# A Coalgebraic Approach to the Y86 Processor Architecture

Nuno Rodrigues

*Departamento de Informática, Universidade do Minho*

*4710-057 Braga, Portugal*

*nunorodrigues@di.uminho.pt*

**Abstract** This communication reports an attempt to use a mathematical approach to model a simplified version of an X86 processor, the Y86 processor. The main mathematic structures used in this formalization are Algebras, Coalgebras and Functors. To construct the Y86 processor model, some examples are presented using both Functorial definitions and Haskell animations.

## 1 Introduction

Many areas of mathematics have contributed for the modern Systems Engineering, namely Turing Machines, Graph Theory, Grammar Theory, Numerical Mathematics, Logic, Algebra. Each has contributed in many ways to accomplish current mass technological world. They have created the basis of modern computers and modern programming practices.

Universal Algebra (UA) is a well studied mathematical structure, which reasons about sets and operations on sets elements following some strict rules. Some knowledge in Algebra is expected from the reader, and only important aspects of algebra for the current work will be pointed out onwards.

Using pre-determined properties in UA things can be proved about the inhabitants of their sets, without really looking into the behaviour of those elements according to the rules, or even without proving it for every element of the algebra set. With this further step in abstraction, by reasoning about things and operations with certain pre-determined properties, rather than the concrete instances of them, a much large universe of elements and rules, can be covered without even concerning on working on finite or infinite sets.

Another well studied area of UA are operations between Algebras, known as morphisms. With this morphism one can jump from one Algebra to another, and prove rules involving more then one Algebra. This kind of operations will by exploit soon, as well as their connection to the programming world, where sometimes one may need to jump from different data representations.

Like in Universal Algebra, reasonings are base on pre-defined properties and rules, in Computer Science one prefers to think at a higher level of abstraction too.

To have a way of representing the entire spectrum containing all possible data type used in some programming language. Properties representation over them would also be appreciated and the proof of new behaviours based on others or on the structure of the data types themselves would be very appealing too.

With this in mind, the structure of data types are more relevant than their particular implementations, i.e., it is better to define lists of things rather than lists of *chars* or *strings*, since the former representation is a particular case of the previous one.

A possible solution for this quest for data type representation is somewhat hidden in Universal Algebra, and its underlying structure. With Algebra and some category theory, we can capture many and essential aspects of data type structures used in Computer Science.

A particular application of Algebra theory are Algebras of Functors, where we get the Functor [1] concept from Category Theory. Algebras of Functors are algebraic structures working over functors or, in other words, where the algebra's set is a set of functors.

To illustrate this idea of algebras working over functors applied to Computer Science, we continue presenting an example.

Using this notation a definition of a list containing some kind of elements is given by the following notation:

$$X \cong 1 + A \times X$$

If we look at $X$ as our data structure, in this case generic Lists, 1 as *, a given single set and $A$ as a generic set of elements, then we can construct an Algebra $(X, < nil, cons >)$ with $[nil, cons] : 1 + A \times X \to X$ where $X$ is representing all generic Lists.

Another way to regard the above formula follows: to construct a List we need an empty list, being it already a list or an element of a list $(A)$ and another list $(X)$.

By using structures of this kind, we can reason about any kind of lists, without looking into their inhabitant types or even structures.

This kind of algebraic methods, known as *algebraic specification* or *abstract data type theory* goes much further in modelling Computer Science data types, and capture important properties and operations between those data types, but for now this should be enough.

In the following sections, we introduce the idea of Coalgebras, why they are useful in Computer Science, some examples and finally apply such structures to the Y86 architecture.

## 2 From Algebras to Coalgebras

The main distinction of this two mathematical structures is that Algebras refer to *Construction* while on the other hand Coalgebras refer to *Destruction/Observation*, in a manner that we will made explicit with some examples. This phenomenon captures the main reason why we say that Coalgebras are dual to Algebras.

While in Algebras we were concerned about constructing objects that fulfill the Algebra's set, like functions $\alpha$ and $\beta$, in Coalgebras we want to observe elements of $X$ by desiccating such elements into more elemental elements that compose the first.

In order to illustrate better this destruction phenomena behind Coalgebras we present a simple example of a Coalgebraic model.

Lets describe a simple object-oriented class that represents 2D points, with two coordinates, $x$ and $y$.

Over our class we define three buttons (known as methods in objet oriented terminology), a *first* : $X \to R$ button retrieving the first coordinate of our class, a

[1] For the interested reader we suggest some investigation over category theory

$second : X \to R$ giving the second coordinate of our point and a third button $move : X \to X^{R \times R}$ moving a point by $x$, $y$ units on horizontal and vertical directions respectively. This three buttons of such a class can be combined into a single function

$$< first, second, move >: X \to \Re \times \Re \times X^{(\Re \times \Re)}$$

which forms a coalgebra on the state space $X$.

Another interesting thing about our object oriented class is that we know anything about its state behaviour, the only observations we can make is throw the $first$ and *second* buttons.

Along with this lack of information about an internal state of our classes, cames the indistinguishability of classes based on the observation functions that we may use. In this way, we can say that two classes returning the same value for both the $first$ and *second* functions are indistinguishable, but not meaning that both classes are equal since we know nothing about their internal state.

Again, by saying that two classes are indistinguishable we mean that they have the same behaviour for the known observation functions at our dispose, which is the same thing if we state that for an outside observer they preform the same visible functionality.

# 3 The Y86 Processor

The Y86 processor is a theoretic processor developed from a real IA32 architecture, from Intel, very close to a subset of the original X86. Basically it is a simplified version of the IA32 architecture, created aiming to better explain some functionalities and characteristics of a real and more complex processor.

Though the Y86 processor is a simplified version of its older brother, its architecture description is still very detailed and fulfilling about 150 pages of exhaustive processor description.

In our particular case, we are just concerned about some specific aspects of the Y86 architecture, so we will only describe the relevant parts of the architecture for our current issues. For more details about the Y86 processor architecture, see [3].

## 3.1 The Y86 Instruction Set Architecture

The Y86 instruction set contains 12 main instructions, some are real instructions, others represent instruction families like *jXX*, that represents the different jump combinations, *jmp*, *jle*, *jl*, *je*, *jne*, *jge*, and *jg*.

The instruction names are self explanatory about the instruction operations over the registers, but to make it clearer, a brief description of each instruction operations is given below.

There are four integer operation instructions, shown in Figure 1 as *OPl*: *addl*, *subl*, *andl*, and *xorl*. They operate only on register data, whereas IA32 also allows operations on memory data.

The seven jump instructions (shown in Figure 1 as *jXX*) are *jmp*, *jle*, *jl*, *je*, *jne*, *jge*, and *jg*. Branches are taken according to the type of branch and the settings of the condition codes. The branch conditions are the same as with IA32.

The *call* instruction pushes the return address on the stack and jumps to the destination address. The ret instruction returns from such a *call*, poping from the stack the return address.

The *pushl* and *popl* instructions implement push and pop, just as they do in IA32.

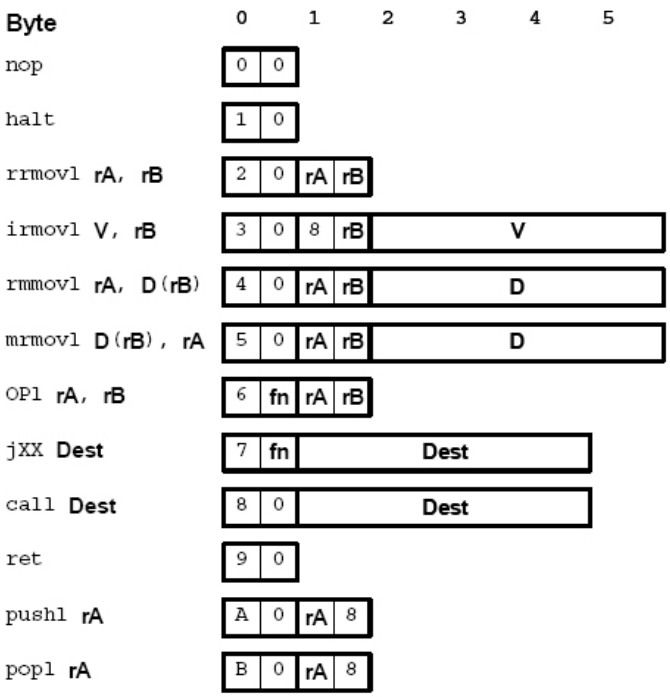


Figure 1: Y86 instruction set. Instruction encodings range between 1 and 6 bytes. An instruction consists of a one-byte instruction specifier, possibly a one-byte register specifier, and possibly a four-byte constant word. Field fn specifies a particular integer operation (OPl) or a particular branch condition (jXX). All numeric values are shown in hexadecimal. (Courtesy of Pearson Education, Inc)

The halt instruction stops instruction execution. IA32 has a comparable instruction, called *hlt*. IA32 application programs are not permitted to use this instruction, since it causes the entire system to stop. We use halt in our Y86 programs to stop the simulator.

As shown in Figure 3, each of the eight program registers has an associated register identifier (ID) ranging from 0 to 7. The numbering of registers in Y86 matches what is used in IA32. The program registers are stored within the CPU in a register file, a small random-access memory where the register IDs serve as addresses. ID value 8 is used in the instruction encodings when we need to indicate that no register should be accessed.

### 3.2 Programming the Y86 processor

With the above architecture it is very simple to develop Y86 programs. Basically a programmer sees 12 instructions and 8 registers.

We this in mind, we just have to make use of the instructions that we dispose, and reference either registers, constants, or memory locations, according to the specific instruction definition that we are using. By doing this we write one operation over the Y86 processor. Now, if we build a logical sequence of well defined operations, we have an Y86 program.

With this sequence of instructions, the processor will sequentially execute every instruction, according to its place in the sequence. It works as a straight forward

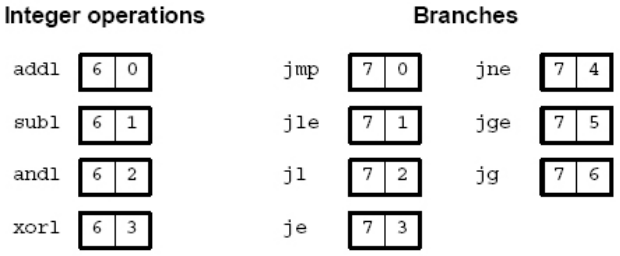


Figure 2: Function codes for Y86 instruction set. The codes specify a particular integer operation or branch condition. These instructions are shown as OPl and jXX in Figure 1.(Courtesy of Pearson Education, Inc)

| Number | Register name |
|---|---|
| 0 | %eax |
| 1 | %ecx |
| 2 | %edx |
| 3 | %ebx |
| 6 | %esi |
| 7 | %edi |
| 4 | %esp |
| 5 | %ebp |
| 8 | No register |

Figure 3: Y86 program register identifiers. Each of the eight program registers has an associated identifer (ID) ranging from 0 to 7. ID 8 in a register field of an instruction indicates the absence of a register operand. (Courtesy of Pearson Education, Inc)

machine, interpreting every instruction in order of appearance in the sequence, only changing this behaviour when running through a *jump* or *branch* instruction. As an example consider the following Y86 program:

```
1   Sum:    pushl %ebp
2           rrmovl %esp,%ebp
3           mrmovl 8(%ebp),%ecx ecx = Start
4           mrmovl 12(%ebp),%edx edx = Count
5           irmovl $0, %eax sum = 0
6           andl %edx,%edx
7           je End
8   Loop:   mrmovl (%ecx),%esi get *Start
9           addl %esi,%eax add to sum
10          irmovl $4,%ebx
11          addl %ebx,%ecx Start++
12          irmovl $-1,%ebx
13          addl %ebx,%edx Count–
14          jne Loop Stop when 0
15  End:
16          popl %ebp
17          ret
```

# 4 Y86 Machine Formal Model

To capture an adequate Y86 model as a working machine, one would like to provide a model capturing the central properties and operations of such a processor. Even more, we would appreciate the model to be able to give us something else, at least the means to formally reason about Y86 programs as a mathematical calculus of instructions. To accomplish this goals we start by defining a Funtor for an Y86 instruction, and then a Functor for an Y86 programm, where the latter relies on the definition of the previous one.

To explain better the model of the Y86 processor, we animate the with Haskell (a functional programming language). This way, every mentioned Functorial definition has its equivalent implementation in Haskell, which can be interpreted[2] with a Haskell interpreter like Hugs.

## 4.1 Y86 Instructions

There is a common denominator to all the instructions of the Y86 processor, presented above. They all take the processor from one state to another, performing something in the processor internal state. Like so, every operation is an instance of the following signature

$$Inst : S \rightarrow S$$

where $S$ denotes the Y86 processor state, from whom we are not concerned to know any details for now.

At a deeper level we can observe different kinds of behaviours in the Y86 processor instructions. Some of them take some arguments in order to compute a new state, while others need any kind of arguments to execute.

[2]For more info about the Haskell language see http://www.haskell.org

A mathematical way of defining this idea, of getting some arguments in order to produce a new processor state, is making use of the exponential functors, resulting in the next result

$$Inst : A \to S^S$$

meaning that this instruction needs some input of kind $A$ in order to give a state transition function $S^S \cong S \to S$. So $Inst(a), a \in A$ is a simple transition function of our known kind $S \to S$, which can be applied to a processor sate and make it evolve to another state.

Again, doing the same exercise we have been doing to formulate these Y86 instructions definitions, we can analyse the kind of our last instructions arguments, that we have simplified to hold a certain kind $A$.

If we look at our instructions, then we realize that we can group them based on the input we have to supply to each one. With this basis we can identify two different aspects of passing input to an instruction, a first one regarding the number of inputs, and a second one regarding their type.

As an approach to defining more then one argument in an instruction, we can define it like

$$Inst : \prod^n A \to S^S$$

where $n \in N$ is the number of arguments our instruction takes.

About the second issue, the different types of our arguments, we can solve it by using several type variables.

Combining all this analysis, we end up with with the following instruction definition

$$Inst : \prod^n A_i \to S^S$$

where the index $i \in N$ defines the different possible types of each instruction input.

Equivalent Haskell definitions can be found at
http://gec.di.uminho.pt/micei/ac0304/icca04/10920_y86haskell.pdf.

### 4.2 Y86 States

In order to specify the Y86 state, in the way we have been leading our definitions, we just need to concerned abut the visible outside effects of such a state.

Like in a digital watch, where we are just concerned about knowing the current time in a specified moment, and not concerned about the internal mechanisms that make it possible to tell the time in a precise moment.

For an outsider and as far as we are concerned, inside a digital clock there are 24 possible values for the hours and 60 possible values for the minutes, and in a certain moment there can be only one value for each of the fields hour and minutes. Everything else are technological problems of the digital clock implementation.

As in the clock example, about the Y86 processor internal state, we are just concerned about representing eight general use registers. Again, and for now, everything else is electronic sugar. This approach would lead us to the following definition for the Y86 State

$$F(X) \cong A^8 \cong A \times A \times A \times A \times A \times A \times A \times A$$

Where $A$ represents the type of a registry.

The equivalent implementation in Haskell is

```
data Y86State = St { regs :: [Double],
                     mach :: [Y86Operation Y86State ()]
                   }
```

where the tag `regs` identifies a list[3] of registries, which in our case has a static length of 8.

### 4.3 Y86 program

Like we have said before a program for the Y86 is merely a sequence of instructions, so here we keep that idea while defining our data type for the Y86 Program. As so, we define it like

$$F(X) \cong Inst^* \cong 1 + Inst \times F(X)$$

which is a simple list of instructions having the already discussed instruction type.

Notice that every instruction in this list has a well determine position inside the instructions sequence. This will be of very importance in the next section where we define the Y86 processor simulator, and will need to perform jumps inside the Y86 Program.

### 4.4 The Y86 Simulator

The Y86 simulator is an attempt to animate the presented definitions, by seing the machine interpreting the various instructions in the same way the Y86 would work. In order to do so, we need to define how the sequence of instructions, defined in 4.3, will be interpreted.

Following our mathematical way of reasoning about the Y86 processor, we will use another mathematical structure, Monads[6] , on top of which we will define the Y86 simulator.

The particular case of Monads that we are interested here is of Monads regarding Functors. The main property that we will exploit out of Monads and at the same time the reason why we use them, is the existence of a function $\mu : F^2(A) \to F(A)$ that somehow flattens our data type and a unit function $u : A \to F(A)$.

With this two functions we can make use of the kleisli composition, which tells us that if have two functions $f : B \to F(C)$ and $g : A \to F(B)$ that can't be composed linearly, we can indeed compose them by using the following definition

$$f \bullet g = \mu \cdot F(f) \cdot g$$

The great thing about this result is that now we can use Kleisli composition to our sequence of Instructions, as long as we provide a definition for functions $f$ and $\mu$ over our instructions Functor.

Before we do so, there is a Monad already defined and studied that matches perfectly our needs, which is the Sate Monad. This Monad works over the following Functor

[3] The use of list a here is just because of future implementation simplification. The more accurate data structure would be a $8^{th} - Tuple$.

$$F(X) \cong (X \to A \times X)$$

that with some instantiations has the same type of our instructions.

The equivalent definition in Haskell and the respective Monad instance are the following

```
data Y86Operation s v = Op (s -> (v, s))

instance Monad (Y86Operation s)
   where return a     = Op (\x -> (a, x))
         (Op f) >>= g = Op (\s -> let (a, s2) = f s
                                      Op fun  = g a
                                  in fun s2)
```

Now, and has stated, we just have to apply >>= which represents the Kleisli composition in Haskell[4] to the sequence of instructions, having then a single instruction that computes the result (something of type Y86 State) whenever we pass it an initial state, typically a state where the registers are all set to 0.

## 5 Conclusions and Future Work

The attempt to use a formal approach to model the Y86 processor is is just a first step towards a Mathematical way of reasoning over the functionality of a machine.

In this case, we have been able to define a formal model of the Y86 processor, by dissecting the processor in its atomic components and then modelling these simpler components. Its this simple components definition that combined into a single model fulfills the Y86 formal model. This represents the basis of our way of reasoning over the Y86 Processor.

During the model development process of the Y86, several simplifications of the original Y86 processor took place. The main simplifications are, not regarding a RAM memory, not all instructions are implemented and some Processor flags (like ZF, SF and OF) were not taken into account. The implemented instructions are, rrmovl, irmovl, addl, subl, jmp, je and jne. We are aware of this well defined forced simplifications, and many of them could be easily implemented by following the presented modelation process.

Different mathematical approaches could have been used to model the Y86 Processor, with particular gains and losses in certain points. Just to elucidate the reader, some of the possible approaches could have been through Turing Machines, Graph Theory, Grammar Theory, Logic.

The selected approach is centered in Coalgebras, and this is due to the announced characteristics of Coalgebras like general representation of objects inside the Coalgebra, properties proof over the entire Coalgebra, easy parallel implementation of the mathematical model in a functional programming environment.

In this article we have only focus the modelling of the Y86 processor, the next interesting step would be to define a Bissimulation relation between Y86 programs in order to derive an equality relation between Y86 Programs. With this equality relation

[4] The full Haskell definition of the Y86 model is at
http://gec.di.uminho.pt/micei/ac0304/icca04/10920_y86haskell.pdf

we mean not a simple list comparison, which is the underlying structure of our Y86 programs, but a more elaborate way of comparing Y86 programs where for example, the two following programs would be the same, since they both

```
prog1 = [ opIRMovl 4.0 1, opIRMovl 5.0 2, opAddl 1 2 ]

prog2 = [ opIRMovl 4.0 1, opIRMovl 5.0 3, opAddl 1 3,
          opRRMov 3 2 ]
```

produce the same observable final state.

Another important issue, would be to define a calculus over the presented model to calculate new programs equivalent to original ones but with less instructions.

A modification of the model to introduce a weight[5] in every instruction, and then use it to derive new optimized programs from previous ones would also be very much welcome as future work over the model.

## References


[1] Luís M. D. C. Soares Barbosa. *Components as Coalgebras.* PhD thesis, Universidade do Minho, 2001.

[2] R. Bird. *Functional Programming Using Haskell.* Series in Computer Science. ph, 1998.

[3] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2002.

[4] H. Peter Gumm. Elements of the general theory of coalgebras. Technical report, Lecture Notes for LUTACS'99, South Africa, 1999.

[5] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–159, 1997.

[6] J. N. Oliveira. A look at monads. Chapter of Book, 2000.

[7] N. Rodrigues and L. S. Barbosa. On the specification of a component repository. In Z. Liu, editor, *Proc. of FACSt03, (*Formal Approaches to Component Software*)*, Pisa, Spetember 2003.

[8] J. Rutten. Universal coalgebra: A theory of systems. Technical report, CWI, Amsterdam, 1996.


[5] This weight can point different physical realities, like time to execute, power consumption, heat generated, etc.