Computer Arithmetic
Ch 8

ALU
Integer Representation
Integer Arithmetic
Floating-Point Representation
Floating-Point Arithmetic

Arithmetic Logical Unit (ALU) (2)

(aritmeettis-looginen yksikkö)

- Does all “work” in CPU — Rest is management!
  - integer & floating point arithmetic's
  - copy values from one register to another
  - comparisons
  - left and right shifts
  - branch and jump address calculations
  - load/store address calculations
- Control signals from CPU control unit
  - what operation to perform and when

ALU Operations (5)

- Data from/to internal registers (latches)
  - input data may have been copied from normal registers, or it may have come from memory
  - output data may go to normal registers, or to memory

Fig. 8.1

- Wait for maximum gate delay
- Result is ready
- Result may (also) be in flags (lipuke)
- Flags may cause an interrupt

Integer Representation (8)

Everything with 0 and 1
no plus/minus signs
no decimal periods
assumed “on the right”

+32 +16 +8 +1

57 = 0011 1001 = 0x39

hexadecimal presentation 3*16 +9*1

- Unsigned integers
- Positive numbers easy
  - normal binary form
- Negative numbers
  - sign-magnitude
  - two’s complement

sign bit = MSB = most significant bit

-57 = 1011 1001

-57 = 1100 0111 +1

“sign” bit complements

Twos Complement

(kahden komplementti)

- Most used
- Have space for 8 bits?
  - use 7 bits for data and 1 bit for sign

+2 = 0000 0010
+1 = 0000 0001
0 = 0000 0000
-1 = 1111 1111
-2 = 1111 1110

  - just like in sign-magnitude or in one’s complement (but presentation is different)

ones complement: -0 = 1111 1111

Why Two’s Complement Presentation? (4)

- Math is easy to implement
  - subtraction becomes addition

X-Y = X + (-Y)

easy to do, simple circuit

- Have just one zero
  - comparisons to zero easy
- Easy to expand to presentation with more bits
  - simple circuit

57 = 0011 1001 = 0000 0000 0011 1001

-57 = 1100 0111 = 1111 1111 1100 0111

sign extension

Why Two's Complement Presentation? (3)

- Range with n bits: $-2^{n-1} \dots 2^{n-1} - 1$

  8 bits: $-2^7 \dots 2^7 - 1 = -128 \dots 127$
  32 bits: $-2^{31} \dots 2^{31} - 1 = -2\,147\,483\,648 \dots 2\,147\,483\,647$

- Overflow easy to recognise
  - add positive & negative - no overflows
  - add 2 positive/negative numbers
    - if sign bit of result is different? ⇒ overflow!

```
   57 = 0011 1001
 + 80 = 0101 0000
 ----------------
  137 = 1000 1001
```


Why Two's Complement Presentation? (5)

- Addition easy if one or both operands negative
  - treat them all as unsigned integers

Same circuit works for both (except for overflow check)

```
 13 = 1101        -3 = 1101        +3 = 0011
 +1 = 0001        +1 = 0001        1100
 ---------        ---------          +1
 14 = 1110        -2 = 1110        ----
                                   1101
```

Digits represent 4 bit unsigned numbers

Digits represent 4 bit two's complement numbers

Integer Arithmetic Operations

- Negation X = -Y
- Addition X = Y+Z
- Subtraction X = Y-Z
- Multiplication X = Y*Z
- Division X = Y / Z

Integer Negation (6)

57 = 0011 1001

- Step 1: negate all bits — 1100 0110
- Step 2: add 1 — +1 → 1100 0111

Step 3: special cases

- ignore carry bit
  - negate 0?

```
   0 = 0000 0000
       1111 1111
             +1
  -0 = 1 0000 0000
```

- check that sign bit really changes
  - can not negate smallest negative
  - results in exception?

```
 -128 = 1000 0000
 bitwise not: 0111 1111
 add 1: 1000 0000
```


Integer Addition and Subtraction (4)

- Normal binary addition
  - 32 bit full adder?
- Ignore carry & monitor sign bit for overflow
- In case of SUB, complement 2nd operand
- 2 circuits
  - addition
  - complement

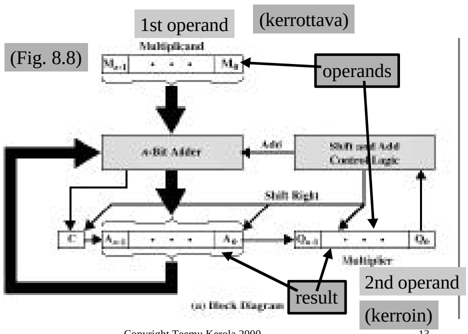
Fig. 8.6

Integer Multiplication (4)

- Complex
- Operands 32 bits ⇒ result 64 bits
- "Just like" you learned at school
  - optimised for binary data
    - it is easy to multiply with 0 or 1!
- Simpler case with unsigned numbers
  - simple circuits
    - adder
    - shifter
    - wires

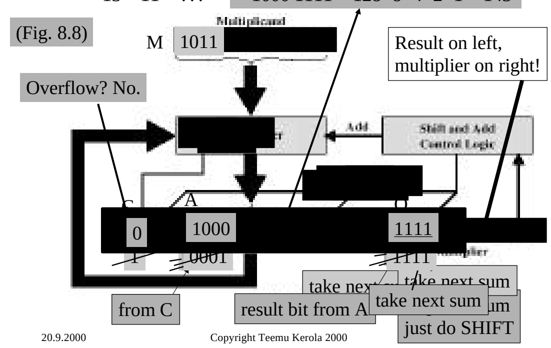
Fig. 8.7

Unsigned Multiplication Example



Unsigned Multiplication Example (19)

13 * 11 = ??? = 1000 1111 = 128+8+4+2+1 = 143



Multiplication with Negative Values

- Multiplication for unsigned numbers does not work for negative numbers
  - algorithm applies only for unsigned integer representation
  - not the same case as with addition
- Could do it all with unsigned values
  - change operands to positive values
  - do multiplication with positive values
  - negate result if needed
  - OK, but can do better, I.e., faster

The Gist in Booth's Algorithm (7)

Unsigned multiplication:
addition for every "1" bit in multiplicand

5 * 7 ⇨ 0101 * 0111 ⇨

```
   0101
+ 01010
+010100
=100011
```

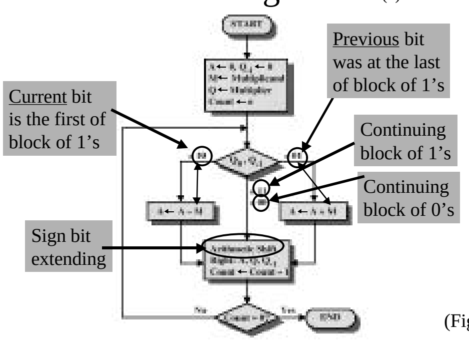
- Booth's algorithm:
  - combine all adjacent 1's in multiplicand together, replace all additions by one subtraction and one addition (to result)

5 * 7 ⇨ 0101 * 0111

⇨ 0101 * (-0001 +1000) ⇨

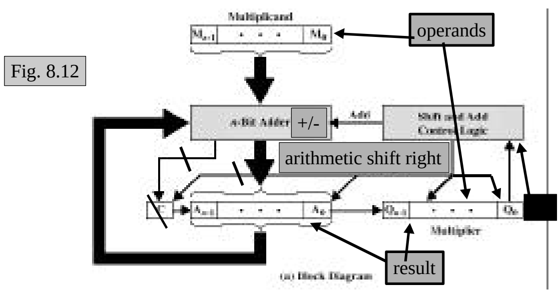
```
 +0101000
   - 0101
=  100011
```


Booth's Algorithm (5)



(Fig. 8.12)

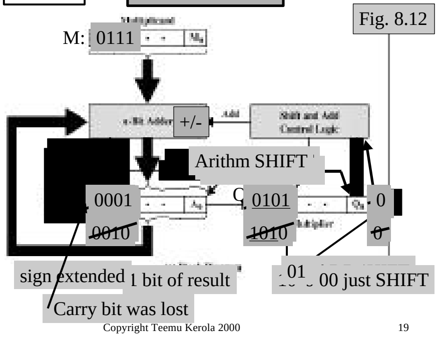
Booth's Algorithm for Twos Complement Multiplication



Booth's Algorithm Example (15)

7 * 3 = ?

= 0001 0101 = 21



Integer Division

- Like in school algorithm (Fig. 8.15)
  - easy: new quotient digit 0 or 1
  - M register for dividend (jaettava)
  - Q register for divisor & quotient (jakaja, osamäärä)
  - A register for (partial) remainder (jakojäännös)

Floating Point Representation

-0.000 000 000 123 = $-1.23 * 10^{-10}$

+0.123 = $+1.23 * 10^{-1}$

+123.0 = $+1.23 * 10^{2}$

+123 000 000 000 000 = $+1.23 * 10^{14}$

"+"	"14"	"1.23"
sign	exponent (exponentti)	mantissa or significand (mantissa)

IEEE 32-bit Floating Point Standard

IEEE Standard 754

"+"	"14"	"1.1875" = "1.0011"
sign	exponent	mantissa or significand

- 1 bit for sign, 1⇒ "-", 0 ⇒ "+"
- I.e., Stored value *S* ⇒ Sign value = $(-1)^S$

IEEE 32-bit FP Standard

"+"	"15"	"1.1875" = "1.0011"
sign	exponent	mantissa or significand

- 8 bits for exponent, 2^8-1= 127 <u>biased form</u>

exponent = 5 —store→ 5+127 = 132 =1000 0100

exponent = -1 —store→ -1+127 = 126 = 0111 1110

exponent = 0 —store→ 0+127 = 127 = 0111 1111

  - stored exponents 0 and 255 are special cases
    - stored range: **1 - 254** ⇒ true range: **-126 - 127**

IEEE 32-bit FP Standard (7)

“+”	“15”	“0.1875” = “0.0011”
sign	exponent	mantissa or significand

1/8 = 0.1250
1/16= 0.0625
0.1875

- 23 bits for mantissa, stored so that

1) Binary point (.) is assumed just right of first digit

2) Mantissa is normalised, so that leftmost digit is 1

3) Leftmost (most significant) digit (1) is not stored (implied bit)

mantissa	exponent
0.0011	“15”
1.100	“12”
1000	“12”

24 bit mantissa!

IEEE 32-bit FP Values

$23 = +10111.0 * 2^0 = +1.0111 * 2^4$ =?

4+127=131

0	1000 0011	011 1000 0000 0000 0000 0000
sign 1 bit	exponent 8 bits	mantissa or significand 23 bits

$1.0 = +1.0000 * 2^0$ = ?

0+127 = 127

0	0111 1111	000 0000 0000 0000 0000 0000
sign 1 bit	exponent 8 bits	mantissa or significand 23 bits

IEEE 32-bit FP Values

0	1000 0000	111 1000 0000 0000 0000 0000
sign 1 bit	exponent 8 bits	mantissa or significand 23 bits

X = ?

$X = (-1)^0 * 1.1111 * 2^{(128-127)}$

$= 1.1111_2 * 2$

= (1+ 1/2 + 1/4 + 1/8 + 1/16) * 2

= (1 + 0.5 + 0.25 + 0.125+0.0625) * 2

= 1.9375 * 2 = 3.875

IEEE-754 Floating-Point Conversion

Christopher Vickery
Computer Science Department at Queens College of CUNY
(The City University of New York)

http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html

IEEE FP Standard

- Single Precision (SP) 32 bits
- Double Precision (DP) 64 bits

(yksin- ja kaksinkertainen tarkkuus)

Table 8.3

- Special values
  - -0, +∞, −∞, NaN
  - denormalized values

Table 8.4

Not a Number

IEEE SP FP Range

- Range
  - 8 bit exponent, effective range: -126 … +127
  - range 2^{-126} ... $2^{127} \approx -10^{-38}$ ... 10^{38}
- Accuracy
  - 23 bit mantissa, 24 bit effective mantissa
  - change least significant digit in mantissa?
  - $2^{24} \approx 1.7 * 10^{-7} \approx 6$ decimal digits

Floating Point Arithmetic (4)

- Relatively simple — Table 8.5
- Done from registers with all bits
  - implied bit included
- Add/subtract
  - more complex than multiplication
  - denormalize first one operand so that both have same exponent
- Multiplication/Division
  - handle mantissa and exponent separately

FP Add or Subtract (4)

- Check for zeroes — $1.234 \bullet 10^4$ + $4.444 \bullet 10^6$
  - trivial if one or both operands zero
- Align mantissas — $0.01234 \bullet 10^6$ $4.444 \bullet 10^6$
  - same exponent
- Add/subtract — $4.45634 \bullet 10^6$
  - carry?
    $\Rightarrow$ shift right and add increase exponent
- Normalize result — $4.45634 \bullet 10^6$
  - shift left, reduce exponent

FP Special Cases

- Exponent overflow — (ylivuoto)
  - above max — Exception Or ±∞ ?
- Exponent underflow — (alivuoto)
  - below min — Exception or zero?
- Mantissa (significant) underflow
  - in denormalizing may move bits too much right
  - all significant bits lost? — Oooops, lost data!
- Mantissa (significant) overflow — Fix it
  - result of adding mantissas may have carry

FP Multiplication (Division) (7)

Check for zeroes — $3.000 \bullet 10^4$ * $4.444 \bullet 10^6$

Result 0, ±∞ ??

Add exponents

Subtract extra bias — + → $\bullet\ 10^{10}$

Report overflow/underflow

Multiply (divide) mantissas — * → $13.332 \bullet 10^{10}$

Normalise — $1.3332 \bullet 10^{11}$

Round (pyöristä) — $1.333 \bullet 10^{11}$

Rounding (4)

- Guard bits — $4.444 \bullet 10^6$
  - extra padding with zeroes
  - used with computations only — $4.44400 \bullet 10^6$
  - computations with more accuracy than data

2.0 - 1.9999 $\approx 1.000000 \bullet 2^1 - 0.1111111 \bullet 2^1$
$= 1.000000 \bullet 2^1 - 1.111111 \bullet 2^0$ normalised

6 bit mantissa

6 bit mantissa		2 guard bits
$1.000000 \bullet 2^1$	Different accuracy!	$1.000000\ 00 \bullet 2^1$
$- 0.111111 \bullet 2^1$		$- 0.111111\ 10 \bullet 2^1$
$= 0.000001 \bullet 2^1$		$= 0.000000\ 10 \bullet 2^1$
$= 1.000000 \bullet 2^{-5}$		$= 1.000000\ 00 \bullet 2^{-6}$

Align mantis-sas

2 guard bits

Rounding Choices (4)

4 digit accuracy in memory? — 3.1234 or -4.5678

• Nearest representable	3.123 or -4.568
• Toward +∞	3.124 or -4.567
• Toward -∞	3.123 or -4.568
• Toward 0	3.123 or -4.567

IEEE ∞ and NaN

- ∞
  - outside range of finite numbers
  - rules for arithmetic with ∞
- NaN
  - invalid operation (E.g., 0.0/0.0) can result to NaN or exception
    - user control
    - quiet NaN instead of exception

Table 8.6

IEEE Denormalized Numbers (4)

- Problem: What to do when can not normalize any more?
  - Exponent would underflow
- Answer: Denormalized representation
  - smallest representable exponent reserved for this purpose
  - mantissa is not normalized
  - smallest (closest to zero) value is now much smaller than with normalized representation

$0.003456 \bullet 10^{-99}$

6 decimal digit mantissa

Smallest representable exponent

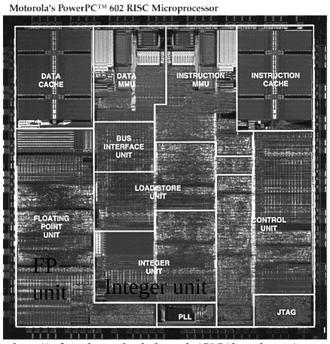
$1.000000 \bullet 10^{-99}$

?

$0.000001 \bullet 10^{-99}$

-- End of Chapter 8: Arithmetic --

Motorola's PowerPC™ 602 RISC Microprocessor



http://infopad.eecs.berkeley.edu/CIC/die_photos/