A Model of Computer Architecture with Applications

Charles Mutigwe, Johnson Kinyua, and Farhad Aghdasi

***Abstract*—In this paper we present a model of computer architecture. The proposed model uses a triplet to describe the static structures of a computing machine; the more dynamic parts of the architecture are modeled as relations between the triplet components. A recursive formalism of the model is developed to facilitate the hierarchical representation of an entire machine's architecture or the architecture of its components. A simple, but functional, 8-bit microprocessor architecture is used to show how the formalisms of the model may be applied to existing architectures. Two additional applications of the model are presented to describe microprogramming and virtualization.**

***Index Terms*—Computer architecture, formal models, instruction sets, ISA, microprogramming, virtualization.**

I. Introduction

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 three features.

1) *Fidelity*: The model 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.
2) *Accessibility:* The model 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.
3) *Extensibility:* The model can be used to model entire systems or the individual components of such systems.

Our development of the model presented in this paper looks at both the static and dynamic aspects of a computers architecture. The static aspects are the associations between the components. While the dynamic aspects describe how the connected components interact in order to perform the computations.

The rest of the paper is organized as follows. Section II describes some related work and defines some key terms. Section III describes the development of the simple model, together with an example of its application. Section IV extends the simple model into the general model and it also provides a small example. Section V applies the generic model to the computer design techniques of microprogramming and virtualization. Section VI concludes the paper.

II. 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 computers 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: (i) ISA, (ii) microarchitecture, also known as computer organization, and (iii) 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 will adopt the definition of computer architecture by Hennessy and Patterson. When dealing with physical machines, the 'program' and 'data' components of our proposed model relate to the ISA part of this definition, while the 'resources' 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 it 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 structural differences between architectures.

Manuscript received December 10, 2012; revised January 20, 2013.

C. Mutigwe is with the School of Electrical and Computer Systems Engineering, Central University of Technology, Bloemfontein, South Africa 9320 (e-mail: cmutigwe@ ieee.org).

J. Kinyua is with the School of Computer Information Systems, Virginia International University, Fairfax, VA 22030 USA (e-mail: jkinyua@viu.edu).

F. Aghdasi is with the Faculty of Science and Agriculture, University of Fort Hare, Alice, South Africa 5700 (e-mail: faghdasi@ufh.ac.za).

DOI: 10.7763/IJCTE.2013.V5.775

Sima *et al*. [9] put forward a recursive formalism to model a computing machine. Their model is similar to Flynn's requestor/server formalism [10]. It defines a computing machine *(CM)* as a doublet, consisting of a microprogram *(μP)*, and a set of resources *(R)*; that is *CM= (μP, R)*. This model uses the microcode in order to abstract away any references to a particular ISA. Our proposed computer architecture model extends this formalism by adding a third component that will be used to model the data processed by the computing machine. This data component will facilitate the modeling of virtual devices and operations on inputs with different data types.

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 the ***k85***. The simple model will follow along the lines of the one proposed by Sima *et al*. [9], 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 *Q* 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:

$$CM = (I, R, D) \tag{1}$$

For the instruction set,

$$I = \{i_1, i_2, \ldots, i_Q\} \tag{2}$$

each instruction, i_k, where $1 \le k \le Q$, is typically represented by an instruction mnemonic or an opcode. Similarly, for the computing resources or functional units

$$R = \{r_1, r_2, \ldots, r_N\} \tag{3}$$

A resource, r_k, $1 \le k \le N$, may represent a component of the microprocessor, such as, an adder or a register.

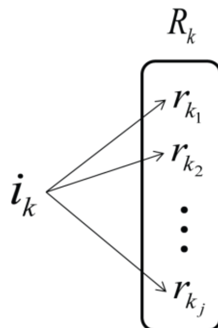

Fig. 1. Instruction to resources relation.

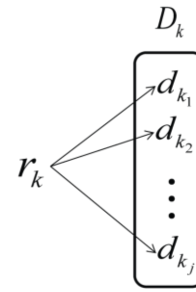

Fig. 2. Resources to data relation.

Each instruction, i_k, controls one or more resources, as shown in Fig. 1. That is, each instruction controls a set of resources, R_k, where:

$$R_k \subset R \quad \text{and} \\ |R_k| > 0 \tag{4}$$

We introduce a set of triggers, T, where these triggers are state transitions that are used to initiate other processes. An example of such triggers is a set of X sequential rising edges of a clock signal that are numbered and represented as $\{clock_1, clock_2, \ldots, clock_X\}$.

$$T = \{t_1, t_2, \ldots, t_X\} \tag{5}$$

Let U_k represents the relation between i_k, R_k and, where $T_k \subset T$ and it represents the set of triggers needed to implement instruction i_k. We are now able to model U_k as a set of triplets, where each instruction's resource pool can now be scheduled up to the point when each element in the pool is required. That is:

$$U_k = \{(i_k, r_{k_1}, t_{k_1}), (i_k, r_{k_2}, t_{k_2}), \ldots, (i_k, r_{k_N}, t_{k_X})\}. \tag{6}$$

Following a similar line of development, we now consider the data types, where

$$D = \{d_1, d_2, \ldots, d_P\}. \tag{7}$$

Each data type, d_k, where $1 \le k \le P$, 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. 2. That is, each resource can process a set of data types, R_k, where:

$$D_k \subset D \quad \text{and} \\ |D_k| \ge 0 \tag{8}$$

Let V_k represent the relation between r_k, D_k and T_k – that is:

$$V_k = \{(r_k, d_{k_1}, t_{k_1}), (r_k, d_{k_2}, t_{k_2}), \ldots, (r_k, d_{k_N}, t_{k_X})\} \tag{9}$$

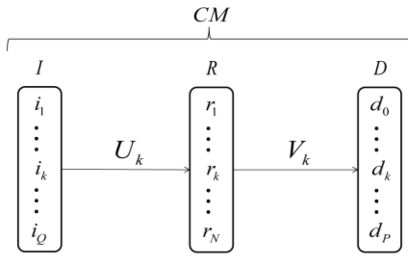

Fig. 3. The triplet components and relations.

Putting the component of the triplet together, as shown in Fig. 3, we see that the composition $U_k \circ V_k$ allows us to express the relationship that exists between the instruction i_k and data of type d_k whenever there exists a suitable resource r_k at interval t_j, that is

$$i_k (U_k \circ V_k) d_k \Leftrightarrow \{\exists r_k \exists t_j | (i_k U_k r_k) \wedge (r_k V_k d_k)\} \quad (10)$$

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

$$\left[i_k (U_k \circ V_k) d_{k_1} \wedge i_k (U_k \circ V_k) d_{k_2} \right] \Rightarrow \left(d_{k_1}\ bOp\ d_{k_2} \right). \quad (11)$$

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

$$\begin{bmatrix} i_k (U_k \circ V_k) d_{k_1} \\ \wedge \ \dots \ \wedge \\ i_k (U_k \circ V_k) d_{k_n} \end{bmatrix} \Rightarrow nOp\left(d_{k_1}, \dots, d_{k_n} \right). \quad (12)$$

The distribution of processor specifications in a format consistent with relationships (10), (11) and (12) should facilitate automated compiler construction for new architectures.

Next, we present a way to model the dynamic aspects of a computer's architecture. The resources needed to execute each instruction, as shown in Fig. 1, are often marshaled using a sequence of micro-operations that are ordered by the elements of the trigger set T . We introduce a set of micro-operations, M , where

$$M = \{m_1, m_2, \dots, m_Y\} \quad (13)$$

For each instruction, i_k, there is a corresponding set of micro-operations, M_k, where $M_k \subset M$, Each micro-operation, m_{k1}, where $m_{k1} \subset M_k$, accesses at most two resources, r_{ki1} and r_{ki2}, for inputs. Each micro-operation places its result in one resource, r_{ko}, at most. Now, we are able to model W_k, the operations of each instruction, i_k over the interval T_k, as a sequence of triplets shown in (14). Now,

$$W_k = \begin{pmatrix} (m_{k_1}, R^w_{k_1}, t_{k_1}), (m_{k_2}, R^w_{k_2}, t_{k_2}), \\ \dots, (m_{k_X}, R^w_{k_X}, t_{k_X}) \end{pmatrix} \quad (14)$$

where

$$R^w_{k_i} = (r_{k_{i1}}, r_{k_{i2}}, r_{k_o})$$

A. Example of the Simple Model

In this section we apply the model to the ***k85***, which is a simple 8-bit microprocessor that is binary-compatible with the Intel 8085. The architecture diagram of the ***k85*** is shown in Fig. 4. We assume that the Control Unit for our processor is hardwired and not microprogrammed. At the top-level we describe the processor model as

$$CM_{k85} = (I_{k85}, R_{k85}, D_{k85}) \quad (15)$$

where

$$I_{k85} = \{ANDA, MOV, CALL, \dots, XHL\} \quad (16)$$

using instruction mnemonics, or using opcode templates

$$I_{k85} = \begin{Bmatrix} 10100XXX, 01XXXXXX \\ 11001101, \dots, 11100011 \end{Bmatrix} \quad (17)$$

The processor has 59 types of instructions, that is:

$$M = |I_{k85}| = 59 \quad (18)$$

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. 4, we can put together R_{k85} as

$$R_{k85} = \{ALU, Control\,Unit, Register\,File, \dots\} \quad (19)$$

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

$$Register\,File = \begin{matrix} A,B,C,D,E,Flags,H,L,IR, \\ PC(16),SP(16),Temp,W,Z \end{matrix} \quad (20)$$

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 8-bit registers, while $REG16$ consists of overlapping register pairs or other 16-bit registers that are available to I_{k85}.

$$REG = \{A, B, C, D, E, H, L\} \quad (21)$$

and

$$REG16 = \{BC, DE, HL, PC(16)\} \quad (22)$$

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

$$D_{k85} = \begin{matrix} REG \cup REG16 \cup \\ \{address(16), data, data(16), port\} \end{matrix} \quad (23)$$

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 ANDA instruction group, that is, $k = ANDA$ in (6) and (9). The ANDA instruction has two options:

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

ANDA *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.

In RTL, the actions performed by the **ANDA** instruction are described as follows:

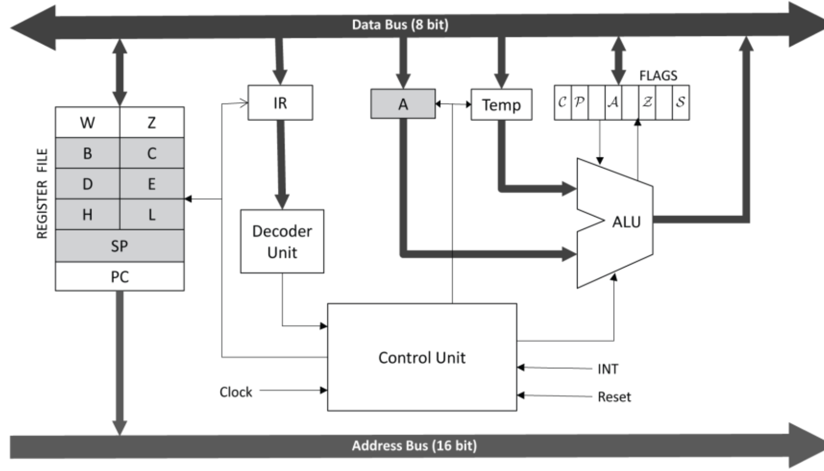

Fig. 4. The k85 processor architecture diagram.

$$
\begin{aligned}
ANDA\ register:\ t_1\quad & A \leftarrow A \wedge register \\
& Z \leftarrow 1 \\
& S \leftarrow 1 \\
& P \leftarrow 1 \\
& C \leftarrow 0 \\
& A \leftarrow A\langle 3\rangle \vee register\langle 3\rangle
\end{aligned}
$$

where $register$ is replaced by M for the second option and the number between each angled bracket represents a bit position. A, C, P, S and Z represent the individual flags in the *Flags* register. We can now model how this instruction operates as

$$
U_{ANDA} = \left\{ \begin{array}{l} (ANDA, ALU), (ANDA, REG), \\ (ANDA, HL), (ANDA, Control\,Unit), \\ (ANDA, IR), (ANDA, Decoder\,Unit) \end{array} \right\} \tag{24}
$$

$$
V_{ANDA} = \left\{ \begin{array}{l} (ALU, REG), (ALU, data), (ALU, Flags), \\ (PC, address(16)), (Control\,Unit, REG), \\ (Decoder\,Unit, data \end{array} \right\} \tag{25}
$$

$$
W_{ANDA} = \left(\begin{array}{l} \left(\wedge, (A, register, A), t_1\right), \\ \left(, (1,, Z), t_1\right), \\ \qquad \ldots \\ \left(\vee, (A\langle 3\rangle, register\langle 3\rangle, A), t_1\right) \end{array} \right). \tag{26}
$$

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:

$$
computing\ object = (instructions, resources, data) \tag{27}
$$

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 (27). We represent the general model of the computing machine using the following triplet:

$$
{}_{architecure}^{level}CM = \left({}_{architecture}^{(level+1)}I,\ {}_{architecture}^{(level+1)}R,\ {}_{architecture}^{(level+1)}D \right) \tag{28}
$$

and the relations as:

$$
{}_{architecture}^{(level+1)}U \quad \text{and} \quad {}_{architecture}^{(level+1)}V \tag{29}
$$

The implementation reference level (IRL) is defined as $level = 0$, as shown in Fig. 5. The IRL can be set arbitrarily, however, it is preferable to set the IRL close to the primary computing device being modeled. In this way the components of the device being modeled will appear at $level > 0$ and any aggregates structures or networks using the device will appear at $level < 0$.

A. Example of the General Model

Let us consider the ALU from our previous example as an assembly of a multiplexor together with 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 processor's *control plane*, while the 8-bit circuits along with their data input and output form the processor's *datapath*.

The ALU can now be modeled as

$$
{}_{k85}^{2}ALU = \left(\begin{array}{l} {}_{MUX}^{3}OpSelLUT, \\ {}_{MUX}^{3}OpModules, \\ {}_{MUX}^{3}ControlSignals \end{array} \right) \tag{30}
$$

where ${}_{MUX}^{3}OpSelLUT$ is the set of codes used to direct the MUX to select the appropriate function circuit, and

$$
{}_{MUX}^{3}OpModules = \left\{ \begin{array}{l} Adder, Shifter, AND, \\ OR, NOT, XOR, MUX \end{array} \right\} \tag{31}
$$

The ${}_{MUX}^{3}ControlSignals$ emanate from the Control Unit shown in Fig. 4. We can now rewrite (28) as

$$
{}_{k85}^{0}CM = ({}_{k85}^{1}I,\ {}_{k85}^{1}R,\ {}_{k85}^{1}D) \tag{32}
$$

where

$$ {}_{k85}^{1}R = \left\{ {}_{k85}^{2}ALU, {}_{k85}^{2}ControlUnit, \ldots \right\} \quad (33) $$

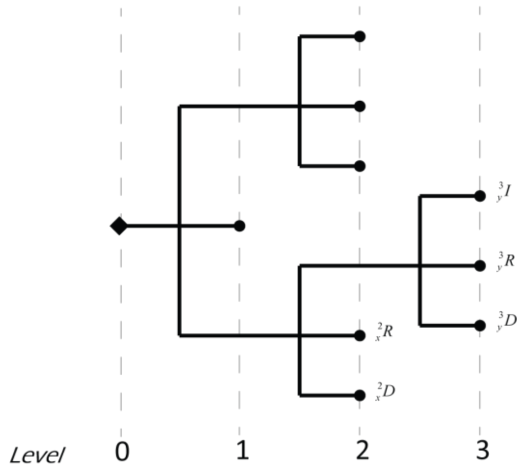

Fig. 5. Tree-based representation of the generic model.

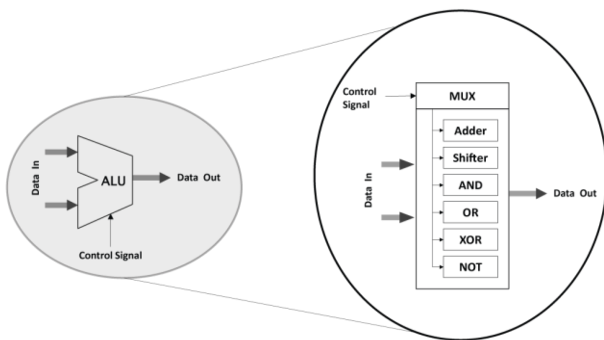

Fig. 6. The arithmetic logic unit (ALU).

V. Applications

Our proposed model can be used to either:

1) Describe the architecture of an entire computing machine, as the virtualization application example below shows; or to
2) Describe an optimized part of an existing machine, as demonstrated by the microprogramming application example below.

An existing lower (numerical) level model may be extended by 'plugging' into it the model of a new higher level component, as shown in Fig. 5.

In general the proposed model can be used in the following manner. Firstly, identify a potential processor application or optimization scheme. Next, generate a triplet covering the whole application area. If required, interface this new model to an existing higher- or lower-level model. Finally, 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, many commercial microprocessors today are microprogrammed. Microprogramming offers the following advantages when compared to hardwired architectures: ease of development and maintenance, flexibility, and lower costs [11].

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 LHL instruction from the ***k85*** architecture shows.

$$
\begin{aligned}
LHL\ addr:\ & t_1 \quad Z \leftarrow M[PC] \\
& \quad\ \ PC{+}{+} \\
& t_2 \quad W \leftarrow M[PC] \\
& \quad\ \ PC{+}{+} \\
& t_3 \quad L \leftarrow M[WZ] \\
& \quad\ \ WZ{+}{+} \\
& t_4 \quad H \leftarrow M[WZ]
\end{aligned}
\quad (34)
$$

The microprogrammed control unit can be implemented using control memory, a control address register and a next address generator unit [12], 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:

$$ {}^{2}ControlUnit = \begin{pmatrix} {}^{3}I_n, \left\{ {}^{3}R_a, {}^{3}R_m \right\}, \\ \left\{ {}^{2}D_i, {}^{3}D_n, {}^{2}D_o \right\} \end{pmatrix} \quad (35) $$

where ${}^{2}D_i$ and ${}^{2}D_o$ map one-to-one with the Control Unit input and output signals from Fig. 4, respectively. ${}^{3}I_n$ is a hardware circuit that generates the next address to be latched into ${}^{3}R_a$ based on ${}^{2}D_i$ and ${}^{3}D_n$. Now, model (35) can be substituted into model (33).

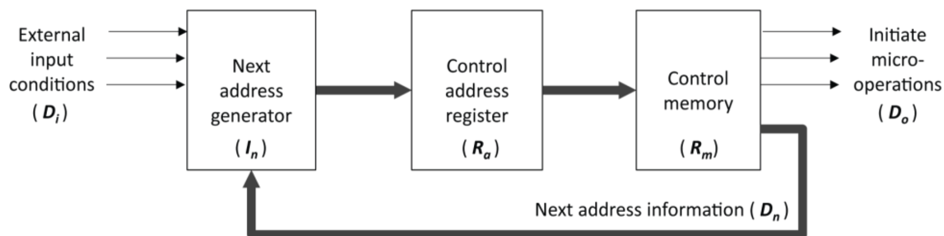

Fig. 7. The arithmetic logic unit (ALU).

The discussion above, as well as the model (35), 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 [13], which uses a 'code

morphing' (CMS) layer [14] 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 ${}^3R_m$, in order to maximize design flexibility and postproduction 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 [15]. That is, for any state transformation in the 'real' system an 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

$$_{architecture}VM = \left(_{architecture}I,\ {}_{architecture}^{soft}R,\ {}_{architecture}D \right) \quad (36)$$

while, the physical or 'real machine is modeled as

$$_{architecture}PM = \left(_{architecture}I,\ {}_{architecture}R,\ {}_{architecture}D \right) \quad (37)$$

where ${}_{architecture}^{soft}R$ is a set of programs that emulate corresponding elements of $architecture^{R}$ in model (37). Note that there are no physical components in $architecture^{VM}$.

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

$$_{arch_1}CM = \left(_{arch_1}VMM,\ {}_{arch_1}PM,\ {}_{arch_2}VM \right) \quad (38)$$

where VMM is the virtual machine monitor or hypervisor.

Traditionally, if $arch_1 \neq arch_2$ 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 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 $arch_2^{vm}$ in (38) with $\left\{ _{arch_1}VM, \ldots, {}_{arch_n}VM \right\}$. In this case we can have, say, an x86 hypervisor that supports ARM, PowerPC and x86 virtual machines modeled as

$$_{x86}CM = \begin{pmatrix} _{x86}VMM,\ {}_{x86}PM, \\ \left\{ _{ARM}VM,\ {}_{PowerPC}VM,\ {}_{x86}VM \right\} \end{pmatrix} \quad (39)$$

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 I triplet-component in the microprogramming example (35) is implemented in hardware, while the I, R and D triplet components in the virtualization example (36) are all implemented in software.

VI. Conclusion

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 programs, hardware/resources and the data of the system under design to an arbitrary level of detail as 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 models triplet into that of a larger superstructure. We believe that this model combined with the quantitative computer architecture tools mentioned in the introduction, can help take the design of new computer architectures from an art into a science.

References

[1] G. A. Blaauw and F. P. Brooks, *Computer Architecture: Concepts and Evolution*, Reading, MA: Addison-Wesley, 1997.

[2] M. J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design*, Boston, MA: Jones and Bartlett, 1995.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA: Morgan Kaufmann, 2011.

[4] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development*, vol. 8, no. 2, pp. 87-101, 1964.

[5] T. Mudge, "Strategic directions in computer architecture," *ACM Computing Surveys*, vol. 28, no. 4, pp. 671-678, 1996.

[6] R. Albrecht, "Modeling of computer architectures," in *Proc. 1st Int. Conf. on Massively Parallel Computing Systems*, pp. 434-442, 1994.

[7] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, San Francisco, CA: Morgan Kaufmann, 2005.

[8] W. Chen, W. Xu, Z. Wang, Q. Dou, Y. Wang, B. Zhao, and B. Wang, "A formalization of an emulation based co-designed virtual machine," in *Proc. 5th Int. Conf. Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 164-168, 2011.

[9] M. Sima, S. Vassiliadis, S. Cotofana, J. T. J. V. Eijndhoven, and K. A. Vissers, "Field-programmable custom computing machines-a taxonomy," in *Proc. Reconfigurable Computing Is Going Mainstream, 12th Int. Conf. Field-Programmable Logic and Applications*, pp. 79-88, 2002.

[10] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. on Computers*, vol. C-21, no. 9, pp. 948-960, 1972.

[11] T. G. Rauscher and P. M. Adams, *Tutorial: Microprogramming and Firmware Engineering*, Piscataway, NJ: IEEE Press, 1989, ch. Microprogramming: a tutorial and survey of recent developments, pp. 2-20.

[12] M. M. Mano, "Digital logic and computer design," *Eastern Economy ed.* New Delhi, India: Prentice-Hall, 1996.

[13] A. Klaiber, "The technology behind Crusoe processors," *Transmeta Corp.*, Tech. Rep., 2000.

[14] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address reallife challenges," in *Proc. Int. Symp. Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pp. 15-24, 2003.

[15] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412-421, 1974.

Charles Mutigwe was born in Zimbabwe in 1970. He obtained a bachelor's degree in electrical engineering from the University of Zimbabwe, Harare, Zimbabwe in 1994; a master's degree in electrical engineering from Western New England University, Springfield, Massachusetts, U.S.A. in 2003; and an M.B.A. from Norwich University, Northfield, Vermont, U.S.A. in 2005. He is a doctoral candidate in electrical engineering at the Central University of Technology, Bloemfontein, South Africa.

He is currently an IT & Data Analyst at the University of Massachusetts (UMass) Amherst. He is also an adjunct professor at the Isenberg School of Management at UMass Amherst, where he teaches Information Management in the online MBA program. He has worked as an IT professional for over 15 years and his previous positions include: Systems Engineer, Systems Administrator, Systems Developer, Lab Manager and IT Director at dot-com start-ups, a university and Fortune500 companies. His research interests are: electronic design automation, reconfigurable computing and RFID systems. Mr. Mutigwe is a member of the IEEE, the IEEE Computer Society and the ACM.

Johnson Kinyua was born in Kenya in 1958. He obtained a bachelor's degree in electrical engineering from University College London (UCL), London in United Kingdom (U.K.) in 1981; a master's degree in digital communications from the University of Kent, Canterbury in U.K. in 1984; and a PhD in computer science from the University of Cambridge, Cambridge in U.K. in 1992. He is currently the Dean and Professor in the School of Computer Information Systems at Virginia International University, Fairfax, VA, USA. He has publishes widely in international journals and conferences. His previous positions include: Research & Innovation Professor, Associate Professor and Director, Senior Lecturer and Lecturer at different universities. His current and previous research interests are: Software Engineering, Cybersecurity, Database Security, Security Engineering, Distributed Systems, multi-agent systems, Fixed and Wireless networks, and Computer Architecture.

Prof. Kinyua is a member of the IEEE, the IEEE Computer Society and the ACM. Prof. Kinyua has acted as an external examiner for several universities, has been a panel member for external program assessment of programs at two universities and was a committee member of the higher education qualifications accreditation committee in South Africa for a number of years. Prof. Kinyua is member of the international technical program committees (TPC) for two international conferences: ITNG networking track (see http://www.symbolicscience.com/ITNG2012.pdf), held annually in Las Vegas; and IASTED African Conference on Modeling and Simulation (http://www.iasted.org/conferences/ipc-685.html).

Farhad Aghdasi was obtained a bachelor’s degree with honors in electronic \& electrical engineering from the University of Manchester, in the U.K.; a master’s degree in electrical \& computer systems engineering from Oregon State University, Corvallis, Oregon, U.S.A.; an M.B.A. degree from the University of Portland, Oregon, U.S.A.; and a PhD in Electrical Engineering from the University of Bristol, in the U.K. He is currently the Dean of the Faculty of Science and Agriculture at the University of Fort Hare in South Africa. Over the past 30 years he has held academic posts in universities in Southern Africa including Professor and Director: School of Electrical and Computer Systems Engineering and Dean of the Faculty of Engineering and Information Technology, Central University of Technology, Bloemfontein, South Africa; Vice-Rector: Academic Affairs and Research, Polytechnic of Namibia, Windhoek, Namibia.

Prof. Aghdasi is currently the NRF grant holder for the Risk and Vulnerability Assessment Centre (RAVAC) for food and water security in the Eastern Cape region of South Africa. Prof. Aghdasi’s passion is to increasingly use the postgraduate research projects and postdoctoral activities for innovations in the betterment of the lives of the community, job creation, collaboration with the industry and adding quality to teaching and learning of undergraduates.