

MIPS arithmetic

- Today we'll review all the important ideas of arithmetic from CS231.
 - Unsigned and signed number representations.
 - Addition and subtraction with two's complement numbers.
 - Overflow detection.
- These issues are important in understanding MIPS arithmetic instructions.
- Finally, we review the construction of an ALU that will appear in our CPU designs in the next few weeks.



Unsigned numbers

- We can store unsigned numbers as their binary equivalents.
- Bit position i has a decimal weight 2^i , so the decimal value v of an n -bit unsigned binary number $a_{n-1}a_{n-2}\dots a_1a_0$ can be calculated as below.

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

- This is just the sum of each digit times the weight of its position.

$$1\ 0\ 1\ 1\ 0\ 1_2 = (2^5 \cdot 1) + (2^4 \cdot 0) + (2^3 \cdot 1) + (2^2 \cdot 1) + (2^1 \cdot 0) + (2^0 \cdot 1) = 45_{10}$$

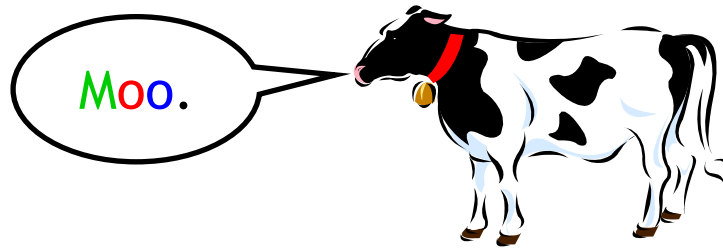
- The smallest and largest possible numbers are 0 and 2^n-1 .
- In MIPS, the largest unsigned number is $2^{32}-1$, or about 4.3 billion.

Hexadecimal notation

- Hexadecimal is frequently used as a shorthand for binary numbers, because one hex digit corresponds to four bits.
- Converting between binary and hex is easy with a table.

$$\text{B E E F}_{16} = 1011\ 1110\ 1110\ 1111_2$$

- In SPIM and many high-level programming languages, the prefix `0x` denotes a hexadecimal constant, as in `0xBEEF`.



Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Logical shifts in MIPS

- MIPS has `sll` (shift left logical) and `srl` (shift right logical) instructions.
 - A constant specifies the number of bits to shift, from 0 to 31.
 - 0s are shifted into the right or left sides, respectively.
- For example, assume that `$t0` contains `0xFEEDBEEF`.

```
sll $t1, $t0, 4      # $t1 contains 0xEEDBEEF0
srl $t2, $t0, 12     # $t2 contains 0x000FEEDB
```

- Shifts are actually R-type instructions, not I-type!



- The constant is stored in the 5-bit “shamt” field that we hadn’t seen up until now.

Unsigned overflow

- One recurring issue in computer arithmetic is dealing with finite amounts of storage, such as 32-bit MIPS registers.
- **Overflow** occurs when the result of an operation is too large to be stored.
- There are many examples of unsigned n -bit overflows.
 - When there is a carry out of position $n-1$ in an addition.
 - When a_{n-1} is 1 in a multiplication by two.
- The 6-bit addition and multiplication operations on page 4 both result in overflow, since the correct answers require 7 bits.



Signed two's-complement numbers

- Signed numbers are represented in **two's complement** format.
- The most significant bit a_{n-1} of each number is a **sign bit**.
 - **0** indicates a positive number.
 - **1** indicates a negative number.
- The **range** of n -bit signed numbers is from -2^{n-1} to $+2^{n-1}-1$
 - For 32-bit values, the range is roughly ± 2.15 billion.
 - The ranges are not exactly even since there are 2^{n-1} negative numbers, but just $2^{n-1}-1$ positive numbers.



Two's-complement values

- The magnitude of a negative number is represented in a funny way that makes $x + -x = 2^n$ for all numbers x .
- The decimal value v of a two's-complement number $a_{n-1}a_{n-2}\dots a_1a_0$ is:

$$v = \left(-2^{n-1} \cdot a_{n-1}\right) + \left(\sum_{i=0}^{n-2} 2^i \cdot a_i\right)$$

↖
This term accounts
for the sign.

↙
The magnitude is
computed like before.

- Here are two examples with $n = 6$.

$$010111 = (-2^5 \cdot 0) + (2^4 \cdot 1) + (2^3 \cdot 0) + (2^2 \cdot 1) + (2^1 \cdot 1) + (2^0 \cdot 1) = 23$$

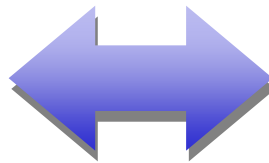
$$101101 = (-2^5 \cdot 1) + (2^4 \cdot 0) + (2^3 \cdot 1) + (2^2 \cdot 1) + (2^1 \cdot 0) + (2^0 \cdot 1) = -19$$

Negating a two's complement number

- There are three main ways to negate two's complement numbers. As an example, let's consider the six-bit value **101101** (-19).
 1. Complement all the bits in the number and then add 1.
 - Complementing the bits in 101101 yields 010010.
 - Adding 1 results in 010011 (+19).
 2. Complement the bits to the left of the rightmost 1.
 - The rightmost 1 in 10110**1** is in position 0.
 - Complementing the bits to its left gives 01001**1** (+19).
 3. Subtract the number from 2^n .
 - 2^6 is 1000000.
 - Then, $1000000 - 101101 = 010011$ (+19).

Two's-complement numbers with $n=4$

Decimal	2C
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111



2C	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Signed overflow

- It's also possible for two's-complement operations to overflow.
- Overflow occurs only when we add two numbers of the *same* sign and get a *different* sign for the result.
 - Adding two positive numbers should result in a positive number.
 - Adding two negative numbers should yield a negative result.

		1	0	1	1	1		
		0	1	0	0	1	1	+19
+		0	1	0	1	1	1	+ 23
	0	1	0	1	0	1	0	- 22

		0	1	0	0	1		
		1	0	1	1	0	1	- 19
+		1	0	1	0	0	1	+ - 23
	1	0	1	0	1	1	0	+ 22

Ariane 5

- In 1996, the European Space Agency's Ariane 5 rocket was launched for the first time... and it exploded 40 seconds after liftoff.
- It turns out the Ariane 5 used software designed for the older Ariane 4.
 - The Ariane 4 stored its horizontal velocity as a 16-bit signed integer.
 - But the Ariane 5 reaches a much higher velocity, which caused an overflow in the 16-bit quantity.
- The overflow error was never caught, so incorrect instructions were sent to the rocket boosters and main engine.



Sign extension

- We often work with signed numbers of different lengths.
 - Instructions like `lw $t0, -4($sp)` add 16-bit and 32-bit values together.
 - `slti $t0, $t0, 256` compares a 16-bit constant to a 32-bit number.
 - The `lb` instruction copies an 8-bit value into a 32-bit register.
- You can **sign extend** a two's complement number by copying the sign bit.
- For instance, we can sign extend 6-bit numbers to 8 bits.

(+23) 010111 → 00010111 (+23)

(-19) 101101 → 11101101 (-19)

- If you add 0s instead of sign extending, you may accidentally change the sign, and hence the value, of the result.

(-19) 101101 → 00101101 (+45)



Interpreting bit patterns

- One of the most important things to remember is that bit patterns have different meanings under different representations!
 - As a six-bit *unsigned* number, 101101 denotes 45 in decimal.
 - But as a *two's complement* number, 101101 denotes -19.
- The example C program below prints the same data (0xFFFFFFFF) twice, but under different interpretations.

```
main()                // CSIL Sun machines
{
    int x = 0xFFFFFFFF; // 32-bit integers
    printf("x = %d\n", x); // Signed; prints -1
    printf("x = %u\n", x); // Unsigned; prints 4294967295
}
```

Signed and unsigned comparisons

- Specifying the interpretation of binary numbers is especially important in untyped assembly language programming.
- For example, how does 111111 compare to 000000?
 - As an unsigned number, 111111 = 63, which is *greater* than 0.
 - As a two's complement number, 111111 = -1, which is *less* than 0.
- MIPS includes two versions of the comparison instructions.
 - The `slt` and `slti` instructions that we saw do signed comparisons.
 - The instructions `sltu` and `sltiu` perform unsigned comparisons.

```
main()
{
    unsigned int u = 0xFFFFFFFF;
    int s = 0xFFFFFFFF;

    printf("%u is %s than 0\n", u, (u < 0 ? "less" : "greater"));
    printf("%d is %s than 0\n", s, (s < 0 ? "less" : "greater"));
}
```


Signed and unsigned addition

- MIPS provides two versions of the main arithmetic instructions.
 - The `add`, `addi` and `sub` instructions will raise an exception if they cause an overflow.
 - But `addu`, `addiu`, and `subu` do not raise exceptions on overflow
- Be careful! Both `addi` and `addiu` sign extend their constant field, even though `addiu` is considered an “unsigned” operation.
- In C, overflow exceptions may or may not be raised, depending on the underlying hardware.

```
main()                // CSIL Sun machines
{
    int x = 0x7FFFFFFF; // Largest 32-bit number
    printf("x = %d\n", x); // Prints 2147483647
    x++;
    printf("x = %d\n", x); // Prints -2147483648
}
```

When to use unsigned numbers

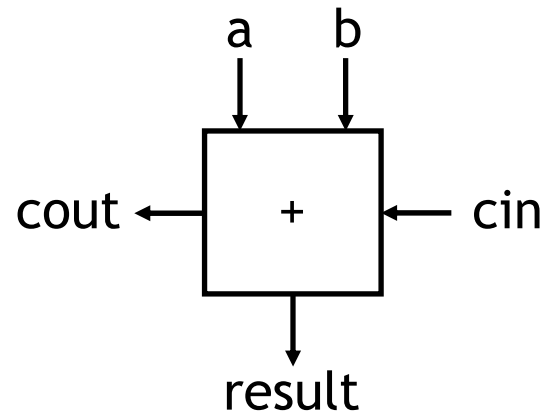
- Memory addresses should be treated as unsigned numbers.
 - The fourth element of an array that starts at 0x7FFFFFF0 is at address 0x80000000. This address computation should not overflow!
 - In fact, stack pushes and pops should be performed using the unsigned add and subtract instructions like `addiu`, and not `addi`.
- Non-numeric data should also be handled using unsigned operations.
 - We can represent up to 32 Boolean values with one MIPS word.
 - Similarly, several fields of data can be packed into one word, such as the encoding of MIPS instructions themselves!
- The `and` and `or` instructions do bitwise manipulations on registers. The immediate variants, `andi` and `ori`, do not sign extend their constants.

MIPS arithmetic instruction summary

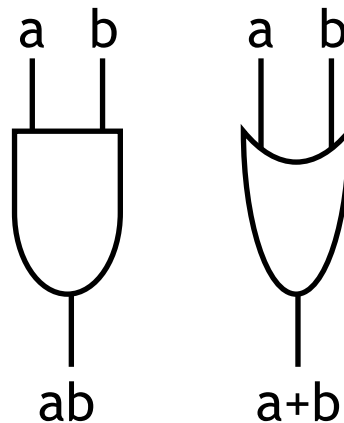
- Addition and subtraction
 - `add`, `addi`, and `sub` raise exceptions on overflow.
 - `addu`, `addiu`, and `subu` do not raise exceptions on overflow.
 - `addi` and `addiu` both sign extend their immediate fields.
- Comparisons
 - `slt` and `slti` are for signed comparisons.
 - `sltu` and `sltiu` do unsigned comparisons.
 - `slti` and `sltiu` both sign extend their immediate fields.
- Data transfers
 - `lb` sign extends the byte that is loaded into a register.
 - `lbu` does not sign extend the byte.
- Shifting
 - `sll` and `srl` are used for logical shifts of up to 31 bits.
- Bitwise operators
 - `andi` and `ori` do not sign extend their immediate field.

Hardware for single-bit operations

- We can use a full adder to add data inputs a , b and a carry in cin . This produces a one-bit $result$ and a carry out $cout$.



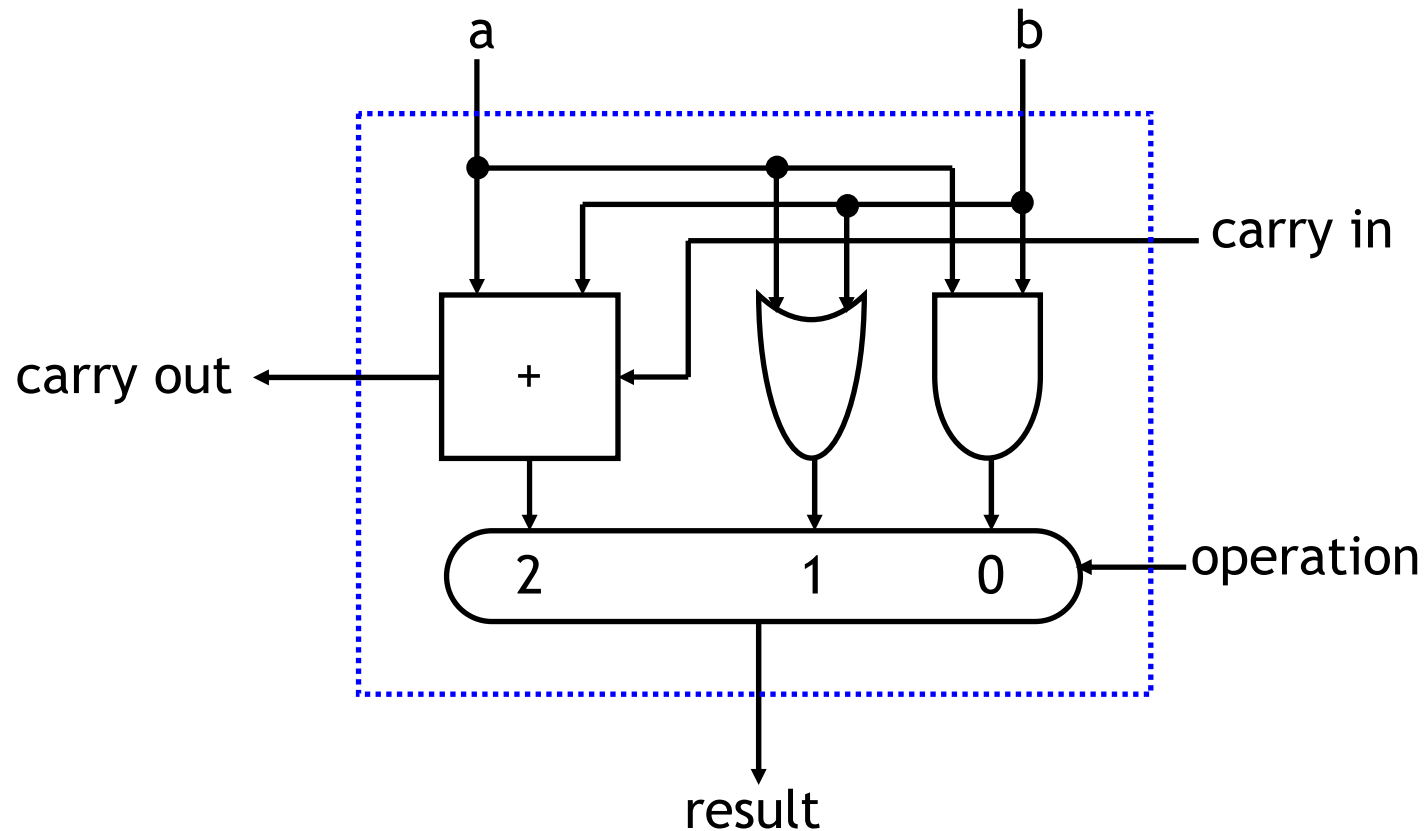
- It's also easy to do one bit and and or operations.



A one-bit ALU

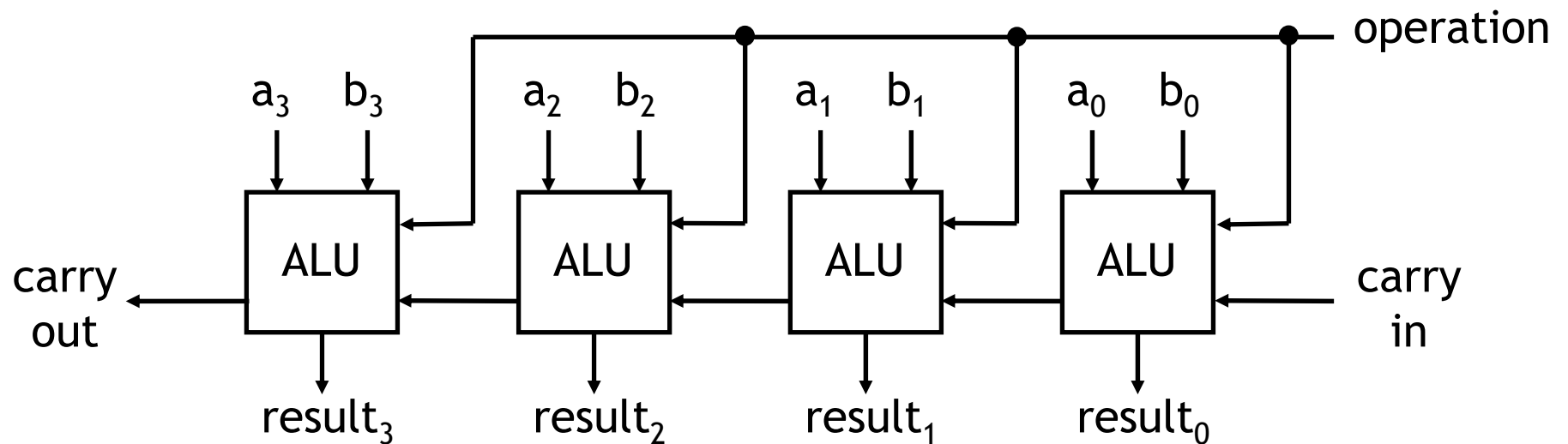
- A simple one-bit ALU can support all three of these functions.
- A two-bit input called **operation** selects the desired function using a 3-to-1 multiplexer.

operation	result
0	a and b
1	a or b
2	a + b + carry in



An n -bit ALU

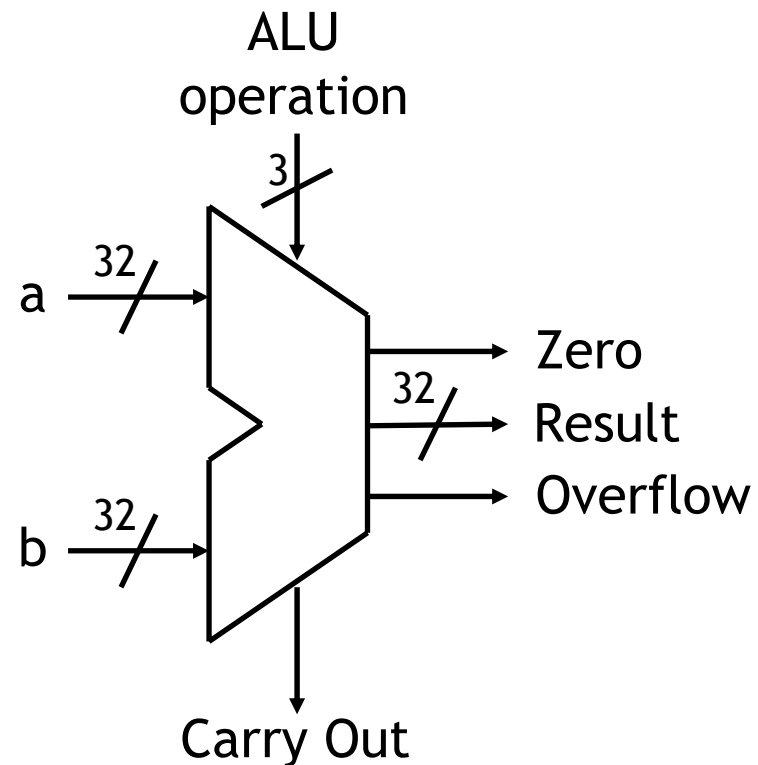
- This one-bit ALU can be replicated to produce an n -bit ALU.
 - The **carry out** from stage i connects to the **carry in** of stage $i+1$.
 - The **operation** input is shared by all of the one-bit circuits.



A whole MIPS ALU

- We can do the **subtraction** of $a - b$ by complementing the b input, setting **carry in** to 1, and performing an addition.
- The ALU also has a **slt** function, which sets **result** to 1 or 0 depending on whether or not a is less than b . This can be determined via subtraction.
- Finally, outputs can be generated to signal overflow or a result of zero.

ALU operation	Result
000	a and b
001	a or b
010	$a + b$
110	$a - b$
111	$a < b$



MIPS ALU design

- ALUs are a good example of modular hardware design.
 - 32 full adders can be connected together to form a 32-bit adder.
 - Bitwise operations can also be handled one bit at a time.
 - Adders can also be used to perform subtraction.
- Shift operations are usually handled outside the ALU by a combinational “barrel shifter” circuit, which can shift an arbitrary number of positions in constant time.



Summary

- Signed numbers are represented using the **two's complement** format.
 - A **sign bit** distinguishes between positive and negative numbers.
 - Numbers are negated by complementing their bits and adding 1.
 - This makes subtraction hardware easy to build.
- Signed and unsigned numbers have many important differences.
 - Some operations require that we **sign extend** the operands.
 - **Overflow** detection differs for signed and unsigned operations.
- The MIPS instruction set includes both signed and unsigned operations, which affect how constant operands and overflows are handled.