

Arithmetic for Computers

Dr. Arjan Durrresi
Louisiana State University
Baton Rouge, LA 70803
durrresi@csc.lsu.edu

These slides are available at:
http://www.csc.lsu.edu/~durrresi/CSC3501_05/



- Signed and Unsigned Numbers
- Addition and Subtraction
- Multiplication and Division
- Floating Point

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: $0 \dots 2^n - 1$
- 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?

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
- Which one is best? Why?

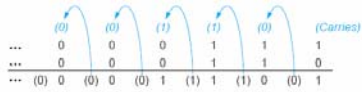
MIPS

- 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} / maxint
 - 0111 1111 1111 1111 1111 1111 1111 1111_{two} = + 2,147,483,647_{ten} / minint
 - 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

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

Addition



Addition & Subtraction

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

$$\begin{array}{r} 0111 \quad 0111 \quad 0110 \\ + 0110 \quad - 0110 \quad - 0101 \\ \hline \end{array}$$
- Two's complement operations easy
 - subtraction using addition of negative numbers

$$\begin{array}{r} 0111 \\ + 1010 \\ \hline \end{array}$$
- Overflow (result too large for finite computer word):
 - e.g., adding two n-bit numbers does not yield an n-bit number

$$\begin{array}{r} 0111 \\ + 0001 \\ \hline 1000 \end{array}$$
 note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

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?

Detecting Overflow

| 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 |

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

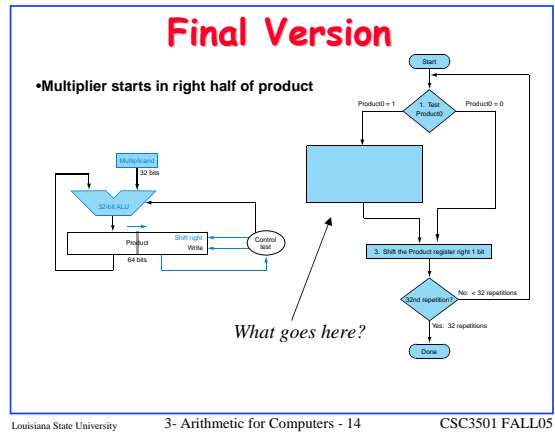
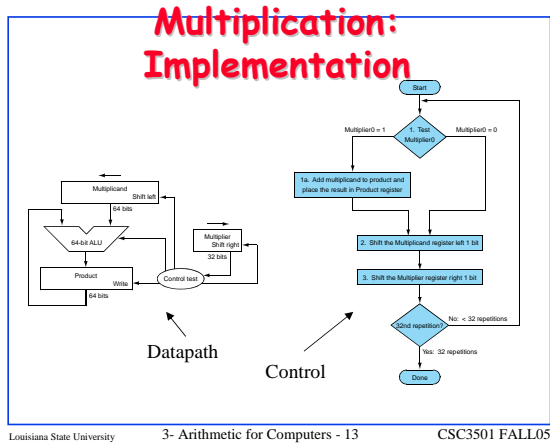
note: addu still sign-extends!
 note: sltu, sltiu for unsigned comparisons

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

$$\begin{array}{r} 0010 \quad (\text{multiplicand}) \\ \times 1011 \quad (\text{multiplier}) \\ \hline \end{array}$$

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

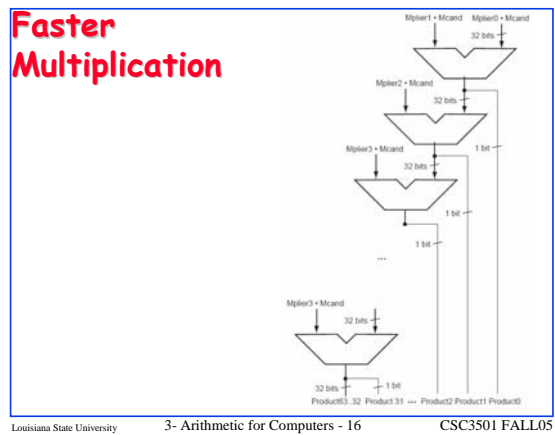


Example

2 x 3 -> 0010x0011

| Iteration | Step | Multiplier | Multiplicand | Product |
|-----------|--|------------|--------------|-----------|
| 0 | Initial values | 0011 | 0000 0010 | 0000 0000 |
| 1 | 1a. $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mrand}$ | 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. $1 \Rightarrow \text{Prod} = \text{Prod} + \text{Mrand}$ | 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 | 0000 1000 | 0000 0110 |
| | 2: Shift left Multiplicand | 0000 | 0010 0000 | 0000 0110 |
| | 3: Shift right Multiplier | 0000 | 0010 0000 | 0000 0110 |

Louisiana State University 3- Arithmetic for Computers - 15 CSC3501 FALL05

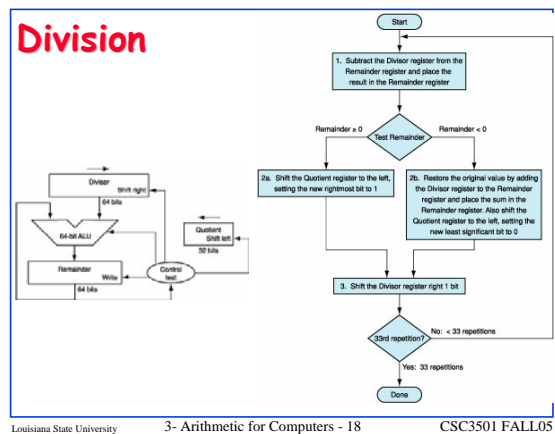


Division

| | | |
|--------------|---------|-----------|
| 1001 | | Quotient |
| Divisor 1000 | 1001010 | Dividend |
| | -1000 | |
| | ----- | |
| | 10 | |
| | 101 | |
| | 1010 | |
| | -1000 | |
| | ----- | |
| | 10 | Remainder |

Dividend = Quotient x Divisor + Remainder

Louisiana State University 3- Arithmetic for Computers - 17 CSC3501 FALL05

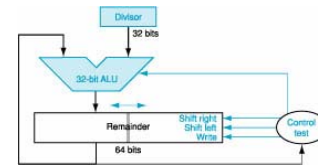


Division: Example

Using 4-bit, divide 7 by 2, 0111 by 0010

| Iteration | Step | Quotient | Divisor | Remainder |
|-----------|-------------------------------|----------|-----------|-----------|
| 0 | Initial values | 0000 | 0010 0000 | 0000 0111 |
| 1 | 1: Rem=Rem-Div | 0000 | 0010 0000 | 1110 0111 |
| | 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 | 0000 | 0010 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0001 0000 | 0000 0111 |
| 2 | 1: Rem=Rem-Div | 0000 | 0001 0000 | 1111 0111 |
| | 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 | 0001 | 0001 0000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 1000 | 0000 0111 |
| 3 | 1: Rem=Rem-Div | 0000 | 0000 1000 | 1111 1111 |
| | 2b: Rem<0 ⇒ +Div, sll Q, Q0=0 | 0000 | 0000 1000 | 0000 0111 |
| | 3: Shift Div right | 0000 | 0000 0100 | 0000 0111 |
| 4 | 1: Rem=Rem-Div | 0000 | 0000 0100 | 0000 0011 |
| | 2a: Rem>0 ⇒ sll Q, Q0=1 | 0001 | 0000 0100 | 0000 0011 |
| | 3: Shift Div right | 0001 | 0000 0010 | 0000 0011 |
| 5 | 1: Rem=Rem-Div | 0001 | 0000 0010 | 0000 0001 |
| | 2a: Rem>0 ⇒ sll Q, Q0=1 | 0011 | 0000 0010 | 0000 0001 |
| | 3: Shift Div right | 0011 | 0000 0001 | 0000 0001 |

Final Version



Can we implement faster division using parallel hardware?

Signed Division

- We must set the sign for both quotient and the remainder
- Dividend = Quotient × Divisor + Remainder
- Example all combinations of $\pm 7 \div \pm 2$
 - $+7 \div +2$: Quotient +3, Remainder +1: $7 = 3 \times 2 + (+1)$
 - $-7 \div +2$: Quotient -3, Remainder = Dividend - Quotient × Divisor = $-7 - (-3 \times 2) = -1$
- Rule: The Dividend and the Remainder must have the same signs
 - $+7 \div -2$: Quotient -3, Remainder +1
 - $-7 \div -2$: Quotient +3, Remainder -1

Division in MIPS

- MIPS has two instructions for both signed and unsigned instructions: `div` and `divu`
- MIPS divide instructions ignore overflow
- MIPS software must check the divisor it is zero as well as overflow.

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
- Scientific notation,
 - Normalized 1.0×10^{-9} , not normalized 0.1×10^{-9}
 - Binary numbers in scientific notation 1.0×2^{-1}
- Floating point: represent numbers in which the binary points is not fixed

Floating Point

- IEEE 754 floating point standard:
 - single precision: 8 bit exponent, 23 bit significand
 - double precision: 11 bit exponent, 52 bit significand
- 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
- Overflow: the exponent is too large to be represented in the exponent field
- Underflow: negative exponent is too large to be represented in the exponent field

Floating Point

- IEEE 754 floating point standard:
 - single precision: 8 bit exponent, 23 bit fraction
 - double precision: 11 bit exponent, 52 bit fraction
 - $(-1)^s \times (1 + \text{Fraction}) \times 2^E$
 - Make the leading bit implicit \rightarrow Significant 24 or 54 bits
 - $(-1)^s \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + \dots) \times 2^E$

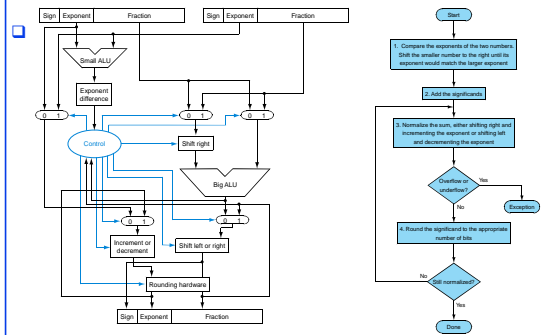
IEEE 754 floating-point standard

- 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: $(-1)^{\text{sign}} \times (1 + \text{significant}) \times 2^{\text{exponent} + \text{bias}}$
- Example:
 - decimal: $-0.75 = -(\frac{1}{2} + \frac{1}{4})$
 - binary: $-0.11 = -1.1 \times 2^{-1}$
 - floating point: exponent = 126 = 01111110
 - IEEE single precision:
1 01111110 10000000000000000000000

Floating point addition

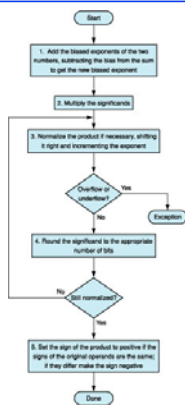
- Add $9.999 \times 10^1 + 1.610 \times 10^{-1}$ using only four decimal digits of the significant and two decimal digits of the exponent
- Step 1: Align the decimal point of the number that has the smaller exponent: $1.610 \times 10^{-1} = 0.01610 \times 10^1$
- Step 2: Add the significands: $9.999 + 0.016 = 10.015$
 - The sum is 10.015×10^1
- Step 3: Normalize: 1.0015×10^2
- Step 4: Round 1.002×10^2

Floating point addition



Multiplication

- $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- Step 1. Calculate the exponent $10 + (-5) = 5$, with biased exponent: $10 + 127 + (-5) + 127 = 259$ too large $10 + 127 + (-5) + 127 - 127 = 5 + 127$
- Step 2. Multiplication of significands: $1.110 \times 9.200 = 10.21200 = 10.212 \times 10^0$
- Step 3. Normalize: 1.0212×10^6
Check for overflow and underflow
- Step 4. Round: 1.021×10^6
- Step 5. The sign of the product depends on the signs of operands



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!

Round and Guard Digits

- In IEEE 754 there are two extra bits on the right during intermediate additions: guard and round
- Add 2.56×10^0 to 2.34×10^0 , using only three significant decimal digits
 - Shift 0.0256×10^0 , using guard digit for 5 and round digit for 6
 - The sum is 2.365
 - Rounded to 2.37
 - Without guard and round the sum = 2.36

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!)