Computer Architecture

Arithmetic

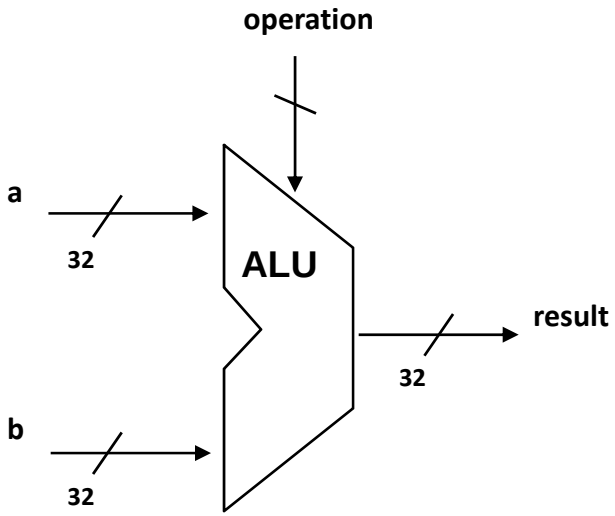
ARASH HABIBI LASHKARI
(April- 2010)

Computer Arithmetic

- This lecture will introduce basic number representations and introduce basic integer and floating point arithmetic.

Arithmetic

- Where we've studied so far:
  - Boolean algebra and basic logic circuits
  - Abstractions:
    Instruction Set
    Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture
  - So we understand instructions better



Number Representation

- Any number can be represented in any base by the simple sum of each digit's value, positional notation:

  $d * base^{i}$

- The value ๑๐๑๑ (binary) equals

  ๑ * ๒^{๓} + ๐ * ๒^{๒} + ๑ * ๒^{๑} + ๑ * ๒^{๐} = ๑๑ (decimal).

Number Representation

- as a ๓๒ bit number this would be represented as follows:

๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๑๐๑๑

- The most significant bit (MSB) is on the left and the least significant bit (LSB) is on the right. In the number above, the LSB is called bit ๐ and the MSB is called bit ๓๑.

Range of Values

- With ๓๒ bits it is possible to have unsigned numbers that range from ๐ to $๒^{๓๒}$ - ๑ = ๔,๒๙๔,๙๖๗,๒๙๕.
- This is quite reasonable for address calculations of present machines, but future machines will surely have more than $๒^{๓๒}$ memory locations.
- In addition to more memory it is highly useful to be able to represent negative as well as positive integers and numbers smaller and larger then those possible with this format.

Microprocessor without Interlocked Pipeline Stages (MIPS)

- 32 bit signed numbers:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$ ***maxint***
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Signed Integers

- Since in the binary number system we have an even number of unique representations for a given number of bits we have two choices:

- Have a balanced system with the same number of negative and positive values, but what about zero? If we represent zero we have an odd number of values to represent. This can be done by allowing zero to be represented by two different bit patterns.

Signed Integers

๒. Have an unbalanced system where zero has one representation, but there is either an extra positive or negative value.

- We would like a balanced system, but this would require two different representations for the value zero.
- This evil (having two zeros) is better to avoid and not worth the benefits of having a balanced system. Consequently, an unbalanced system has been nearly universally adopted.

One's Complement

- How can we represent negative and positive numbers in the binary system?

- The ๑'s complement number system using ๓๒ bits has a range from -(๒^๓๑ -๑) to +(๒^๓๑ - ๑).

- Zero can be represented as either positive or negative. The two forms of zero are represented by

  ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ (all zeros) or

  ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ (all ones).

One's Complement

- A negative number is formed by representing its magnitude in normal, positive binary format and then inverting each bit.
- ₈-bit examples:

  0000 0000 = -0

  0000 0000 = -0

  0000 0000 = +0

  0000 0000 = +0

- Having two forms of zero is a problem, but arithmetic is simple.

Two's Complement

- The ๒'s complement number system using ๓๒ bits has a range from -๒^๓๑ to ๒^๓๑ - ๑.
- Zero has only one representation: all zeros, but there is one extra negative value.
- Negative numbers always have the MSB set to ๑ -- thus making sign detection extremely simple.
- Converting ๒'s complement numbers to signed decimal is extremely simple.

Example ๒'s complement

Convert ๑๐๑๑ 1010 to signed decimal.

$$๑ * -๒^{๗} + ๐ * ๒^{๖} + ๑ * ๒^{๕} + ๑ * ๒^{๔} + ๑ * ๒^{๓} + ๐ * ๒^{๒} + ๑ * ๒^{๑} + ๐ * ๒^{๐}$$

$$= -๑๒๘ + ๐ + ๓๒ + ๑๖ + ๘ + ๐ + ๒ + ๐$$

$$= -๗๐.$$

- This approach is not feasible in hardware due to speed considerations.

Two's Complement Shortcuts

- Negation in two's complement is accomplished by inverting each bit of the number and then adding one to the result.
- Example: Convert -๗๐ (decimal) to two's complement binary.

  |-๗๐| = ๗๐ = ๐๑๑๐ 0110 (binary)

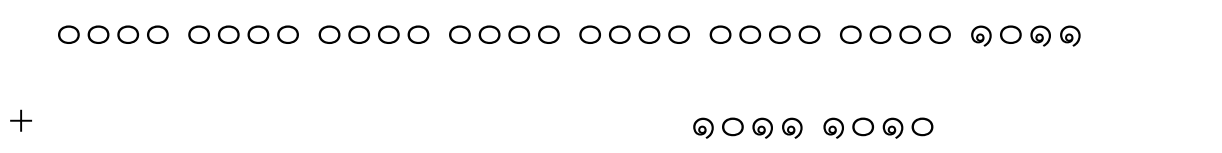
  ~ ๐๑๑๐ 0110 = ๑๐๐๑ 1001 (where ~ denotes bit inversion)

  1011 1001 + ๑ = ๑๐๐๑ 1010 = -๗๐ (๒'s comp).

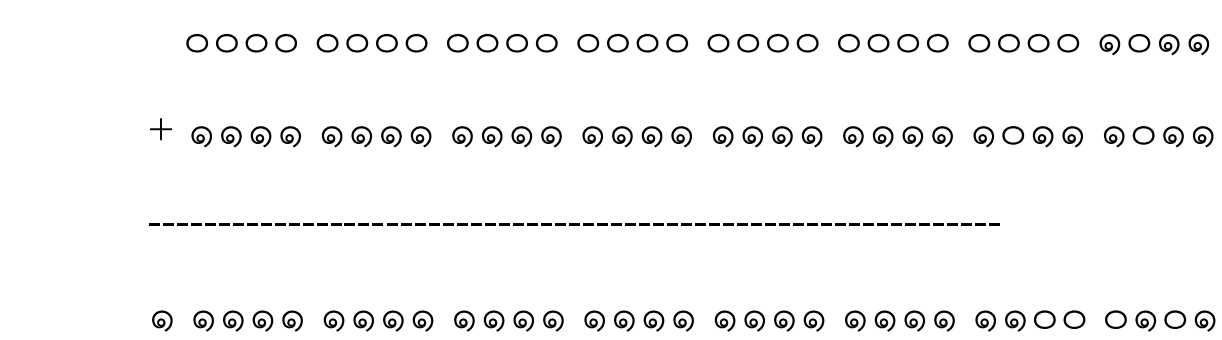
Sign Extension

- To add a ๑๖-bit value to a ๓๒-bit value, the ๑๖-bit value must first be sign extended. The two numbers are right aligned and the sign bit of the shorter number is replicated to the left until the two numbers are equal in length.

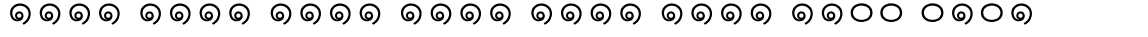
Example Sign Extension



we must sign extend the second number and then add:



- The extra bit on the left is simply discarded, leaving us with



for our answer.

Possible Number Representations

- Sign Magnitude: One's Complement Two's Complement

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

Integer Addition

- Straight forward approach consists of adding each bit together from right to left and propagating

  the carry from one bit to the next.

  Perform the operation ๗ + ๖

  ๐๑๑๑ + ๐๑๑๐ = ๑๑๐๑

Integer Addition

Add the numbers ๒,๑๔๗,๔๘๓,๖๔๗ + ๒

This is known as overflow

Overflow can be handled in several ways

- An exception or interrupt can be asserted
- A bit in a status register can be set
- It can be completely ignored

Integer Subtraction

- Straight forward approach consists of negating one term and performing integer addition.

Perform the operation ๗ - ๖

๐๑๑๑ - ๐๑๑๐ = ๐๐๐๑

Perform the operation ๖ - ๗

๐๑๑๐ - ๐๑๑๑ = ๑๑๑๑

Integer Subtraction

Add the numbers -๒,๑๔๗,๔๘๓,๖๔๗ - ๒

๑๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๐ ๐๐๐๑

๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๑ ๑๑๑๐

This is also overflow

Overflow can be handled in several ways

- An exception or interrupt can be asserted
- A bit in a status register can be set
- It can be completely ignored

Integer Overflow

- Overflow can occur whenever the sum of two N bit numbers cannot be represented by another N bit number.
- Unsigned numbers must be treated differently than signed numbers.
- Unsigned numbers are usually used for address calculations and it is convenient to ignore overflow.
- In many architectures this is accomplished by having arithmetic instructions that ignore the overflow condition.

Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs when the value affects the sign:
  - overflow when adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive and get a negative
  - or, subtract a positive from a negative and get a positive
- Consider the operations A + B, and A – B
  - Can overflow occur if B is 0 ?
  - Can overflow occur if A is 0 ?

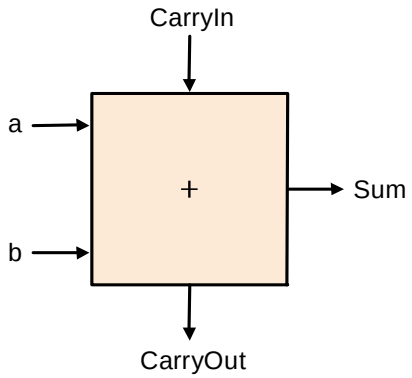
Effects of Overflow

- Most processors hardware automatically generates an exception (interrupt)
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
  - Details will be studied in a later chapter
- Don't always want to detect overflow
  - new MIPS instructions: `addu, addiu, subu`
  - Here, “u” stands for “un-overflowed”.

  note: `addiu` *still sign-extends!*
  note: `sltu, sltiu` *for unsigned comparisons*

Designing an ALU

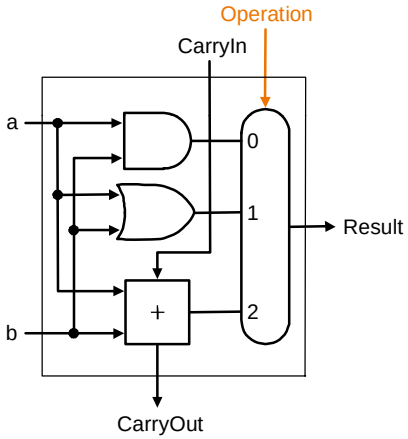
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:
-



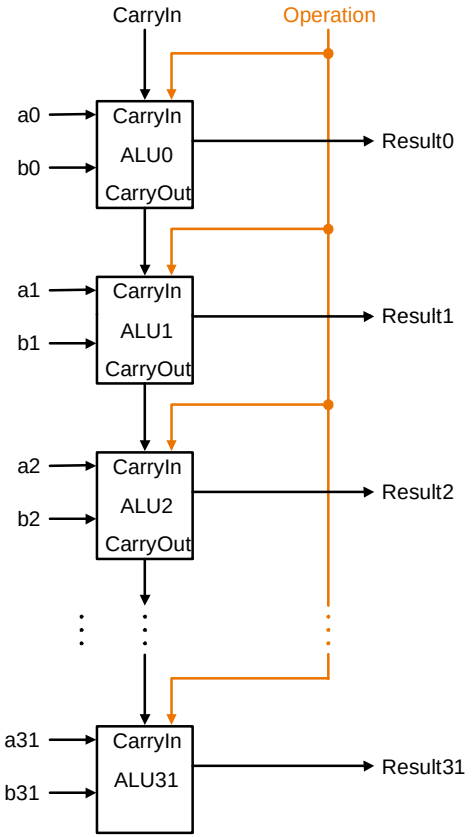
$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for *add, and*, and *or*?
- How could we build a 32-bit ALU?

Building a 32 bit ALU



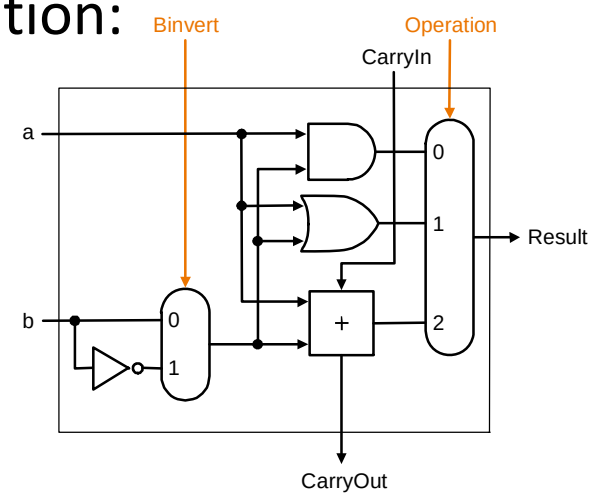
Each box is a 1bit ALU



What about subtraction (a – b) ?

- Two's complement approach: just negate b and add.
- How do we negate?

- A very clever solution:

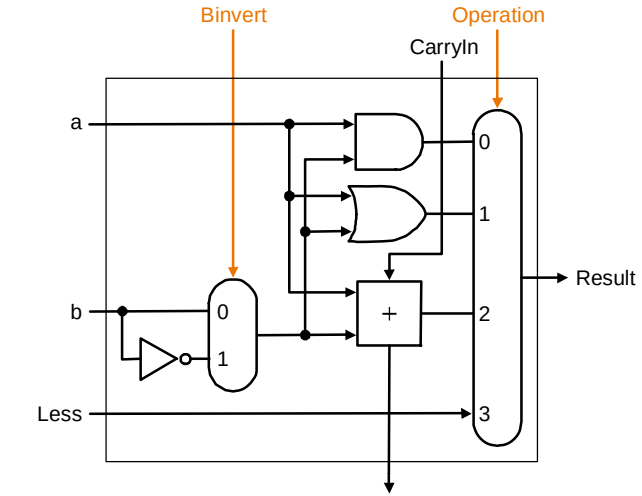


1-bit ALU
w/ negation

Tailoring the ALU to the MIPS

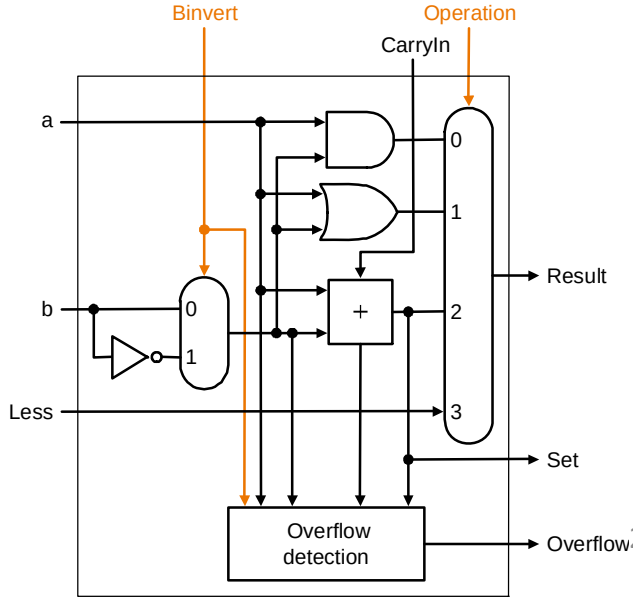
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

Supporting Set Less Than (slt)

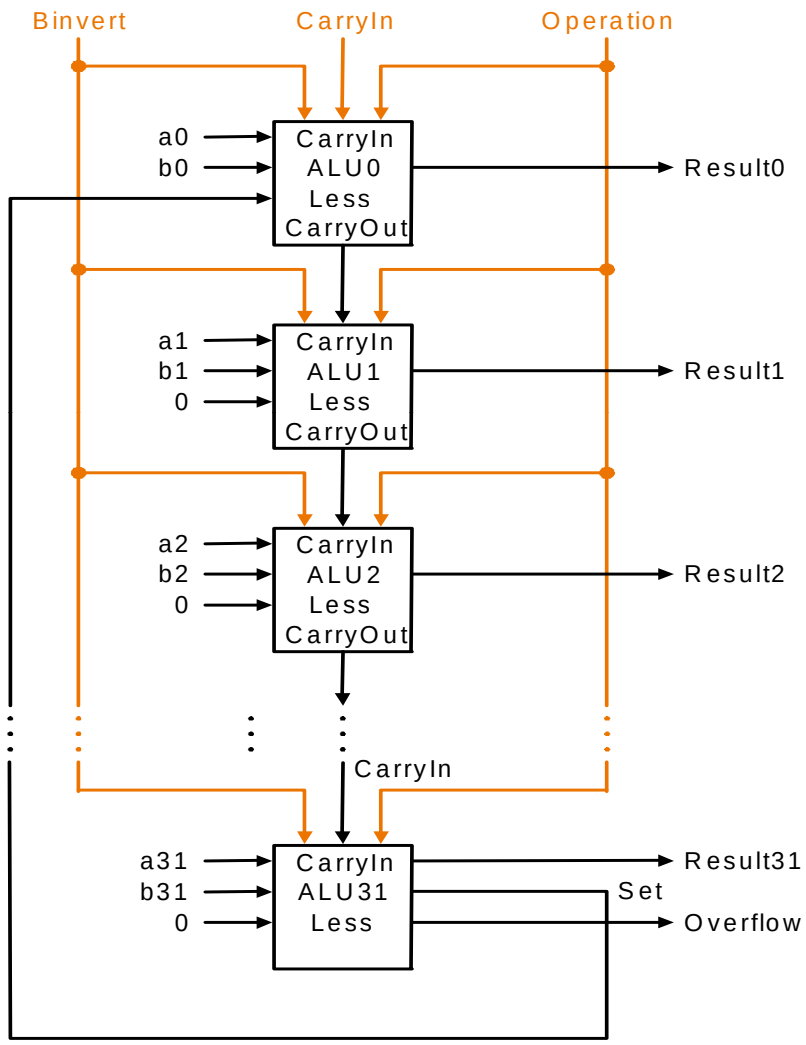


a.

- Can we figure out the idea?



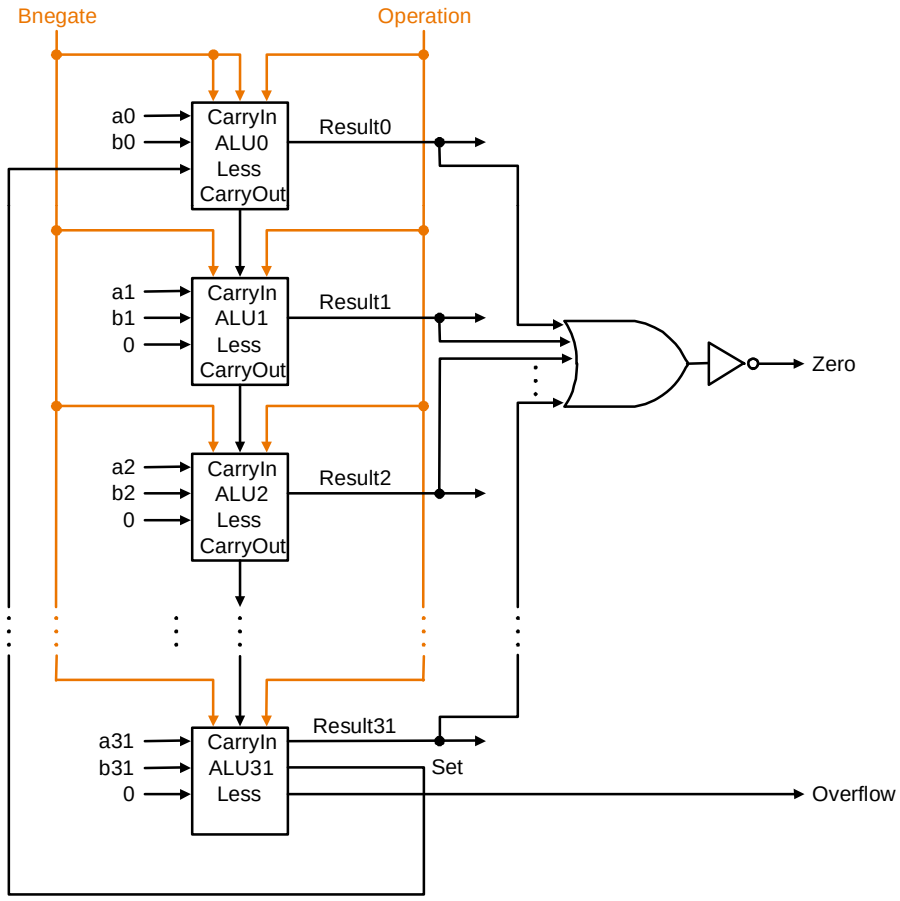
b.


Binvert
CarryIn
Operation
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow

Test for equality

- Notice control lines:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```



•***Note: zero is a 1 when the result is zero!***

Questions