G22.3350 — Honors Theory

Solution for HW2

March 4, 1999

1 Question 1: Why Two's Complement?

[You do not need to say all of the following to get full credit!]

Let us assume that the representation uses exactly n bits. If we are talking about infinite strings, as in the notes on register machines, the discussion can be modified slightly.

First notice that applying two's complement twice amounts to the identity operation. This is exactly what we expect for the negation of an integer.

Next, note that the 2's complement of 0 is still 0. In other words, there is only one representation for zero. In 1's complement or using a dedicated sign bit, there would be two representations of zero ($+0$ and -0). So using 2's complement, we can 2^n (instead of $2^n - 1$) integers using n bits. THIS IS ONE ADVANTAGE OF TWO'S COMPLEMENT.

Computing the negation of a number (i.e., computing the 2's complement is easy. Thus addition and subtraction uses essentially the same algorithm: to subtract y from x, first compute the 2's complement of y then add it to x.

Addition and multiplication of 2's complement can be carried out UNIFORMLY, without taking special actions depending on whether the values being added or multiplied are positive or negative. (For example, if you use 1's complement or use a dedicated sign bit, then you need to check the signs of the operands first).

EXERCISE: Prove this remark! MORE PRECISELY: Let w be a binary string of length n. Write $val(w)$ for the integer represented by w in two's complement. Let $num(w)$ denote the natural number represented by w as an unsigned. Then $val(w) = num(w)$ if the MSB of w is 0 and $val(w) = 2^n - num(w)$ otherwise. Suppose w, v are two such binary strings. Let u be the result of adding or multiplying w and v as if these are unsigned numbers. SHOW: if the addition or multiplication does not cause an overflow, then the result is correct EVEN if w and v are treated as 2's complement.

2 Question 2: Translating a code into a vector machine program

This is mostly mechanical, so I won't write out the details.

There is, however, one trick: our instruction SHIFT X,A is a leftshift when A is positive. If you want to do rightshift by the value in register A, you need to compute the negative (2's complement) of A. This cannot be done in constant time! It can be done in logarithmic time (see next question). But this is within the loop, and will make the overall cost to be log-square.

The solution is simply to initialize A to -1. If you now do a SHIFTLEFT A repeatedly, you get -2. -4, etc.

The exercise is a way to encourage you to describe vector machine algorithms in a high level language (using while loops, case statements, etc). Do this from now on!

3 Question 3: Addition in Logarithmic Time

NOTE: this problem was too hard (I should have given some hints. Never mind if you did not solve this, as I will grade generously.

3.1 Suppose you want to add A and B. Let the ith bits of A and B be A_i and B_i ($i = 1$) is the least significant bit. Let $Z = A + B$ and C be the corresponding carry vector. It is easy to see that $Z_i = 1$ iff $A_i + B_i + C_i$ is odd. We can write this as

$$Z = A \oplus B \oplus C \tag{1}$$

where $\oplus$ is exclusive-or. (It is easy to implement exclusive-or using the usual instruction set.)

Hence, it is enough to show how to compute the vector C in logarithmic time. Notice that a straightforward to compute C need $|C|$ steps (the ith carry bit C_i depends on $C_1, C_2, \ldots, C_{i-1}$). Instead, we must compute C_i in $O(\log i)$ steps.

3.2 The following algorithm solves the problem.

$G \leftarrow A \wedge B$; // *Generate carry bits*
$P \leftarrow A \vee B$; // *Generate propaget bits*
$I \leftarrow 1$; // *index register*
while $(P \neq 0)$ {
    $G \leftarrow G \vee ((G \uparrow I) \wedge P)$; // *spread G left*
    $P \leftarrow P \wedge (P \uparrow I)$;
    $I \leftarrow I \uparrow 1$;
}
return$(A \oplus B \oplus (G \uparrow 1))$;

This is called the "carry and propagate method" in hardward design. G and P are called the "generate" and "propagate" vectors, respectively. Let P^t, G^t $(t \geq 0)$ denote the values of P and G

after the tth iteration. Thus $P^0 = A \vee B$ and $G^0 = A \wedge B$. Notice that the algorithm returns (in line 9) the result

$$A \oplus B \oplus (G \uparrow 1)).$$

Comparing this with (1), we conclude that $G \uparrow 1$ must be the carry vector C. This is what we need to prove for the correctness of this algorithm.

Before we proving $C = G \uparrow 1$, let us see how to define C from P and G. Let $i > j \geq 1$. We say "i sees j" and write

$$i \to j$$

if $P_k = 1$ for all k such that $i > k \geq j$. It is easy to see that $C_i = 1$ if and only if there is some j such that $i \to j$ and $G_j = 1$.

3.3 Notice that P does not depend on G at all and it relatively easy to understand how P transforms. We introduce a helpful notation. For each $i = 1, \ldots, n$, define $d(i)$ to be the distance from the ith bit to the first 0 bit at position $\leq i$. By definition, $P_0 = 0$ so $d(i)$ is well-defined. For instance, $d(i) = 0$ iff $P_i = 0$, and $d(i) = 1$ iff $P_i = 1$ and $P_{i-1} = 0$.

E.g., if $P = 1, 1011, 1101$ (the comma is just to improve readability) then

$$P^1 = 1, 0011, 1000, \quad P^2 = 0, 0010, 0000, \quad P^3 = 0, 0000, 0000.$$

Note that P^t depends only on P and t but not on G, so it is easier to understand than G. Now define $d^t(i)$ analogously for P^j, namely, it is equal to the distance from the ith bit of P^t to the first 0 bit at position $\leq i$. Note that $d(i)$ is the same as $d^0(i)$. We have:

FACT P: The ith bit of P^t is 1 iff $d(i) \geq 2^t$.

We omit this proof. So as time progresses, more and more bits of P will be turned off. After $\lg n$ steps, P is identically 0. This proves that the algorithm takes $O(\log n)$ time.

3.4 Next, let us describe how G transforms. Notice that G is cumulative (if a bit of G is 1, it remains 1 in subsequent loops). Analogous to $d(i)$, let us define:

$$c(i) = \min_{j \geq 0} \{i - j : G_j = 1 \text{ and } \ d(i) > i - j\}.$$

If the minimization in this definition is over an empty set then we let $c(i) = \infty$. It is clear from this definition that if $c(i) < \infty$ then $d(i) > c(i)$. Also, it is not hard to see that $G_i = 1$ implies $c(i) = 0$ (note that $d(i) \geq 1$ in this case).

E.g., with P as above, suppose $G = 0, 0000, 1001$. Then $c(i) = 0$ iff $i = 1$ or 4. Also, $c(5) = 1, c(6) = 2, c(7) = \infty$.

In general, $c(i) < \infty$ and $P_i = 1$ implies $c(i + 1) = 1 + c(i)$.

FACT G: $G_i^t = 1$ iff $c(i) < 2^t$.

Proof. If $t = 0$ then $G_i^0 = 1$ iff $c(i) = 0$. So suppose $t > 0$. Note that, by line 5 of the algorithm,

$$G_i^t = 1 \Leftrightarrow G_i^{t-1} = 1 \text{or} \ \ ((G^{t-1} \uparrow 2^{t-1}) \wedge P^{t-1})_i = 1.$$

By induction, $G_1^{t-1} = 1$ iff $c(i) < 2^{t-1}$. But $G_i^{t-1} = 1$ implies $G_i^t = 1$. So the result is true if $G_i^{t-1} = 1$.

Suppose $G_i^{t-1} = 0$. Hence $c(i) \geq 2^{t-1}$. This means

$$c(i) = 2^{t-1} + c(i - 2^{t-1}). \tag{2}$$

But now,

$$\begin{array}{rll} G_i^t = 1 & \text{iff} & ((G^{t-1} \uparrow 2^{t-1}) \wedge P^{t-1})_i = 1 \\ & \text{iff} & (c(i - 2^{t-1}) < 2^{t-1}) \wedge (d(i) \geq 2^{t-1}) \\ & \text{iff} & c(i) < 2^t \quad \text{(by (2))} \end{array}$$

This proves FACT G. **Q.E.D.**

4 Question 4: Proof of Theorem A

The detailed hint given is practically all that there is to it! You just need to make sure that evaluating the predicate $BIT(i, j, k)$ takes space t^2.