CSE471—Computer Design and Organization

Prof. Steven Burns *Autumn 1994*

Lecture Notes: Fourth Day

Computer Arithmetic

- We will cover Chapter 4 completely:
  - Integer representations
  - Addition and subtraction circuits
  - Multiplication and division circuits
  - IEEE floating-point representation
  - Floating-point addition and multiplication algorithms

Representations of Non-Negative Numbers

- Binary numbers

$$34_{10} = 100010_2$$

- Binary-coded decimal (BCD) numbers

$$34_{10} = 0011_2\ 0100_2$$

Representations of Signed Numbers

- Sign and magnitude: extra bit to represent the sign

$$-34_{10} = 1\ 100010$$

- Twos complement: most significant bit contributes a negative amount

$$\begin{aligned} -34_{10} &= 1011110_{\text{twos}} \\ &= -64 + 30 \end{aligned}$$

- Biased: store 63 (or any bias) plus the number

$$\begin{aligned} -34_{10} &= 11101_{\text{biased}} \\ &= 29 - 63 \end{aligned}$$

Twos Complement

- What are the largest and smallest numbers that can be expressed with n bits in twos complement form?
- Largest number: $0111\ldots1 = 2^{n-1} - 1$.
- Smallest number: $1000\ldots0 = -2^{n-1}$.
- How many numbers can be represented all together?
- How do you negate a twos complement number?

Overflow

- Let's look at an example for $n = 6$:

$$\begin{aligned} 011110 + 000011 &= 100001 \\ 30 + 3 &\neq -31 \end{aligned}$$

- For addition, we never get overflow if the operands have *opposite* signs. We can detect overflow by comparing the common signs of the operands with the sign of the result:

$$\mathit{overflow} = a_{\text{sign}}\, b_{\text{sign}}\, r'_{\text{sign}} + a'_{\text{sign}}\, b'_{\text{sign}}\, r_{\text{sign}}$$

- Overflow causes an exception on `add` but not `addu`.

Overflow With Subtraction

- For $a - b$, we can negate b and add. There is one exceptional case. What is it?
- For subtraction, we never get an overflow if the operands have the *same* sign. We have overflow if the signs of the operands are different and the sign of the first operand is different than the sign of the result:

$$\mathit{overflow} = a_{\text{sign}}\, b'_{\text{sign}}\, r'_{\text{sign}} + a'_{\text{sign}}\, b_{\text{sign}}\, r_{\text{sign}}$$

Extending Numbers and Immediate Instructions

- How do we convert 16-bit numbers into 32-bit numbers?
- Sign extension
- Zero extension
- The instructions `be`, `addi`, `addiu`, `slti`, and `sltiu` perform sign extension of their immediate values.
- Logical instructions, such as `andi` and `ori` perform zero extension of their immediate values.

Logical Operations and Shifts

- Bitwise logical operations: `and`, `or`, `xor`, `nor`.
- Shift operations: shift left logical, shift right logical. The instructions `sll` and `srl` use the *shamt* field of the R format. The instructions `sllv` and `srlv` use the shift amount specified by a third register.
- There is also a `sra` instruction.