

CHAPTER 3

ARITHMETIC FOR COMPUTERS

1

Introduction

- Numbers are represented in binary form within a computer.
- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if a computer operation produces a number bigger than what can be represented in a word?
- What about fractions and other real numbers?
- How does hardware add, subtract, multiply and divide numbers that are inherently stored in binary form?
- The goal of this chapter is to explain how computer hardware deals with these issues.

2

Converting between positive decimal and binary

- Convert decimal **49** to 8-bit binary:

$49/2 = 24 \text{ r } 1$
 $24/2 = 12 \text{ r } 0$
 $12/2 = 6 \text{ r } 0$
 $6/2 = 3 \text{ r } 0$
 $3/2 = 1 \text{ r } 1$
 $1/2 = 0 \text{ r } 1$

2	49	1
	24	0
	12	0
	6	0
	3	1
	1	

Now read the remainders from bottom to top: the binary equivalent is **110001**.
 In 8 bit form, it is: **0011 0001**. In 16 bit form it is: **0000 0000 0011 0001**

- It is very easy to convert from a binary number to a decimal number. Just like the decimal system, we multiply each digit by its weighted position, and add each of the weighted values together. For example, the binary value **0100 1010** represents:

$$0100\ 1010 = 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$= 64 + 8 + 2 = 74$$

$$\begin{array}{cccc}
 0 & 1 & 0 & 1 \\
 \uparrow & & \uparrow & \uparrow \\
 2^6 & & 2^3 & 2^1
 \end{array}
 0100\ 1010 = 64 + 8 + 2 = 74$$

5

Signed Integers

- To represent signed integers, use the two's complement system.
 - Splits the binary pattern into two halves: for positive and negatives.
 - For example taking the 3-bit patterns

Decimal	0	1	2	3	-4	-3	-2	-1
Binary	000	001	010	011	100	101	110	111

- The digits in each base "increase" as you go to the right except for one jump (from negatives to positives).
- The 4 bit patterns are

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

- The largest positive 32bit signed bit pattern is $01111\dots11111_{\text{two}} = 2^{31}-1_{\text{ten}}$.
- The smallest negative 32bit signed bit pattern is $1000\dots00000_{\text{two}} = -2^{31}_{\text{ten}}$.
- The value of $-1_{\text{ten}} = 11111\dots11111_{\text{two}}$ and $0_{\text{ten}} = 0000\dots00000_{\text{two}}$.

6

Signed integers in MIPS

- 32 bit signed integers:

```

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

```

7

Converting between decimal and signed binary

- Assume that x is a binary number and \bar{x} is its complement.
- Then using binary addition: $x + \bar{x} = 1111 \dots 1111 = -1_{ten}$
- Thus $x + \bar{x} = -1_{ten}$

$$\bar{x} + 1_{ten} = -x$$

$$-x = \bar{x} + 1_{ten}$$

- To get $-x$ (negate x) a shortcut is to

- Invert the bits
- Add 1

- Example:** Negate 2_{ten} in MIPS and check the result by negating -2_{ten} in MIPS.

$$2_{ten} = 0000\dots00010_{two}$$

$$\text{Invert } 1111\dots11101_{two}$$

$$+ \quad \quad \quad 1_{two}$$

$$-2_{ten} = 1111\dots11110_{two}$$

$$-2_{ten} = 1111\dots11110_{two}$$

$$\text{Invert } 0000\dots00001_{two}$$

$$+ \quad \quad \quad 1_{two}$$

$$2_{ten} = 0000\dots00010_{two}$$

8

Converting between decimal and signed binary

- Convert decimal **-50** to 8-bit signed binary.
 - Convert 50 to signed binary:

2	50	0
	25	1
	12	0
	6	0
	3	1
	1	

Now read the remainders from bottom upwards $50 = 110010$

- Extend to 8 bits: **0011 0010**
 - Invert bits: 1100 1100
 - Add 1: $-50 = 1100 1101$
- } Get 2's compliment
- Answer is **1100 1101**
- Convert the signed binary 1101 1011 to decimal.
 - Invert bits: 0010 0100
 - Add 1: 0010 0101
 - Convert to decimal: $2^0 + 2^2 + 2^5 = 1 + 4 + 32 = 37$ Final answer = -37. 9

Converting between decimal and signed binary

- We know that negating twice has no effect on a number:
 2 (negate to get) $\rightarrow -2$ (negate to get) $\rightarrow 2$
- Start off with $2_{\text{ten}} = 0000\ 0010_{\text{two}}$.
 Negate this number by inverting bits $1111\ 1101_{\text{two}}$ and then add 1 to get $1111\ 1110_{\text{two}}$.
 Thus $-2_{\text{ten}} = 1111\ 1110_{\text{two}}$.
- Start off with $-2_{\text{ten}} = 1111\ 1110_{\text{two}}$.
 Negate this number by inverting bits $0000\ 0001_{\text{two}}$ and then add 1 to get $0000\ 0010_{\text{two}}$.
 And this is the binary pattern for 2_{ten} .
- To extend a bit pattern for a positive integer, add 0's to the left.
- To extend a bit pattern for a negative integer, add 1's to the left.
- The ALU check a determines an integer to be positive or negative by checking the most significant bit alone!

Addition & Subtraction (4 bit word)

- **Exercise** Convert the signed binary 0101 1011 to decimal.
- Do the calculations by **addition of signed binary** representations of the operands: 5+12, 12-5, -12-5, and 12-12 in signed binary.
 - Then convert back to decimal numbers. Are your answers valid?

5+12	12-5 = (12)+(-5)	-12-5 = (-12)+(-5)	12-12 = (12)+(-12)
00000101 <u>00001100</u> 00010001	00001100 <u>11111011</u> 00000111	11110100 <u>11111011</u> 11101111	00001100 <u>11110100</u> 00000000
17	7	- 17	0

11

Examples

- Convert 37, -56, 2, -5 into 8 signed binary:
- Convert the 8-bit signed binary to decimal:
 - (a) 0010 0110 (b) 1001 1011 (c) 0110 1111
- Carryout the addition by first converting into 8 bits signed binary:
 - (a) 89+23 (b) 49-23 (c) 5+56

12

Overflow conditions for addition and subtraction

- Conditions indicating overflow are as follows.

Operation	Operand A	Operand B	Result indicating overflow
A+B	= 0	= 0	< 0
A+B	< 0	< 0	= 0
A-B	= 0	< 0	< 0
A-B	< 0	= 0	= 0

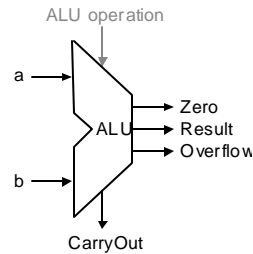
- Conditions not indicated cannot result into overflow, such as adding a positive integer to a negative integer.

- An interface representation of a MIPS ALU

Each of the operands: **a**, **b** is 32 bits wide and the **Result** is 32 bits wide as well.

- The **ALU operation** indicates whether to do + or - and is set by the control unit.
- The **overflow** flag is 1 bit wide.

- The implementation will be shown later.



13

Multiplication

- More complicated than addition.
 - Accomplished via shifting and addition.
- Accomplished via shifting and addition.
- Let's look at 2 versions based on grade school algorithm.
- Negative numbers: Convert to positive, multiply, then negate if initial signs were different
 - there are better techniques, we won't look at them.

14

Multiplication

- Note that the number of digits of the product is considerably larger.

147	10010011	Multiplicand
* 85	* 01010101	Multiplier
	10010011	
	00000000	
	10010011	
	00000000	
	10010011	
	00000000	
	10010011	
	00000000	
12495	011000011001111	Product

- For binary addition we use digits 0 and 1.
 - If the multiplier digit is a 1, we add the multiplicand (1 x multiplicand) to the product.
 - If the multiplier digit is a 0, we add 0 (0 x multiplicand) to the product.

15

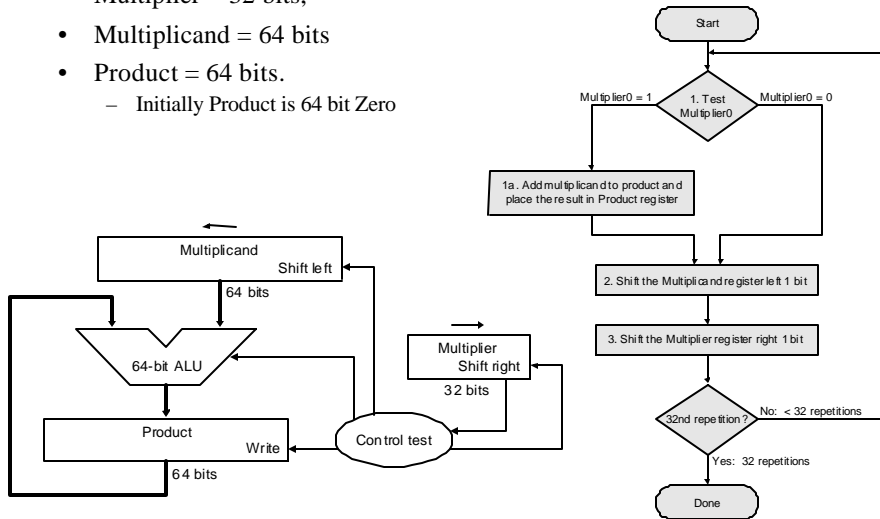
Introducing the Sequential Algorithm

- Maintain:
 - Multiplier register
 - Product registers with a size double the size of the multiplier
 - Multiplicand register also with a size double the size of the multiplier so that it can be added to the product. The **multiplicand** is shifted to the **left** each time a multiplier digit is examined.
- Shift the **multiplier** to the **right** so that only rightmost digit (multiplier0) is examined.
- Product Register is initialized to all 0's. The multiplier will be added to the product when multiplier0 = 1.
- Control: decides when to shift the Multiplier and Multiplicand registers and when to write new values to Product register
- At the end, the multiplier is zero and the product contains the result.

16

Sequential Version of the Multiplication Algorithm and Hardware

- Multiplier = 32 bits,
- Multiplicand = 64 bits
- Product = 64 bits.
 - Initially Product is 64 bit Zero



Example: Sequential Version

Multiplicand x Multiplier: 2ten x 3ten = 0010two x 0011two = 0110two

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 => Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 => Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 => no operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 => no operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

Problem : 64 bit arithmetic, Half of multiplicand = 0!

Improving the Sequential Algorithm

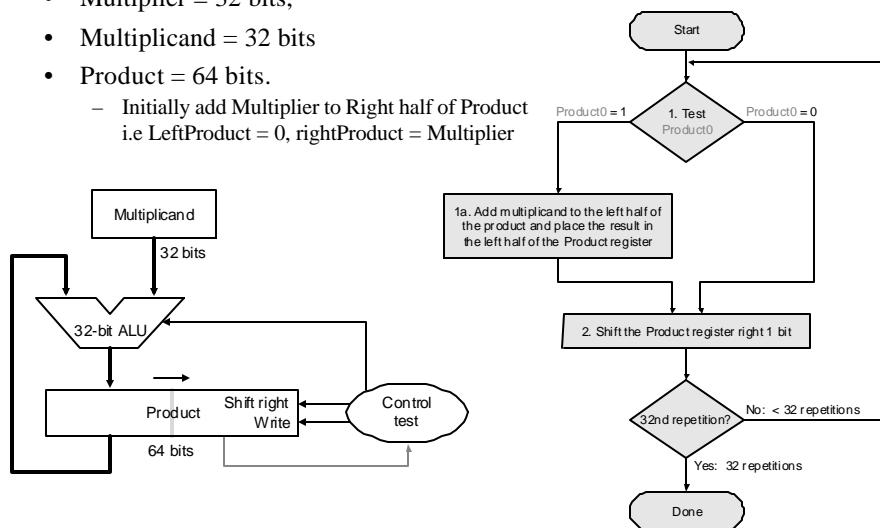
Changes can be made:

- Instead of shifting multiplicand left, shift product right.
- Adder can be shortened to length of the multiplier & multiplicand (4 bits in our example).
- Place multiplier in right half of the product register (multiplier will disappear as product is shifted right).
- Add multiplicand to left half of product.
- At the end, the multiplier is out of the product register and the product contains the result.

19

Refined Version of the Multiplication Algorithm and Hardware

- Multiplier = 32 bits,
- Multiplicand = 32 bits
- Product = 64 bits.
 - Initially add Multiplier to Right half of Product
i.e LeftProduct = 0, rightProduct = Multiplier



Example: Refined Version

Multiplicand x Multiplier: 2ten x 3ten = 0010two x 0011two = 0110two			
Iteration	Step	Multiplicand	Product
0	Initial values	0010	0000 0011
1	1a: 1 => Prod = Prod + Mcand	0010	0010 0011
	2: Shift right Product	0010	0001 0001
2	1a: 1 => Prod = Prod + Mcand	0010	0011 0001
	2: Shift right Product	0010	0001 1000
3	1: 0 => no operation	0010	0001 1000
	2: Shift right Product	0010	0000 1100
4	1: 0 => no operation	0010	0000 1100
	2: Shift right Product	0010	0000 0110

Achieving Multiplication

- Multiplying by powers of two can be better done by shifting to the left.
 - For example to multiply 23 by 4, shift the binary of 23 to the left 2 times.
 - For example to multiply 23 by 2, shift the binary of 23 to the left 1 time.
- One can multiply variable by any integer using MIPS sll (shift left logical) and add instructions:

```
C++:  i = i * 10; # assume i: $s0
```

MIPS:

```
sll $t0, $s0, 3           # i * 23
add $t1, $zero, $t0
sll $t0, $s0, 1           # i * 21
add $s0, $t1, $t0
```

- Recall that $10_{\text{ten}} = 1010_{\text{two}} = (2^3 + 2^1)_{\text{ten}}$

MIPS multiplication

- MIPS provides two 32-bit registers: Hi and Lo

```
mflo $s0 # move from lo (to $s0)
mfhi $s0 # move from hi (to $s0)
```
- Multiplication Operations

```
mult $s0, $s1    # {Hi,Lo}= $s0 x $s1
multu $s0, $s1   # unsigned version
```
- If all variables are same kind, overflow can occur and will NOT be flagged off by MIPS.
- **Example:** Do $z = x \cdot y$. Assume $\$s1=x$, $\$s2=y$ and $\$s3=z$.
The MIPS is:

```
mult $s1, $s2
mflo $s3
```

Floating Point (a brief look)

- We need a way to represent
 - numbers with fractions, e.g., 3.1416
 - very small numbers, e.g., .000000001
 - very large numbers, e.g., 3.15576×10^9
- Representation:
 - sign, exponent, significand: $(-1)^{\text{sign}} \times \text{significand} \times 2^{\text{exponent}}$
 - more bits for significand gives more accuracy
 - more bits for exponent increases range
- IEEE 754 floating point standard:
 - single precision [1 word]: 8 bit exponent, 23 bit significand
 - double precision [2 words] : 11 bit exponent, 52 bit significand

IEEE 754 floating-point standard

- Leading “1” bit of significand is implicit
- Exponent is “biased” to make sorting easier
 - all 0s is smallest exponent all 1s is largest
 - bias of 127 for single precision and 1023 for double precision
 - summary: $(-1)^{\text{sign}} \times (1+\text{significand}) \times 2^{\text{exponent} - \text{bias}}$
- Example 1:
 - decimal: $-0.75 = -3/4 = -3/2^2$
 - binary: $-0.11 = -1.1 \times 2^{-1}$
 - floating point: exponent = bias + -1 = 127 - 1 = 126 = 01111110
 - IEEE single precision: 1 01111110 1000000000000000000000
- Example 2:
 - decimal: 0.356
 - binary: 0.01011011001 (can be more accurate) = $1.011011001 \times 2^{-2}$
 - floating point: exponent = 127 - 2 = 125 = 01111101
 - IEEE single precision: 0 01111101 0110110010000000000000

MIPS Floating Point

- MIPS chips use the IEEE 754 floating point standard, both the 32 bit and the 64 bit versions. However these notes cover only the 32 bit instructions.
- Floating point on MIPS was originally done in a separate chip called coprocessor 1 (also called the FPA for Floating Point Accelerator).
- Modern MIPS chips include floating point operations on the main processor chip. But the instructions sometimes act as if there were still a separate chip.
- There are 32 registers (see table below). The register \$f0 is not \$zero!!

32 floating-point registers	\$f0, \$f1, \$f2, ..., \$f31	MIPS floating point registers are used in pairs for double precision numbers. Odd numbered registers cannot be used for arithmetic or branch, just for data transfer of the right "half" of double precision register pairs.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4293967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls

MIPS Floating Point Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	add.s \$f2,\$f4,\$f6	$f2 = f4 + f6$	Floating-Point add (single precision)
	FP subtract single	sub.s \$f2,\$f4,\$f6	$f2 = f4 - f6$	Floating-Point sub (single precision)
	FP multiply single	mul.s \$f2,\$f4,\$f6	$f2 = f4 * f6$	Floating-Point multiply (single precision)
	FP divide single	div.s \$f2,\$f4,\$f6	$f2 = f4 / f6$	Floating-Point divide (single precision)
	FP add double	add.d \$f2,\$f4,\$f6	$f2 = f4 + f6$	Floating-Point add (double precision)
	FP.dubtract double	.dub.d \$f2,\$f4,\$f6	$f2 = f4 - f6$	Floating-Point sub (double precision)
	FP multiply double	mul.d \$f2,\$f4,\$f6	$f2 = f4 * f6$	Floating-Point multiply (double precision)
	FP divide double	div.d \$f2,\$f4,\$f6	$f2 = f4 / f6$	Floating-Point divide (double precision)
Data transfer	load word coprocessor 1	lwc1 \$f1,100(\$2)	$f1 = \text{Memory}[S2+100]$	32-bit data to FP register
	store word coprocessor 1	swc1 \$f1,100(\$2)	$\text{Memory}[S2+100] = f1$	32-bit data to memory
Arithmetic	branch on FP true	bc1t 100	if (cond == 1) go to PC+4+100	PC relative branch if FP condition
	branch on FP false	bc1f 100	if (cond == 0) go to PC+4+100	PC relative branch if not condition
	FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s \$f2,\$f4	if ($f2 < f4$) cond=1; else cond=0	Floating-point compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d \$f2,\$f4	if ($f2 < f4$) cond=1; else cond=0	Floating-point compare less than double precision

Summary

- We have looked at how to represent integer in MIPS in both signed and unsigned form.
- The largest integer that can fit in a computer word in both signed and unsigned form can be obtained using formulas involving powers of 2.
- Overflow can result if the result cannot be accommodated in a word.
- Two algorithms for multiplication are presented.
- Floating point numbers and double floating point numbers are stored in a special form different from integers.
- MIPS has special instructions for handling multiplication, and floating point numbers.