Philosophy 3
Fall 2008
Prof. Dennett

The Secrets of Computer Power Revealed

We are surrounded by computers these days, and there seems to be no limit on what they can do if properly equipped with input and output interfaces–transducers and effectors–but although it is often said, as a sort of mantra, that a computer is a Universal Turing Machine, and that a Universal Turing Machine can compute any computable function, this is not well understood, even by some people working on computers. The underlying reasons for computer power are philosophically interesting in themselves, and worth understanding at an elementary level. This series of simple expositions and exercises is intended to provide that understanding.

I start by considering what is probably the simplest imaginable computer, a *Register Machine*, and seeing just what its powers are and why. I then go on to show how a *Turing Machine*, and a *Von Neumann Machine* (such as your laptop) are "just" Register Machines with speedups and more memory. We can then also see how other "architectures" could further multiply the speed and capacity of our basic machine, the Register Machine. The architecture of the human brain is, of course, one of the most interesting and important architectures to consider. In *Kinds of Minds*, we trace some of the key innovations in evolutionary history that lead from the simplest proto-minds (of macromoleules and bacteria) through animal brains/minds to brains/minds like ours. In this brief account, we will consider a parallel *imaginary* evolution from the simplest computers through your laptop to something that is not yet built, but describable in sketch at least: a computer that equals the powers of your mind. Many experts–not just philosophers, but neuroscientists and psychologists and linguists and other cognitive scientists–have argued that "the computer metaphor" for the human brain/mind is deeply misleading, and, more dramatically, that brains can do things that computers can't do. Usually, if not always, these criticisms presuppose a very naive view of what a computer is or must be, and end up proving only the obvious (and irrelevant) truth that brains can do lots of things that your laptop can't do (given its meager supply of transducers and effectors, its paltry memory, its speed limit). If we are to evaluate these strong skeptical claims about the powers of computers *in general*, we need to understand where computer power *in general* comes from and how it is, or can be, exercised.

1. The Register Machine

This brilliant idea was introduced at the dawn of the computer age by the logician Hao

Wang (a student of Gödel's, by the way, and a philosopher).[1] It is an elegant tool for thinking, and I want you to have this tool in your own kit. It is not anywhere near as well known as it should be.

A Register Machine is an idealized, imaginary (but perfectly possible) computer that consists of nothing but some (finite number of) *registers* and a *processing unit.*

The *registers* are memory locations, each with a unique address (register 1, register 2, register 3, . . .) and each able to have, as *contents*, a single integer (0,1,2,3, ...). For added vividness you can think of each register as a large box, which can contain any number of beans, from 0 to . . . . however large the box is . . . . We usually consider the boxes to be capable of holding *any* integer as contents, which would require infinitely large boxes, of course. Very large boxes will do for our purposes.

The *processing unit* contains a program that tells it exactly what to do. There are only three instructions it can follow:

End

or

Increment register *n* (add 1 to the contents of register *n*) and go to step *m*

or

Decrement register *n* (subtract 1 from the contents of register *n*) and go to step *m*

The *Dec*rement instruction works just like the *Inc*rement instruction, except for a single complication: what should it do if the number in register *n* is 0? It cannot subtract 1 from this (since registers cannot hold negative integers as contents), so, stymied, it must *Branch*. That is, it must go to some new place in the program to get its next instruction. This requires every *Dec*rement instruction to list the place in the program to go to next if the current register has content 0. So the full definition is:

Decrement register *n* (subtract 1 from the contents of register *n*) and go to step *m*
OR (if you can't decrement register *n*) *Branch* to step *p* (*Deb* for short).

At first glance, one would not think such a simple machine could do anything very interesting; all it can do is *put a bean in the box* or *take a bean out of the box* (if it can find one–and branch to another instruction if it can't). In fact, however, it can compute anything your laptop can compute.

Let's start with simple addition. Suppose you wanted it to *add* the contents of one register (let's say register 1) to the contents of another register (register 2). So, if register 1 has contents

---

[1]Hao Wang, 1957 "A Variant to Turing's Theory of Computing Machines", *Journal of the Association for Computing Machinery*, pp63-92.

[3] and register 2 has contents [4] we want the program to end up with register 2 having contents [7] since 3 + 4 = 7. Here is a program that will do the job, written in a simple language we can call RAP, for Register Assembly Programing:

program 1: ADD [1,2]

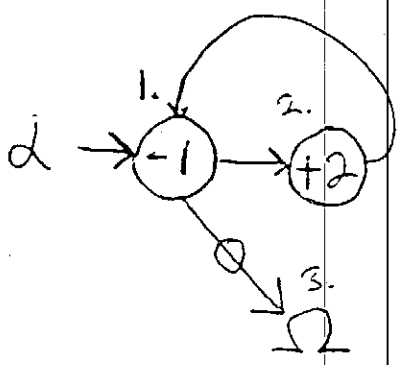
Step	Instruction	Register	go to step	[branch to step]
1.	*Deb*	1	2	3
2.	*Inc*	2	1	
3.	*End*			

The first two instructions form a simple *loop*, decrementing register 1 and incrementing register 2, over and over, *until register 1 is empty*, which the processing unit notices and thereupon *branches* to step 3, which tells it to halt. The processing unit cannot tell what the contents of a register is except in the case where the contents are 0. In terms of the beans-in-boxes image, you can think of the processing unit as blind, unable to see what is in a register until it is empty, something it can detect by groping. But in spite of the fact that it cannot tell, in general, what the contents of its registers are, if it is given program 1 to run, it will always add the contents of register 1 to the contents of register 2 and then stop. (Can you see why this must always work? Go through a few cases to make sure.)

Exercise 1

a. How many steps will it take the Register machine to add 2 + 5 and get 7, running Program 1 (counting End as a step)?
b. How many steps will it take to add 5 + 2?
c. What conclusion do you draw from this?

There is a nice way to diagram this process, in what is known as a *flow graph*. Each circle stands for an instruction. The number inside the circle stands for the address of the register to be manipulated. The program always starts at α, alpha, and stops when it arrives at Ω, omega. The arrows lead to the next instruction. Note that every *Deb* instruction has two outbound arrows, one for where to go when it can decrement, and one for where to go when it can't decrement, because the contents of the register is 0. (*branching on zero*).

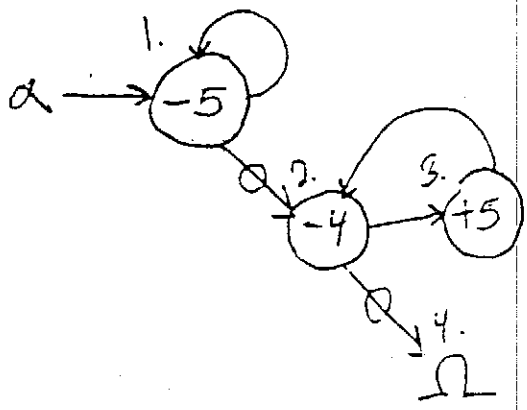


Now let's write a program that simply *moves* the contents of one register to another register:

program 2: MOVE [4,5]

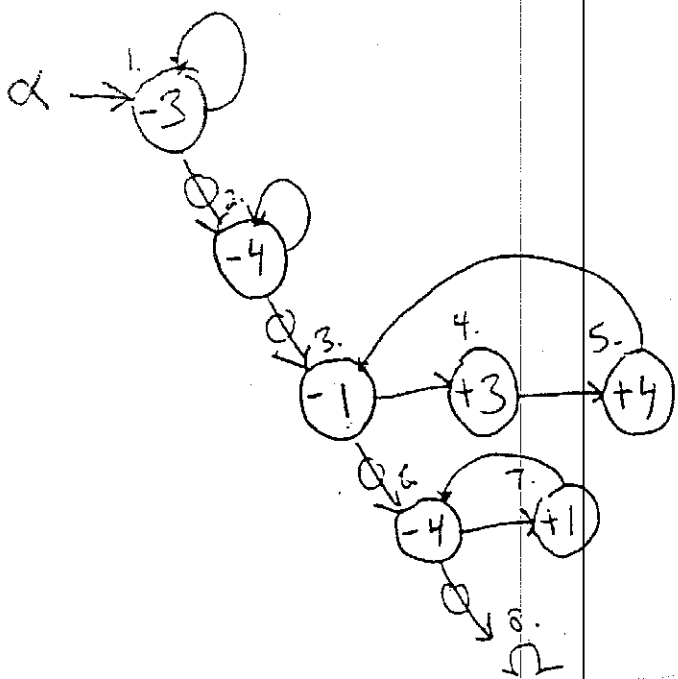
Step	Instruction	Register	go to step	[branch to step]
1.	*Deb*	5	1	2
2.	*Deb*	4	3	4
3.	*Inc*	5	2	
4.	*End*			

Here is the flow graph:



Notice that the first loop in this program cleans out register 5, so that whatever it had as contents at the beginning won't contaminate what is built up in register 5 by the second loop (which is just our addition loop, adding the contents of register 4 to the [0] in register 5). This initializing step is known as *zeroing out* the register, and it is a very useful, standard operation. You will use it constantly to prepare registers for use.

A third simple program *copies* the contents of one register to another register, leaving the original contents unchanged. Consider the flow graph, and then the program:



program 3: COPY [1,3]

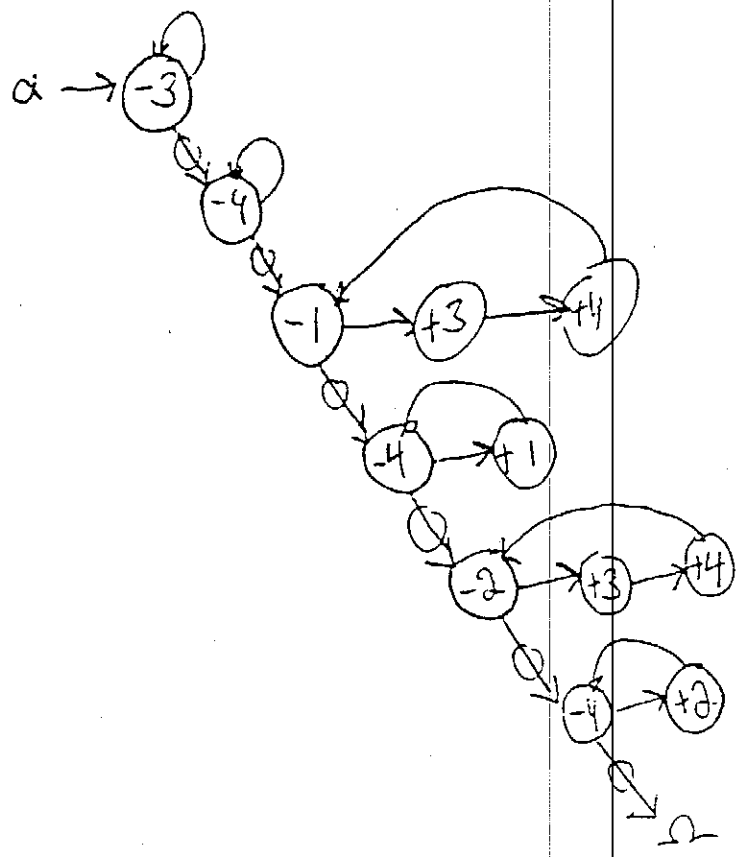
Step	Instruction	Register	go to step	[branch to step]
1.	*Deb*	3	1	2
2.	*Deb*	4	2	3
3.	*Deb*	1	4	6
4.	*Inc*	3	5	
5.	*Inc*	4	3	
6.	*Deb*	4	7	8
7.	*Inc*	1	6	
8.	*End*			

This is certainly a roundabout way of copying, since we do it by first *moving* the contents of register 1 to register 3 while making a duplicate copy in register 4, and then moving that copy back into register 1. But it works. Always. No matter what the contents of registers 1, 3, and 4 are at the beginning, when the program halts, whatever was in register 1 will still be there and a copy of those contents will be in register 3.

With *moving, copying*, and *zeroing out* in our kit, we are ready to go back to our addition program and improve it. Program 1 puts the right answer to our addition problem in register 2, but in the process it destroys the original contents of registers 1 and 2. We might want to have a fancier addition program, that saves these values for some later use, while putting the answer

somewhere else. So let's consider the task of adding the contents of register 1 to the contents of register 2, putting the answer in register 3 and leaving the contents of registers 1 and 2 intact.

Here is a flow graph that will accomplish that:



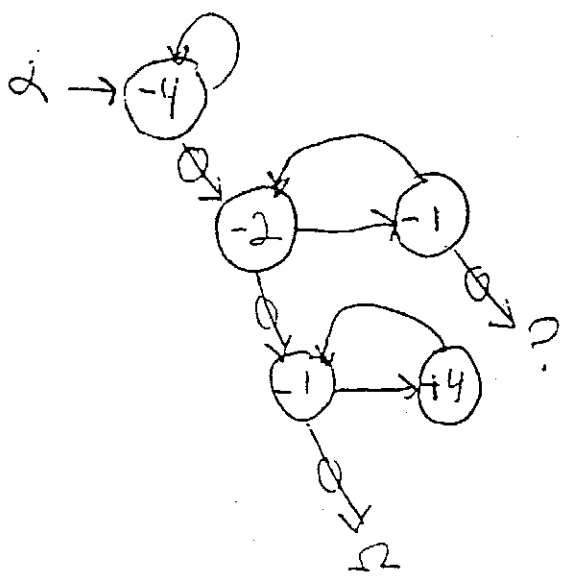
We can analyze the loops, to see what each does. First we zero out the *answer register*, register 3, and then we zero out a spare register (register 4) to use as a temporary holding tank or *buffer*. Then we copy the contents of register 1 to both 3 and 4, and move those contents back from the buffer to 1, restoring it (and in the process, zeroing out register 4 for use again as a buffer). Then we repeat this operation using register 2, having the effect of adding the contents of register 2 to the contents we'd already moved to register 3. When the program halts, buffer 4 is empty again, the answer is in register 3, and the two numbers we added are back in their original places, registers 1 and 2.

Here is the RAP program, which puts all the information in the flow graph in the form that the processing unit can read:

program 4: Non-destructive ADD [1,2,3]

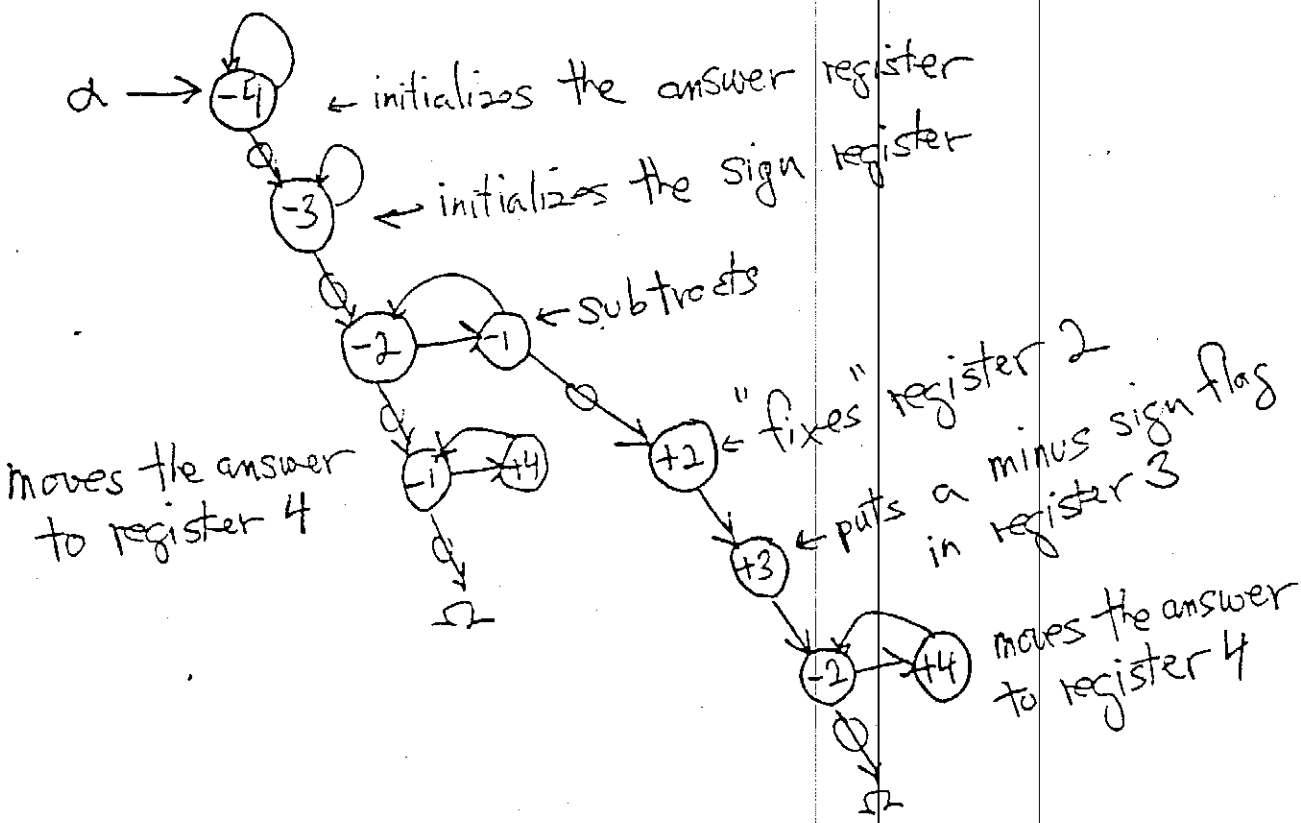
Step	Instruction	Register	go to step	[branch to step]
1.	*Deb*	3	1	2
2.	*Deb*	4	2	3
3.	*Deb*	1	4	6
4.	*Inc*	3	5	
5.	*Inc*	4	3	
6.	*Deb*	4	7	8
7.	*Inc*	1	6	
8.	*Deb*	2	9	8.
9.	*Inc*	3	10	
10.	*Inc*	4	11	
11.	*Deb*	4	12	13
12.	*Inc*	2	11	
13.	*End*			

Now turn to subtraction. Here is a first stab at a flow graph for subtracting the contents of register 2 from the contents of register 1, putting the answer in register 4. Can you see what is wrong with it?



This will only work when the contents of register 1 is greater than the contents of register 2. But what if this isn't so? Register 1 will "zero out" halfway through one pass in the subtraction loop, before it can finish the subtraction. What should happen then? We can't just ask the computer to

end, for this leaves the wrong answer (0) in register 4. We can use this zeroing out to start a new process, which first backs up half a loop and undoes the provisional decrementing from register 2. At this point the contents of register 2 (not register 1) gives the right answer if we interpret it as a *negative* number, so the simplest thing to do is to *move* those contents to register 4 (which is already zeroed out) and put a sign somewhere indicating that the answer is a negative number. The obvious thing to do is to reserve a register for just this task–let's say register 3. Zero it out at the beginning, along with register 4, and then have the program put a "flag" in register 3 as the sign of the answer, with [0] meaning [+] and [1] meaning [-]. Here is the flow graph, with comments explaining what each step or loop does. (**You can put such comments in your RAP programs, in between # marks. They are for you and other human beings; RodRego will ignore them.**):



Exercise 2

a. Write the RAP program for this flow graph. (Note that since the program branches, you can number the steps in several different ways. It doesn't matter which way you choose as long as the "go to" commands point to the right steps.)
b. What happens when the program tries to subtract 3 from 3 or 4 from 4?
c. What possible error is prevented by zeroing out register 3 before trying the subtraction at step 3 instead of after step 5?

With addition and subtraction under our belts, multiplication and division are easily devised. Multiplying *n* times *m* is just adding *n* to itself *m* times. So we can instruct the computer to do just that, using one register as a *counter*, counting down from *m* to 0 by decrementing once each time the addition loop is completed.

Exercise 3

a. Draw a flow graph (and write the RAP program) for multiplying the contents of register 1 by the contents of register 3, putting the answer in register 5.

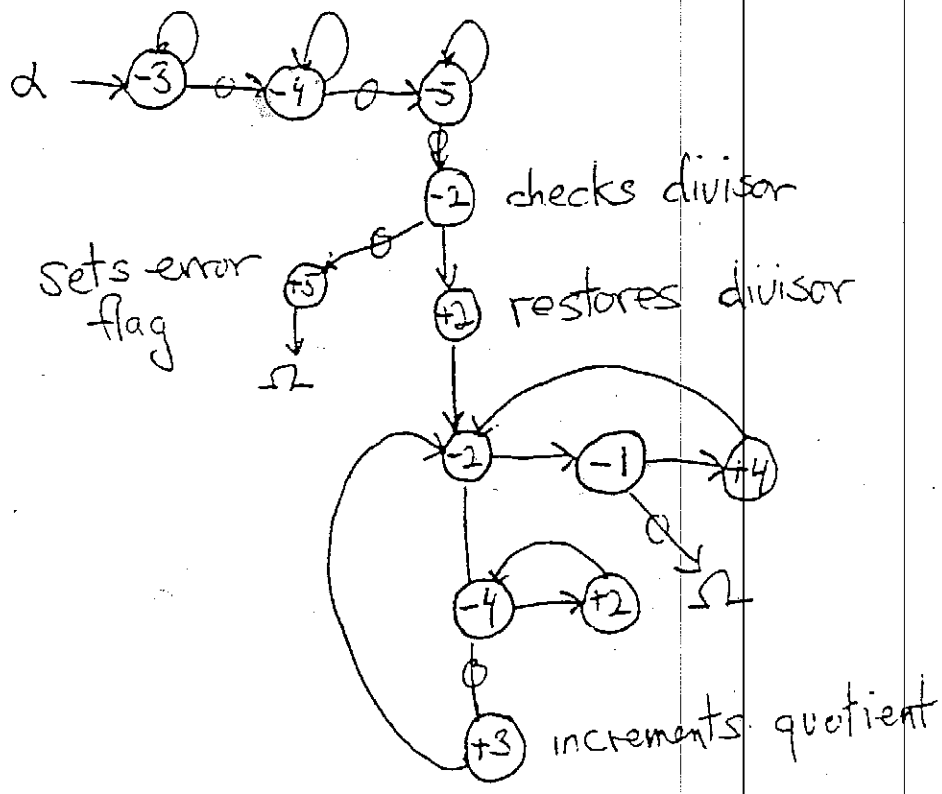
b. Optional! Using Copy and Move, improve your multiplier: when it stops, the original contents of register 1 and register 3 are restored, so that you can easily check the inputs and output for correctness after a run.

c. Optional! Does it take longer for your RAP program to multiply 8 times 2 than to multiply 2 times 8? Would it save time to "test" the numbers in register 1 and register 3 to see which is bigger before having the Register Machine "decide" which of two paths to take? Draw a flow graph and write a RAP program that examines the contents of register 1 and register 3 (without destroying them!) and writes the address (1 or 3) of the smaller content in register 2. (After this program has executed, the contents of register 1 and register 3 should be unchanged, and register 2 should say which of those two registers has the smaller content.)

Division, similarly, can be done by subtracting the divisor over and over from the dividend and counting up the number of times we can do that. We can leave the remainder, if any, in a special remainder register. But here we must be careful to add one crucial safety measure: we mustn't divide by zero (must we?), so before any division starts, we should just run a simple check on the divisor, by trying to decrement it. If we can decrement it, we should just increment it once (restoring it to its proper value) and then proceed with the division. If we hit zero, however, when we try to decrement it, we need to raise an alarm. We can do this by reserving a register for an ERROR flag: a 1 in register 5 can mean "TILT! I've just been asked to

divide by zero!"

Here is the flow graph for dividing the contents of register 1 by the contents of register 2, putting the answer in register 3, the remainder in register 4, and highlighting register 5 for an "error message" (1 means "I was asked to divide by zero").



Walk through the flow graph, and notice how zero in the divisor aborts the operation and raises a flag. Notice, too, that register 4 is doing double duty, serving as a copy of the divisor, for restoring the divisor for each successive subtraction, but also serving as a potential remainder register. If register 1 zeros out before register 4 can dump its contents back into 2 for another subtraction, those contents are the remainder, right where it belongs.

2. Today arithmetic, tomorrow the world

As you can now see, *Deb*, Decrement-or-Branch, is the key to the power of the Register Machine. It is the only instruction that allows the computer to "notice" anything in the world and use what it notices to guide its next step. And in fact, this *conditional branching* is the key to the power of all stored program computers, a fact that was recognized by Ada Lovelace back in the 19th century when she wrote her historic discussion of Babbage's Analytical Engine, the prototype of all computers.

Assembling these programs out of their parts can become a rather mindless exercise once you get the hang of it. In fact, once you have composed each of the arithmetic routines, you can use them again and again. Suppose we numbered them, so that ADD was operation 0 and SUBTRACT was operation 1, MULTIPLY was operation 2 and so forth. COPY could be operation 5, MOVE could be operation 6, ... Then we could use a register to store an instruction, by number.

Exercise 4 (optional)

Draw a flow graph, and write a RAP program that turns a Register machine into a simple pocket calculator, as follows:
a. use register 1 for the operation:
0 = add
1 = subtract
2 = multiply
3 = divide
b. put the signs and the values in registers 2,3, 4, and 5
(thus 2 0 6 0 7 instructs the Register Machine to multiply plus-6 by plus-7)
putting the sign and value of the answer in registers 6 and 7, and
using register 8 for remainders (in division)
and register 9 for a divide-by-zero flag (1 = I've been asked to divide by zero)

Notice that in this example, we are using the contents of registers (in each case, a number) to stand for four very different things: a number, an error flag, the sign of a number, and an arithmetical operation. *What a number in a register stands for depends on the program that we have composed.* This is one of the fundamental facts about computers.

Using our building blocks we have already created, we can construct more impressive operations. With enough patience you could draw the flow graph and write the program for SQUARING the number in register 7, or a program to FIND THE AVERAGE of the contents in registers 1 through 20, or FACTOR the contents of register 6, putting a 1 in register 5 if 5 is a

factor, or COMPARE the contents of register 3 and register 4 and put the larger content in register 5 UNLESS it is exactly twice as large, in which case put a flag in register 7. And so forth.

A particularly useful routine would SEARCH through a hundred registers to see if any of them had a particular content, putting the number of that register's *address* in register 101. (How would it work? Put the TARGET number in register 102, and a copy of the target in register 103; zero out register 101, then, starting at register 1, subtract its contents from the contents of 103 (after incrementing register 101), looking for a zero answer. If you don't get it, go on to register 2, and so forth. IF any register has the target number, halt; the address of that register will be in register 101.) Thanks to the basic "sensory" power embodied in *DEB*–its capacity to "notice" a zero when it tries to decrement a register–we can turn the Register machine's "eyes" in on itself, so it can examine its own registers, moving contents around and switching operations depending on what it finds where. And since a number in a register can stand for anything, this means that the Register machine can, in principle, be designed to "notice" anything, to "discriminate" any pattern or feature that can be associated with a number–or a number of numbers. For instance, a black and white picture–*any* black and white picture, including a picture of this page, of course–can be represented by a large bank of registers, one for each pixel, with 0 for a white spot and 1 for a black spot. Now, write the Register Machine program that can search through thousands of pictures looking for a picture of a straight black vertical line on a white background, or a black and white checkerboard, or . . . . a (capital) letter "A"!

Since we can use a number in a register to stand for an instruction, such as ADD or SUBTRACT or MOVE or SEARCH, and to stand for addresses, we can store a whole sequence of instructions in a series of registers. If we then have a main program (program A) that instructs the machine to go from register to register doing whatever that register instructs it to do, then we can store a second program B in those registers. When we start the machine running program A, the first thing it does is to consult the registers that tell it to run program B, which it thereupon does. This means that we could store program A in the Register machine's Central Processing Unit once and for all ("hard-wired" or "firmware" burnt into the ROM), and then get it to run programs B, C, D, ... depending on what numbers we put in the registers. And that is the secret of the stored program computer, or Universal Turing Machine.

We can devise a general program-reading program, program A, which will faithfully execute whatever instructions we put (by number) into its registers. Every possible program consists of a series of numbers, in order, that Program A will consult, in order, doing whatever each number specifies. And if we devise a system for putting these instructions in unambiguous form (for instance, each instruction is exactly 32 bits long), we can treat the whole series that composes the B program as one great big long number. There is program 869284290850285490285490 and program 2845702975902875290754892749027542485092842854-0423 and so forth (but typically much longer, of course, megabytes long).

Alan Turing was the brilliant theoretician and philosopher who worked this scheme out,

using another simple imaginary computer (which chugs back and forth along a paper tape, making its behavior depend (*aha*--conditional branching) on whether it read a 0 or a 1 and the tape square under its reading head. All the Turing machine can do is flip the bit (erasing 0, writing 1 or vice versa) or leave the bit alone, and then move left or right one tape square *and go to its next instruction*. I think you will agree that writing Turing machine programs to ADD and SUBTRACT and so forth, using just the binary symbols 0 and 1, and moving just one square at a time, is a more daunting exercise than our Register machine exercises, but the point Turing made is exactly the same. A Universal Turing machine is a device with a Program A (hardwired, if you like) that permits it to "read" its program B off its paper tape and then execute that program using whatever else is on the tape as data or input to program B. Hao Wang's Register Machine can execute any program that can be reduced to arithmetic and conditional branching, and so can Turing's Turing Machine. And among the programs these machines can run are programs like program A that has the wonderful power to take the *number* of any other program and execute *it*. Instead of building thousands of different computing machines, each hard-wired to execute a particular complicated task, we build a single, general purpose Universal machine (with program A installed) and then we can get it to do our bidding by feeding it programs–software that create *virtual machines*. The Universal Turing Machine is a universal mimic, in other words. So is our less well known Universal Register Machine.

So is your laptop. There is nothing your laptop can do that the Universal Register Machine can't do, and vice versa. But don't hold your breath. Nobody said that all machines were equal in speed. We've already seen that our Register Machine is achingly slow at something as laborious as division, which it does by subtraction, for heaven's sake! Are there no ways to speed things up? Indeed there are. In fact, the history of computers since Turing is precisely the history of ever faster ways of doing what the Register machine does–and nothing else.

For instance, John von Neuman created the architecture for the first serious working computer, and in order to speed it up, he widened the window of Turing's machine from one-bit-at-a-time to many-bits-at-a-time (so, many early computers read 8 bit "words" or 16-bit "words" or even 12 bit words. Today 32-bit words are widely used. Simplifying somewhat, we can say that each word is COPIED from memory one at a time, into a special register (the Instruction Register) where it is READ and executed. A word typically has two parts, the Operation Code (e.g. ADD, MULTIPLY, MOVE, COMPARE, JUMP-if-ZERO) and an Address, which tells the computer which register to go to for the contents to be operated on. So, **10101110** 11101010101 might tell the computer to perform operation **10101110** on the contents of register 1110101010101, putting the answer, always, in a special register called the Accumulator. The big difference between the Register Machine and a Von Neumann Machine is that a Register Machine can operate on any register (Inc and Deb only, of course), while a Von Neumann Machine does all the arithmetic work in the Accumulator, and simply COPIES and MOVES (or STORES) contents to the registers that make up the memory. It pays for all this extra moving and copying by being able to perform many different fundamental operations, each hard-wired.

How many primitive operations in real computers these days? It can be hundreds, or

thousands, or in a return to the good old days, you can have a RISC (Reduced Instruction Set Computer), which gets by with a few dozen primitives, but makes up for it in the blinding speed with which they are executed. (If you could do *Inc* and *Deb* a million times faster than you could do a hard-wired ADD operation, it would pay to compose ADD out of *Inc* and *Deb*, as we did above, and for all additions with less than a million steps, you'd come out ahead.)

How many registers in real computers these days? Millions (but they're each finite, of course, so that really large numbers have to be spread out over large numbers of registers). A byte is 8 bits. If you have 64 megabytes of RAM on your computer, you have 16 million 32-bit registers, or the equivalent. We saw that numbers in registers can stand for things other than positive integers. Real numbers are stored (to whatever approximation is called for) using a system of "floating point" representations, where one number stands for the abscissa and the other for its exponent. Floating point operations are just arithmetical operations (particularly multiplications and divisions) using these floating point numbers as values, and the fastest computer you can buy today can perform over 4 megaflops: over 4 million FLoating point Operations Per Second.

If that isn't fast enough for you, it helps, of course, to yoke together many such machines in parallel. *There is nothing that such a parallel machine can do that a purely serial machine cannot do, slower!* In fact, most of the parallel machines that have been actively studied in the last twenty years have been virtual machines simulated on standard Von Neumann machines. Special purpose parallel hardware has been developed, and computer designers are busily exploring the costs and benefits of widening the von Neumann bottleneck, and speeding up the traffic through it, in all sorts of ways, with co-processors, cache memories, and various other speed-ups.

No parallel computer yet built is remotely as wide, as multi-channel, as your brain. You have somewhere in the neighborhood of 10 billion neurons, and the optic nerve is, all by itself, several million channels wide. But neurons operate much, much slower than computer circuits. A neuron can switch state and send a pulse (plausibly, its version of *Inc* or *Deb*) in a few milliseconds–thousandths, not millions or billionths, of a second. Computers move bits around at near the speed of light (which is why making computers smaller is a key move in making them faster; it takes roughly a billionth of a second for light to travel a foot, so if you want to have two processes communicate faster than that, they have to be closer together than that.)