

Number Representation

Ref: Chapter 4

Unsigned Integers

- We are already familiar with the representation of positive (unsigned) integers.
- Positional Notation: each bit represents a power of 2.

... 512 256 128 64 32 16 8 4 2 1

32 Bit Words

- Range of values is:

0_{10} : 00000000000000000000000000000000

$4,294,967,295_{10}$: 11111111111111111111111111111111

- What if we add $4,294,967,295+1$?
– we get 0! (and an *overflow*)

Unsigned Binary Addition

- Use the same *algorithm* we use for decimal addition:

$$\begin{array}{r} \\ \leftarrow \text{Carry} \\ + \\ \hline \end{array}$$

CompOrg Fall 2000 - Number Representation

4

What about negative integers?

- Options:
 - use one of the 32 bits as a *sign* bit.
 - add another bit, 33rd bit is the sign.
- Both lead to 2 representations for the value 0 (+0 and -0).
 - could cause problems for programmers.

CompOrg Fall 2000 - Number Representation

5

Alternative representation

- Most computers don't use a *sign and magnitude* representation for signed integers.
- Signed numbers are represented using *2s complement representation*
 - simplifies the hardware – the same circuit that adds unsigned integers can be used to add signed integers!

CompOrg Fall 2000 - Number Representation

6

2s complement

- *Leading zeros*: the integer is positive
- *Leading ones*: the integer is negative

00101010	10000000
00000001	10110101
01111111	11111111
positive 8 bit integers	negative 8 bit integers

CompOrg Fall 2000 - Number Representation

7

8 bit 2s complement

- positive numbers
 - 00000001 1_{10}
 - 01111111 127_{10}
 - negative numbers
 - 11111111 -1_{10}
 - 10000000 -128_{10}
 - zero: 00000000
-

CompOrg Fall 2000 - Number Representation

8

8 bit 2s complement *positional notation*

<u>128</u>	<u>64</u>	<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>
-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$10010010 = 1x-2^7 + 1x2^4 + 1x2^1$$

CompOrg Fall 2000 - Number Representation

9

8 bit *signed* addition

$$\begin{array}{r}
 \begin{array}{r}
 \overset{1}{\leftarrow \text{Carry}} \\
 00001001 \\
 + \underline{11101010} \\
 \hline
 11110011
 \end{array}
 \qquad
 \begin{array}{r}
 9 \\
 + \underline{-22} \\
 \hline
 -13 \\
 \uparrow \\
 \text{Base 10}
 \end{array}
 \end{array}$$

The algorithm is the same!

Try another one

$$\begin{array}{r}
 11111111 \leftarrow \text{Carry} \\
 11111111 \\
 + \underline{11111111} \\
 \hline
 111111110 \\
 \uparrow \\
 \text{Too many bits! That's OK (this time)} \\
 \text{- just ignore the last bit (more on this later)!}
 \end{array}$$

32 bit *signed* integers

- Same idea as 8 bit signed integers, but the MS bit is -2^{31} .

	...	128	64	32	16	8	4	2	1
$-2,147,483,648$...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- Range of values is:

-2,147,483,648 to 2,147,483,647

32 bit 2s complement integers

0_{ten} is 00000000000000000000000000000000

-1_{ten} is 11111111111111111111111111111111

1_{ten} is 00000000000000000000000000000001

CompOrg Fall 2000 - Number Representation 13

32 bit signed addition

- Same algorithm!
- Just like in 8-bit addition, sometimes having too many bits in the result doesn't matter!
 - sometimes it does (overflow is possible).

CompOrg Fall 2000 - Number Representation 14

addition algorithm

- How can we tell when too many bits in the result means *overflow* and when it's OK?
- *overflow* means the right answer won't fit !
- If the sign of the numbers are the same:
 - too many bits means overflow
 - otherwise everything may be OK (ignore the last bit).

CompOrg Fall 2000 - Number Representation 15

Overflow and 8 bit addition

$$\begin{array}{r} 1111 \\ 01111000 \\ + 01111000 \\ \hline 11110000 \end{array}$$

It fits, but it's still overflow!

120
+120
-16

??

Overflow, overflow, overflow

If the sign of the numbers is the same
-and-
the sign of the result is different
(than the sign of the numbers)

We have overflow!

Converting binary numbers

- Suppose you get a test question that asks:
 - what is the 8 bit 2s complement representation for the number **-85**?
- For 8 bits, it's probably not that hard to go through the positional notation and figure things out.
- What if the question is for 32 bits!

neg ↔ pos shortcut

To negate any 2s complement number:

invert all the bits.

add 1 (binary addition).

Negating -1:

11111111 $\xrightarrow{\text{invert}}$ 00000000 $\xrightarrow{+1}$ 00000001

CompOrg Fall 2000 - Number Representation

19

Exercise

- What is the 8 bit 2s complement representation for the following:

17

-17

-85

CompOrg Fall 2000 - Number Representation

20

Important Note!

- “2s complement” (or “twos complement”) does not mean *negative*!
- 2s complement is a representation used to represent *signed* integers, not just negative integers!

CompOrg Fall 2000 - Number Representation

21

Converting 8 to 32 bit

- Conversion from 8 bit signed integers to 32 bit signed integers is easy:
 - take the leftmost bit of the 8 bit integer and make 24 copies of it – put them all on the left.
 - *sign extension*
- Remember the **lb** instruction (load byte).
- The byte is *sign extended* in the 32 bit register!

CompOrg Fall 2000 - Number Representation

22

Comparing numbers

- The **slt** instruction does a *signed* comparison.
 - 11...1111** is less than **00...0000**
- The **sltu** instruction does an *unsigned* comparison.
 - 11...1111** is *not* less than **00...0000**

CompOrg Fall 2000 - Number Representation

23

Subtraction

- Subtraction is easy: $A - B$
 - negate B and add to A.
 - we can use our good buddy, the binary addition algorithm.
- Signed vs. unsigned subtraction use the same algorithm (just like addition).
 - the only difference is the handling of *overflow*

CompOrg Fall 2000 - Number Representation

24

Handling Overflow

- Some high level languages require that the processor *detect* overflow.
 - detect means “let the program know”.
 - Fortran requires this, C does not.
- MIPS provides 2 kinds of addition & subtraction instructions:
 - signed: detect overflow
 - unsigned: ignore overflow

CompOrg Fall 2000 - Number Representation

25

MIPS instructions

- **add, sub, addi**: all are *signed* operations.
 - overflow is detected and an *interrupt* is generated.
- **addu, subu, addiu**: are *unsigned* operations.
 - overflow is ignored.

CompOrg Fall 2000 - Number Representation

26

Detecting Overflow

- The following instruction/conditions mean overflow has occurred:
 - **add**: both operands positive and negative result.
 - **add**: both operands negative and positive result.
 - **sub**: pos - neg and negative result.
 - **sub**: neg - pos and positive result.

CompOrg Fall 2000 - Number Representation

27

lbu Instruction

- **lbu** is “load byte unsigned”.
- Moves a byte from memory to the rightmost 8 bits of a register.
- Does *not* do sign extension – fills with 0s.
– **lb** does do sign extension!

CompOrg Fall 2000 - Number Representation

28

Logical Operations

- There are instructions that compute AND and OR. Hooray!
- These instructions operate on an entire word, not just on a single bit.
– bit-by-bit operation

CompOrg Fall 2000 - Number Representation

29

and destreg, reg1, reg2

Computes bit-by-bit AND of *reg1* and *reg2* and stores the result in *destreg*.

reg1: 00101001001011100100101011011111

reg2: 11011010101100101011010101100101

destreg: 0000100000100010000000001000101

CompOrg Fall 2000 - Number Representation

30

or destreg, reg1, reg2

Computes bit-by-bit OR of *reg1* and *reg2* and stores the result in *destreg*.

reg1: 00101001001011100100101011011111

reg2: 11011010101100101011010101100101

destreg: 11111011101111101111111111111111

More Logical Instructions

ori destreg, reg1, const

andi destreg, reg1, const

nor destreg, reg1, reg2

not destreg, srcreg pseudoinstruction

xor destreg, reg1, reg2

xori destreg, reg1, const

Logical Immediates

- ***andi***, ***ori*** and ***xori*** are *I-type* instructions
 - 16 bit constant.
- The 16 bit immediate value is extended with all 0s before the logical operation.
 - treated as *unsigned*.

Shift Gears

- There are also logical instructions that *shift* the bits in a register to the left or right.
 - typically used to align *bit fields* for operations.
- A *shift amount* specifies how many bits to shift.

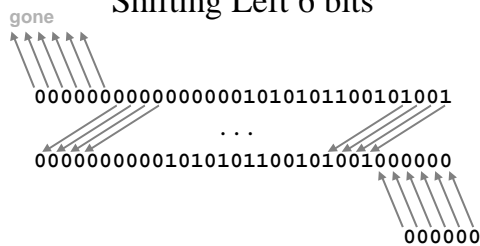
Shift Left

`sll destreg, srcreg, const`

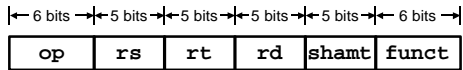
shift left logical

- Shifts the bits in *srcreg* and store the result in *destreg*.
- *const* specifies the number of times to shift.
- empty bits (on the right) are filled with 0s.

Shifting Left 6 bits



Remember *R-Type* Instructions?



`sll` is an *R-type* instruction.

`shamt` is the shift amount.

5 bits can encode numbers up to 31

CompOrg Fall 2000 - Number Representation

37

Another Shift Left

`sllv destreg, srcreg, shamt_reg`

Shift Left Logical Variable

The shift amount is in the register `shamt_reg`

CompOrg Fall 2000 - Number Representation

38

Multiplication by shifting

- A shift left of 1 bit on an *unsigned* integer is the same as multiplication by 2.
- A shift left of n bits is multiplication by 2^n

CompOrg Fall 2000 - Number Representation

39

Shifting Right

```
srl destreg, srcreg, const  
srlv destreg, srcreg, shamt_reg
```

Both instruction fill with zeros on the left.

Shifting right by n bits is division by 2^n

CompOrg Fall 2000 - Number Representation

40

Arithmetic Shift

- There are also right shift instructions that fill the left with copies of the *sign bit* (the MS bit).
- Arithmetic shifting to the right on signed numbers is still division.

CompOrg Fall 2000 - Number Representation

41

Shift Right Arithmetic Instructions

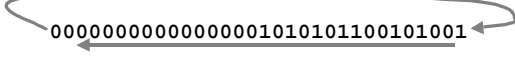
```
sra destreg, srcreg, const  
  
srav destreg, srcreg, shamt_reg
```

CompOrg Fall 2000 - Number Representation

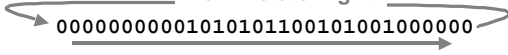
42

Rotate Instructions

`rol`: Rotate Left



`ror`: Rotate Right



`rol` and `ror` are pseudoinstructions

CompOrg Fall 2000 - Number Representation

43
