FU05 Computer Architecture

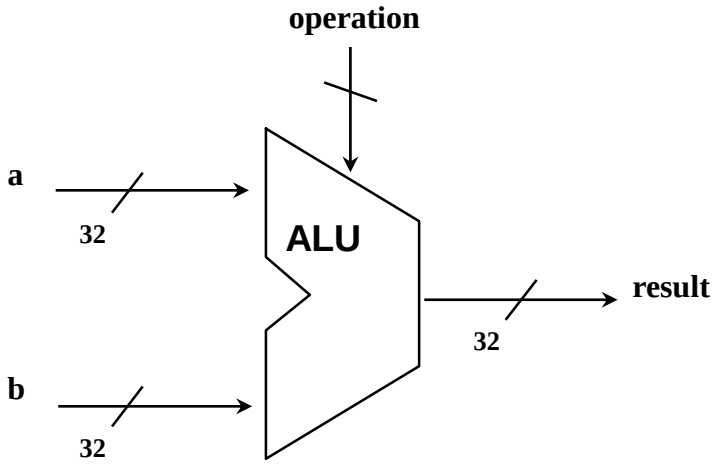
5. Arithmetic (演算回路)

Ben Abdallah Abderazek
E-mail: benab@u-aizu.ac.jp

Lectures of this course are adopted from the slides of the Book ‘’Computer Organization and Design - The Hardware/Software Interface, 5th Ed’’ by David. A. Patterson & John L. Hennessy

Arithmetic

- So far, we have seen:
  - Performance (seconds, cycles, instructions)
  - Instruction Set Architecture
  - Assembly Language and Machine Language
- What's next:
  - Implementing the Architecture



Arithmetic (Cont.)

- Operations on integers
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Dealing with overflow

*People use base 10, Computers use base 2

Signed binary numbers

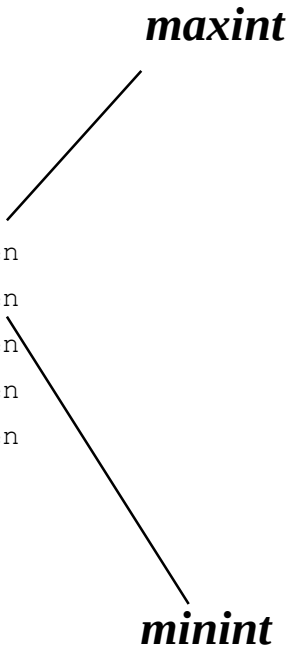
Possible representations:

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

- Issues: balance, number of zeros, ease of operations

32 bit signed numbers

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



Two's Complement Operations
2の補数演算

- Negating a two's complement number: invert all bits and add 1
  - remember: “negate” and “invert” are quite different!

Two's Complement Operations
2の補数演算

Converting n bit numbers into numbers with more than n bits:

- MIPS 8 bit, 16 bit values / immediates converted to 32 bits
- Copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

- MIPS "sign extension" example instructions:

lb	load byte (signed)
lbu	load byte (unsigned)
slti	set less than immediate (signed)
sltiu	set less than immediate (unsigned)

Addition & Subtraction

加減算

- Just like in grade school (carry/borrow 1s)

0111	0111	0110
+ 0110	- 0110	- 0101
?	?	?

- Two's complement operations easy
  - subtraction using addition of negative numbers

0110	0110
- 0101	+ 1010

- Overflow (result too large for finite computer word):
  - e.g., adding two n-bit numbers does not yield an n-bit number

```
  0111
+ 0001
------
  1000
```

note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

Detecting Overflow
Overflowの検出

- 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 ?

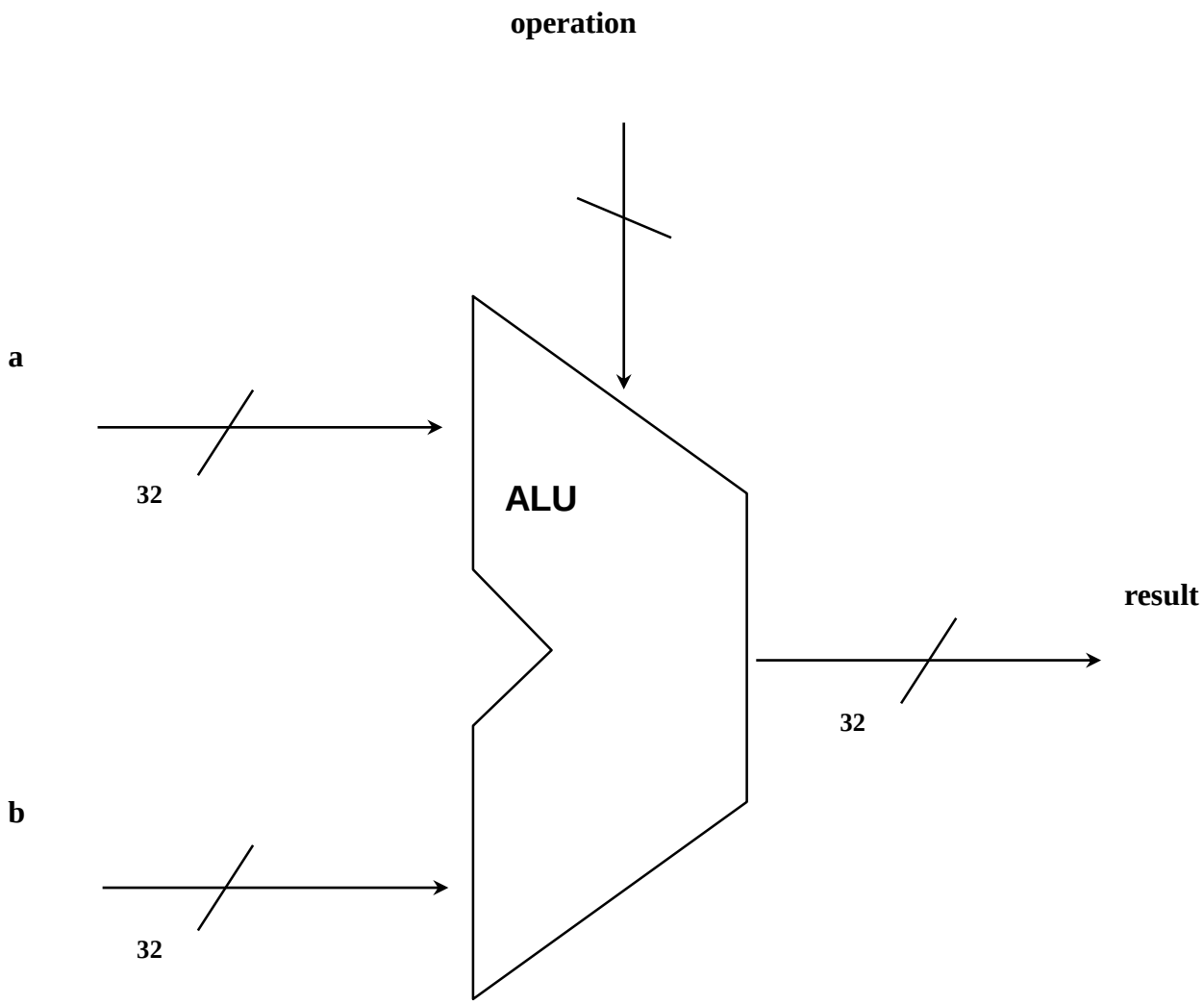
Impacts of Overflow
Overflowの影響

- When an exception (interrupt) occurs:
  - Control jumps to predefined address for exception (*interrupt vector*)
  - Interrupted address is saved for possible resumption in *exception program counter* (EPC); new instruction: `mfc0` `(move from coprocessor0)`
  - *Interrupt handler* handles exception (part of OS). registers `$k0` and `$k1` reserved for OS

- Details based on software system / language
  - C ignores integer overflow; FORTRAN not

- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`
  note: `addiu` and `sltiu` *still sign-extends.*

ALU: Arithmetic Logic Unit
算術論理演算ユニット



Logic operations

- Sometimes operations on individual bits needed:

Logic operation	C operation	MIPS instruction
Shift left logical	`<<`	`sll`
Shift right logical	`>>`	`srl`
Bit-by-bit AND	`&`	`and, andi`
Bit-by-bit OR	`|`	`or, ori`

- `and` and `andi` can be used to turn off some bits; `or` and `ori` turn on certain bits
- Of course, AND en OR can be used for logic operations.
  - Note: Language C's logical AND (`&&`) and OR (`||`) are *conditional*
- `andi` and `ori` perform no sign extension !

Review: Gates (論理ゲート)

Exercise: Given 3-input logic function of A, B and C, 2-outputs

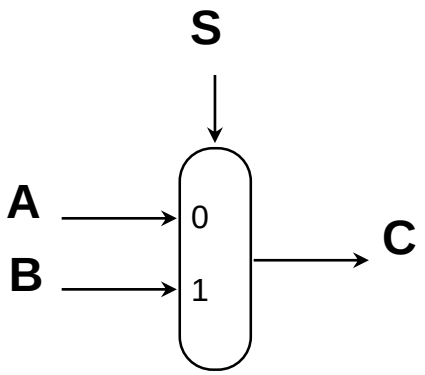
Output D is true if at least 2 inputs are true

Output E is true if odd number of inputs true

- Give truth-table
- Give logic equations
- Give implementation with AND and OR gates, and Inverters.

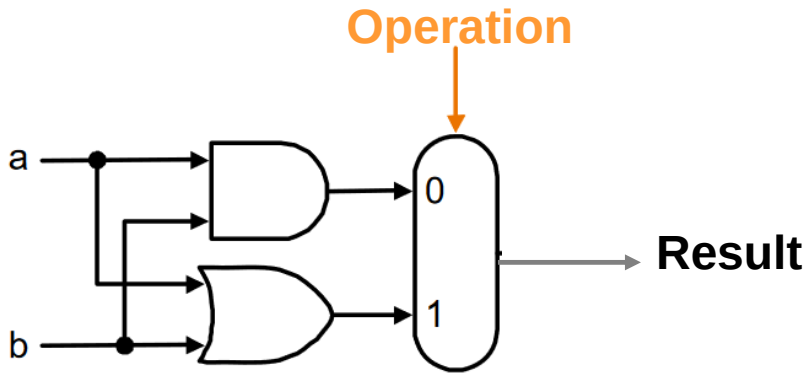
Review: The Multiplexor (マルチプレクサ)

- Selects one of the inputs to be the output, based on a control input



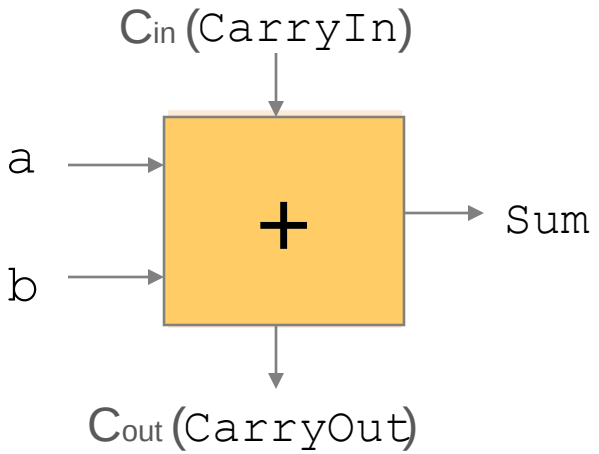
A 2-input mux.

- Lets build our ALU and use a MUX to select the outcome for the chosen operation



ALU Implementation
ALUの実装方法

- 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:

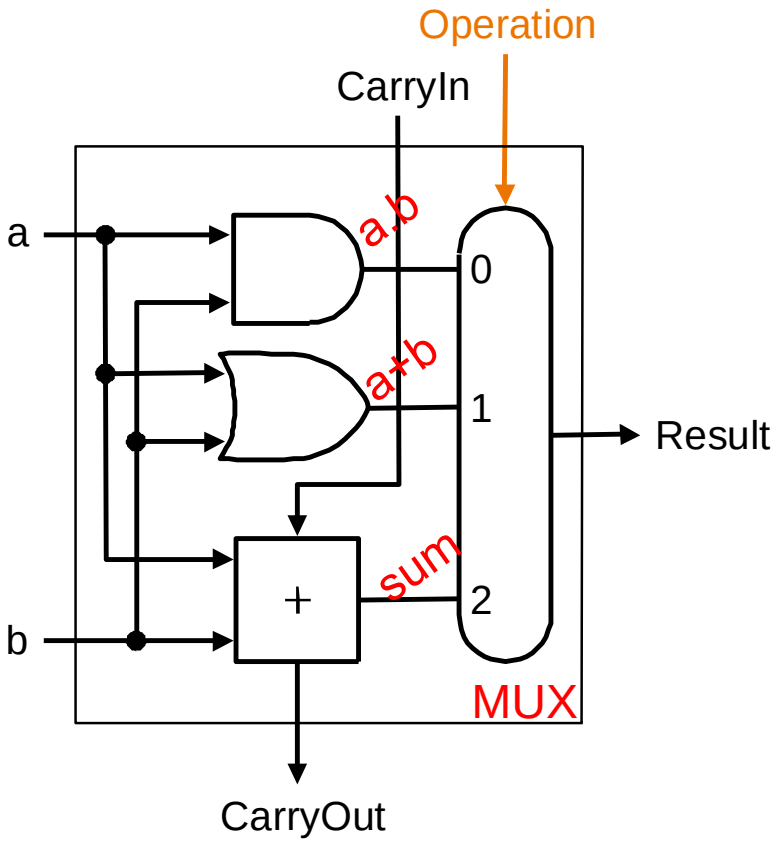


$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$
$sum = a \text{ xor } b \text{ xor } c_{in}$

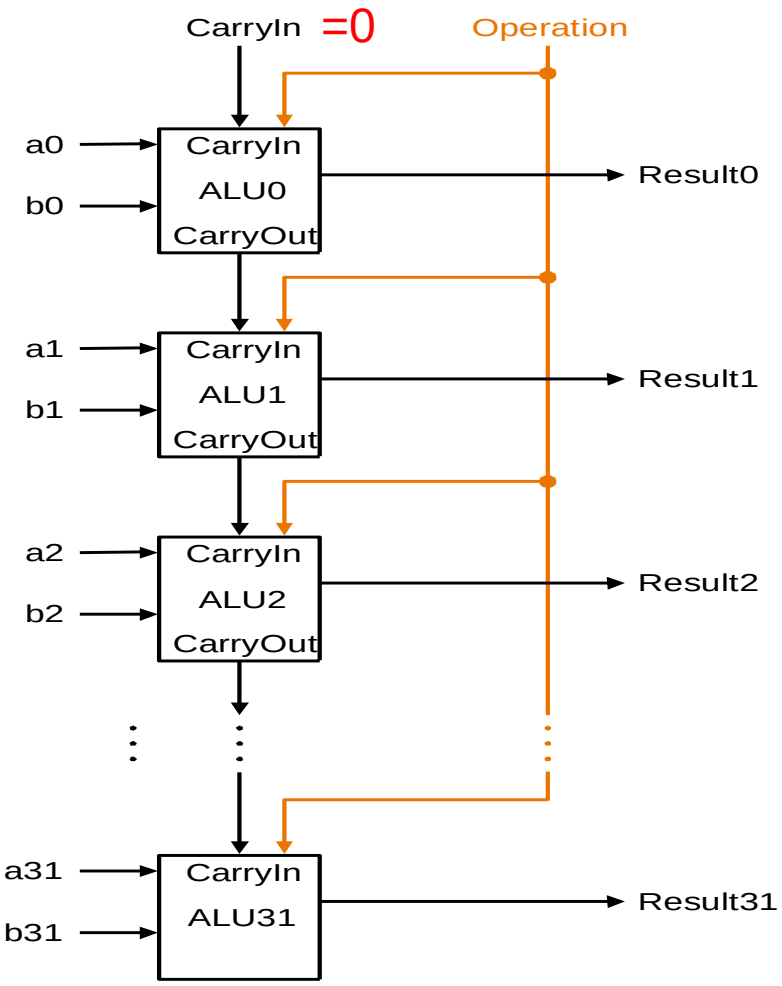
- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU

1bit ALU



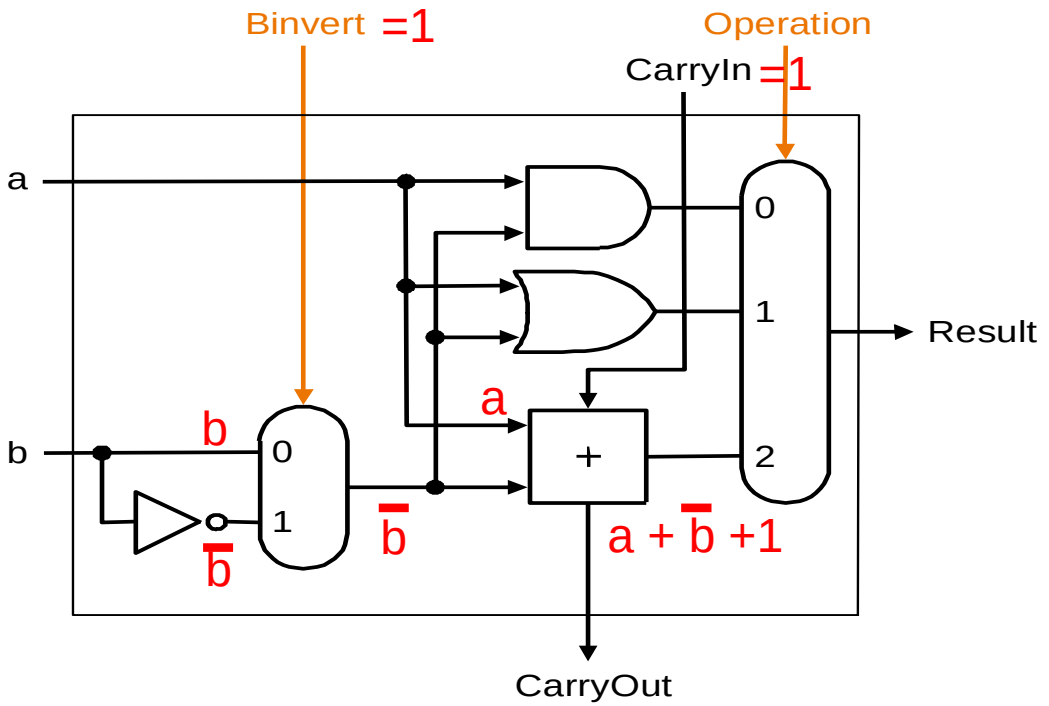
32bit ALU構成



1-bit ALUs are connected ‘’in series’’ with the carry-out of 1 box going into the carry-in of the next box

What about subtraction (a – b) ?

- **Must invert bits of ''b'' and add a 1**
  - Include an inverter
  - CarryIn for the first bit is 1
  - The CarryIn signal (for the first bit) can be the same as the Binvert signal



$$a - b = a + \bar{b} + 1$$

Execution of: AND, OR, ADD, SUB

What about NOR and NAND?

NOR演算の実現は?

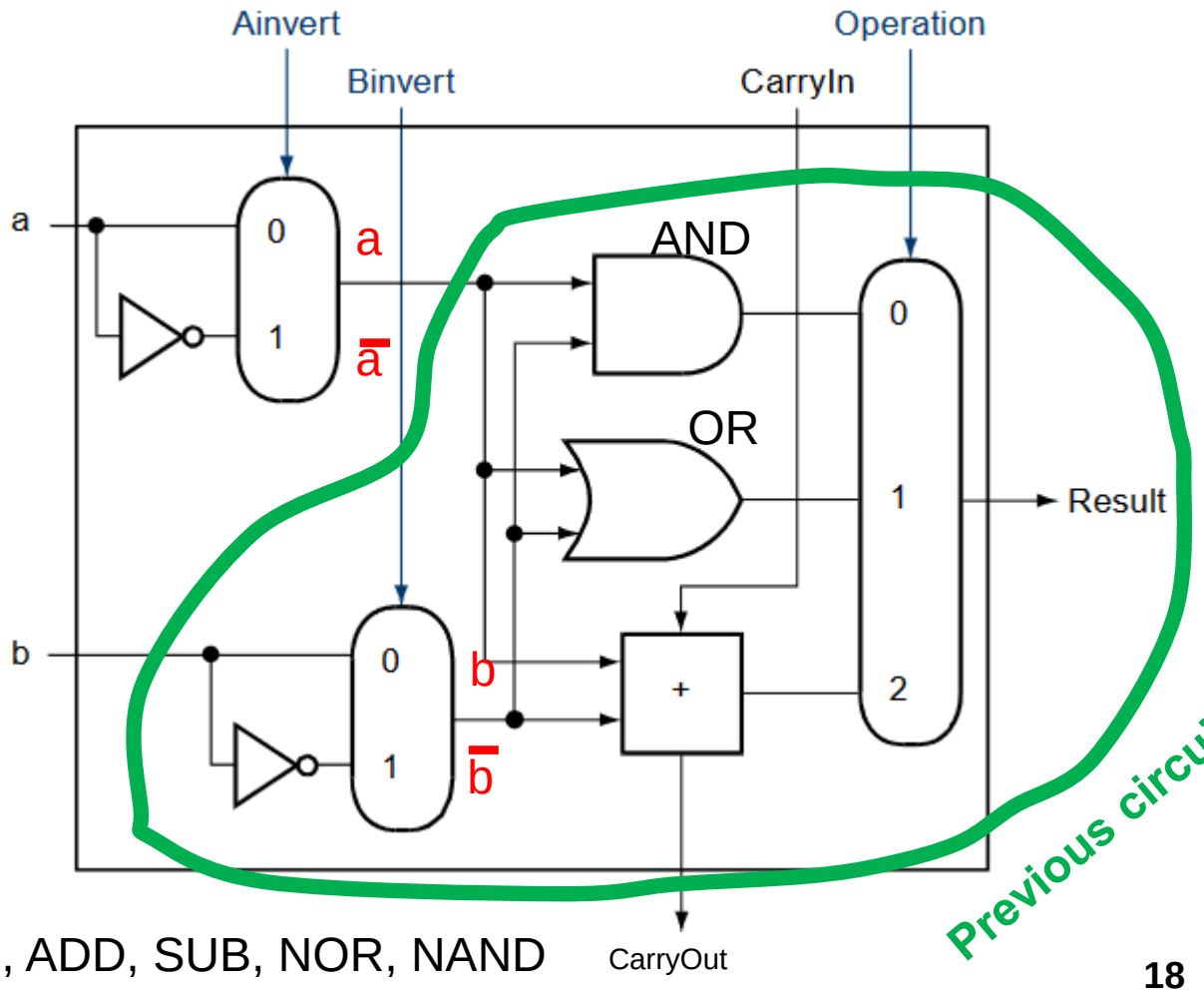
- Addition of bit inversion circuit to ‘’a’’ input

NOR

$$\overline{a \vee b} = \overline{a} \wedge \overline{b}$$

NAND

$$\overline{a \wedge b} = \overline{a} \vee \overline{b}$$



Execution of: AND, OR, ADD, SUB, NOR, NAND

Tailoring the ALU to the MIPS

32ビットALU のMIPSへの統合

- Need to support the set-on-less-than instruction (`slt`)
  - remember: `slt rd,rs,rt` 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, $t6, label`
  - jump to `label` if `$t5 = $t6`
  - use subtraction: (a-b) = 0 implies a = b

Supporting 'slt'

slt $4, $5, $6

- Perform a-b and check the sign
- New signal (Less) that is zero for ALU boxes 1-31
- The 31st box has a unit to detect overflow and sign – the sigh bit serves as the Less signal for the 0th box

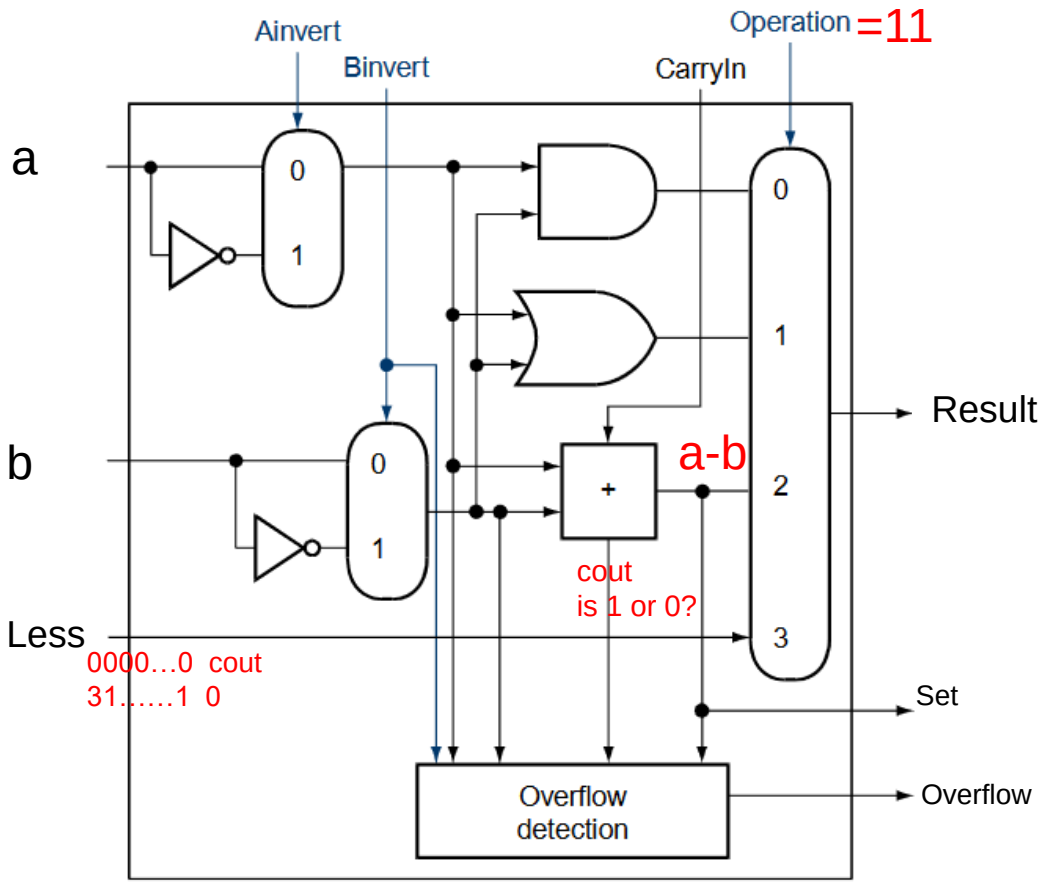


Figure B5.10

Supporting “beq”

最終版ＡＬＵ

beq $1, $2, label

- Perform a-b and confirm that the result is all zero’s
- signal Zero is a 1 when the result is zero.
- The Zero output is always calculated
- Control lines:

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

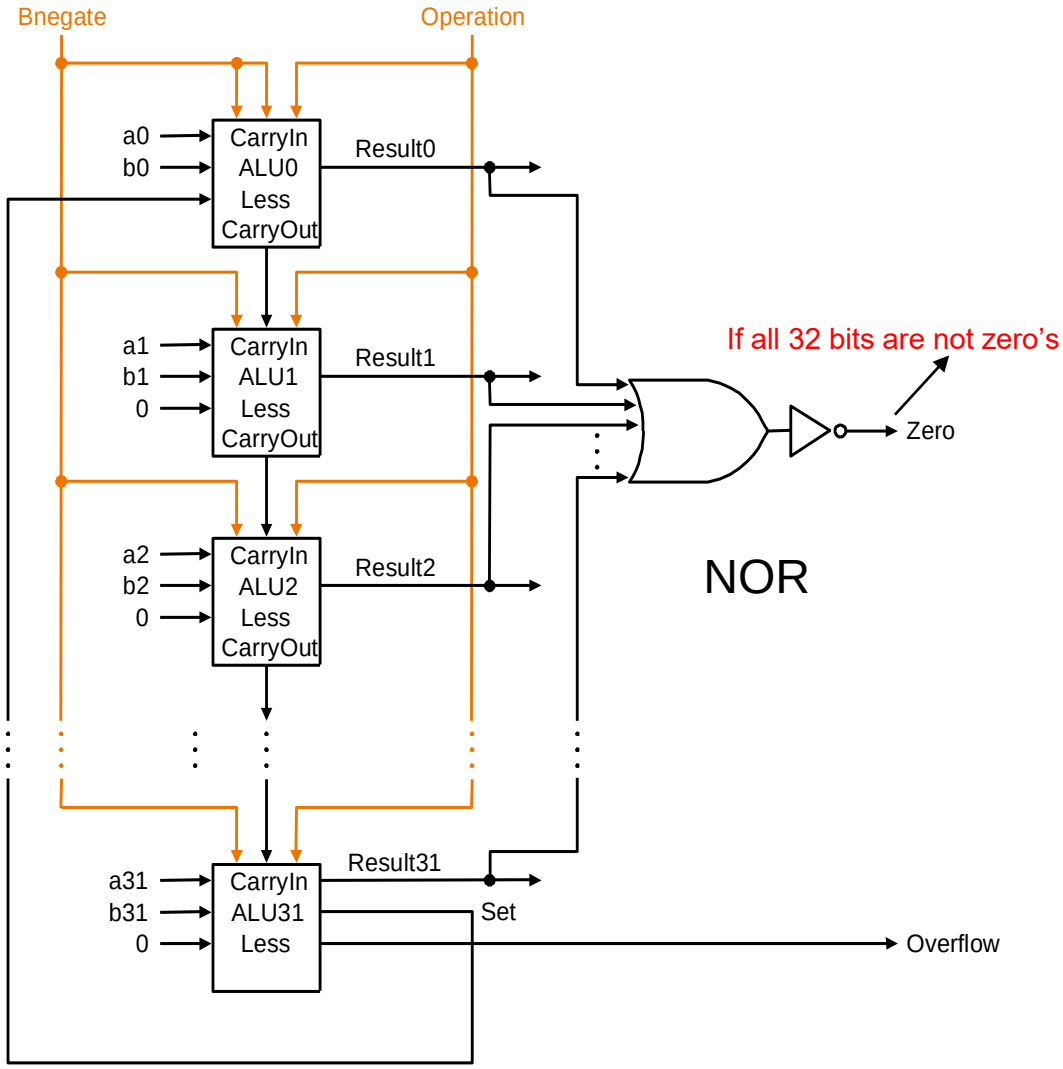
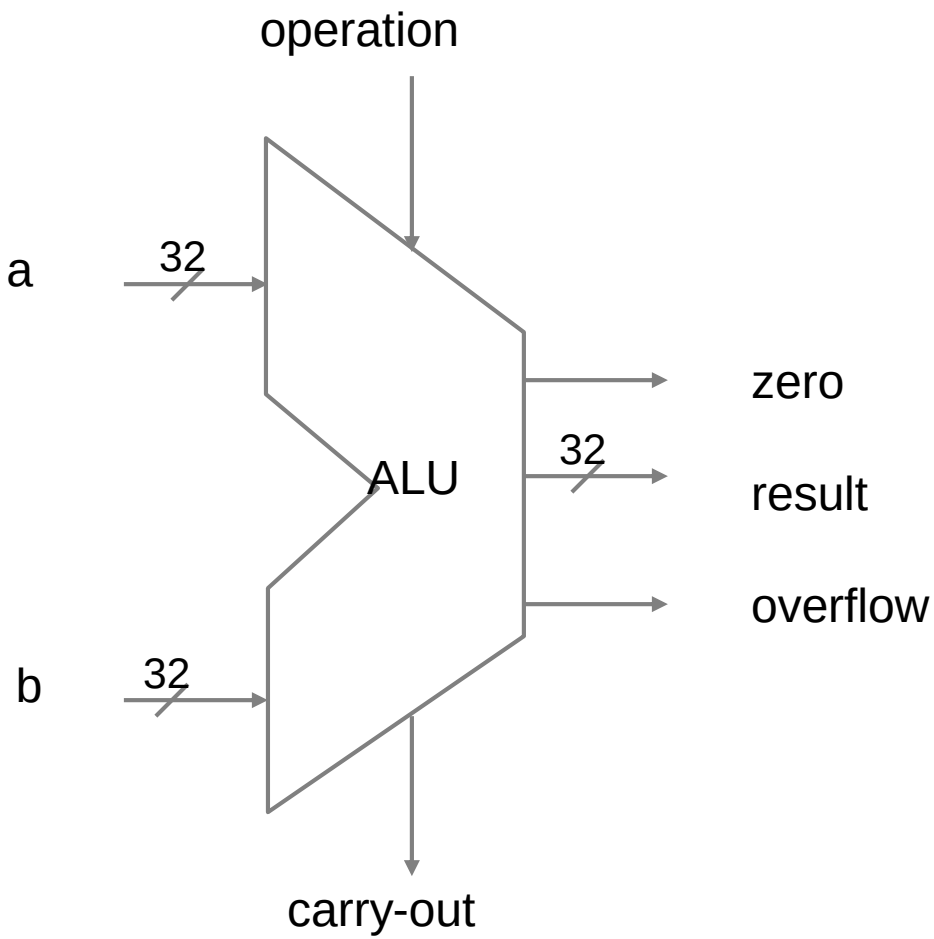


Fig. B.5.12

ALU symbol

What are the values of the control lines and what operations do they correspond to ?

	Ai	Bn	Op
AND	0	0	00
OR	0	0	01
Add	0	0	10
Sub	0	1	10
SLT	0	1	11
NOR	1	1	00
NAND	1	1	01
beq	0	1	x



Conclusions

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of connected outputs it has to drive (so-called Fan-Out)
  - the speed of a circuit is affected by the number of gates in series

Quiz

In MIPS assembly, write an assembly language version of the following C code segment:

```
int A[100], B[100];
for (i=1; i < 100; i++) {
A[i] = A[i-1] + B[i];
}
```

At the beginning of this code segment, the only values in registers are the base address of arrays A and B

in registers $a1 and $a2. Avoid the use of multiplication instructions–they are unnecessary.

Solution

The MIPS assembly sequence is as follows:

```
        li $t0, 1               # Starting index of i
        li $t5, 100             # Loop bound
loop:
        lw $t1, 0($a1)          # Load A[i-1]
        lw $t2, 4($a2)          # Load B[i]
        add $t3, $t1, $t2        # A[i-1] + B[i]
        sw $t3, 4($a1)          # A[i] = A[i-1] + B[i]
        addi $a1, 4             # Go to i+1
        addi $a2, 4             # Go to i+1
        addi $t0, 1             # Increment index variable
        bne $t0, $t5, loop      # Compare with Loop Bound
halt:
        nop
```