CSE331 – Computer Arithmetic 1

Unsigned Binary Integers

- Unsigned binary numbers are typically used to represent computer addresses or other values that are guaranteed not to be negative.
- An n-bit unsigned binary integer $A = a_{n-1}\, a_{n-2} \ldots\, a_1\, a_0$ has a value of

$$\sum_{i=0}^{n-1} a_i \cdot 2^i$$

- What is 1011 as an unsigned integer?
- An n-bit unsigned binary integer has a range from 0 to $2^n - 1$.
- What is the value of of the 8-bit unsigned integer 10000001?

Signed Binary Integers

- Signed binary numbers are typically used to represent data that is either positive or negative.
- The most common representation for signed binary integers is the two's complement format.
- An n-bit 2’s comp. binary integer A = $a_{n-1}\, a_{n-2} \ldots a_1\, a_0$ has a value of

$$-a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^{i}$$

- What is 1011 as a 2’s comp. integer?
- An n-bit 2’s comp. binary integer has a range from -2^{n-1} to $2^{n-1} - 1$.
- What is the value of the 2’s comp. Integer 10000001?

Two's Complement Negation

- To negate a two's complement integer, invert all the bits and add a one to the least significant bit.

- What are the two's complements of

$$
\begin{array}{l}
6 = 0110 \Rightarrow 1001 \\
\quad\quad\quad\quad\quad\quad + \quad\;\; 1 \\
\hline
\quad\quad\quad\quad\quad\quad\; 1010 = -6
\end{array}
\qquad\qquad
\begin{array}{l}
-4 = 1100 \Rightarrow 0011 \\
\quad\quad\quad\quad\quad\quad\quad + \quad\;\; 1 \\
\hline
\quad\quad\quad\quad\quad\quad\quad 0100 = 4
\end{array}
$$

- What is the value of the two's complement integer 1111 1111 1111 1101 in decimal?

- What is the value of the unsigned integer 1111 1111 1111 1101 in decimal?

Two's Complement Addition

- To add two's complement numbers, add the corresponding bits of both numbers with carry between bits.

- For example,

$$\begin{array}{r} 3 = 0011 \\ +\ 2 = 0010 \\ \hline 5 = 0101 \end{array} \quad \begin{array}{r} -3 = 1101 \\ +-2 = 1110 \\ \hline -5 = 1011 \end{array} \quad \begin{array}{r} -3 = 1101 \\ +\ 2 = 0010 \\ \hline -1 = 1111 \end{array} \quad \begin{array}{r} 3 = 0011 \\ +\ -2 = 1110 \\ \hline 1 = 0001 \end{array}$$

- Unsigned and two's complement addition are performed exactly the same way, but how they detect overflow differs.

Two's Complement Subtraction

- To subtract two's complement numbers we first negate the second number and then add the corresponding bits of both numbers.

- For example:

$$
\begin{array}{r}
3 = 0011 \\
-\quad 2 = 0010 \\
\hline
\end{array}
\quad \Rightarrow \quad
\begin{array}{r}
3 = 0011 \\
+\ -2 = 1110 \\
\hline
1 = 0001
\end{array}
$$

Overflow

- When adding or subtracting numbers, the sum or difference can go beyond the range of representable numbers.

- This is known as overflow. For example, for two's complement numbers,

```
   5 = 0101          -5 = 1011        5 = 0101        -5 = 1011
 + 6 = 0110        + -6 = 1010     - -6 = 1010     - +6 = 0110
--------------   ---------------  ---------------  ----------------
  -5 = 1011           5 = 0101       -5 = 1011          5 = 0101
```

- Overflow creates an incorrect result that should be detected.

2's Comp – Detecting Overflow

- When adding two's complement numbers, overflow will only occur if
  - the numbers being added have the same sign
  - the sign of the result is different

- If we perform the addition

$$
\begin{array}{r}
a_{n-1}\ a_{n-2}\ \ldots\ a_1\ a_0 \\
+\ b_{n-1}\ b_{n-2}\ \ldots\ b_1\ b_0 \\
\hline
=\ s_{n-1}\ s_{n-2}\ \ldots\ s_1\ s_0
\end{array}
$$

- Overflow can be detected as $V = a_{n-1} \cdot b_{n-1} \cdot \overline{s_{n-1}} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}$

- Overflow can also be detected as $V = c_n \otimes c_{n-1}$, where c_{n-1} and c_n are the carry in and carry out of the most significant bit.

Unsigned – Detecting Overflow

- For unsigned numbers, overflow occurs if there is carry out of the most significant bit. $V = c_n$
- For example,

$$\begin{array}{rl} 1001 & = 9 \\ +\ 1000 & = 8 \\ \hline 0001 & = 1 \end{array}$$

- With the MIPS architecture
  - Overflow exceptions occur for two's complement arithmetic
    - add, sub, addi
  - Overflow exceptions do not occur for unsigned arithmetic
    - addu, subu, addiu

Shift Operations

- The MIPS architecture defines various shift operations:

  (a) sll r1, r2, 3   r2 = 10101100   (shift left logical)
  r1 = 01100000
  – shift in zeros to the least significant bits

  (b) srl r1, r2, 3   r2 = 10101100   (shift right logical)
  r1 = 00010101
  – shift in zeros to the most significant bits

  (c) sra r1, r2, 3   r2 = 10101100   (shift right arithmetic)
  r1 = 11110101
  – copy the sign bit to the most significant bits

- There are also versions of these instructions that take three register operands.

Logical Operations

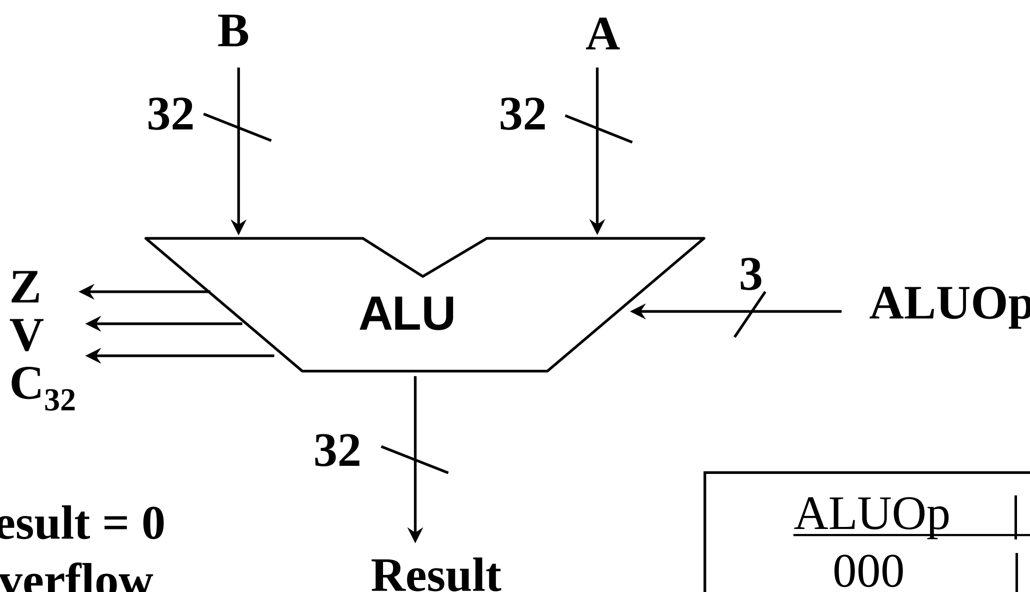
- In the MIPS architecture logical operations (and, or, xor) correspond to bit-wise operations.

(a) and r1, r2, r3	r3 = 1010	(r1 is 1 if r2 and r3 are both one)
	r2 = 0110	
	r1 = 0010	
(b) or r1, r2, r3	r3 = 1010	(r1 is 1 if r2 or r3 is one)
	r2 = 0110	
	r1 = 1110	
(c) xor r1, r2, r3	r3 = 1010	(r1 is 1 if r2 and r3 are different)
	r2 = 0110	
	r1 = 1100	

- Immediate versions of these instructions are also supported.

ALU Interface

- We will be designing a 32-bit ALU with the following interface.



Z = 1, if Result = 0
V = 1, if Overflow
C_{32} = 1, if Carry-Out

ALUOp	Function
000	AND
001	OR
010	ADD
110	SUBTRACT
111	SET-ON-LESS-THAN

Set-on-less-than

- The set-on-less instruction
  slt $s1, $s2, $s3
  sets $s1 to '1' if ($s2 < $s3) and to '0' otherwise.

- This can be accomplished by
  - subtracting $s3 from $s2
  - setting the least significant bit to the sign bit of the result
  - setting all other bits to zero
  - if overflow occurs the sign bit needs to be inverted

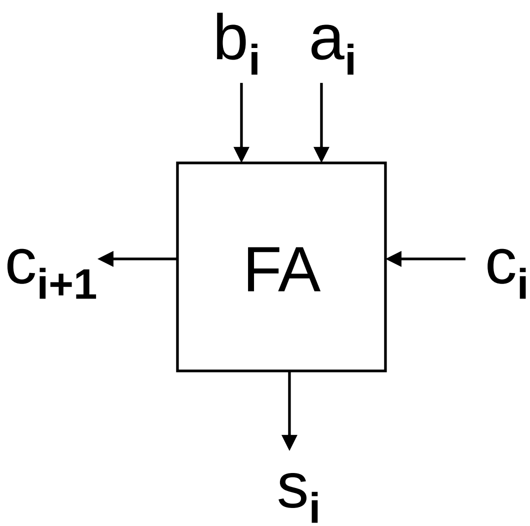
- For example,

$s2 = 1010	$s2= 0111
-$s3 = 1011	-$s3 = 0100
= 1111	= 0011
$s1 = 0001	$s1 = 0000

Full Adder

- A fundamental building block in the ALU is a full adder (FA).
- A FA performs a one bit addition.

$$a_i + b_i + c_i = 2c_{i+1} + s_i$$



c_{i+1}	c_i	1 1 1 0
	a_i	0 1 0 1
	b_i	0 1 1 1
	s_i	1 1 0 0

Full Adder Logic Equations

- s_i is '1' if an odd number of inputs are '1'.
- c_{i+1} is '1' if two or more inputs are '1' .

a_i	b_i	c_i	c_{i+1}	s_i
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

$$s_i = \overline{a}_i\overline{b}_i c_i + \overline{a}_i b_i \overline{c}_i + a_i \overline{b}_i \overline{c}_i + a_i b_i c_i$$

$$s_i = a_i \otimes b_i \otimes c_i$$

$$c_{i+1} = \overline{a}_i b_i c_i + a_i \overline{b}_i c_i + a_i b_i \overline{c}_i + a_i b_i c_i$$

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

$$c_{i+1} = a_i b_i + c_i(a_i + b_i)$$

$$c_{i+1} = a_i b_i + c_i(a_i \otimes b_i)$$

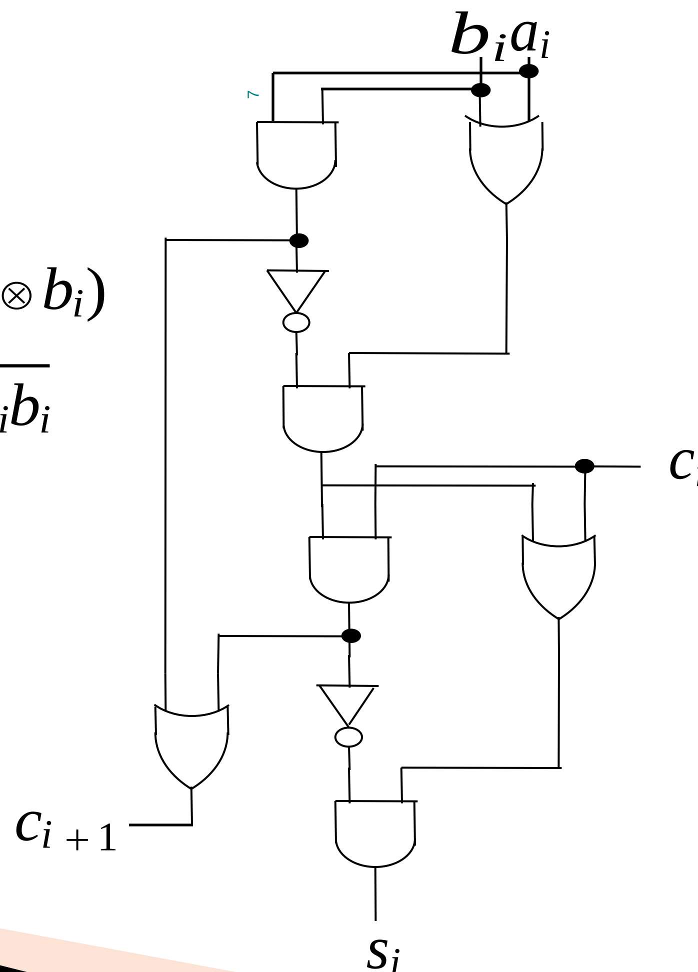
Full Adder Design

- One possible implementation of a full adder uses nine gates.

$$s_i = a_i \otimes b_i \otimes c_i$$

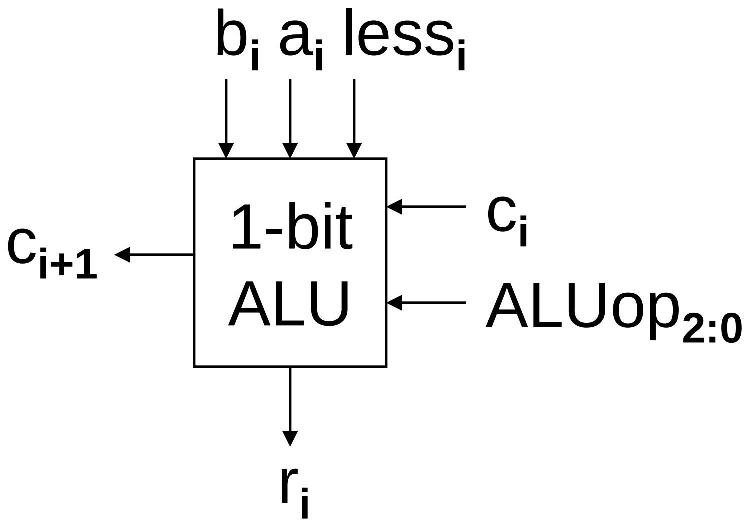
$$c_{i+1} = a_i b_i + c_i(a_i \otimes b_i)$$

$$a_i \otimes b_i = (a_i + b_i)\overline{a_i b_i}$$

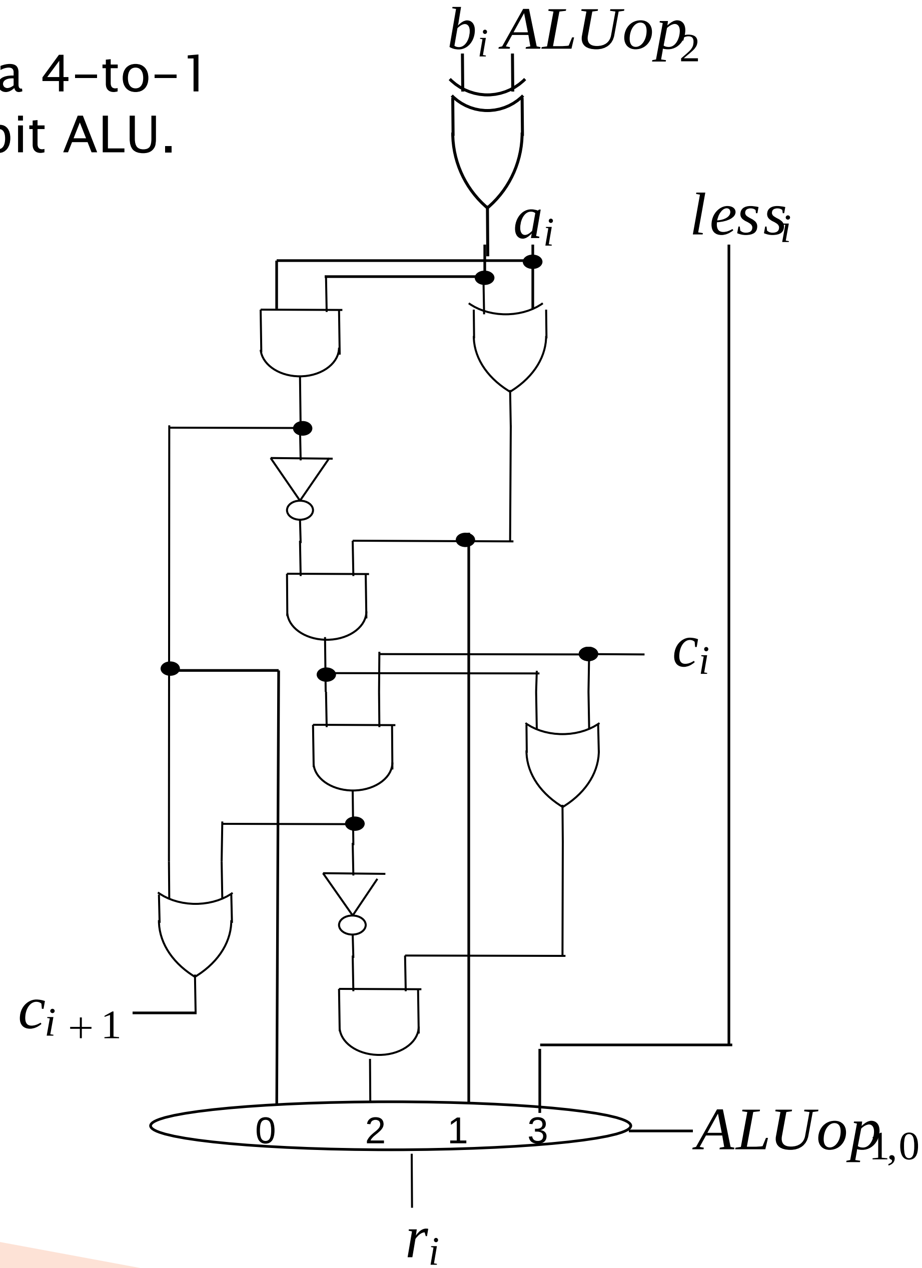


1-Bit ALU

- The full adder, an xor gate, and a 4-to-1 mux are combined to form a 1-bit ALU.



ALUOp	Function
000	AND
001	OR
010	ADD
110	SUBTRACT
111	SET-ON-LESS-THAN

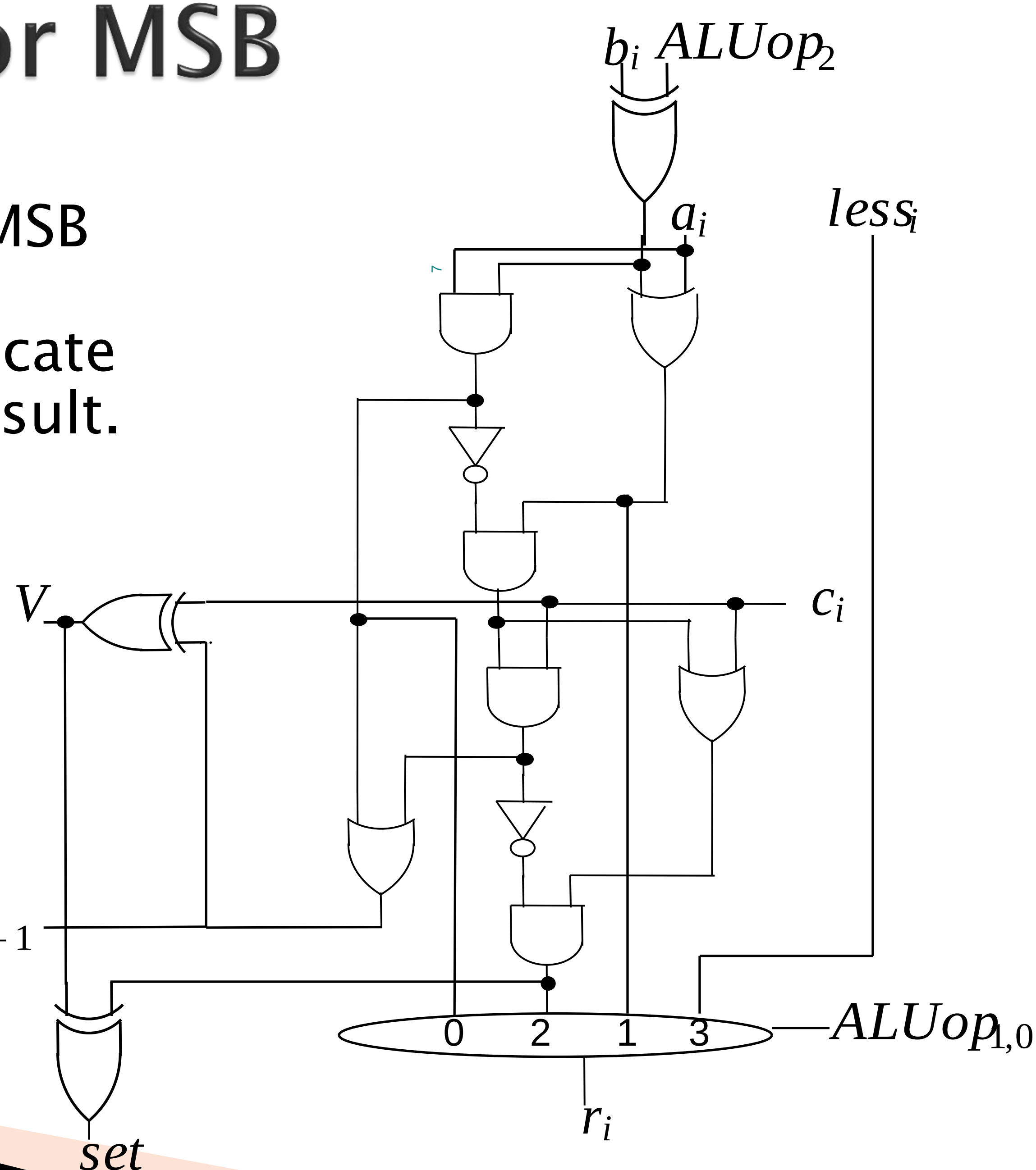


1-bit ALU for MSB

- The ALU for the MSB must also detect overflow and indicate the sign of the result.

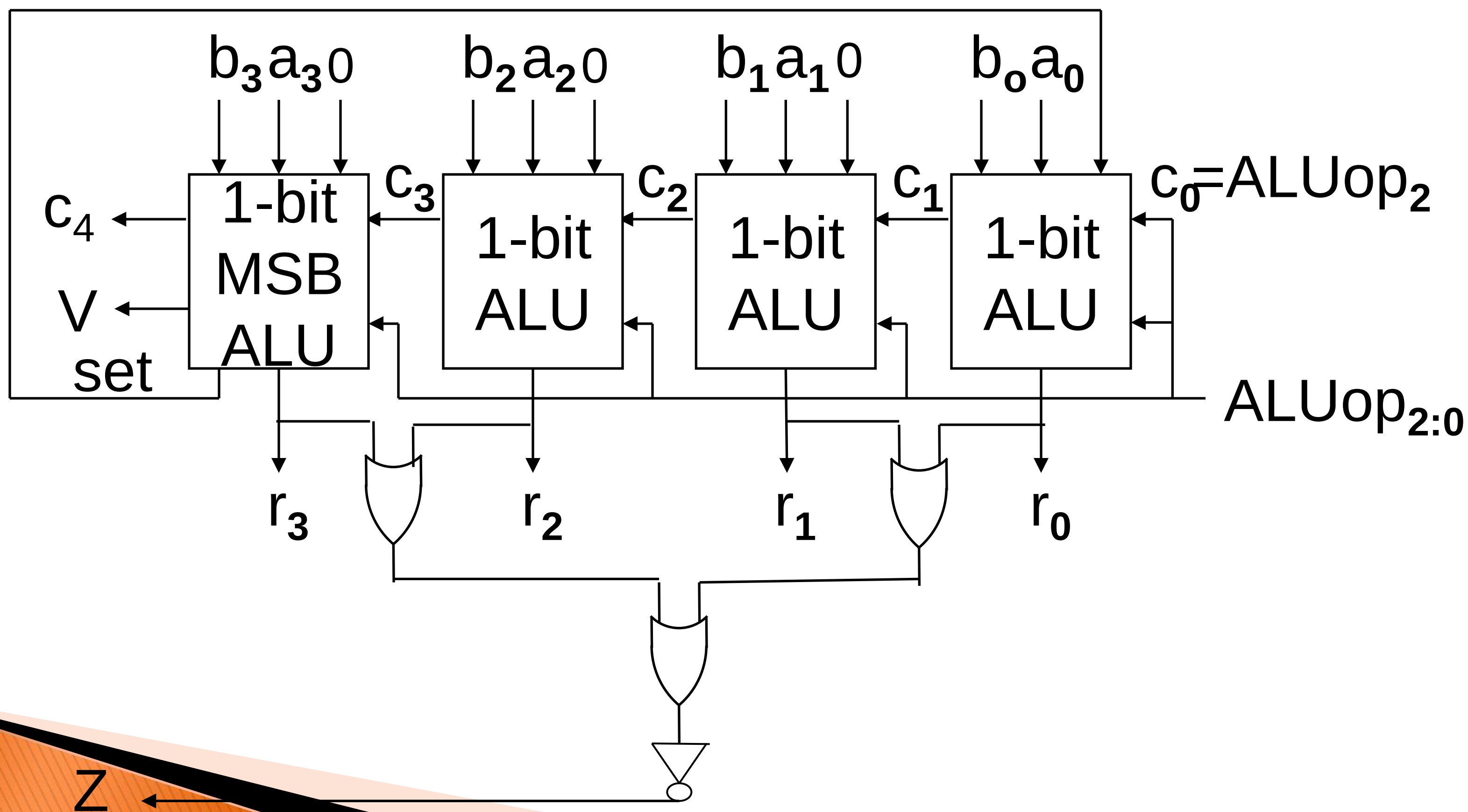
$$V = c_n \otimes c_{n-1}$$

$$set = (A < B)$$



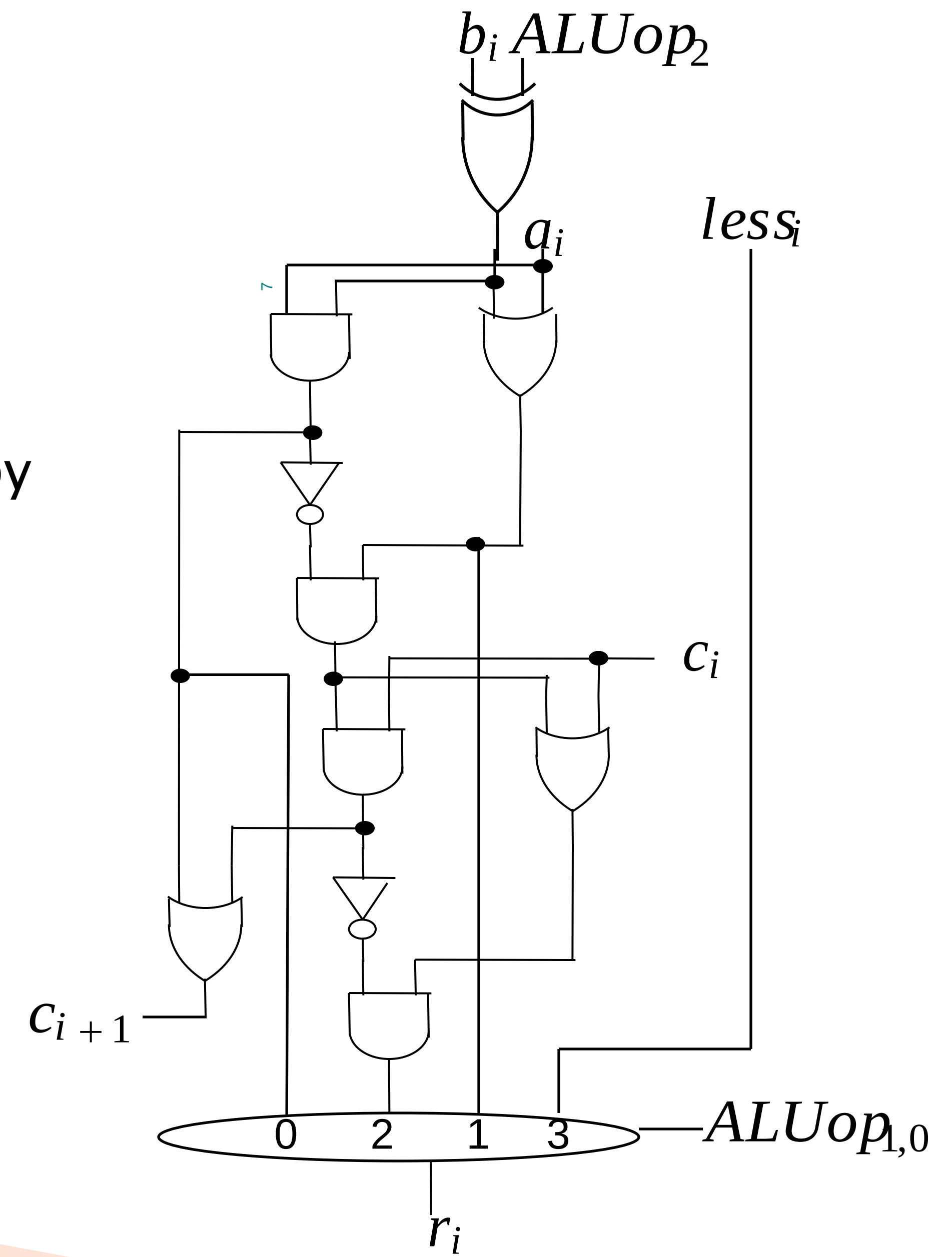
Larger ALUs

- Three 1-bit ALUs, a 1-bit MSB ALU, and a 4-input NOR gate can be concatenated to form a 4-bit ALU.



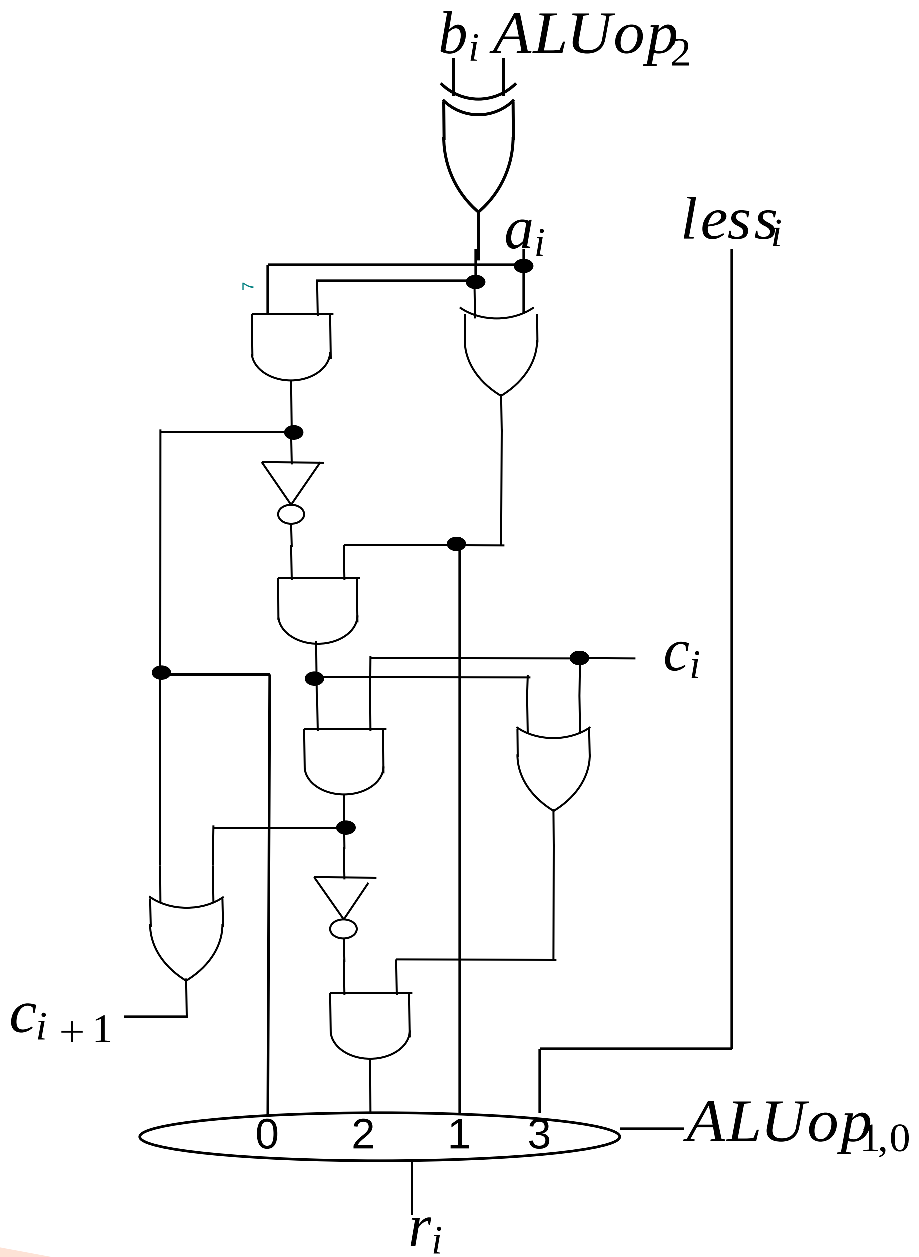
Gate Counts

- Assume
  - 4-input mux = 5 gates
  - XOR gate = 3 gates
  - AND/OR gate = 1 gate
  - Inverter = 0.5 gates.
- How many gates are required by
  - A 1-bit ALU?
  - A 4-bit ALU?
  - A 32-bit ALU?
  - An n-bit ALU?
- Additional gates needed to compute V and Z



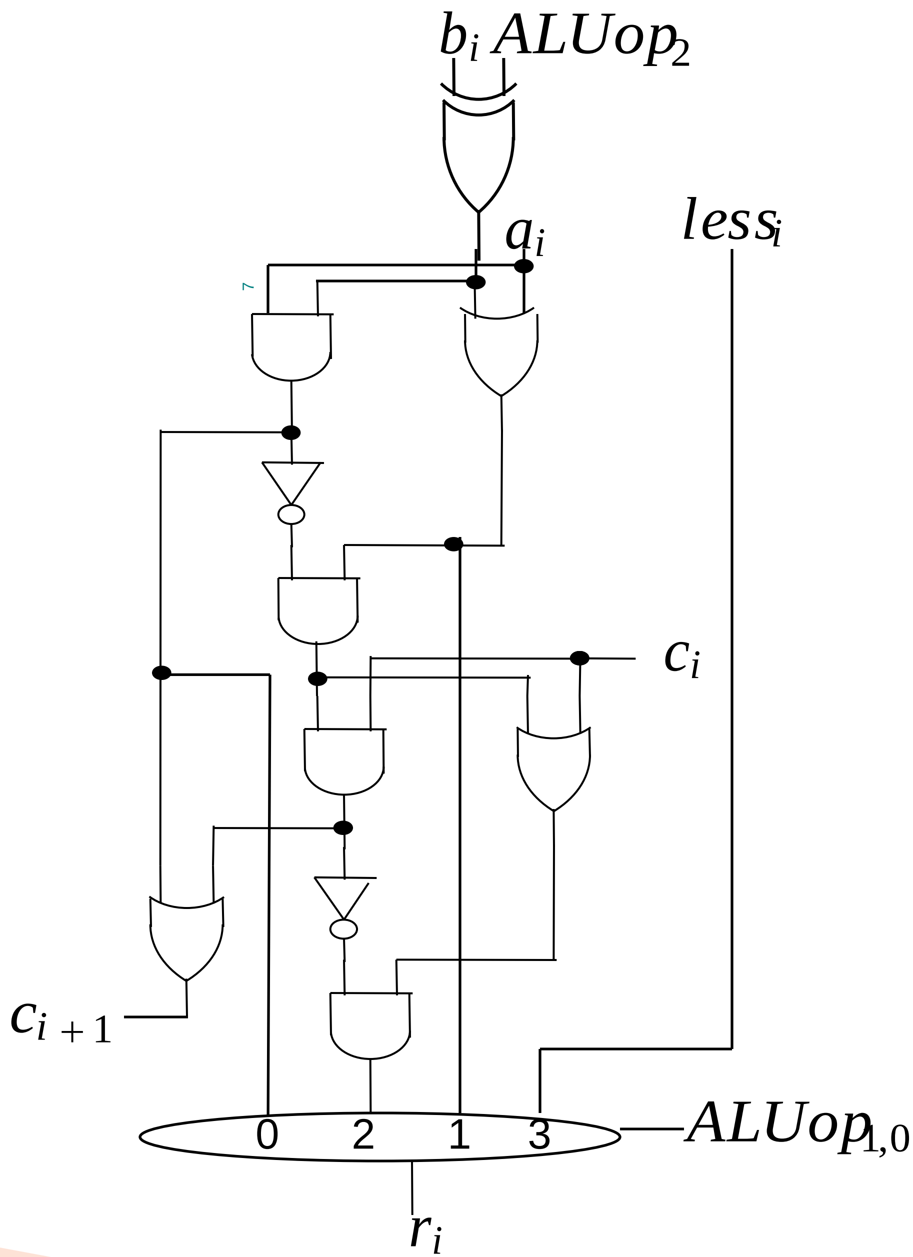
Gate Counts

- Assume
  - 4-input mux = 5 gates
  - XOR gate = 3 gates
  - AND/OR gate = 1 gate
  - Inverter = 0.5 gates.
- How many gates are required by
  - A 1-bit ALU? 16
  - A 4-bit ALU? 16x4
  - A 32-bit ALU? 16x32
  - An n-bit ALU? 16xn
- (n-1) 2-input OR gates, 1 inverter and 1 XOR gate are needed to compute V and Z for an n-bit ALU



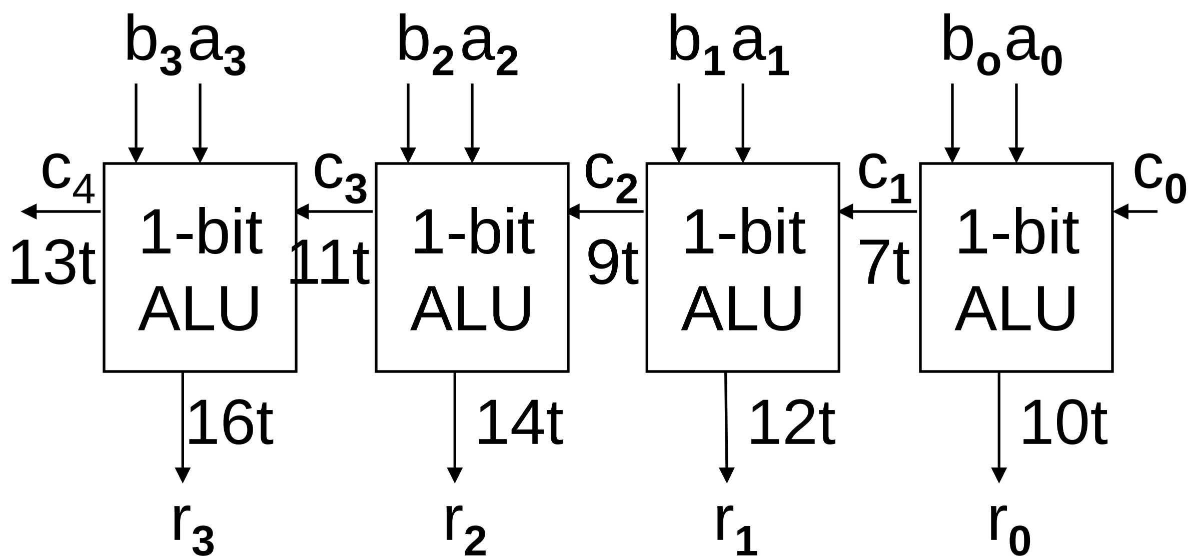
Gate Delays

- Assume delays of
  - 4-input mux = 2t
  - XOR gate = 2t
  - AND/OR gate = 1t
  - Inverter = 1t
- What is the delay of
  - A 1-bit ALU?
  - A 4-bit ALU?
  - A 32-bit ALU?
  - An n-bit ALU?
- Additional delay needed to compute Z



Ripple Carry Adder (RCA)

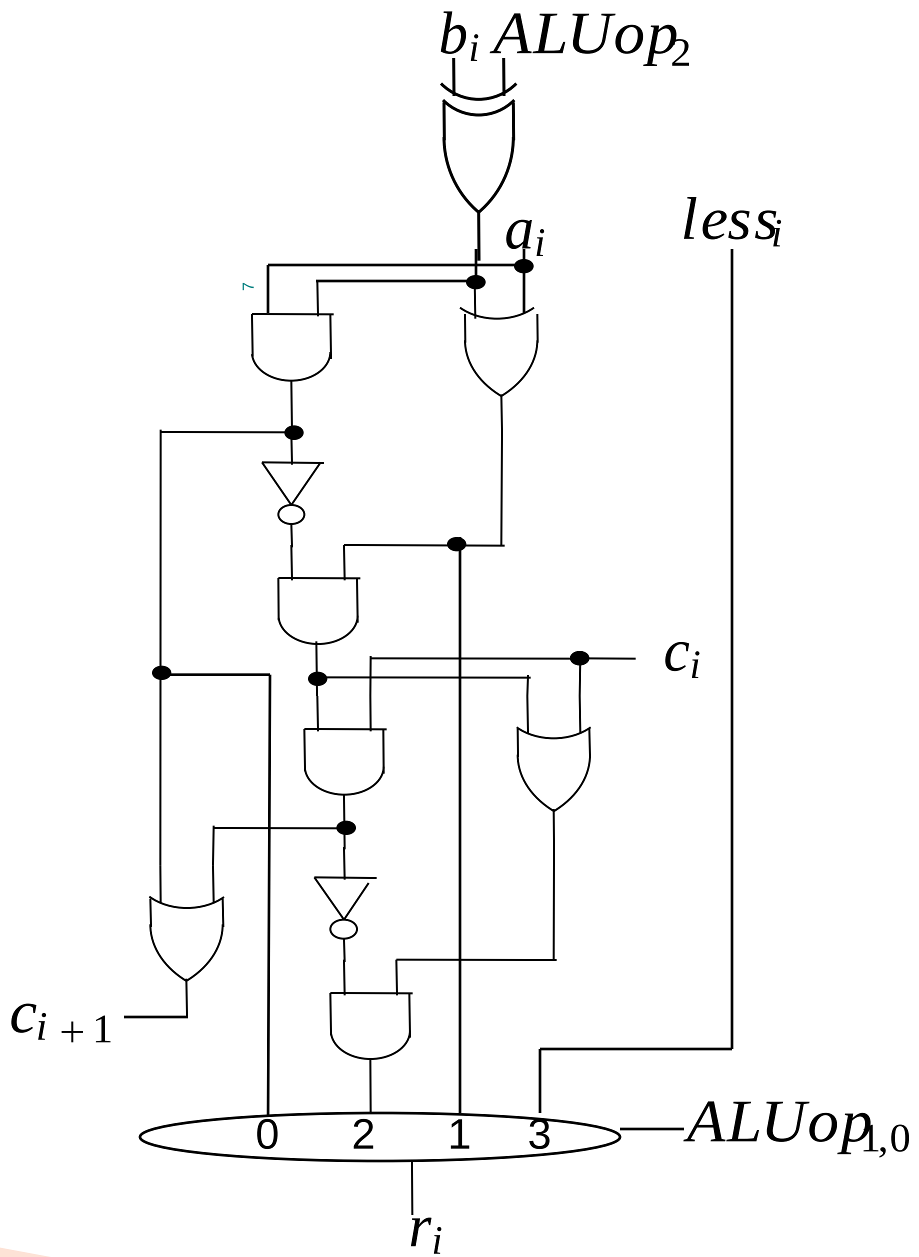
- With the previous design the carry "rippled" from one 1-bit ALU to the next.



- These leads to a relatively slow design.
- Z is ready at 19 t

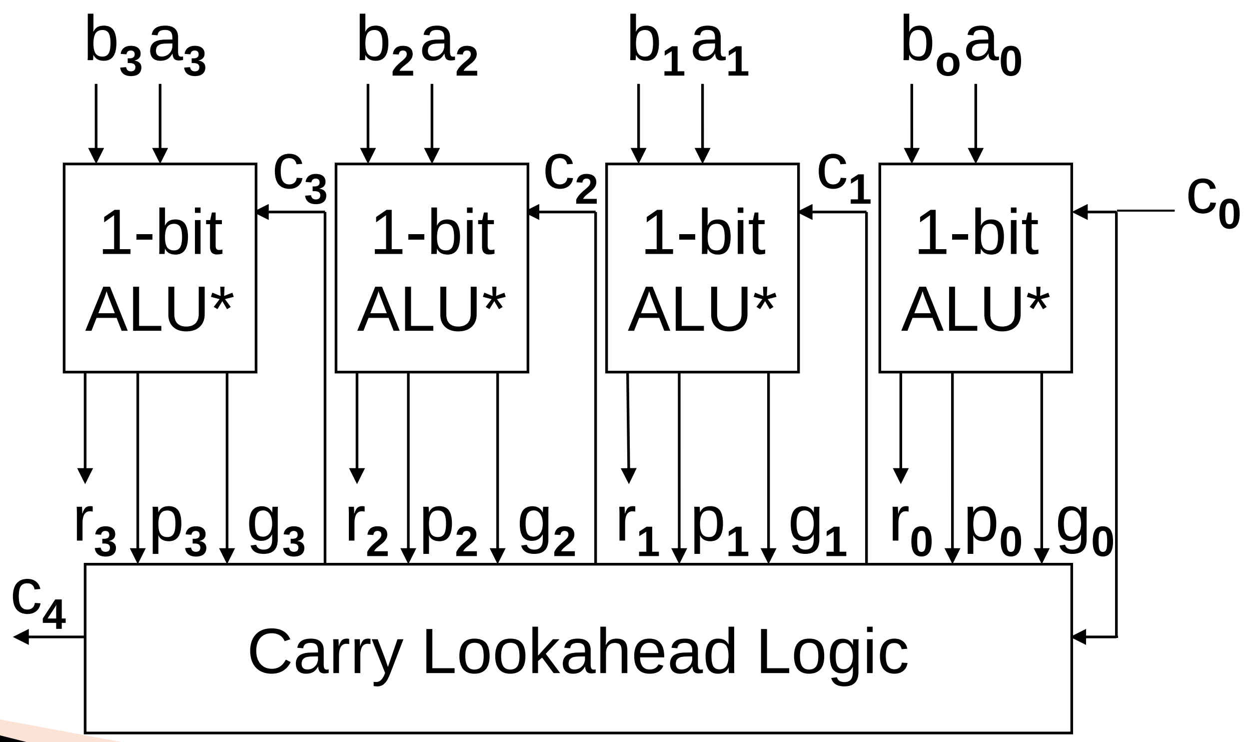
Gate Delays

- Assume delays of
  - 4-input mux = 2t
  - XOR gate = 2t
  - AND/OR gate = 1t
  - Inverter = 1t
- What is the delay of
  - A 1-bit ALU? 10t
  - A 4-bit ALU? 16t
  - A 32-bit ALU? (2x32+8)t = 72t
  - An n-bit ALU? (2n+8)t
- $\lceil \log_2(n) \rceil$ levels of 2-input OR gates and 1 inverter are needed to compute Z.



Carry Lookahead Adder (CLA)

- With a CLA, the carries are computed in parallel using carry lookahead logic (CLL).



Carry Logic Equation

- The carry logic equation is

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i$$

- We define a <u>propagate</u> signal

$$p_i = a_i + b_i$$

  and a <u>generate</u> signal

$$g_i = a_i b_i$$

- This allows the carry logic equation to be rewritten as

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

Carry Lookahead Logic

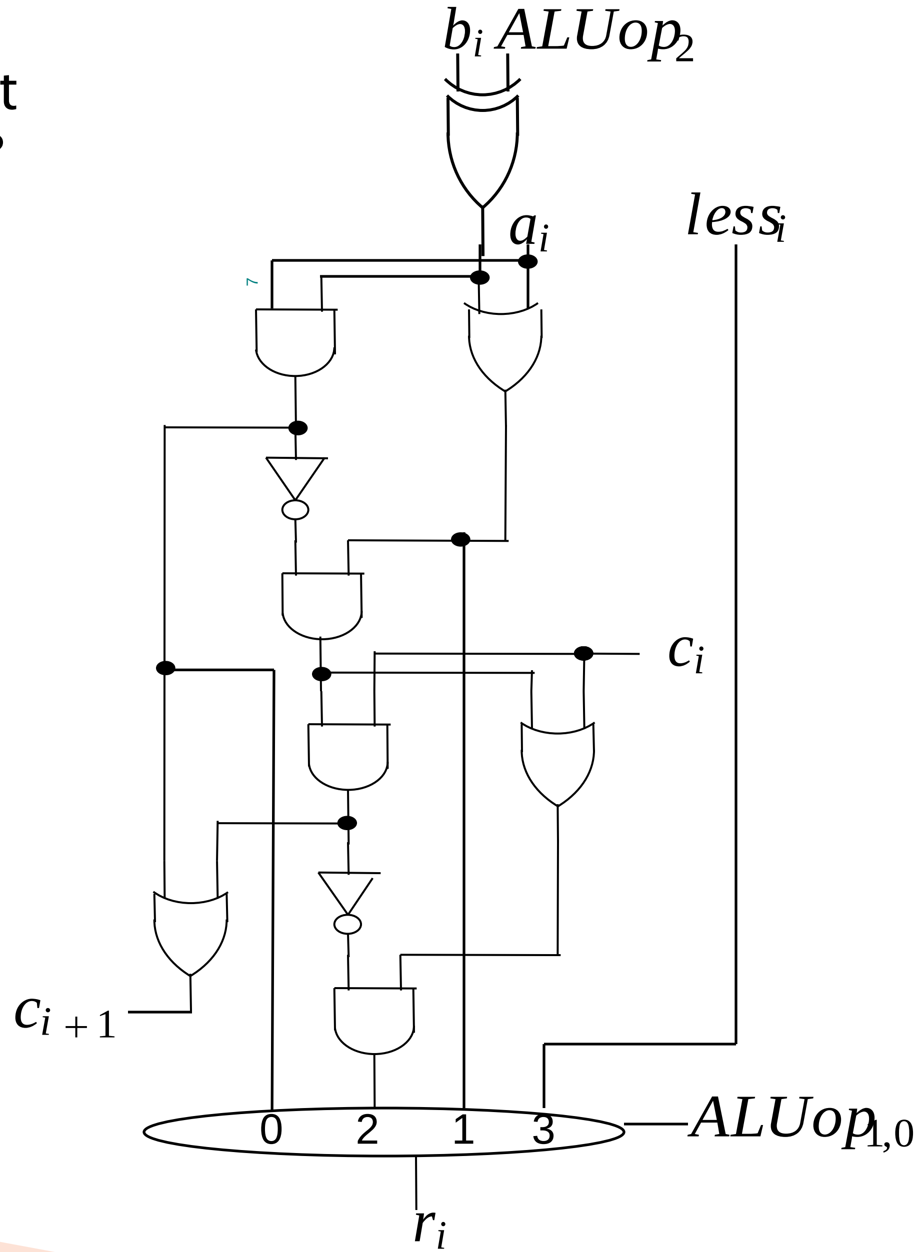
- For a 4-bit carry lookahead adder, the carries are computed as

$$c_1 = g_0 + p_0c_0$$
$$c_2 = g_1 + p_1c_1 = g_1 + p_1(g_0 + p_0c_0)$$
$$= g_1 + p_1g_0 + p_1p_0c_0$$
$$c_3 = g_2 + p_2c_2 = g_2 + p_2(g_1 + p_1g_0 + p_1p_0c_0)$$
$$= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$$
$$c_4 = g_3 + p_3c_3 = g_3 + p_3(g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0)$$
$$= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

- How many gates does the 4-bit CLL require, if gates can have unlimited fan-in?
- If each logic level has a delay of only 1t, the CLL has a delay of 2t. => In practice this may not be realistic.

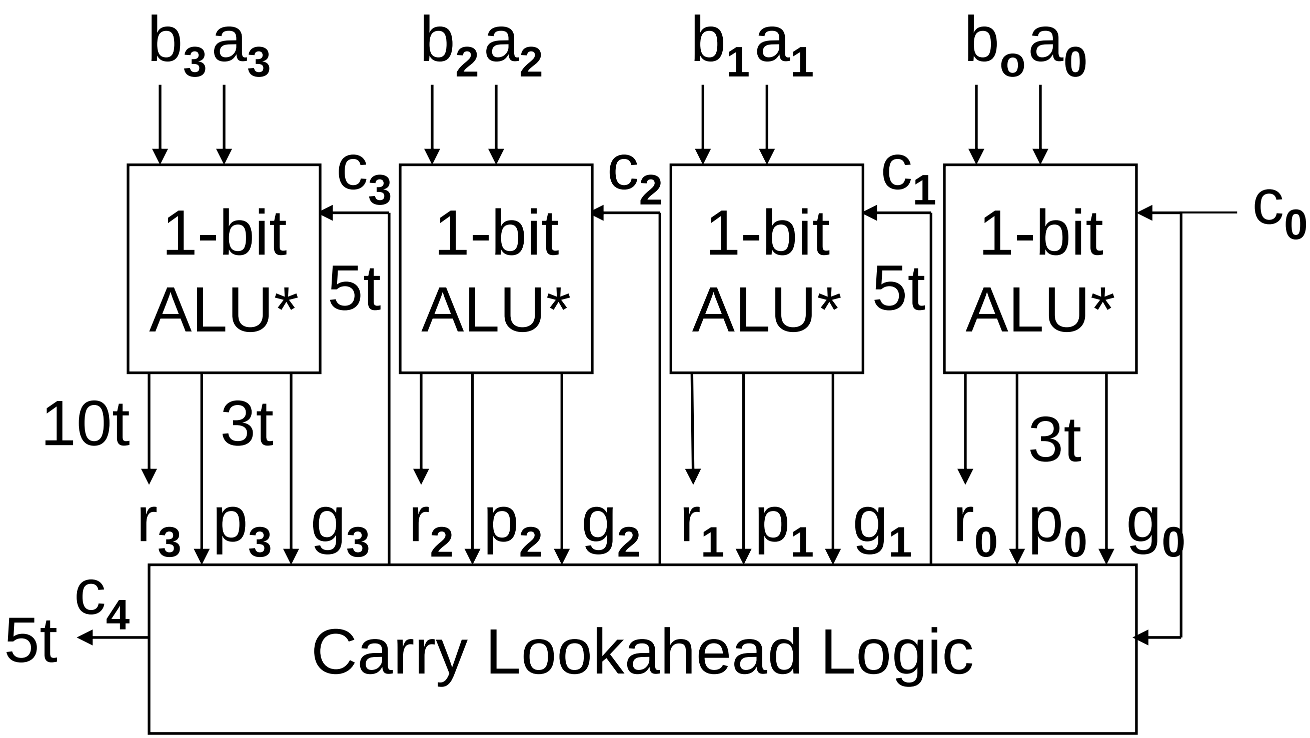
Modifying the 1-bit ALU

- How would we modify our 1-bit ALU if it is to be used in a CLA?
- How many gates does the modified 1-bit ALU require?
- How many gates does a 4-bit CLA require?
- How many gate delays until p_i and g_i are ready?



4-bit CLA Timing

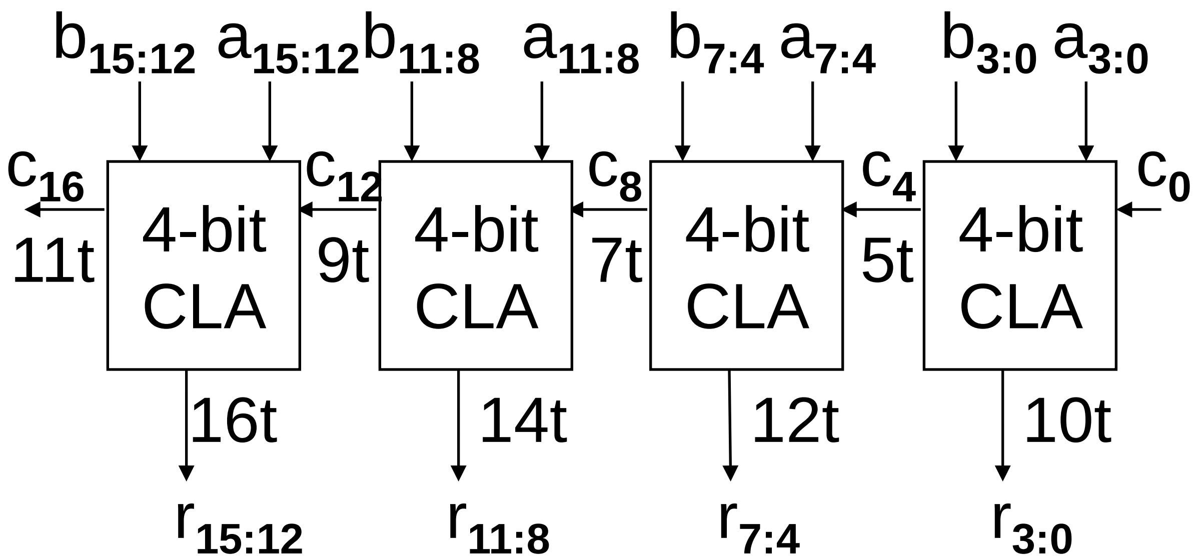
- With a carry lookahead adder, the carries are computed in parallel using carry lookahead logic.



- This design requires 15x4 +14 = 74 gates, without computing V or Z

16-bit ALU – Version 1

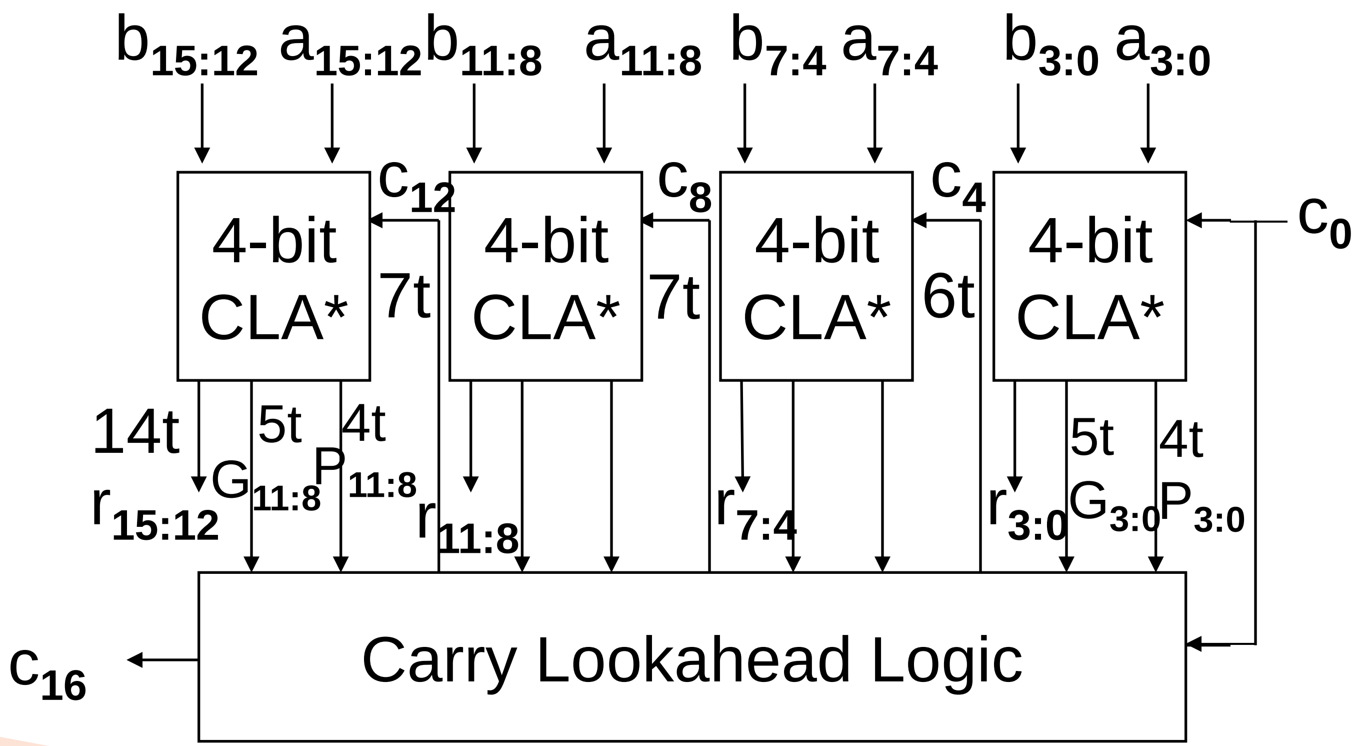
- A 16-bit ALU could be constructed by concatenating four 4-bit CLAs and letting the carry "ripple" between 4-bit "blocks".



- This design requires 74x4 = 296 gates, without computing V or Z.

16-bit ALU – Version 2

- Another approach is to use a second level of carry lookahead logic.
- This approach is faster, but requires more gates   16x15 + 5x14 = 310 gates



4-bit CLA*

- The 4-bit CLA* (Block CLA) is similar to the first 4-bit CLA, except the CLL computes a "block" generate and "block propagate", instead of a carry out.
- Thus the computation

  $$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

  is replaced by

  $$P_{3:0} = p_3p_2p_1p_0$$

  $$G_{3:0} = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$$

- Note: $c_4 = G_{3:0} + P_{3:0}c_0$
- This approach limits the maximum fan-in to four, and the carry-lookahead logic still requires 14 gates.

Conclusions

- An *n*–bit ALU can be designed by concatenating *n* 1-bit ALUs.
- Carry lookahead logic can be used to improve the speed of the computation.
- A variety of design options exist for implementing the ALU.
- The best design depends on area, delay, and power requirements, which vary based on the underlying technology.