Bit Copying: The Ultimate Computational Simplicity

Oleg Mazonka

Defence Science and Technology Organisation
Edinburgh SA 5011 Australia
mazonka@gmail.com

A computational abstract machine based on two operations, referencing and bit copying, is presented. These operations are sufficient for carrying out any computation and can be used as the primitives for a Turing-complete programming language. The interesting point is that computations can be performed without logic operations such as `AND` or `OR`. The compiler and emulator of this language with sample programs are available on the internet.

1. Introduction

In a quest to build an imperative language with the smallest possible number of instructions, several one instruction set computer (OISC) languages have been invented. One example, the ultimate RISC architecture [1], utilizes the single instruction copy memory to memory. Complex behavior of such a machine is achieved by mapping the machine registers onto memory cells. For example, a memory cell with the address 0 is the instruction pointer, so copying to this address effectively realizes an unconditional jump. Arithmetic operations are also achieved by using special registers in memory that perform more complex operations at the hardware level.

Another example, Subleq [2], does not have memory mapped registers. Its computational power is based on program self-modification and a sufficiently complex instruction set. The *abstract machine* is defined as a process working on an infinite array of memory cells with each instruction having three operands. The processor reads three operands from the memory, subsequent cells `A B C`, subtracts the value of the cell addressed by `A` from the value of the cell addressed by `B`, and stores the result in the same cell addressed by `B`. If the result is less than or equal to zero, the execution jumps to the address `C`, and the processor reads the next instruction from there; otherwise the next three operands are read from memory. This language is proven to be Turing-complete. There are a few variations of this language that are similar in principle. A compiler from a simple C-like language has been written to compile programs into Subleq processor code [3]. Attempts to reduce the complexity of the atomic operation have been undertaken. For example, Rojas [4] proves that conditional branching is not necessary for universal computation given the ability of code self-modification.

Although OISC languages have just one instruction, the instruction does a number of manipulations or computations under the hood. Hence, there is a question: which language has the simplest instruction and is it possible to make a language with a simpler instruction?

Another interesting question relates to logic operations. It is commonly known that classical computations are usually done using bit logic operations: `AND`, `OR`, `XOR`, and `NOT`. These operations are neither a complete set nor a minimal set required for computation. `OR` and `XOR` can easily be expressed via `AND` and `NOT` and vice versa. However, it is commonly assumed that at least `AND` or `OR`-like operations are needed to make real computations. It is not possible to combine `OR` and `XOR` operations to erase a single bit. Therefore, they are unable to produce classical computations. Hence, a question arises: is it possible to make programmed computation without using logical operations like `AND` and `OR`?

2. Referencing as a Computational Operation

Surprisingly, `OR` and `XOR` reversible operations can produce irreversible results if they are used in combination with referencing. In Table 1 the first row has the initial three bits. The second row has the same bits with one of them inverted (`NOT` operation applied). The inverted bit is referenced by all three, as the index of the bit equal to its binary representation taken modulo 3. It can be seen that two initial states (001 and 010) produce the same final result (011), making this entire operation irreversible.

000	001	010	011	100	101	110	111
100	011	011	111	110	100	010	101

Table 1.

A machine, similar to register machines described by S. Wolfram [5], can be realized by using a continuous process of bit inversion on the same set of bits. For example, a 3-bit machine produces a sequence (000) (100) (110) (010) (011) (111) (101) (100) ....

Figure 1 shows Wolfram diagrams for 2-, 3-, 4-, 5-, 6-, and 7-bit machines. On the right side of each diagram bits are represented as squares with dark for 1 and white for 0. On the left side, a small square shows the interpreted value of the bits—the address of the next bit to be inverted. The address is calculated as the binary representation of some integer taken modulo the number of bits.

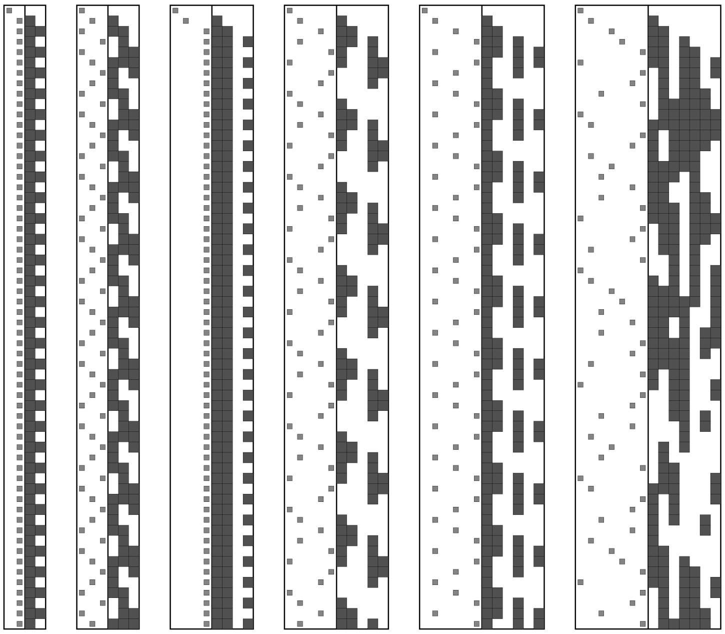


Figure 1.

Taking the binary value and modulo may seem like complex operations. However, there is a simple model simulating such behavior. Allow bits to be positioned in a circle with an arrow pointing to any bit. Each bit is assigned a rule for how to rotate the arrow if its value is 1. At each step the arrow rotates according to the rules and the values of all bits, and at the end of the step the bit pointed to by the arrow is inverted.

Figure 2 shows the diagrams for 17-, 18-, and 19-bit machines similar to those shown in Figure 1. The picture is scaled 1 to 10 in the vertical direction and 1000 steps are shown.

Table 2 shows the size of the loop (the size of the pattern on the diagram) that the machine eventually enters when started with the initial zero values of all bits. Bigger loop sizes correspond to more complex behavior of the model.

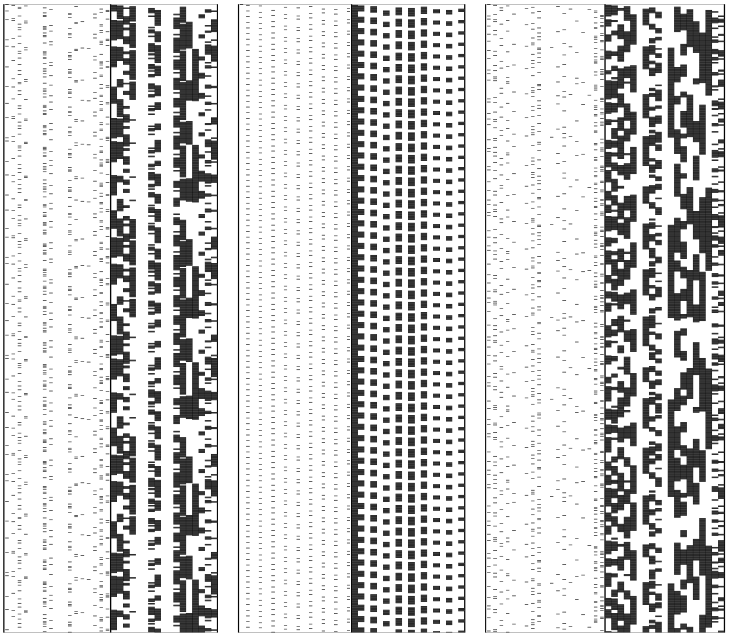


Figure 2.

Bits	Loop Size	Bits	Loop Size	Bits	Loop Size
3	6	13	66	23	18 812
4	2	14	50	24	6
5	8	15	162	25	48 000
6	6	16	2	26	544
7	50	17	346	27	54
8	2	18	18	28	62
9	18	19	1700	29	128 116
10	10	20	10	30	30
11	112	21	12	31	635 908
12	6	22	118	32	2

Table 2.

In another example, shown in Table 3, the second row has one bit modified by the formula

$$(\texttt{p[A] XOR } p\texttt{[B]}) \rightarrow \texttt{p[A]},$$

where A is the value of the first two bits, B is the value of the last two bits, and p[] is the bit taken by the index.

0000	0001	0010	0011	0100	0101	0110
0000	0001	1010	1011	1100	0001	...

Table 3.

The operations applied to bits are reversible, but referencing makes the whole process irreversible. Even if it feels like this process could do calculations, it is still difficult to bring it into play to make a framework that is able to do desirable computations.

3. Bit Copying Language

It turns out that by taking the converse approach—that is, to combine a bit copying operation, which is irreversible, with referencing—the programmable computation is possible at the bit level.

A bit copying instruction always erases one bit. On first inspection it would seem that the entire amount of information in the system is being forever reduced until no changes are made. However, this is not the case. For example, in the process *forever do* ($a \to b$, $c \to a$, $b \to c$), where a, b, and c are bits, and arrow means copy, the bits will circle in the loop forever. This would be interpreted as a steady but not static state. To do something more interesting, let us add a meaningful representation to each collection of bits by associating an address with each.

Let us define the imperative language, in which the abstract machine operates on an array of memory bits addressed from 0. Bits are grouped into *words* of a particular size (memory cells). For examples of 8-bit words see Table 4.

Memory	01010101	00001111	10101010	11001100	00110011	...
Address	0	8	16	24	32	40

Table 4.

Each instruction consists of three operands: `A B C`, where each operand is one word. The instruction copies the bit addressed by `A` into the bit addressed by `B`, then jumps the process control to address `C`. The operand `C` is read after the bit copying is done. This allows the instruction to be self-modifiable (even though just one bit can be modified at a single point in time).

Since each word represents a bit address in the memory, the exact way of address interpretation can be left undefined without sacrificing the concept. In this implementation, however, it is assumed to be little-endian binary bit representation.

Another undefined feature that is up to the implementation is how to consider grouping memory bits into words: physical—all words are aligned on memory cell size address, or logical—words are grouped counting from the current address. In the second case, the operand C is allowed to specify an address of any bit, not necessarily aligned to the memory cell.

This sole instruction does not do any more than copy a bit from one place to another, and yet this simple single instruction is enough to make the language able to execute a preprogrammed sequence of operations. The abstract machine obviously does something more than copying bits: it references bits and transfers the process control to the next address of execution. However, this work may not need logic operations AND or OR, and is done outside of the execution model, which means it can be emulated by the same bit copying process.

4. Assembly

To simplify the presentation of the language instruction set, we use the following assembly notation. Each word is denoted as L:V'*x*, where L is an optional label serving as the address of this memory cell, V is the value of this memory, and *x* is the optional bit offset within the word (memory cell). Each instruction is written on a separate line. For example:

```
A'0 B'1 A
A:18 B:7 0
```

gives two instructions. The first instruction copies the lowest bit of cell A (whose value is 18) into the second bit of cell B (whose value is 7), then jumps to the instruction addressed by A, which is the next instruction. After the first instruction is executed, the value of cell B is changed to 5. For example, in 8-bit word memory these two instructions are

```
24 33 24 18 7 0.
```

If the bit offset is omitted, it is considered to be 0. So, A is the same as A'0.

If the operand C is absent, it is assumed to have the value of the next cell address, that is,

```
A B
```

is the same as

```
A B C
C: …
```

which is the same as

```
A B ?
```

Question mark (`?`) is the address of the next memory cell, that is, the address of the first bit of the following word. Let us denote (`n?`) as a multiple to cell size counted from the current position. So, (`0?`) means the address of this position; (`1?`) is the same as (`?`) and is the next cell; (`2?`) is one after the next cell; and (`-2?`) is one before the previous cell. For example, the instruction

```
A B -2?
```

is the same as

```
C:A B C
```

and is an infinite loop (assuming that it does not modify `C`), as the bit referenced by `A` is copied to the bit referenced by `B` and the control is transferred to the address of the cell `C:A`, which is the beginning of the original instruction. Remember that `A` is the value and `C` is the address of `C:A`.

Given an assembly text, we also need an environment to run a program that is written in this language. Therefore two steps are necessary: (1) Compile the text into binary code as an array of bits to form instructions. (2) Execute the binary coded instructions on the abstract machine. A program called an assembler does the first step, and an emulator can do the second.

5. Macro Commands

To make a program description shorter and more readable, let us define a macro substitution mechanism as in the following example.

```
.copy A B
...
.def copy X Y
X'0 Y'0
X'1 Y'1
...
X'w Y'w
.end
```

The first line is the macro command that is substituted by the macro definition body starting with "`.def`" and ending with "`.end`". The name after "`.def`" becomes the name of the macro and all subsequent names are formal arguments to it. After macro substitution, the code becomes:

```
A'0 B'0
A'1 B'1
...
A'w B'w
```

Here, w is the index of the highest bit and is equal to the size of the memory cell minus one. Let us denote the word size as W, $W = w + 1$.

Two other useful macro definitions are shift and roll. `shiftL` shifts bits in the memory cell by 1 from lower to higher, and the lowest bit is set to 0. It is the same as arithmetic multiplication by 2, or the C programming language operation `"<<=1"`.

```
.def shiftL X : ZERO
X'(w-1) X'w
X'(w-2) X'(w-1)
…
X'1 X'2
X'0 X'1
ZERO X
.end
```

`ZERO` is defined at some place as `ZERO:0`. It appears after the colon at the end of the macro definition argument list to signify that this name is defined outside of the macro definition. This is necessary because the assembler tries to resolve all names within the body of a macro definition or to tie them to the formal arguments. Note that the last instruction copies the lower bit of the `ZERO` memory cell to the lower bit of `X`.

`shiftR` is the same as `shiftL` but works in the opposite direction and corresponds to integer division by 2, or the C shift operator `">>=1"`.

Roll macros are similar to shift macros with the exception that they copy the end bit back to the front. They can be defined in terms of the shift macro definitions.

```
.def rollR X : TMP
X TMP
.shiftR X
TMP X'w
.end

.def rollL X : TMP
X'w TMP
.shiftL X
TMP X
.end
```

The `TMP` memory cell is a placeholder and is defined in an external library.

The macros copy, shift, and roll are useful, but lack the logic to be able to perform useful calculations.

6. Conditional Jump

Consider the following code.

```
.def jump01 A b
A'b 2?'k
0 J'0
A'b 2?'k
1 J'1
A'b 2?'k
2 J'2
…
A'b 2?'k
(w-2) J'(w-2)
A'b 2?'k
(w-1) J'(w-1)
A'b 2?'k
w J'w J:0
.end
```

The offset k is defined such that $2^k = W$. Since a word is an address of a bit in the memory, there are k bits corresponding to the offset within a word. The rest of the word's bits specify the address of a memory cell. For example, for a 32-bit word k is 5 because modifications in the sixth bit and higher change the address of the memory cell, but not the offset inside the memory cell. Note that when writing to the offset, k updates the sixth bit if $k = 5$.

The first line moves bit `b` of memory cell `A` to the k^{th} bit of the first operand of the next instruction. After this is done, the first operand of the next instruction is zero or equal to W. The next instruction copies the value of the first bit of the cell addressed as either 0 or W to the cell labeled `J`—which is the last memory cell in this list of instructions, and which is the address the process control will go to after the last instruction is executed. The subsequent lines [`(A'b 2?'k)(1 J'1)`] copy the second bit to cell `J`, and so on.

When the last bit is copied, cell `J` holds the same value as the cell with address 0 (the first word in the memory) or the cell with address W (the second word in the memory) depending on whether `A'b` was 0 or 1.

By marking the first two memory cells in the program as special, in the sense that they can be used only for this operation, it is possible to write a generic test for a particular bit.

```
Z0:0 Z1:0
.def test A b B0 B1 : Z0 Z1
.copy L0 Z0
.copy L1 Z1
.jump01 A b
L0:B0 L1:B1
.end
```

This code defines a macro that tests bit `b` of memory cell `A` and jumps to either address `B0` or `B1` if the bit is 0 or 1 correspondingly.

Testing a bit is a core requirement of all the higher-level computations described later. In most of these cases testing a bit involves checking the lowest or highest bit in the word:

```
.def testL A B0 B1
.test A 0 B0 B1
.end
.def testH A B0 B1
.test A w B0 B1
.end
```

7. Arithmetic

One of the most basic operations, which will be required for definitions of other more complex constructions, is the increment operation. To increment a memory cell A, the roll, shift, and test macros can be combined in the following way.

```
          .copy ONE ctr

begin:  .testL A test0 test1

test0:  ONE A rollback
test1:  ZERO A
          .testH ctr next rollback

next:  .shiftL ctr
          .rollR A
          0 0 begin

rollback: .testL ctr roll End

roll:  .shiftR ctr
          .rollL A
          0 0 rollback

          End:0 0
          ...
          ctr:0 0
```

The first line initializes counter ctr to 1. The lowest bit of the operand A is swapped, and the operand and counter are rolled until either the operand bit is zero or the counter bit 1 reaches the highest bit position, which means that all w bits of the operand were processed. After this the operand can be rolled back to the original bit position.

In the code ZERO and ONE are defined as ZERO:0 and ONE:1. The instruction 0 0 label is used as an unconditional jump to address label. It copies the first bit in the memory to itself, and does not change its value.

Addition can be defined in a similar way with the exception that there are four operands. These are in order: first number, second number, result, and the adder (for passing over an extra bit). The lowest bits of the adder and first and second numbers are added, giving two bits, one of which goes to the lowest bit of the result, and the other goes to the second bit of the adder. Then, all four operands are rolled and the process continues. Code for the addition command `add` is given in Appendix B.1.

Subtraction can simply be defined as

```
.def sub X Y Z
.inv Y
.inc Y
.add X Y Z
.end
```

where `inc` is increment, `add` is addition (`Z=X+Y`), and `inv` is the inversion operation, which simply inverts all bits in a memory cell. The inversion operation code is simpler than increment and is given in Appendix B.2.

8. Process Control and Pointers

The following definitions can be used to test for particular values of variables.

```
.def ifeq X Y yes no
.sub X Y Z
.ifzero Z yes no
Z:0 0
.end

.def ifzero Z yes no
.testH Z cont no
cont: .copy Z A
.inv A
.inc A
.testH A yes no
A:0 0
.end

.def iflt A B yes no
.sub A B Z
.testH Z no yes
Z:0 0
.end
```

The first macro `ifeq` checks if both arguments are equal to each other by testing the result of the subtraction. The second macro `ifzero` tests whether its argument is equal to zero. This is done by testing the highest bit (negative value); if it is zero then negate the argument (in a simple binary signed representation inversion and increment produce the same result as negation) and test the highest bit again. Note that the argument is copied before it is negated because it should not be changed. The third macro `iflt` tests if the first argument is less than the second.

To write the classical "Hello, World!" program by iterating a pointer over an array of cells, the tricky operation of pointer dereferencing needs to be defined. Consider the following program.

```
      Z0:0 Z1:0

start: .deref p X
      .testH X print -1
print: .out X
      .add p W p
      0 0 start

      p:H X:0
      H:72 101 108
      108 111 44
      32 87 111
      114 108 100
      33 10 -1
```

The label `H` is the address of a string holding the ASCII code for "Hello, World!" followed by the end-of-line sentinel. `p` is a pointer—a cell initialized with the address of the string.

The first instruction does not do anything since it copies the bit addressed 0 to itself. It is necessary because the conditional jump (which uses the first two words of the memory) is part of other macro commands. The next command dereferences `p` by copying the value of the cell, whose address is stored in `p`, into cell `X` (this operation is discussed below). Check if `X` is negative. If it is, go to the address (`-1`), otherwise continue execution with the next line. The address (`-1`) is special because we assume that the program halts if control is passed to it. In fact, this is similar to how halt is defined in other OISC languages, for example, Subleq [2]. The next line prints the ASCII character in cell `X`. (The specific implementation of printing is discussed in Section 10.) If `X` was not negative, the pointer `p` has not reached the end of the array and still points to a valid array element. The pointer is incremented by the size of the memory cell and this process is continued until the halting instruction is executed.

It is possible to copy a memory cell referenced by another memory cell by setting up an iterative instruction with the source and target addresses. The instruction is repeated W times, incrementing the addresses each time until the whole word is copied. An example is given in the following code.

```
        .copy ONE ctr
        .copy P A
        .copy L B

  begin:  A:0 B:0
        .testH ctr next End

  next:  .shiftL ctr
        .inc A
        .inc B
        0 0 begin

        End: …
        L:X ctr:0
```

This block of code performs the same task as the C programming language statement `X=*P`. The counter is prepared as in the previous examples. The pointer value is copied to the first operand of the iterative instruction (`A:0 B:0`), then the address of the result cell is copied into the second operand of the iterative instruction. Now, the iterative instruction is executed W times with each execution incrementing the addresses—the values of the operands.

This approach can be used for copying a value into a memory cell pointed to by another pointer. It is just a matter of swapping the `A` and `B` operands in the iterative instruction.

9. More Arithmetic

Multiplication is quite simple once shift and addition are implemented. (The algorithm does not properly handle negative values; the sacrifice is made for the sake of simplicity.)

```
        .copy ZERO Z

  begin:  .ifzero X End L1
  L1:    .testL X next L2
  L2:    .add Z Y Z
  next:  .shiftR X
        .shiftL Y
        0 0 begin

        End:0 0
```

This code shifts the first multiplier to the left and the second multiplier to the right while at the same time accumulating the result by adding the second multiplier if the lowest bit of the first multiplier is 1. This algorithm is expressed in a simple formula:

$$X \times Y = \begin{cases} X / 2 \times 2\, Y, & \text{if } X \text{ even;} \\ (X - 1) / 2 \times 2\, Y + Y, & \text{if } X \text{ odd.} \end{cases}$$

Division is slightly more complex. Given two numbers X and Y, increase Y by 2 until the next increase gives Y greater than X. At the same time, increase a variable Z by 2, which is initialized to 1. Now, Z holds part of the result of division (the rest is to be calculated further using $X - Y$) and Y, which is done iteratively accumulating all Z. At the last step when $X < Y$, X is the remainder. The code for division is presented in Appendix B.3.

The division operation is imperative for printing numbers as decimal strings. The algorithm implementing this divides the value by 10 and stores the remainders into an array. When the value becomes 0, it iterates backward over the array, printing numbers in ASCII code.

```
      .testH X begin negate

negate: .inv X
      .inc X
      .out minus

begin:  .div X ten X Z
      .toref Z p
      .add p W p
      .ifzero X print begin

print:  .sub p W p
      .deref p Z
      .add Z d0 Z
      .out Z
      .ifeq p q End print

      End:0 0
      ...
      Z:0 d0:48 ten:10
      p:A q:A minus:45
      A:0 0 0
      ...
```

The first portion, labeled `negate`, checks whether the argument is less than 0. If so, then the argument is negated and the minus sign is printed. The second section repeatedly divides the argument and stores the results into the array `A` by a dereferencing operation through the pointer `p`. The command `div` divides `X` by 10, stores the result back to `X`, and the remainder to `Z`. The following command `toref` writes the value of `Z` into the cell pointed to by `p`. This process continues until `X` is zero. In the next portion, marked by the label `print`, the pointer `p` runs back until it is equal to `q`, which is initialized to A, which is the beginning of the array. The command deref copies the value from the array to Z. Then, the ASCII code (48) for character 0 is added and the byte is printed. (It is assumed that the memory cell is a byte not less than eight bits.)

10. Input and Output

At this point, it is possible to write a program that can add, subtract, multiply, divide, iterate, dereference, and jump. To produce an output or receive an input, we have to define what is the output and input. This is called the *pragmatics* of the language or the *environment* of the abstract machine that implements the language. Any definition of input to or output from the abstract machine will be a burden of the environment, or in our case, the emulator of the language (or processor if implemented as hardware). Since the program can copy only bits, it is natural to define a stream of bits as bits copied to or from a particular address. The special address (-1) has already been introduced as the halt address; a program halts if the process control is passed to it. The same address can be used without ambiguity.

```
.def out H
H'0 -1
H'1 -1
H'2 -1
H'3 -1
H'4 -1
H'5 -1
H'6 -1
H'7 -1
.end

.def in H
-1 H'0
-1 H'1
-1 H'2
-1 H'3
-1 H'4
-1 H'5
-1 H'6
-1 H'7
.end
```

Note that only the lower eight bits are copied to and from the word. This is for practical reasons. With this definition it is possible to write assembly code that is independent of word size that will input and output characters as 8-bit symbols.

The emulator keeps buffers of up to eight bits. When the program outputs a bit, it is placed into the buffer. When the buffer is full, a character in ASCII code is flushed to the standard emulator's output from the buffer. When the program copies a bit from the input, it is removed from the input buffer; and if the buffer is empty, a character is read and its bits are placed into the buffer.

Here is a program that prints the first 12 factorials.

```
    Z0:0 Z1:0

start:.prn X
    .mul X Y Y
    .out ex
    .out eq
    .prn Y
    .out eol
    .inc X

    .ifeq X TH -1 start

    X:1 Y:1 ex:33
    eol:10 eq:61 TH:13
```

The macro `prn` is a printing command described in Section 9. Here is the output of the program.

```
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
11!=39916800
12!=479001600
```

This program runs sufficiently quickly on a modern computer with the current implementation of the assembler, emulator, and a collection of macro-defined commands. The word size is 32 bits and the size of the program (after assembling) is about 10 000 instructions.

11. Functions and Library

It is handy to put all macro definitions into one file—a library—and use it with any program. For this, a third keyword command is defined (the other two are `def` and `end`):

```
.include library_file_name
```

Any program using a library is required to include it and start with the line (`Z0:0 Z1:0`). For example, the following code prints “Hi”.

```
Z0:0 Z1:0

.out H
.out i
0 0 -1

H:72 i:105

.include lib
```

If all the command definitions described in this paper were defined as macros, the resulting code for even a simple program would be enormous. This is because macros are heavily defined through other macros, meaning that any command is expanded or inlined at every place it is used. One substitution triggers other substitutions down the hierarchy of macro definitions (see Appendix A). To deal with this problem a command can be defined as the actual code working with its own arguments. Such pieces of code are called *functions*. The macro definition copies the formal arguments to the function's arguments and passes the process control to the function's entry point. The caller code also has to pass its current address to enable the process control to be returned back to the caller code. Once control is returned from the function, the macro definition can copy the result to the arguments if necessary. Obviously, these functions cannot be recursive because there is no concept of stack. However, it does not mean that this concept cannot be introduced; it is implemented for Higher Subleq [3].

For example, the subtraction `sub` macro and function are defined in the following.

```
.def sub X Y Z : sub_f_X sub_f_Y sub_f_RET sub_f
     .copy X sub_f_X
     .copy Y sub_f_Y
     .copy L sub_f_RET
     0 0 sub_f
     L:J 0
     J:.copy sub_f_X Z
.end

:sub_f: .sub_f_def sub_f_X  sub_f_Y
sub_f_RET:0 sub_f_X:0 sub_f_Y:0

# sub internal macro definition
.def sub_f_def X Y : sub_f_RET

     .copy sub_f_RET Return

     .inv Y
     .inc Y
     .add X Y X
```

```
        End:0 0 Return:0

  .end
```

First, there is a macro definition that copies two arguments into global arguments for the function. Next is the global definition of the entry point for the function. The body of the function is defined again through the macro just to keep the internal names outside of the global scope. Ignore for now that a colon precedes the label for the function entry point. The next line defines memory cells for the return value and the two arguments. Two are enough, because the result is passed back inside the first function argument. The next line is a comment. Then there is the body of the function. Its first command is to copy the return address to its last instruction, an unconditional jump back to the caller's code. This copy command can be saved if the outer macro can copy directly to this memory cell.

Functions allow the same code to be executed multiple times instead of replicating code in every place where an operation is required. However, there is a side effect: since the entry point is global (not inside the macro definition) the code for the function will be present in the program even if this function is not used. This is undesirable. Small programs have to remain small after assembling, and should not include the whole library. To cope with this situation an additional mechanism has been added to the assembler. It marks a command, an instruction or a macro command, as conditional if the line begins with a colon. If its name—the label—becomes an unresolved symbol, the command is added to the program. This is why the line (`sub_f`) in the previous example begins with a colon.

12. Conclusion

In this paper two goals have been achieved. One is that another one instruction set computer (OISC) language has been invented that seems to have a much simpler instruction set than the currently known OISC languages; it does not explicitly require logic gates. In February 2010 Marc Scibetta published on his web page a model incorporating bit-inversion and a conditional jump.

The other goal has been to prove that bit copying operations coupled with referencing (or addressing) is enough to build a model that allows Turing-complete calculations. It turns out that the goal is not only possible in principle, but also practically achievable. Simple programs written in this bit-to-bit copying language work within reasonable time and space resource limits. For example, using the emulator on a personal computer, a program can calculate the factorial of 12 within seconds. The program multiplies numbers from 1 to 12, and then uses modular division to print digits of the result.

Only assembly languages with a few library macro commands can be regarded exactly as Turing-complete because they do not have the memory cell size boundary, which limits the address space. Bit copying instructions are loosely Turing-complete, or more precisely, they are of the linearly bounded automaton computational class, which is the class that real computers belong to. A formal proof can be found in [6] where an interpreter of a Turing-complete language DBFI described in [7] is presented. Keymaker (esolangs.org user) argued that the instruction language could be made Turing-complete if addressing is relative, not absolute. It seems that it is possible to redefine the language to use relative addressing, but that is outside the scope of this paper.

The language presented in this paper has been implemented. Its assembler, emulator, and the library can be downloaded from [6].

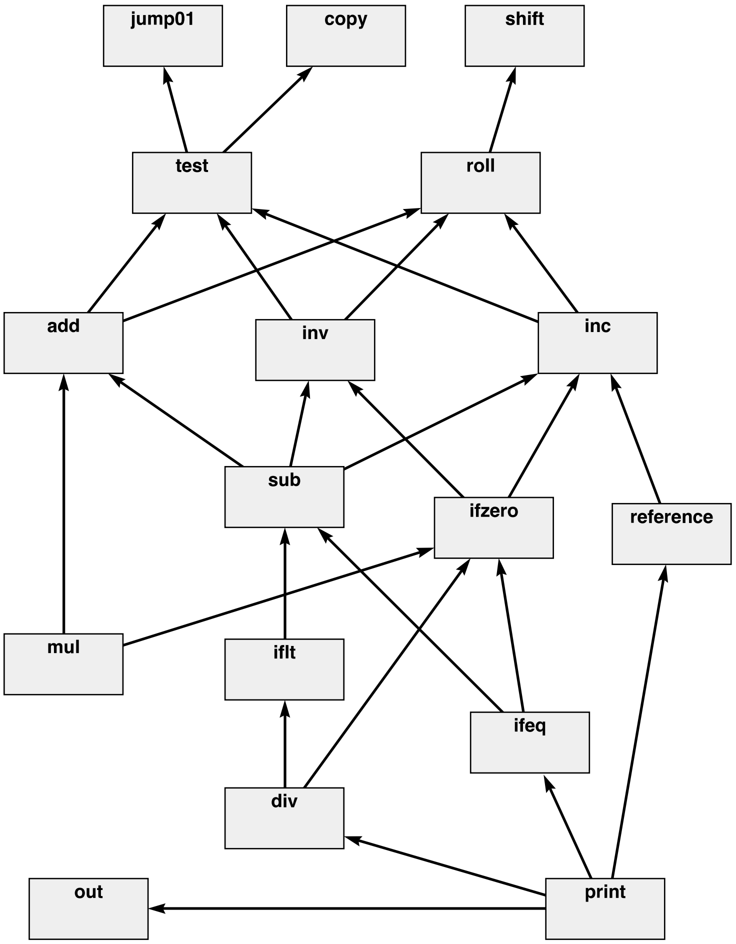
Acknowledgments

I would like to thank my daughter, Sophia Mazonka, for helping me with English grammar. I also would like to thank James Tebneff for valuable comments that greatly improved the clarity of this paper.

Appendix

A.

The following diagram represents dependencies between functions and macros in the library of the current implementation [6]. Direct dependencies, which are also indirect, are omitted. Different implementation algorithms would result in different dependency diagrams, but general dependency levels would be the same.



B.

B.1 .add

This code defines the addition operation as described in Section 7.

```
     .copy ONE ctr
     .copy ZERO adr

begin:  .copy ZERO btr
     .testL adr testx inctestx

inctestx: .inc btr
testx:  .testL X testy inctesty

inctesty: .inc btr
testy:  .testL Y testz inctestz

inctestz: .inc btr
testz:  btr Z
     btr'1 adr'1

     .testH ctr rollcont rollback

rollcont: .shiftL ctr
     .rollR adr
     .rollR X
     .rollR Y
     .rollR Z
     0 0 begin

rollback: .testL ctr roll End

roll:  .shiftR ctr
     .rollL Z
     0 0 rollback

     End:0 0
     …
     ctr:0 adr:0 btr:0
```

The ancillary variable `ctr` is used to count the number of rolls applied to the arguments. The variable `adr` is the adder, which is used for passing over bits to the next bit position. The variable `btr` is the sum of three bits taken from the same bit position of the two summing arguments and the adder.

B.2 .inv

The code inverting bits in one word is straightforward. ctr is, as usual, an ancillary variable.

```
      .copy ONE ctr

begin:  .testL ARG copy1 copy0

copy1:  ONE ARG 4?
copy0:  ZERO ARG

      .testH ctr rollcont rollback

rollcont: .shiftL ctr
      .rollR ARG
      0 0 begin

rollback: .testL ctr roll End

roll:  .shiftR ctr
      .rollL ARG
      0 0 rollback

      End:0 0
```

B.3 .div

Here is the working code to implement the division algorithm described in Section 9. Its arguments are: X—dividend, Y—divisor, Z—result of integer division, and R—remainder.

```
      .copy ZERO Z

      .testH X L1 End
L1:    .testH Y L2 End
L2:    .ifzero Y End begin

begin:  .iflt X Y L3 L4

L3:    .copy X R
      0 0 End

L4:    .copy Y b1
      .copy ONE i1

next:  .copy b1 bp
      .copy i1 ip
      .shiftL b1
      .shiftL i1
      .iflt X b1 rec L5

rec:   .sub X bp X
      .add Z ip Z
      0 0 begin

L5:   .testH b1 next End

      End:0 0
      …
      b1:0 bp:0 0
      i1:0 ip:0 0
```

References

[1] D. W. Jones, "The Ultimate RISC," *ACM SIGARCH Computer Architecture News*, **16**(3), 1988 pp. 48–55.

[2] "Subleq." (Nov 24, 2010) http://esolangs.org/wiki/Subleq.

[3] O. Mazonka. "Higher Subleq." (Aug 17, 2009) http://mazonka.com/subleq/hsq.html.

[4] R. Rojas, "Conditional Branching Is Not Necessary for Universal Computation in von Neumann Computers," *Journal of Universal Computer Science*, **2**(11), 1996 pp. 756–767.

[5] S. Wolfram, "Register Machines," in *A New Kind of Science*, Champaign IL: Wolfram Media, Inc., 2002 pp. 97–102.

[6] O. Mazonka. "BitBitJump." (Sept 2009) http://mazonka.com/bbj.

[7] O. Mazonka and D. B. Cristofani. "A Very Short Self-Interpreter." (Nov 21, 2003) arXiv:cs/0311032v1.