# NUMBER OPERATIONS

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

# Overview

- Homework 4 is due tonight
  - Verify your uploaded file before the deadline
- This lecture
  - Number representations and operations

# Binary Representation

- The binary number

  11011000 00010101 00101110 11100111

  Most significant bit        Least significant bit

- The number quantity (decimal)

  $$1 \times 2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0} = 3625266919$$

- A 32-bit word can represent $2^{32}$ numbers between 0 and $2^{32}$-1 (4,294,967,295)
  - Represent only positive numbers
  - Also known as the unsigned representation

# Negative Numbers

- The binary number

  1 1011000 00010101 00101110 11100111

  Sign bit

- Sign-magnitude representation
  - 1. Quantify the magnitude (31 bits)

    $1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0} = 1477783271$

  - 2. Determine the sign based in the sign bit

    -1477783271

- Example: 3-bit sing-magnitude
  - How many numbers
  - How to do arithmetic

# Negative Numbers

- The binary number

  `11011000 00010101 00101110 11100111`

  Sign bit

- 1's complement: -x is represented by inverting x's bits
  - 1. Invert the bits if the sign bit is set

    `00100111 11101010 11010001 00011000`

  - 2. Quantify the magnitude (31 bits)

    $-1 \times (1 \times 2^{29} + \ldots + 0 \times 2^{0}) = -669700376$

- Example: 3-bit 1's complement
  - How many numbers
  - How to do arithmetic

# Negative Numbers

- The binary number

  11011000 00010101 00101110 11100111

  Sign bit

- Sign-magnitude and 1's complement are not favorable
  - Relatively complex implementation of arithmetic operations

- A 32-bit word represents $2^{32}$-1 numbers between $-2^{31}+1$ and $+2^{31}-1$
  - Two different representations for zero

# Negative Numbers

- The binary number

  11011000 00010101 00101110 11100111

  Sign bit

- 2's complement representation
  - Give the sign bit a negative weight

$$1 \times -2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0} = -669700377$$

- Example: 3-bit 2's complement
  - How many numbers
  - How to do arithmetic

# Negative Numbers

- The binary number

  11011000 00010101 00101110 11100111

  Sign bit
- 2's complement representation
  - Give the sign bit a negative weight

  $1 \times -2^{31} + 1 \times 2^{30} + 0 \times 2^{29} + \ldots + 1 \times 2^{0} = -669700377$

- A 32-bit word represents $2^{32}$ numbers between $-2^{31}$ and $+2^{31}-1$.
  - No repeated numbers and simple arithmetic implementation

# Example: 2’s Complement

- Compute the 32-bit 2’s complement representations for the following decimal numbers:
  - 5, -5, -6

# Example: 2's Complement

- Compute the 32-bit 2's complement representations for the following decimal numbers:
  - 5, -5, -6
- Given -5, verify that negating and adding 1 yields the number 5
-

| | |
|---|---|
| 5: | 0000 0000 0000 0000 0000 0000 0000 0101 |
| -5: | 1111 1111 1111 1111 1111 1111 1111 1011 |
| -6: | 1111 1111 1111 1111 1111 1111 1111 1010 |

# Example

- All 32-bit 2's complement representations

```
int num = 0;
do {
  num++;
} while(num != 0);
```

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2^{31}-1$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2^{31}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -(2^{31} - 1)$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -(2^{31} - 2)$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1$

# Signed and Unsigned

- The hardware recognizes two formats:

- Unsigned
  - All numbers are positive, a 1 in the most significant bit just means it is a really large number
  - Example: the unsigned int declaration in C/C++

- Signed
  - Numbers can be +/- , a 1 in the MSB means the number is negative
  - Example: the signed int or int declaration in C/C++

- Why would I need both?
  - To represent twice as many numbers when we're sure that we don't need negatives

# Example: MIPS Instructions

- Example: consider a comparison instruction
  - `slt   $t0, $t1, $zero`
- and $t1 contains the 32-bit number
  - 11110111 11001010 00010100 00011110
- What gets stored in $t0?

# Example: MIPS Instructions

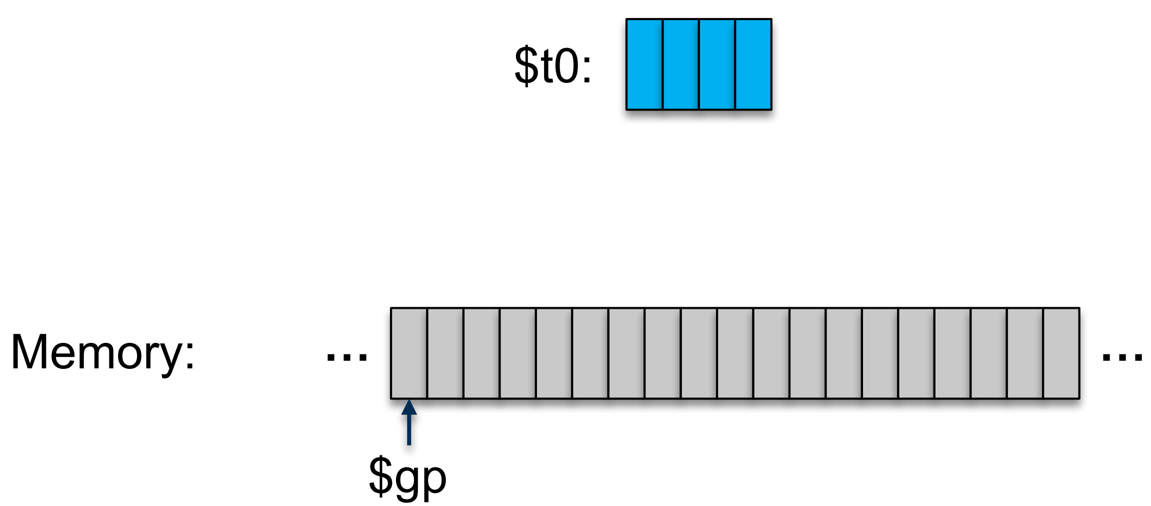
- Example: consider a comparison instruction
  - `slt    $t0, $t1, $zero`
- and $t1 contains the 32-bit number
  - 11110111 11001010 00010100 00011110
- What gets stored in $t0?

whether $t1 is a signed or unsigned number
the compiler/programmer must track this and accordingly use either slt or sltu

```
slt    $t0, $t1, $zero    #stores  1 in $t0
sltu   $t0, $t1, $zero    #stores  0 in $t0
```

# Recall: Dealing with Characters

- Instructions are also provided to deal with byte-sized and half-word quantities: lb (load-byte), sb, lh, sh
- Example: loading a byte from memory
  - Is the byte signed or unsigned?



# Sign Extension

- Signed 8-/16-bit numbers must be converted into 32-bit signed numbers
  - Example:
    - `addi  $s0, $zero, 0x8000`
    - `addi  $s0, $zero, 0x4000`
- Conversion: take the most significant bit and use it to fill up the additional bits on the left

```
11111111 11111111 10000000 00000000 = -32768

00000000 00000000 01000000 00000000 = 16384
```

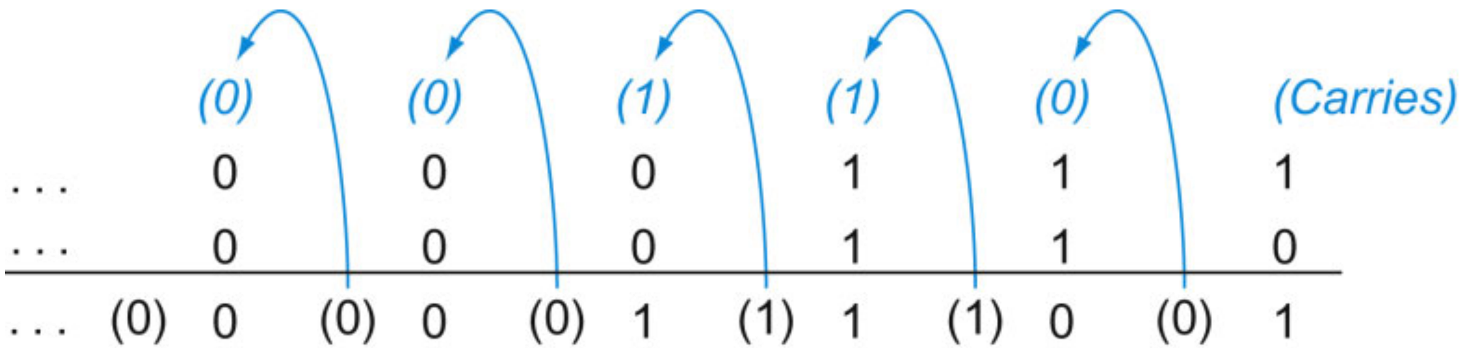
# Unsigned Conversion

- Unsigned 8-/16-bit numbers must be converted into 32-bit signed numbers
  - Example:
    - `addiu $s0, $zero, 0x8000`
    - `addiu $s0, $zero, 0x4000`
- Conversion: fill up the additional bits on the left with zeroes

```
00000000 00000000 10000000 00000000 = 32768
00000000 00000000 01000000 00000000 = 16384
```

# Addition and Subtraction

- Addition is similar to decimal arithmetic




- For subtraction, simply add the negative number
  - 4-bit example: 6 – 5 = 6 + (-5)

```
   0 1 1 0
+  1 0 1 1
```

# Overflows

- Note: machines have limited number of buts for representing each number

- For an unsigned number, overflow happens when the last carry (1) cannot be accommodated

- For a signed number, overflow happens when the most significant bit is not the same as every bit to its left
  - when the sum of two positive numbers is a negative result
  - when the sum of two negative numbers is a positive result
  - The sum of a positive and negative number will never overflow

# MIPS Instructions

- Instructions add, addi, sub may cause exceptions on overflow
  - Software needs to handle exceptions

- MIPS allows addu and subu instructions that work with unsigned integers and never flag an overflow – to detect the overflow, other instructions will have to be executed

# Multiplication Example

| | |
|---|---|
| Multiplicand | $1000_{ten}$ |
| Multiplier | x $1001_{ten}$ |

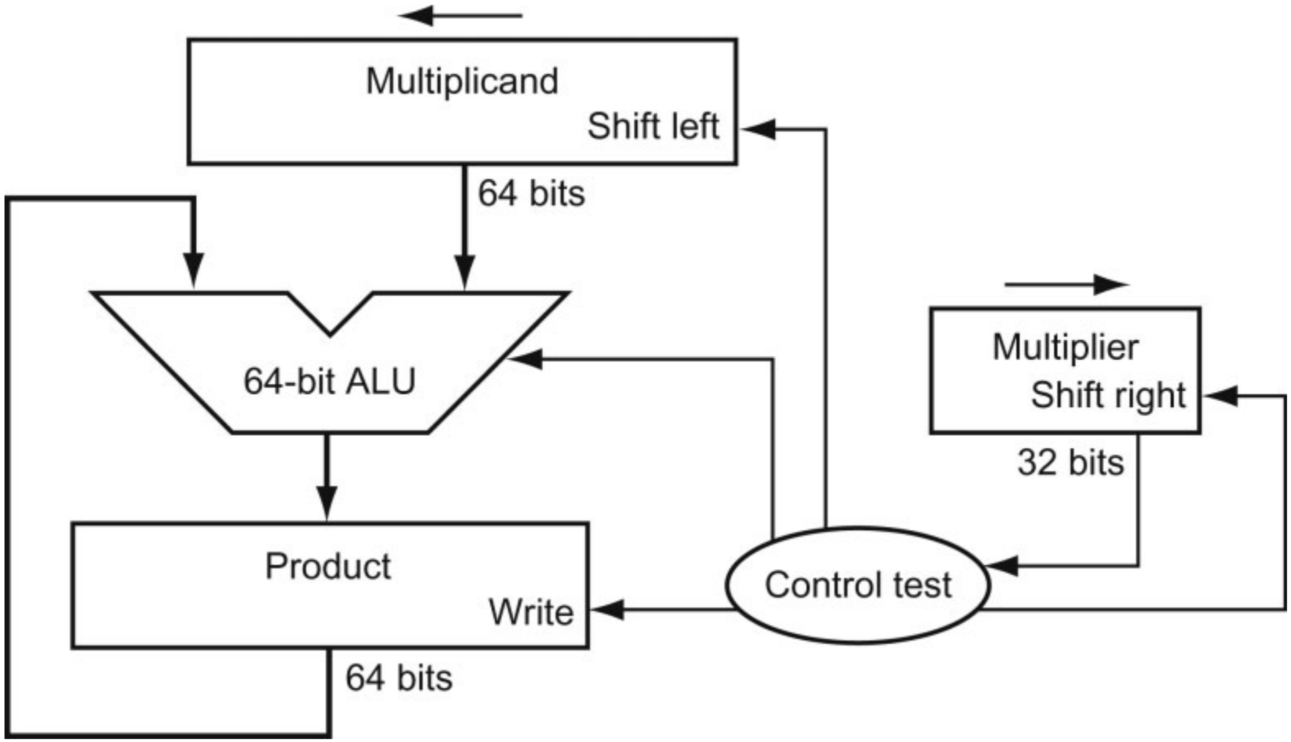
```
        1000
       0000
      0000
     1000
```

| | |
|---|---|
| Product | $1001000_{ten}$ |

In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

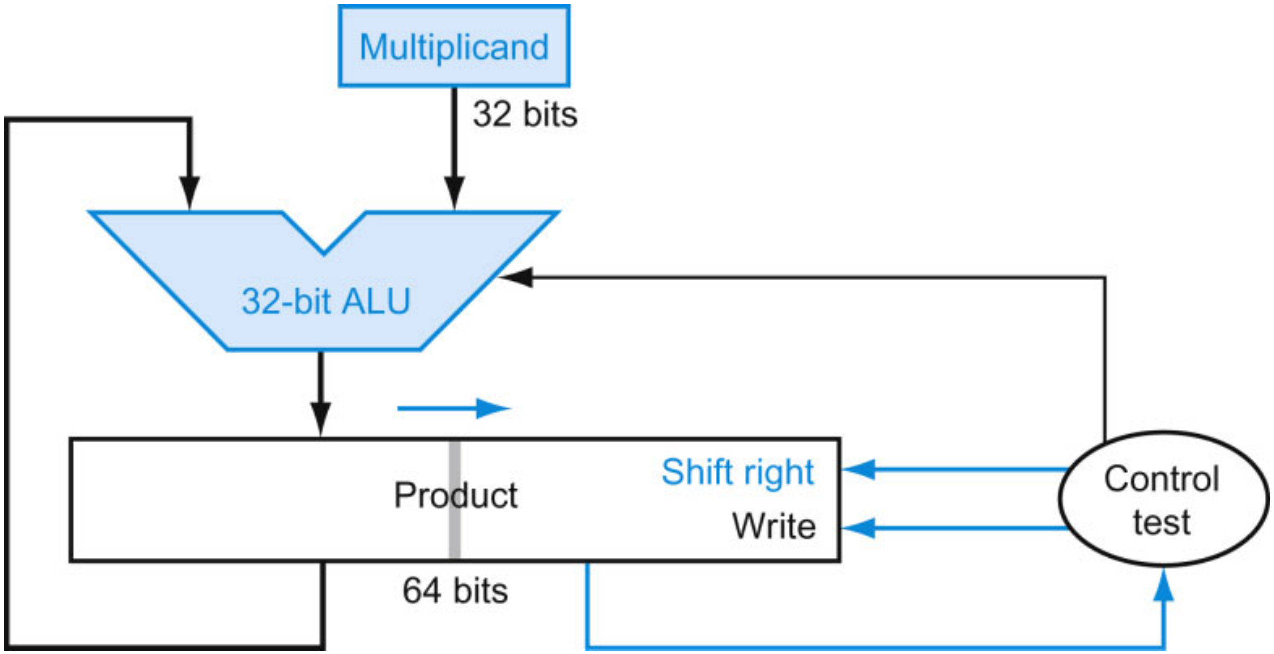
# Multiplication Algorithm 1




In every step

- multiplicand is shifted
- next bit of multiplier is examined (also a shifting step)
- if this bit is 1, shifted multiplicand is added to the product

# Multiplication Algorithm 2




- 32-bit ALU and multiplicand is untouched
- the sum keeps shifting right
- at every step, number of bits in product + multiplier = 64, hence, they share a single 64-bit register

# Multiplication Notes

- The previous algorithm also works for signed numbers (negative numbers in 2's complement form)
- We can also convert negative numbers to positive, multiply the magnitudes, and convert to negative if signs disagree
- The product of two 32-bit numbers can be a 64-bit number -- hence, in MIPS, the product is saved in two 32-bit registers

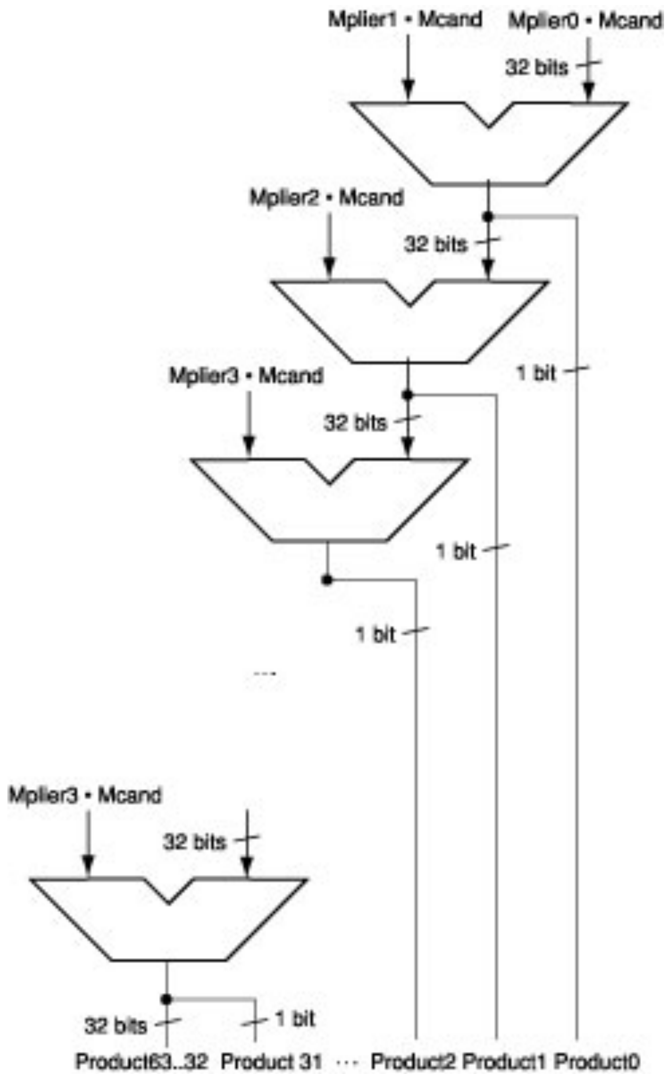
# MIPS Instructions

- mult $s2, $s3 computes the product and stores
- it in two "internal" registers that
- can be referred to as hi and lo

- mfhi $s0 moves the value in hi into $s0
- mflo $s1 moves the value in lo into $s1

- Similarly for multu

# Fast Algorithm




- The previous algorithm requires a clock to ensure that the earlier addition has completed before shifting

- This algorithm can quickly set up most inputs – it then has to wait for the result of each add to propagate down – faster because no clock is involved

  -- Note: high transistor cost

Source: H&P textbook

# Division

| | | | |
|---|---|---|---|
| | | $1001_{ten}$ | Quotient |
| Divisor | $1000_{ten}$ | $1001010_{ten}$ | Dividend |
| | | -1000 | |
| | | 10 | |
| | | 101 | |
| | | 1010 | |
| | | -1000 | |
| | | $10_{ten}$ | Remainder |

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

# Division

| | | | | |
|---|---|---|---|---|
| | | $1001_{ten}$ | Quotient | |
| Divisor | $1000_{ten}$ | $1001010_{ten}$ | Dividend | |

| | | | |
|---|---|---|---|
| 0001001010 | 0001001010 | 0000001010 | 0000001010 |
| 100000000000 → | 0001000000 → | 0000100000 → | 0000001000 |
| Quo: 0 | 000001 | 0000010 | 000001001 |

At every step,

- shift divisor right and compare it with current dividend
- if divisor is larger, shift 0 as the next bit of the quotient
- if divisor is smaller, subtract to get new dividend and shift 1 as the next bit of the quotient

# Divide Example

- Divide $7_{ten}$ ($0000\ 0111_{two}$) by $2_{ten}$ ($0010_{two}$)

| Iter | Step | Quot | Divisor | Remainder |
|---|---|---|---|---|
| 0 | Initial values | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |

# Divide Example

- Divide $7_{ten}$ ($0000\ 0111_{two}$) by $2_{ten}$ ($0010_{two}$)

| Iter | Step | Quot | Divisor | Remainder |
|---|---|---|---|---|
| 0 | Initial values | 0000 | 0010 0000 | 0000 0111 |
| 1 | Rem = Rem – Div | 0000 | 0010 0000 | 1110 0111 |
| | Rem < 0 → +Div, shift 0 into Q | 0000 | 0010 0000 | 0000 0111 |
| | Shift Div right | 0000 | 0001 0000 | 0000 0111 |
| 2 | Same steps as 1 | 0000 | 0001 0000 | 1111 0111 |
| | | 0000 | 0001 0000 | 0000 0111 |
| | | 0000 | 0000 1000 | 0000 0111 |
| 3 | Same steps as 1 | 0000 | 0000 0100 | 0000 0111 |
| 4 | Rem = Rem – Div | 0000 | 0000 0100 | 0000 0011 |
| | Rem >= 0 → shift 1 into Q | 0001 | 0000 0100 | 0000 0011 |
| | Shift Div right | 0001 | 0000 0010 | 0000 0011 |
| 5 | Same steps as 4 | 0011 | 0000 0001 | 0000 0001 |