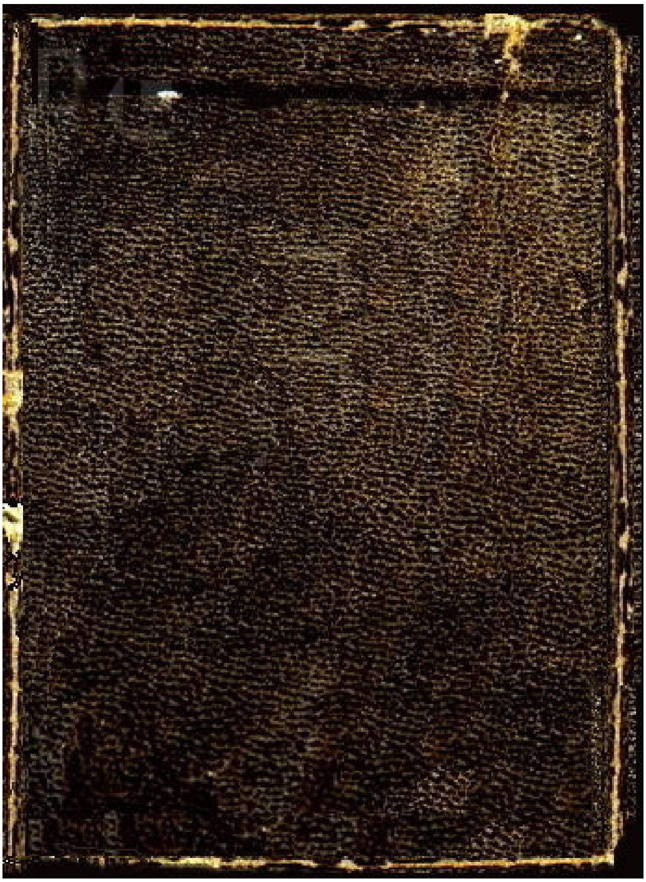
Haecceitics



JLIAT

I met a traveller from an antique land
Who said: "Two vast and trunkless legs of stone
Stand in the desert. Near them on the sand,
Half sunk, a shattered visage lies, whose frown
And wrinkled lip and sneer of cold command
Tell that its sculptor well those passions read
Which yet survive, stamped on these lifeless things,
The hand that mocked them and the heart that fed.
And on the pedestal these words appear:
`My name is Ozymandias, King of Kings:
Look on my works, ye mighty, and despair!'
Nothing beside remains. Round the decay
Of that colossal wreck, boundless and bare,
The lone and level sands stretch far away".

PART 1 LIVING WITH CYBORGS

How can we live in an age of technology which constantly appears to dehumanise? How can we live in an age where whatever we are is overwhelmed by global communication and information flows? How can we live in an age where our opinions are instantly subject to global knowledge bases, where trends and ideas flow faster than our thoughts? How can we live in an age of science so advanced we fail to understand or comprehend it other than by simplistic metaphor which resembles a religious faith? How can we live in an age where Art is anything and everything we want it to be and so nothing?

Is it a case of sliding into consumerist ignorance, or joining in the mass faith of technophiles. Or …

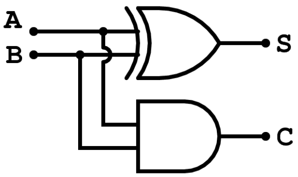
The Art of simple representation of a truth without metaphor.

An Alternative to Science
An Alternative to Philosophy
An Alternative to Metaphysics
An Alternative to Science Fiction
An Alternative to Complexity Theory
An Alternative to Art as Affect
An Alternative to Art as Sensation
An Alternative to Art as Abstraction
An Alternative to Metaphor
An Alternative to Tropes
An Alternative to Simile – Atoms are like…. Electrons are like…

Art as Representation.

Art as the revelation of individual immanence, as in the religious art of the revelation of the world to the individual as fully comprehendible and experiential by the individual.

In a techno-scientific world how is the average individual to experience directly anything other than in confused ignorance. However in the history of western art such problematics of complex theology were rendered in and by simple stories – narratives that everyone could understand. Bereft of simile and metaphor the stories are of simple immanent real events which render fully their meaning without recourse to transcendental metaphysics. This work attempts to do the self same by a full exposition of what is the current dominant technology which is increasingly being (wrongly) mythologized into some future absolute being.



This is not intended as an introduction to Computer Science – even if it is - but as a work of anti-representation and experiential ~~philosophy Science Art.~~

Anti-representation of ideas regarding emerging technologies such as Artificial Intelligence, smart algorithms and similar conceptualizations of computer technologies. From the trivial and cosmetic regard given to smart phones, new programming languages and computer applications – through to Dr. Bostrom's idea of the probability of this world being a computer simulation, Professor Tippler's idea of a future computer emulation of the dead, the ideas of Mind Uploading, Democratic transhumanism, cybernetic immortalism, the anthropomorphism and animism of digital technology (mythological and neo-spiritualisms) in general but especially in the arts, humanities and political ideologies of digital solutions to all and every problem, from learning to read through to immortality.. as well as meta explanations and philosophical mash-up problematics associated with Object Oriented Philosophies and Speculative Realism's pseudo scientific – ultra scientific - objects of transcendence.

Experiential in that knowing is not experiencing and in the above ideas and trends of super-naturalizing logic lies a mistake in the idea that it is not possible to fully and totally experience an object. This mistake arises from confusing quantity with quality. It is quality which gives value, which in the end becomes transcendental and so gains a religiosity. It is quantity which capitalizes the world. All non-experiential philosophy, all rejection of the idea of quantity and its replacement with quality becomes accidentally, despite itself or deliberately, a doxa. All experiential philosophy becomes non philosophy as it becomes an experience of a thing in it self as a totality.

This work addresses only digital computing and whilst other methods involving Quantum or Biological materials may offer complex solutions to problematics and desires, within the realm of digital processing, all processing is essentially of a very simplistic nature. There are no "sophisticated" systems – only ever larger ones in quantity NOT quality, digital processing structures which all operate and depend on the simplest of all logics and simplest of all devices- The Switch.

If there are to be terminator cyborgs and Matrix Realities then these will be truly remarkable structures in how the simple two way light systems used in ordinary domestic houses can produce such artefacts. It is not the case that a future artificial and higher intelligence will be an unknown to us, as humanity is unknown to an euglena, but that these realities if they materialize will be completely knowable to us in their fundamental mechanisms.

I wish therefore to remove or exorcise the ghost in the machine before it gets put there. Any object or machine's remoteness from us, from our understanding is only a remoteness of physical quantity.

Introduction.

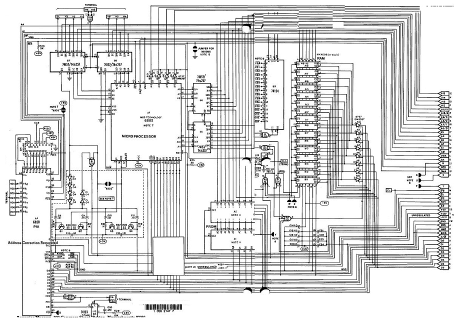
There are no “sophisticated” systems – only ever larger ones in quantity NOT quality.

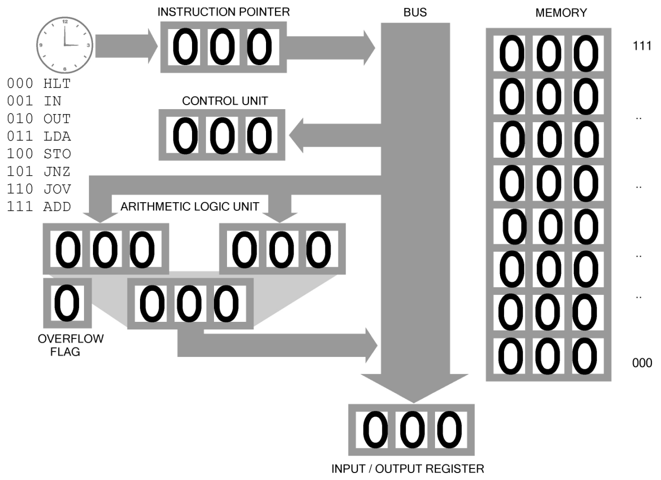
The manner in which a digital computer works is identical no matter what size or power the machine has. It is not the case that newer and more exotic devices have been added to enhance a basic design. The basic design has in the quantity of devices – not in quality - been increased, the speed of processing increased but without any fundamental change in architecture- and the size or compactness has been decreased. But the “model” CPU (Central Processing Unit) used here has no essential difference from the most powerful processing machines available today, or in future, using digital technology.

The model uses “registers” – small bits of memory to process data, buses, wires which carry data, and storage or memory. All the processing is undertaken by simple switching, of a bi stable, on/off binary logic.

“Starting with the AMD Opteron processor, the x86 architecture extended the 32-bit registers into 64-bit registers in a way similar to how the 16 to 32-bit extension was done (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, RFLAGS, RIP), and eight additional 64-bit general registers (R8-R15) were also introduced in the creation of x86-64. However, these extensions are only usable in 64-bit mode,”

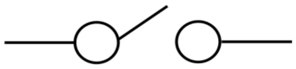
Our processor has 1 register for processing of 3 bits and a memory capacity (total) of 24 bits. Rather than a 64 bit bus it has a 3 bit bus and only 8 instructions. However all of the functionality of a modern CPU can be accomplished with such an instruction set. And we can fully know the totality of this functionality.

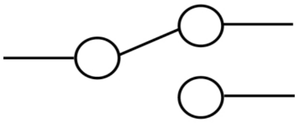



INSTRUCTION POINTER
BUS
MEMORY
000
111
..
..
..
..
000 HLT
001 IN
010 OUT
011 LDA
100 STO
101 JNZ
110 JOV
111 ADD
CONTROL UNIT
ARITHMETIC LOGIC UNIT
OVERFLOW FLAG
INPUT / OUTPUT REGISTER
0

Each of the devices in the C.P.U. (Central Processing Unit) will be described and then the functioning of the device. This will enable not only a phenomenological access but a logical/physical access to the object. Apart from the trivial "understanding" the idea of a totalizable object as a possibility will be presented, this presentation has more of the idea of an Art as non sensationalism – but as full disclosure of the world.

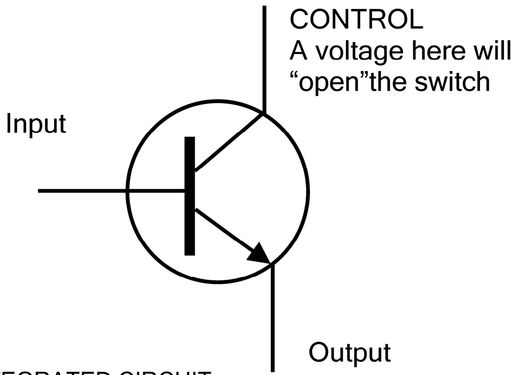
The basic component is a switch. There are two types – a simple on-off switch (one way) and what is known as a two way switch.


ONE WAY

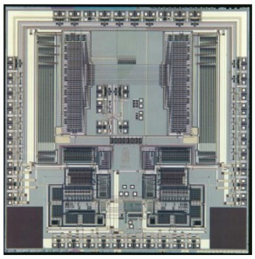

TWO WAY

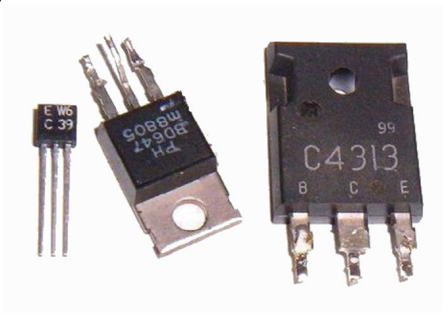
We can construct all the devices of a CPU with these switches. The switches can be made of anything, if we are using electricity then they need to be electrical switches, but we could equally use water in which case the switches would be "taps" - spigot or faucet in the U.S. In contemporary devices transistors are used. A transistor is an electronic device which can either amplify a source, or it can act as a switch. It is called a solid state device, it has no mechanical moving parts, earlier computers used values (tubes) which perform the same switching functions, or they used relays and even mechanical cogs. The use of transistors and electricity is simply to reduce the size of the device and increase the speed. It is not necessary to know how these switches work, the functioning of the CPU will be identical no matter from what material they are made, or if the CPU exists merely as a diagram or model.

TRANSISTORS ARE DRAWN DIAGRAMMATICALLY LIKE THIS

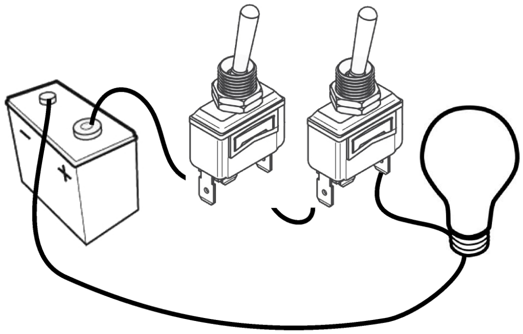


THE INTEGRATED CIRCUIT
OR CHIP IS SIMPLY MANY
TRANSISTORS MADE ON
A PIECE OF SILICON.

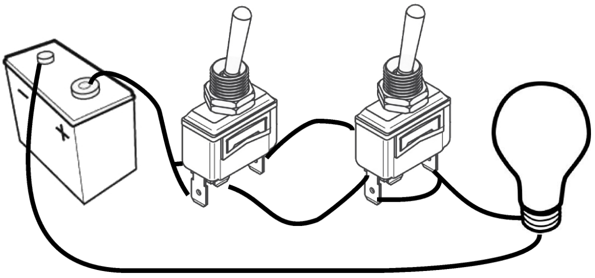




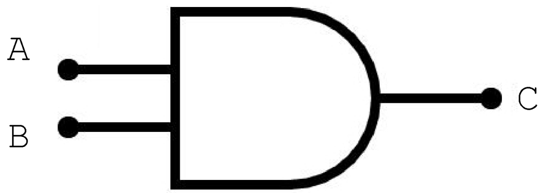
Given these switches it is possible to construct simple devices for performing logic and arithmetic. What we mean by logic here is very simple, they are the AND logic "gate" and the OR logic gate. Logic gates will pass a signal in certain defined circumstances, this is treated as a "TRUE" output. For instance with an AND gate with two inputs both inputs must be true for the output to be true. For an OR gate if either input is true then the output will be true. The OR gate is used in two way switches on staircases in houses. So if a switch is ON or closed then the output is true, or 1. If the switch is OFF then the output is false or 0. There are numerous simple gates, we will just use the two mentioned here.



AND GATE - BOTH SWITCHES NEED TO BE ON

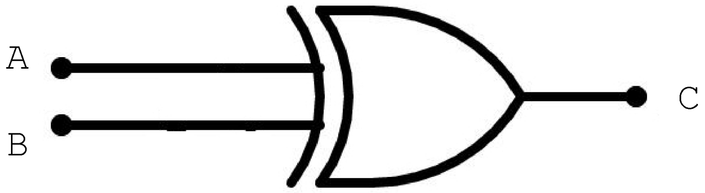


TWO WAY SWITCH - EITHER SWITCH ON - TURNS ON LIGHT
BOTH SWITCHES ON TURNS LIGHT OFF



THE SYMBOL FOR THE AND LOGIC GATE

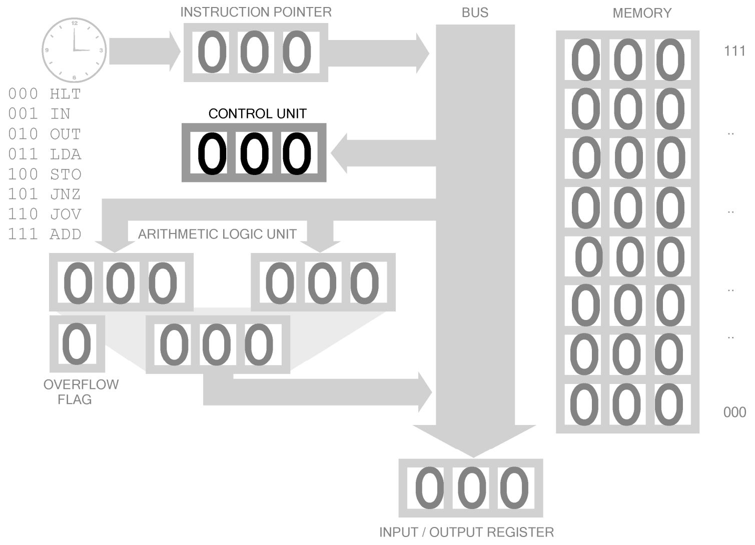
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

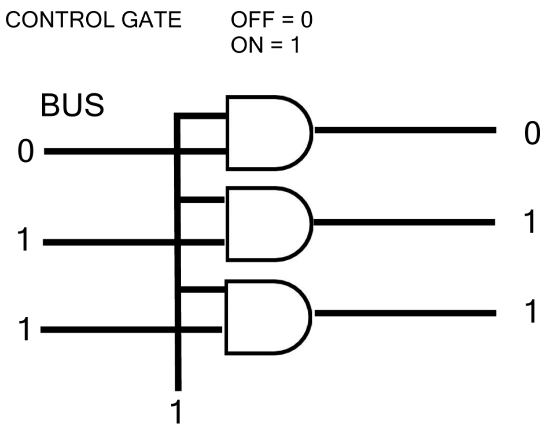


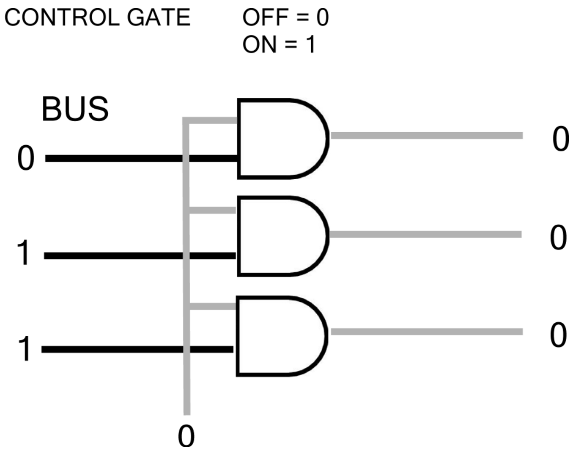
THE SYMBOL FOR THE XOR LOGIC GATE

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

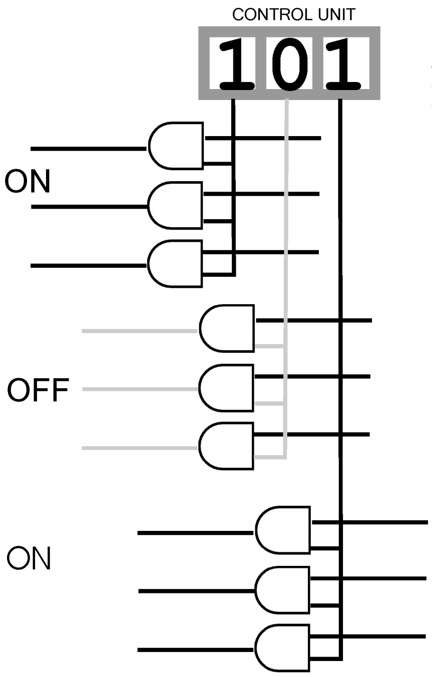
Now we can construct the devices we need for the CPU to function. First the Control Unit. This routes signals along the BUS – the bus is a set of wires linking the devices within the CPU. These wires have switches to allow data (electricity) to flow or not. The switches are like railway points (switches in the U.S.). So if the switches open the BUS at a memory location and the Arithmetic Logic Unit (A.L.U) the signal will flow from the memory location to the ALU.







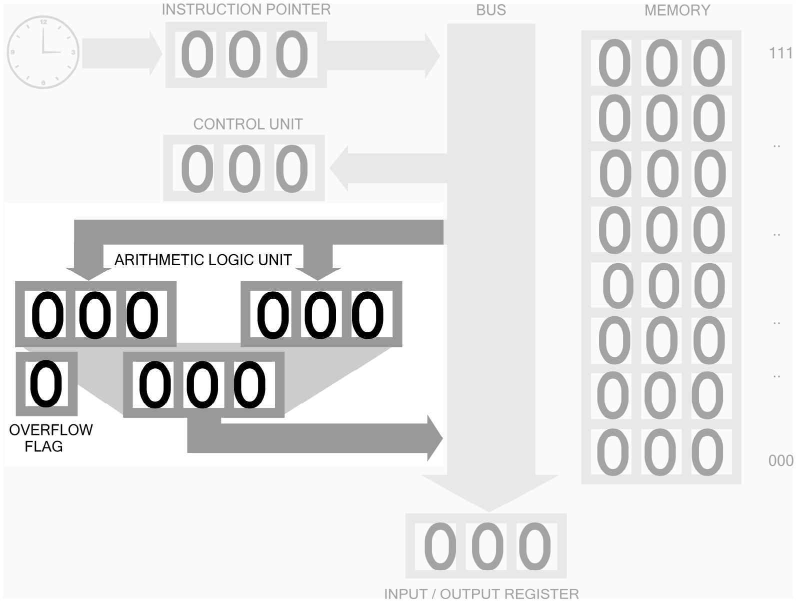
The input of the control gate comes from the bit pattern in the Control Unit. This is termed "decoding" the instruction, but in fact all that occurs is that the 0s and 1s open and close gates on busses using AND gates as above.



The term “decode” is misleading as all that occurs is if the instruction has a zero then those AND gates connected to that part of the Control Unit will be turned OFF and if there is a 1 then those gates will be turned ON - opened.

This whole process of “decoding” or carrying out an instruction is simply the setting of switches on or off. Each instruction’s bit pattern relates directly to a set of switch settings. Even in more complex computers in the final instance this is what occurs, no mater how sophisticated the devices or whatever complicated instructions in whatever computer language.

The Arithmetic Logic Unit (A.L.U.) performs all the data processing. It is again a very simple device which adds binary numbers. Subtraction is performed by a special method of adding numbers. This means a separate Subtraction device is not needed. The “logic” is a simple “If” test. In this simple microprocessor only one register is used, this is sometimes referred to as “The Accumulator”. Other processors might have several “general purpose” registers for achieving the same.



The ALU adds the two inputs and stores the result in the output register. The overflow flag is set to 1 when the result can not be stored in the output. Arithmetic is Binary.

```
8s  4s   2s  Units
--------------------
1    0    0     1        = 8 + 1 = 9 in decimal
```

This number system suits processing using bi-stable (two possible states, on / off of switches) devices. The decimal system uses a base 10. So place values are powers of 10. In binary, place values are powers of 2.

Subtraction is done by using complementary arithmetic. This seems odd but it means no subtraction devices are required, and know programmed subtraction ‘behaviour’ is therefore needed. Multiplication and Division are done by repeated addition and subtractions. 3 x 5 would be accomplished by adding 3s 5 times.

Multiplication and Division can also be achieve by “shifting” bits to the right (division by 2) or to the left (multiplication by 2).

Using 3 bits the highest number we can store is + 7, 111 in binary = 1 x 4 + 1 x 2 + 1 = 7. So the range of positive numbers is zero to seven.

If we use complementary numbers then the range is - 4 to +3.

- 4	2s	UNITS	
0	1	1	= +3
1	0	0	= -4
1	1	0	= -2
1	0	1	= -3
0	1	1	= +3
0	1	0	= +2

If we ADD -4 to +2 this is the result.

```
100 +
010
-----
110
```

Looking above 110 is -2.

So if I want to subtract 3 from 2 we compliment 3 and add. To complement we alter all 1s to 0s and all 0s to 1s and then add 1.

```
011 = 3
100 = (1s complement)
+ 1
----
101 = -3 (2s complement = -4 + 1 = -3)
```

Now add this to 2 (10 in binary)

```
101 +
010
----
111 = -1 (-4 + 2 + 1)  The correct answer without knowing how to subtract!
```

In larger CPUs other circuits are used as well to increase speed but the method above is sufficient for arithmetic. The important point is to realize the underlying principle of keeping things as simple as possible. In order to add larger numbers than the size of registers the programmer simply breaks up the numbers into smaller chunks and adds these with carry when needed. Typical CPUs use 8, 16, 32, and 64 bit registers.

To hold other data such as text some kind of code is needed. For instance A = 00 B = 01 C = 10 D = 11. Our 3 bits limits us to 4 alphabetic characters. 8 bits gives us 256 possible letters (as used in acsii files) – enough for the alphabet of upper and lower case. We could using 3 bits store the alphabet using something like Morse code. If 000 = DASH 111 = DOT and 101 = SPACE. (110 is not used)

```
— —    — — —    • — •    • • •    •
M      O        R        S        E
```

000000 101 000000000 101 111000111 101 111111111 101 111

To perform "Logic" simple arithmetic is used. If we want to test if two words are the same for instance – you type in a password or user name and its checked to see if it is "known" – if it is known the user is logged in if not access is denied. Each letter being represented by a binary number the numbers are subtracted and if the result is not zero then the match is false.

Using A = 00 B = 01 C = 10 D = 11

A – B is 00 – 01 which is -1 so A is not B

C – C is 10-10 which is 0 so C = C

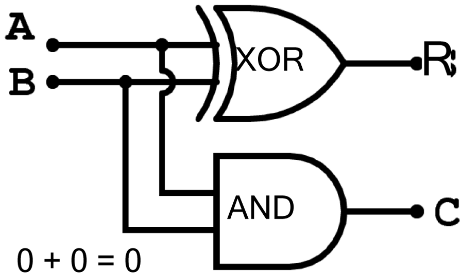
We now have sufficient mathematics, logic and data storage to run a computer program – application or app. And in 'principle' a program of any complexity we currently see around us in games consoles, mobile phones, personal computers and mainframes.

The circuit which is used in the ALU is a compound of "half adder" circuits. The half adder performs simple addition – without a carry input. Full adders are made by combining 2 or more half adders with other simple logic (AND) & (OR) gates. The full adder having 3 inputs – 1 for a carry from any previous addition.

The half adder is made from an OR gate and an AND gate of simple switches as shown above. Notice the output of the half adder has all the rules for addition we need in binary arithmetic.

HALF ADDER

OFF = 0
ON = 1



A	B	R Result	C Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 carry 1

10 = 2 in decimal

HUNDRED TEN UNITS

FOUR TWO UNITS

1 + 1 = 10

1 + 0 = 1

0 + 1 = 1

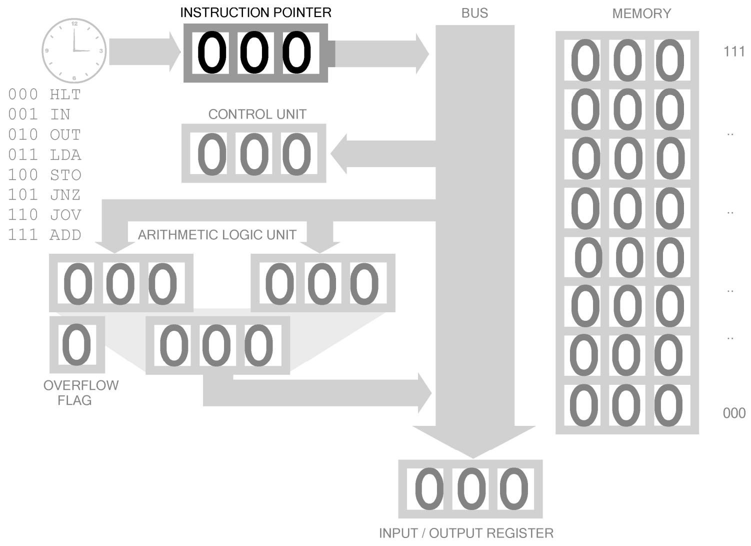
0 + 0 = 0

The rules for binary arithmetic are much simpler than for decimal.

0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 1 = 3
3 + 1 = 4
1 + 3 = 4
2 + 3 = 5
3 + 2 = 5
Etc.

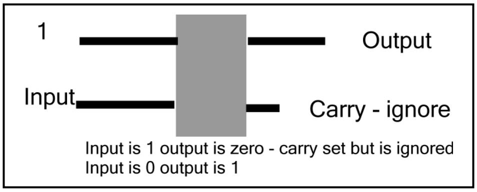
We all “know” these rules of addition now – but we first had to learn these… if a computer was to use decimal numbers the electronics would have to encode all these rules. Only then would the computer “know” the rules. Epistemologically knowledge is deeply problematic in philosophy, however the CPU can perform arithmetic by simple coded operations no different in the input and the output (or result) of human arithmetic. Knowing how to do arithmetic is in this phenomenological aspect identical. The processes of human arithmetic might be philosophically opaque or require some complex neurological explanation, this is not the case in our example CPU. It consists of simple switching mechanisms.

The Instruction Pointer (or program counter) points to an address in memory. Some kind of clock device will increment it each "tick" by adding one to it. In our example this addition could be done via the ALU or the Instruction Pointer (I.P.) or Program counter (P.C.) might have its own ALU adder circuit for this purpose.



How these Addresses in the Instruction Pointer are used to locate a specific memory address is via the use of a set of control gates for each location. These are set by locating the address pointed to by the I.P. using a De-multiplexer. This uses AND gates and NOT gates. A NOT gate simply reverses the input logic, the state. 1 become 0, 0 becomes 1. Or true becomes false, false becomes true. Its very simple to construct using a transistor - in effect it's a switch which is open when it receives power and closes - allowing a current to flow when power is stopped. Everyday examples are solenoids and relays.

It can be constructed using a half adder with one input permanently high (1)

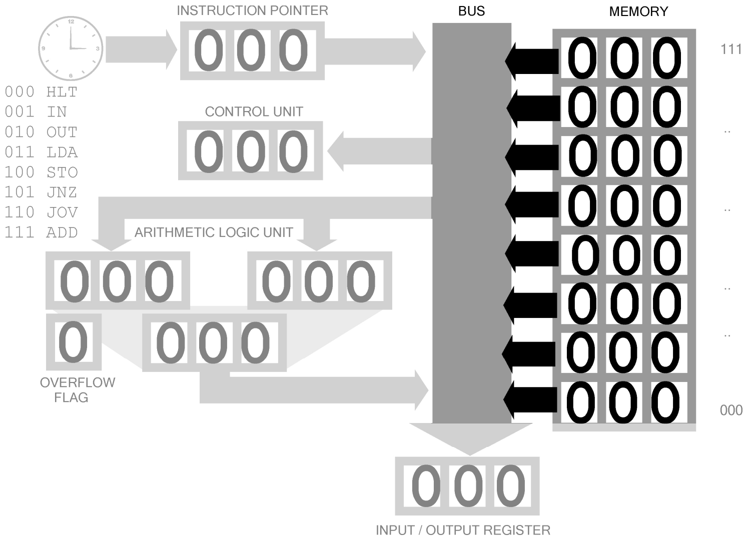


It is a feature of these simple logic gates that we can make one type of gate out of a mixture of others.

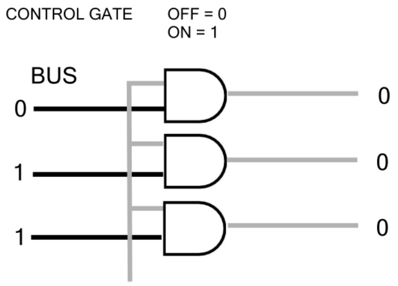
The AND gates here have more than two inputs but the logic remains the same. ALL the inputs must be true for the OUTPUT to be true.

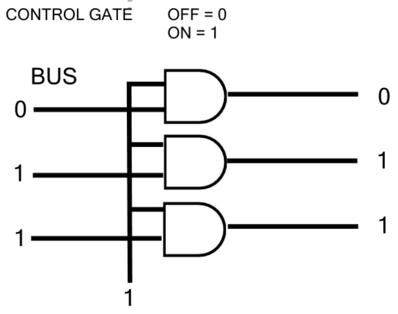
Inputs	Output
000	0
001	0
010	0
011	0
100	0
101	0
110	0
111	1

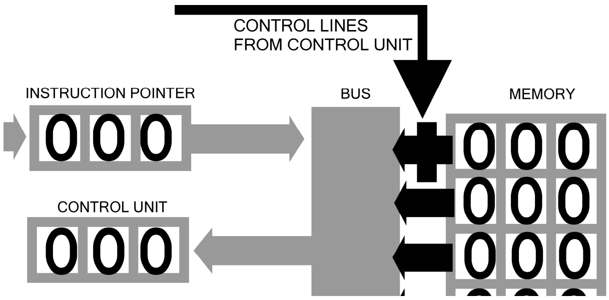
The data and instructions from memory are selected via decoding an address which sets the control lines – blocking all unwanted locations and selecting the one to which the address points. This is called decoding, which can have anthropomorphic implications for a process which uses just simple switches.

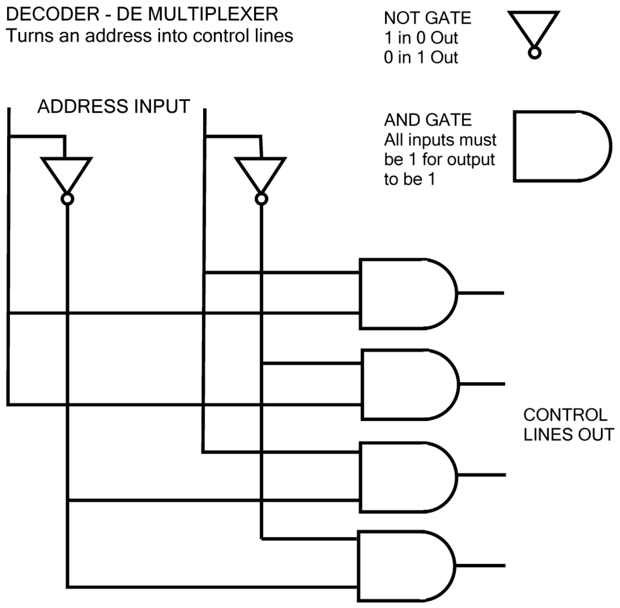


A control gate uses a single line to control several other lines – or BUS.

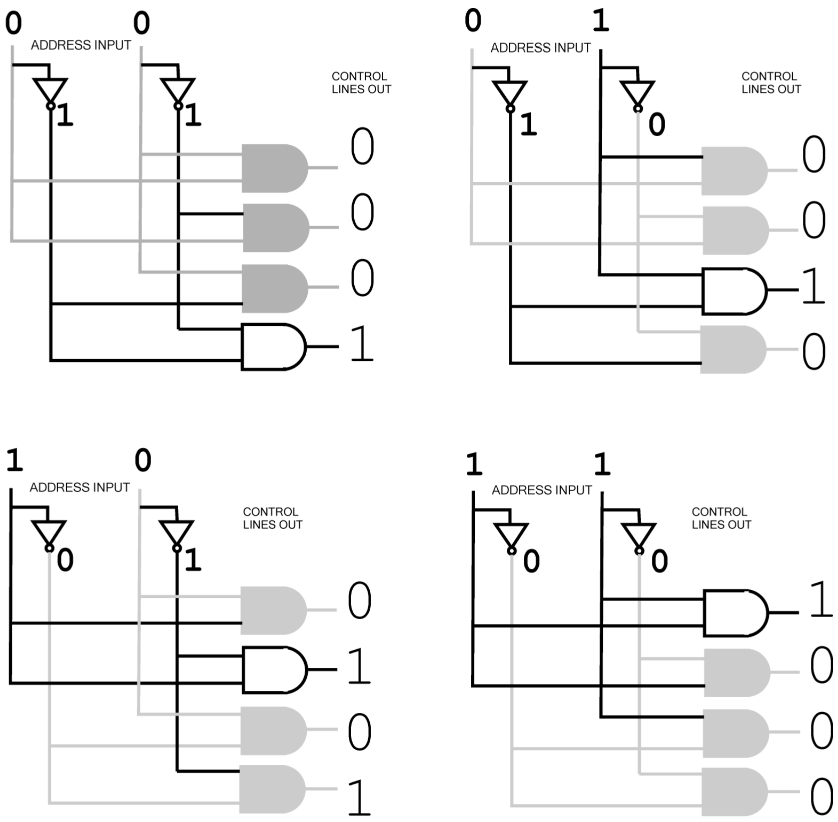

CONTROL GATE
OFF = 0
ON = 1
BUS
0
1
1
0
0
0


CONTROL GATE
OFF = 0
ON = 1
BUS
0
1
1
0
1
1
1


CONTROL LINES
FROM CONTROL UNIT
INSTRUCTION POINTER
000
BUS
MEMORY
000
000
000
CONTROL UNIT
000



THE PRINCIPLE OF "DECODING" USES SIMPLE SWITCHES
MORE LINES ARE ADDED TO DECODE LARGER ADDRESSES
BUT THE MECHANISM REMAINS THE SAME.



MEMORY ADDRESSES 00 - 11 (ZERO TO THREE)
ARE "DECODED" TO ALLOW ONLY THE
APPROPRIATE CONTROL LINE TO OPEN THE GATE
FOR THIS ADDRESS ONTO THE BUS

The example above uses a 2 bit address and 4 memory locations. Our CPU uses a 3 bit address and 8 memory locations. The principle is the same - the AND gates merely take 3 inputs. Our CPU is a theoretical object simplified in order to gain a direct experience and knowledge of the object. However actual computer hardware we have said is no different in quality. In the case of a three address to 8 location decoder there actually exist physical devices for this task.

FAIRCHILD
SEMICONDUCTOR™

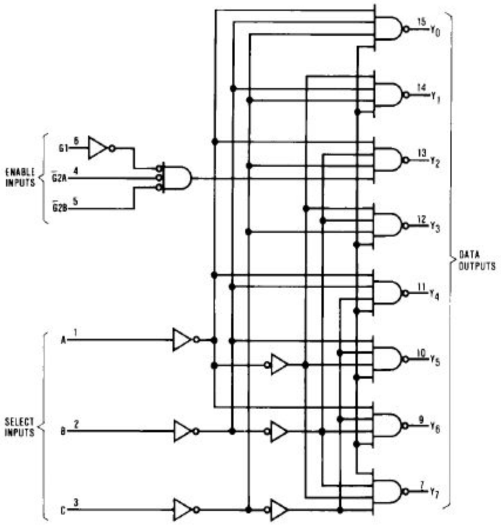
September 1986
Revised February 2000

DM74ALS138
3 to 8 Line Decoder/Demultiplexer

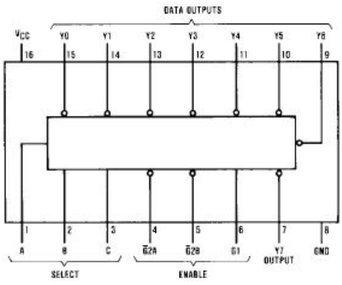
Features

- Designed specifically for high speed:
  Memory decoders
  Data transmission systems
- 3- to 8-line decoder incorporates 3 enable inputs to simplify cascading and/or data reception

Logic Diagram



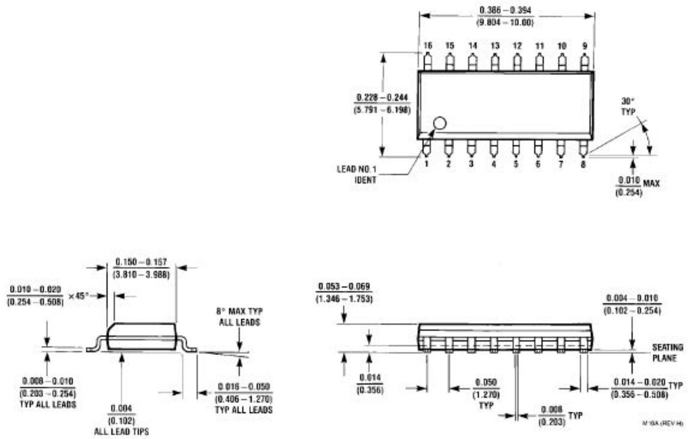
Connection Diagram



Function Table

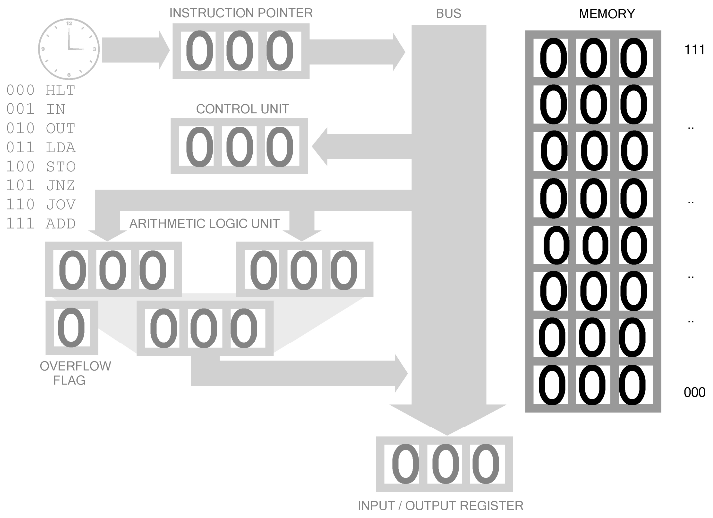
Enable Inputs		Select Inputs			Outputs							
G1	$\overline{G2}$ (Note 1)	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

Physical Dimensions inches (millimeters) unless otherwise noted



16-Lead Small Outline Integrated Circuit (SOIC), JEDEC MS-012, 0.150 Narrow
Package Number M16A

The extra inputs just select the device’s operation mode – again using simple AND logic.



The simple nature of using only a three bit address means that only 8 locations can be used as memory, we can have a maximum of 8 instructions, and can count only up to decimal 7. (111 is 1 four plus 1 2 plus 1 one.)

2^3 is 8. Given 8 locations of 3 bits gives us 24 bits of memory.

000000000000000000000000 = 24 bits all set to zero

Running through the possible sequences of zeros and ones in memory we in effect count from 1 to 2^{24}.

000000000000000000000000
000000000000000000000001
000000000000000000000010
000000000000000000000011
000000000000000000000100
000000000000000000000101
000000000000000000000110
000000000000000000000111
000000000000000000001000
....
111111111111111111111111

Each of these represents a “Program” which we can store and attempt to execute in the CPU.

Working through all the possible sequences allowed using 0s and 1s in 24 bits will give us ALL of the possible "programs" / "data" which our CPU can store and process. Some – 0000000000000000000 will do nothing at all, most will do nothing at all.

Counting up all these sequences we find there is in total 16,777,216. This is the totality of what this simple CPU can code. The CPU has a totalizable universe, however even with only 3 bits this is over 16 million variations. Most personal computers have memories of gigabytes. 1 gigabyte is 8,589,934,592 bits, 8 billion bits – so a potential for well over $2^{8 \text{ billion}}$ possible programs/data exists for personal computers of 1 gigabyte of RAM or more.

Each time any input is required during any of the programs, (The execution of the IN Instruction) any one of 8 possible inputs can occur, and will be loaded into the Arithmetic Logic Unit. Therefore all possible input permutations would also need to be examined if a "total" picture of he CPUs totality of states is to be 'known'. In the case of our CPU this is 8 possible inputs – (000, 001, 010, 011, 100, 101, 110, 111) for every time the IN Instruction was encountered in the list of all total programs. Larger computers allow for far more data to be input which would further increase the number of totalizable machine states for those machines.

To compare human brains to computer memories is difficult but it has been estimated "the total information storage capacity of the synapses in the cortex would be of the order of 500 to 1,000 terabytes." A simple mechanistic view shows the difficulty in thinking 'metaphysically' with these very large numbers. By which I mean the scales though known and determinate are not easy to experience. Human thinking in terms of logic and arithmetic can appear similar if not identical in its results if not methods to those of digital computers, but other human behaviour exhibits far more subtle variations. This supposed ability for digital computers to make logical decisions yet fail to exhibit "emotion" is an obvious source of novelty for science fiction. As if a few more "gates", and larger registers would or could somehow alter the behaviour of a fixed state machine to the extent it becomes "human". The more 'pessimistic' idea that human novelty and subtly might just be an appearance found in larger fixed state machines is not as popular a theme in fiction but is in the cybernetics of "uploading" human minds into computers in order to gain some kind of evolutionary advantage, such as possible immortality.

Even with possibly the simplest of 'objects' – our 'theoretical' fixed state CPU the permutations of states are large enough to require years of study to fully 'know' and experience all of these states. Real objects such as simple as micro-organisms, molecules or crystals will have far many more possible states. I'm aware of the ideas of 'probability' and 'uncertainty' associated with Quantum Physics in the study of such objects, though I am in no position to discuss this further complication of our possible knowledge of actual objects, however given a set of probabilities I can not see why in theory if not in practice all of these possible states can not be 'known' and explored. The result would be a set of possible states of a colossal number, but it would still be finite. Such numbers are incomprehensible *for humans*. Such numbers – colossal determinate states - relate to even the simplest of 'real' objects. This offers an alternative explanation for the seeming limits of human knowledge which is not *metaphysically* excluded or limited, but *physically, quantitatively* removed or excluded from human experience. Metaphysics' 'unknowable' 'Thing in itself' – Kant's 'Ding an sich' - might just have in its totality a fixed and knowable limit, but one that is far greater than the possibility of 'human' comprehension. Contemporary philosophers (Object Oriented Ontologists) argue that the simplest of objects "withdraw" from us in some infinite and mysterious way, this might be the case, or it might be that their totalizable permutations are not totalizable *for humans*. This implies some form of correlationism in their thinking – an anthropocentricity which seems to think that the unthinkable and the unthinkable *for-humans* are in some way identical, when in simple binary instruction sets, their unthinkability lies only in the size of permutations and not in some subtleties of possible or probable essences.

Similar fallacies of spookiness occur in imagining sub-atomic particles or the scale of cosmic distances. Our pictures of the subatomic and the cosmic are representations which omit the magnitudes involved. For instance if our sun was the size of an audio CD (5 inches / 9 cm) then the nearest star would be a disk just over half an inch, 830 miles away. Yet in science fiction "worlds" can and often do collide.

Though we have knowledge of simple objects, we cannot fully experience them, and we resort to a metaphysics of remoteness, of "withdrawnness" which becomes like if not actually 'spiritual' religiosity, a poetics, towards these physical realities which are present to us. Because I can not experience the totalizable possibilities of an object then I can simply call this infinite and say its withdrawn from me. Even a simple object like a personal computer. However the impossibility is not one of sophistication but simply of quantity. Our simple CPU allows us to grasp its totality, even if its 16 million. If I count or write 2 digits each second it would take, working an 8 hour day, 277 days to count and experience each possible state. To know each of the states these programs would put the CPU into would require much longer time, perhaps many years, but it is with our simple CPU achievable. This would represent, would *be*, the full experience of our CPU's totality. This is possible, however counting to $2^{8 \text{ billion}}$ is not possible, so not experienceable. But the reason this is not experienceable is not that it is more intellectually difficult, but simply we do not live long enough to be able to do this.

Given we walk at 4 miles an hour and live some 70-80 years that gives us our longest possible walk. (39,967,500 miles – our sun is 91 million miles away) There is however nothing mysterious, transcendentally mysterious, metaphysically mysterious or alluring, in longer distances, the difference is that we can not directly experience them, even if we can know them. Their withdrawnness is nothing mysterious, is no more mysterious than the object being on a shelf too high for us to reach. The actual experience of a simple CPU or of a walk of 20 miles gives us the ability to appreciate this withdrawnness as what it is for us.

Our simple CPU has a total of 16,777,216 programs and no more. It *is* possible to know each of these and discover what they do. This might take some years but at the end of this laborious process the object will have been fully realised to *us*. There is then no problem such as Turing’s Halting problem, we will know all the outcomes, there will be no uncertainty, no Gödel like “events”.

Why we cannot do this for other objects – a least other digital objects is simply a matter of time. We can never complete these tasks due to our limited life expectancy and not due to some limitation of cognition.

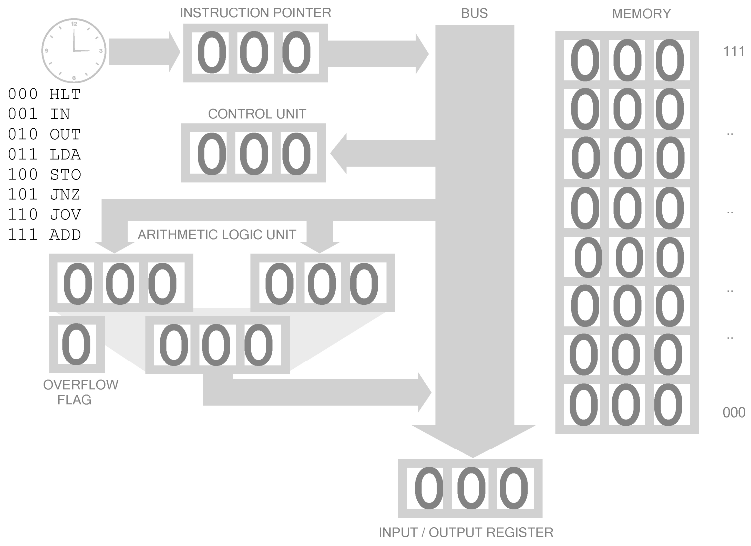
An even simpler “universe” or object of two bits consists of 4 states in total. These can be “fully” realised.

0	0
0	1
1	0
1	1

Once we have understood the basic CPU operations rather than scale these up into the humanly incomprehendable sizes of contemporary computers we can scale down to a “two bit universe” where our knowledge can be complete.

And this is what the second part of this book will actually do, and in doing so we can have an “aesthetic” of a complete object. And in doing so strangely there will be no uncertainty, no Gödel like “events”. There will be no ‘Halting problem’, we will know all the outcomes, we will even see that any program which terminates prior to its totality has – without recourse to its totality – a fixed outcome.

If we wanted to fully experience and know the totality of states of our simple CPU we first need to understand its machine code. Machine code is code written with the actual instructions of the processor, ours here has 8, the early Intel 4004 had 256 Instructions the "The total number of x86 instructions is well above one thousand".... These “Machine Code” instructions of our CPU are again very simple, but are no different in kind to the much larger instruction sets of “real” CPUs. Furthermore ALL programming languages are decoded or translated into these simple instructions, it is these simple machine instructions – AND NONE OTHER – that any computer actually executes.



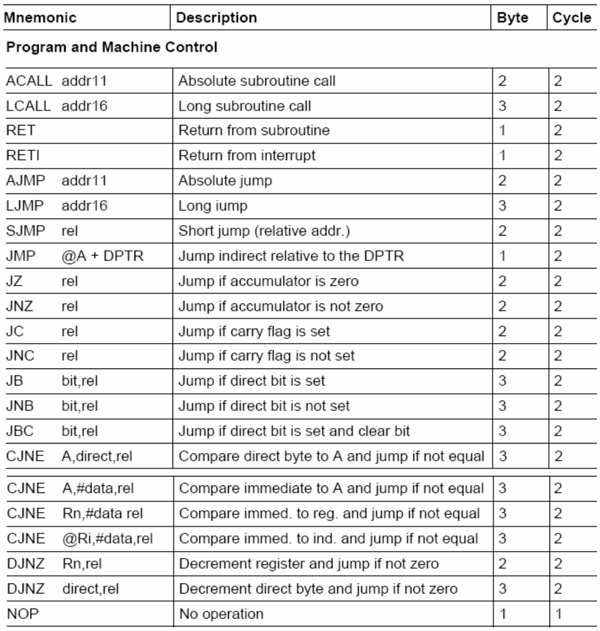
Operation Code - Op Code	Mnemonic (abbreviated text)	Meaning
000	HLT	HALT - Stop running
001	IN	Get input from input reg and load it into the ALU
010	OUT	Put the output of the ALU into the output register
011	LDA	Load the ALU with data from the given (following) address
100	STO	Store the ALU's output at the given address
101	JNZ	Branch (go to) the given address if the output of the ALU is NOT zero
110	JOV	Branch to given address if overflow flag set
111	ADD	Add the two registers in the ALU

All the first computer languages were machine code. Hardwired circuits sometimes, or switches set appropriately. i.e. to set the LDA instruction the memory location would have 3 switches set to OFF ON ON. The difficulty of "reading" these programs led to the first artificial computer language called "Assembly" Code. Here rather than use binary digits simple mnemonics are used. As shown above in our instruction set chart. A program was then used which translates these into actual binary code. There is a one to one relationship between an assembly mnemonic and the ACTUAL binary code. Only the actual binary code is ever executed. Later "Higher Level" languages were introduced to make programming simpler for humans, these still need to be "translated" into machine code. Because of the seeming sophistication of these high level languages there has developed amongst even computer programmers "myths" about languages and operating systems. All languages such as those used in Artificial Intelligence, or in searching databases… no matter what function they perform – unless they are first translated into simple machine code, these programs, cannot be executed. A powerful search algorithm such as this-

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern
```

cannot run on any computer without being translated into machine code. In this case many lines of simple machine code would be generated to perform the task by shifting and comparing (by subtraction) bits. Programs written in C++, Java, operating systems such as Linux and Windows, Browsers, computer games, video playback and music… all run as simple machine code. Java runs in a "virtual computer" – the Java Runtime – which will be machine code. Because Machine Code is specific to a particular processor, a program written for an Intel CPU will not run on any other processor whose architecture (Bus size, registers etc) is different. AMD processors "copy" the Intel architecture. This creates a problem for 'porting' programs from one CPU architecture to another. A program running on an Intel / Windows "platform" (Platform is the Hardware and operating system) will not run on an Android device, without re-compiling – re-translating. The Java language tries to solve this problem by creating a "virtual" java computer for each "platform", so the java programs can run independent of any particular platform. If your phone or computer has a Java Runtime written for it and installed on it, it can run any Java Application. The downside is the running of the "Virtual Java" machine slows things down.

In our instruction set some instructions are “complete”, HLT, IN, OUT. These use just 3 bits. LDA, JNZ, JOV, STO, ADD all require an address following the instruction. For instance the STO instruction stores the output of the ALU at some address - so the next 3 bits following it will be that address. Our computer uses a 3 bit “word”. Most computers use 16, 32 or 64 bit words. So some instructions are “Double Word” instructions. Again this is typical in many CPU architectures.



Mnemonic	Description	Byte	Cycle
Program and Machine Control			
ACALL addr11	Absolute subroutine call	2	2
LCALL addr16	Long subroutine call	3	2
RET	Return from subroutine	1	2
RETI	Return from interrupt	1	2
AJMP addr11	Absolute jump	2	2
LJMP addr16	Long iump	3	2
SJMP rel	Short jump (relative addr.)	2	2
JMP @A + DPTR	Jump indirect relative to the DPTR	1	2
JZ rel	Jump if accumulator is zero	2	2
JNZ rel	Jump if accumulator is not zero	2	2
JC rel	Jump if carry flag is set	2	2
JNC rel	Jump if carry flag is not set	2	2
JB bit,rel	Jump if direct bit is set	3	2
JNB bit,rel	Jump if direct bit is not set	3	2
JBC bit,rel	Jump if direct bit is set and clear bit	3	2
CJNE A,direct,rel	Compare direct byte to A and jump if not equal	3	2
CJNE A,#data,rel	Compare immediate to A and jump if not equal	3	2
CJNE Rn,#data rel	Compare immed. to reg. and jump if not equal	3	2
CJNE @Ri,#data,rel	Compare immed. to ind. and jump if not equal	3	2
DJNZ Rn,rel	Decrement register and jump if not zero	2	2
DJNZ direct,rel	Decrement direct byte and jump if not zero	3	2
NOP	No operation	1	1

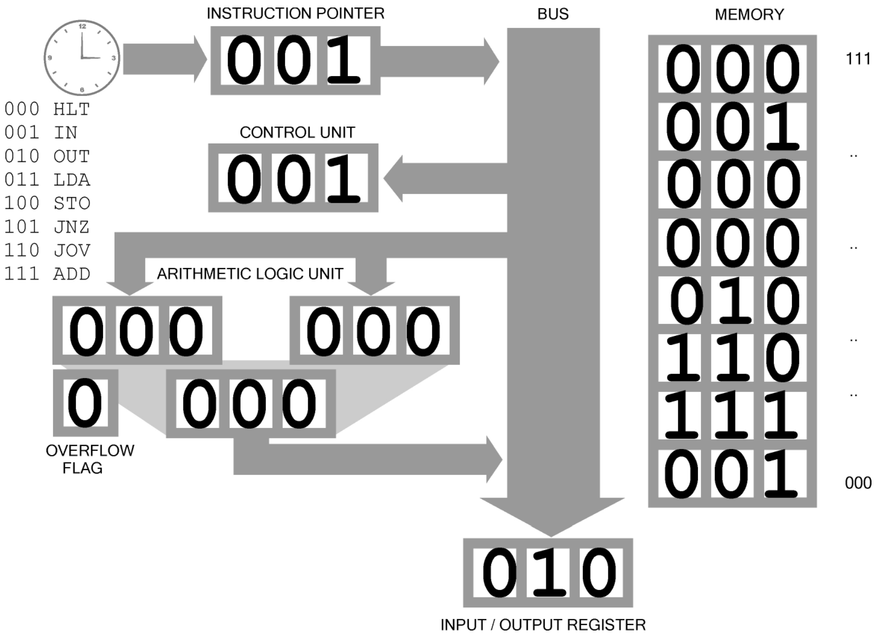
Above is an “REAL” instruction set. It uses an 8 bit WORD and has 1,2, and 3 word instructions.

The operation sequence of the CPU. The CPU uses the Instruction Pointer to find the address of the next required instruction, this instruction is loaded into the Control Unit. The CPU then Increments the Instruction Pointer and only then is the current instruction executed by decoding it, by setting the appropriate gates via control lines to the computers BUS. For 2 word instructions this “cycle” occurs twice, the second time to access the address. Each Fetch (Instruction) , Reset (Increment) and Execute is a cycle. So in our CPU some instructions take one cycle others two.

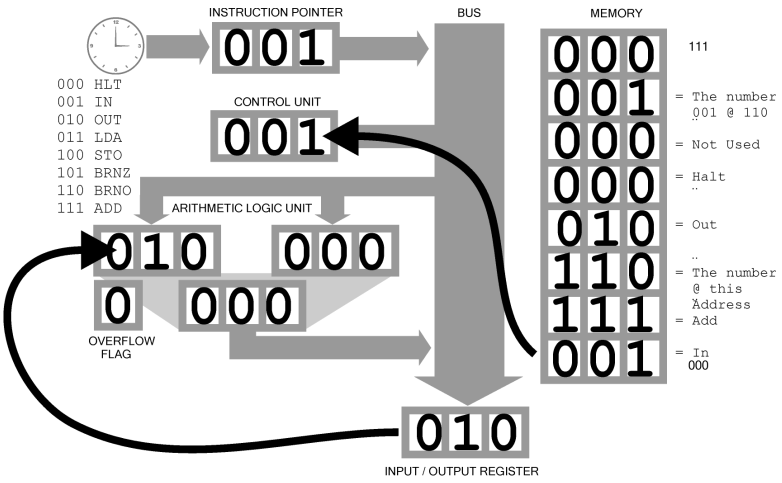
We will now look at each instruction.

IN

This gets the contents of the Input/Output register and stores it in the ALU.



We have a simple program in memory. How this is executed in detail will be explained latter.



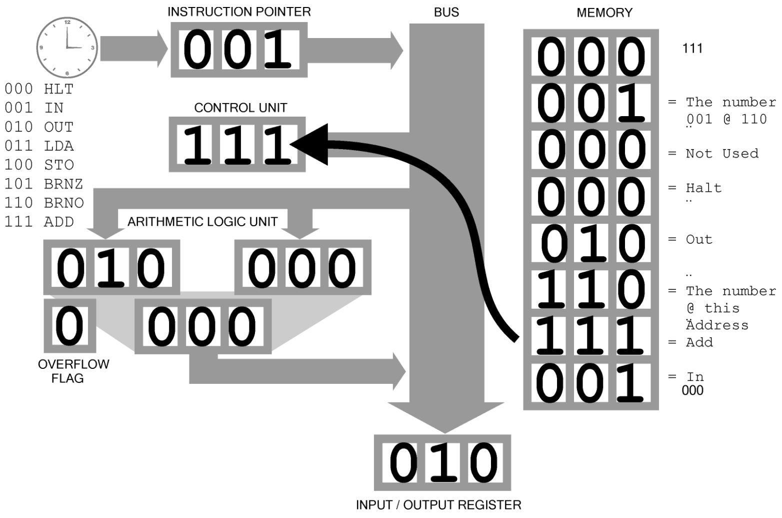
The IN instruction copies the contents of the Input/Output register to the left side of the ALU (Sometimes called the Accumulator). Other CPUs might not have a single “Accumulator” register but several. Again to speed up processing - more quantity - not quality.

How the I/O register is copied to the ALU is accomplished is by the “Decoding” and setting of control gates so the data in the I/O register flows into the left side of the Accumulator. All the other gates being set closed.

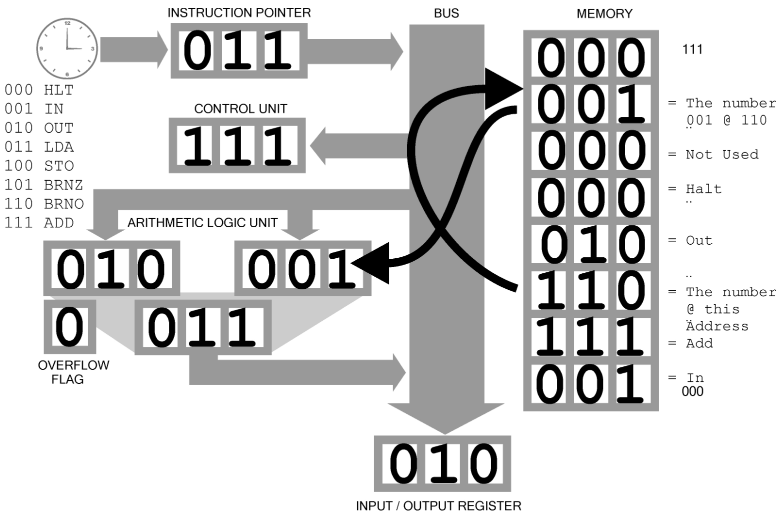
ADD

This copies the data at the address given by the next 3 bits into the right side of the ALU (accumulator), again by setting the appropriate control gates, then adds using the adder circuits described above, storing the result in the output side of the ALU. If when adding an overflow occurs the overflow flag is set.

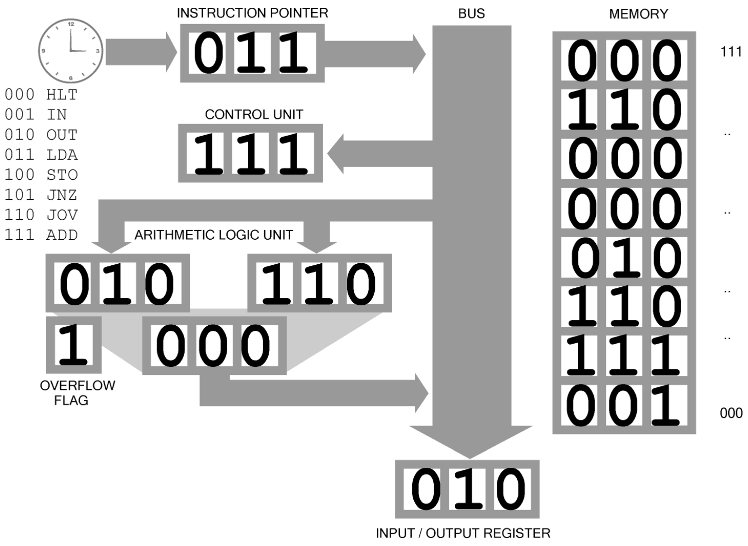
The ADD instruction 111 is at memory location 001. It is placed in the Control Unit.



Here the following the ADD instruction, 111, is the address of where to get the data to add. In this case 110. The location 110 has the value 001 stored in it.



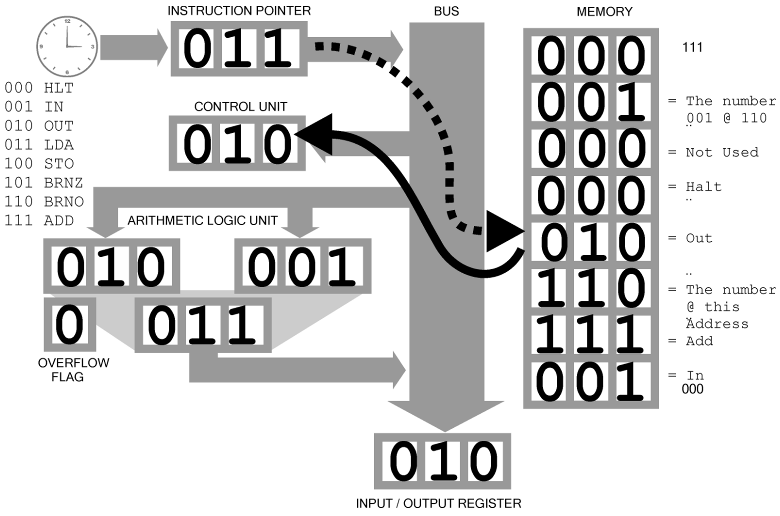
The data is fetched and placed in the right side of the ALU. The two numbers are Added using the Adder circuits described above and the result stored in the ALU's Output register. 010 + 001 = 011 (2 + 1 = 3)



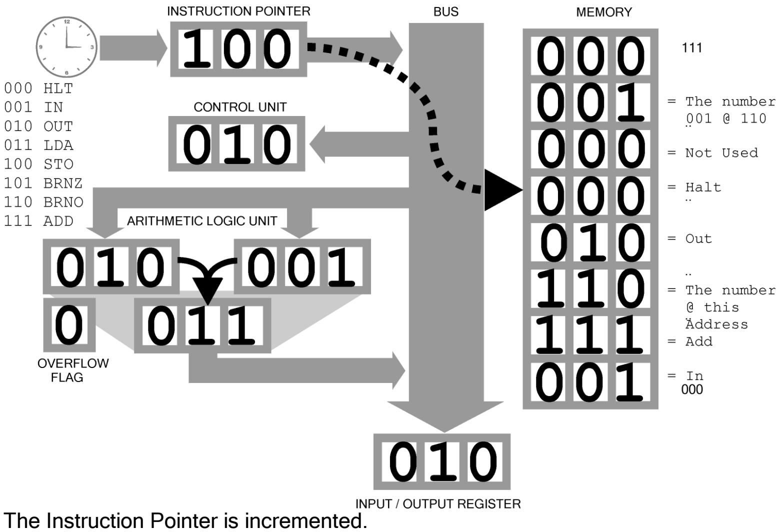
Here the data at Address 110 is 110. When added to 010 the result is an overflow. 010 + 110 = 1000 (2 + 6 = 8) . In practice programmers would need to be aware of the possibility by testing the flag after addition then completing the addition by using the carry into another addition process. Our limited CPU is not capable of this, not from any lack of processing ability but by the size of its memory.

OUT

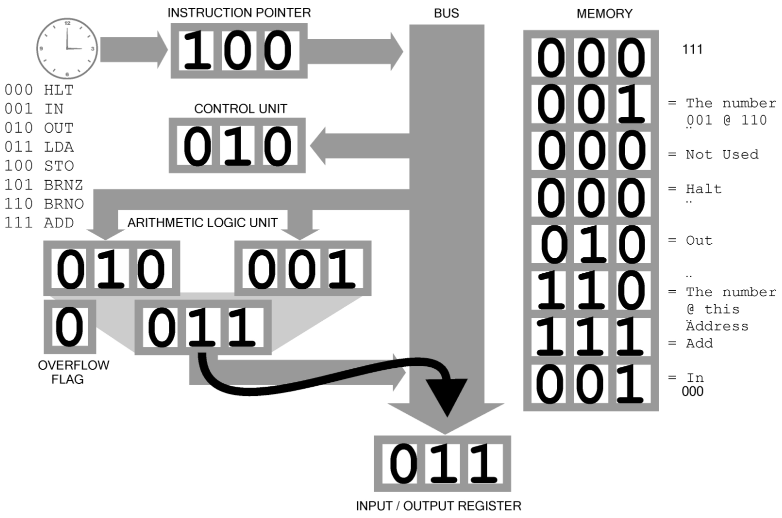
The OUT instruction copies the contents of the ALU's output register onto the BUS and into the Input/Output Register.



Above: The instruction is FETCHED and placed into the Control Unit



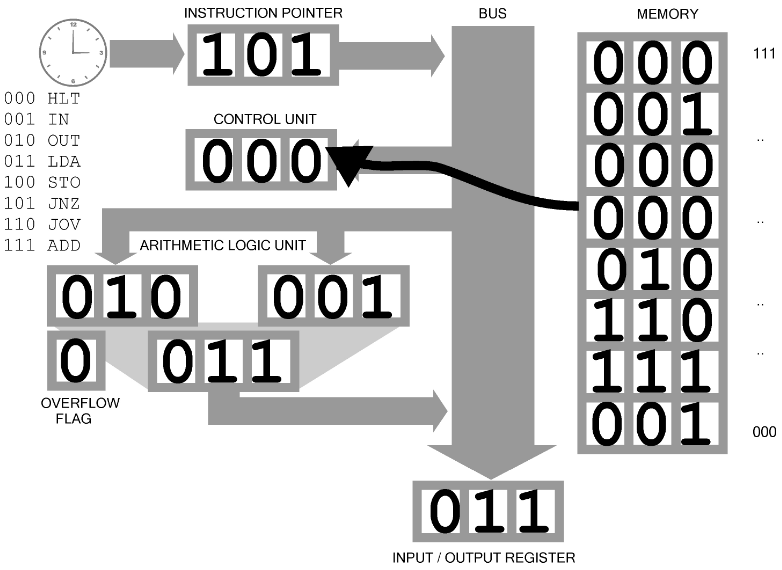
The Instruction Pointer is incremented.



So we display the output of the result of our addition. The Instruction pointer is now pointing to 100 in memory.

HLT

At location 100 in memory is the code 000. The HALT instruction. This is loaded into the Control Unit and executed. The Instruction pointer is incremented and then the program stops.



The Halt instruction is loaded into the Control Unit.

The Instruction pointer is now pointing to location 101. This is because the processing sequence in CPUs is typically – Fetch Reset Execute.

Fetch = Get the instruction from the address in the Instruction pointer and load it into (copy it into) the Control Unit.

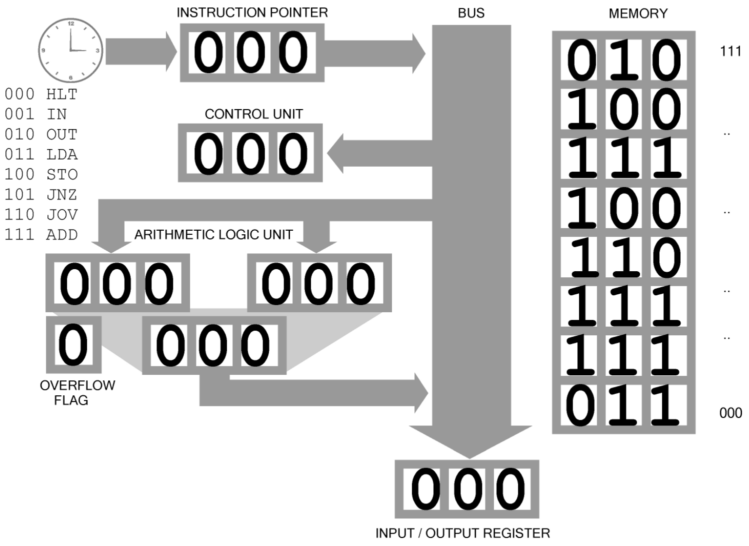
Reset = Increment (in our case add 1) the Instruction pointer (so its pointing to the next instruction- (the address in memory of this instruction).

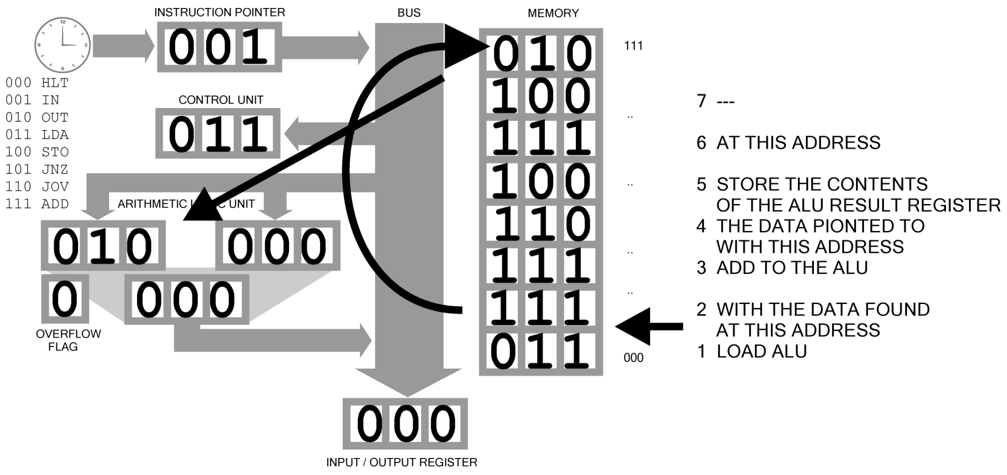
Execute = Decode the instruction.

This F.R.E. cycle occurs at set intervals dictated by the clock, (The speed of the CPU) or if manually processed by simply changing the values in the Instruction pointer – using switches. The latter used to take place in de-bugging, stepping through the program slowly examining the registers to see precisely what was happening. Modern high level languages still allow this to take place – even at the level of monitoring the CPUs registers as the code is run one instruction at a time.

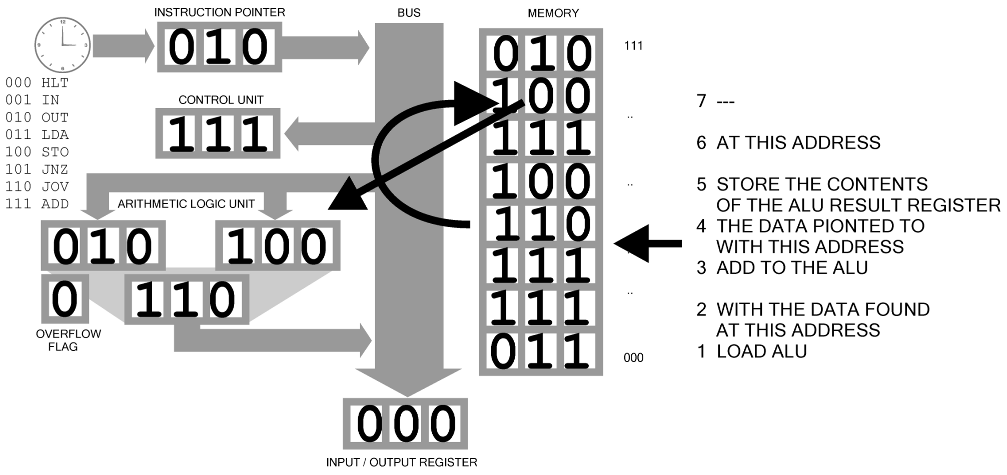
LDA
STO

Load ALU (Sometimes called the accumulator – "ACC") & Store the contents of the result register – the output of the ALU.

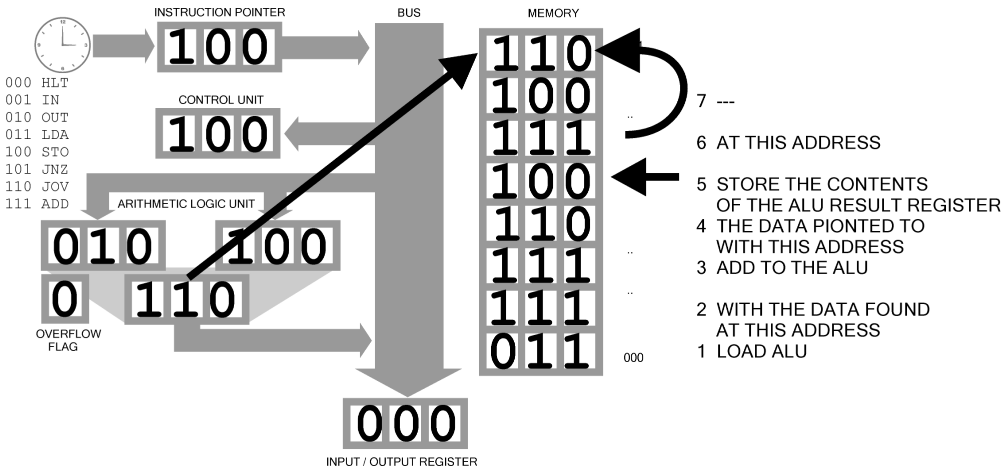




The ALU is loaded with the data at the address 111 in memory = 010 (2).



The ADD instruction uses the address following it - 110, to locate the data. Looking at location 110 we see it contains the data 100. This is placed in the right side of the ALU and added to the left, the result stored in the ALU's output register. 010 + 100 = 110



The STORE instruction stores the result at the location given - 111. This overwrites what was in that location. The data is 110 or 6 in decimal. The program loaded the ALU with the number 4, ADDed the number 2 and stored the result 6 at location 111.

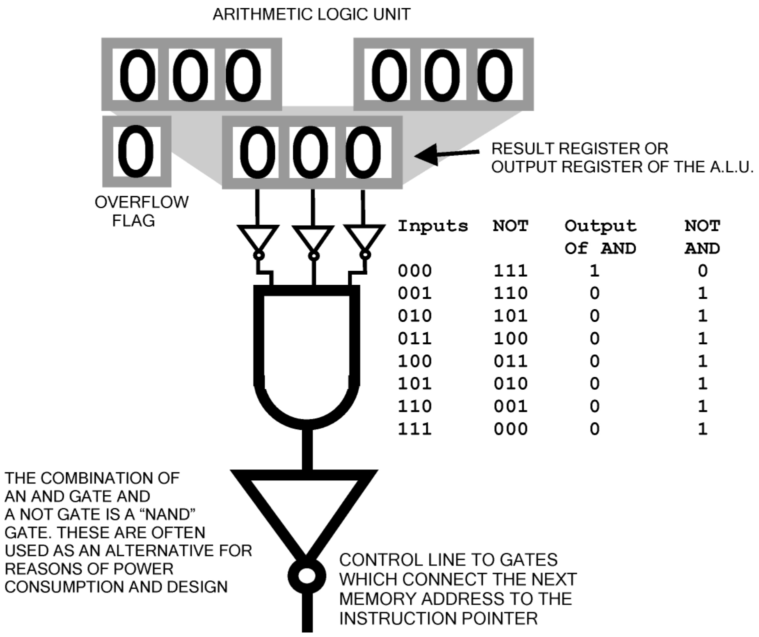
JNZ
JOV

The two jump instructions allow the CPU to makes "decisions". Anthropomorphist interpretations of this action should be avoided unless a radical re-appraisal of human thinking is to take place, and this is not suggested here! Once again it is the operation of simple switches, no more complex than those found in a domestic home. The decision allows not only one of two outcomes to be selected it also allows looping and repetition of code.

The JNZ instruction (op code) will copy the contents of the address given by the next 3 bits into the Instruction Pointer, and so program execution will continue at that address and not sequentially, i.e. not the address following the JNZ instruction. This occurs only if the contents of the Accumulator's output register is not zero. This allows all decisions to be made. The JOV instruction does the same if the overflow flag is set, and is needed in arithmetic where numbers or results are larger than 3 bits. Once again "real" processors are no different other than the number of JUMP instructions will be greater- an increase in quantity not quality. These might include an unconditional JUMP or goto, jump if zero, jump if greater than zero etc.

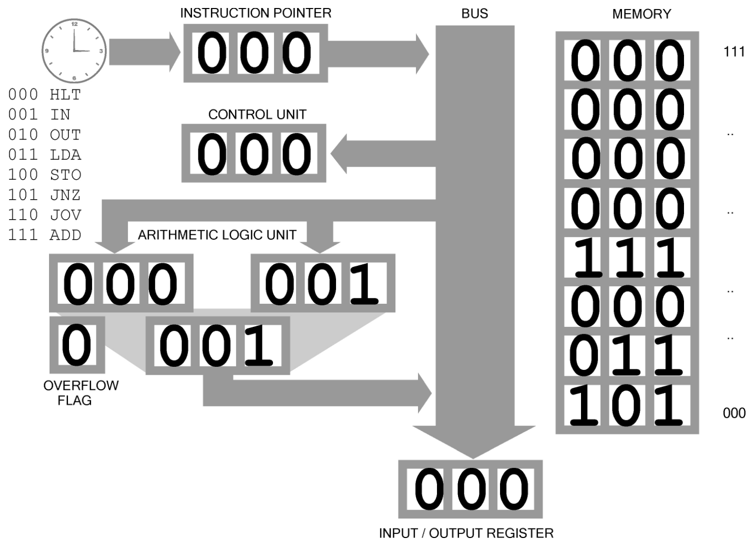
The "decision" process is a simple decoding and setting of switches on the bus which copies the contents of the following memory location to the Instruction Pointer. This is similar to railway yard shunting operations even to the extent that the U.S. term for railway "points" (U.K.) is "switch".

The "test" is extremely simple. The 3 bits of the output register (A.K.A. result register) of the A.L.U are inverted (negated) via 3 "NOT" logic gates, "ANDED" and this result inverted through a NOT gate. This gives a "true" control line only if the ALU result register is not zero. This can be seen by looking at the "truth table" for the operation.



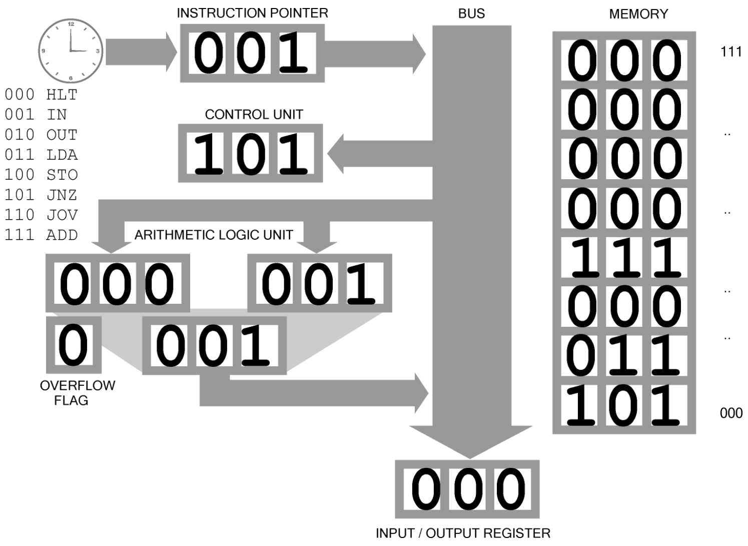
Inputs	NOT	Output Of AND	NOT AND
000	111	1	0
001	110	0	1
010	101	0	1
011	100	0	1
100	011	0	1
101	010	0	1
110	001	0	1
111	000	0	1

Now we will step through the process of execution of the JNZ - Jump IF NOT Zero Instruction.

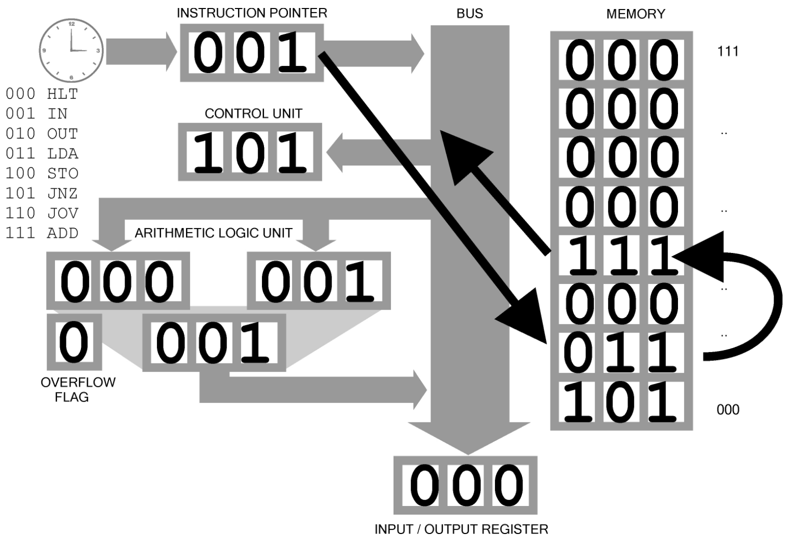


For simplicity the program begins with a non zero in the ALU result/output register, yet we start at address 000. (normally some operation by the ALU would precede a Test and Jump instruction.) The first instruction to be loaded into the Control Unit will be 101 - the JNZ - Jump if NOT Zero instruction. At the next location which is 001, is a pointer to where the address is to be loaded into the Instruction Register IF THE ALU is NOT ZERO. Use of such addressing techniques is common in CPU architecture. Though this may seem at first confusing it allows greater flexibility in programming and is no more sophisticated than regarding someone's address as “two doors down the road” - or “take the second turning on the right”..

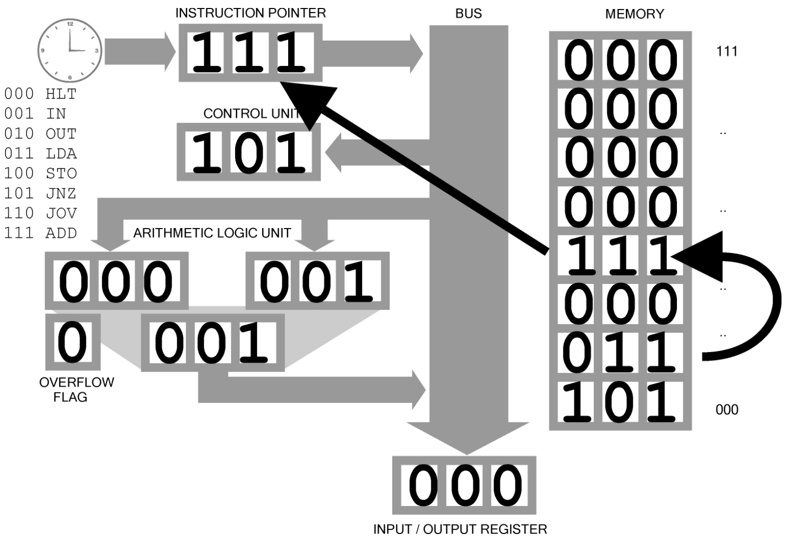
Addresses can be ABSOLUTE i.e. 12 OAK Street- or RELATIVE - take the third left from where you are now.



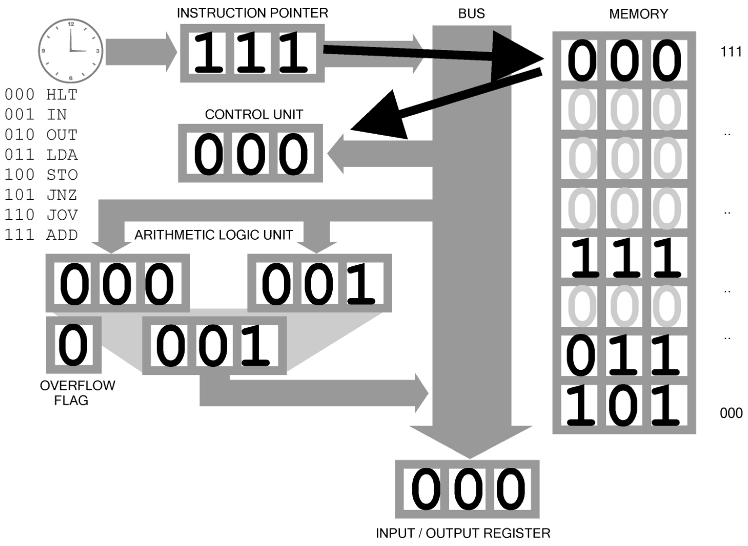
Above the JNZ instruction is LOADED with the "FETCH" and the Instruction Pointer incremented to point to the next memory location - 001 "RESET"



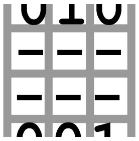
The JNZ is decoded which uses the pointer at 001 (011) - goes to that address - which contains 111, and then will load that (111) into the Instruction Pointer, causing it to “JUMP” (or point to) to that (111) location.



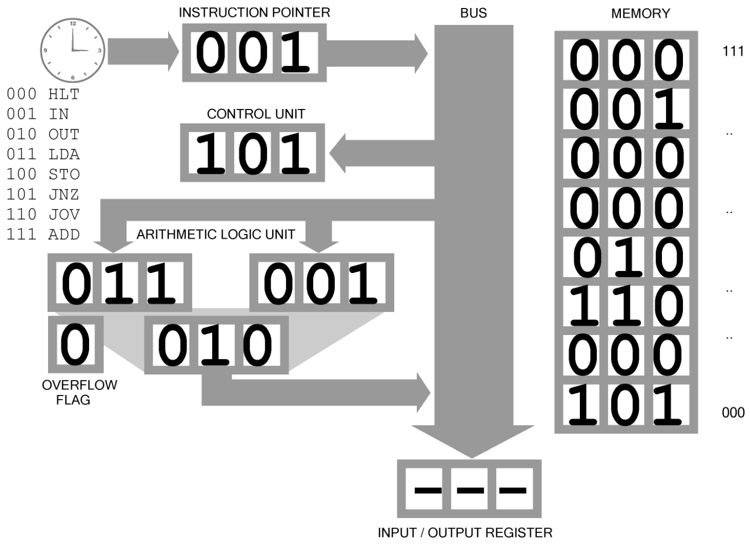
The Instruction Pointer now points to address 111 which contains 000 which will be interpreted by the Control Unit as HLT- Halt. NOTE: The binary 000 can be a number (0) an address (the first location in memory) or an Instruction – HLT (Halt). Likewise 010 is either decimal 2, or the third location in memory, or the OUT instruction. This is a common feature in CPU architecture. Data, Addresses and Instruction all share the same memory space. This is a feature of Von Neumann architecture (Named after its inventor).



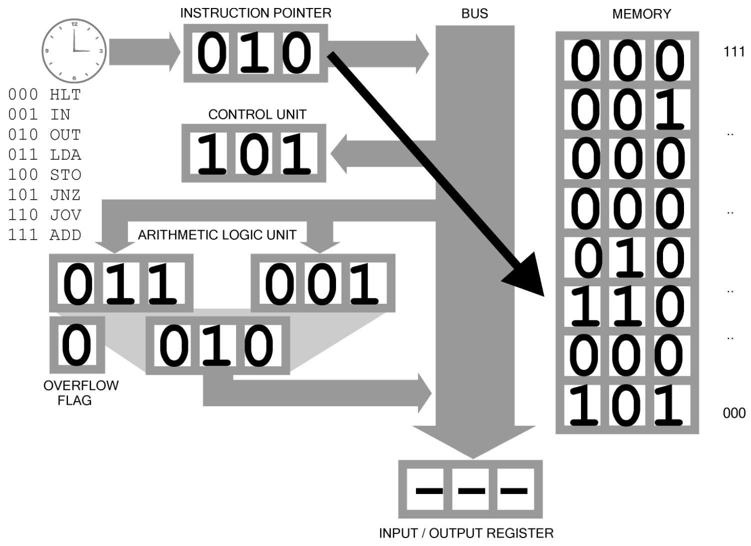
Now the Control Unit executes the HLT. NOTE: The lighter locations in Memory are never used, they are "Jumped" over. Sometimes this is shown as –

---  i.e. The contents are unimportant as they are never used.

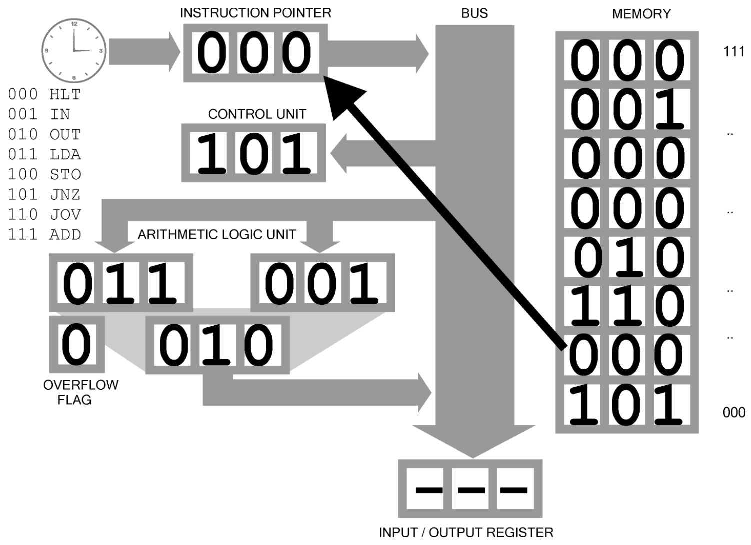
This shows how a program can continually loop for ever.



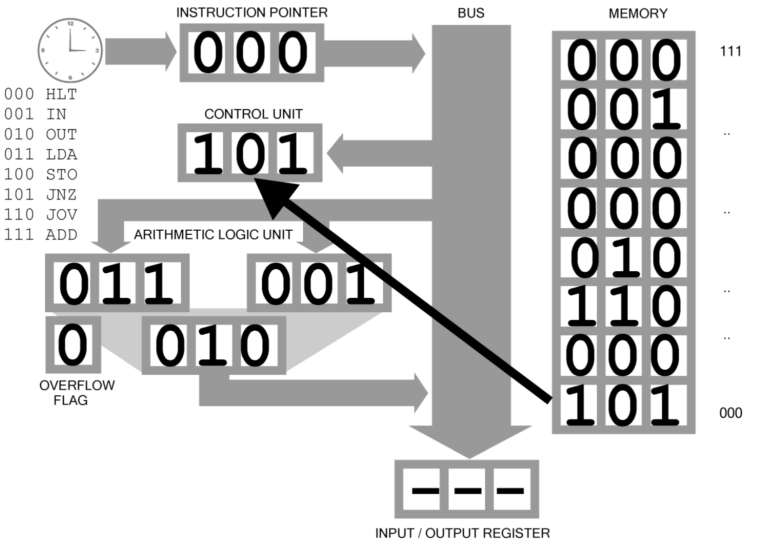
The ALU has been preset with data to show the looping.



The Instruction has been FETCHed and the Instruction Pointer Incremented to The next instruction @ 010.



When the instruction is EXECUTED the Instruction pointer is overwritten with the JUMP Address – 000 – The beginning of the program!



So the program will loop forever. Normally a loop is controlled by a number – loaded into the ALU and reduced by 1 each time. This is how the program can loop through a sequence a fixed number of times. Or it might loop until the overflow is set or some other condition. If the programmer and or logic creates a situation where the test is never met then a continuous and perpetual loop will occur. Famously described in Turing's Halting Problem. Without running all combinations of the program it is impossible to decide if it will run and halt or loop forever.

Program to compare input with memory, Loop if not zero otherwise halt.

This is the prototype of some kind of password or user login to a system, it shows how “decisions” are made in machine code. Our program will loop until the correct input and then stop. In actuality 3 tries might be given and far from halting other programs loaded for the user to interface with.

Basic Algorithm

1. Get Input
2. Load Accumulator with Stored Data (Stored Data is in 2s compliment form)
3. Add the stored data to the input
4. If not zero (input = “password”) jump to start
5. Otherwise HALT

```
000  INP          - input data
001  ADD 010      - add data pointed to at address 010
010  110          - 010 is data stored in 2s compliment
011  JNZ 111      - If Accumulator not zero jump to start
100  000          - Address of start of program
110  HLT          - End of program
110  110          - data “password in 2s compliment”
111  000          - Address for JUMP
```

Programmers sometimes “run” a program using pencil and paper to follow the logic. I effect “running” the program without a computer. Called “Dry Running”

Here all the possible outputs of the ALU’s ‘response’ to input are shown. Only the ‘correct’ input leaves the output of the ALU as zero.

Possible inputs are 000 001 010* 011 100 101 110 111
*correct password

```
000  Input
110  2s comp
--- +
110  Result
```

```
001  Input
110  2s comp
--- +
111  Result
```

```
010* Input
110  2s comp
--- +
000  Result
```

```
011  Input
110  2s comp
--- +
001  Result
```

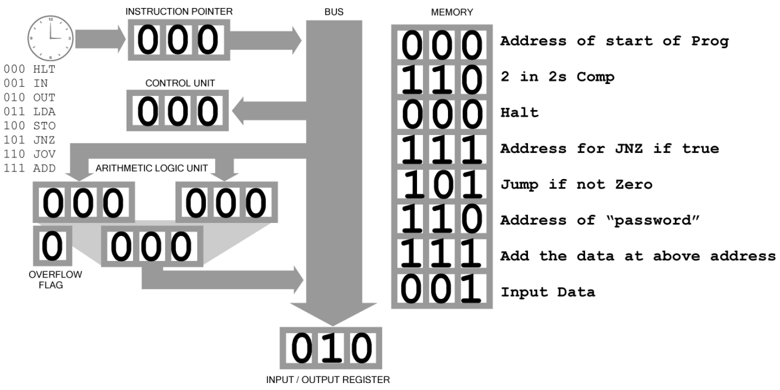
```
100  Input
110  2s comp
--- +
010  Result
```

```
101  Input
110  2s comp
--- +
011  Result
```

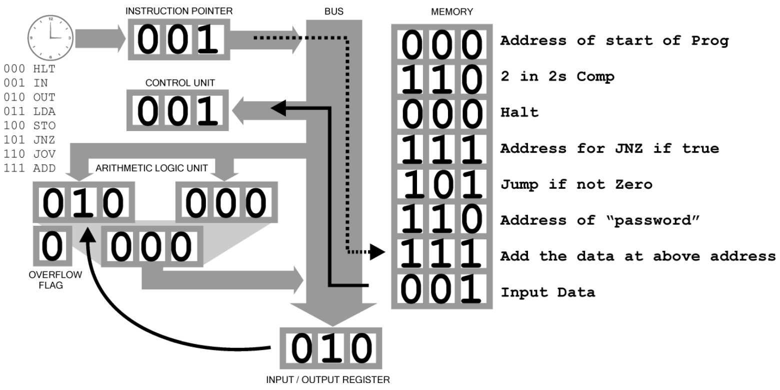
```
110  Input
110  2s comp
--- +
100  Result
```

```
111  Input
110  2s comp
--- +
001  Result
```

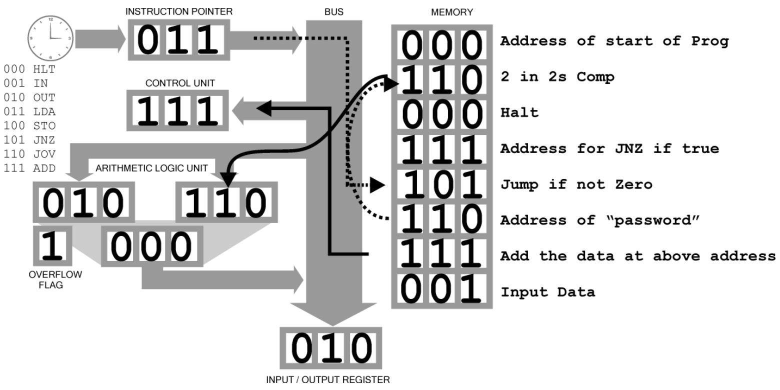
Carry flag not set in first two examples only.



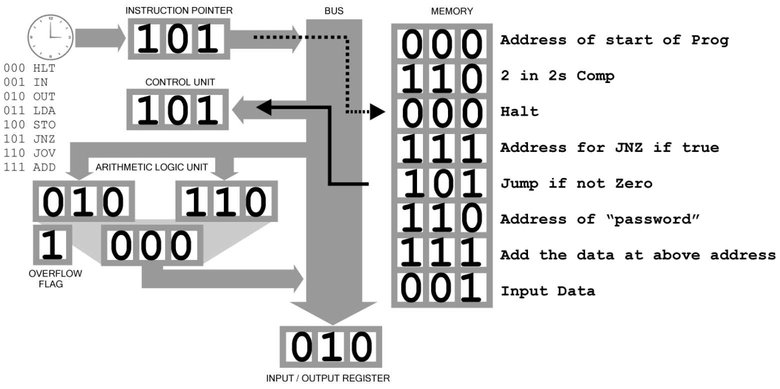
The program – here we will input 010, the correct password.



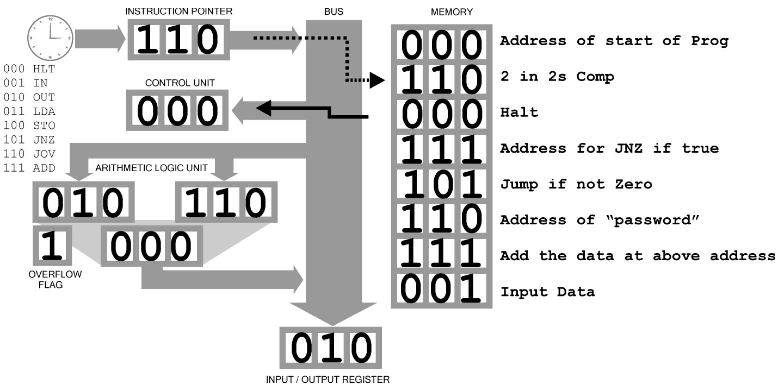
The DATA is input to the ALU.



The add instruction ADDS the data to the ALU from memory location 110. The data is 2 in 2s complement. The addition results in a Zero – the overflow is ignored.



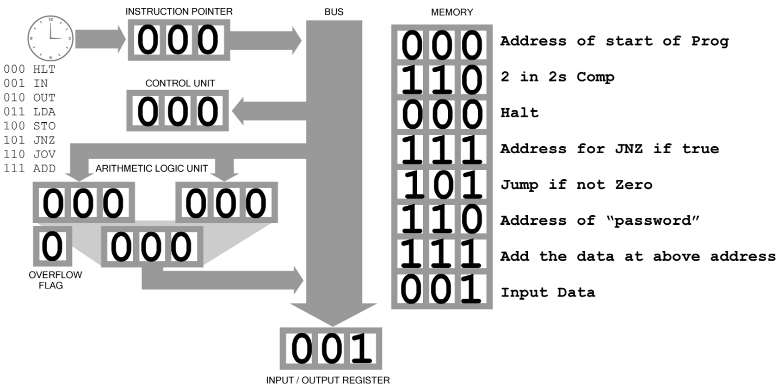
The JUMP IF NOT ZERO is executed – the Result is zero – NO Jump takes place – the Instruction Pointer is pointing to the next instruction at 101, which is HLT.



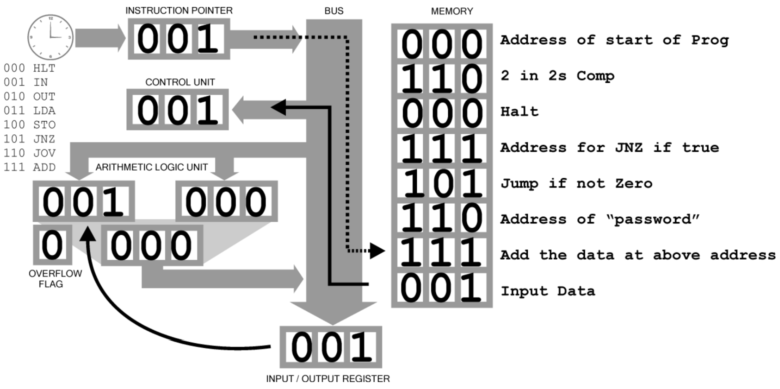
The HLT instruction is executed – the program stops. Notice the Instruction pointer is now pointing to 110 – where data is stored. This never gets executed as the HLT stops the program.

A common error is when data gets treated as code. Notice there is no way of telling code from data, or from an address. In 'real' systems such situations where data or code gets confused with addresses through some error an "illegal address" or "illegal instruction" can be created.

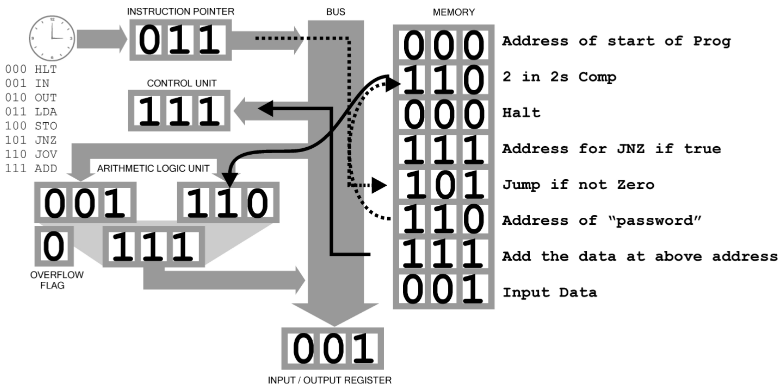
Philosophically this is a 'real' example of the arbitrariness of signs, and demonstrates that the meaning of a sign, at least binary digits is dependent on "context" and not some innate quality.



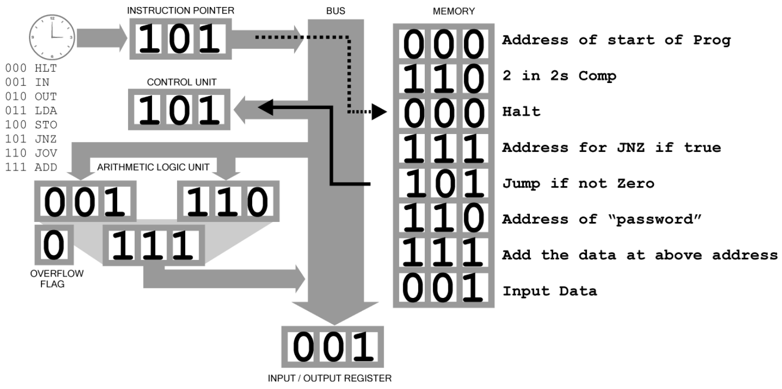
Now we will input 001. Not the correct "password".



As before the Data is loaded into the ALU.

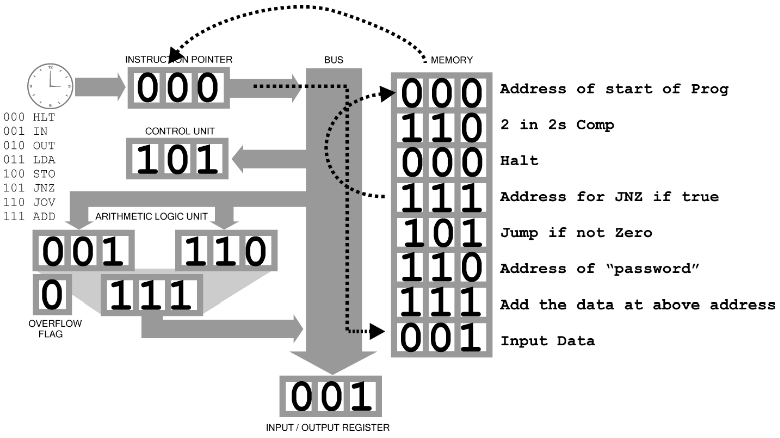


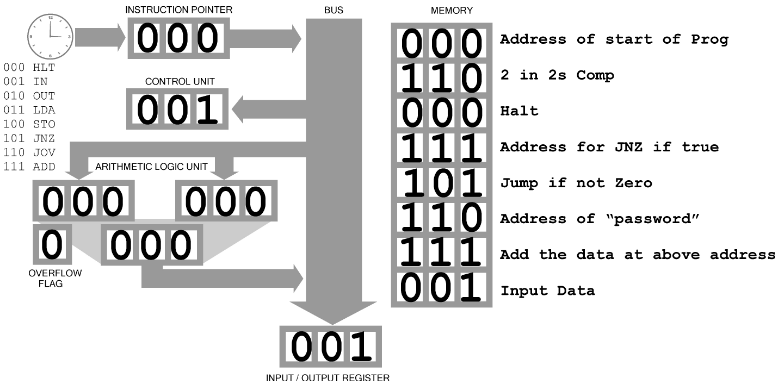
The Addition takes place resulting in 111.



The jump if not zero is executed. This time the IF statement is true. The Instruction pointer is pointing to the next instruction-

But on execution the Program Counter is overwritten by the address stored in location 111.





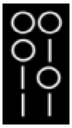
The program loops back to get another TRY.

This completes our overview of the simple CPU and its operations. Even with three bit data, address and instruction set as mentioned above there are a totality of over 16 million programs. Though this totality is fixed, and many of these will not be 'runable' programs they are not removed from our (possible) full and total knowledge of them.

As a possible alternative model, if not to an understanding or definitive knowledge it is surprising to experience the exponential nature of such simple binary systems. If we increase our "word" size from 3 to 4 bits our address space becomes 16 locations of 4 bits, or 64 bits. It follows that there are then in this case 2^{64} programs in total. 2^{64} is 18,446,744,073,709,551,616 or Eighteen quintillion, four hundred forty-six quadrillion, seven hundred forty-four trillion, seventy-three billion, seven hundred nine million, five hundred fifty-one thousand, six hundred and 16. An uncountable number and un-knowable number of fixed programs. Within the process of increasing sizes even more by only a few bits the limits of what is physically possible given the age and size of the universe as we know it is insufficient for a total knowledge of these computer models.

Computer scientists do not need to concern themselves with such possibilities, though they represent not a contingency but an actuality which is not experienceable to *us*. Here are limits like the dragons on old maps but these limits are manifestly real. These realms of higher bit sizes are those which we use in our daily interfaces with digital technology. They offer richer and seemingly actual objects as rich as those outside of the domain of bit strings - if such a domain is the one in which we find ourselves. They can and do allow for the richness and subtitles of images, sounds, languages. Of philosophies and metaphysical speculations in on-line blogs, pdf books and recorded seminars and lectures which postulate objects and our knowledge of them. Being a non-scientist and working from a simple 'Artistic' perceptive of representation, a dangerous position with such domains, the scope is beyond representation, even a 4 bit universe eludes any representation which is fully shown and known.

We can however move in an opposite direction. With 1 bit we can only have two states - 1 or 0. With 2 bits we can have 4 - 00,01,10,11. An address space of 4, each "word" now only 2* bits, we can even count the size, 8. 2^8 is 256. 256 possible programs and not so many more total states for such a machine universe of fixed states. So it is possible to map these "virtual" machine universes… my last thoughts here are such a mapping would be too naïve to be computer science. It's vocabulary appears insufficient for philosophy or metaphysics, it might be a simple ontology, a naïve ontology, but it would *not* be speculative or contingent but would be *actual*. Is the actualization of these simple ontologies possibly art? If not what is their ontological status - given their epistemological states are totally known?



**There are more than one possible such machine universe using only 2 bits but each is fixed and totalizable.*

PART 2 HAECCEITICS. THE AESTHETICS OF TOTALIZABLE OBJECTS.

The actualization of the aesthetic object as a totality.

A simpler CPU of only 2 bits gives us a 2 bit “word”, 4 possible instructions or operations, (op codes) and a memory space of 4 words. A total of 256 programs in all.

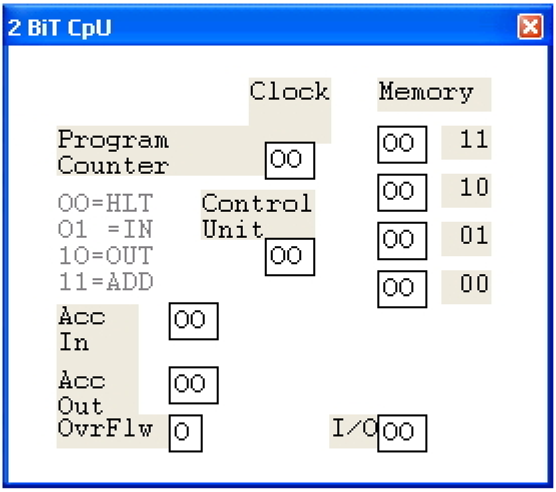
An even simpler instruction set.
one of many possible for a 2 bit CPU

00 = stop
01 = in – Load ALU (Accumulator) from input/output register
10 = out Display ALU (Accumulator) – on input/output register
11 = add the data following this instruction - the following 2 x 2 bits

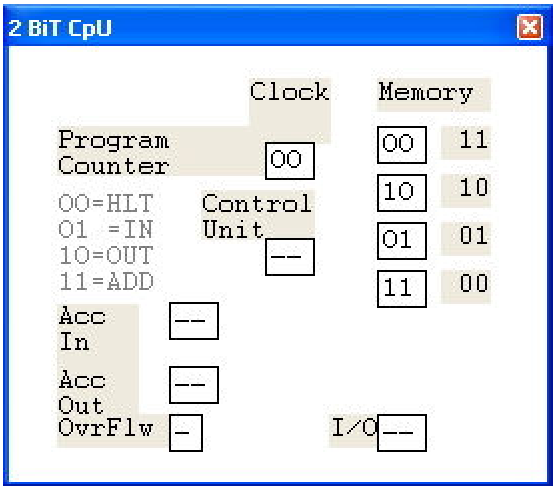
256 possible programs –

00000000	00110100	01101000	10011100	11010000
00000001	00110101	01101001	10011101	11010001
00000010	00110110	01101010	10011110	11010010
00000011	00110111	01101011	10011111	11010011
00000100	00111000	01101100	10100000	11010100
00000101	00111001	01101101	10100001	11010101
00000110	00111010	01101110	10100010	11010110
00000111	00111011	01101111	10100011	11010111
00001000	00111100	01110000	10100100	11011000
00001001	00111101	01110001	10100101	11011001
00001010	00111110	01110010	10100110	11011010
00001011	00111111	01110011	10100111	11011011
00001100	01000000	01110100	10101000	11011100
00001101	01000001	01110101	10101001	11011101
00001110	01000010	01110110	10101010	11011110
00001111	01000011	01110111	10101011	11011111
00010000	01000100	01111000	10101100	11100000
00010001	01000101	01111001	10101101	11100001
00010010	01000110	01111010	10101110	11100010
00010011	01000111	01111011	10101111	11100011
00010100	01001000	01111100	10110000	11100100
00010101	01001001	01111101	10110001	11100101
00010110	01001010	01111110	10110010	11100110
00010111	01001011	01111111	10110011	11100111
00011000	01001100	10000000	10110100	11101000
00011001	01001101	10000001	10110101	11101001
00011010	01001110	10000010	10110110	11101010
00011011	01001111	10000011	10110111	11101011
00011100	01010000	10000100	10111000	11101100
00011101	01010001	10000101	10111001	11101101
00011110	01010010	10000110	10111010	11101110
00011111	01010011	10000111	10111011	11101111
00100000	01010100	10001000	10111100	11110000
00100001	01010101	10001001	10111101	11110001
00100010	01010110	10001010	10111110	11110010
00100011	01010111	10001011	10111111	11110011
00100100	01011000	10001100	11000000	11110100
00100101	01011001	10001101	11000001	11110101
00100110	01011010	10001110	11000010	11110110
00100111	01011011	10001111	11000011	11110111
00101000	01011100	10010000	11000100	11111000
00101001	01011101	10010001	11000101	11111001
00101010	01011110	10010010	11000110	11111010
00101011	01011111	10010011	11000111	11111011
00101100	01100000	10010100	11001000	11111100
00101101	01100001	10010101	11001001	11111101
00101110	01100010	10010110	11001010	11111110
00101111	01100011	10010111	11001011	11111111
00110000	01100100	10011000	11001100	
00110001	01100101	10011001	11001101	
00110010	01100110	10011010	11001110	
00110011	01100111	10011011	11001111	

Such an totality of programs can be catalogued as to their actions given this instruction set. This can be achieved simply by working through each program ‘manually’ - sometimes called ‘dry running’, or a computer program can ‘model’ the actions of this hypothetical CPU. An emulation. Or one could be built from discrete hardware components.



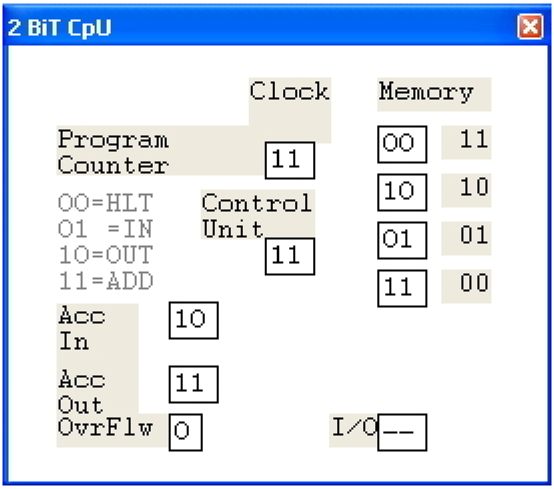
Here is a screen shot of an emulation of our 2 bit CPU. The primitive instruction set is shown, the 8 bits of memory, program counter, the ALU / Acc (Accumulator) with an overflow flag which is set when addition results in overflow. In this case any number higher than 3.



```
2 BiT CpU

                         Clock     Memory
Program                            00   11
Counter                 00         10   10
00=HLT       Control               01   01
01  =IN      Unit                  11   00
10=OUT                  --
11=ADD
Acc         --
In
Acc         --
Out
OvrFlw      -                I/O --
```

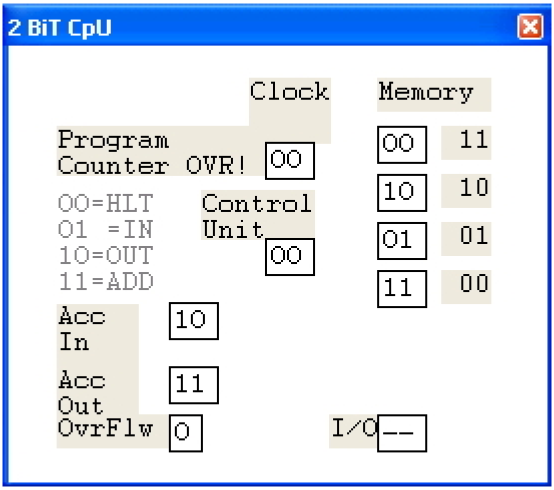
Above a program is loaded into memory which will Add two numbers (01 + 10) then HALT. The “- -“ indicates the contents are not known, some CPUs will clear registers and memories. Before running…



```
2 BiT CpU

                         Clock     Memory
Program                            00   11
Counter                 11         10   10
00=HLT       Control               01   01
01  =IN      Unit                  11   00
10=OUT                  11
11=ADD
Acc         10
In
Acc         11
Out
OvrFlw      0                I/O --
```

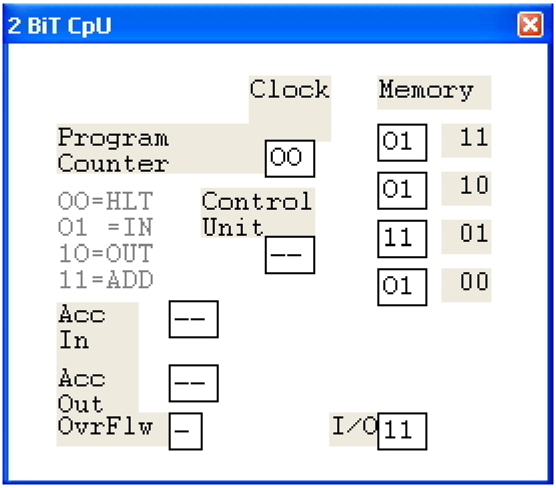
The Control unit executes the ADD instruction. 01 + 10 = 11.
(1 + 2 = 3) The Program Counter is pointing to the next instruction.



The Control Unit has the instruction @ 11 loaded which is 00 - the Halt instruction. Note the Program Counter has overflowed. As the Program Counter is incremented BEFORE the instruction is carried out. This doesn't matter as the HLT stops the program. If it did not the Program counter is now pointing to address 00 so the program would run again, and forever.

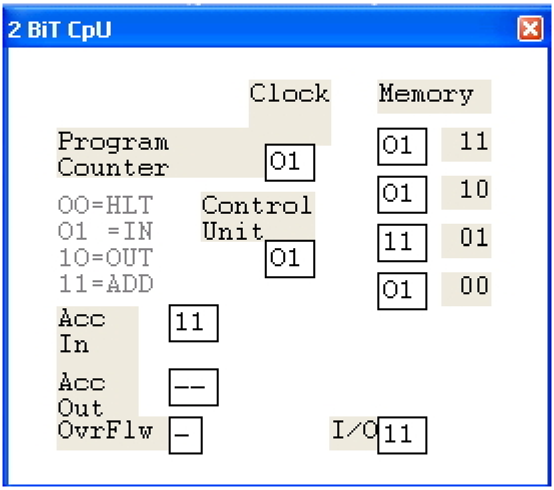
So even such a simple CPU and Instruction set can perform arithmetic, and the outcome of a program might be that it HALTs or it could run forever. So this duplicates - demonstrates Turing's Halting Problem.

This program "00100111" is the # 40 in the list of all possible programs given above.



This program demonstrates the continuous looping performed by not Halting program execution.

The program will input the Data in the I/O register, then Add location 10 to location 11 storing the answer in the Acc Output. The output of the ALU. This will overwrite the input from the I/O register. (The I/O functions in such a simple model have little functionality..) As the Program counter before the ADD instruction is completed will be incremented by 1 from 11, it will overflow, leaving 00 in the Program counter. As this model CPU does not take any action when an overflow occurs it will resume processing @ address 00, and so repeat the program without ever halting.



2 BiT CpU

Clock

Memory

01	11
01	10
11	01
01	00

Program Counter 01

00=HLT
01 =IN
10=OUT
11=ADD

Control Unit 01

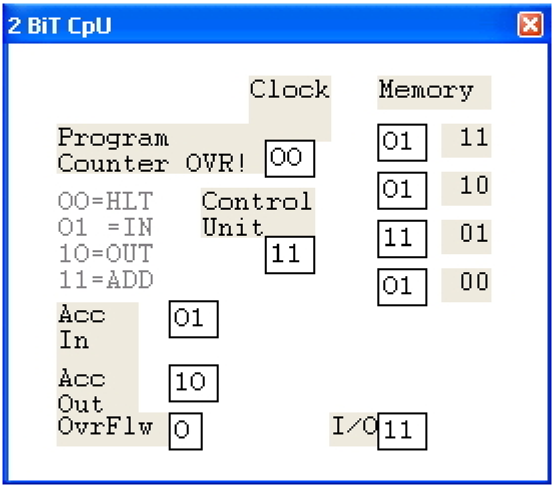
Acc In 11

Acc Out --

OvrFlw -

I/O 11

The program copies the I/O register into the ALU. The input of the Accumulator.



2 BiT CpU

Clock

Memory

01	11
01	10
11	01
01	00

Program Counter OVR! 00

00=HLT
01 =IN
10=OUT
11=ADD

Control Unit 11

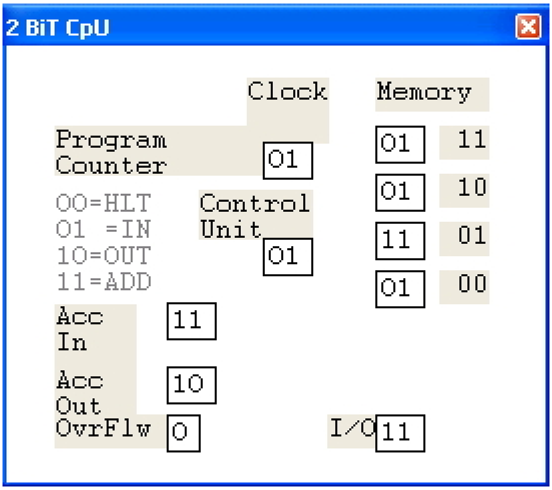
Acc In 01

Acc Out 10

OvrFlw 0

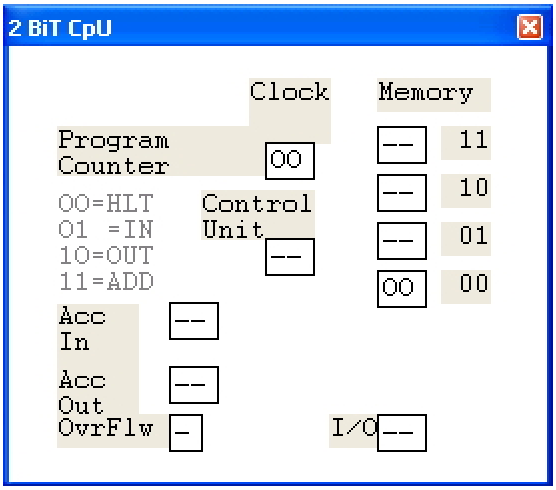
I/O 11

The Acc Out shows the correct answer. The Program Counter has overflowed and now points to address 00.

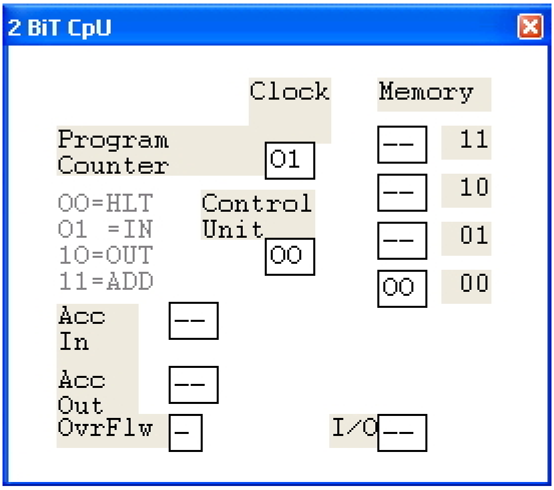


```
00=HLT
01 =IN
10=OUT
11=ADD
```

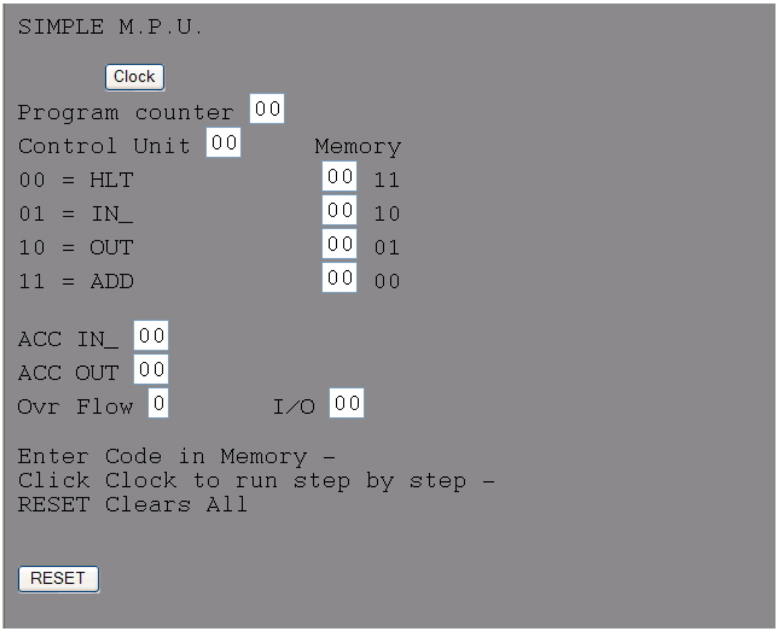
The program loops - re inputs the I/O - will then add the two numbers… etc.



This program halts immediately as do all programs which begin with 00.
There are 64 out of the 256 programs which do this.



An online version of this emulator is available here WWW.JLIAT.COM/SMPU



```
SIMPLE M.P.U.

        Clock

Program counter 00
Control Unit 00       Memory
00 = HLT                 00 11
01 = IN_                 00 10
10 = OUT                 00 01
11 = ADD                 00 00

ACC IN_  00
ACC OUT  00
Ovr Flow 0        I/O 00

Enter Code in Memory -
Click Clock to run step by step -
RESET Clears All

RESET
```

Any program which begins with 00 will immediately HALT. Therefore these will not need to be examined further to see if they will not do so. It might be thought that any program not ending with 00 will not halt, but this is not the case. A full examination of all these programs follows.

```
Instruction set  Number # 1
Acc = Accumulator

00 = stop
01 = in - lda with input
10 = out  disp acc
11 = add  dta following 2 x 2bits
```

```
256 possible programs -
+---+--+--+--+--+------------------------------------------------------------+
! # !Program    ! Action
+---+--+--+--+--+------------------------------------------------------------+
!  1!00 00 00 00! Program Immediately Halts
+---+--+--+--+--+------------------------------------------------------------+
!  2!00 00 00 01! Loads Acc with Data (00,01,10,11)then Halts
+---+--+--+--+--+------------------------------------------------------------+
!  3!00 00 00 10! Output Acc, then Halts
+---+--+--+--+--+------------------------------------------------------------+
!  4!00 00 00 11! Adds 00 to 00 then Halts
+---+--+--+--+--+------------------------------------------------------------+
!  5!00 00 01 00! Program Immediately Halts
+---+--+--+--+--+------------------------------------------------------------+
!  6!00 00 01 01! Loads Acc with Data - twice - & then Halts
+---+--+--+--+--+------------------------------------------------------------+
!  7!00 00 01 10! Output, Acc Load Acc with Data then Halts
+---+--+--+--+--+------------------------------------------------------------+
!  8!00 00 01 11! Adds 01 to 00,  then Halts
+---+--+--+--+--+------------------------------------------------------------+
!  9!00 00 10 00! Program Immediately Halts
+---+--+--+--+--+------------------------------------------------------------+
! 10!00 00 10 01! Load Acc with Data, then outputs this Data, then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 11!00 00 10 10! Outputs Acc x 2, then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 12!00 00 10 11! Adds 10+0 then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 13!00 00 11 00! Program Immediately Halts
+---+--+--+--+--+------------------------------------------------------------+
! 14!00 00 11 01! Inputs Data to Acc, adds 00 to 00, PC increments to 100 ovrflw
                  Gets 01 @ 00 inputs data.. endless loop
+---+--+--+--+--+------------------------------------------------------------+
! 15!00 00 11 10! Outputs Acc, adds 0+0, PC increments to 100 ovrflw
                  Gets 10 @ 00 inputs data.. endless loop
+---+--+--+--+--+------------------------------------------------------------+
! 16!00 00 11 11! Adds 11+0, then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 17!00 01 00 00! Program Immediately Halts
+---+--+--+--+--+------------------------------------------------------------+
! 18!00 01 00 01! Lds Acc then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 19!00 01 00 10! Outputs Acc then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 20!00 01 00 11! Adds 0+1, then Halts
+---+--+--+--+--+------------------------------------------------------------+
! 21!00 01 01 00! Program Immediately Halts
+---+--+--+--+--+------------------------------------------------------------+
! 22!00 01 01 01! Lds Acc x 3 then Halts
```

```
+---+--+--+--+--------------------------------------------------------------+
! 23!00 01 01 10! Output Acc, Lds Acc x 2 then Halts
+---+--+--+--+--------------------------------------------------------------+
! 24!00 01 01 11! Adds 1+1 then Halts
+---+--+--+--+--------------------------------------------------------------+
! 25!00 01 10 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 26!00 01 10 01! Lds Acc, Output Acc, Lds Acc then Halts
+---+--+--+--+--------------------------------------------------------------+
! 27!00 01 10 10! Output Acc x 2, Lds Acc then Halts
+---+--+--+--+--------------------------------------------------------------+
! 28!00 01 10 11! Adds 10+1 then Halts
+---+--+--+--+--------------------------------------------------------------+
! 29!00 01 11 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 30!00 01 11 01! Lds Acc, Adds 1+0 then endless loop
+---+--+--+--+--------------------------------------------------------------+
! 31!00 01 11 10! Output Acc Adds 1+0 then endless loop
+---+--+--+--+--------------------------------------------------------------+
! 32!00 01 11 11! Adds 11+01 then Halts
+---+--+--+--+--------------------------------------------------------------+
! 33!00 10 00 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 34!00 10 00 01! Lds Acc then Halts
+---+--+--+--+--------------------------------------------------------------+
! 35!00 10 00 10! Output Acc, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 36!00 10 00 11! Adds 0+10 then Halts
+---+--+--+--+--------------------------------------------------------------+
! 37!00 10 01 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 38!00 10 01 01! Lds Acc x 2, Output Acc the Halts
+---+--+--+--+--------------------------------------------------------------+
! 39!00 10 01 10! Output Acc, Lds Acc, Output Acc, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 40!00 10 01 11! Adds 1+10, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 41!00 10 10 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 42!00 10 10 01! Lds Acc, Output Acc x 2, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 43!00 10 10 10! Output Acc x 3, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 44!00 10 10 11! Adds 10+10, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 45!00 10 11 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 46!00 10 11 01! Lds Acc Adds 10+0, then endless loop
+---+--+--+--+--------------------------------------------------------------+
! 47!00 10 11 10! Output Acc, Adds 10+0, then endless loop
+---+--+--+--+--------------------------------------------------------------+
! 48!00 10 11 11! Adds 11+10, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 49!00 11 00 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 50!00 11 00 01! Lds Acc, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 51!00 11 00 10! Output Acc, Then Halts
+---+--+--+--+--------------------------------------------------------------+
! 52!00 11 00 11! Adds 0+11, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 53!00 11 01 00! Program Immediately Halts
+---+--+--+--+--------------------------------------------------------------+
! 54!00 11 01 01! Lds Acc x 2, Adds 0+1, Lds acc Adds 0+1.. endless loop
+---+--+--+--+--------------------------------------------------------------+
! 55!00 11 01 10! Output Acc, Lds Acc, Adds 0+10, Lds Acc.. endless loop
+---+--+--+--+--------------------------------------------------------------+
! 56!00 11 01 11! Adds 1+11, then Halts
+---+--+--+--+--------------------------------------------------------------+
! 57!00 11 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 58!00 11 10 01! Lds Acc, Output Acc, Adds 0+1,Output Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 59!00 11 10 10! Output Acc x 2, Adds 0+10, Outputs Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 60!00 11 10 11! Adds 10+11, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 61!00 11 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 62!00 11 11 01! Lds Acc, Adds 11+0, Lds acc, Adds 11+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 63!00 11 11 10! Output Acc, Adds 11+0, Outputs Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 64!00 11 11 11! Adds 11+11, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 65!01 00 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 66!01 00 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 67!01 00 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 68!01 00 00 11! Adds 0+0, Lds Acc, Adds 0+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 69!01 00 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 70!01 00 01 01! Lds Acc x 2, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 71!01 00 01 10! Output Acc, Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 72!01 00 01 11! Adds 1+0, Lds Acc, Adds 1+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 73!01 00 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 74!01 00 10 01! Lds Acc, Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 75!01 00 10 10! Output Acc x 2, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 76!01 00 10 11! Adds 10+0, Lds Acc, ! Adds 10+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 77!01 00 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 78!01 00 11 01! Lds Acc, Adds 0+1, Lds Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 79!01 00 11 10! Output Acc, Adds 0+1, Output Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 80!01 00 11 11! Adds 11+0, Lds Acc, Adds 11+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 81!01 01 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 82!01 01 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 83!01 01 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 84!01 01 00 11! Adds 0+1, Lds Acc, Adds 00+1.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 85!01 01 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 86!01 01 01 01! Lds Acc endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 87!01 01 01 10! Output Acc, Lds Acc x 3.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 88!01 01 01 11! Adds 1+1, Lds Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 89!01 01 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
! 90!01 01 10 01! Lds Acc, Output Acc, Lds Acc x 2.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
! 91!01 01 10 10! Output Acc x 2, Lds Acc x 2.. endless loop
```

```
+---+--+--+--+---------------------------------------------------------------+
! 92!01 01 10 11! Adds 10+1, Lds Acc, Adds 10+1.. endless loop
+---+--+--+--+---------------------------------------------------------------+
! 93!01 01 11 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
! 94!01 01 11 01! Lds Acc, Adds 1+1, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
! 95!01 01 11 10! Output Acc, Adds 1+1, Output Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
! 96!01 01 11 11! Adds 11+1, Lds Acc, Adds 11+1.. endless loop
+---+--+--+--+---------------------------------------------------------------+
! 97!01 10 00 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
! 98!01 10 00 01! Lds Acc, then Halts
+---+--+--+--+---------------------------------------------------------------+
! 99!01 10 00 10! Output Acc, then Halts
+---+--+--+--+---------------------------------------------------------------+
!100!01 10 00 11! Adds 0+10, Lds Acc, Add 0+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!101!01 10 01 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!102!01 10 01 01! Lds Acc x 2, Output Acc, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!103!01 10 01 10! Output Acc, Lds Acc, Output Acc, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!104!01 10 01 11! Adds 1+10, Lds Acc, Adds 1+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!105!01 10 10 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!106!01 10 10 01! Lds Acc, Output Acc x 2, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!107!01 10 10 10! Output Acc x 3, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!108!01 10 10 11! Adds 10+10, Lds Acc, Adds 10+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!109!01 10 11 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!110!01 10 11 01! Lds Acc, Adds 10+1, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!111!01 10 11 10! Output Acc, Adds 10+1, Output Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!112!01 10 11 11! Adds 11+10, Lds Acc, Adds 11+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!113!01 11 00 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!114!01 11 00 01! Lds Acc, then Halts
+---+--+--+--+---------------------------------------------------------------+
!115!01 11 00 10! Output Acc, then Halts
+---+--+--+--+---------------------------------------------------------------+
!116!01 11 00 11! Adds 0+11, Lds Acc, Adds 0+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!117!01 11 01 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!118!01 11 01 01! Lds Acc x 2, Adds 1+1, Lds Acc, Adds 1+1.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!119!01 11 01 10! Output Acc, Lds Acc,  Adds 1+10, Lds Acc.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!120!01 11 01 11! Adds 1+11, Lds Acc, Adds 1+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!121!01 11 10 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!122!01 11 10 01! Lds Acc, Output Acc, Adds 1+1, Output Acc, Adds.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!123!01 11 10 10! Output Acc x 2, Adds 1+10, Output Acc, Adds.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!124!01 11 10 11! Adds 10+11, Lds Acc, Adds 10+10.. endless loop
+---+--+--+--+---------------------------------------------------------------+
!125!01 11 11 00! Program Immediately Halts
+---+--+--+--+---------------------------------------------------------------+
!126!01 11 11 01! Lds Acc, Adds 11+1, Lds Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!127!01 11 11 10! Output Acc, Adds 11+11, Output Acc, Output Acc, Adds.. endless
loop
+---+--+--+--+--+-----------------------------------------------------------+
!128!01 11 11 11! Adds 11+11, Lds Acc, Adds 11+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!129!10 00 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!130!10 00 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!131!10 00 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!132!10 00 00 11! Adds 0+0, Output Acc, Adds 0+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!133!10 00 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!134!10 00 01 01! Lds Acc x 2, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!135!10 00 01 10! Output Acc, Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!136!10 00 01 11! Adds 1+0, Output Acc, Adds 1+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!137!10 00 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!138!10 00 10 01! Lds Acc, Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!139!10 00 10 10! Output Acc x 2, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!140!10 00 10 11! Add 10+0, Output Acc, Add 10+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!141!10 00 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!142!10 00 11 01! Lds Acc, Add 0+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!143!10 00 11 10! Output Acc, Adds 0+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!144!10 00 11 11! Adds 11+0, Output Acc, Adds 11+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!145!10 01 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!146!10 01 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!147!10 01 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!148!10 01 00 11! Adds 11+0, Output Acc, Adds 11+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!149!10 01 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!150!10 01 01 01! Lds Acc x 3, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!151!10 01 01 10! Output Acc, Lds Acc x 2, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!152!10 01 01 11! Adds 1+1, Output Acc, Adds 1+1.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!153!10 01 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!154!10 01 10 01! Lds Acc, Output Acc, Lds Acc, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!155!10 01 10 10! Output Acc x 2, Lds Acc, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!156!10 01 10 11! Adds 10+1, Output Acc, Adds 10+1.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!157!10 01 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!158!10 01 11 01! Lds Acc, Adds 1+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!159!10 01 11 10! Output Acc, Adds 1+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!160!10 01 11 11! Adds 11+1, Output Acc, Adds 11+1.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!161!10 10 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!162!10 10 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!163!10 10 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!164!10 10 00 11! Adds 11+0, Output Acc, Adds 11+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!165!10 10 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!166!10 10 01 01! Lds Acc x 2, Output acc x 2.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!167!10 10 01 10! Output Acc, Lds Acc, Output Acc x2.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!168!10 10 01 11! Adds 11+0, Output Acc, Adds 11+0.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!169!10 10 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!170!10 10 10 01! Lds Acc, Output Acc x3.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!171!10 10 10 10! Output Acc x4.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!172!10 10 10 11! Adds 10+10, Output Acc, Adds 10+10.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!173!10 10 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!174!10 10 11 01! Lds Acc, Adds 10+10, Lds Acc, Adds.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!175!10 10 11 10! Output Acc, Adds 10+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!176!10 10 11 11! Adds 11+10, Output Acc, Adds 11+10.. Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!177!10 11 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!178!10 11 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!179!10 11 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!180!10 11 00 11! Adds 0+11, Output Acc, Adds 0+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!181!10 11 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!182!10 11 01 01! Lds Acc x 2, Adds 10+1, Lds Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!183!10 11 01 10! Output Acc, Lds Acc, Adds 10+10, Lds Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!184!10 11 01 11! Adds 1+11, Output Acc, Adds 1+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!185!10 11 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!186!10 11 10 01! Lds Acc, Output Acc, Adds 10+1, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!187!10 11 10 10! Output Acc x 2, Adds 10+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!188!10 11 10 11! Adds 10+11, Output Acc, Adds 10+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!189!10 11 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!190!10 11 11 01! Lds Acc, Adds 11+10, Lds Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!191!10 11 11 10! Output Acc, Adds 11+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!192!10 11 11 11! Adds 11+11, Output Acc, Adds 11+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!193!11 00 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!194!11 00 00 01! Lds Acc, then Halts
```

```
+---+--+--+--+--+-----------------------------------------------------------+
!195!11 00 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!196!11 00 00 11! Adds 0+0, Adds 11+0, then Halts (2 passes through code)
+---+--+--+--+--+-----------------------------------------------------------+
!197!11 00 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!198!11 00 01 01! Lds Acc x 2, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!199!11 00 01 10! Output Acc, Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!200!11 00 01 11! Adds 1+0, Adds,  Adds 11+1 then Halts (2 passes through code)
+---+--+--+--+--+-----------------------------------------------------------+
!201!11 00 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!202!11 00 10 01! Lds Acc, Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!203!11 00 10 10! Output Acc x 2, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!204!11 00 10 11! Adds 10+0, Adds 11+10, then Halts (2 passes through code)
+---+--+--+--+--+-----------------------------------------------------------+
!205!11 00 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!206!11 00 11 01! Lds Acc, Adds 0+11, Lds Acc, Adds 0+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!207!11 00 11 10! Output Acc, 0+11, Output Acc, Adds 0+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!208!11 00 11 11! Adds 11+0, Adds 11+11, then Halts (2 passes through code)
+---+--+--+--+--+-----------------------------------------------------------+
!209!11 01 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!210!11 01 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!211!11 01 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!212!11 01 00 11! Adds 0+1, Adds 11+0, Lds Acc, Adds 11+0  .. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!213!11 01 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!214!11 01 01 01! Lds Acc x 3, Adds 1+1, Lds Acc, Adds 1+1.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!215!11 01 01 10! Output Acc, Lds Acc x 2, Adds 10+1, Lds Acc, Adds 10+1..
endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!216!11 01 01 11! Adds 1+1, Adds 11+1, Lds Acc, Adds 11+1.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!217!11 01 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!218!11 01 10 01! Lds Acc, Output Acc, Adds 1+10, Lds Acc, Adds 1+10.. endless
loop
+---+--+--+--+--+-----------------------------------------------------------+
!219!11 01 10 10! Output Acc x 2, Lds Acc, Adds 10+10.. Lds Acc endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!220!11 01 10 11! Adds 10+1, Adds 11+10, Lds Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!221!11 01 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!222!11 01 11 01! Lds Acc, Adds 1+11, Lds Acc, Adds 1+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!223!11 01 11 10! Output Acc, Adds 1+11, Output Acc, Adds 1+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!224!11 01 11 11! Adds 11+1, Adds 11+11, Lds Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!225!11 10 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!226!11 10 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!227!11 10 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!228!11 10 00 11! Adds 0+10, Adds 11+0, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!229!11 10 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!230!11 10 01 01! Lds Acc x 2, Output Acc, Adds 1+1, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!231!11 10 01 10! Output Acc, Lds Acc, Adds 10+1, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!232!11 10 01 11! Adds 1+10, Adds 11+1, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!233!11 10 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!234!11 10 10 01! Lds Acc, Output Acc x 2, Adds 01+10.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!235!11 10 10 10! Output Acc x 3, Adds 10+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!236!11 10 10 11! Adds 10+10, Adds 11+10, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!237!11 10 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!238!11 10 11 01! Lds Acc, Adds 10 + 11, Lds Acc, Adds 10 + 11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!239!11 10 11 10! Output Acc, Adds 10 + 11, Output Acc, Adds 10 + 11.. endless
loop
+---+--+--+--+--+-----------------------------------------------------------+
!240!11 10 11 11! Adds 11+10, Adds 11+11, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!241!11 11 00 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!242!11 11 00 01! Lds Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!243!11 11 00 10! Output Acc, then Halts
+---+--+--+--+--+-----------------------------------------------------------+
!244!11 11 00 11! Adds 0+11, Adds 11+0, Adds 11+11 then Halts (2 passes through
code)
+---+--+--+--+--+-----------------------------------------------------------+
!245!11 11 01 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!246!11 11 01 01! Lds Acc x 2, Adds 11 + 01, Lds acc, Adds 11 + 01.. endless
loop
+---+--+--+--+--+-----------------------------------------------------------+
!247!11 11 01 10! Output Acc, Lds Acc, Adds 11 + 10, Lds Acc, Adds 11 + 1..
endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!248!11 11 01 11! Adds 1+11, Adds 11+11, Lds Acc, Adds 11+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!249!11 11 10 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!250!11 11 10 01! Lds Acc, Output Acc, Adds 11+1, Output Acc, Adds 11+1..
endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!251!11 11 10 10! Output Acc x 2, Adds 11+10, Output Acc, Adds 11+10.. endless
loop
+---+--+--+--+--+-----------------------------------------------------------+
!252!11 11 10 11! Adds 10+11, Adds 11+10, Adds 11+11, Output Acc.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!253!11 11 11 00! Program Immediately Halts
+---+--+--+--+--+-----------------------------------------------------------+
!254!11 11 11 01! Lds Acc, Adds 11+11, Lds Acc, Adds 11+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!255!11 11 11 10! Output Acc, Adds 11+11, Output Acc, Adds 11+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
!256!11 11 11 11! Adds 11+11, Adds 11+11, Adds 11+11.. endless loop
+---+--+--+--+--+-----------------------------------------------------------+
```

Each of these programs with the total set of 256 2 bit programs can be considered as a fully 'knowable' / 'experienceable' object. Within the class of the 256 are shared characteristics, halting non halting for instance. Some programs exhibit 'reuse' of data as instructions. Others exhibit symmetries of Input / process etc. It is possible therefore not only to know or experience these objects, it would and is also possible to construct further classifications. For instance each program could be thought of as a 'species' within the 'Class'. Halting / non halting could be a 'Order', and Families and Genus' further identified. The origin and cause of these taxonomic properties is also interesting. Obviously these taxonomies are derived not from any developmental process but from a set which is apriori in existence given the address space.

Philosophically these objects are fully knowable in themselves. And so it is possible, or even preferable to not use a critical / epistemological approach, (Post Kantian), but apply a dogmatic metaphysics of pre-Kantian Scholasticism.

The essence of these objects is knowable – the Haecceity, the particular 'thisness' of a definite program (object), and its quiddity or hypokeimenon, those properties it shares with others in the set. These object's haecceity is clear in that they are 'individuals' – no two are identical, yet they share commonalities. In such a simple set, a 2 bit universe, these distinctions are obvious, trivial, yet *real*. What causes both an object's "thisness" from its quiddity are given without recourse to epistemology, as they are given apriori any thought. Their ontology is also apriori given. Here a philosophic objection might be that these are not (philosophically) appropriate objects. They may well not be appropriate to 'real world' philosophy any more than they are so naive as to not be 'appropriate' to 'real world' computer science. Yet as a set of 'aesthetic' objects, a critique- knowledge – phenomenology, presentation .. in these terms, can be made – 'aesthetically', or rather than a critique a dogmatic metaphysics can be made.

"Locke theorised that when all sensible properties were abstracted away from an object, such as its colour, weight, density or taste, there would still be something left to which the properties had adhered— something which allowed the object to exist independently of the sensible properties that it manifested in the beholder. Locke saw this ontological ingredient as necessary if we are to be able to consider objects as existing independently of our own minds. The material substratum proved a difficult idea for Locke as by its very nature its existence could not be directly proved in the manner endorsed by empiricists (i.e., proof by exhibition in experience). Nevertheless, he believed that the philosophical reasons for it were strong enough for its existence to be considered proved.

The existence of the substratum was denied by Berkeley. In his Three Dialogues Between Hylas and Philonous, Berkeley maintained that an object consists of nothing more than those sensible properties (or possible sensible properties) that the object manifests, and that those sensible properties only exist so long as the act of perceiving them does."

(http://en.wikipedia.org/w/index.php?title=Hypokeimenon&oldid=509912150)

It appears that whilst these objects can be empirically experienced, their ontology, and their Haecceity, quiddity, hypokeimenon, exist apriori.

If these 'universes' are to be considered as metaphors of larger universes (our own included) as any experiential knowledge of these larger universes for us is impossible, by virtue of scale, such access to these must become speculative, contingent, and provisional, even given the idea of the limited universes metaphorical use. Even a 3 bit universe is fairly difficult for us to know experientially due to the limitations of time, so such larger universes require 'speculation' – universes less than three bits do not.

A Dogmatic Metaphysics, *for us*, now seems possible in a 2 bit universe, (an odd feeling being able to think in a pre-Kantian way).. even if these objects are trivial and of no direct consequence or use in our everyday lives in larger universes, they represent a play of 2 bit characters, and as such we can watch their totality of performances. This offers a ground for doing a metaphysics which echoes Heidegger's notion of cybernetics – albeit very naive cybernetics allowing, (he says 'replacing') metaphysics.

Notably this instruction set does not use input or calculation which is then placed in memory. Input of data (via an I/O device) though not 'uncertain' will if stored in memory effectively change the program as it runs. This simple instruction set does not do this so its determinacy is easier to see. However some programs exhibit pseudo morphing - in that code which is used as data, is on a second or third loop used as an instruction. In these cases the Halt instruction terminates the program on the second and third loops through the address space.

```
196 11000011  loop ctr 1   Halts___ Executes Data
200 11000111  loop ctr 1   Halts___ Executes Data
204 11001011  loop ctr 1   Halts___ Executes Data
208 11001111  loop ctr 1   Halts___ Executes Data
244 11110011  loop ctr 2   Halts___ Executes Data
```

Such techniques are not uncommon in low level programming, though here there is no 'programmer' - it is a feature of a selection of the totality. In our first example of a simple 3 bit CPU input could alter program execution dynamically, however it would only alter it to one of the (determined) 16 million states.

At each instruction where an input occurs in the 2 bit CPU, there will be 4 possible inputs, 00, 01, 10, 11. Where two inputs occur 16 possible inputs, three inputs - 64 and four - 256. These represent additional states, however they do not alter memory, so the given outcomes above remain true and fixed. If we map the registers as well as memory we have another 12 memory locations. The PC, Control Unit, ACC in / out, The I/O reg and 2 flags. 2^{12} gives all of these states. 4096 states. If we map all of these with memory we have 2^{20} or 1,048,576 states. We might want to then consider the control lines from the Control unit and its gates, these are not shown in our theoretical model but would need to be implemented in an actual physical device. They are emulated programmatically in the emulator.

Over a million states would be time consuming to manually dry run but in principle these could be listed via running the emulator with all 256 programs with all possible inputs where they occur, and printing these out. In fact the possible states using this instruction set is less than 2^{20} as there are 256 programs there will be 256 states for the registers with the exception of the I/O register which will have 4, 16, 64 or 256 states depending on how many Input instructions occur in a given program. There are 512 in total so a maximum of 10,000 states for this instruction set.

```
Total States for all programs with ALL I/O operations

Program i/o #       Program i/o #       Program i/o #       Program i/o #
OO OO OO OO 0 1       Ol OO OO OO 1 4       lO OO OO OO 1 4       ll OO OO OO 0 1
OO OO OO Ol 1 4       Ol OO OO Ol 2 16      lO OO OO Ol 2 16      ll OO OO Ol 1 4
OO OO OO lO 1 4       Ol OO OO lO 2 16      lO OO OO lO 2 16      ll OO OO lO 1 4
OO OO OO ll 0 1       Ol OO OO ll 1 4       lO OO OO ll 1 4       ll OO OO ll 0 1
OO OO Ol OO 1 4       Ol OO Ol OO 2 16      lO OO Ol OO 2 16      ll OO Ol OO 1 4
OO OO Ol Ol 2 16      Ol OO Ol Ol 3 64      lO OO Ol Ol 3 64      ll OO Ol Ol 2 16
OO OO Ol lO 2 16      Ol OO Ol lO 3 64      lO OO Ol lO 3 64      ll OO Ol lO 2 16
OO OO Ol ll 1 4       Ol OO Ol ll 2 16      lO OO Ol ll 2 16      ll OO Ol ll 1 4
OO OO lO OO 1 4       Ol OO lO OO 2 16      lO OO lO OO 2 16      ll OO lO OO 1 4
OO OO lO Ol 2 16      Ol OO lO Ol 3 64      lO OO lO Ol 3 64      ll OO lO Ol 2 16
OO OO lO lO 2 16      Ol OO lO lO 3 64      lO OO lO lO 3 64      ll OO lO lO 2 16
OO OO lO ll 1 4       Ol OO lO ll 2 16      lO OO lO ll 2 16      ll OO lO ll 1 4
OO OO ll OO 0 1       Ol OO ll OO 1 4       lO OO ll OO 1 4       ll OO ll OO 0 1
OO OO ll Ol 1 4       Ol OO ll Ol 2 16      lO OO ll Ol 2 16      ll OO ll Ol 1 4
OO OO ll lO 1 4       Ol OO ll lO 2 16      lO OO ll lO 2 16      ll OO ll lO 1 4
OO OO ll ll 0 1       Ol OO ll ll 1 4       lO OO ll ll 1 4       ll OO ll ll 0 1
OO Ol OO OO 1 4       Ol Ol OO OO 2 16      lO Ol OO OO 2 16      ll Ol OO OO 1 4
OO Ol OO Ol 2 16      Ol Ol OO Ol 3 64      lO Ol OO Ol 3 64      ll Ol OO Ol 2 16
OO Ol OO lO 2 16      Ol Ol OO lO 3 64      lO Ol OO lO 3 64      ll Ol OO lO 2 16
OO Ol OO ll 1 4       Ol Ol OO ll 2 16      lO Ol OO ll 2 16      ll Ol OO ll 1 4
OO Ol Ol OO 2 16      Ol Ol Ol OO 3 64      lO Ol Ol OO 3 64      ll Ol Ol OO 2 16
OO Ol Ol Ol 3 64      Ol Ol Ol Ol 4 256     lO Ol Ol Ol 4 256     ll Ol Ol Ol 3 64
OO Ol Ol lO 3 64      Ol Ol Ol lO 4 256     lO Ol Ol lO 4 256     ll Ol Ol lO 3 64
OO Ol Ol ll 2 16      Ol Ol Ol ll 3 64      lO Ol Ol ll 3 64      ll Ol Ol ll 2 16
OO Ol lO OO 2 16      Ol Ol lO OO 3 64      lO Ol lO OO 3 64      ll Ol lO OO 2 16
OO Ol lO Ol 3 64      Ol Ol lO Ol 4 256     lO Ol lO Ol 4 256     ll Ol lO Ol 3 64
OO Ol lO lO 3 64      Ol Ol lO lO 4 256     lO Ol lO lO 4 256     ll Ol lO lO 3 64
OO Ol lO ll 2 16      Ol Ol lO ll 3 64      lO Ol lO ll 3 64      ll Ol lO ll 2 16
OO Ol ll OO 1 4       Ol Ol ll OO 2 16      lO Ol ll OO 2 16      ll Ol ll OO 1 4
OO Ol ll Ol 2 16      Ol Ol ll Ol 3 64      lO Ol ll Ol 3 64      ll Ol ll Ol 2 16
OO Ol ll lO 2 16      Ol Ol ll lO 3 64      lO Ol ll lO 3 64      ll Ol ll lO 2 16
OO Ol ll ll 1 4       Ol Ol ll ll 2 16      lO Ol ll ll 2 16      ll Ol ll ll 1 4
OO lO OO OO 1 4       Ol lO OO OO 2 16      lO lO OO OO 2 16      ll lO OO OO 1 4
OO lO OO Ol 2 16      Ol lO OO Ol 3 64      lO lO OO Ol 3 64      ll lO OO Ol 2 16
OO lO OO lO 2 16      Ol lO OO lO 3 64      lO lO OO lO 3 64      ll lO OO lO 2 16
OO lO OO ll 1 4       Ol lO OO ll 2 16      lO lO OO ll 2 16      ll lO OO ll 1 4
OO lO Ol OO 2 16      Ol lO Ol OO 3 64      lO lO Ol OO 3 64      ll lO Ol OO 2 16
OO lO Ol Ol 3 64      Ol lO Ol Ol 4 256     lO lO Ol Ol 4 256     ll lO Ol Ol 3 64
OO lO Ol lO 3 64      Ol lO Ol lO 4 256     lO lO Ol lO 4 256     ll lO Ol lO 3 64
OO lO Ol ll 2 16      Ol lO Ol ll 3 64      lO lO Ol ll 3 64      ll lO Ol ll 2 16
OO lO lO OO 2 16      Ol lO lO OO 3 64      lO lO lO OO 3 64      ll lO lO OO 2 16
OO lO lO Ol 3 64      Ol lO lO Ol 4 256     lO lO lO Ol 4 256     ll lO lO Ol 3 64
OO lO lO lO 3 64      Ol lO lO lO 4 256     lO lO lO lO 4 256     ll lO lO lO 3 64
OO lO lO ll 2 16      Ol lO lO ll 3 64      lO lO lO ll 3 64      ll lO lO ll 2 16
OO lO ll OO 1 4       Ol lO ll OO 2 16      lO lO ll OO 2 16      ll lO ll OO 1 4
OO lO ll Ol 2 16      Ol lO ll Ol 3 64      lO lO ll Ol 3 64      ll lO ll Ol 2 16
OO lO ll lO 2 16      Ol lO ll lO 3 64      lO lO ll lO 3 64      ll lO ll lO 2 16
OO lO ll ll 1 4       Ol lO ll ll 2 16      lO lO ll ll 2 16      ll lO ll ll 1 4
OO ll OO OO 0 1       Ol ll OO OO 1 4       lO ll OO OO 1 4       ll ll OO OO 0 1
OO ll OO Ol 1 4       Ol ll OO Ol 2 16      lO ll OO Ol 2 16      ll ll OO Ol 1 4
OO ll OO lO 1 4       Ol ll OO lO 2 16      lO ll OO lO 2 16      ll ll OO lO 1 4
OO ll OO ll 0 1       Ol ll OO ll 1 4       lO ll OO ll 1 4       ll ll OO ll 0 1
OO ll Ol OO 1 4       Ol ll Ol OO 2 16      lO ll Ol OO 2 16      ll ll Ol OO 1 4
OO ll Ol Ol 2 16      Ol ll Ol Ol 3 64      lO ll Ol Ol 3 64      ll ll Ol Ol 2 16
OO ll Ol lO 2 16      Ol ll Ol lO 3 64      lO ll Ol lO 3 64      ll ll Ol lO 2 16
OO ll Ol ll 1 4       Ol ll Ol ll 2 16      lO ll Ol ll 2 16      ll ll Ol ll 1 4
OO ll lO OO 1 4       Ol ll lO OO 2 16      lO ll lO OO 2 16      ll ll lO OO 1 4
OO ll lO Ol 2 16      Ol ll lO Ol 3 64      lO ll lO Ol 3 64      ll ll lO Ol 2 16
OO ll lO lO 2 16      Ol ll lO lO 3 64      lO ll lO lO 3 64      ll ll lO lO 2 16
OO ll lO ll 1 4       Ol ll lO ll 2 16      lO ll lO ll 2 16      ll ll lO ll 1 4
OO ll ll OO 0 1       Ol ll ll OO 1 4       lO ll ll OO 1 4       ll ll ll OO 0 1
OO ll ll Ol 1 4       Ol ll ll Ol 2 16      lO ll ll Ol 2 16      ll ll ll Ol 1 4
OO ll ll lO 1 4       Ol ll ll lO 2 16      lO ll ll lO 2 16      ll ll ll lO 1 4
OO ll ll ll 0 1       Ol ll ll ll 1 4       lO ll ll ll 1 4       ll ll ll ll 0 1


       Total States  =                                              10,000
```

From examining the Set of total programs, algorithms can be built for analysis of outcomes. As above, any program which begins with a 00 halts. Any which doesn't begin with an ADD but then has a 00 in the second location will halt. Loops, where the program will never halt can be detected when two passes through memory occur which are identical. i.e. the program counter resets to zero on overflow, and a pass through memory does not encounter a halt instruction.

Here is an analysis of programs by a simple program which does not execute all statements but detects Halts.

```
 1   00000000  loop ctr 0   Halts
 2   00000001  loop ctr 0   Halts
 3   00000010  loop ctr 0   Halts
 4   00000011  loop ctr 0   Halts
 5   00000100  loop ctr 0   Halts
 6   00000101  loop ctr 0   Halts
 7   00000110  loop ctr 0   Halts
 8   00000111  loop ctr 0   Halts
 9   00001000  loop ctr 0   Halts
 10  00001001  loop ctr 0   Halts
 11  00001010  loop ctr 0   Halts
 12  00001011  loop ctr 0   Halts
 13  00001100  loop ctr 0   Halts
 14  00001101  loop ctr 5   Loops *
 15  00001110  loop ctr 5   Loops *
 16  00001111  loop ctr 0   Halts
 17  00010000  loop ctr 0   Halts
 18  00010001  loop ctr 0   Halts
 19  00010010  loop ctr 0   Halts
 20  00010011  loop ctr 0   Halts
 21  00010100  loop ctr 0   Halts
 22  00010101  loop ctr 0   Halts
 23  00010110  loop ctr 0   Halts
 24  00010111  loop ctr 0   Halts
 25  00011000  loop ctr 0   Halts
 26  00011001  loop ctr 0   Halts
 27  00011010  loop ctr 0   Halts
 28  00011011  loop ctr 0   Halts
 29  00011100  loop ctr 0   Halts
 30  00011101  loop ctr 5   Loops *
 31  00011110  loop ctr 5   Loops *
 32  00011111  loop ctr 0   Halts
 33  00100000  loop ctr 0   Halts
 34  00100001  loop ctr 0   Halts
 35  00100010  loop ctr 0   Halts
 36  00100011  loop ctr 0   Halts
 37  00100100  loop ctr 0   Halts
 38  00100101  loop ctr 0   Halts
 39  00100110  loop ctr 0   Halts
 40  00100111  loop ctr 0   Halts
 41  00101000  loop ctr 0   Halts
 42  00101001  loop ctr 0   Halts
 43  00101010  loop ctr 0   Halts
 44  00101011  loop ctr 0   Halts
 45  00101100  loop ctr 0   Halts
 46  00101101  loop ctr 5   Loops *
 47  00101110  loop ctr 5   Loops *
 48  00101111  loop ctr 0   Halts
 49  00110000  loop ctr 0   Halts
 50  00110001  loop ctr 0   Halts
 51  00110010  loop ctr 0   Halts
 52  00110011  loop ctr 0   Halts
 53  00110100  loop ctr 0   Halts
 54  00110101  loop ctr 5   Loops *
 55  00110110  loop ctr 5   Loops *
```

```
56  00110111  loop ctr 0   Halts
57  00111000  loop ctr 0   Halts
58  00111001  loop ctr 5   Loops *
59  00111010  loop ctr 5   Loops *
60  00111011  loop ctr 0   Halts
61  00111100  loop ctr 0   Halts
62  00111101  loop ctr 5   Loops *
63  00111110  loop ctr 5   Loops *
64  00111111  loop ctr 0   Halts
65  01000000  loop ctr 0   Halts
66  01000001  loop ctr 0   Halts
67  01000010  loop ctr 0   Halts
68  01000011  loop ctr 5   Loops *
69  01000100  loop ctr 0   Halts
70  01000101  loop ctr 0   Halts
71  01000110  loop ctr 0   Halts
72  01000111  loop ctr 5   Loops *
73  01001000  loop ctr 0   Halts
74  01001001  loop ctr 0   Halts
75  01001010  loop ctr 0   Halts
76  01001011  loop ctr 5   Loops *
77  01001100  loop ctr 0   Halts
78  01001101  loop ctr 5   Loops *
79  01001110  loop ctr 5   Loops *
80  01001111  loop ctr 5   Loops *
81  01010000  loop ctr 0   Halts
82  01010001  loop ctr 0   Halts
83  01010010  loop ctr 0   Halts
84  01010011  loop ctr 5   Loops *
85  01010100  loop ctr 0   Halts
86  01010101  loop ctr 5   Loops *
87  01010110  loop ctr 5   Loops *
88  01010111  loop ctr 5   Loops *
89  01011000  loop ctr 0   Halts
90  01011001  loop ctr 5   Loops *
91  01011010  loop ctr 5   Loops *
92  01011011  loop ctr 5   Loops *
93  01011100  loop ctr 0   Halts
94  01011101  loop ctr 5   Loops *
95  01011110  loop ctr 5   Loops *
96  01011111  loop ctr 5   Loops *
97  01100000  loop ctr 0   Halts
98  01100001  loop ctr 0   Halts
99  01100010  loop ctr 0   Halts
100 01100011  loop ctr 5   Loops *
101 01100100  loop ctr 0   Halts
102 01100101  loop ctr 5   Loops *
103 01100110  loop ctr 5   Loops *
104 01100111  loop ctr 5   Loops *
105 01101000  loop ctr 0   Halts
106 01101001  loop ctr 5   Loops *
107 01101010  loop ctr 5   Loops *
108 01101011  loop ctr 5   Loops *
109 01101100  loop ctr 0   Halts
110 01101101  loop ctr 5   Loops *
111 01101110  loop ctr 5   Loops *
112 01101111  loop ctr 5   Loops *
113 01110000  loop ctr 0   Halts
114 01110001  loop ctr 0   Halts
115 01110010  loop ctr 0   Halts
116 01110011  loop ctr 5   Loops *
117 01110100  loop ctr 0   Halts
118 01110101  loop ctr 5   Loops *
119 01110110  loop ctr 5   Loops *
120 01110111  loop ctr 5   Loops *
121 01111000  loop ctr 0   Halts
122 01111001  loop ctr 5   Loops *
123 01111010  loop ctr 5   Loops *
124 01111011  loop ctr 5   Loops *
```

```
125 01111100  loop ctr 0   Halts
126 01111101  loop ctr 5   Loops *
127 01111110  loop ctr 5   Loops *
128 01111111  loop ctr 5   Loops *
129 10000000  loop ctr 0   Halts
130 10000001  loop ctr 0   Halts
131 10000010  loop ctr 0   Halts
132 10000011  loop ctr 5   Loops *
133 10000100  loop ctr 0   Halts
134 10000101  loop ctr 0   Halts
135 10000110  loop ctr 0   Halts
136 10000111  loop ctr 5   Loops *
137 10001000  loop ctr 0   Halts
138 10001001  loop ctr 0   Halts
139 10001010  loop ctr 0   Halts
140 10001011  loop ctr 5   Loops *
141 10001100  loop ctr 0   Halts
142 10001101  loop ctr 5   Loops *
143 10001110  loop ctr 5   Loops *
144 10001111  loop ctr 5   Loops *
145 10010000  loop ctr 0   Halts
146 10010001  loop ctr 0   Halts
147 10010010  loop ctr 0   Halts
148 10010011  loop ctr 5   Loops *
149 10010100  loop ctr 0   Halts
150 10010101  loop ctr 5   Loops *
151 10010110  loop ctr 5   Loops *
152 10010111  loop ctr 5   Loops *
153 10011000  loop ctr 0   Halts
154 10011001  loop ctr 5   Loops *
155 10011010  loop ctr 5   Loops *
156 10011011  loop ctr 5   Loops *
157 10011100  loop ctr 0   Halts
158 10011101  loop ctr 5   Loops *
159 10011110  loop ctr 5   Loops *
160 10011111  loop ctr 5   Loops *
161 10100000  loop ctr 0   Halts
162 10100001  loop ctr 0   Halts
163 10100010  loop ctr 0   Halts
164 10100011  loop ctr 5   Loops *
165 10100100  loop ctr 0   Halts
166 10100101  loop ctr 5   Loops *
167 10100110  loop ctr 5   Loops *
168 10100111  loop ctr 5   Loops *
169 10101000  loop ctr 0   Halts
170 10101001  loop ctr 5   Loops *
171 10101010  loop ctr 5   Loops *
172 10101011  loop ctr 5   Loops *
173 10101100  loop ctr 0   Halts
174 10101101  loop ctr 5   Loops *
175 10101110  loop ctr 5   Loops *
176 10101111  loop ctr 5   Loops *
177 10110000  loop ctr 0   Halts
178 10110001  loop ctr 0   Halts
179 10110010  loop ctr 0   Halts
180 10110011  loop ctr 5   Loops *
181 10110100  loop ctr 0   Halts
182 10110101  loop ctr 5   Loops *
183 10110110  loop ctr 5   Loops *
184 10110111  loop ctr 5   Loops *
185 10111000  loop ctr 0   Halts
186 10111001  loop ctr 5   Loops *
187 10111010  loop ctr 5   Loops *
188 10111011  loop ctr 5   Loops *
189 10111100  loop ctr 0   Halts
190 10111101  loop ctr 5   Loops *
191 10111110  loop ctr 5   Loops *
192 10111111  loop ctr 5   Loops *
193 11000000  loop ctr 0   Halts
```

```
194 11000001  loop ctr 0   Halts
195 11000010  loop ctr 0   Halts
196 11000011  loop ctr 1   Halts___ Executes Data
197 11000100  loop ctr 0   Halts
198 11000101  loop ctr 0   Halts
199 11000110  loop ctr 0   Halts
200 11000111  loop ctr 1   Halts___ Executes Data
201 11001000  loop ctr 0   Halts
202 11001001  loop ctr 0   Halts
203 11001010  loop ctr 0   Halts
204 11001011  loop ctr 1   Halts___ Executes Data
205 11001100  loop ctr 0   Halts
206 11001101  loop ctr 5   Loops *
207 11001110  loop ctr 5   Loops *
208 11001111  loop ctr 1   Halts___ Executes Data
209 11010000  loop ctr 0   Halts
210 11010001  loop ctr 0   Halts
211 11010010  loop ctr 0   Halts
212 11010011  loop ctr 5   Loops *
213 11010100  loop ctr 0   Halts
214 11010101  loop ctr 5   Loops *
215 11010110  loop ctr 5   Loops *
216 11010111  loop ctr 5   Loops *
217 11011000  loop ctr 0   Halts
218 11011001  loop ctr 5   Loops *
219 11011010  loop ctr 5   Loops *
220 11011011  loop ctr 5   Loops *
221 11011100  loop ctr 0   Halts
222 11011101  loop ctr 5   Loops *
223 11011110  loop ctr 5   Loops *
224 11011111  loop ctr 5   Loops *
225 11100000  loop ctr 0   Halts
226 11100001  loop ctr 0   Halts
227 11100010  loop ctr 0   Halts
228 11100011  loop ctr 5   Loops *
229 11100100  loop ctr 0   Halts
230 11100101  loop ctr 5   Loops *
231 11100110  loop ctr 5   Loops *
232 11100111  loop ctr 5   Loops *
233 11101000  loop ctr 0   Halts
234 11101001  loop ctr 5   Loops *
235 11101010  loop ctr 5   Loops *
236 11101011  loop ctr 5   Loops *
237 11101100  loop ctr 0   Halts
238 11101101  loop ctr 5   Loops *
239 11101110  loop ctr 5   Loops *
240 11101111  loop ctr 5   Loops *
241 11110000  loop ctr 0   Halts
242 11110001  loop ctr 0   Halts
243 11110010  loop ctr 0   Halts
244 11110011  loop ctr 2   Halts___ Executes Data
245 11110100  loop ctr 0   Halts
246 11110101  loop ctr 5   Loops *
247 11110110  loop ctr 5   Loops *
248 11110111  loop ctr 5   Loops *
249 11111000  loop ctr 0   Halts
250 11111001  loop ctr 5   Loops *
251 11111010  loop ctr 5   Loops *
252 11111011  loop ctr 5   Loops *
253 11111100  loop ctr 0   Halts
254 11111101  loop ctr 5   Loops *
255 11111110  loop ctr 5   Loops *
256 11111111  loop ctr 5   Loops *
```

Distinguishing halting from non-halting programs algorithmically seems odd as from the above a computer program can be written (and was) which would be able to distinguish halting programs from non-halting programs. This violates the limitation given in the Turing Halting Problem. Therefore this CPU is not a Turing Machine, for if it were it would violate Turning's proof as well as others – Gödel et al?.

This highlights a problem with generalizations about decidability and incompleteness. (especially in the Arts) Such generalizations are not true. It is possible to make systems which are completely decidable, though *perhaps* not using higher mathematics, or languages with complex syntax? However the important point here is not to confuse or conflate un decidability (in advance of an action) with uncertainty as to what alternatives are possible, the outcomes are not uncertain but fixed and knowable. Our objects here, certainly the very simple 2 bit CPU are humanly determinable.

It appears in this case the CPU is predictable because it does not test an input of the kind – "Halt if input is true". A program with such an instruction is un decidable until it runs and acquires a definite input. However its possible states given the inputs can be precisely known. Its only by linking it to something which is indeterminate that indeterminacy is generated, though again this will be in not knowing which of a fixed set of states will occur. It is not a generator of some absolute scepticism. (There is a danger in generalizing mathematical and scientific statements.) An un decidability here (of the halting problem?) is linked to the fact that such (Turing) Machines are linked in some way so that the 'outside' world affects their processing. That would be a source of un decidability. However its possible to construct objects which are not un decidable, both in theory and practice.

It is possible to make 'knowable' objects and these can be fully known – therefore known both logically and aesthetically. As things in themselves. Given even very simple instruction sets many of these objects can be constructed. That they exhibit this determinacy may make them for many aesthetically dull and trivial. They lack the "allure" of Harman's objects, they lack any 'poetry' if poetry is to be considered as something non-mundane.
(yet for some poets mundanity is not a poetic exclusion. i.e. Conceptual poetry http://www.poets.org/viewmedia.php/prmMID/22097)

The status of these programs in such a limited 'universe' may be difficult to define with regard to 'science' and 'philosophy', in not being scientific, (computer science). Or as 'art', appearing naively scientific/mechanical, and not having an 'optical' aesthetic. However there is nothing novel in these phenomena, in their having a potential to be objects with regards to "Art". Duchamp's ready-mades are naïve 'mechanical' objects which 'pose questions.' "The great glass" is an algorithmic mechanism, or model of one.

"The Large Glass depicts a chain reaction among abstract forces. That's why Duchamp subtitled it "a delay in glass" - because it shows a sequence of interactions suspended in time. This chain of events involves two component sequences, which intersect.

One sequence describes the interaction of female and male desire. Lets call in the Amours Pursuit. It has a beginning and an end.

The other sequence describes the influence of chance and destiny. Lets call it the Fate Machine. It is continually in motion.

Andrew Stafford
http://www.understandingduchamp.com/

And there are many other examples of Non-aesthetic objects functioning in or as an Art Context. Kosuth's work, Art & Language's 'The Air Conditioning Show' – for example.

A return to dogmatic metaphysics.

Within our program there are "impossibilities". This excludes the non-halting programs as they are definite, but certain programs which have code after the Halt statement, this code is never executed. `10010100` Here `0101` - "input" x 2 and `10` – "output" are never executed. They are not 'realizable'. Other impossibilities exist. In the given instruction set above for instance certain 'states' can never arise. For example "`00`" in the Program Counter on a Halt instruction with no overflow. Instruction sets could be devised for making these possible, realizable. These programs do not represent anything in terms of higher levels, they are not metaphysics but alternative universes in which 'laws' of one 'universe' are different, but not inferior or superior to laws in other universes. Other alternative universes could or would have different laws which would realize such 'impossibilities'.

Other instruction sets can be created and physically/ phenomenologically explored so long as we restrict these to two bits. (The restriction only being made by human longevity rather than cognition!)These represent a collection of objects with which we can gain an 'absolute' knowledge. And the possibility therefore of a 'Dogmatic Metaphysics'. They have such properties, *for us*, as a result of their limited sets of states. Larger 'Universes' are, for us, phenomenologically opaque. These "2 bit" universes might appear trivial and non radical, yet they can be thought of as being open to dogmatic metaphysical thought.

These objects do not radically challenge any other systems, or offer any concepts unless we regard these apriori states as 'indicative'. This is a philosophical speculation and not an dogmatic metaphysical aesthetic. A speculation which might claim that they do seem to represent 'states' no different from others. Pace Adorno / Derrida's impossibilities…… Or is it that these impossibilities which are located in our universe are not bound by the same logic as 2 bit universes, are in someway radically different.

An absolute knowledge?

Question – is the number of instruction sets fixed? Are there a fixed number of operations. Given that compound instructions are not allowed – for that effectively increases the (2 bit) Address space, the instruction set size would appear fixed. To create infinites multiple operations of the same operation would be needed and these couldn't be effected using the given address space. (Esoteric – infinite - operations are not allowed unless the physical devices are present to allow them.) The model for our instruction set is finite, and knowable, the model of possible instruction sets looks also finite though not knowable fully. Its hard to create infinities with finite objects other than by repetitions.

The nature of these "Address spaces" – an address space is the maximum space a CPU can address by virtue of its Instruction Pointer – i.e. 2 bits gives 2^2 of an address space which is 4 total locations. Each location has a number of bits which in our case was 3 and then 2, which is the 'word' size of the given CPU. This gives the total states that space can be in- which is $2^{\text{Address Space x Word Size}}$. For 2 bits 256. 4 locations of 2 bits each. 4 x 2 = 8. 2^8 is 256. For 3 bit word / I.P. = 2^{24} or 16,777,216. These are not Synthetic Apriori states as they do not follow from the facts of the Address space, they neither precede or follow – they are not logical consequences but simultaneous. Just as 16,777,216 does not follow from 2^{24}… or the sequence 000000000000000000000000, 000000000000000000000001… does not follow, they are 'the same' immediacy. Secondly, larger address spaces its been noted cannot be aesthetically engaged with by humans. A property of these larger spaces is that they become in appearance more and more like the world of our external experiences. They can capture pictures, sounds, and moving images, model buildings and structures. However as in the case of our simple CPU all these models will have limits. All these will have rules, and sets of possible rules will create different laws governing each universe. Some will be useful if they match our own- those we experience in the world in which we live, in designing bridges and buildings etc. But it is just as possible to have digital universes with different "laws". However these are not possibilities but "givens" when we have a finite address space- or infinite address space. It might be that such givens are non correlationist, non anthropological, dogmatic metaphysical objects, which lie outside of any science which is in effect only 'local'.

A definite absolute.

'The real world' including 'real computers' have sufficient states they can be in, to make their behaviour opaque for us. Science simply generalizes models and notes their correlation to the properties of nature which emerge from this opacity. Philosophy pursues a more radical commitment to actuality, but it too can only generalize. The gap between the complexities and accounts of them can be regarded as dealt with by treating the many as one, as in the case of insurance companies (and others) statistical assessments of reality.

Any move from 2 bits to 3 or 4 means we too would soon only be able to 'generalize' as to the possible or actual states of an object. These can take the form of mathematical models, or empirical observations. Computer systems behave in ways we can observe yet probably can never fully explain simply due to the possible states they are capable of being in. We might take such observations as signs of irrationality and / or intelligence. Philosophy in the history of metaphysics has also attempted to conceptualize reality. Ignoring this we can take one final move in the 'opposite' direction in terms of quantities. A 1 bit system represents the lower limit of possible states, before non at all. In the opposite upper direction we might consider 10^{100} or so as a theoretical upper limit of possible states, as given by the estimated number of particles in the universe. However its quite possible that there are other universes and if so this figure would need to revised to an infinity. This presents both philosophy and science with a problem, which is, given this infinity, all laws, not just those we find here, would become realized. This realization is not a probability but an actuality. As we have seen once you decide on an address space, apriori the combinations of bit patterns, which represent programs and machine states is fixed. An infinite address space would be no different. However the above is 'lost' to me as a non philosopher, non scientist. What is not lost to me are objects that I can know at first hand. These have been reduced to nothing substantial in the 'real world'.
I may not know the fundamental state of the real world, but I do know the totality of scenarios of a 2 bit universe.

With a one bit universe I can move to knowing every state and well as every program, I can then see not just a given one bit CPU instruction set, and its behaviour but ALL possible behaviours. For now the address space is 2, with therefore 4 programs in total, 00,01,10,11. And given 1 bit I have 2 possible instructions. 0 = halt 1 = add, for instance. We considered above, and worked through programs which alter the machines state, the contents of its registers. The 2 bit universe has a memory space of 8 locations and a program counter with overflow (3 possible states i.e. 00, 01,10, 11 and overflow either 1 or 0) a control unit (2), the Accumulator (5 – 2 inputs of 2, an output and overflow), and the I/O register (2). This gives 20 possible states – some will not be possible in certain instruction sets, but no more than 2^{20} possible states no matter what instruction set we devise. 2^{20} is 1,048,576. Again the size removes the possibility of our immediate total comprehension.

However with a 1 bit CPU (Universe) I can map the registers as well as knowing all possible programs (scenarios) – only 4. If there are 5 registers, A program counter, the Control Unit, the inputs to the ALU and its output, each being 1 bit, plus the 2 bits of memory, we can only address 2 bits, location 0 and location 1, then we have 7 in total bits which represent the CPUs possible states. 2^7 is 128.

Here are the 128 total *states* for this 1 bit CPU.

```
0000000     0110011
0000001     0110100
0000010     0110101
0000011     0110110
0000100     0110111
0000101     0111000
0000110     0111001
0000111     0111010
0001000     0111011
0001001     0111100
0001010     0111101
0001011     0111110
0001100     0111111
0001101     1000000
0001110     1000001
0001111     1000010
0010000     1000011
0010001     1000100
0010010     1000101
0010011     1000110
0010100     1000111
0010101     1001000
0010110     1001001
0010111     1001010
0011000     1001011
0011001     1001100
0011010     1001101
0011011     1001110
0011100     1001111
0011101     1010000
0011110     1010001
0011111     1010010
0100000     1010011
0100001     1010100
0100010     1010101
0100011     1010110
0100100     1010111
0100101     1011000
0100110     1011001
0100111     1011010
0101000     1011011
0101001     1011100
0101010     1011101
0101011     1011110
0101100     1011111
0101101     1100000
0101110     1100001
0101111     1100010
0110000     1100011
0110001     1100100
0110010     1100101
```

```
1100110
1100111
1101000
1101001
1101010
1101011
1101100
1101101
1101110
1101111
1110000
1110001
1110010
1110011
1110100
1110101
1110110
1110111
1111000
1111001
1111010
1111011
1111100
1111101
1111110
1111111
```

Where each of the first 5 bits represent a register, the last 2 the memory. Now given our example instruction set above.

0 = halt
1 = Add

We can see that this list represents all possible states that this instruction set could put the CPU into. But further it represents all the possible states ANY instruction set can put the CPU into.

0 = move 1 to the Program counter
1 = Halt

0 = Halt
1 = set location 0 to 1

….

Etc.

The 128 states in the list above not only represents 1 instruction set's programs as we examined in our 2 bit universe – whose instruction set we fixed – i.e. its laws of possible behaviours, but ALL states for ALL instruction sets. All possible behaviours – i.e. The totality of all laws and states or histories this CPU can have.

If we regard a program as an object, it will have component parts, instructions and data. In our 2 bit example each of these was 2 bits and we noted in some cases a program can treat an instruction on a second or third pass as data. That is a case where meaning is not fixed, in others meaning is fixed. We can undermine these objects, programs, to their constituent instructions, and these to their bits. However this doesn't mean a program can be understood by knowing its precise fundamental constituents, as in the case of

```
196 11000011  loop ctr 1   Halts___ Executes Data
200 11000111  loop ctr 1   Halts___ Executes Data
204 11001011  loop ctr 1   Halts___ Executes Data
208 11001111  loop ctr 1   Halts___ Executes Data
244 11110011  loop ctr 2   Halts___ Executes Data
```

These treat 00 as data in one pass and as the Halt instruction in another. Knowing the 00 component as a fundamental part of the object doesn't reveal fully what the object 'is'. However we can by direct experience of the object 'know' it fully, in this case know that on one pass 00 is data on another it is HALT, for this particular program / object. We can its been seen also 'overmine' in classification of 'types', species of programs. Perhaps we can not properly regard such activities as 'science' or 'metaphysics'. We might want to regard `11000011` as an 'object', as of singular interest, as an 'art' object. (a picture or poem!)

From our understanding of the CPU's function we can establish certain facts. For instance 'identity'. Within simple computer science this test was given when the result of a subtraction is zero. 'Meaning' is the bit string's context, i.e. 111 is ADD or 7 or -1, within the given laws of the instruction set.

A dogmatic metaphysics can discover 'facts' from these systems. (*though of interest, their wider scope is outside of this present work*) Here I present 10 as a beginning, but I doubt this list is definitive.

1. An objects self annihilation is its identity.
2. A definite absolute is a genetic device.
3. The essence of an object is the totality of the total multiple universes.
4. Onticity and Ontology are identical.
5. Justice is the acknowledgement of an object's identity.
6. Law is the acknowledgement of the essences of an object.
7. An ethics can be made by seeing that laws should acknowledge an objects identity.
8. Truth is that objects are just.
9. Thinking is the pursuit of Justice.
10. The concept and the object are the same.

Dogmatic Metaphysics: 1

Difference and absolute Difference.

Within a single bit CPU only a difference between 1 & 0 exits. Hence there can only be 2 possible actions in its instruction sets. However there are 128 possible states, each being different. Analysis of differences is not however needed even though this is possible. For instance 1000000 & 0000000 have differences – one has 7 0s the other 6, one has one 1, the other none, one has a 1 in the first left position of the string, the subtractive difference is 128, the play of differences is even here quite rich. However the absolute difference *is* as we have seen the simple matter of subtraction… in that any non zero result is different. If x-y = 0 they are identical. An objects self annihilation is its identity, or its failure at annihilation the proof of its non-identity.

Dogmatic Metaphysics: 2

`1101011`

`1101011` is an absolute state (1 of 128). If we ask what `1101011` means, given the 128 states it could be any of the total states which the totality of instruction sets are. This is an ontological ground which represents a totality. It is an absolute state.

`1101011`

Is epistemologically transparent, we can know it, it is ontologically transparent, we can see this. And it is ontically a totality, and therefore a genetic device…

A definite absolute is a genetic device.

Dogmatic Metaphysics: 3

Essence, existence and Identity.

Essence is given apriori, as is existence. Identity is the objects self annihilation. Thus its existence *is* regardless of its essence. 1000000 – x = 0,
Here x's essence is known- 1000000, x-y = 0, x's essence is unknown but its existence is. Similarly essences of 'large' objects – real objects are very large, here they are 7 bits, hence knowable. Existence is knowable as the possibility of annihilation, removal.. though this does not reveal essence. The essence of an object is the totality of the total multiple universes. The totality is the finitude of all programs and states. As in 2+5 = 2^7 states is the multiverse which gives the finite essence of each object.

Dogmatic Metaphysics: 4

Representation: A object represents itself. Onticity and Ontology are identical.

Dogmatic Metaphysics: 5

Justice is the acknowledgement of an object's identity.

Dogmatic Metaphysics: 6

A law describes the violation of a state, the impossibility of a state within the given laws. This possibility then can only be actualized by another set of laws. Laws here are instruction sets. A morality can be constructed by seeing that laws must acknowledge an objects identity as a matter of its existence even if its essence is unknown, and that justice is apriori in the existence of an object. Law is the acknowledgement of the essences of an object.

Dogmatic Metaphysics: 7

An ethics can be made by seeing that laws should acknowledge an objects identity.

Dogmatic Metaphysics: 8

Truth is that objects are just.

Dogmatic Metaphysics: 9

Thinking is the pursuit of Justice.

Dogmatic Metaphysics: 10

The concept and the object are the same.

`1101011` and the concept `1101011` are identical. If they were not identical then there relationship would be of difference, of two different objects. That the "concept" 'triangle' doesn't 'capture' all triangles shows that it is not a concept which is complete, but a generalization. Such as 7 binary digits. How '7 binary digits' is in no way the concept of `1101011` should be obvious- given an instruction set we would not know anything from this 'concept' '7 binary digits'. Within epistemological metaphysics language becomes separate from its object- within dogmatic metaphysics they are one-and-the-same. Hence the idea of 'poetry' becoming the metaphysics of objects. A poem's object is itself. What is lost in translation is 'everything', is its 'being'.