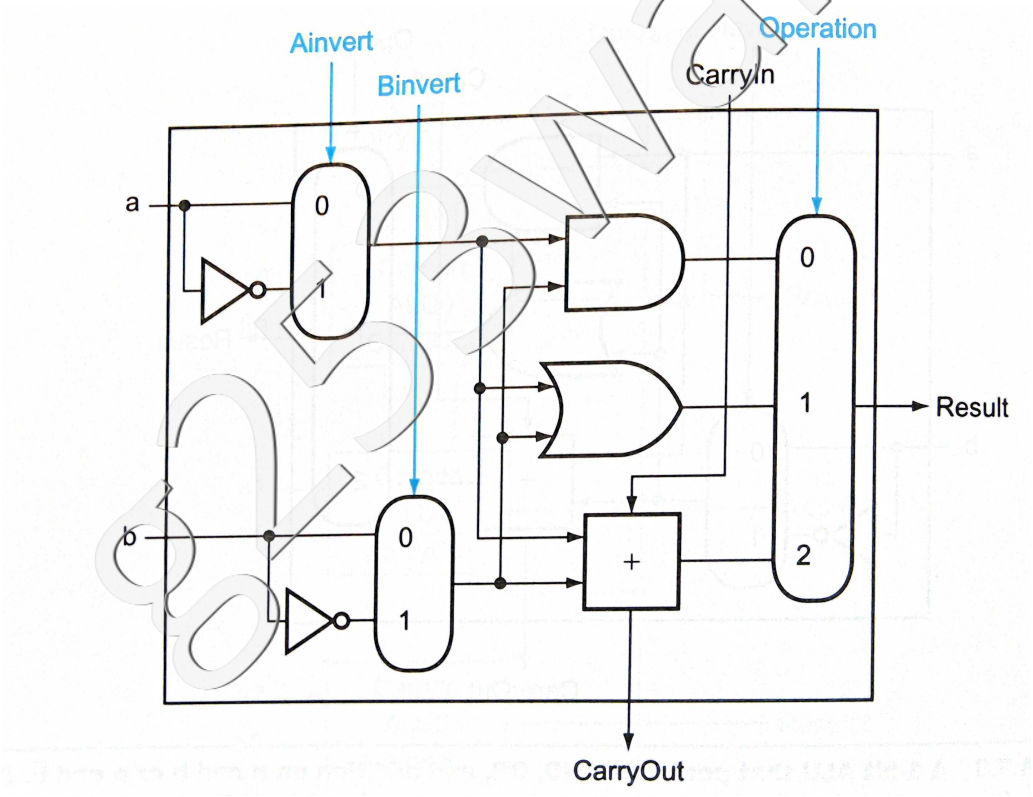
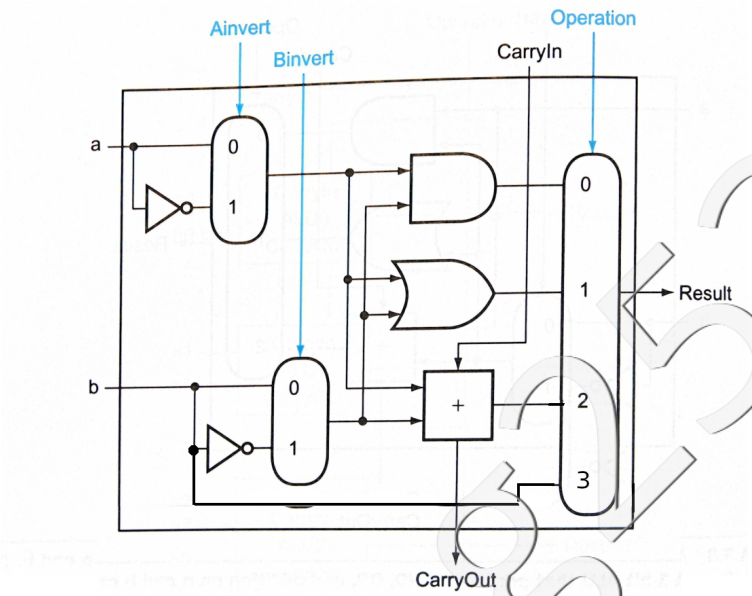


Figure A.5.9 from the text

Pass-B

- Pass-B, no ALU operation is needed in B.
- Set `Operation` $= (11)_2$
- Set `BInvert` $= (0)_2$
- The pass-B operation is needed for CBZ, which tests a single register to see if all 64-bits are zero.



Modified Figure A.5.9 from the text

Think About It

How can we use Pass-B to test for zero?

Solution: Zero Detection Unit

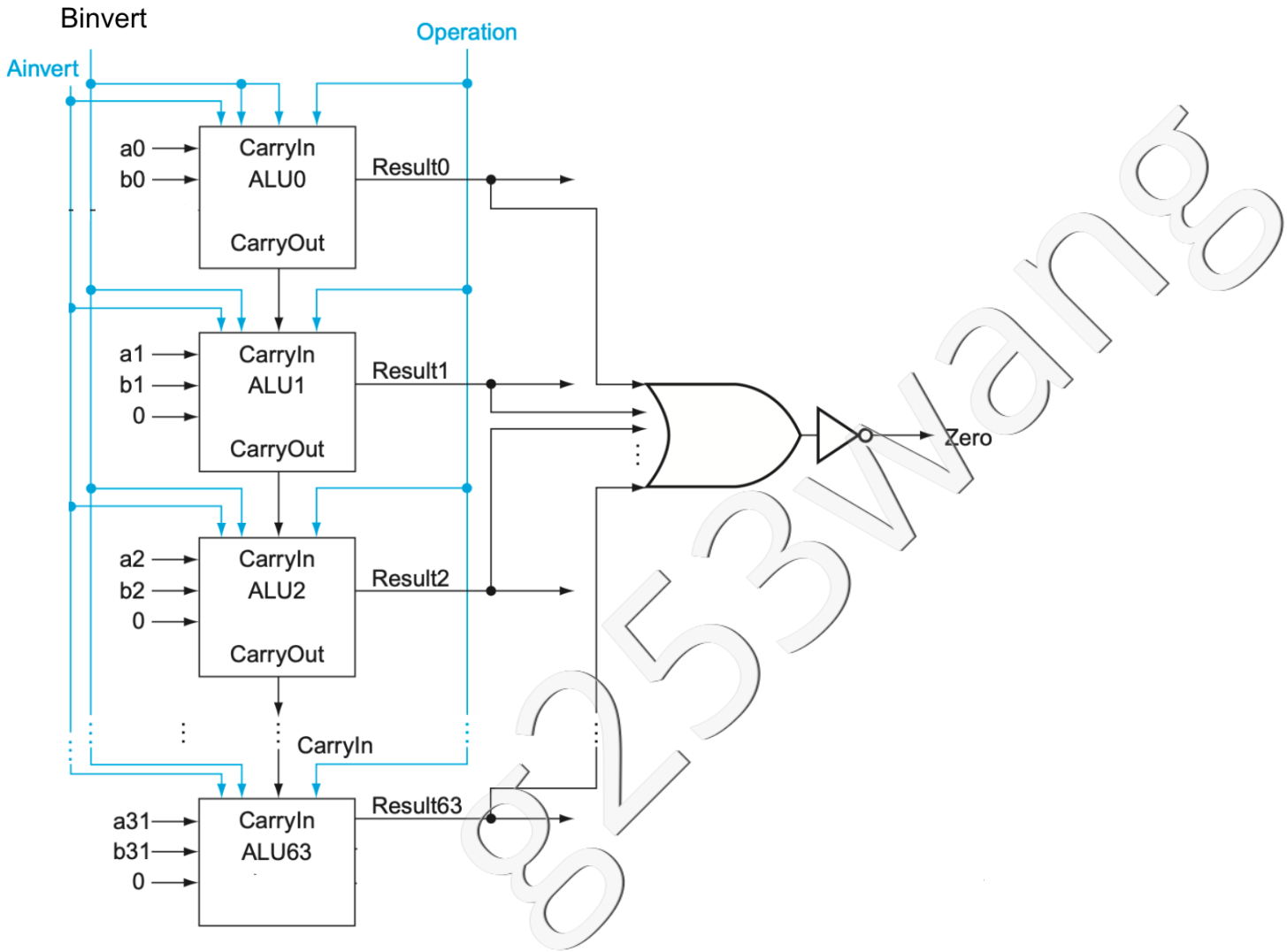
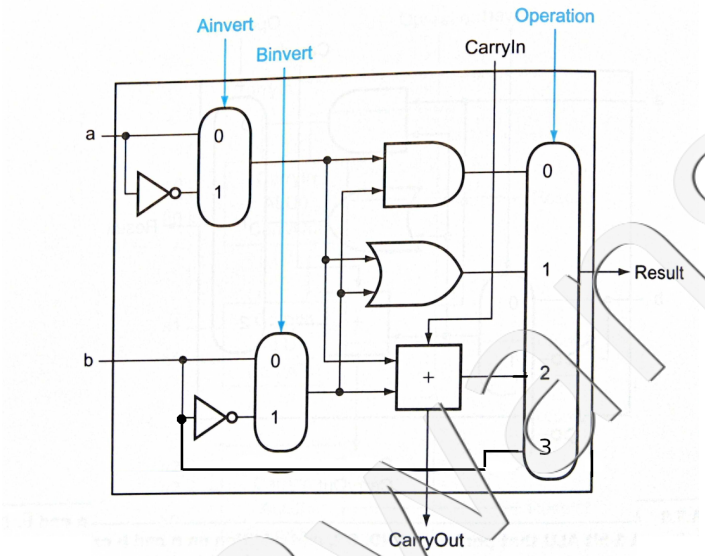


FIGURE A.5.12 The final 64-bit ALU. This adds a Zero detector to Figure A.5.11.

[2]

[2] image modifed for correctness

Operation Table

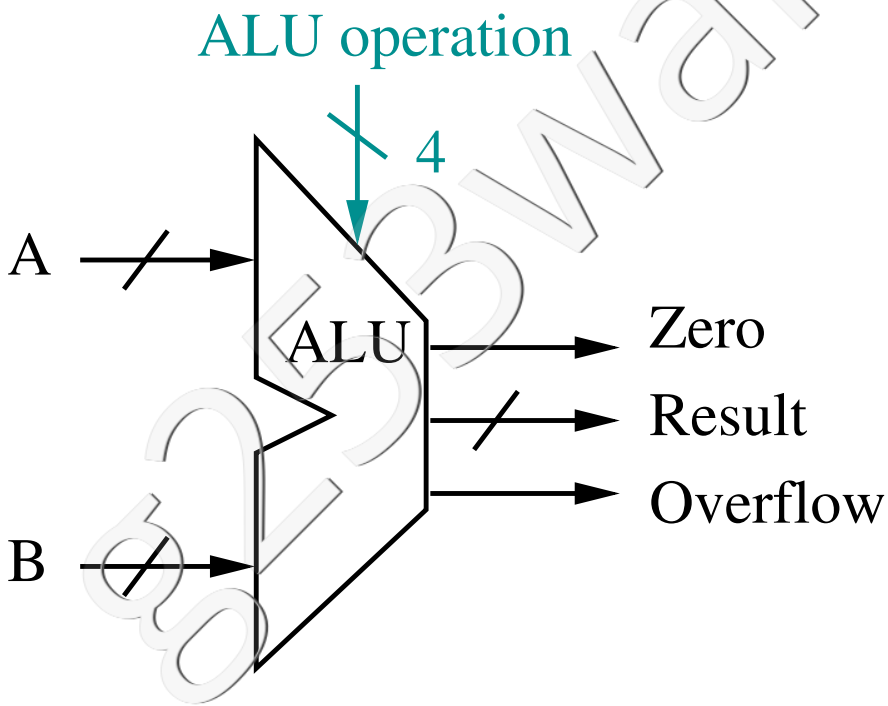


Modified Figure A.5.9 from the text

Ainvert	Binvert CarryIn	Operation	Result
0	0	00	a AND b
0	0	01	a OR b
0	0	10	$a + b$
0	1	10	$a - b = a + \bar{b} + 1$
1	1	00	a NOR b
0	0	11	b (Pass-B)

Abstracting Away ALU Details

- From now on, we use symbol below
- Same shape used for ripple-carry adder, **so remember to label them**



Textbook Readings

- Readings:
  - 2.9 (pages 110 and 111), 2.4, 3.1, 3.2
  - 2.6 (pages 90 and 91)
  - Additional details in Appendix A.5, Zero, Overflow, 64-bit
    Note: Figure A.5.12 wrongly labels pass-B as less
- ignore ARM instructions for now