



# URM - Unlimited Register Machine

## A basic C# implementation

2015-10-02

### 1 Abstract

This paper briefly describes the **design choices** of a simple **URM processor** as described in the Computability class. During the process some interesting design issues arised and we are discussing them here.

The design is very simple and the implementation is far from being robust and well tested (this is no production code). Please refrain from bad production code considerations, such as thread safety, performance, fine grained error handling and such comments. We already know.

Funny story is that the model was implemented just to have a mean to test the logic of a simple program P we were writing on paper as a study session. Since we wanted an automated check on our assumptions "is this program returning the expected results?", we decided to implement the model.

By the way P was incorrect, for that matters ... (^\_^);

### 2 The URM model

The URM is defined as a **machine** (the processor) with an **unlimited tape** (the storage memory) that operates on a **program** in order to solve a **problem**. The problem is an unary function :  $N \rightarrow N$ .

#### 2.1 The Tape

The tape has "unlimited" squares (in the software model it is of course limited by the underlying implementation : we used a List<long> having max 4G squares). Each square contains a **non negative integer value** (implemented as a 64 bit long).

The tape ensures the **square value constraints** (for example values  $\geq 0$ ) and exposes two methods only : **GetValue** and **SetValue**. The tape seems "unlimited" to the client because it **expands as needed** when new values are added or retrieved ("touched").

For instance: `tape.Set(5,1000)` (sets the value 5 in the square at position 1000) : the tape silently expands to the 1000<sup>th</sup> position so that the client can use that position.

This is because we want the client to be able to do things such as :

- `z(5)`
- `s(25)`
- ...

... even if the underlying 5<sup>th</sup> and 25<sup>th</sup> squares do not exist : there is no need for the client to declare the existence of a square.

#### 2.2 The Program

The program is an **ordered list of statements**. The URM statement set is limited to four statements but our implementation is meant to be **extendable** : we can easily implement new statements to define a more evolved machine.

The implemented **statements** are the following :

- `z(n)` : the **zero** statement
- `s(n)` : the **increment** statement
- `t(n,m)` : the **transfer** function
- `c(n,m,q)` : the **conditional** statement

We also implemented a non standard URM statement which sets an arbitrary value in a square. More on this on the Initial Conditions chapter.



## 2.3 The Machine

The machine **defines a tape** and **expects a program** as an argument of the execute method.

The machine is **immutable**, which means that the Program Counter, which is supposed to points to the next statement, is not defined as a machine state field. Instead, it is defined on the stack as a local variable of the execute method. This is a design choice and it allows the machine to be thread safe, at least with regards of the program counter so far. We don't care of thread safety here, but immutability is always a good goal.

The lack of the program counter state, results in a problem for the statements which need to alter it during the program execution (i.e. the conditional statement). We have to deal with the problem of defining whose responsibility is updating the program counter. More on this on the program counter chapter.

Since we need to **track the loop condition**, we define a **MaxStep counter** and we state that a machine is not looping if the processing finishes in a number of steps which is limited to MaxSteps.

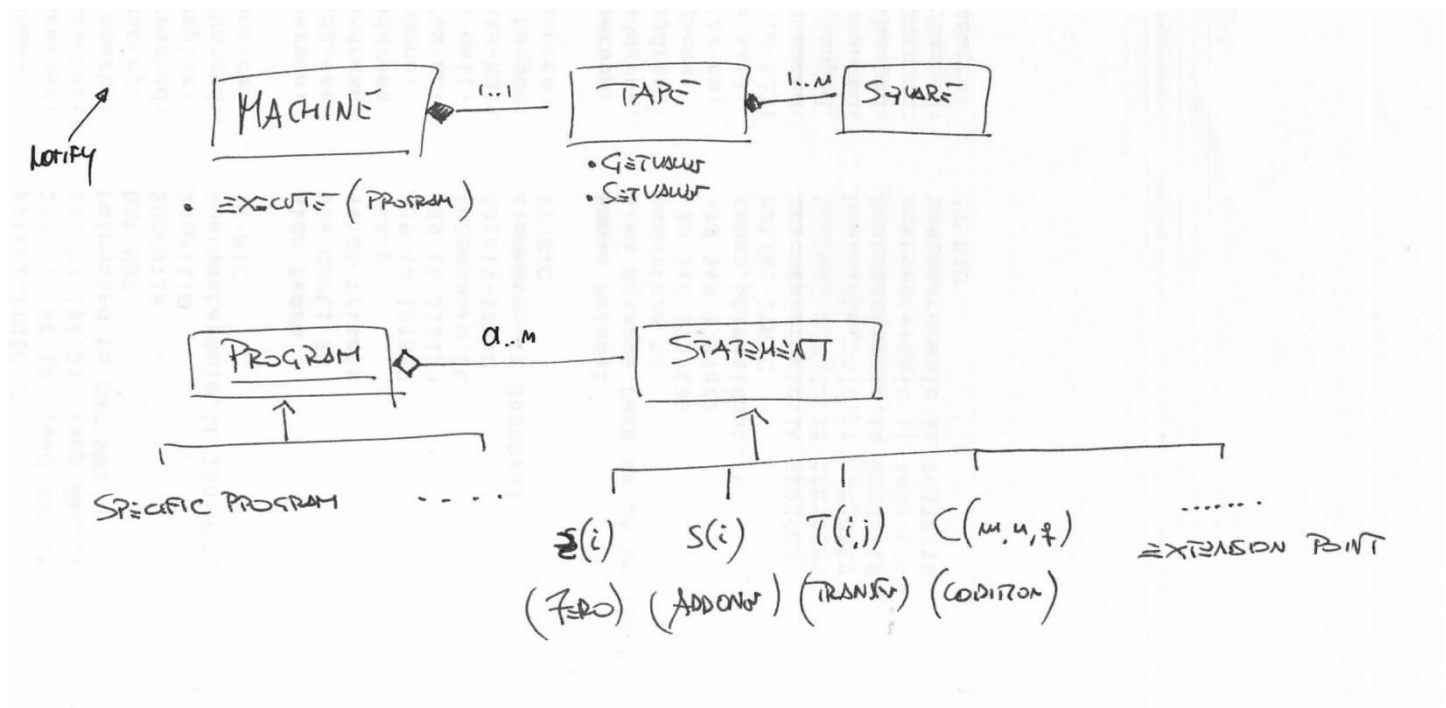
If the loop condition is reached, the machine throws a LoopDetectedException. This is of course a design choice in order to be able to determine if the machine is looping or not (for example, unit tests need to "solve" the halting problem, for the assertions to evaluate).

## 3 The implementation

The URM is implemented in OOP fashion with some functional coding style, such as a tendency to immutability of types. We don't deal here with a detailed code implementation, because it is easier to inspect the attached source code : we just point out some basic information and use cases.

### 3.1 The Class Diagram

The simplified class diagram :



### 3.2 The statements

We implemented the following basic standard URM statements :

- z(n) : CStatementZero
- s(n) : CStatementAddOne
- t(n,m) : CStatementTransfer
- c(n,m,q) : CStatementCondition



We also implemented the following non-standard URM statement :

- set(n, v) : CStatementSet

### 3.3 The program

A concrete program can be coded by inheriting from an abstract program and by defining its statement list, as the following example shows :

```
public sealed class CProgramXMinus1 : CAbstractProgram
{
    // =====
    // F(x) =
    //      : x - 1      when x > 0
    //      : 0          when x == 0
    // =====
    public CProgramXMinus1() : base (
        new IStatement[]
        {
            /* 01 */ new CStatementCondition(1,4,99), // x == 0 --> jpm end
            /* 02 */ new CStatementAddOne(3),         // k++
            /* 03 */ new CStatementCondition(1,3,7),   // x == k+1 --> jmp 7
            /* 04 */ new CStatementAddOne(2),          // k++
            /* 05 */ new CStatementAddOne(3),          // (k+1)++
            /* 06 */ new CStatementCondition(3),       // jmp 3
            /* 07 */ new CStatementTransfer(2,1)       // mov 2 -> 1
        }
    )
}
```

### 3.4 The client code

The client can code a program and invoke its execution on a machine with the following code :

```
try
{
    IProgram program = new CProgramXMinus1();
    long x = 1;
    long lResult = new CMachine(new long[] {x}).Execute(program);
}
catch (CMachineInLoopException ex)
{
}
```

The code :

- creates a specific program
- creates a machine passing the initial condition (an IEnumerable of values)
- invokes the execute method on the machine, passing the program
- retrieve the result of the execute method, which is the result of computation
- if the program enters a loop (which is a simulated loop condition) the machine throws a LoopDetectedException.

## 4 Points of interest

### 4.1 Initial Condition

The URM defines as "**initial condition**" the input values only. This is somehow counter-intuitive to us, since the "condition" term might refer to the initial "state" (i.e. the state of the entire tape : both the input values and the initial state which is defined by the program). In URM, the tape defaults to zero in each and every cell, and the input values are set from the leftmost squares before the computation starts.

In order to implement the definition of such initial condition, we can follow two strategies :

- we can define a new non-URM standard statement – say SetValue(n, v) - which allows us to explicitly set a value in a tape square



- we can pass an array of values to the machine before executing the program. The machine asks the tape to load such values.

#### 4.1.1 Initial condition defined by the Set Statement

We used this strategy mainly to try the extensibility feature of the statements as we are aware this is not the best way to set the initial condition.

If we set the initial conditions by executing such statements :

- those statements belong to the program, which is held responsible for defining its initial condition.
- as a consequence, we couple the initial condition to the program, which means that the program is not "parametric" any more (essentially we are killing the ability of the program to start in any initial condition)
- on the other hand, executing a program in any initial condition can lead to unexpected results, because the program itself "encodes" the squared being used and their semantics.

#### 4.1.2 Initial condition defined by the client before the program execution

This strategy is better because it reflects the URM model : since the model states that the initial condition is defined by setting values starting from the left side of the tape, we can simply pass the machine an ordered set of values during its construction. The machine will then ask the tape to load them one by one.

This way, the program does not need the SetValue statement any more and it is not held responsible for setting the initial condition. As a consequence, the program can run in any initial condition, which can also lead to unexpected results.

### 4.2 The program counter (PC)

The program execution is implemented by the machine by :

- setting the program counter to 1 (the PC is an integer "pointer" to the next statement waiting for execution)
- asking the program to fetch the next execution statement
- executing the next execution statement
- moving the program counter to the "next statement"

#### 4.2.1 Responsibility of the program counter updates

We need to decide who is **responsible** for the program counter update. It could reasonably be either the **machine** or the **statement**. This is a design task and we face more than one design choice.

##### 4.2.1.1 The PC is updated by statements

Since the program counter is **incremented** after some types of statements (e.g. arithmetic statements) and it is **updated in a different fashion** after other types of statements (e.g. conditional statements), we might be tempted to state that updating the PC is a responsibility of the statement.

The default statement behaviour might be "PC++" which might be overridden by statements in need of a different behaviour.

At first, this doesn't sound so bad but we should notice that:

- to do so, the machine needs a mutable state (the PC), which we would like to avoid
- such a state must be observable and updatable by the statement and not by other types. As a consequence, since we don't have "friend types" in c#, we should encapsulate the statement types as nested types of the machine type. There is nothing wrong about that, apart from unnecessary complexity.

In other words, this way we couple the statements to the machine which executes them and we do it more than necessary. Of course if we state that a statement S can be executed by the machine M only, such a coupling is by design. However, for sake of genericity and simplicity we would like to avoid that.

##### 4.2.1.2 The PC is updated by the machine

We think a better design would be to let the machine update the PC using :

- a default behaviour (i.e. PC++)
- using the information provided by the statements after their execution, if they provide it (PC = statement result)



By doing so, we can let the machine be immutable (no PC state) in a more functional coding style.

Each statement returns a `StatementResult` which may or may not encapsulate the next PC value. If such a value is defined, we use it, otherwise we use the default behaviour (PC++) which is defined by the machine (as opposed to be defined by the statement).

### 4.3 The halting problem : faking the loop

In order to pragmatically "solve" the unsolvable halting problem (we need programs that somehow terminate), we can state that "after N steps, the machine is meant to have entered a loop". The bigger the N, the less the discrepancy from the "fake" and the "real" loop condition.

In order to do so, we define a **Statement Instance counter** and we increment it after each statement execution. A guard throws an exception when such a counter exceeds the maximum allowed steps for the machine.

### 4.4 Machine state notification

Since we want to **test and debug** our written-on-paper-program, we are interested in inspecting the state of the machine, after each statement execution.

To satisfy this requirement, the machine notifies its state :

- before the first statement execution (i.e. the initial condition)
- after each statement execution (i.e. the current tape state)

The notification is implemented by the **observer pattern**, simplified by the DotNet event notion. A client can subscribe such an event in order to be notified of the machine state after each execution step.

## 5 Addenda

### 5.1 Addendum 1 : the URM-program text parser

Upto now, we defined a mini framework and we can use it to **define a program by c# code**. Since the project aims to simulate a URM which can be fed with many programs, coding an URM-program in c# is not very friendly on a write-run-and-test basis.

We want to be able to write **URM programs as text files**, feed them into a **command line application** and let the application parse them into a DotNet `IProgram` object, which will be executed by the machine.

To do so, we define a **simple parser** which creates an `IProgram` by parsing a text containing URM statements. The parser is of no interest in this paper, therefore it is implemented in a quick-and-dirty fashion and as such it is not technically documented here.

#### 5.1.1 Statements

The parser is able to parse the URM statements which are **written in the same fashion** we use in our Computability class, therefore :

- "z(n)" : clear
- "s(n)" : increment by 1
- "t(m,n)" : move
- "c(m,n,q)" : conditional

We can :

- use the parser to parse the text and obtain an `IProgram` as a list of `IStatement`
- ask the program to transform back to text (this procedure drops comments and formatting)

Please notice that, for sake of simplicity, the parser is **case-sensitive**.

#### 5.1.2 Statement separators

The statements can be separated by "#" or by a line feed '\r'. While the line feed is useful in text files containing urm programs, in which each statements is defined in its own line, the "#" is useful if we want to pass a small program straight to the command line, where we cannot input a multi line text.



### 5.1.3 Spaces and other characters

Spaces are always ignored. Other unexpected characters make the scanner raise an exception.

### 5.1.4 Comments

In order to describe URM programs, **comments** can be used. A comment starts with a ';' and terminates :

- at the end of the line '\n'
- when a statement separator '#' is reached.

### 5.1.5 Line numbers

Line numbers are useful in order to be able to comfortably read the conditional instruction branches.

Given our minimality requirements, we do not implement a line numbering strategy because we can easily define **line numbers inside comments**. For example :

```
c(2,3,99)      ; 001 : x == y => last
s(1)           ; 002 : x++
s(3)           ; 003 : k++
c(1,1,1)       ; 004 : jmp back to 1
```

### 5.1.6 Example

For example, we want to be able to compile the following :

```
string source = "z(1)|s(2)|t(1,3)|c(1,2,3)";
IProgram program = new CProgramParser().Parse(source);
```

## 5.2 Addendum 2 : the application

Since we are now able – by Addendum 1 – to parse a text into a IProgram, we can write a console application which is fed a text program, executes it, shows the intermediate states and prints the processing result.

### 5.2.1 Command line arguments

We need to feed the application the following arguments :

- the program : this can be a file path or an inline program, with '#' separated instructions
- initial value 1
- initial value 2
- .....
- initial value i

Therefore we can call a program as follows :

- >com.scrigner.RAMModel.exe s(1)#(s1)....#(s1) 2 : **inline** urm statements and one input
- >com.scrigner.RAMModel.exe progs/MyProgram.urm 2 3 4 : **path to program** and three inputs

### 5.2.2 Example

Given the  $f(x)=x-1$  program, saved into the file "programs\XMinus1.urm" :

```
; =====
; f(x) =
;           x - 1 when x >= 1
;           0    when x < 1
; =====
; x1 : x
; x2 : j (= x-1)
; x3 : k (incremented from 0 to x)
; =====

c(1,4,99)      ; 001 : x == 0 --> end
s(3)           ; 002 : k <- 1
c(1,3,7)       ; 003 : x == x => jmp to last
```





```
s(2)      ; 004 : j++
s(3)      ; 005 : k++
c(1,1,3)   ; 006 : jmp back
t(2,1)     ; 007 : result <- x-1
```

We can execute it as follows : (we pass  $X = 0$ ) :  $0-1 = 0$  (by definition)

```
D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>com.scrigner.RAMModel.exe Programs\XMinus1.urm 0
#STEP STIDX      TAPE
00000 00001 c(1,4,99)  000
00001 00099      000 000 000 000 [004.192]
Execution Completed. Result=[0]

D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>
```

The result is the shown table :

- the first column shows the step of the computation
- the second column shows the program counter (i.e. the next instruction address)
- the third column shows the instruction located at the program counter address
- the following columns show the touched tape values

In the previous example, we can see that, since  $x1 == x4$  ( $= 0$ ) we jumped to 99 : the result in  $X1$  (i.e. 0).

We change the argument to 1 :  $1-1 = 0$

```
D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>com.scrigner.RAMModel.exe Programs\XMinus1.urm 1
#STEP STIDX      TAPE
00000 00001 c(1,4,99)  001
00001 00002 s(3)      001 000 000 000
00002 00003 c(1,3,7)   001 000 001 000
00003 00007 t(2,1)     001 000 001 000
00004 00008          000 000 001 000
Execution Completed. Result=[0]

D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>
```

We get a longer path through the program :

- we don't jump to 99, since  $1 \neq 0$
- we increment  $X3$
- since  $X1 == X3$  we jump to I7
- we move  $X2$  to  $X1$
- we return  $X1 = 0$

Just another example : we change the argument to 3 :  $3-1 = 2$



```
D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>com.scrigner.RAMModel.exe Programs\XMinus1.urm 3
#STEP STIDX TAPE
00000 00001 c(1,4,99) 003
00001 00002 s(3) 003 000 000 000
00002 00003 c(1,3,7) 003 000 001 000
00003 00004 s(2) 003 000 001 000
00004 00005 s(3) 003 001 001 000
00005 00006 c(1,1,3) 003 001 002 000
00006 00003 c(1,3,7) 003 001 002 000
00007 00004 s(2) 003 001 002 000
00008 00005 s(3) 003 002 002 000
00009 00006 c(1,1,3) 003 002 003 000
00010 00003 c(1,3,7) 003 002 003 000
00011 00007 t(2,1) 003 002 003 000
00012 00008 002 002 003 000
Execution Completed. Result=[2]

D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>
```

Here we can see iterations.

As a loop detection example, we can use the program loop.urm :

```
D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>com.scrigner.RAMModel.exe Programs\Loop.urm
#STEP STIDX TAPE
00000 00001 c(1,1,1)
00001 00001 c(1,1,1)
00002 00001 c(1,1,1)
00003 00001 c(1,1,1)
00004 00001 c(1,1,1)
00005 00001 c(1,1,1)
00006 00001 c(1,1,1)
00007 00001 c(1,1,1)
00008 00001 c(1,1,1)
00009 00001 c(1,1,1)
00010 00001 c(1,1,1)
Execution halted in a loop.

D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>
```

If we can execute 10 steps at most (value we set before executing the code), we enter a loop and the program ends in a loop condition.

As a last example, we execute the XMinusY program :





```
D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>com.scrigner.RAMModel.exe Programs\XMinusY.urm 5 2
#STEP STIDX      TAPE
00000 00001 c(1,2,5) 005 002
00001 00002 s(2)    005 002
00002 00003 s(3)    005 003
00003 00004 c(1,1,1) 005 003 001
00004 00001 c(1,2,5) 005 003 001
00005 00002 s(2)    005 003 001
00006 00003 s(3)    005 004 001
00007 00004 c(1,1,1) 005 004 002
00008 00001 c(1,2,5) 005 004 002
00009 00002 s(2)    005 004 002
00010 00003 s(3)    005 005 002
00011 00004 c(1,1,1) 005 005 003
00012 00001 c(1,2,5) 005 005 003
00013 00005 t(3,1)  005 005 003
00014 00006          003 005 003
Execution Completed. Result=[3]

D:\Research\Univ\ComputabilitaELinguaggi\com.scrigner.RAMModel\com.scrigner.RAMModel\bin\Debug>
```

## 6 Final observations

### 6.1 Statement Relocability

It might be useful to be able to define **offsets** and/or **labels** for the q argument of the c statement (i.e. relative or symbolic addressing). This is because the absolute addressing must be updated every time we add or remove a statement, which is quite frustrating.

### 6.2 AddOne and how operation performance depends on data

Since one is the only value we can add, operating on big numbers takes more cycles than operating on little numbers. This is because adding 5 means cycling 5 times adding one each time. Adding 100 means cycling 100 times.

In x86 or 68k assembly languages (which we used a lot long ago), adding a value V takes X cycles, whatever V is.

### 6.3 EQ Only

Since we can use the equality compare statement only (no GT,GE,LT,LE), it is easy to do :

- IF EQUALS .... ELSE **LOOP**

It is not so easy to do :

- IF EQUALS ... ELSE **SOMETHING ELSE**

Also, since we can add just 1, we must add 1 in cycles and we must check for equality in **every and each cycle**. Missing some cycles comparison, can be lethal to the execution, which enters a loop.

The following example is an implementation for integer division :

```
; =====
; f(x, y) =
;          x / y  when y > 0
;          loop  when y == 0
; =====
; a1 : x
; a2 : y
```



```
; a3 : kx : 0..y      : we inc the result every time kx == y
; a4 : ky : 0..x      : we exit when ky == x
; a5 : x / y
; =====

c(2,3,1)      ; 001 : y == 0 --> loop

c(4,1,11)     ; 002      when x4 == x --> goto last

      c(3,2,8)      ; 003 We inc x3 until it reaches y (and on each step we also inc x4)
      s(3)          ; 004
      c(4,1,11)      ; 005 BEFORE incrementing x4, we check if it reached X (should it happen, we exit)
      s(4)          ; 006 ... otherwise we inc x4
      c(1,1,3)      ; 007

s(5)          ; 008 each time x3 reaches y we did a full subtraction step : we inc x5 (which holds the
division result) ...
z(3)          ; 009 ... and we clear x3 (which ranges from 0 to y)
c(1,1,2)      ; 010

t(5,1)        ; 011
```

The instruction in red is required to be able to correctly calculate a division having a remainder which is greater than zero.

## 6.4 Visualization framework, helpers and a few ideas

We might need visualization rules to visualize the computation paths. It is very inconvenient to use natural language in comments. For example, the ability to assign meaningful names to the squares might be able to greatly clarify the program concepts. Also some kind of paper visualization model might be helpful.

For example , instead of :

```
c(2,3,99)      ; 001 : x == y -> goto last
s(1)           ; 002 : x++
s(3)           ; 003 : k++
c(1,1,1)       ; 004 : jmp back
```

It would be much better something like that :

```
start : c(x,y,end)
      s(x)
      s(k)
      c(x,x,start)
end:
```

## 6.5 Coding as a "thinking virus"

"Programming languages teach you not to want what they cannot provide" (cit. missing, sorry).

Among the many meanings, one is that programming languages train our brains to always think the same way, which is good if we aim to reach high productive standards. What is bad, is that when we are forced to think in a different thinking pattern, for example because of technology constraints, big thinking problems arise.

Coding the subtraction function using the addition is a surprisingly difficult task for me, because it takes me lot of effort to think in an unfamiliar pattern. As a consequence, it makes me feel very stupid.

The reason for this extraordinary amount of effort might be traced back to the fact that computer programming seems to rewrite pathways in the brain, making us think that a familiar problem, which we always solved the same way, is solvable in that very same way only. This induced misconception is sometimes referred to as a thinking virus, which dulls critical thoughts and favours the least expensive (in terms of effort) thinking pathway over a most expensive one. Also, the positive feedback given by the familiarity with the problem (using a subtraction to make subtraction) reinforces such a pathway.

Basically, this is why one should learn more than one programming paradigm.



## 7 A few URM programs

### 7.1 Abstract

We include here a few programs for the URM machine. Such programs were **moderately tested** and should work as expected.

### 7.2 Loop

```
; =====  
; f(x) = undefined (loop)  
; =====
```

c(1,1,1) ; 001 Always jump back to instruction 1

### 7.3 $F(x) = 0$

```
; =====  
; f(x) = 0  
; =====
```

z(1)

### 7.4 $F(x) = 1$

```
; =====  
; f(x) = 1 (Algo A)  
; =====
```

z(1)

s(1)

### 7.5 $F(x) = 1$ (Another version)

```
; =====  
; f(x) = 1 (Algo B)  
; =====
```

s(2)

t(2,1)

### 7.6 $F(x) = 3$

```
; =====  
; f(x) = 3  
; =====
```

z(1)

s(1)

s(1)

s(1)

### 7.7 $F(x) = x + 2$

```
; =====  
; f(x) = x + 2  
; =====
```

s(1) ; 001 : x++

s(1) ; 002 : x++



## 7.8 $F(x) = x + y$

```
; =====
; f(x, y) = x + y
; =====
; x0 : x
; x1 : y
; x3 : k
; =====

c(2,3,99)      ; 001 : x == y -> goto last
s(1)           ; 002 : x++
s(3)           ; 003 : k++
c(1,1,1)       ; 004 : jmp back
```

## 7.9 $F(x) = x - 1$

```
; =====
; f(x) =
;           x - 1 when x >= 1
;           0     when x < 1
; =====
; x1 : x
; x2 : j (= x-1)
; x3 : k (incremented from 0 to x)
; =====

c(1,4,99)      ; 001 : x == 0 --> end
s(3)           ; 002 : k <- 1
c(1,3,7)       ; 003 : x == x => jmp to last
s(2)           ; 004 : j++
s(3)           ; 005 : k++
c(1,1,3)       ; 006 : jmp back
t(2,1)         ; 007 : result <- x-1
```

## 7.10 $F(x) = x - y$ (loop on error)

```
; =====
; f(x, y) =
;           x - y when x >= y
;           loop  when x < 1
; =====

c(1,2,5)       ; 001 : x == y => last
s(2)           ; 002 : y++
s(3)           ; 003 : k++
c(1,1,1)       ; 004 : jmp back
t(3,1)         ; 005 : mov k > x
```

## 7.11 $F(x) = x - y$ (extension on error)

```
; =====
; f(x, y) =
;           x - y when x >= y
;           0     when x < 1
; =====
; a1 : x
; a2 : y
```



```
; a3 : kx
; a4 : ky
; a5 : result
; =====
```

```
t(1,3)      ; 001 we copy x and y because we need to inc from 0 to x and from 0 to y separately
t(2,4)      ; 002
```

```
    c(1,4,9)      ; 003 if x>=y, ky can reach x. if they are equal we jump
    c(2,3,11)     ; 004 if y>=x, kx can reach y. this is the error condition : result = 0
    s(3)          ; 005 we increment both kx, ky and the result
    s(4)          ; 006
    s(5)          ; 007
    c(1,1,3)      ; 008
```

```
t(5,1)      ; 009 we use a5 as result
c(1,1,99)   ; 010
;t(6,1)     ; 011
z(1)        ; 011 we use 0 as result
```

## 7.12 F(x) = x \* y

```
; =====
; f(x, y) = x * y
; =====
; a1 : x
; a2 : y
; a3 : kx : 0..x
; a4 : ky : 0..y
; a5 : result : x * y
; =====
```

```
c(2,4,9)      ; 001 : y == ky --> last
c(1,3,6)      ; 002 : x == kx
s(3)          ; 003 : kx++          This is the inner cycle
s(5)          ; 004 : result++
c(1,1,2)      ; 005 : back
z(3)          ; 006 : kx <- 0      We clear the kx ...
s(4)          ; 007 : ky++          and advance the ky
c(1,1,1)      ; 008 : back
t(5,1)        ; 009 : result -> a1
```

## 7.13 F(x) = x / y

```
; =====
; f(x, y) =
;          x / y when y > 0
;          loop  when y == 0
; =====
; a1 : x
; a2 : y
; a3 : kx : 0..y      : we inc the result every time kx == y
; a4 : ky : 0..x      : we exit when ky == x
; a5 : x / y
; =====
```

```
c(2,3,1)      ; 001 : y == 0 --> loop

c(4,1,11)     ; 002          when x4 == x --> goto last
```



```
c(3,2,8)      ; 003 We inc x3 until it reaches y (and on each step we also inc x4)
s(3)          ; 004
c(4,1,11)     ; 005 BEFORE incrementing x4, we check if it reached X (should it happen, we exit)
s(4)          ; 006 ... otherwise we inc x4
c(1,1,3)      ; 007

s(5)          ; 008 each time x3 reaches y we did a full subtraction step : we inc x5 (which holds the
division result) ...
z(3)          ; 009 ... and we clear x3 (which ranges from 0 to y)
c(1,1,2)      ; 010

t(5,1)        ; 011
```

**Scrigner S.r.l.**

Address:	Via Milano 25, 34100 Trieste (TS) - ITALY
P.IVA/C.F.:	IT-01141600328
E-Mail:	s-info@scrigner.com
C.C.:	Banca Generali, Filiale di Trieste - Via Machiavelli 4, 34100 Trieste
IBAN :	IT 21 X 03075 02200 CC8500246027