Chapter Three

 1

Numbers

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

 2

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

$0-(-4)$

- Issues: balance, number of zeros, ease of operations
- Which one is best? Why?

$\boxed{x+(-x)} = 000$

 3

MIPS

- 32 bit signed numbers:

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

 4

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: "negate" and "invert" are quite different!
- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate gets converted to 32 bits for arithmetic
  - copy the most significant bit (the sign bit) into the other bits
    0010 -> 0000 0010
    1010 -> 1111 1010
  - "sign extension" (lbu vs. lb)

 5

Addition & Subtraction

- 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

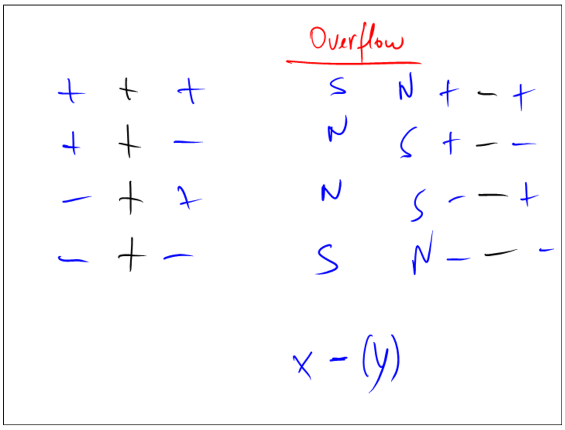
```
  0111
+ 1010
```

- 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"

 6



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 ?

A + B
0
A - B

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`

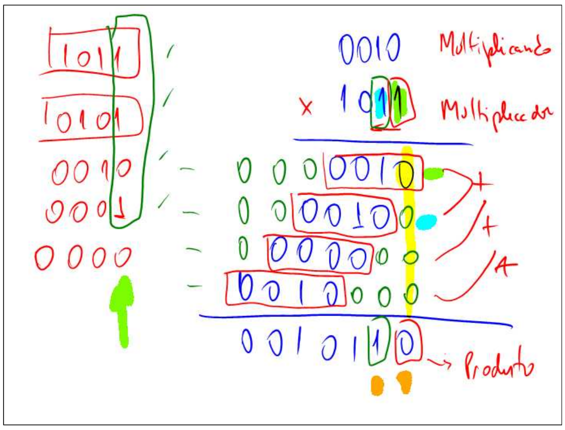
note: `addiu` *still sign-extends!*
note: `sltu, sltiu` *for unsigned comparisons*

Multiplication

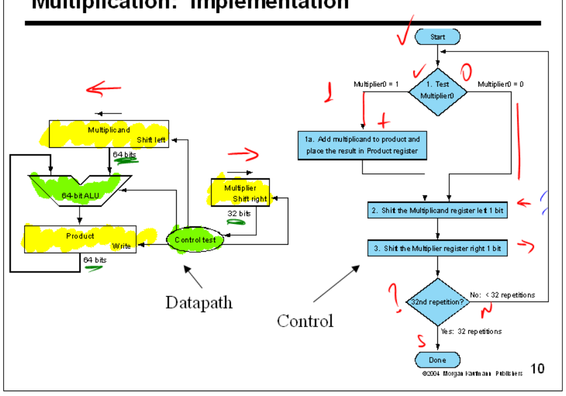
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on a gradeschool algorithm

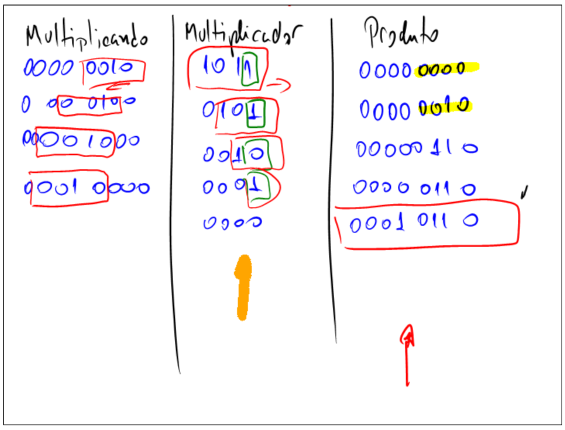
```
    0010  (multiplicand)
  x 1011  (multiplier)
```

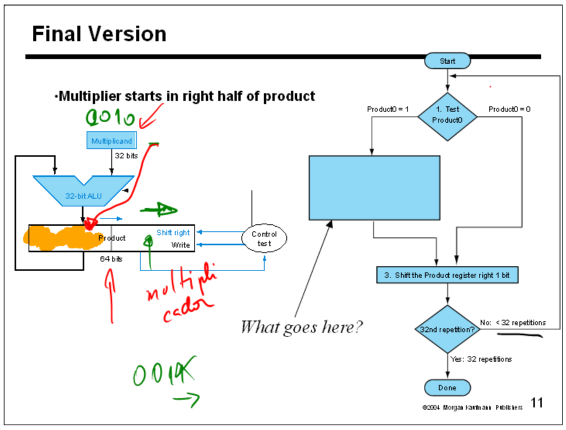
- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

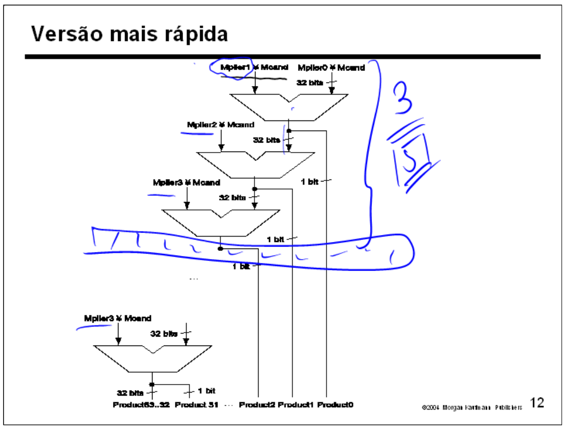


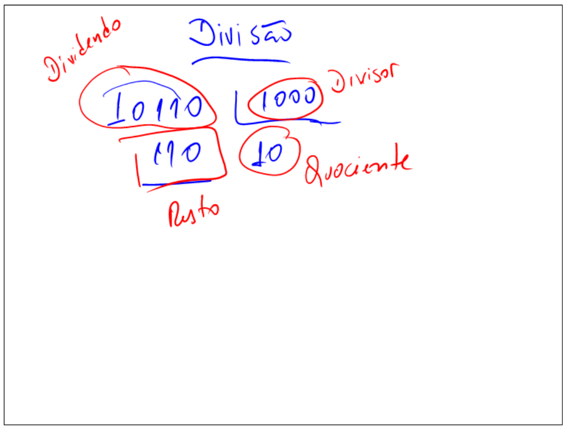
Multiplication: Implementation

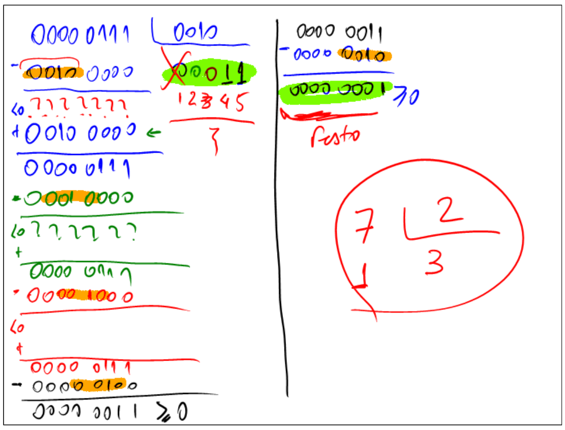


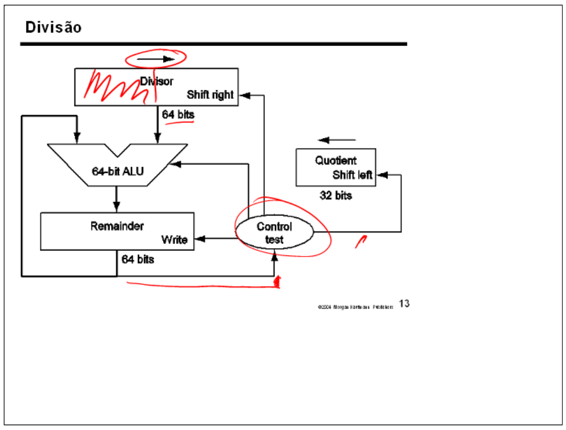

Multiplicando
Multiplicador
Produto
0000 0010
1011
0000 0000
0001 0110


Final Version
•Multiplier starts in right half of product
Multiplicand
32 bits
32-bit ALU
Product
Shift right
Write
Control test
64 bits
multiplicador
What goes here?
Start
1. Test Product0
Product0 = 1
Product0 = 0
3. Shift the Product register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done
11


Versão mais rápida
Mplier1 • Mcand
Mplier0 • Mcand
32 bits
Mplier2 • Mcand
Mplier3 • Mcand
1 bit
Product63..32
Product 31
Product2
Product1
Product0
12

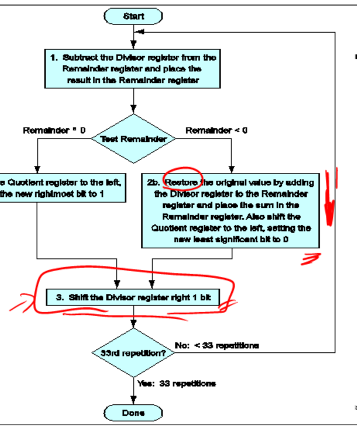

Dividendo
Divisão
10110
1000
Divisor
110
10
Quociente
Resto


0000 0111
0010
0000 0011
Resto
7
2
3
1


Divisão
Divisor
Shift right
64 bits
64-bit ALU
Quotient
Shift left
32 bits
Remainder
Write
Control test
13

Quociente	Divisor	Resto
0000	0010 0000	0000 0111
0000	0001 0000	0000 0111
0000	0000 1000	0000 0111
0000	0000 0100	0000 0111
0001	0000 0010	0000 0011
0011		0000 0001

Algoritmo



14

int i, j

float a, b

4.75

π

3,333333...

134.37

1343.7×10^{-1}

1.3437×10^{2} *

Floating Point

- 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: $(-1)^{sign} \times significand \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- Overflow
- Underflow

 15

$\cancel{1}.11 \times 2^{2}$

Como representar?

IEEE 754

- Números normalizados
  - Números da forma 1,XXXXXXX
  - Não é necessário armazenar o 1,
- Representação com 32 bits (precisão simples)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
S	8 Expoente																Mantissa 23														

- Representação com 64 bits (precisão dupla)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
S	Expoente 11																					Mantissa 52									

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
															Mantissa																

 16

IEEE 754 floating-point standard

- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand
- 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: $(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}$
- Example:
  - decimal: -.75 = - (½ + ¼)
  - binary: -.11 = -1.1 x 2^{-1}
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: 10111111010000000000000000000000

-1.

$(-1)^1 \times (1+0.1) \times 2^{-1}$

s e m

$1.610 \times 10^{-1} + 1.781 \times 10^{2}$

+

178.1 ← 1781×10^{-1}

+ 1.610

1.782610×10^{2} ← 1782.610×10^{-1}

$0{,}11 = 1{,}1 \times 2^{-1}$ + $\cancel{1{,}1 \times 2^{4}}$

$\cancel{0{,}000011 \times 2^{4}}$ $110000{,}0 \times 2^{-1}$

S | 126 | 1. . . . .

S | 131 | 1 . . . . . .

126 + 131 = 257

127 127

126 + 131 − 127 = 130

3

Exemplos

- Comparar em binário
  - 0,75; -0,75; 4,25; 0,625; 19

Constantes

Precisão Simples		Precisão Dupla		Valor
Expoente	Mantissa	Expoente	Mantissa	
0	0	0	0	0
0	<> 0	0	<> 0	Número não normalizado
1-254	Qualquer	1-2046	Qualquer	Número em ponto flutuante
255	0	2047	0	Infinito
255	<> 0	2047	<> 0	NaN (Not a Number)

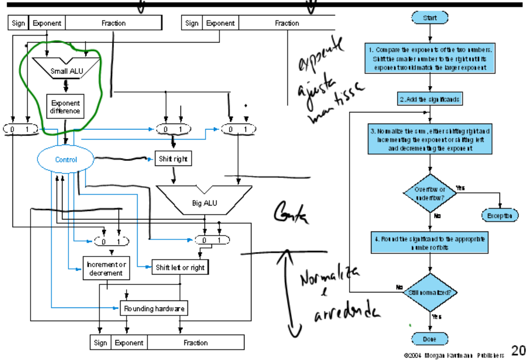
1×2^{-1}

f | exp | mantissa

4

| 0000...11

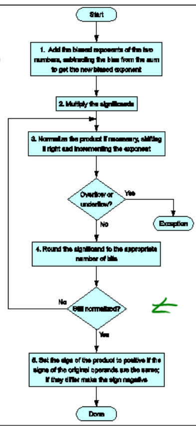
Floating point addition



©2004 Morgan Kaufmann Publishers 20

Multiplication

$1{,}5 \times 10^{-5} * 2{,}7 \times 10^{8}$



organ Kaufmann Publishers 21

Precisão

m

- X + 1 – X = 1?
- Internamente, o processador armazena os números de ponto flutuante com ao menos 2 bits a mais: *guard* e *round*
- Um terceiro bit, o *sticky* indica se algum conteúdo significativo foi perdido em arredondamento
- Formas de arredondar:
  - Em direção a +infinito
  - Em direção a –infinito
  - Truncar
  - Em direção ao par mais próximo

float
2^{32}
int
2^{32}

©2004 Morgan Kaufmann Publishers 22

$(X) + 1 - X$

$X >>> 1$

Floating Point Complexities

- Operations are somewhat more complicated (see text)
- In addition to overflow we can have "underflow"
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields "infinity"
  - zero divide by zero yields "not a number"
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

©2004 Morgan Kaufmann Publishers 23

Chapter Three Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two's complement
  - IEEE 754 floating point
- Computer instructions determine "meaning" of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines
- Algorithm choice is important and may lead to hardware optimizations for both space and time (e.g., multiplication)

- You may want to look back (Section 3.10 is great reading!)

©2004 Morgan Kaufmann Publishers 24