ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **9**, Number 3, 2006, 227–243

A Universal Turing Machine with Zero Internal States

Gheorghe ŞTEFAN

"Politehnica" University of Bucharest,
Faculty of Electronics, Tc. & Inf. Tech.

E-mail: `stefan@arh.pub.ro`

Abstract. Can we define the *simplest* Universal Turing Machine (UTM) as a structure containing only uniform circuits? Is there possible a 0-*state* machine containing only structures having constant sized simple definitions. In a 0-state UTM all random structures are missing, and the resulting system exhibits only uniform circuits. *Removing the finite automaton* (*FA*) *from the definition of UTM* is the main structural effect of our approach. The state of the computational process is stored only on the "tape", in contrast with the current versions of UTM in which the state of the computational process is given by both, the state of FA and the content of the "tape". In the current UTMs, FA *interprets* the description of a certain Turing Machine (TM) stored on the "tape". In the proposed, 0-state UTM, the description of a certain TM, stored on the same "tape", is *executed* only by uniform simple circuits.

1. Introduction

The 0-*state UTM* we propose in this paper is important mainly for its structural **simplicity**. The reduced number of states is only a side effect, allowing to segregate the entire *complexity* of computation into the symbolic stream of "instructions" stored on the "tape". Thus the physical structure of the machine is freed by the need of the FA, the only complex subsystem inside of each TM. But, what does it mean simple or complex structure? What is the main effect of reducing the *number of states* of FA that interprets any TM's description stored on the "tape" of a certain UTM? 50 years ago, Claude Shannon reduced this number to two [3]. Did he obtain a simple or a complex machine? We will see. Our first goal is now to obtain a **simple machine**. Going from 2 states to 0 states is not a meaningful target *per se*, it is only a way to minimize the complexity of UTM. Therefore, we must first state precisely the meaning of the term **complexity** when it refers to the physical structure of a digital machine.

Definition 1. The *apparent complexity* of a circuit is given by the size of its definition, measured in the number of symbols used to express it.

Definition 2. The (actual) *complexity* of a circuit is given by the size of its shortest definition.

According with these definitions we will define what means simple or complex circuit structure.

Definition 3. If a circuit (or a digital machine), M, characterized by a parameter n has the size, $S_M(n)$, and the complexity, $C_M(n)$, then we say that M is *simple* if

$$S_M(n) \in O(f(n)), \;\; C_M(n) \in O(1)$$

and we say that M is *complex* if

$$S_M(n) \sim C_M(n).$$

For a simple structure exists anytime an n_0 so that if $n > n_0$:

$$S_M(n) >> C_M(n).$$

Example 1. Let be an n-bit high-speed counter. Its size is in $O(n \times \log n)$. But, its complexity is in $O(1)$ because it can be described in a Hardware Description Language (HDL) by a constant length module. (Only when the HDL module is used for an actual circuit, its size is in $O(\log n)$, because the actual value of n is expressed by $\log_b n$ "digits" in base b.) Therefore, the counter is a **simple** circuit. □

Example 2. Let be an n-state finite automaton with 1-bit input (for the sake of simplicity n is a power of 2). The combinational logic circuit (CLC) which computes its transition functions has then $1 + \log_2 n$ binary inputs. Results, the size of CLC is $S_{CLC}(n) \in O(n)$.

The HDL description of this finite automaton has at least one line for each state, specifying the output and the next state. Therefore, $C_{CLC}(n) \in O(n)$.

Because the state register of the automaton has $S_{reg}(n) \in O(\log n)$ and $C_{reg}(n) \in O(1)$, results:

$$S_{finite\_automaton}(n) \sim C_{finite\_automaton}(n).$$

Therefore, our finite automaton is a **complex** circuit. □

The previous definitions are suggested by Chaitin's *algorithmic complexity* [1]. (In this approach we will ignore the "user" who must "understand", "execute" or "interpret" the definition. Sometimes the "user" is a human mind other times it is a machine, both having their own "complexity" to be taken into account. But this is another story.)

According to the previous definitions we will prove that the complexity of a TM is given only by the complexity of the combinational circuit connected on the loop of the FA. This circuit computes the output and the next state of FA. The rest of the machine, the "head" and the "tape", are both simple according to the previous definition of simplicity. In order to minimize the complexity of the UTM we will start with an up-dated representation for the TM and we continue presenting the 0-state UTM.

2. A New Look for the Turing Machine

The standard definition of TM, described with the storage "tape" and of the access "head", hides some essential features of the concept. In order to be able to emphasize aspects related to the complexity of TM, we will give an equivalent up to date definition of it. Instead of the "tape" accessed through a "head" we will use a *random access memory* (RAM) addressed with an *up-down counter*.

Definition 4. *Turing Machine* (TM) is a finite automaton (FA) loop connected with an infinite RAM addressed by an infinite up-down counter (U/DCOUNTER) commanded by FA (see Fig. 1). In each clock cycle the following operations are executed:

- FA receives from the output DOUT of RAM the content of the current accessed memory cell: `mem[addr]`
- according to its current state, `q_i`, and to the received symbol `mem[addr]`:
  1. FA computes a new symbol to be stored in RAM at the same address and applies it on the input DIN of RAM: `data(q_i, mem[addr])`
  2. FA determines the accessing mode of the next cell selecting one of the following actions:
     – increments the counter (UP): the accessed memory cell in the next clock cycle will be located at `[addr + 1]`
     – decrements the counter (DOWN): the accessed memory cell in the next clock cycle will be located at `[addr - 1]`
     – maintains the counter to the same value (-): the accessed memory cell in the next clock cycle will be located at `[addr]`
  3. FA computes the next state of the automaton:
     `next_q(q_i, mem[addr])`.

All changes are triggered with the next active clock transition.

More formal:

$$TM = (I,\ Q, f; q_0, \#)$$

where:

- I is the finite alphabet of TM,
- Q is the finite set of states of FA,

- $q_0 \in Q$ is the initial state of FA,
- $\# \in I$ is a symbol stored in all non active cells of the memory
- the transition function of the entire TM, f, is the parallel composition of the following functions:
  - `f_1:(I × Q)→I`, where `f_1(mem[addr], q) = data`
  - `f_2:(I × Q)→{UP, DOWN,-}`, where `f_2(mem[addr], q) = com`
  - `f_3:(I × Q)→Q`, where `f_1(mem[addr], q) = next_q`.

The initial state of FA is q_0. The infinite memory contains the finite string to be processed, and the rest is full of $\# \in I$. The U/DCOUNTER points to the first symbol in the string.

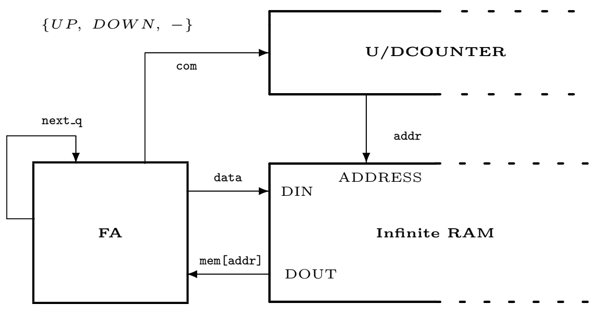

Fig. 1. The structure of a Turing Machine. The *complex* finite automaton is loop connected with a *simple* "infinite" RAM, and the resulting subsystem is again loop connected with a *simple* "infinite" up-down counter.

3. Turing Machine's Complexity

In Fig. 2 a detailed structure of TM is presented. It is used to evaluate TM's complexity. This form of TM contains:

1. a **finite automaton** (FA) composed by loop connecting:
   - a finite REGISTER: a **simple** and small structure
   - a combinational logic circuit (CLC): a **complex** structure used to compute the next state of FA and its current output
2. an **"infinite automaton"** embodied as a reversible counter, a simple, recursive defined device composed by loop connecting:
   - INFINITE REGISTER: a big sized but **simple** structure

- an “infinite” incrementer/decrementer (INC/DEC): a big sized but **simple** structure

3. an INFINITE RAM, also a big sized but **simple** structure.

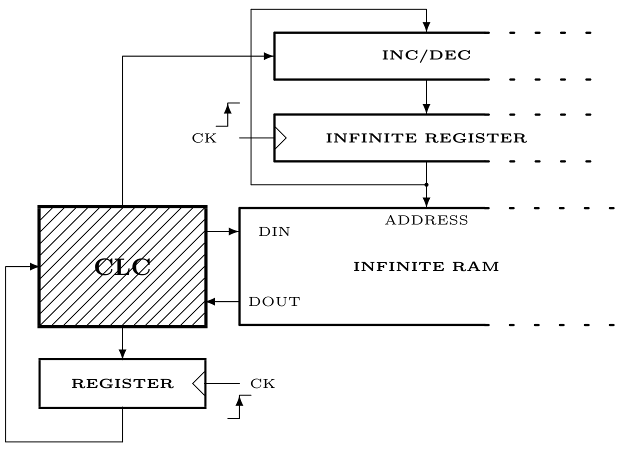

Fig. 2. The detailed structure of a Turing Machine. The only complex component of a TM is CLC closing the loop of FA (an immediate Mealy finite automaton). The state register is small and simple. The remaining components are “infinite” and simple: an “infinite” automaton and an “infinite” memory.

In order to evaluate the (apparent) complexity of a TM used to compute M we provide its HDL description. The following Verilog code is written to emphasize, in two distinct files, the simple components of TM separated from the complex part of TM. The first file with the module `turing_machine` is the following:

```
module turing_machine(state, symbol, reset, clock);
    parameter   n = 10, // address dimension
                m = 3 , // state register dimension
                t = 2 ; // data size
    input             reset, clock;
    output  [m-1:0] state   ;
    output  [t-1:0] symbol  ;
    reg     [m-1:0] state;
    reg     [m-1:0] next_state;
    reg     [t-1:0] data;
    reg     [1:0]   com;
// THE ONLY RANDOM PART: THE COMBINATIONAL LOGIC CIRCUIT
    ‘include "clc(M).v"
// THE STATE REGISTER
    always @(posedge clock) if (reset)  state <= 0;
                                else    state <= next_state;
```

```
// THE ADDRESS UP/DOWN COUNTER
    reg     [n-1:0] address;
    always @(posedge clock) if (reset)      address <= 1;
                            else case (com)
                                    2'b00: address <= address;
                                    2'b01: address <= address + 1;
                                    2'b10: address <= address - 1;
                                   default address <= address;
                                 endcase
// THE MEMORY
    reg     [t-1:0] mem[0:((1 << n) - 1)];
    always @(posedge clock) mem[address] <= data;
    assign symbol = mem[address];
endmodule
```

The previous module describes explicitly only the simple part of TM: the state register, the address counter and the memory. The description remains the same for any values of n (used to specify the dimension of the "infinite" memory), m (the size of the state register), and t (the length of the binary words used to code the elements of TM's alphabet). The only complex part (random in Chaitin's view) is contained in the included file: `clc(M).v`. The generic form of this file is codded in the following form:

```
// the generic form of the file clc(M).v describing the FA's CLC
    always @(state or symbol)
     case (state)
        4'b0000: case (symbol)
                    8'b0000_0000: {data, com, next_state} = {...};
                    8'b0000_0001: {data, com, next_state} = {...};
                    ...
                    8'b1111_1111: {data, com, next_state} = {...};
                 endcase
        4'b0001: case (symbol)
                    8'b0000_0000: {data, com, next_state} = {...};
                    8'b0000_0001: {data, com, next_state} = {...};
                    ...
                    8'b1111_1111: {data, com, next_state} = {...};
                 endcase
        ...
        4'b1111: case (symbol)
                    8'b0000_0000: {data, com, next_state} = {...};
                    8'b0000_0001: {data, com, next_state} = {...};
                    ...
                    8'b1111_1111: {data, com, next_state} = {...};
                 endcase
     endcase
```

The actual complex (random) content of this file is given by the way to fill up the parenthesis {...}. The two level of `case` forms corresponds to the two inputs in this combinational circuit: `state`, and `symbol`. The maximal complexity occurs when for each pair `(state × symbol)` the automaton has a distinct behavior. Thus, the maximum number of lines in the file `clc(M).v` is $const + dim(Q) \times dim(I)$ and the minimum number of lines is bigger than $const + dim(Q)$.

Example 3. Let us take as example a TM that computes the parity of an 1-ary represented number. If the final state is q_3, then the number is even, else it is odd. The initial value of T (the content of "tape") is:

$$\ldots \#0111\ldots 10\# \ldots,$$

the "head" points the first 0 and the description of the function f is:

$f(q_0, -) = (q_1, 0, U)$ /The head moves to the first 1/

$f(q_1, 1) = (q_2, 1, U)$ /Passes over the 1's switching between q_1 and q_2/
$f(q_1, \neq 1) = (q_3, 0, -)$ /The end of 1's is found and the number is even/

$f(q_2, 1) = (q_1, 1, U)$ /Passes over the 1's switching between q_2 and q_1/
$f(q_2, \neq 1) = (q_4, 0, -)$ /The end of 1's is found and the number is odd/

$f(q_3, -) = (q_3, 0, -)$ /Final state for an even number/

$f(q_4, -) = (q_4, 0, -)$ /Final state for an odd number/

The corresponding `clc.v` is:

```
// the file clc(parity_of_unary_numbers).v
always @(state or symbol)
 case (state)
  3'b000:          {data, com, next_state} = {2'b00, 2'b01, 3'b001};
  3'b001: case (symbol)
             2'b01: {data, com, next_state} = {2'b01, 2'b01, 3'b010};
           default {data, com, next_state} = {2'b00, 2'b00, 3'b011};
          endcase
  3'b010: case (symbol)
             2'b01: {data, com, next_state} = {2'b01, 2'b01, 3'b001};
           default {data, com, next_state} = {2'b00, 2'b00, 3'b100};
          endcase
  3'b011:          {data, com, next_state} = {2'b00, 2'b00, 3'b011};
  3'b100:          {data, com, next_state} = {2'b00, 2'b00, 3'b100};
  default          {data, com, next_state} = {2'bxx, 2'bxx, 3'bxxx};
 endcase
```

□

CLC is the single complex structure of TM, because it "contains" the algorithm. All the others components of TM have the complexity in $O(1)$. Therefore, it is obvious that C_{TM} is in the same magnitude order with C_{CLC}, i.e., $C_{TM} \sim C_{CLC}$.

Proposition 1. Let be $TM = (I, Q, f; q_0, \#)$. Then:

$$C_{TM} \in O((\log(\dim(I)) + \log(\dim(Q))) \times \dim(I) \times \dim(Q)).$$

Proof. Suppose for the sake of simplicity that $\dim(I)$ and $\dim(Q)$ are power of 2. Our TM is characterized by n, the number of bits used to code the alphabet, and by m, the number of bits used to code the states of FA, where:

$$n = \log_2(\dim(I))$$

$$m = \log_2(\dim(Q)).$$

The complexity of TM, C_{TM}, is given by the sum of the complexity associated of each component added with the complexity of their interconnections. Excepting CLC, all the others components and the interconnections have constant descriptions that not depends by n and m. Thus CLC has $n+m$ inputs and $n+m+2$ outputs (see Fig. 2). Therefore, its definition in the worst case is a random (uncompressible) table of binary symbols having 2^{n+m} rows and $n+m+2$ columns (see also the generic form of the file `clc.v`). Therefore:

$$C_{TM}(n, m) = \text{const.} + C_{CLC} \in O((n+m)2^{n+m}).$$

By substituting the value of n and m we prove the proposition. □

The computation performed by TM called M on a data string of n symbols is characterized by the following three parameters:

- C_M: the **complexity** of the TM called M, computed as the sum between the size of the Verilog module `turing_machine` and the size of the file `clc(M).v`:

$$C_M = S_{turing\_machine} + S_{clc(M).v} = k_{TM} + S_{clc(M).v}$$

- $S_M(n)$: the **size** of memory used to perform the computation M
- $T_M(n)$: the **time**, expressed in the number of clock cycles used to complete the computation M.

We must notice that C_M does not depend on n. This is the main idea of computability expressed by *"finite"* from the *finite automaton.* For any value of n (even for an *infinite* value) the complexity of the algorithm is the same.

Example 4. The TM `parity_of_unary_numbers` (see **Example 3**) with an n-bit input string "on the tape" is characterized as following:

- $C_{parity\_of\_unary\_numbers} = k_{TM} + S_{clc(parity\_of\_unary\_numbers).v}$
- $S_M(n) = n + 2 \in O(n)$
- $T_M(n) = n \in O(n)$ □

4. 0-State Universal Turing Machine: the Simplest Computational Structure

We will prove that the *structural complexity* of TM can be reduced only implementing it as a Universal Turing Machine (UTM).

Early theoretical studies where devoted to reduce the number of states of the finite automaton, that control UTM, with a minimal increasing of the number of symbols in the alphabet I [3]. But we believe that the more important thing is to reduce the structural complexity of UTM. In this respect we will present the simplest UTM built only with simple, recursive defined sub-systems. Some of them are small & simple, others are "infinite" & simple circuits. We start with an n-state UTM, we continue showing how this machine can be transformed in a 0-state UTM, and we end by defining more clearly the meaning of the term **information** in the computation process.

4.1. An n-State UTM

The problem leading to UTM is to define a machine whose structure can remain unchanged when the executed function changes. In this case we need a machine with:

- an abstract representation for the needed TM, as a string of symbols stored in the memory
- an automaton, useful for all computable functions, that "understands" and "executes" by *interpretation* the abstract representation of any automaton associated to a TM stored on the tape.

Interpretation is a process that acts on a string encoded representation of an abstract machine, to emulate the behavior of that machine. It allows us to deal with *representations* of machines rather than with the machine themselves.

Let be a TM called M with the initial content of the tape T: $M(T)$. An interpreter of $M(T)$ will be the TM

$$U(< e(M), T >)$$

where $e(M)$ is the string that describes the behavior of the FA of the TM called M. On the tape of the TM called U there is the description of M and the string, T, to be processed by M.

Definition 5. An *UTM* is a TM, $U(< e(M), T >)$, that has a finite automaton that interprets any TM's description, $e(M)$, stored in the same memory with T, the string to be processed.

In order to implement an UTM we start from the fact that the transition function f from the state q_i **can be reduced** to a set of pairs having the following form:

$$f(q_i, x) = (q_j, y, c_l)$$

$$f(q_i, \neq x) = (q_k, z, c_m)$$

where: $q_i, q_j \in Q$, $x, y, z \in I$, and $c_l, c_m \in \{U,\ D,\ -\}$ with the following meaning:

```
if currently accessed symbol is $x$
  then the next state is $q_j$
       the stored back symbol is y
       the access head command is $c_l$
  else the next state is $q_k$
       the stored back symbol is z
       the access head command is $c_m$
```

Such a pair will be associated with each state of the automaton. Therefore, any state can be represented as a string of nine symbols having the form:

$$def(q_i) = \&q_i x q_j y c_l q_k z c_m$$

where & is a symbol indicating the beginning of the defining string associated with the state q_i.

A TM can be completely described by specifying the function f, associated to the *random structure* of the machine, using the above defined strings to compose a "program" P as a stream of $def(q_i)$ strings.

Example 5. Revisiting **Example 3** results the correspondent representation of the program P (the string $e(M)$):
$P = \{def(q_0),\ def(q_1),\ def(q_2),\ def(q_3),\ def(q_4)\}$,
$P = \&q_0{-}q_1 0 U q_1 0 U \&q_1 1 q_2 1 U q_3 0{-}\&q_2 1 q_1 1 U q_4 0{-}\&q_3{-}q_3 0{-}q_3 0{-}\&q_4{-}q_4 0{-}q_4 0{-}$ □

The tape of UTM will be divided in two sections, one for T, the string to be processed by the machine M, and another containing the description P of the machine M. The content of the tape will be

$$\ldots \#P@T\# \ldots$$

where:

- @ is a special symbol which delimits the "program" from the "data"
- the string $P \in (I \cup Q \cup \{D,\ U,\ -\} \cup \{\&\})^*$ is the "program" that describes the algorithm
- the string $T \in I^*$ represents the "data".

The automaton of UTM "knows" how to interpret the string P in order to process the string T. Its CLC used to compute the transition function of UTM is the only random physical structure. The questions are: what is the way to minimize the structure of this CLC? Is it possible to reduce to zero the dimension of this CLC? Preliminary answers: with a new more efficient definition for $def(q_i)$, for the first question; with a new *more efficient & lucky* definition for $def(qx_i)$, for the second question.

4.2. A 0-State UTM

For simplicity, we use an **equivalent** TM having two "heads", one for reading P and one for reading/writing T. The data access "head" and the program access "head" are implemented as two counters (in the new representation of TM). This machine has an actual implementation using a RAM with two ports, one for *read* the program and another port for *read/write* from/in the data space.

The previous form of P must be *translated* in P' that uses for each state, instead of the string $\&q_ixq_jyc_lq_kzc_m$ stored in 9 successive memory cells, the following form, call it `instruction`, as a single entity stored in one memory cell:

```
instruction = {test\_symbol, yes\_inc, yes\_symbol, yes\_com,
                             no\_inc,  no\_symbol,  no\_com }
```

Any instruction is interpreted as follows:

```
if current accessed symbol is test_symbol
  then move program access head yes_inc positions
       store yes_symbol in data space
       data access head command is yes_com
  else move program access head no_inc positions
       store no_symbol in data space
       data access head command is no_com
```

where: `yes_inc` and `no_inc` are signed integers. Obviously, each program P has a correspondent P' form.

Example 6. Looking back to the previous example, the string P, with the next form:

$\&q_0 - q_10Uq_10U\&q_11q_21Uq_30 - \&q_21q_11Uq_40 - \&q_3 - q_30 - q_30 - \&q_4 - q_40 - q_40-$

stored in 45 successive cells can be translated in a string of P' type, stored in five larger successive memory cells. If the field `com` is codded as follows: `nop = 00, up = 01, down = 10`, results the P' form of P:

```
instruction_0 = xx, +1,00, 01, +1,00,01
instruction_1 = 01, +1,01, 01, +2,00,00
instruction_2 = 01, -1,01, 01, +2,00,00
instruction_3 = xx, +0,00, 00, +0,00,00
instruction_4 = xx, +0,00, 00, +0,00,00
```

□

The structure of UTM using `instruction` to define the transition from the state q_i is presented in Fig. 3, where the counters are detailed and some simple combinational circuits are added. The program P' is stored in RAM starting with the address n where the description of the state q_0, `instruction_0`, is loaded. In the following memory cells are stored the next instructions: `instruction_1`, `instruction_2`, `instruction_3`, `instruction_4`. The string to be processed, in our case the 1-ary represented number, is stored starting with the address m, greater than the address containing `instruction_4`. The initial value of the program address counter (ADD & $R1$) is n, and for the data address counter (Inc/Dec & $R2$) the initial value is m. The multiplexer MUX selects (see Fig. 3), according to the output of the comparator $Comp$ (whose output is 1 if its two inputs are equal), the appropriate values for:

- the value (`yes_symbol` or `no_symbol`) to be written in RAM to the current address generated by Inc/Dec & $R2$ (the value to be written on the tape in the current cycle of the simulated TM)
- the signed number to be added to the current value of "program counter" implemented by ADD & $R1$ (the relative address of the cell that stores the description of the next state: the next instruction)
- the command applied to the counter (Inc/Dec & $R2$) that points in the data part of the memory.

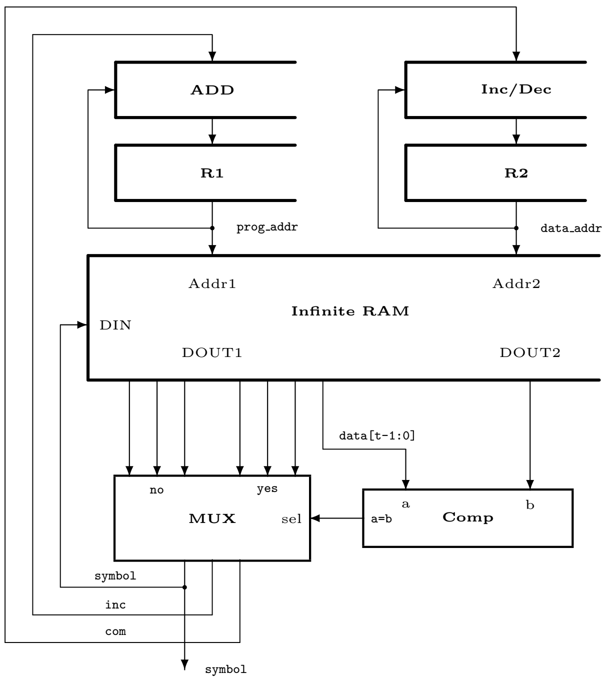

Fig. 3. A 0-state UTM. The structure of a recursive defined UTM contains only simple, small or "infinite" circuits. DOUT1 generates in each clock cycle the instruction `instruction_i`, corresponding to the current state, q_i, *executed* by UTM. DOUT2 generates in each clock cycle the symbol currently accessed by the TM that is *executed*.

The HDL description is the following:

```
module universal_turing_machine(symbol, reset, clock);
    parameter   n = 10, // address size
                t = 8;  // data word size
    output  [t-1:0] symbol;
    input           reset, clock;
    reg     [(2*t+3*n+3):0] mem[0:((1 << n) - 1)];
    reg     [n-1:0] data_addr, prog_addr;
    wire    [(2*t+3*n+3):0] data;
    wire    [t-1:0] test_symbol, yes_symbol, no_symbol;
    wire    [n-1:0] yes_inc, no_inc, inc;
    wire    [1:0]   yes_com, no_com, com;
    // ADDRESS GENERATORS
    always @(posedge clock)
        if (reset)  prog_addr <= 0;
            else    prog_addr <= prog_addr + inc;
    always @(posedge clock)
        if (reset)  data_addr <= 1 << (n - 1);
            else    data_addr <= com[1] ?   data_addr     :
                                 com[0] ?   data_addr + 1 :
                                            data_addr - 1 ;
    // MEMORY
    always @(posedge clock) mem[data_addr] <= symbol;
    assign  data = mem[data_addr],
            {test_symbol,
             yes_inc, yes_symbol, yes_com,
             no_inc,  no_symbol,  no_com } = mem[prog_addr];
    // COMPARATOR & MULTIPLEXOR
    assign {symbol, inc, com} = (test_symbol == data[t-1:0])   ?
                                {yes_symbol, yes_inc, yes_com} :
                                {no_symbol, no_inc, no_com}    ;
endmodule
```

The resulting UTM is characterized by **a strong functional segregation between the simple** physical structure **and the complex** symbolic structure, $e(M)$, stored in its memory. Indeed, the UTM as a machine has *no random components (circuits).* The randomness of the initially defined UTM is totally shifted into the content of the memory, where a "random" string describes a certain TM. Instead of random circuits (the CLC) and a random string of symbols, we have now only the random string of symbols (the stream of instructions). The *hard* random structure of the circuits is completely converted into the *soft* random structure of the string describing the function executed by the emulated TM.

In this last UTM version the *interpretation* of $e(M)$ is substituted with the *execution* of a differently organized $e(M)$. The interpretation is a controlled process that involves a finite automaton, while the execution is made by simple circuits (in this case, the combinational circuits like selectors, comparators, simple arithmetic circuits). Because *Comp*, MUX, ADD, *Inc/Dec* are simple circuits (see Fig. 3) they are able only to execute a code, not to interpret it. The size of their definition does not depend on the dimension of the sets I and Q. **Removing the finite automaton** from the structure of UTM the machine substitutes the *interpretation* of P with the *execution* of P'.

Proposition 2. *The 0-state UTM is a simple machine.*

Proof. There is no random part in UTM because **the finite automaton is removed** and the interpretation is substituted with the direct execution using simple circuits. □

The computation M performed by the UTM on a data string of n symbols by a program, P, stored in m p-bit memory cells is characterized by the following three parameters:

- C_M: the **complexity** of computation M, computed as the sum between the size of the Verilog module `universal_turing_machine` and the size of the memory used to store the program:

$$C_M = S_{universal\_turing\_machine} + p \times m = C_{UTM} + p \times m$$

- $S_M(n)$: the **size** of memory used to perform the computation M
- $T_M(n)$: the **time**, expressed in the number of clock cycles used to complete the computation M.

In the constant value of C_M the size of the program P is correlated with the complexity of UTM, C_{UTM} executing it. The size of the program P can be reduced if the complexity of UTM is increased. For the simplest UTM we expect to have the biggest program size.

4.3. Information Based Computation

In [2] **general information** is defined as a *meaningful syntactic structure*. Starting from this general definition the concept of information in the general theory of computation becomes a very well stated concept.

Definition 6. Let be the interpreter

$$U(< e(M), T >)$$

of TM called M with the initial data T, where $e(M)$ is the stream of symbols describing the behavior of TM. We call $e(M)$ the *information* stored on the "tape" of the machine U, and T the *data* to be processed by U which interprets the information $e(M)$.

The content of $e(M)$ **acts** when it is interpreted by an UTM. Therefore, computation means: a *process controlled by information.* Two distinct cases are emphasized:

- *interpreting* information in the n-state UTM: the FA of the machine $U(< e(M), T >)$ "knows" how to use (to interpret) $e(M)$, which thus **acts** in order to process T;
- *executing* information in 0-state UTM: the content of $e(M)$ **acts** mediated only by simple circuits in order to process T.

There is a very important difference between $e(M)$ and T. The stream $e(M)$ has a very precise meaning: it describes an algorithm. The stream T usually has no meaning for the way the machine behaves processing it. T is a sort of "passive" stream of symbols, while $e(M)$ is an "active" stream of symbols.

Generally speaking, **information acts by its meaning**, which is interpreted or executed on a physical support. In the particular case of computing this physical support is an UTM with n or 0 states. Thus, execution can be seen as a limit case of interpretation.

Usually we refer to computers, asserting that computers *process information.* More correct is to phrase: computers *process data by information.* Thus the distinction between the symbolic structure of *data* and the symbolic structure of *information* becomes clear: the second acts, mediated by a physical structure, on the first.

5. Conclusions

The reasons of reducing the number of states of a UTM now, in *Giga Gates per Chip Era*, are completely different from the reasons to make the same thing 50 years ago, in the *no-chip era.* Now we are driven by the problems generated by unmanageable complexities, while a half century before only pure theoretic issues pushed ahead a similar research. (The solution proposed by Claude Shannon in 1956 increased very much the complexity of the combinational circuit of the 2-state finite automaton.)

1. The complexity of a computation performed by a 0-state UTM depends only by the algorithmic complexity of the string $e(M)$. The structural complexity of the TM is *completely converted* in the complexity of the symbolic description of the computation that will be executed (not interpreted) in 0-state UTM. A *hard* complexity is converted into a *soft* complexity even for the problem having solutions with a less powerful machine than a TM.

2. The present day technological evolutions offer the possibility to design machines having very big sizes, but the complexity of this big structures can not follow this tremendously accelerated process. **The complexity can not grow as fast as the size grows**. Else, because of their too big complexity machines become uncontrollable, more, they become unutterable. Let us imagine a complex circuit containing 10^9 gates! In order to be "accepted" in an automatic design environment we have no solution to "express" it. The simplicity of the 0-state UTM supports our steps toward using the new technologies for building very big and powerful machines maintaining their complexity at a manageable level. Thus, we have a theoretical support

to declutch the complexity of the computation by the complexity of the machine. Is this a good or a bad way? This is another issue, out of our purpose now.

3. The **segregation** between the *simple machine* and the *complex symbolic description* of computation is the main process helping us to avoid the big apparent complexity of computation. In this way we have the chance to reach the actual complexity, preserving the *competence.* In order to improve the *performance* we must add the *architectural approach*, as a process that offers a good balance between the physical structures and the symbolic structures involved in the computation process.

4. Any computation done by a TM is characterized by *complexity* (the size of description of the associated TM), *size* (the amount of memory used to make the computation) and *time* (the number of clock cycle used by TM to complete the computation). For a UTM, the complexity is given by adding the complexity of UTM with the size of the program.

5. The complexity of computation must take into account both, the complexity of the machine performing the computation and the size of the program used to describe the computation.

6. Information is defined as the syntactic correct symbolic structure acting by its meaning defined in the context of an UTM. The meaning carried by information is *interpreted* in n-state UTM's or it is *executed* in 0-state UTMs. The active *information* must be differentiated by the passive *data*, both stored in the "infinite" memory of an UTM.

7. The distinction between information and data is contextual. Data can be seen as information in the context of another computation (a compiler generating an executable code works on data which will become information once loaded as a program on the same machine).

Because computation implies simple machines that interpret (execute) complex information we have the optimistic view of accepting the exponential growth of the *hard machines*, while the complexity remains to be managed by modelling the *soft information.*

References

[1] CHAITIN, G., *Algorithmic Information Theory*, IBM J. Res. Develop., July 1977.

[2] DRĂGĂNESCU, M., *Information, Heuristics, Creation*, in Plauder, I. (ed): *Artificial Inteligence and Information Control System of Robots*, Elsevier Publishers B. V. (North-Holland), 1984.

[3] SHANNON, C. E., *A Universal Turing Machine with Two Internal States*, in *Annals of Mathematics Studies*, no. **34**: *Automata Studies*, Princeton Univ. Press, pp. 157–165, 1956.

[4] ŞTEFAN, G., *Circuit Complexity, Recursion, Grammars and Information. Multiple Morphisms*, Ed. Transilvania University of Braşov, 1997.

[5] ŞTEFAN, G., *No-State Universal Turing Machine*, communication at *Fundamentals of Computation Theory*, Iaşi, 1999.

[6] ŞTEFAN, G., *Loops & Complexity in Digital Systems. Lecture notes on digital design in the Giga-Gate per Chip Era*, partially posted at `http://arh.pub.ro/george/`

[7] ŞTEFAN, G., 0-*State Universal Turing Machine*, in `http://arh.pub.ro/george/0stateUTM.htm`

[8] ŞTEFAN, G., *Functional Information* in `http://arh.pub.ro/george/functionalInformation.html`