Abstract—In this paper we present a model of computer architecture. Several examples demonstrating the use of the model to describe the architectures of real and virtual computers are provided. The proposed model uses a triplet to describe a computing machine's static structures; the more dynamic parts of the machine's architecture are expressed using the relations between the triplet components. The model's recursive formalism allows for the architecture of an entire machine and its components to be represented hierarchically.

Keywords: computer architecture; formal models; instruction sets; microprogramming; reconfigurable computing; virtualization

I. INTRODUCTION

Thanks in large measure to Alan Turing, the theory of computing is now a well-established area in computer science and a wide array of abstract computer models, such as finite-state machines, random-access machines, push-down automatoms and Turing machines can now be proposed and analyzed. However, the same cannot be said when it comes to the theoretical foundations for the engineering areas involved in the design and realization of these computing devices. The design of general-purpose computers has, until recently, been largely a qualitative exercise [1]. The work by Flynn [2] and Hennessy and Patterson [3] has helped start a reversal of that trend by putting the design of computers on a quantitative footing. Along those lines, in this work we propose to formalize the description of computer architectures. We develop a generic model for computing architecture that has the following features:

- **fidelity** — it accurately represents the structural parts of a given architecture and how they are related, while at the same time it allows for abstract computer models to represent the operations of these parts and their interactions.

- **accessibility** — it is an intuitive, algebraic model that can be used by computer architects and other designers of digital systems with little training in formal mathematical methods.

- **extensibility** — it can be used to model entire systems or the individual components of such systems.

The rest of the paper is organized as follows. Section 2 describes the larger research project of which this work is a part. It also introduces some related works and defines some key terms. Section 3 describes the development of the simple model, together with an example of its application. Section 4 extends the simple model into the general model and it also provides a small example. Section 5 applies the generic model to the computer design techniques of microprogramming and virtualization. Section 6 describes some future work. Section 7 concludes the paper.

II. BACKGROUND AND RELATED WORK

A. Background

The work presented here is part of the author's doctoral research project that is focused on the development of a toolchain for automatic synthesis of application-specific processors that are then implemented on reconfigurable computing devices. This paper reports on the progress made so far in providing a formal description of the application-specific processors that are automatically generated by the toolchain.

A major goal of this part of the research is to develop a general computer architecture model that can be used to describe computing machines wholly or in part, as shown in Fig. 1. The parts that that are demonstrated in this paper are shown as blocks with a shaded background in the figure.

![Figure 1](https://via.placeholder.com/150)
It is important to note that our proposed model only addresses the logical level of computer architecture, that is, it deals with the functional logic of the components. The implementation level is technology-dependent and it involves the production of the physical devices that realize the functions described at the logical level. Once an architecture is fully described using our formalism, its model is transcribed using a standard hardware description language (HDL), such as Verilog or VHDL. The HDL model can then be implemented using current technologies.

B. Related Work

The term "computer architecture" was first used to describe the attributes of the IBM System/360 as seen by the programmer [4][5], today this aspect of a computer’s design is commonly known as its instruction set architecture (ISA). Over time the concept of computer architecture has grown to be more encompassing. Mudge [5], defines computer architecture as the ISA together with its implementation using hardware components. He adds that computer architecture influences and is influenced by the existing technology, the applications targeted to run on the computer, and other constraints such as costs, compatibility and the marketplace. Hennessy and Patterson [3], define computer architecture as the design specifications for a computer, which include the description of its: (1) ISA, (2) microarchitecture, also known as computer organization, and (3) hardware. These design specifications or blueprints when implemented should result in a computer that maximizes performance while subject to constraints, such as costs and power. In this paper we are going to use the definition of computer architecture by Hennessy and Patterson. When dealing with physical machines, the I and D components of our proposed model relate to the ISA part of this definition, while the R part of our model relates to the microarchitecture and the hardware descriptions.

A constructive computation-based theoretical framework for modeling the underlying structures of computer architecture is presented by Albrecht [6]. While this framework is generic, it has some limitations in that it is not intuitive and is mainly focused on modeling the operations of the components. Furthermore, it is only accessible to computer architects with advanced mathematical training in formal models.

In the literature the architecture of physical computers and virtual computers are treated as a separate subjects [3],[7]. Given the growing importance of virtualization in the computer industry we are of the view that a framework which seamlessly handles both physical and virtual computer architectures will be advantageous. Chen et al. [8] proposed a virtual machine model that extends an existing model that is used for real machines. This is a state machine-based model that does not easily lend itself to modeling the architecture of the primary objects in our larger research effort - application-specific processor cores on a reconfigurable computing fabric.

Reconfigurable computing refers to "systems incorporating some form of [runtime] hardware programmability - customizing how the hardware is used using a number of physical control points. These control points can then be changed periodically in order to execute different applications using the same hardware" [9]. We are looking for a computer architecture model that takes into account the possible use of reconfigurable computing resources. Sima et al [10], put forward an architectural-based taxonomy for field-programmable devices. Their taxonomy introduced a recursive formalism (similar to Flynn’s requestor/server formalism [11]) that is based on microcode in order to abstract away any references to a particular ISA. This formalism defines a computing machine (CM) as a doublet, consisting of a microprogram (μP), and a set of resources (R); that is CM={μP,R}. Our proposed computer architecture model extends this formalism.

III. FORMULATING THE SIMPLE MODEL

For the development of the first iteration of our model we are going to use, as an example, a simple microprocessor that has a hard-wired control unit without pipelining and no other sophisticated microarchitecture. Ways to add new microarchitectural features to the processor model will be discussed later, in the Applications section. We will call this simple microprocessor kk55. The simple model will follow along the lines of the one proposed by Sima et al. [10], however we make significant extensions to that model in order to facilitate later generalizations. Our model uses a triplet instead of a doublet as the primary structure and we introduce the use of relations to describe the relationship between the triplet elements.

Let us assume that we have a computing machine (CM) which consists of a set of N computing resources (R) that use a set of M operations or instructions (I) to operate on a set of P data types (D). An implementation of a CM can be formalized by means of a triplet:

\[ CM = \{I, R, D\} \]  

(1)

For the instruction set:

\[ I = \{i_1, i_2, ..., i_M\} \]  

(2)

Each instruction, \( i_k \), where \( 1 \leq k \leq M \) is typically represented by an instruction mnemonic or an opcode.

![Figure 2. Instruction to resources relation](image-url)
Similarly, for the computing resources or functional units:

\[ R = \{ r_1, r_2, \ldots, r_N \} \]  \hspace{1cm} (3)

A resource, \( r_k \), where \( 1 \leq k \leq N \) may represent an adder, a register, or some other physical component of the microprocessor.

Each instruction, \( i_s \), controls one or more resources, as shown in Fig. 2. That is, each instruction controls a set of resources, \( R_s \), where:

\[ R_s \subseteq R \]
\[ \left| R_s \right| > 0 \]  \hspace{1cm} (4)

Let \( S_s \) represents the relation between \( i_s \) and \( R_s \) - that is:

\[ S_s = \{ (i_s, r_{n_i}), (i_s, r_{n_2}), \ldots, (i_s, r_{n_N}) \} \]  \hspace{1cm} (5)

Now considering the data types:

\[ D = \{ d_1, d_2, \ldots, d_P \} \]  \hspace{1cm} (6)

Each data type, \( d_j \), where \( 1 \leq k \leq P \), represents a data format or addressing mode. Data types may represent immediate data, or indirect data. Immediate data is embedded in the instruction and as such is available for immediate processing by the computing resources, while indirect data represents a location where the computing resources can find the data to be processed as part of the instruction execution.

Each resource, \( r_k \), can operate on zero or more data types, Fig. 3. That is, each resource can process a set of data types, \( D_k \), where:

\[ D_k \subseteq D \]
\[ \left| D_k \right| \geq 0 \]  \hspace{1cm} (7)

Let \( T_k \) represent the relation between \( r_k \) and \( D_k \) - that is:

\[ T_k = \{ (r_k, d_0), (r_k, d_1), \ldots, (r_k, d_p) \} \]  \hspace{1cm} (8)

Putting the component of the triplet together, as shown in Fig. 4, we see that the composition \( S_k \circ T_k \) allows us to express the relationship that exists between the instruction \( i_s \) and data of type \( d_k \) whenever there exists a suitable resource \( r_k \):

\[ i_s (S_k \circ T_k) d_k \iff \left( \exists r_k \right) \left( (i_s, r_k) \land (r_k, d_k) \right) \]  \hspace{1cm} (9)

Suppose \( bOp \) is a generic binary operation, such as addition, and that \( i_s \) is a specific instance of that operation acting on two data items of types \( d_{k1} \) and \( d_{k2} \). An example of \( i_s \) would be the addition of an unsigned integer and a floating point number. We have \( i_s \in bOp \) and:

\[ \left[ i_s (S_k \circ T_k) d_{k1} \land i_s (S_k \circ T_k) d_{k2} \right] \Rightarrow (d_{k1} bOp d_{k2}) \]  \hspace{1cm} (10)

Generalizing the result above to \( i_s \) that is an \( n \)-ary operation \( (nOp) \) acting on \( n \) data items, we have:

\[ CM \]

\[ I \rightarrow R \rightarrow D \]

Figure 4. The triplet components and relations
\[
\left[ i_k \left( S_k \circ T_k \right) d_k \right] \land \ldots \land \left[ i_k \left( S_k \circ T_k \right) d_k \right] \Rightarrow nO\left( d_k, \ldots, d_k \right)
\] (11)

The distribution of processor specifications in a format consistent with relationships (9), (10) and (11) should facilitate automated compiler construction for new architectures.

A. Simple Model Example

In this Section we apply the model to k85, a simple 8-bit microprocessor that is binary-compatible with the Intel 8085. The architecture diagram of k85 is shown in Fig. 5. We assume that the controller for our processor is hardwired and not microprogrammed. At the top-level we describe the processor model as:

\[
CM_{k85} = \{ I_{k85}, R_{k85}, D_{k85} \}
\] (12)

where:

\[
I_{k85} = \{ ANA, MOV, CALL, \ldots, XTHL \}
\] (13)

using instruction mnemonics, or using opcode templates

\[
I_{k85} = \begin{cases} 
10100.XXX, 01XX\ldots XX, \\
11001101, \ldots, 11100011
\end{cases}
\] (14)

The processor has 59 types of instructions, that is:

\[
M = |I_{k85}| = 59
\] (15)

The width of the data bus is 8 bits; we will use that as the default size of each resource. Any resource with a different size will be shown with its size in parenthesis next to the resource name. Using Fig. 5, we can put together \( R_{k85} \) as:

\[
R_{k85} = \{ ALU, Control Unit, Register File, \ldots \}
\] (16)

where Register File is a macro for an array of all the registers in the processor, that is:

\[
\text{Register File} = A, B, C, D, E, Flags, H, L, IR, PC(16), SP(16), Temp, W, Z
\] (17)

Some registers or combinations of registers are directly available to \( I_{k85} \) and these represent data types. In the case of our processor \( REG \) is the set of available registers, while \( REG16 \) consists of overlapping register pairs or other 16-bit registers that are available to \( I_{k85} \).

\[
REG = \{ A, B, C, D, E, H, L \}
\] (18)

and

\[
REG16 = \{ BC, DE, HL, SP(16) \}
\] (19)

Now we can put together the data types for our processor.

\[
D_{k85} = REG \cup REG16 \cup \{ \text{address}(16), \text{data}, \text{data}(16), \text{port} \}
\] (20)

where data and port are any 8 bit numbers representing data or a port respectively. While, data(16) and address(16) are any 16 bit numbers representing data or a memory address respectively.
Next the relations $S$ and $T$ are specified. Let us consider the ANA instruction group, that is, $k = \text{ANA}$ in (5) and (8).

The ANA instruction has two options:

**ANA register** - the register is ANDed with the A register and the result is stored in A.

**ANA M** - the data in the memory location pointed to by the contents of the HL register is ANDed with the A register and the result is stored in A.

$$S_{\text{ANA}} = \{ (\text{ANA}, \text{ALU}), (\text{ANA}, \text{Control Unit}), (\text{ANA}, \text{IR}), (\text{ANA}, \text{Decoder Unit}), (\text{ANA}, \text{REG}), (\text{ANA}, \text{HL}) \}$$ (21)

and

$$T_{\text{ANA}} = \{ (\text{ALU}, \text{REG}), (\text{ALU}, \text{data}), (\text{ALU}, \text{Flags}), (\text{PC}, \text{address}(16)), (\text{Control Unit, REG}), (\text{Decoder Unit, data}) \}$$ (22)

IV. THE GENERAL MODEL

We generalize our model by transforming (1) into a recursive formalism. Our simple model may be viewed as a computing object consisting of three related components: the program or ordered set of instructions that direct some computing resources to act on some data. The computing object can be represented by the triplet:

$$\text{computing object} = \{ \text{program, resources, data} \}$$ (23)

In the general model, parts of $I$, $R$ and $D$ from (1) may be replaced by models for computing objects that each have a format corresponding to (23). We represent the general model of the computing machine using the following triplet:

$$\text{architecture CM} = \{ \text{architecture I}, \text{architecture R}, \text{architecture D} \}$$ (24)

and the relations as:

$$\text{architecture} S \text{ and architecture} R$$ (25)

The implementation reference level (IRL) is defined as $\text{level} = 0$ and it can be set arbitrarily. However, it is preferable to set the IRL as close to the primary computing device being modeled, in this way its components will appear at $\text{level} > 0$ and any aggregates structures or networks using the device will appear at $\text{level} < 0$.

A. General Model Example

Let us consider the ALU from our previous example as an assembly of a multiplexer together with the following six 8-bit circuits that implement the **Adder, Shifter, AND, OR, XOR and NOT** functions, as shown in Fig. 6. The MUX along with its input and output signals forms the CPU's control plane, while the 8-bit circuits along with their data input and output form the CPU's datapath.

The ALU can now be modeled as:

$$\text{MUX}^2_{\text{OpSelLUT}} \text{MUX}^3_{\text{OpModules}}$$ (26)

where $\text{MUX}^2_{\text{OpSelLUT}}$ is the set of codes used to direct the MUX to select the appropriate function circuit, and

$$\text{MUX}^3_{\text{OpModules}} = \{ \text{Adder, Shifter, AND, OR, NOT, XOR, MUX} \}$$ (27)

The $\text{MUX}^3_{\text{ControlSignals}}$ emanate from the Control Unit shown in Fig. 5. We can now rewrite (12) as:

$$\text{MUX}^2_{\text{OpSelLUT}} \{ \text{MUX}^2_{\text{OpSelLUT}}, \text{MUX}^2_{\text{OpModules}}, \text{MUX}^2_{\text{ControlSignals}} \}$$ (28)

where

$$\text{MUX}^2_{\text{ControlUnit}}$$ (29)

V. APPLICATIONS

Our proposed model can be used to either:

a) Describe the architecture of an entire computing machine, as the virtualization application example below shows; or

b) Describe an optimized part of an existing machine, as demonstrated by the microprogramming application example below.

![Figure 6. The Arithmetic Logic Unit (ALU)](image-url)
An existing lower (numerical) level model may be extended by 'plugging' into it the model of the new higher level component, and conversely.

In general the proposed model can be used in the following manner. Identify a potential application and generate a triplet covering the whole application area. If required, interface the new generic model to an existing higher- or lower-level model. Then, define the relationships between the triplet components.

A. Microprogramming

While hard-wired processors, such as the one in our microprocessor example above, may offer a performance advantage over their microprogrammed counterparts most commercial microprocessors today are microprogrammed. Microprogramming offers the following advantages when compared to hard-wired architectures: ease of development and maintenance, flexibility, and lower costs [12].

A microprogram is a sequence of microinstructions that are not directly accessible to the programs running on the machine. Each microinstruction corresponds to a primitive operation that the machine can perform, often referred to as a micro-operation. The microinstructions are often described using register-transfer level. A processor’s programmer-visible instruction can then be described by a microprogram, as the example of the LHLD instruction from the k85 architecture shows.

\[
\begin{align*}
LHLD \text{ addr:} & \quad t_1, Z \leftarrow M[PC] \\
& \quad PC++ \\
& \quad t_2, W \leftarrow M[PC] \\
& \quad PC++ \\
& \quad t_3, L \leftarrow M[WZ] \\
& \quad WZ++ \\
& \quad t_4, H \leftarrow M[WZ]
\end{align*}
\]

(30)

The microprogrammed control unit can be implemented using control memory, a control address register and a next address generator unit [13], as shown in Fig. 7. Each microinstruction is stored as a word in the control memory.

We model the microprogrammed control unit as the triplet:

\[
\begin{align*}
^2\text{Control Unit} = \{ & \{^3I_n, ^3R_n, ^3R_a\}, \{^2D_n, ^3D_n, ^2D_a\} \}
\end{align*}
\]

(31)

where $^2D_n$ and $^2D_a$ map one-to-one with the Control Unit input and output signals from Fig. 5, respectively. $^3I_n$ is a hardware circuit that generates the next address to be latched into $^3R_n$ based on $^3D_n$ and $^2D_n$. Now, model (31) can be substituted into model (29).

The discussion above, as well as the model (31), can be applied to co-designed virtual machines. By way of contrast, Chen et al. provide state machine-based model [8] of such machines. In a co-designed virtual machine the source architecture, that is the one visible to the binary applications running on the machine, is emulated on a target architecture. One of the most well-known co-designed virtual machines is the Transmeta Crusoe processor [14], which uses a ‘code morphing’ (CMS) layer [15] to transparently run Intel IA-32-based software (source architecture) on an underlying VLIW (Very Long Instruction Word) architecture (target architecture). Using our model, this CMS layer can be implemented in $^3I_n$, in order to maximize performance or it can be implemented in $^2R_n$, in order to maximize design flexibility and post-production maintenance.

B. Virtualization

Virtualization is formally described as a one-one homomorphism between a ‘real’ system and a ‘virtual’ system, with respect to all the operators in an instruction sequence set [16]. That is, for any state transformation in the ‘real’ system and equivalent transformation can be performed in the ‘virtual’ system. One realization of virtualization is through virtual machines (VM). A VM is a software layer that emulates a desired machine’s architecture [7]. The VM executes (runs) on a real machine whose architecture may or may not be the same as that emulated by the VM.

We will model the generic virtual machine as:

\[
architecture^{\text{VM}} = \{\text{architecture}^I, \text{architecture}^S, \text{architecture}^D\}
\]

(32)

while, the physical or ‘real’ machine is modeled as:

\[
architecture^{\text{PM}} = \{\text{architecture}^I, \text{architecture}^R, \text{architecture}^D\}
\]

(33)

![Figure 7. Microprogramming Control Unit](image)
where \( \text{architecture}\ R \) is a set of programs that emulate corresponding elements of \( \text{architecture}\ R \) in model (33). There are no physical components in \( \text{architecture}\ VM\).

The model for a physical machine that is hosting a VM is:

\[
\text{arch}_1\text{CM} = \{ \text{arch}_1\text{VMM}, \text{arch}_1\text{PM}, \text{arch}_1\text{VM} \}
\]  

(34)

where VMM is the virtual machine monitor or hypervisor.

Traditionally, if \( \text{arch}_1 \neq \text{arch}_2 \) the host machine model above is said to represent a simulation, else, the model represents virtualization.

The host machine model presented above allows us to more easily propose and describe extensions to the existing virtualization technologies. An example is a multi-tenancy hypervisor that can support multiple virtual machines with different architectures. Such a system can be modeled by replacing \( \text{arch}_1\text{VM} \) in (34) with \( \{ \text{arch}_1\text{VM}, ..., \text{arch}_n\text{VM} \} \). In this case we can have, say, an x86 hypervisor that supports ARM, PowerPC and x86 virtual machines modeled as:

\[
\text{x86CM} = \{ \text{x86VMM}, \text{x86PM}, \text{x86VM} \}
\]  

\[
\{ \text{ARMVMM}, \text{PowerPCVMM}, \text{x86VM} \}
\]  

(35)

We note that all the elements of the model triplet can be implemented as hardware or software, depending on the purpose of the model. For example, the \( \text{I} \) triplet-component in the microprogramming example (31) is implemented in hardware, while the \( \text{I} \), \( \text{R} \) and \( \text{D} \) triplet components in the virtualization example (32) are all implemented in software.

VI. FUTURE WORK

In the future we will expand our model to more clearly describe and provide examples of how the relations \( S \) and \( T \) can be used to describe the process part of the architecture that is being modeled. We will also complete the modeling of the other components of Fig. 1 that are not shaded. In the longer term, related research may be undertaken that: (1) uses this model to automatically generate computer architectures for design exploration purposes and (2) automates the generation of compilers for computer architectures described using this model.

VII. CONCLUSIONS

In this paper, we have developed a formal model of computer architecture. We have also shown how the model can be used to describe hard-wired processors, microprogramming and virtualization. The triplet components of the model can be used to represent hardware/resources, programs and the data of the system under design to whatever level of detail is required by the designer. The model requires an understanding of some basic set theory and Boolean logic operators, both of which are almost universally accessible to computer architects and other digital designers. The level notation combined with the recursive nature of the formalism allows the model to be extended by including detailed sub-component triplets or by adding the primary model’s triplet into that of a larger super structure. We conclude by discussing some possible future avenues of research that we believe can follow on from this work. This model combined with the quantitative computer architecture tools mentioned in the Introduction, we believe can help take the design of new computers architectures from an art form [1] into a science form.

REFERENCES