Computer Architecture and Organization

Chapter 4
Arithmetic for Computers

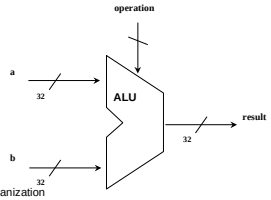
Henk Corporaal / Ben Juurlink
Delft University of Technology
Spring 2000
Email: h.corporaal@et.tudelft.nl
URL : http://cardit.et.tudelft.nl/~heco/courses/et2-010

Topics

- Arithmetic
- Signed and unsigned numbers
- Addition and Subtraction
- Logical operations
- ALU: arithmetic and logic unit
- Multiply
- Divide
- Floating Point
  - notation
  - add
  - multiply

Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    Instruction Set Architecture
    Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture



Binary numbers (1)

- Bits have no inherent meaning (no semantics)
- Decimal number system, e.g.:
  $4382 = 4x10^3 + 3x10^2 + 8x10^1 + 2x10^0$
- Can use arbitrary base g; value of digit c at position i:
  $c \times g^i$
- Binary numbers (base 2)

n-1	n-2	…	1	0	position
a_{n-1}	a_{n-2}	…	a_1	a_0	digit
2^{n-1}	2^{n-2}	…	2^1	2^0	weight

- $(a_{n-1}\, a_{n-2} \ldots a_1\, a_0)_{two} = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \ldots + a_0 \times 2^0$

Binary numbers (2)

- So far numbers are *unsigned*
- With n bits 2^n possible combinations

1 bit	2 bits	3 bits	4 bits	decimal value
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9

- a_0 : *least significant* bit (lsb)
- a_{n-1}: *most significant* bit (msb)

Binary numbers (3)

- 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?

Conversion

- Decimaal -> binair

Divide by 2	Remainder
4382	
2191	0
1095	1
547	1
273	1
136	1
68	0
34	0
17	0
8	1
4	0
2	0
1	0
0	1

4382_{ten} = $1\ 0001\ 0001\ 1110_{two}$

- Hexadecimal: base 16. Octal: base 8

$1010\ 1011\ 0011\ 1111_{two} = ab3f_{hex}$

computer architecture and organization 7

Signed binary numbers

Possible representations:

- Sign Magnitude:

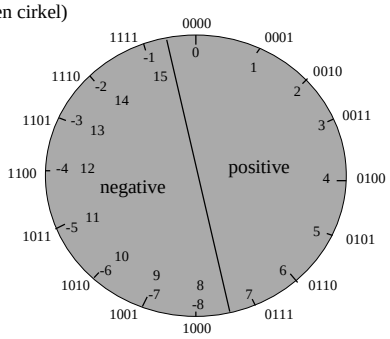
Sign Magnitude:	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

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

computer architecture and organization 8

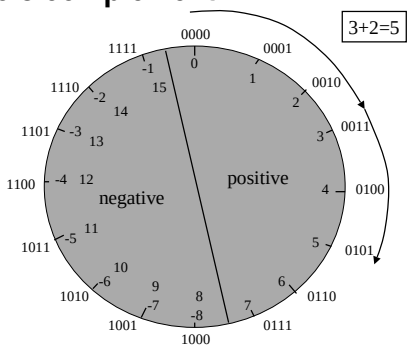
Two's complement

(getallen cirkel)



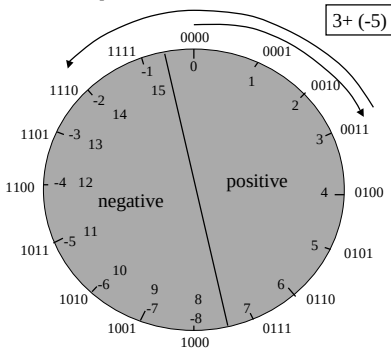
computer architecture and organization 9

Two's complement



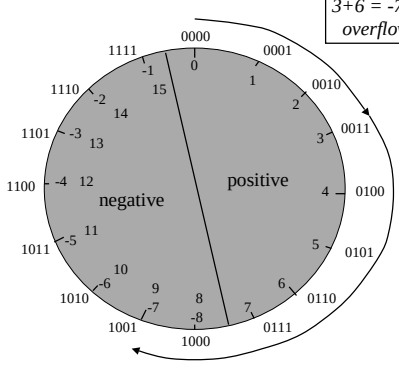
computer architecture and organization 10

Two's complement



computer architecture and organization 11

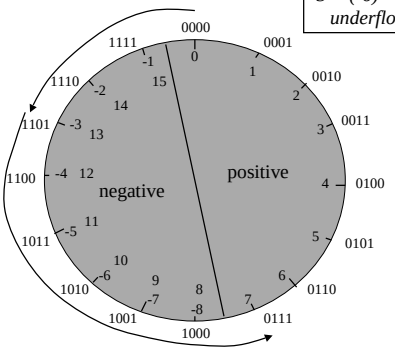
Two's complement



computer architecture and organization 12

Two's complement

-3 + (-6) = 7 !!
underflow



Two's complement

- 32 bit signed numbers:

```
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
```

maxint
minint

  - Range [-2^{31} .. $2^{31}-1$]
- $(a_{n-1}\, a_{n-2} \ldots a_1\, a_0)_{2\text{'s-compl}} = -a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \ldots + a_0 \times 2^0$
  $= -2^n + a_{n-1} \times 2^{n-1} + \ldots + a_0 \times 2^0$

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - remember: "negate" and "invert" are quite different!
- Proof:

  $a + \bar{a} = 1111.1111b = -1$ d =>

  $-a = \bar{a} + 1$

Two's Complement Operations

Converting n bit numbers into numbers with more than n bits:

- MIPS 8 bit, 16 bit values / immediates converted to 32 bits
- Copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

- MIPS "sign extension" example instructions:

lb	load byte (signed)
lbu	load byte (unsigned)
slti	set less than immediate (signed)
sltiu	set less than immediate (unsigned)

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

```
  0110      0110
- 0101    + 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   note that overflow term is somewhat misleading,
1000       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 ?

Effects of Overflow

- When an exception (interrupt) occurs:
  - Control jumps to predefined address for exception (*interrupt vector*)
  - Interrupted address is saved for possible resumption in *exception program counter* (EPC); new instruction: `mfc0`
  - *Interrupt handler* handles exception (part of OS). registers `$k0` and `$k1` reserved for OS
- Details based on software system / language
  - C ignores integer overflow; FORTRAN not
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`
  note: `addiu` and `sltiu` *still sign-extends!*

Logic operations

- Sometimes operations on individual bits needed:

Logic operation	C operation	MIPS instruction
Shift left logical	<<	`sll`
Shift right logical	>>	`srl`
Bit-by-bit AND	&	`and, andi`
Bit-by-bit OR	|	`or, ori`

- `and` and `andi` can be used to turn off some bits; `or` and `ori` turn on certain bits
- Of course, AND en OR can be used for logic operations. Note: C's logische AND (&&) and OR (||) are *conditional*
- `andi` and `ori` perform no sign extension !

Exercise: gates

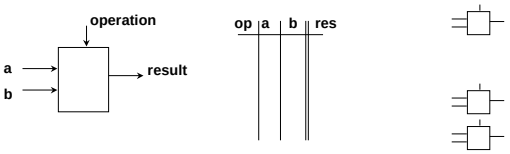
Given: 3-input logic function of A, B and C, 2-outputs

Output D is true if at least 2 inputs are true
Output E is true if odd number of inputs true

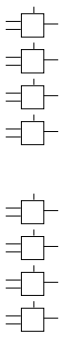
- Give truth-table
- Give logic equations
- Give implementation with AND and OR gates, and Inverters.

An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them

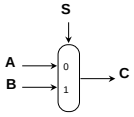


- Possible Implementation (sum-of-products)



Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

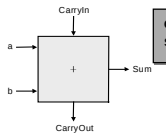


note: we call this a 2-input mux even though it has 3 inputs!

- Lets build our ALU and use a MUX to select the outcome for the chosen operation

Different Implementations

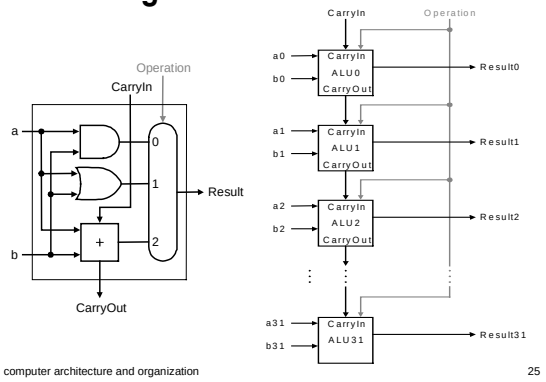
- Not easy to decide the "best" way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - For our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:



$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$
$sum = a\ xor\ b\ xor\ c_{in}$

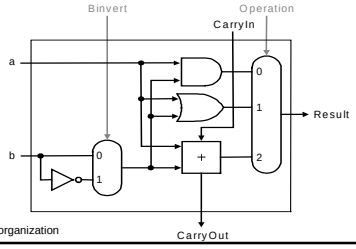
- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU



What about subtraction (a – b) ?

- Two's complement approach: just negate **b** and add
- How do we negate?
- A very clever solution:

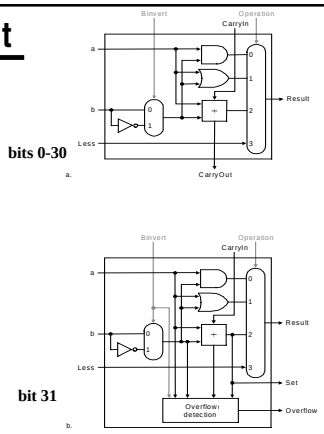


Tailoring the ALU to the MIPS

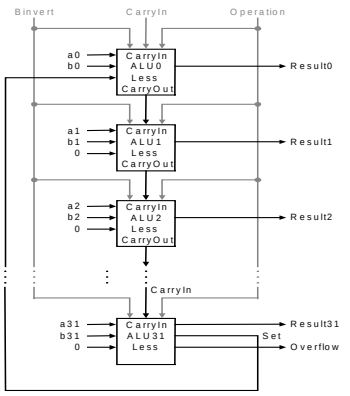
- Need to support the set-on-less-than instruction (`slt`)
  - remember: `slt rd,rs,rt` is an arithmetic instruction
  - produces a 1 if `rs < rt` and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality
  - `beq $t5, $t6, label`
  - jump to `label` if `$t5 = $t6`
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt

- Can we figure out the idea? (fig. 4.17)



Supporting the 'slt' operation

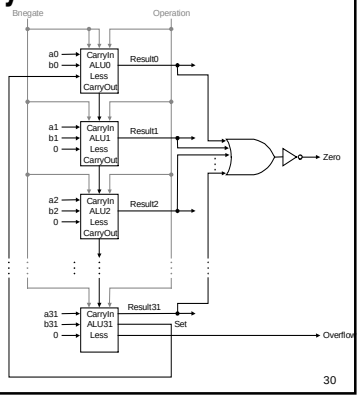


Test for equality

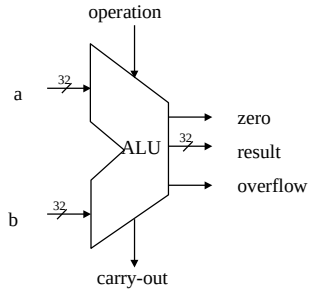
- a-b = 0 ⇔a=b
- Notice control lines:

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

- *Note: zero is a 1 when the result is zero!*
- *Zero output is always calculated*



ALU symbol



Conclusions

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

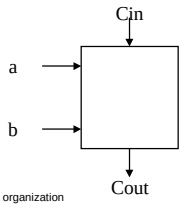
Can you see the ripple? How could you get rid of it?

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$
$c_2 = b_1c_1 + a_1c_1 + a_1b_1$ $\quad c_2 = (..subst\ c_1..)$
$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ $\quad c_3 =$
$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ $\quad c_4 =$

Not feasible! Why not?

Carry-lookahead adder (1)

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i\ b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$



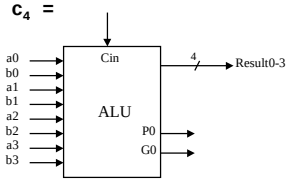
Cout = Gi + PiCin

Carry-lookahead adder (2)

- Did we get rid of the ripple?

$c_1 = g_0 + p_0c_0$
$c_2 = g_1 + p_1c_1$ $\quad c_2 = g_1 + p_1(g_0 + p_0c_0)$
$c_3 = g_2 + p_2c_2$ $\quad c_3 =$
$c_4 = g_3 + p_3c_3$ $\quad c_4 =$

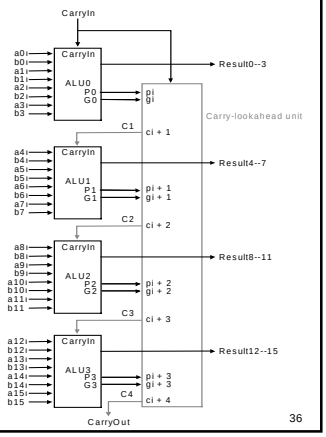
- Feasible ?



$P0 = p_0.p_1.p_2.p_3$
$G0 = g_3+(p_3.g_2)+(p_3.p_2.g_1)+(p_3.p_2.p_1.g_0)$

Carry-lookahead adder (3)

- Use principle to build bigger adders
- Can't build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!



Multiplication (1)

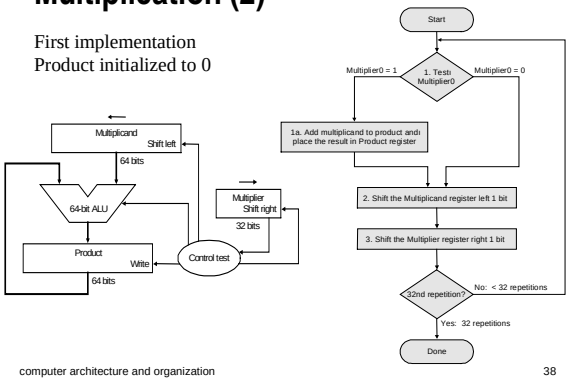
- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on gradeschool algorithm

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

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

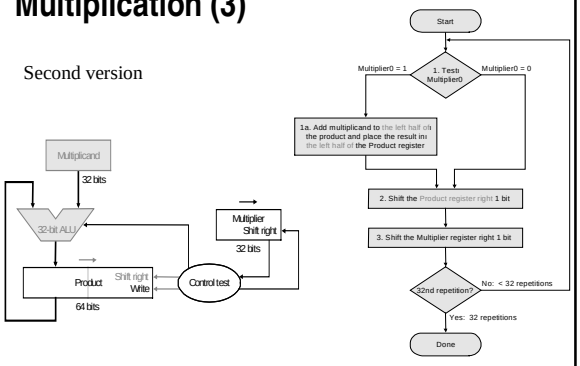
Multiplication (2)

First implementation
Product initialized to 0



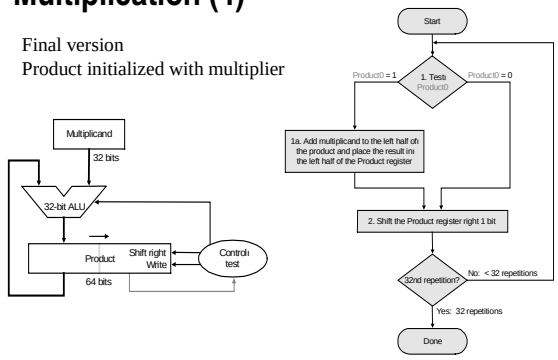
Multiplication (3)

Second version



Multiplication (4)

Final version
Product initialized with multiplier



Fast multiply: Booth's Algorithm

- Exploit the fact that: 011111 = 100000 - 1
  Therefore we can replace multiplier, e.g.:

  0 0 0 1 1 1 1 1 0 0 = 0 0 1 0 0 0 0 0 -1 0 0
- Rules:

Current bit	*Bit to the right*	*Explanation*	*Operation*
1	0	Begin 1s	Subtract multiplicand
1	1	Middle of 1s	nothing
0	1	End of 1s	Add multiplicand
0	0	Middle of 0s	nothing

Booth's Algorithm (2)

- Booth's algorithm works for signed 2's complement as well (without any modification)
- Proof: let's multiply b * a
  $(a_{i-1} - a_i)$ indicates what to do: 0 : do nothing
  +1: add b
  -1 : subtract

We get b*a = $\sum_{i=0}^{31} (a_{i-1} - a_i) * b * 2^i =$

$$b * \left[a_{31} * -2^{31} + \sum_{i=0}^{30} a_i * 2^i \right]$$

This is exactly what we need !

Division

- Similar to multiplication: repeated subtract
- The book discusses again three versions

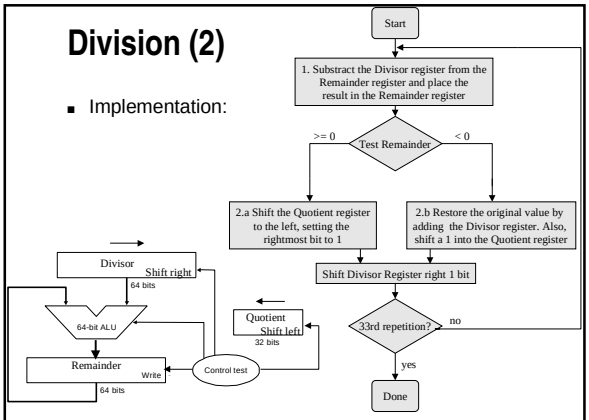
Divide (1)

- Well known algorithm:

```
                    Dividend
Divisor  1000/1001010\1001  Quotient
              -1000
                  10
                 101
                1010
               -1000
                  10   Remainder
```


Division (2)

- Implementation:



Multiply / Divide in MIPS

- MIPS provides a separate *pair of 32-bit registers* for the result of a multiply and divide: Hi and Lo

```
mult  $s2,$s3   # Hi,Lo = $s2 * $s3
div   $s2,$s3   # Hi,Lo = $s2 mod $s3,
                          $s2 / $s3
```

- Copy result to general purpose register

```
mfhi  $s1       # $s1 = Hi
mflo  $s1       # $s1 = Lo
```

- There are also unsigned variants of mult and div: `multu` and `divu`

Shift instructions

- `sll`
- `srl`
- `sra`

- Why not '`sla`' instruction ?

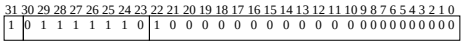
Shift: a quick way to multiply and divide with power of 2 (strength reduction). Is this always allowed?

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: $(-1)^{sign} \times \text{significand} \times 2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision : 8 bit exponent, 23 bit significand
  - double precision: 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: $(-1)^{sign} \times (1+significand) \times 2^{exponent - bias}$
- Example:
  - decimal: -.75 = -3/4 = -3/2^2
  - binary : -.11 = -1.1 x 2^{-1}
  - floating point: exponent = 126 = 01111110
  - IEEE single precision:



Floating Point Complexities

- Operations more complicated: align, renormalize, ...
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round, and additional sticky bit (indicating if one of the remaining bits unequal zero
  - 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!

Floating Point on MIPS

- Separate register file for floats: 32 single precision registers; can be used as 16 doubles
- MIPS-1 floating point instruction set (pg 288)
  - addition add.f (f =s (single) or f=d (double)
  - subtraction sub.f
  - multiplication mul.f
  - division div.f
  - comparison c.x.f where x=eq, neq, lt, le, gt or ge
    - sets a bit in (implicit) condition reg. to true or false
  - branch bc1t (branch if true) and bclf (branch if false)
    - c1 means instruction from *coprocessor one* !
  - load and store: lwc1, swc1
- Study examples on page 293, and 294-296

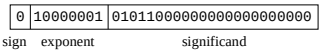
Floating Point on MIPS

- MIPS has 32 single-precision FP registers (`$f0,$f1, …,$f31`) or 16 double-precision (`$f0,$f2,...`)
- MIPS FP instructions:

FP add single	add.s	$f0,$f1,$f2	$f0 = $f1+$f2
FP substract single	sub.s	$f0,$f1,$f2	$f0 = $f1-$f2
FP multiply single	mul.s	$f0,$f1,$f2	$f0 = $f1x$f2
FP divide single	div.s	$f0,$f1,$f2	$f0 = $f1/$f2
FP add double	add.d	$f0,$f2,$f4	$f0 = $f2+$f4
FP substract double	sub.d	$f0,$f2,$f4	$f0 = $f2-$f4
FP multiply double	mul.d	$f0,$f2,$f4	$f0 = $f2x$f4
FP divide double	div.d	$f0,$f2,$f4	$f0 = $f2/$f4
load word coprocessor 1	lwc1	$f0,100($s1)	$f0 = Memory[$s1+100]
store word coprocessor 1	swc1	$f0,100($s1)	Memory[$s1+100] = $f0
branch on copr.1 true	bc1t	25	if (cond) goto PC+4+100
branch on copr.1 false	bc1f	25	if (!cond) goto PC+4+100
FP compare single	c.lt.s	$f0,$f1	cond = ($f0 < $f1)
FP compare double	c.ge.d	$f0,$f2	cond = ($f0 >= $f2)

Conversion: decimal ➨ IEEE 754 FP

- Decimal number (base 10)
  $123.456 = 1x10^2+2x10^1+3x10^0+4x10^{-1}+5x10^{-2}+6x10^{-3}$
- Binary number (base 2)
  $101.011 = 1x2^2+0x2^1+1x2^0+0x2^{-1}+1x2^{-2}+1x2^{-3}$
- Example conversion: 5.375
  - Multiply with power of 2, to get rid of fraction:
    $5.375 = 5.375x16 / 16 = 86 \times 2^{-4}$
  - Convert to binary, and normalize to 1.xxxxx
    $86 \times 2^{-4} = 1010110 \times 2^{-4} = 1.01011 \times 2^2$
  - Add bias (127 for single precision) to exponent:
    exponent field = 2 + 127 = 129 = 1000 0001
  - IEEE single precision format (remind the leading “1” bit):



Floating point on Intel 80x86

- 8087 coprocessor announced in 1980 as an extension of the 8086 (similar 80287, 80387)
- 80 bit internal precision (extended double format)
- 8 entry stack architecture
- addressing modes:
  - one operand = TOS
  - other operand is TOS, ST(i) or in Memory
- Four types of instructions (table 4.49 page 303):
  - data transfer
  - arithmetic
  - compare
  - transcendental (tan, sin, cos, arctan, exponent, logarithm)

Fallacies and pitfalls

- Associative law does not hold:

  (x+y) + z is not always equal x + (y+z)

  (see example pg 304)

Chapter Four 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 (i.e., algorithms and implementation).

Exercises

From Chapter four:

- 4.1, 4.2, 4.4 - 4.8
- 4.10, 4.12, 4.13
- 4.14
- 4.16, 4.20
- 4.25
- 4.36
- 4.40
- 4.49, 4.50