

# The Smallest Arithmetic Logic Unit

Raul Rojas  
Freie Universität Berlin  
January 2010

## Sequential logic

There have been many designs for „small universal computers“. The Turing machine (1936), for example, remains unsurpassed: it consists only of memory, and a small processor with a few number of internal states. The memory can be designed to store only binary digits (0 and 1) and the number of states can be minimized. Scheming minimal universal Turing machines has been a mental sport for years [1].

However, the Turing machine makes use of an unrealistic assumption in terms of today`s computers. The processor “moves” along the memory and no memory-addresses have to be used. Only the symbol below a read/write head can be read or written. In modern computers, memory chips are addressable. The processor must indicate which exact memory address is being read or written. Therefore, since the processor is fixed and immutable, and data will be read from memory, one interesting question to ask is how much logic we really need for the processor of a universal computer with addressable memory. A Turing machine can be simulated by using an address register which is advanced or decremented by one.

Alan Turing himself tried to design a minimal computer with his “Pilot ACE” [2]. This machine used registers which were read bit by bit (by shifting the registers). Instead of adding two 20-bit numbers reading the two numbers in parallel, for example, the Pilot ACE would read one bit from each number, add them, keep score of the carry, and continue adding the next two bits. This is called “sequential logic”. Here, a compromise is made between the need for speed (parallel operations) and the need for simplicity (reading data sequentially, that is bit by bit). Parallel logic is fast, sequential logic is simple.

Alan Turing was not the only one who designed sequential-logic machines: The ABC, built by John Atanasoff in Iowa in 1939 would read two numbers, each one of them bit by bit, and would also store the result of an addition sequentially [3]. The numbers were stored on capacitors arranged on sectors of rotating cylinders. The read head was activated each time a bit passed by. Also, some computers built in the 1950s used “mercury stores”, where bits were stored as trains of pressure waves circulating in a tank. The processor could just “sit there” and wait for all bits in memory to pass by the reading sensors. John von Neuman designed a sequential logic machine for the Institute of Advanced Studies at Princeton [4].

## The processor

Fig. 1 shows the complete design of the first processor we want to consider in this paper. The memory is bit-addressable. The Program counter (PC) keeps score of the next address to be fetched from memory during execution of a program. Instructions can be  $m$ -bit long, that is, as long as decided by the designers (32 bits, 64 bits, etc.). The instruction read from

memory is stored in the Instruction register (IR). The bits for the IR are read one by one until they fill the IR, at which point the PC stops retrieving bits from memory and the instruction stored in the IR is executed.

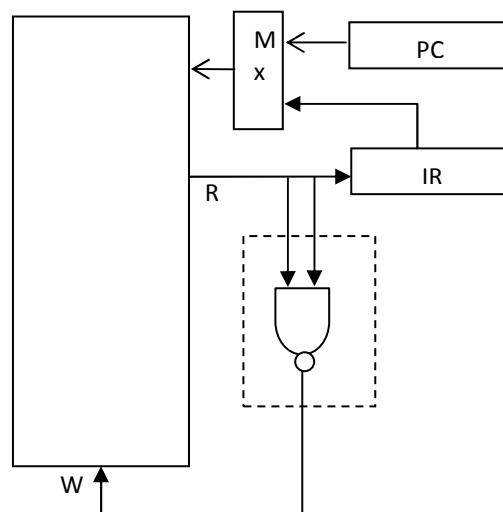
There are only two instructions in this machine: Load A and Store A, where A is a memory address. For example a program could be:

```

Load 100      ; load bit at address 100
Load 101      ; load bit at address 101
Store 200     ; store bit at address 200

```

When the first bit is loaded, it is used as the first argument for a NAND gate, the only logical circuit in the ALU. When the second bit is loaded, it is used as the second argument. The second argument triggers the execution of the NAND. When a store is executed, it stores the result of the NAND operation. The program given above, therefore, computes the NAND of the two bits at address 100 and 101, and stores the result at address 200. The PC and IR share the addressing bus, using a multiplexer.



That's it. Some logic is needed to keep track of the fill-up process of register IR, some more to advance and reset the PC (an incrementer), and some logic to keep everything nicely synchronized. In this design, the PC always advances forward, address by address, until it reaches the last address in the program, at which point the PC resets to zero, and the execution loop starts again. This could be done by using two's complement arithmetic. If we only had 16 addresses for the program code, for example, the PC would cycle through all numbers 0,1, .., 15, and then zero again at the next increment of the PC.

The basic operation loop of the machine is the following:

```

For i=1 to m
  IR(i) = readbit (PC)
  PC    = PC + 1

A= readbit(IR)

```

```

For i=1 to m
  IR(i) = readbit (PC)
  PC    = PC + 1

B= readbit(IR)

For i=1 to m
  IR(i) = readbit (PC)
  PC    = PC + 1

C= A NAND B
Writebit(IR,C)

```

Here, we call A and B the two inputs to the NAND gate, while C is the output of this gate. If two load operations are always followed by a store, there is no need to store the opcode of the operations in memory. The program consists only of sequences of memory addresses.

In order to stop the machine, we could use a special address: if a zero is written to that address, the machine stops. We would need a special decoder for that address. Note however, that modern computers never stop! The operating system is running continuously, and even during standby some part of the processor is running. You have to pull the plug or kindly ask the OS to stop running the electricity bill.

The machine pictured above executes relentlessly a giant loop of code, instruction by instruction. Every time, only two bits are combined through a NAND into a result bit, which is stored again in memory. The basic cycle of the machine consists of three instructions (Load, Load, Store) executed always in the same pattern.

### Why it works

This simple machine is universal in the sense of Turing. It can compute anything a Turing machine can compute (provided we have enough memory) and anything a real computer can compute.

The proof is simple. A NAND gate can simulate AND, OR, and NOT gates, as it is well-known. Now think of *any* processor in *any* computer: it reads an instruction, executes it, and starts again, getting the next instruction from memory. The processor is a fixed circuit, it consists of interconnected AND, OR, and NOT gates. Since we can simulate each gate using combinations of NANDs, we just write a program which reads individual input bits and simulates each gate in the processor, one by one. We can keep track of the complete result of the whole circuit because we can store bits in auxiliary memory locations. Therefore, we can simulate the *complete processor* with a fixed program, which we can execute step by step. Synchronization and sequencing of results is done by the simulation program that we write. Our really small ALU can therefore simulate any complex processor in the world! Of course, it takes some time to run the simulation. This processor will be awfully slow compared with the processor being simulated, which probably has several levels of pipelining and operates with 64 bits or more at a time.

Any processor can be simulated, so much is clear. Now, how can we show that we can program any high-level program we would want to have? If we would allow the machine to

have conditional jumps (that is, if we had a way of changing the PC or reading it from memory) there is not much to prove. A processor which can do all types of arithmetic (as this processor can), which can read and write to memory, possibly modifying its own program, and can execute conditional jumps is universal [5].

It is known that a universal computer can be programmed using only a fixed sequence of operations within a single loop, which checks for termination at the beginning of every pass, by testing a single bit [6]. The main idea is to think of program code as divided in sections, “guarded” by an arithmetical variable. We enumerate the code sections in the single loop as  $s=1$ ,  $s=2$ , and so on. When an arithmetical expression in the original program is computed, such as

$a := b + c$

we transform this code into

$a := (1 - t) * (b + c) + t * a$

The variable  $t$  is 0 if this code section is to be executed and one if not. If  $t$  is zero,  $a$  is not modified, and it is as if this code section was never executed. We can set the variable  $t$  to 1 or zero according to its equality to a number  $s$ , using a bitwise XOR of  $t$  and  $s$ , followed by an OR of the result bits. If the result is zero,  $t$  and  $s$  are equal. If not,  $t$  and  $n$  are not equal. The function “equality” in the code below performs an equality test using only logical operations.

The source code of a program containing arithmetical operations in the main loop can be divided into sections. At the beginning section 1 is executed. Control changes from section to section setting the variable  $s$  to the next code section to be executed. The loop is executed completely, instruction by instruction, but only the active code section modifies the memory. In this way conditional jumps can be simulated. In the example below, we start with code section 1 and control is then transferred to code section 23. Every code section tests the values of  $s$  at the beginning and sets the guard variable appropriately.

```

s:=1
                                ; section 1 of code
t:= equality(s,1)
a:= (1-t)*(operations) + t*a
b:=...
...
s:= (1-t)*23 + t*s                ; guarded jump to section 23
...
                                ;section k of code
t:= equality(s,k)
w:= (1-t)*(operations) + t*w
...

```

### Relative addressing and self-modifying programs

One last remaining difficulty is this: Universal computation requires the ability to manipulate a memory and data structures of any possible size. In general, the result of a computation can have a number of bits which is unknown and non-computable in advance. Modern

computers just avoid this issue: if more memory is needed, buy more of it. Addressing that memory should be possible, so that the PC and IR must have the necessary number of bits.

In our design, the program is stored in memory and can be modified by the program itself. Self-modifying programs are very powerful and allow us to modify the address referred to by an instruction. This means that we can have counters or index arrays, for retrieving information from memory. If we want to address an array A using the index i, we can refer to A(i). Incrementing the index i by as many bits as an element of A contains, gives us the next element of A.

## Simplicity

This machine is uncomplicated in the sense that the PC is always advancing one by one. Only an incrementer is needed, conditional jumps are not executed, they are handled implicitly by the guards of code sections. The IR is filled up sequentially. A multiplexer takes the address for memory from the IR or from the PC according to the current execution phase (reading instruction or executing instruction). We need an address decoder inside the memory module. And we need some clocking mechanism in order to keep everything synchronized. The ALU in the CPU shown above contains a single gate.

Shockingly, we can remove this last gate. Assume that we want to process the bits at address 100 and 101, as above. Since the NAND function of two bits consists of a table with four entries, we can write the following program:

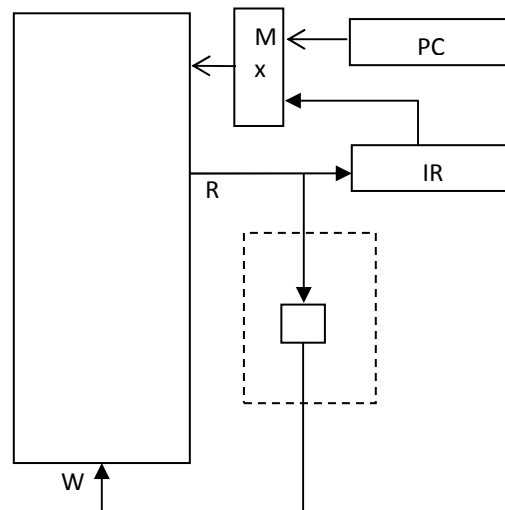
```
n      Load 100
n+1    Store n+4+(m-1)
n+2    Load 101
n+3    Store n+4+(m-2)
n+4    0
n+5    Store 200
```

With his program we construct an instruction at address n+4. It is the instruction “Load (addr)” where addr is 0,1,2,3 (since we stored two bits before). At the addresses 0,1,2,3 we place the NAND gate table, which is:

```
0 NAND 0 = 1
0 NAND 1 = 1
1 NAND 0 = 1
1 NAND 1 = 0
```

that is, we store the bits 1,1,1,0, in that order at addresses 0,1,2,3. Of course, we can store this table in any other addresses (so that the table does not collide with the rest of the code), just by storing more bits or initializing address n+4 to another value.

The resulting machine has the structure shown in the diagram below, where the bit which has been read is kept temporarily in the small box (which is some kind of storage).



## Conclusions

A universal computer with memory stored program, and strict sequential addressing of instructions, can be built, which only executes the two instructions Load and Store. In the diagram above PC and IR provide the address to the memory bank, through a multiplexer. The memory bank can be read or written bit-wise. Addresses are bit addresses.

Konrad Zuse designed in 1945 a computer with a very similar structure [7]. It was called the “logistic machine” (logistic meaning logic). It had only two gates, and AND and an OR gate. The input bits could be read as they were or negated. The program was stored in a punched tape, so that it could only be advanced instruction by instruction. The programs could not be modified, because they were in the punched tape. Although the machine was not universal, it really came close to the possibilities discussed here and should be mentioned.

## Exercises

1) Show that NAND gates can be used to build AND, OR, and NOT gates. Do the proof arithmetically, using these definitions:

$$\begin{aligned}
 x \text{ AND } y &= xy \\
 x \text{ OR } y &= x + y - xy \\
 \text{NOT } x &= 1-x \\
 x \text{ NAND } y &= 1-xy
 \end{aligned}$$

2) Show that blocks of code can be guarded using a single bit and logic operations, instead of multiplication.

## References

- [1] M. Minsky, *Computation: Finite and Infinite Machines*, Prentice Hall, 1967 .
- [2] Carpenter, B. E.; Doran, R. W., *A. M. Turing's ACE Report of 1946 and Other Papers*, Cambridge, MIT Press, 1986.
- [3] Clark R. Mollenhoff, *Atanasoff: Forgotten Father of the Computer*, Iowa State Press, 1988.

- [4] William Aspray, *John von Neumann and the origins of modern computing*, MIT Press, 1990.
- [5] R. Rojas, *Conditional Branching is not Necessary for Universal Computation in von Neumann Computers*, JUCS, V. 2, N. 11, 1996.
- [6] D. Harel, *On Folk theorems*, Communications of the ACM, V. 23, N. 7, 1980.
- [7] Konrad Zuse, *Logistische Maschine*, unpublished manuscript.