Lecture 2, Jan 16, 2024

Number Representation

- Ultimately everything is represented in binary; but a number in binary is meaningless without an attached interpretation!
- Representing a number in binary requires us to know the signedness, bit width, and target representation
  - Integers can be represented in a variety of ways including sign and magnitude (using one bit for the sign and remaining for magnitude), 1's complement, and 2's complement
    - 2's complement has the advantage of not having two representations for zero and allowing for subtraction with normal addition logic
- To convert to binary, successively divide by 2 and take the remainder each time as the bit value, from right to left
- For non-integers, we use either a *fixed-point* or *floating-point* representation
  - In a fixed-point representation, a fixed number of bits is allocated to represent the integer part and another fixed number of bits for the fractional part
    - To get the fractional part, we successively multiply by 2 and take any integer part as the bit value, from left to right
    - Example: 27.2
      - Using 8-bit unsigned for the integer part gives `0001 1011`
      - Fractional part:
        - $0.1 \times 2 = 0 + 0.2$
        - $0.2 \times 2 = 0 + 0.4$
        - $0.4 \times 2 = 0 + 0.8$
        - $0.8 \times 2 = 1 + 0.6$
        - $0.6 \times 2 = 1 + 0.2$
        - $\dots$
      - The fractional bits would repeat forever as `0.0001100011...`
  - In a floating-point representation the position of the decimal point is not fixed
    - The bits are split into a sign bit, a signed exponent, and an unsigned *mantissa*
    - The number is recovered by multiplying the mantissa by a base raised to the power of the exponent, times the sign
    - Compared to fixed point, this can store a much wider range of numbers
    - However doing math on them requires them to be *denormalized* first, essentially converting this back to fixed point
      - This is slow, can lose precision, and can lead to bugs when the magnitudes of the floats differ by too much
- Note since the binary representation is a repeating decimal, if we truncate it at any finite digit count and convert back to decimal, we won't get the original number back
  - Often by converting a decimal number to binary, we lose precision; when we do arithmetic operations on these numbers the precision loss is compounded
  - Both fixed and floating point are *lossy*
- The result of an arithmetic operation may not fit in the number representation of the operands
  - A *carry-in* can be used to implement addition at bit widths higher than a single instruction allows
  - A *carry-out* indicates that the result doesn't fit within the bit width allocated
  - If the carry-out is unhandled, it becomes an *arithmetic overflow*
    - A 1 in the carry-out position indicates an overflow
    - If a carry-out has occurred, it is usually put into a special register
  - An *arithmetic underflow* can also happen when we subtract a larger number from a smaller one when we're working only with unsigned numbers
    - This doesn't happen with 2's complement since everything gets converted to addition, so it uses overflow for both numbers that are too small or too big
- Bits can be *shifted* or *rotated*
  - An arithmetic shift left shifts the MSB into the carry and 0 into the LSB

  - An arithmetic shift right shifts duplicates the MSB and shifts the LSB into the carry
- Two's complement representation is a common way to represent negative numbers
  - The MSB acts as a sign bit; a 1 in MSB indicates a negative number
  - The remaining bits are used to store the magnitude
  - If the number is negative, the magnitude is complemented (inverted) and has 1 added to it
  - This allows us to unify the addition and subtraction logic
  - Underflow or overflow is indicated by the carry-out being different from the carry-on into the MSB (sign bit) of the result
- When we move a signed value into storage that has more bits, we perform *sign extension*, i.e. duplicating the MSB to fill any unfilled bits
  - e.g. $0b11100000$ becomes $0b11111111'11100000$ upon sign extension to 16 bits
- For multiplication/division, most modern RISC systems will do this in a number of steps:
  1. Check the signs of the operands
  2. Take 2's complement of any negative operands
  3. Multiply/divide as unsigned
  4. If operands had different signs, take 2's complement of the result
- Number representations affect our choice of microcontroller – if we need to work with a lot of large numbers or floating-point numbers, we need to choose processors that have the corresponding features, otherwise performance will be very poor
- Data can be stored in multiple different locations:
  - Data memory (often but not always RAM)
  - Instruction memory (often non-volatile, non-runtime-modifiable memory)
  - Registers
  - Memory inherent to instruction
- Going down this list, the amount of memory available gets smaller but the memory is faster

Important

Key takeaways:
1. We need to be aware of our data representation limits
2. Any calculations using data from non-controlled sources must be assertion-checked to avoid errors
3. Use assertions to be more aware of potential overflow causes in intermediate math
4. Use completley specified typing for variables; do not assume anything about variable sizes, etc
5. Avoid non-deterministic stop conditions (i.e. have fallbacks), especially when external data is involved