Register Machines are Turing Machines are Register Machines

While the Turing Machine captures the fundamental essence of computation, it does not "look like" the register based machines which sit upon our desks. Here we demonstrate that the operation of a Turing machine is equivalent to a register-based machine like those CISC and RISC machines we discussed in the first session. So I say to you

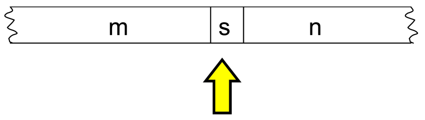
"If it runs on a Turing Machine, then it will run on a Pentium. But also, if it runs on a Pentium, it will run on a Turing Machine"

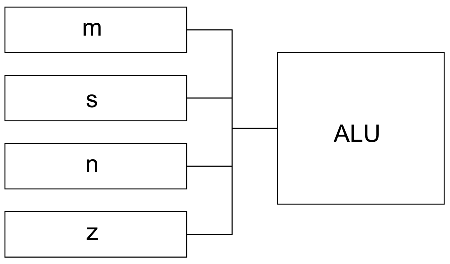
During our demonstration, we will *discover* the smallest number of programming language "constructs" required to produce ***any*** computer programme!

This session was both backwards looking and forwards looking. Starting with the *central* notion of the Turing Machine, we looked *back* to the register-based CISC and RISC machines we discussed in the first session. We also look forward to the next session where we will explore some unusual languages (and I don't mean C, C++ or Java).

So let's start the story. This is illustrated in the diagram below, where we wish to replace the Turing Machine (it's tape and read-write head, and its "programme" expressed as a table which moves from state S1 to S2) with a bunch or registers and an ALU. The registers hold numbers and the ALU does arithmetic operations such as add, multiply divide. Here's the situation:

(a)





(b)

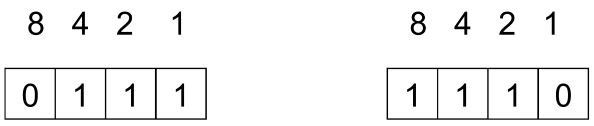
In (a) we have the Turing Machine and in (b) we have the register machine. We work like this. Let's divide the Turing tape into three sections, the left part *m*, the right part *n* and a middle part *s*. Think, to keep it fairly simple, as the symbols on the tape as either 0's or 1's. In that case, we can think of *m ,n,* and *s* as binary numbers, and binary numbers can be integers (1, 2, 5, 10, 456). So *m* represents the left part of the tape as a number which we can put into register **m**, *n* represents the right part of the tape as a number which we can put into register **n**, and *s* is the current symbol which we are reading and writing which we can put into register **s**. Register **z** is just an additional 'help' register.

That's fine, so what does the ALU do? Good question. Remember that the Turing Machine started in some state, read a symbol, wrote a symbol, shifted left or right and entered a new state. So that's what we must do. The first thing is the shifting left or right. This must be "simulated" by the register machine (whose ALU can do things like add, subtract, multiply and divide).

There are also some more *fundamental* questions which we must ask. How many **registers** do we need to simulate a Turing Machine, and how many ALU **instructions** do we need? We shall see.

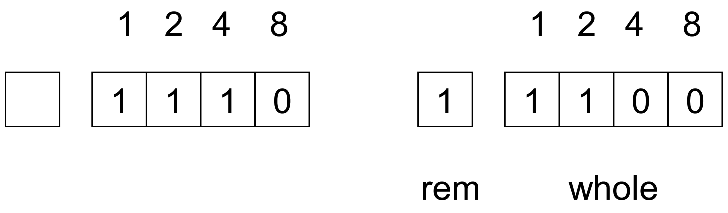
1. The Register-Machine equivalent of Moving the Read/Write head to the Right.

What follows here is not a cast-iron *proof.* That's beyond our grasp. It's a demonstration which proceeds by examples, but which I believe stands up quite well. Its limitation is that the symbols on the tape are restricted to 0's and 1's, since this simplifies the discussion a lot. The full *proof* is given in section 4 (careful going there!). Before we begin, let's just revisit some facts concerning binary numbers and shifting them. Here is a classic example of a binary number being shifted to the left. Let's have a think:

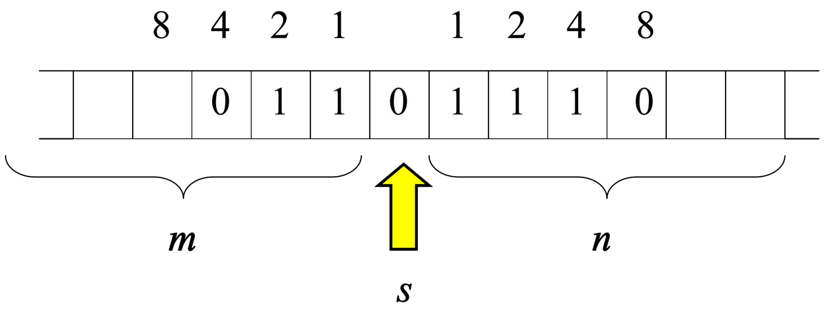


On the left, we have our starting number in binary with the value 0*8 + 1*4 + 1*2 + 1*1 = 7. A shift to the left gives us 1*8 + 1*4 + 1*2 + 0*1 = 14 (right-most diagram). So a shift to the left is equivalent to multiplying the number by 2. This arrangement of bits is the "normal" or "classical" arrangement, where the value of the bits gets greater (by a factor of 2) as we move to the left. But in our current thinking, we need to consider the "mirror" arrangement of bits, where the value of bits gets greater (by a factor of 2) as we move to the right. Trust me!

Let's look at this "mirror" situation, below. On the left we have a binary number with decimal value 7. Then we shift it to the left. The resulting whole part of the number is 3 and the remainder is one. Shifting left is dividing by 2, so 7/2 = 3 rem1.



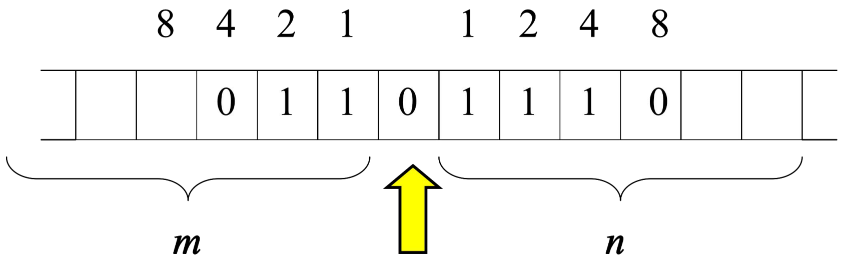
OK, now let's take a couple of examples of a real Turing Machine Tape, and move the read/write head one place to the right. We can describe the situation of the tape using three numbers, *m,s,n*. as shown below. The number *m* is the number on the tape to the left of the cell we are looking at. The number *n* is the number on the tape to the right of the cell we are looking at. The number *s* is the value in the cell the read/write head is looking at (0 or 1). Here's an example of a Turing Machine at a particular moment.



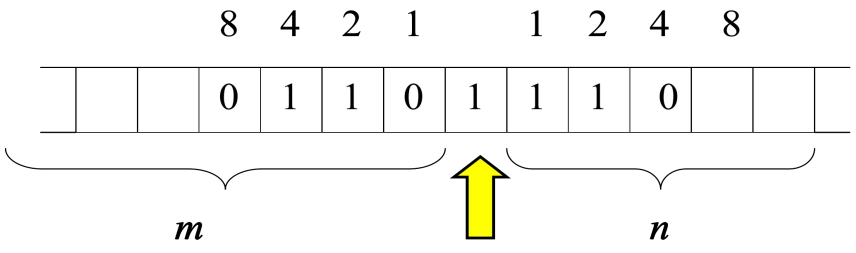
Remember, the Turing machine, at any time, is in a particular state, and it reads a symbol *s* from the tape and writes a new symbol to the tape, and moves left or right on the tape, and enters a new state. In the above diagram we *capture* the Turing Machine in her current state. We read the symbol *s* (0 or 1); here it is 0. Also we *capture* the data to the left of the "arrow" as the number *m*, and the data to the right, as the number "n". So what happens when the "arrow" moves to the right? There are two possibilities, depending on whether *s* = 0 or *s* = 1. Let's explore these.

The case for *s* = 0.

Here's the starting state for the first example. Note the symmetrical arrangement of the value of the bits.



Here we have *m = 3, s = 0* (pointed to by the arrow) and *n = 7*. That's fine, so let's see what happens when the head moves to the right,



So the new values are *m' = 6, s' = 1, n' 3*. So what's happened to the numbers? Well it appears that the following has happened:

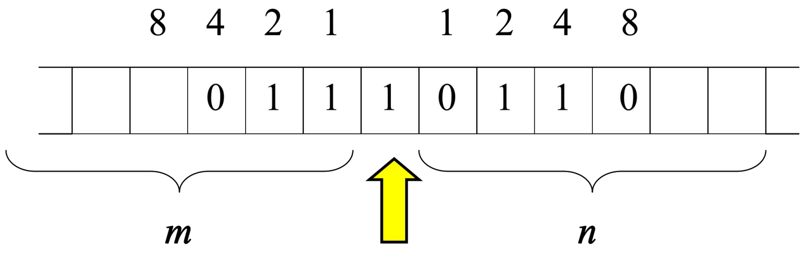
m' = 2 x m 6 = 2 x 3

n' = n/2 (whole number division) 3 = 7/2 (remainder 1)

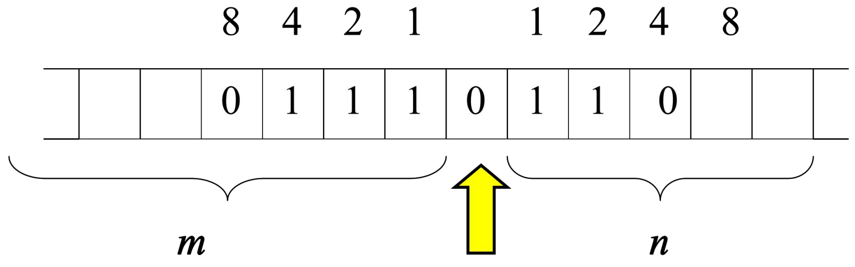
s' = remainder n/2 remainder 7/2 = 1

So the results of the tape read/write head moving to the write seem to be expressed in these three arithmetical calculations. Cool. They are close to being correct, but they are not exactly correct. We can why they are not correct this in the following example.

The case for *s* = 1.



Here we have *m = 3, s = 1* (pointed to by the arrow), and *n=6*. Note the difference with the previous example. Importantly here we have *s = 1* which will help us clear up our incorrectness. OK let's shift to the right. Here's what we get.



So now we have in this example, *m'=7, s' = 0, n'=3* .This agrees with our previous example (almost). Let's see what agrees. First we have

n' = n/2 (whole number) 3 = 6/2 (remainder 0)

s' = remainder n/2 remainder 6/2 = 0

The problem was with the calculation of *m'*. In the first example, we said that *m' = 2m* but here that does not work because 7 does not equal 2 x 6. The reason is that the value of *s* has been shifted into *m*. Here, *s=1* so it adds a 1 to the value of *m'*. In the first example we had *s*=0 so the 0 was added in and we did not notice it. But now we must, so in general, we write

m' = 2 x m + s 7 = 2 x 3 + 1

That's put it to bed. We have discovered that the movement of the read/write head to the right on a Turing Machine tape is equivalent to the following three basic arithmetic calculations.

$$s^* = rem(n/2)$$

$$m^* = s + 2m$$

$$n^* = n/2$$

These calculations, which only use the operations of addition, multiplication, division and getting the remainder, do exactly the same as a Turing Machine moving to the right on a tape. So we have discoverer that a Turing Machine moving to the right on its tape can be emulated by some simple arithmetic operations executed by a register-based machine. This is the first part of the demonstration that register machines and Turing Machines are the same.

Before we take the demonstration further, it is interesting to find the minimal number of instructions required to support these three calculations.

2. A Set of Minimal Instructions

As we have seen, in our first session, complex instructions may be better realized as simple instructions. The complex instructions provided by a CISC processor are rarely used; it is the RISC instructions which are ultimately produced by a compiler and are executed by our machine. So we ask what are these *minimal* instructions, and *how many do we need?* The answer is just two. Let's tell this story.

We have identified that we need addition, multiplication, division and remaindering, but these are not the simplest operations we have to use. It's possible to replace all of these operations based upon the following instruction set:

`mov reg,0`	Put 0 into register "reg"
`inc reg`	Add 1 to the contents of register "reg"
`decjmpreg reg,lab`	If register "reg" is zero, jump to the instruction labelled "lab" else subtract 1 from "reg" and do the next instruction
`jmp lab`	Jump to the instruction labelled "lab"
`hlt`	Halt

We shall also show in a moment that the first, fourth and fifth instructions can be replaced by combinations of the second and third. While most instructions are easy to understand, the third appears a little daunting, so let's see an example or two of this instruction in operation. This will help us understand how to perform the three basic arithmetic operations required for the shift-right operation. Let's start with this one.

m* = s + 2m

So here we go. The three columns show how the registers change as we jump around the instructions.

Columns are in sequence; we do column 1 then 2 then 3. We need three columns because there are jumps and therefore loops.

		eax	ebx	ecx	edx	eax	ebx	ecx	edx	eax	ebx	ecx	edx
	`mov ecx,3`			3									
	`mov ebx,0`		0										
L3:	`decjmpreg ecx,L4`			2				1				0	
	`inc ebx`		1				3				5		
	`inc ebx`		2				4				6		
	`jmp L3`												
L4:	`hlt`												

The three columns show how the registers change as we jump around the instructions. So we start by moving 3 into ecx and 0 into ebx. Then we hit the `decjmpreg` instruction. This checks ecx to see if it is zero (if so, we would jump to L4), but it is not zero, so we decrement it by one and continue. Then we have two inc ebx instructions so we increment ebx by two. Then we jump to L3. Here again the `decjmpreg` instruction looks at ecx, finds it is not zero so decrements it by one giving the result 1 and then executes the following instruction. Register ebx is incremented twice so that ebx becomes 4. We jump back up to L3 and check if ecx is zero, it's not so we decrement it to zero. Two more inc's sets ebx to 6. Jump again to L3 where the `decjmpreg` instruction has ecx as 0, so we finally jump to label L4 and halt. Whew! It's interesting to see what's happened. Register ecx was initially 3, and the program has ended with ebx as 6, in other words two times ebx. In other words the program multiplies the contents of ecx by two and sticks the result into ebx. It's not hard to see why. In the loop, every time we decrement ecx by one, we increment ebx by two. So that's how we do the multiplication by 2 in the second basic arithmetic calculation $m^* = s + 2m$. I guess you can work out how to multiply by 5!

I said we can simplify the "mov reg, number" instruction and the "jmp lab" instruction. First let's see how we can get rid of the mov reg, number instruction. Let's consider mov ecx,3. Well the following code will do this:

```
assume ecx is zero
inc ecx
inc ecx
inc ecx
```

Not much to say about that, we've replaced a "move reg, number" instruction by an equivalent number of "inc"'s. Now how do we reduce the "jmp label" instruction? Remember that `decjmpreg` reg,lab instruction jumps to "lab" when "reg" is zero, so the following code will give us an unconditional jump, jmp lab:

```
          assume ecx is zero
     L1:  …
          …
          decjmpreg ecx,L1
```

But hang on a bit, you're probably getting a little worried that in both of these I've *assumed* that a register is zero, that's a pretty hard assumption. Can we not *set* a register to zero using our two "atomic" instructions? Well, yes, and this is how it is done.

```
     …
     …
L1:  decjmpreg ecx,L2
     jmp L1
L2   …
```

eax	ebx	ecx	edx	eax	ebx	ecx	edx	eax	ebx	ecx	edx
		3									
		2				1				0	

The register ecx is assumed to contain 3 at the start, but this could be any number. It's easy to see what this program does, it loops decrementing ecx until ecx is zero then it jumps to L2 and continues with the rest of the program. So that's how to set a register to zero.

Now we need to understand how we can add the value of *s* to *2m* to complete the basic arithmetic calculation $m^* = s + 2m$. Actually, this is not a general addition, since *s* can only be 0 or 1. So, we only need to add a 1 to m^* if s=1. The following code does just this.

```
          decjmpreg ebx,L3
          inc edx
     L3:
```

Remember that edx holds the value of the intermediate variable *z* which is used to calculate $m^* = s + 2m$. and that ebx holds the value of *s*. So if $s = 1$ (then ebx holds 1) the `decjmpreg` instruction decrements ebx by one (since it is not zero) and does not jump to L3. So the following instruction inc edx increments *z* by one which is adding the value of *s* (=1) onto *2m*. On the other hand if $s = 0$ (ebx holds 0) then the d `decjmpreg` instruction jumps straight to L3 not incrementing edx, in other words adding 0 to *z*.

Now let's turn to the other two basic arithmetic operations

$$s^* = rem(n/2)$$
$$n^* = n/2$$

Look at this code:

		eax	ebx	ecx	edx	eax	ebx	ecx	edx	eax	ebx	ecx	edx
	`mov ecx,5`			5									
	`mov eax,0`	0											
L3:	`mov ebx,0`	0					0				0		
	`decjmpreg ecx,L4`			4				2				0	
	`inc ebx`		1				1				1		
	`decjmpreg ecx,L4`			3				1					
	`inc eax`	1				2							
	`jmp L3`												
L4:	`hlt`												

It starts off with 5 in ecx and ends up with 2 in eax and 1 in ebx. So what? Well, this code is actually computing two parts of the Turing Machine we needed, as seen above. We have 5/2 = 2 (whole numbers) which computes $n^* = n/2$ and we have remainder(5/2) = 1 which computes $s^* = rem(n/2)$. Try it out for other values in ecx (e.g. 4) and check that it works.

So at this point we have demonstrated that we can build a register based Turing Machine using four registers and two instructions. That's all the hardware and software atoms we need to construct any computer that can execute algorithms (well-ordered randomless programs). Here's our instruction set:

`inc reg`	Add 1 to the contents of register "reg"
`decjmpreg reg,lab`	If register "reg" is zero, jump to the label "lab" else subtract 1 from "reg" and do the next instruction

The *complete* code for the Turing machine when the read/write head moves once space to the right is shown below. There's annotations to make things quite clear. Remember that all jmp's and mov's can be replaced with `decjmpreg`'s and inc's. This machine is running on 4 regs and 2 instructions.

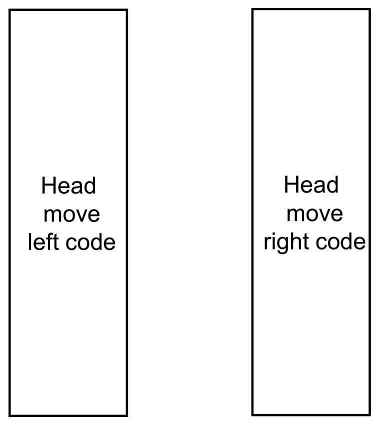
```
L2:     decjmpreg eax,L1                         eax holds m
        inc edx                                  ebx holds s
        inc edx          z = 2m                  ecx holds n
        jmp L2                                   edx holds z

L1:     decjmpreg ebx,L3
        inc edx               If s is 1 z = z + 1

L3:     decjmpreg edx,L4
        inc eax
        jmp L3               m* = z

L4:     mov ebx,0
        decjmpreg ecx,L6
        inc ebx
        decjmpreg ecx,L6     s* = rem(n/2)
        inc edx              z = n/2  whole
        jmp L4

L6:     decjmpreg edx,L7
        inc ecx
        jmp L6              n* = z

L7:     hlt
```

Finally, let's write the code for our General Purpose Register (GPR) Turing Machine based on four registers and our two atomic operations, translating the mov's and the jmp's into our two primitive operations. This is shown in Section 5.

3. The Register Machine Equivalent of the Entire Turing Machine

The above register machine provided the same functionality of the "move to right" movement of the read/write head. But remember that a Turing machine operation consists of the following steps. The machine starts in State *S*. It reads the symbol at the current head position. It writes a symbol at the current head position. The head moves left or right. The machine goes into state *S**, depending on its previous state and the current symbol read *s*. So we have to code these additional elements. All of these elements are written as code. Here's an overview of how the code is structured. It consists of repeated blocks, one block for each state. Remember that eax contains the entire tape to the left and ecx contains the entire tape to the right.

We presented the head-movement code for move-left above. There is similar code for move-right. When we are in state *S* and prepare to move to state *S**, then we will either select the move-left or the move-right code. So each state will contain two blocks of "movement" code like this.



So let's say we are in state *S0* and we read the symbol *s2*. To know what to do next for the case of the tape Turing machine, we looked up *S0* (as primary key) and *s2* as secondary key in the Turing Machine database table;

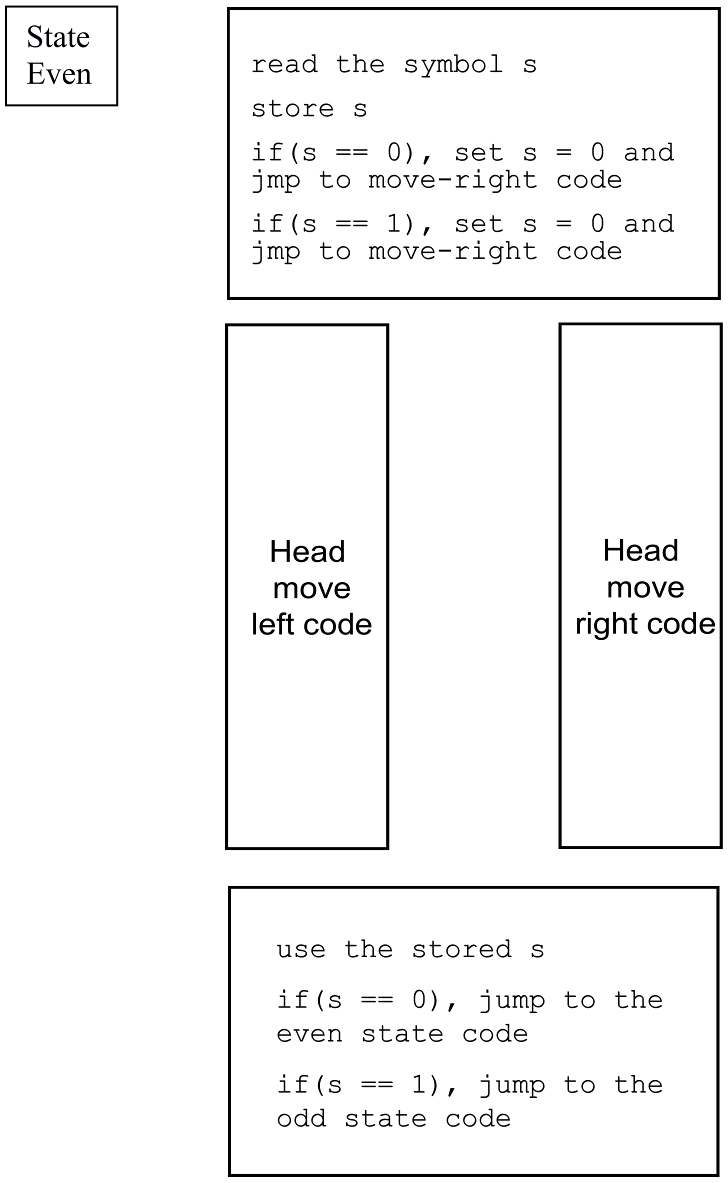
Current State *S*	Symbol read *s*	Symbol to write *s**	Direction to move *d*	New state *S**
S0	*s1*	*s*1*	*R*	*S1*
S0	*s2*	*s*2*	*L*	*S1*
S0	*s3*	*s*3*	*R*	*S2*

So when we are in state *S0* and read symbol *s2* then we are looking at the second row in the table which tells us to write *s*2* and move to the left and into the state *S1*. Let's run this example again, but taking a portion of the Turing Machine table for the parity detector we saw previously. Here, the first three lines are as follows:

Current State *S*	Symbol read *s*	Symbol to write *s**	Direction to move *d*	New state *S**
Even	*0*	*0*	*R*	*Even*
Even	*1*	*0*	*R*	*Odd*
Even	*@*	*0*	*N*	*Halt*

So if we are in the *Even* state and read a *1* then we write a 0, move to the right and go into the *Odd* state.

So how do we do this with code? We sandwich the head movement code with two other blocks of code like this, for the parity detector example.



```
read the symbol s
store s
if(s == 0), set s = 0 and
jmp to move-right code
if(s == 1), set s = 0 and
jmp to move-right code
```

```
use the stored s
if(s == 0), jump to the
even state code
if(s == 1), jump to the
odd state code
```

So we are in state *Even* we read the symbol *s* which is a 1, we save it in a register and then we jump to the head move right code. Then we used the stored read symbol to jump to the new state code which is odd.

You will see that the Turing Machine table rows (one for each state – read symbol pair) are coded through if (x = a) type statements. We have seen that it is possible using our two atomic operations. So that's how we do that. To effect these enhancements, we have had to add an additional register to our principal set of 4 registers to store *s*. Also, as explained in Section 5, we need an additional 'help' register to be able to zero our 4 principal registers. However, in section 6, we will show how to reduce these 6 registers to 2, to create a register-based machine with just two instructions and two registers.

4. Generalization of the Head Moving "Proof".

5. Code for the Right head movement using only two instructions.

Here is the full code listing for the right head movement using only two instructions. We have added an additional register "esi" in order to be able to set our four principal registers to zero.

```
; TuringMachineRightFull.asm
;
; Computation of Right Branch of Turing machine using only two assembler instructions
;
; Register mapping            m = eax,    s = ebx,    n = ecx,    z = edx
; Register containing a zero (for jumps) esi.
;
; CBPrice   February 19 2010

includelib kernel32.lib
includelib user32.lib

.586
.MODEL flat,stdcall
.STACK 4096

.code ;===================== CODE SEGMENT ======================

decjmpreg   macro reg, target
                  cmp reg,0
                  jz target
                  dec reg
                  endm

main PROC

; ===============================================================

; clear registers to zero ---------
; we must assume that one register is set to zero. Use esi (This increases our
machine to 5 registers)
      sub esi,esi ; sets register esi to zero. (Subtracts what ever is in this
register from itself to give a solid zero).

;     zero eax --------------------
;     sub eax,eax
A1:   decjmpreg eax, A2
      decjmpreg esi, A1 ; remember this is an unconditional jmp
A2:

;     zero ebx --------------------
;     sub ebx,ebx
A3:   decjmpreg ebx, A4
      decjmpreg esi, A3
A4:


;     zero ecx --------------------
;     sub ecx,ecx
A5:   decjmpreg ecx, A6
      decjmpreg esi, A5
A6:

;     zero edx --------------------
;     sub edx,edx
```

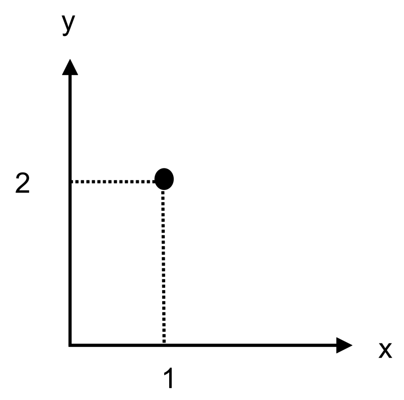
```
A7:   decjmpreg edx, A8
      decjmpreg esi, A7
A8:

; ==============================================================

;     Now, set m, s and n to some initial values. These will be set by the Turing
Machine starting state; here we do it explicitly.
;
;     The values are, m=11, n=8 and s=1


;     Set eax to the value of m (11) ----------------------
;     mov eax,0bh (decimal 11)
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
      inc eax
;
;     Set ecx to the value of n (8) -----------------------
;     mov ecx,8
      inc ecx
      inc ecx
      inc ecx
      inc ecx
      inc ecx
      inc ecx
      inc ecx
      inc ecx
;
;     Set ebx to the value of s (1) -----------------------
;     mov ebx,1
      inc ebx
;

; Here is the code to do a shift to the right

L2:   decjmpreg eax,L1
      inc edx
      inc edx
;     jmp L2
      decjmpreg esi, L2
;
L1:   decjmpreg ebx,L3
      inc edx
;
L3:   decjmpreg edx,L4
      inc eax
;     jmp L3
      decjmpreg esi, L3
;
L4:   mov ebx,0
      decjmpreg ecx,L6
      inc ebx
      decjmpreg ecx,L6
      inc edx
;     jmp L4
      decjmpreg esi,L4
;
L6:   decjmpreg edx,L7
      inc ecx
;     jmp L6
      decjmpreg esi, L6
;
L7:   hlt

main ENDP

END main
```

6. Reducing the Number of Registers needed to two.

In our development above, we have seen that we need two instructions and six registers to emulate a Turing machine. I was concerned about the lack of ‘symmetry’ in this result and have worked out how to reduce the number of registers to two. So I have discovered that we only need two registers as well as two instructions to emulate the Turing machine and therefore all computers. Here’s what I did.

It’s all to do with ‘prime numbers’ (1,2,3,5,7,11,…). At the same time, this gave me an answer to a problem which has been troubling me over the last few days, the concept of “Godelization of location in space”. So now I shall attempt an explanation of how to get down to two registers, based on a Godelization of 2D space, though I shall show you how this can be extended to *n-dimensional* space. It’s quite easy to understand.

Let’s consider the following situation. We have a point on the 2D plane with co-ordinates $(x,y) = (1,2)$. This could be the location of an asteroid in a 2D game. How can we create a single unique number to represent this position, ie a number which is a Godelization of this location in 2D space?



First we have to understand something fundamental. The x-value of 1 and the y-value of 2 are **independent** which means they are not related. To get to the location (1,2) we have to take one step in the x-direction and then two steps in the y-direction. No movement in the x-direction will change the y location and *vice-versa*.

Let's proceed by example. Let's refer to the Godelization of this location as *G(1,2)*, remembering we wish to associate a unique number (or code) for this location. We must be able to retrieve the location (1,2) by decoding the associated Godel number in the future. The x-axis is associated with the prime number 2 and the y-axis is associated with the prime number 3.

First let's look at the *Godelization* of this location. We take the prime numbers 2 and 3 as mentioned above as a basis. We define the Godelization of 2D space *GS(x,y),* as follows

$$GS(x,y) = 2^x.3^y$$

In English this means 'take 2 to the power of *x* and multiply it by 3 to the power of *y*'. Ok. Let's deal with the 'power' concept first. For example '2 to the power of 4' (x = 4) means 2x2x2x2, that is 2 multiplied by itself four times. Easy.

So let's work the above example, for the point at location (1,2). Here, $x = 1$ and $y = 2$. So we have

$$G(1,2) = 2^1.3^2$$

which means 2 x (3 x 3) = 18. This is a unique Godel number which identifies this location in 2D space.

To show that this Godel number is unique for this location, we must be able to decode (retrieve) the location (x=1, y=2) from this Godel number. Here's how we do it. There are two stages, one for *x* and one for *y*.

First, lets decode *x*. To do this, we repeatedly divide the Godel number by the associated prime number, 2, as long as we do not get a remainder. Here we go

(i) 18 / 2 = 9
(ii) 9 / 2 = 'can't do' (we got a remainder) so stop.
(iii) Count the successful divisions. We have done one successful division. So $x = 1$.

Now let's decode *y*. To do this we repeatedly divide the Godel number by the associated prime number, 3. Here we go
(i) 18 / 3 = 6
(ii) 6 /3 = 2
(iii) 2 / 3 = 'cant do' (we got a remainder) so stop.
(iv) Count the successful divisions. We have done two successful divisions. So $y = 2$.

We have decoded the values (x,y) = (1, 2) from the Godel number 18. This is happening because GS(1,2) = 2x3x3, and this number is divisible by 2 once and by 3 twice.

It's easy to see how we could extend this into Godelization of 3D space. We need to use the next prime number (5) for the *z*-dimension and calculate the function

$$GS(x,y,z) = 2^x.3^y.5^z$$

What's going on here? Well we are building a Godel number which is combining several elements of data in a way that we can extract each element from the Godel number. The use of 'prime' numbers is crucial here, so in the process of combining the elements, the elements remain separate, they do not 'interfere' with each other, and so we can extract the elements from the single Godel number.

So, for example $GS(3,4,5) = 2^3.3^4.5^5$ = (2x2x2)x(3x3x3x3)x(5x5x5x5x5) which is divisible by 2 three times, and is divisible by 3 four times and is divisible by 5 five times. So we can retrieve our starting umbers 3,4,5! To extend this procedure to Godelize *n*-space we form the product

$GS(x_1, x_2, ..., x_n) = 2^{x_1}.3^{x_2}....N^{x_n}$ where N is the n'th prime number.

How do we use this to reduce the number of registers? Let's say we have three registers **a,b,c** holding numbers *a,b,c.* Then we Godelize these numbers (3D-space) as above to produce a single number *r* which we put into a single register **r**, like this:

$$r = GS(a,b,c) = 2^a.3^b5^c$$

We can then use this single register in our register machine, and can recover the values in **a,b,c** by dividing by 2,3,5 respectively.

But how do we use this strange register in the machine presented in sections 2,3 and 5? Well, remember that we only need two instructions **inc reg** and **decjmpreg reg,L** which work on the registers **reg**. How can these instructions work on our new single register **r**?

Lets take the **inc** instruction first. We assume that $r = GS(a,b,c) = 2^a.3^b5^c$. Let's take the case of incrementing *a* by 1 to *a* + 1. This is done using the formula

$$2^{a+1} = 2.2^a$$

for example, let's say that *a* = 4 which we want to **inc** to 5. Now $2^5 = 2.2^4$ or in basic English (hehe) 2x2x2x2x2 = 2x(2x2x2x2). So to **inc** register **a** (contained in **r**) all we need to do is to multiply the number in **r** by 2! We saw how to do this in the lecture. Similarly, if we want to **inc** register **b** then we use the equivalent formula

$$3^{b+1} = 3.3^b$$

so we multiply the number in register **b** by 3 (which we also saw in class) and so on.

Now we need to think about how to *decrement* a register, since this is needed for the **decjmpreg reg,L** instruction. (Remember this instruction jumps to label **L** if the contents of **reg** is zero, else it decrements the number in **reg.** Let's see how to decrement the value in register **a**. Here we use the formula

$$2^{a-1} = \frac{2^a}{2}$$

For example, if *a* = 4 we want to decrement to *a* = 3. So we calculate

$$2^3 = \frac{2^4}{2} = \frac{2x2x2x2}{2}$$

In other words we have to divide by 2! It looks like we need two mathematical operations to produce our Turing Machine, multiplication and division. But we have seen that multiplication is repeated addition and that division is repeated subtraction, so we only need the two mathematical operations of addition and subtraction in other words incrementing and decrementing.

So we have seen that the first requirement of the **decjmpreg reg,L** can be met, by division. But what about the "jump" part, where we jump to label **L** if the content of **reg** is zero. We have seen in Section 5 the need to have a register with initial value zero. Let' call this register **z.** Then, the instruction **decjmpreg z,L** will make an unconditional **jmp** (jump) to the label **L.** So to round-up, we conclude that we need two registers for our machine, one is **r** and the other is a "zeroed-out" register **z**. So therefore I state the following *theorem*.

Theorem 1. For every Turing Machine **T** there exists a register-based machine **R** which exhibits the same behaviour as **T**. This register-based machine consists of *two registers* and *two instructions*.

7. Solutions to Worksheet Problems

Here is the code corresponding to the worksheet problems.

```
; CBPrice   February 2010
;
; Solutions to the Session 4 exercises

includelib kernel32.lib
includelib user32.lib

.586
.MODEL flat,stdcall
.STACK 4096


decjmpz     macro target
            jcxz target
            dec ecx
            endm
decjmpreg   macro reg, target
                  cmp reg,0
                  jz target
                  dec reg
                  endm

.code
main PROC
;
; assigning a number to a register --------------------------------------

      inc eax
      inc eax
      inc eax

; setting a register to zero --------------------------------------------

      mov eax,5   ;put a number in eax to be zeroed

L1:   decjmpreg eax,L2
      jmp L1
L2:   nop
```

```
; copy the contents of eax into ebx -------------------------------------

      mov eax,5
      mov ebx,0

L3:   decjmpreg eax,L4
      inc ebx
      jmp L3
L4:   nop

; adds the numbers in eax and ebx and puts the result into ecx -----------

      mov eax,3
      mov ebx,4
      mov ecx,0

L5:   decjmpreg ebx,L6
      inc ecx
      jmp L5
L6:   decjmpreg eax,L7
      inc ecx
      jmp L6
L7:   nop

; if(x == 0) then execute code block A else if(x == 1) then execute code blockB.

      mov eax,1

      decjmpreg eax,L8

      ;code for block B here

      jmp L9

L8:   ;code for block A here

L9:   nop   ; done

; code to subtract ebx from eax

      mov eax,5
      mov ebx,3

L12:decjmpreg ebx,L10
      decjmpreg eax,L11
      jmp L12

L10:nop
L11:nop

; code to test if eax and ebx are equal (looks like subtraction! -------------

      mov eax,5
      mov ebx,5

L15:decjmpreg ebx,L13
      decjmpreg eax,L14
      jmp L15

L13:mov eax,1
```

```
L14:nop


;; code to test if x == 3. if(x == 3) then goto code block A else goto code block B


      mov eax,2

      decjmpreg eax,L16
      decjmpreg eax,L16
      decjmpreg eax,L16
      decjmpreg eax,L17

L16:nop     ; code block B

L17:nop ; code block A
```

8. Postscript

In writing this session I have become aware of something truly fundamental. A computer can be seen as (i) numbers stored in two registers, and (ii) two mathematical operations on these numbers. Why does mathematics have the power to represent all potential computers and their applications and not philosophy or physics? I do not restrict this question to electronic, mechanical, or electro-mechanical computers, but embrace "natural" computers such as Stonehenge. I've experienced this awareness once before while working during my early research. Why does mathematics capture nature? This includes all science, from the reproduction of rabbits, the formation of animal skin patterns (zebra stripes, leopard spots), the trajectories of balls, bullets and spacecraft. Simulations of all of these can be performed by computer programs. We have seen that these programs can be reduced to two instructions and two registers, where these programs express algorithms. Therefore we conclude with this theorem

Theorem 2. Since the natural world (in all of its complexity) can be simulated by a computer, and since any algorithmic computer can be reduced to two registers and two instructions, the natural world consists of two registers and two instructions and proceeds by a "natural" algorithm.