Lecture C
NOTES ON REGISTER MACHINES

§1. Register Machines

A simple computational model introduced by Shepherdson and Sturgis (1963) is the *register machine model.* There is an infinite number of registers indexed by the integers. Each register stores an integer. We write $c_t(n)$ for the **contents** of register $n \in \mathbb{Z}$ at time $t \in \mathrm{N}$. We omit t when convenient and unambiguous. The following are instructions:

- ZERO n: This is the assignment $c(n) \leftarrow 0$.
- SUCC n: This is the assignment $c(n) \leftarrow c(n) + 1$. We also allow ADD n, m or SUB n, m as instructions: this means $c(n) \leftarrow c(n) \pm c(m)$.
- SET n, m: This is just $c(n) \leftarrow c(m)$.
- JUMP n, L: If $c(n) = 0$ then go to instruction L; else proceed normally to the next instruction. L is a positive integer.
- ACCEPT: The computation halts and accepts. If an output number is desired, this can be $c(0)$. The computation may also come to a halt because there is no next instruction – this is regarded as rejection.
- CHOOSE L, L': this allows the choice mode of computation.

A **register machine** is just a finite sequence of instructions. The instructions are numbered consecutively, starting from 1 (this is a convention for the labels in JUMP instructions). This model is obviously geared towards computation over natural numbers. We may introduce conventions about input and output, etc. Note that a register machine can only refer to a finite number of registers – and yet its computation need not be bounded because the registers can store arbitrarily large numbers.

§2. The RAM model

An extension of the register machines is to allow indirect addressing: we can use the contents of a register to name another register that will be used as the operand of our instructions. That is, instead of the argument n in the above instructions, we can also have an argument of the form "@n". This means that we are using the register $c(n)$ instead of the register n. The resulting model is called the **random-access memory model** (RAM model). This can be viewed as an abstract simple assembly language.

§3. Vector Machines

The RAM model here may be called **integer RAM model**. An alternative is to work with bit vectors. The motivation for vector machines is to introduce parallelism into the register model. This was first discussed by Pratt, Stockmeyer and Rabin (1974).

We now work with semi-infinite bit string $\{0,1\}^\omega$. Usually, such a string $w = a_0a_1a_2\cdots$ is written with the first bit a_0 on the left. But because we want to interprete w as a binary number, we write it as $\cdots a_2a_1a_0$. The bit a_0 is sometimes called the "least significant bit" (lsb). We are actually interested in a special subclass of $\{0,1\}^\omega$, namely those semi-infinite bit strings that is **ultimately constant**. Call such string **$*$-strings**. We can finitely represent a $*$-string as follows: $w = *a_na_{n-1}\cdots a_1a_0$ (where $a_i \in \{0,1\}$ mean the string in which the bit a_n is repeated infinitely to the left. For instance, $w_1 = *01011$ can also be written as $w_1 = *0001011$. Similarly, $w_2 = *10$ is equivalent to $w_2 = *11110$. The **length** of a $*$-string w is defined as follows: if u is the shortest finite string such that $w = *u$, then $|w| = |u|$. Note that $|w| \geq 1$.

We are interested in bitwise Boolean operations. Thus the **complement** of w_1 is denoted $\sim w_1$ and equals $*10100$. The **logical-and** of w_1 and w_2 is denoted

$$w_1 \wedge w_2 = *01011 \wedge *11110 = *01010.$$

Note that the rightmost bit of w_1 and w_2 are aligned for the purposes of such operations.

When viewed as integers, an ultimately constant semi-infinite bit strings are interpreted as a positive it is ultimately zero, and negative when ultimately one. We use the standard *two's complement* notation for negative numbers: that is, if w represents the positive number n, then $(\sim w) + 1$ represents the negative number $-n$. The complement $\sim w$ is also called *one's compement* of w. Eg, 5 and -5 in this notation are $*0101$ and $*1011$.

Exercise 1: For the purposes of representing negative numbers, what are advantages of two's complement over one's complement?

A vector machine (VM) is basically a register machine in which we use the following alternative set of instructions. Note that $c(n)$ now denotes the semi-infinite bit string in register n.

- ZERO n: This is the assignment $c(n) \leftarrow *0$.
- NOT n: This is the assignment $c(n) \leftarrow \sim c(n)$. We also allow AND n,m and OR n,m as instructions: $c(n) \leftarrow c(n) \wedge c(m)$ and $c(n) \leftarrow c(n) \vee c(m)$.
- SHIFT n,m: Here we left shift $c(n)$ by $c(m)$, where the vacated bits are set to 0. If $c(m)$ is negative, this is really a right shift.
- JUMP n, L: If $c(n) = *0$ then go to instruction L; else proceed normally to the next instruction.
- ACCEPT, CHOOSE L, L': This is similar to integer register machines.

It turns out that unrestricted shifting is too powerful for making important complexity distinctions. To restrict shifting, our programs must specify certain registers as "shift registers" (as opposed to the original "vector registers"). Only shift registers may be used as the second argument of $SHIFTn, m$. Moreover, if m is a shift register, then the only way to shift its contents are to use the limited instructions "$SHIFTLEFTm$" or "$SHIFTRIGHTm$". These instructions shift the contents of m by only one position. Boolean operations are not allowed to mix vector registers with shift registers.

Exercise 2: in unrestricted vector machines, how fast can the length of a vector grow? Use vector machines to recognize some language L in a way that uses unrestricted shifting in some essential way.

§4. Complexity of Vector Machines

We shall focus on determinstic vector machines only. **Time** measure on a vector machines is just the number of instructions executed by the machine until it halts. We need to be careful when defining space: the space usage of a register n is 0 if it is never referenced during a computation. Otherwise, it is

$$\max_t |c_t(n)|.$$

The **space** of a computation is the sum of the space usage over all registers. Let $VTIME(t)$ and $VSPACE(s)$ denote the complexity classes defined by vector machines that accepts in time t and space s, respectively.

Clearly VM's are a model of parallelism. Also VM time should really be identified with reversals on Turing machines. Note that VM's only accept languages over $\{0,1\}$. However, we shall say that a language $L \subseteq \Sigma^*$ is accepted by a VM if there is function $f : \Sigma \to \{0,1\}^*$ such that for all $x \in \Sigma^*$, the VM accepts $f(x)$ iff $x \in L$. The complexity classes $VTIME(t)$ and $VSPACE(s)$ will in this way be regarded as incorporating arbitrary languages as well.

The two key results here are:

THEOREM A: $VTIME(t) \subseteq DSPACE(t(n)(t(n) + \log n))$.

This proof uses Savitch's technique.

THEOREM B: $NSPACE(s) \subseteq VTIME(O(s(n) + \log n)^2)$.

This proof requires vector machine techniques for transposing a Boolean matrix M (and hence matrix muliplication) in time $O(\log |M|)$. This leads to Boolean matrix transitive closure in time $O(\log^2 |M|)$. The connection between transitive closure and turing machine computation is discussed in Chapter 2.6 of Lecture Notes.

Some programming exercises. We may give our registers names in the following program. This is similar to the use of defined constants in assembly languages to help readability. Our registers are named A, B, etc. Let X, Y be the input registers. We write $A \uparrow B$ for left shift by B, and $A \downarrow B$ for right shift.

(1) We can recognize negative vectors w in time $O(\log |w|)$ on a VM. Here is the program:

```
A ← 1;
while (X ≠ 0 and ∼ X ≠ 0) do
      X ← X ↓ A;
      A ← A ↑ 1;
```

Note that A is a shift register.

Exercise 3: rewrite this as vector machine program.

(2) The sum of two numbers X and Y can be computed in time $O(\log(|X| + |Y|))$.

Exercise 4: show this. Unlike Exercise 3, you may henceforth use higher level informal constructs, justifying your constructs when necessary.

(3) The product of two non-negative numbers X and Y can be computed in time $O(\log(|X|+|Y|))$.

Exercise 5: show this.