

Genetics-IT

Eine Esoterische Programmiersprache über genetische IT und universelle FPGA Logiken



**Stand:** 29.4.22  
**Autor:** Zoran Vranešević

**Das Projekt im Internet:**<https://github.com/Zoltan-X/Genetics-IT>

Vorwort: 2

Gene Basics: 2

Definitionen für ein Minimum an Informationsverarbeitung. 2

Technischer ansatz 2

Aminosäuren und eine frei gewählte Zuordnung zu Codeelemente 3

Adressieren von Registern, Variablen und Direktwerten: 3

Informationen müssen verglichen werden: 3

Informationen müssen verändert werden können: 3

Flussrichtungen über die Bits der Register 3

Definition vom If: 3

Beispiele: 4

Minimum Information processing Examples 4

Definition Increment 4

Definition Decrement 5

Definition CompareLower 6

Definitions of Compare Equal 7

Definitions of Compares 8

Definition a negation 9

## Vorwort:

Dies ist eine Esoterische Programmiersprache, welche sich ursprünglich an den Möglichkeiten der Genetik (DNS/RNA) orientierte. Hierbei waren Polymorphe Rekursionen in der Natur der Dinge im Vordergrund ebenso wie vollständig mit einem absurden Minimum an Befehlen tauglich zu bekommen. Ferner ist die Flip-Flop Logik höchst geeignet für FPGA Programmierung.

## Gene Basics:

Gene haben folgende Aminosäuren:

* (C) Cytosin
* (G) Guanin
* (A) Adenosin
* (T) Thymin
* (U) Uracil

Diese haben folgende 4 Paarungen:

* CG
* GC
* AT (DNA only) AU (RNA only)
* TA (DNA only) UA (RNA only)

Daher mit Respekt für die esoterische Programmiersprache „Brainfuck“ ist GeneticsIT noch abstrakter und der esoterische Grad ebenso hoch, wenn nicht höher. GenteticsIT arbeitet mit Bit Flipping in Registern. Dies ist entlehnt an die Grundlagen der Digitaltechnik - den Flip Flops.   
Flip Flops sind Grundlage aller Logischen Schaltungen wie logisches und & oder, einfaches negieren, exklusivem und & oder. Des Weiteren sind Registerverarbeitungsrichtungen von LowEndian zu BigEndian wechselbar und das Adressieren der einzelnen Bits geschieht reihum entsprechend der Big- oder LowEndian Weise und der Spin Richtungsangabe.

## Definitionen für ein Minimum an Informationsverarbeitung.

### Technischer ansatz

Informationen oder Ressourcen müssen adressiert werden können.

* Adressieren von Registern, Variablen und Direktwerten A := 123, A := B
* Informationen müssen verglichen werden. If CompareEqual(A, B) Branch
* Informationen müssen verändert werden können. FlipBit(A, R1..Rx, Spin-L/R)
* Flussrichtungen über die Bits der Register Spin(L oder R)

Damit haben wir ein Maximum von 4 Befehlen. Weitere Hochsprachen Möglichkeiten wie Schleifen in der Codeflusssteuerung werden über Bedingte Rekursionen (Selbstaufrufe) erreicht und das Register­bit im Register wird automatisiert bei jeder Rekursion um eine Stelle in Flussrichtung fortgeführt.  
Rekursion ist in diesem Zusammenhang hier ein erneuter Selbstaufruf bei dem der Funktionsstack nicht mit neuen Parametern belastet wird, da die lokalen Variablen global innerhalb der obersten Ebene des Aufrufs einer solchen Methode bleiben. Das bedeutet im Fazit der Rekursive Selbstaufruf ist eigentlich keiner, sondern eine Schleife aus dem Methodenkörper an den Anfang des Methodenkörpers.   
Somit folgt eine Auflistung und mögliche Zuordnung von Befehlen basierend auf dem Konzept von vier Aminosäurepaarungen als Codeelemente.

##### Aminosäuren und eine frei gewählte Zuordnung zu Codeelemente

* CG (Flip)
* GC (compare EQU)
* AU(RNA) oder AT (Spin-R)
* UA(RNA) oder TA (Spin-L)

##### Adressieren von Registern, Variablen und Direktwerten:

An dieser Stelle müssen wir eine Hochsprachenlogik voraussetzen, da wir mit nur einem Befehl logisch nur ein Register ansprechen können. In diesem Zusammenhang betrachten wir Variablen als eigene Dimension und jede Variable als anzusprechendes Register. Auch Direktwerte werden nach genetischem Vorbild nicht händelbar, würden aber ebenso wie Variablen als Registerwert behandelt werden.

##### Informationen müssen verglichen werden:

Da wir nur eine universellen Vergleichsart anstreben müssen wir uns mit einem „Compare Equal“ vergleich begnügen und da heraus die anderen Vergleichsformen ableiten. Für die Negierung brauchen wir dann des Weiteren eine Negierung, welche wir bequem mit Inversion eines Registers erreichen.

##### Informationen müssen verändert werden können:

Dieses ist Konzeptionelle nur durch Bit Flips zu bewerkstelligen.

##### Flussrichtungen über die Bits der Register

Mit der Spin-R/Spin-L Wird die Flussrichtung über die Bits eines Registers bestimmt bzw. umgekehrt.  
Im genetischen Ansatz würde es sich dabei um ein Sequenz Anfang- und Ende-Konstrukt handeln. Hierbei ist zu unterscheiden, ob es sich um Big- oder Low-Endian Verhaltensweisen des Registers handelt. Über eine vorherbestimmte Flussrichtung kann auf solche Details Rücksicht genommen werden.

##### Definition vom If:

Wenn wir multidimensionalen code Bedenken, so haben wir über Spin-L und Spin-R folgende Dimensionen der Anwendungsfälle:

* Programmablaufflusssteuerung (nur if thens)
* Register Flussrichtung <BigEndian|LowEndian>

Ebenso müssen wir bei der Adressierung von Variablen bedenken das wir sie als eigenständige Dimension betrachten, wo wir sie wie Register adressieren.

Wenn wir diese speziellen Rekursionen durch Selbstaufruf als Schleifenlogik verstehen wollen, so müssen wir definieren, wie dieser passiert:

1. Register Bit Indizes werden entsprechend dem Spin um einen erhöht oder vermindert
2. Rücksprung an den Funktionsanfang
3. Variablen werden nicht übergeben und bleiben lokal erhalten
4. Es entsteht kein iterieren der Aufrufskette im Sinne einer Rekursion

## Beispiele:

### Minimum Information processing Examples

#### *Definition In*c*rement*

*##########################################################################################  
#  
# Recursive definition of an incrementation process with   
# automatic abort of recursion if the processing flow reaches its Limit.   
# Go through memory register and flip first Zero Bit.   
# Flip the Flow Direction and flip all bits returning Flow Direction   
#  
# Information => Limited Set of Bits,   
# BitIndex => 1 -> n /\*x as Limes value\*/ and   
# n -> 1 /\*as opposit Limes value\*/   
# FlowDirection => Sequential Information processing Flow Direction   
# L-Spin or R-Spin   
# originalSpin => local Constant only to be changed on initial function call   
#   
##########################################################################################*

Definition Increment( Information,   
 BitIndex := auto\_default(1..bit\_length(Information)),   
 FlowDirection := default(L<-R),   
 originalSpin := default(L<-R)   
 )

// As Long as no Register overflow occured and Flow Direction unchanged   
 if BitIndex equ overflow return Information   
 if FlowDirection equ originalSpin   
 if Information(BitIndex) equ 0   
 Flip(Information,BitIndex)   
 Flip(FlowDirection)  ENDIF   
 call self  
 ENDIF

// After change/swap Flow-Direction invert/flip every single Bit   
 Flip(Information, BitIndex)   
 call self

End-Definition

*/\*   
  
Increment(0, default, default, default)*

*BitIndex Range L<-R 4 3 2 1 0   
BitIndex Range L->R 0 1 2 3 4*

*Run\Iterations => \ First Second Third Fourth Fifth  
 Run*

*1 = 0 0 0, 0 0 1, End  
 2 = 0 0 1, 0 1 1, 0 1 0, End   
 3 = 0 1 0, 0 1 1, End  
 4 = 0 1 1, 1 1 1, 1 0 1, 1 0 0, End  
 5 = 1 0 0, 1 0 1, End  
 6 = 1 0 1, 1 1 1, 1 1 0, End  
 7 = 1 1 0, 1 1 1, End  
 8 = 1 1 1, overflow, End  
\*/*

#### *Definition De*c*rement*

*##########################################################################################  
#  
# Recursive definition of an decrementation process with   
# automatic abort of recursion if the processing flow reaches its Limit.   
# Go through memory register and flip first non-Zero Bit.   
# Flip FlowDirection and flip all bits at returning Flow-Direction   
#  
# Information => Limited Set of Bits,   
# BitIndex => 1 -> n /\*x as Limes value\*/ and   
# n -> 1 /\*as opposit Limes value\*/   
# FlowDirection => Sequential Information processing Flow Direction   
# L-Spin or R-Spin   
# originalSpin => local Constant only to be changed on initial function call   
#  
##########################################################################################*

Definition Decrement( Information,   
 BitIndex := auto\_default(1..bit\_length(Information)),   
 FlowDirection := default(L<-R),   
 originalSpin := default(L<-R)   
 )

// As Long as no Register overflow occured and Flow Direction unchanged   
 if BitIndex equ overflow return Information  if FlowDirection equ originalSpin  if Information(BitIndex) equ 1   
 Flip(Information,BitIndex)   
 Flip(FlowDirection)  ENDIF   
 call self   
 ENDIF

// After change/swap Flow-Direction invert/flip every single Bit   
 Flip(Information, BitIndex)   
 call self

End-Definition

*/\**Decrement*(0, auto\_default, default, default)*

*BitIndex Range L<-R 4 3 2 1 0   
BitIndex Range L->R 0 1 2 3 4   
   
Run\Iterations => \ First Second Third Fourth Fifth  
 Run  
 1 = 1 1 1, 1 1 0, End  
 2 = 1 1 0, 1 0 0, 1 0 1, End  
 3 = 1 0 1, 1 0 0, End  
 4 = 1 0 0, 0 0 0, 0 1 0, 0 1 1, End  
 5 = 0 1 1, 0 1 0, End  
 6 = 0 1 0, 0 0 0, 0 0 1, End  
 7 = 0 0 1, 0 0 0, End   
 8 = 0 0 0, Overflow, End  
   
\*/*

#### *Definition CompareLower*

*##########################################################################################  
#  
# Recursive definition of an decrementation process with   
# automatic abort of recursion if the processing flow reaches its Limit.   
# Go through memory register and flip first non-Zero Bit.   
# Flip FlowDirection and flip all bits at returning Flow-Direction   
#  
# a => comparand,   
# b => comparator  
# BitIndex => Index for bits in registers   
#  
##########################################################################################*  
Definition CompareLower(   
 a,   
 b,   
)

if a Equal b return False  
 return \_\_CompareLower(a, b)  
  
 Definition \_\_CompareLower(   
 a,   
 b,   
 BitIndex := auto\_default(bit\_length(a)..1),   
 )

if a(BitIndex) Equ b(BitIndex)   
 call self  
 End-If  
   
 // match?  
 if b(BitIndex) equ 1 return True   
  
 //not matched yet?, then its false!  
 return False  
  
 End-Definition

End-Definition

*/\*   
 Compare if a lower b   
 CompareLower(a, b)   
  
case (100, 100) = 0 (False)   
Run (A) (B)  
1 100 = 100 //End with returning False*

*case (101, 100) = 0 (False)   
Run (A) (B)  
1 101 = 100 //continue   
1 1 = 1 //continue  
3 0 = 0 //continue  
4 1 = 0 // End with return False*

*case (100, 101) = 1 (True)   
Run (A) (B)  
1 100 = 100 //continue   
1 1 = 1 //continue  
3 0 = 0 //continue  
4 1 = 0 // End with return True*

*\*/*

#### *Definitions of Compare Equal*

##########################################################################################  
#  
# Definitions for Bitwise Equal compare Operator   
#  
# variables:  
# a => comparand,   
# b => comparator  
#  
# defines;  
# ZeroIndex => auto Index for bits in BitIndex Register with (Decrement)   
# BitIndex => auto Index for bits in registers with (Decrement)   
# Spin => LowEndian   
#  
##########################################################################################  
Definition \_\_Equal(a, b)  
 defines   
 Spin := default(LowEndian)   
 BitIndex := auto\_default(bit\_length(a)..1)  
 False := 0  
 True := 1  
 End-Defines  
   
 private   
 //iterates in Lowendian direction through a,b and   
 //finds first different Bits  
 Definition \_\_firstDiffBit  
 if a(BitIndex) Equ b(BitIndex)   
 call self  
 return bitIndex  
 End-Defintion  
  
 //Needed for local automatic Register Bit-Index of Zero Compare  
 Definition \_\_BitIndexZero(bitIndex)  
 defines  
 //Automatic local Register-Bit-Index  
 ZeroIndex := auto\_default(bit\_length(a)..1)   
 End-Defines  
  
 //Method Body  
 if bitIndex(ZeroIndex) Equ 1 return False  
 if bitIndex(ZeroIndex) Equ 0 \_\_BitIndexZero(ZeroIndex)  
 return True  
  
 End-Defintion  
  
 End-Private  
   
 return \_\_BitIndexZero(\_firstDiffBit)  
End-Defintion  
  
defines: \_, Equal, \_ = \_\_Equal

*/\*   
 Compare Equal   
 a Equal b [True/False]   
  
case 100 Equal 100) = 0 (True)   
Run (A) (B) ZeroIndex Result  
1 100 100 - True  
2 101 100 1 False  
3 100 110 2 False*  
*4 000 111 3 False*

*\*/*

#### *Definitions of Compares*

##########################################################################################  
#  
# Definitions of atomic compares defined on top of Bitwise Equ and Expressionwise Equal  
#  
# variables:  
# a => comparand,   
# b => comparator  
# Condition => condition  
#  
# defines;  
# BitIndex => auto Index for bits in registers with (Decrement)   
# Spin => LowEndian   
#  
##########################################################################################  
Definition Compare( a, b, Condition=[”LOW”, ”LEQ”, ”EQU”, “GRE”, “GRE”])  
  
 private  
  
 defines   
 BitIndex := auto\_default(bit\_length(a)..1)  
 Spin := default(LowEndian)  
 End-Defines  
  
 Definition \_\_firstDiffBit  
 if a(BitIndex) Equ b(BitIndex)   
 call self  
 return bitIndex  
 End-Defintion  
  
 Definition \_\_CompareLowerLowerEqual  
 if Condition Equal “LOW” return True  
 if Condition Equal “LEQ” return True  
 End-Defintion  
   
 Definition \_\_CompareGreaterGreaterEqual  
 if Condition Equal “GRE” return True  
 if Condition Equal “GEQ” return True  
 End-Defintion  
  
 Definition \_\_CompareLowerGreaterEqualEqual  
 if Condition Equal “LEQ” return True  
 if Condition Equal “EQU” return True  
 if Condition Equal “GEQ” return True  
 End-Defintion  
   
 End-Private  
  
 \_\_firstDiffBit(a, b, BitIndex)  
  
 // match?   
 if BitIndex Equal 0 \_\_CompareLowerGreaterEqualEqual   
 if b(BitIndex) Equ 1 \_\_CompareLowerLowerEqual  
 if A(BitIndex) Equ 1 \_\_CompareGreaterGreaterEqual  
   
 //not matched yet? Then it’s false!  
 return False  
  
End-Definition   
  
defines: \_, Lower, \_ = Compare( a, b, “LOW”)  
defines: \_, LowerEqual, \_ = Compare( a, b, ”LEQ”)  
//defines: \_, Equal, \_ = Compare( a, b, ”EQU”) //already Defined  
defines: \_, GreaterEqual, \_ = Compare( a, b, “GEQ”)  
defines: \_, Greater, \_ = Compare( a, b, “GRE”)

#### *Definition a negation*

*##########################################################################################  
#  
# Definition of a register bits negation process.   
#  
# Information => Limited Set of Bits,   
# BitIndex => 1 -> n /\*x as Limes value\*/ and   
# n -> 1 /\*as opposit Limes value\*/   
#   
##########################################################################################*

Definition Neg(Information)   
 defines  
 BitIndex := auto\_default(1..bit\_length(Information)), )  
 End-defines

Flip(Information,BitIndex)  
 call self

End-Definition

defines: \_, Not, \_ = Not(a, b)   
  
/\*  
 Information 01010100  
 Negation 10101011  
\*/