Synthesis of Processor Instruction Sets from High-level ISA Specifications

Andrey Mokhov†, Alexei Iliasov‡, Danil Sokolov†, Maxim Rykunov†, Alex Yakovlev†, Alexander Romanovsky‡
†School of Electrical and Electronic Engineering, Newcastle University, UK
‡School of Computing Science, Newcastle University, UK

***Abstract*—As processors continue to get exponentially cheaper for end users following Moore's law, the costs involved in their design keep growing, also at an exponential rate. The reason is ever increasing complexity of processors, which modern EDA tools struggle to keep up with. This paper focuses on the design of Instruction Set Architecture (ISA), a significant part of the whole processor design flow. Optimal design of an instruction set for a particular combination of available hardware resources and software requirements is crucial for building processors with high performance and energy efficiency, and is a challenging task involving a lot of heuristics and high-level design decisions.**

This paper presents a new compositional approach to formal specification and synthesis of ISAs. The approach is based on a new formalism, called Conditional Partial Order Graphs, capable of capturing common behavioural patterns shared by processor instructions, and therefore providing a very compact and efficient way to represent and manipulate ISAs. The Event-B modelling framework is used as a formal specification and verification back-end to guarantee correctness of ISA specifications.

We demonstrate benefits of the presented methodology on several examples, including Intel 8051 microcontroller.

Index terms: microprocessor, instruction set, synthesis.

I. Introduction

Modern microprocessors become increasingly diversified in terms of power modes, heterogeneous hardware platforms, and requirements for legacy software reuse. This is amplified by the ever growing demand for high performance and low power consumption. As a result, under the pressure of time-to-market constraints, a computer architect faces a *design productivity gap* [4]: the capacity of modern CAD tools is insufficient for exploring the variety of possible architectural solutions.

An important part of a microprocessor design is determining the optimal Instruction Set Architecture (ISA) for the target application domain. This is a very computationally intensive task whose search space grows exponentially with the number of instructions and supported operating modes. Furthermore, the ISA development process often goes beyond a one-time effort of a single designer as the ISA may need to be extended at the customer side, e.g., as in Application Specific Instruction set Processors (ASIPs) [37]. ASIPs allow adding new functionality to an extensible baseline ISA in the form of Instruction Set Extensions (ISEs), thereby combining flexibility of a general purpose CPU and performance of an ASIC. The key idea is to analyse the application domain and identify repetitive source code fragments that can be replaced by custom ISE instructions to reduce overheads associated with the instruction fetch cycle and storage of temporary values [22], as well as to enable additional optimisation opportunities in resource allocation, register binding, and port assignment [15][34].

Modern embedded systems often require yet another dimension of ISA flexibility – dynamic reconfigurability. For example, a baseband processor whose core functionality is signal processing may need to be reconfigured upon standardisation of a new communication protocol. Reconfigurable ASIPs address this requirement by combining a static general purpose ISA with a reconfigurable fabric to introduce new functionality when it becomes needed [11][12]. Reconfigurability and custom instructions also address the issue of energy efficiency (a major concern for the microelectronics industry, particularly in mobile and embedded domains) by power elasticity [38] and by moving computationally intensive algorithms from software to hardware [22][27].

One of the key difficulties in designing instruction sets is the necessity to comprehend and deal with a large number of instructions, whose microcontrol implementation may be altered to suit a particular hardware platform or a particular operating mode. To overcome this instructions and groups of instructions have to be managed in a compositional way: an ISA specification should be composable from specifications of its constituent parts. Furthermore, one should be able to transform and optimise ISA specifications in a fully formal way to guarantee correctness without computationally expensive verification after each incremental modification of an ISA.

A. Instruction Set Architecture (ISA) criteria

There are several criteria which determine the choice of an instruction set and a particular processor microarchitecture.

Functionality. Each instruction is associated with a sequence of atomic *actions* (usually acyclic) to complete the corresponding computational task. Note that while a sequential run of actions is sufficient to achieve the instruction functionality, it is often practical to enable some of the actions concurrently, e.g., in order to speed up the instruction execution and to efficiently utilise the available energy. The distinctive classes of instruction functionality are arithmetic operations, data handling, memory access and flow control.

The amount of computation per instruction is an important characteristic of an ISA, which can be illustrated by comparison of Complex Instruction Set Computer (CISC), Reduced Instruction Set Computer (RISC), and Very Long Instruction Word (VLIW) architectures. The CISC architecture is based on a semantically rich instruction set, which provides operand access in several addressing modes and can execute complex multi-cycle operations without storing the intermediate results [24]. In contrast, the RISC architecture employs a relatively small set of basic instructions to build a complex functionality at the level of software [16]. The microarchitecture complexity is further reduced in the VLIW architecture,

where the scheduling for Instruction Level Parallelism (ILP) is performed statically during the program compilation [19].

Operation modes. The same functionality can be achieved in different ways targeting various optimisation criteria. For example, an arithmetic operation can be executed either in an energy efficient way but slowly, or in a low latency mode at the price of extra energy consumption. Alternatively, for security applications, the operation can be combined with power masking and data scrambling. The choice of available operation modes is usually made at the design time and is limited by the circuit area and the timing constraints. Selection of the operation mode can be encoded in the instruction set at two levels: *coarse-grain*, as a separate class of mode-switching instructions or *fine-grain*, as a part of each instruction code.

For example, in the ARM architecture [21], apart from the standard RISC-like operation mode with a 32-bit instruction set there are several special modes, e.g., Thumb and Jazelle. In the Thumb mode the processor switches to a compact 16-bit encoding of a subset of ARM instructions and makes the instruction operands implicit. This reduces the processor functionality but improves its power efficiency through increased code density, usually at the expense of performance. In the Jazelle mode the instruction set is changed to natively execute Java Bytecode and to support just-in-time compilation [33].

Resources. At least one functional unit must be available for each type of atomic actions comprising the instructions. The conflicting situations, when the same hardware resource is requested by several actions, are resolved through scheduling and may also involve dynamic arbitration. Quantity of each resource type is therefore decided by trading resource idle time against the frequency of potential conflicts to resolve.

Modern CPUs, while often referred to as RISC-like, also exhibit the features of CISC and VLIW architectures. For example, they often have complex multi-clock DSP/multimedia instructions, which is typical for CISC. They also combine the compile-time VLIW scheduling with dynamic arbitration of resources to employ ILP for instruction pipelining, out-of-order and speculative execution. Such a diversity of instruction functionality, combined with various operation modes and resource constraints, makes ISA design extremely challenging.

B. Existing ISA approaches and challenges

There are several well-established approaches for the functional-level description and formal verification of ISA. *Event-B* [39] is a widely adopted language for specifying *first-order logic* systems and doing refinements on these representations. Being combined with the RODIN theorem prover [6], it becomes a powerful platform for proving that a (refined) system satisfies the initial specification, e.g., does not leave a certain set of 'good' states during its operation. *HOL* [20] is a computer-assisted proving environment for constructing verifiably correct mathematical proofs. Although its expressiveness is unrivalled, the generic nature of a tool such as ISABELLE/HOL makes it more suitable for analysing individual instructions with deep mathematical properties; see, for example, verification of IA-64 division algorithm [23].

These formal ISA methods have a history of being used for reasoning about hardware implementations, however they are more targeted to the software-related aspects of processor functionality. No hardware implementation issues are usually taken into consideration apart from those directly visible to the instructions, such as the size of addressable memory, the number and type of available registers, etc. As a result, an ISA designer does not have the full control on how the specified functionality is achieved in hardware, what are the costs of every instruction in terms of energy consumption and computation resources, how to minimise latency of instruction decoding logic, or how to dynamically adapt the processor to the current operating conditions. Modelling such low-level implementation details in Event-B or HOL is costly; a more targeted formalism is needed to interface the representation of knowledge about instructions sets with that of knowledge about their execution.

There is clearly a niche in microprocessor EDA where the following design requirements need to be addressed:

- description of individual instruction functionalities at the microcode level as partial orders of atomic actions;
- efficient representation and manipulation with complete instruction sets (re-encoding, re-targeting, etc.);
- compositional approach to ISA design to facilitate modularity, extensibility and reuse;
- explicit capturing of processor operation modes;
- possibility to express the resource availability constraints.

We propose to address these requirements using Conditional Partial Order Graphs (CPOGs) [32]. This model is particularly convenient for composition and representation of large sets of partial orders in a compact form. It can be equipped with a suite of mathematical tools for the refinement, optimisation, encoding and synthesis of the control hardware which implements the required instruction set, similar in spirit to the approach based on *control automata* [9]. We envisage that the model can be used as a complementary formalism for the existing ISA methodologies providing a formal link between the software and hardware domains. Although general-purpose modelling languages and proving environments, such as Event-B or HOL, may be used to a similar effect, the CPOG model offers a superior mathematical construction permitting automated analysis and synthesis.

Fig. 1 shows the proposed pathway from a high-level specification of an ISA to a low-level microcontroller implementation. Our specification and synthesis flow comprises four distinct levels. At the *architectural level* the ISA is modelled using the Event-B formalism. Given available hardware resources and operating modes we can refine the ISA and descend to the *microarchitectural level*. At the *transformation level* the refined instructions are composed into a single CPOG representation which is then iteratively optimised for a set of design constraints, such as requirements to the instruction opcodes and ILP support. Finally, at the *implementation level* the ISA is synthesised into a set of hardware components, such as instruction decoder and microcontrol logic. The relevant sections are denoted in the flow diagram for convenient navigation through the paper.

This paper presents a significant contribution to the relatively new concept of CPOGs. The previous CPOG-related publications focused on algebraic CPOG properties [29],

optimal encoding of instructions [30], and controller synthesis [31][32], while this work brings all these methods to the area of formal ISA specification and introduces *CPOG transformations* as an efficient way of ISA development.

This work also contributes to the area of ASIP research by providing a methodology to systematically manipulate instruction sets in order to explore the space of possible solutions. Our approach can simplify the design of ASIPs and synthesis of ISEs, as it naturally supports incremental and compositional development of instruction sets. Moreover, we utilise the same formal model throughout the whole design process: specification of individual instructions, combining them into instruction sets, exploring the design space, and synthesis of the control logic [31], which facilitates productivity and persistency of the design flow.

The organisation of the paper is as follows. Section II gives the background on the CPOG model and explains how to use it for specification of processor instruction sets. It is followed by Section III, where we describe ISA composition, several transformations defined on CPOGs, and outline synthesis of a physical microcontroller implementation. In Section IV we demonstrate how to formally reason about correctness of CPOG constructs with respect to the given functional ISA descriptions using the Event-B model. Case study in Section V demonstrates how CPOGs can be used for capturing different hardware configurations and operation modes. The paper is concluded with experiments, Section VI, where we demonstrate applicability of the presented approach to design of Intel 8051 microcontroller.

II. CPOGs and instruction set specification

This section presents the basic definitions behind the Conditional Partial Order Graph model and establishes a formal correspondence between CPOGs and instruction sets.

A. CPOG essentials

A *Conditional Partial Order Graph* [32] (further referred to as *CPOG* or *graph*) is a quintuple $H = (V, E, X, \rho, \phi)$ where:

- V is a set of *vertices* which correspond to events (or atomic actions) in a modelled system.
- $E \subseteq V \times V$ is a set of *arcs* representing dependencies between the events.
- *Operational vector* X is a set of Boolean variables. An *opcode* is an assignment $(x_1, x_2, \ldots, x_{|X|}) \in \{0,1\}^{|X|}$ of these variables. An opcode selects a particular partial order from those contained in the graph.
- $\rho \in \mathcal{F}(X)$ is a *restriction function,* where $\mathcal{F}(X)$ is the set of all Boolean functions over variables in X. ρ defines the *operational domain* of the graph: X can be assigned only those opcodes $(x_1, x_2, \ldots, x_{|X|})$ which satisfy the restriction function, i.e. $\rho(x_1, x_2, \ldots, x_{|X|}) = 1$.
- Function $\phi : (V \cup E) \to \mathcal{F}(X)$ assigns a Boolean *condition* $\phi(z) \in \mathcal{F}(X)$ to every vertex and arc $z \in V \cup E$ in the graph. Let us also define $\phi(z) \stackrel{\text{df}}{=} 0$ for $z \notin V \cup E$ for convenience.

CPOGs are represented graphically by drawing a labelled circle ◯ for every vertex and drawing a labelled arrow ⟶ for every arc. The label of a vertex v consists of the vertex name, a colon and the vertex condition $\phi(v)$, while every arc e is labelled with the corresponding arc condition $\phi(e)$. The restriction function ρ is depicted in a box next to the graph; operational variables X can therefore be observed as parameters of ρ.

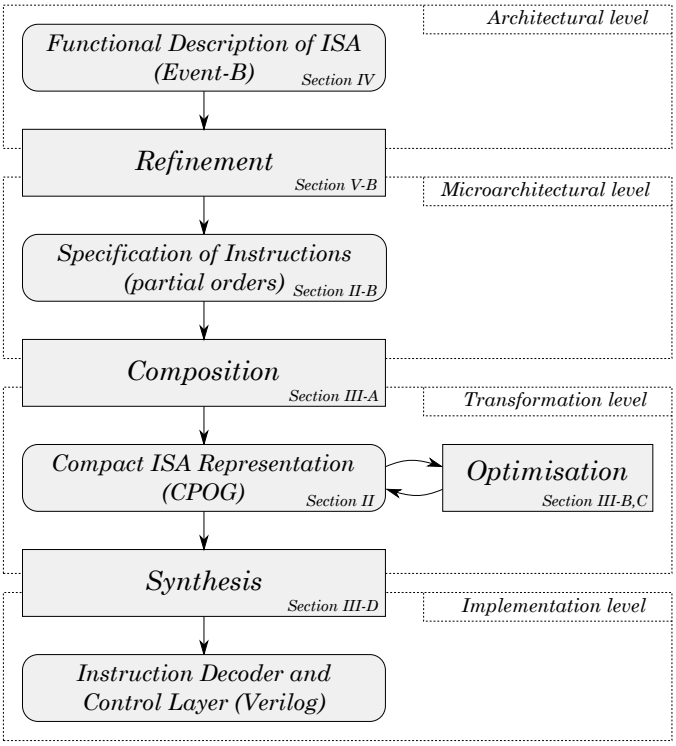
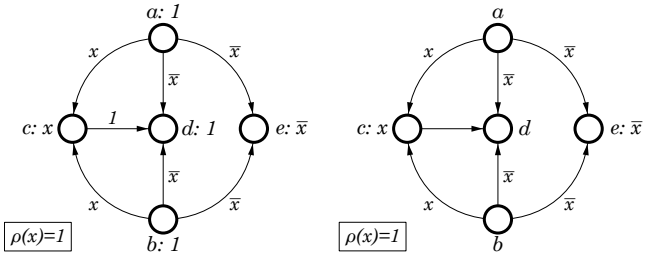


Figure 1: Specification and synthesis flow



(a) Full notation (b) Simplified notation

Figure 2: Graphical representation of CPOGs

Fig. 2(a) shows an example of a CPOG with $|V| = 5$ vertices and $|E| = 7$ arcs. There is a single operational variable x; the restriction function is $\rho(x) = 1$, hence both opcodes $x = 0$ and $x = 1$ are allowed. Vertices $\{a, b, d\}$ have constant $\phi = 1$ conditions and are called *unconditional*, while vertices $\{c, e\}$ are *conditional* and have conditions $\phi(c) = x$ and $\phi(e) = \overline{x}$ respectively. Arcs also fall into two classes: *unconditional* (arc $c \to d$) and *conditional* (all the rest). As CPOGs tend to have many unconditional vertices and arcs we use a simplified notation in which conditions equal to 1 are not depicted in the graph; see Fig. 2(b).

The purpose of conditions ϕ is to 'switch off' some vertices and/or arcs in a CPOG according to a given opcode, thereby producing different *CPOG projections*. An example of a graph and its two projections is presented in Fig. 3.

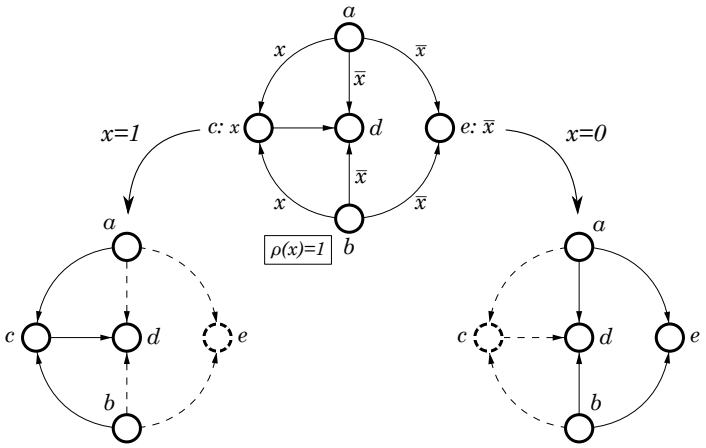


Figure 3: CPOG projections: $H|_{x=1}$ (left) and $H|_{x=0}$ (right)

Instruction	Addition	Exchange
Action sequence	a) Load A b) Load B c) Add B to A d) Save A	a) Load A b) Load B d) Save A e) Save B
Partial order with maximum concurrency	P_{ADD}	P_{XCHG}

Table I: Two instructions specified as partial orders

The leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to 1 after substitution of variable x with 1 (such projections are conventionally denoted by $H|_{x=1}$). Hence, vertex e disappears (shown as a dashed circle ◌), because its condition evaluates to 0: $\phi(e) = \overline{x} = \overline{1} = 0$. Arcs $\{a \to d, a \to e, b \to d, b \to e\}$ disappear for the same reason; they are shown as dashed arrows ⇢. The rightmost projection is obtained in the same way with the only difference that variable x is set to 0; it is denoted by $H|_{x=0}$, respectively. Note that although the condition of arc $c \to d$ evaluates to 1 (in fact it is constant 1) the arc is still excluded from the resultant graph because one of the vertices it connects, viz. vertex c, is excluded and naturally an arc cannot appear in a graph without one of its vertices. Each of the obtained projections can be regarded as specification of a particular behavioural scenario of the modelled system, e.g. as specification of a processor instruction. Potentially, a CPOG $H = (V, E, X, \rho, \phi)$ can specify an exponential number of different instructions (each composed from atomic actions in V) according to one of $2^{|X|}$ different possible opcodes.

B. Specification of instructions

Consider a processing unit that has two registers A and B, and can perform two different instructions: *addition* and *exchange* of two variables stored in memory. The processor contains five datapath components (denoted by $a \ldots e$) that can perform the following atomic actions:

a) Load register A from memory;
b) Load register B from memory;
c) Compute sum $A + B$ and store it in A;
d) Save register A into memory;
e) Save register B into memory.

Table I describes the addition and exchange instructions in terms of usage of these atomic actions.

The addition instruction consists of loading the two operands from memory (actions a and b, causally independent and thus possibly concurrent), their addition (action c), and saving the result (action d). Whether a and b are to be performed concurrently depends on: i) the system architecture, e.g., if concurrent read memory access is allowed, ii) static and dynamic resources availability (the processor hardware configuration must physically contain two memory access components and they both have to be immediately available for use), and iii) the current operation mode which determines the scheduling strategy, e.g. 'execute a and b concurrently to minimise latency', or 'execute a and b in sequence to reduce peak power'. Let us assume for simplicity that in this example all causally independent actions are always performed concurrently, see the corresponding partial order P_{ADD} in Table I[1]. Section V will address joint specification of different scheduling strategies of an instruction.

The operation of exchange consists of loading the operands (concurrent actions a and b), and saving them into swapped memory locations (concurrent actions d and e), as captured by P_{XCHG}. Note that in order to start saving one of the registers it is necessary to wait until both of them have been loaded to avoid overwriting one of the values.

One can see that the two partial orders in Table I appear to be the two projections shown in Fig. 3, thus the corresponding graph can be considered as a joint specification of both instructions. Two important characteristics of such a specification are that the common events $\{a, b, d\}$ are overlaid and the choice between the two operations is distributed in the Boolean expressions associated with the vertices and arcs of the graph. As a result, in our model there is no need for 'nodal point' of choice, which tend to appear in alternative specification models (a Petri Net [18] would have an explicit choice place, a Finite State Machine [28] – an explicit choice state, and a specification written in a Hardware Description Language [28] would describe the two instructions by two separate branches of a conditional statement `if` or `case`).

One downside of a purely graph-based approach to instruction sets is the inability to reason about functional correctness; specifically, the relationship between an instruction behaviour and the functionality of blocks it is made of. Clearly, a designer would seek some form of assurance that an instruction is correct in respect to original requirements and an evidence of correctness is exhibited. An ultimate form of evidence is a formal proof. In Section IV we will show how to obtain the proof of an instruction correctness with a refinement-based derivation of instruction logic.

[1]In this paper we describe partial orders using *Hasse diagrams* [10], i.e. without depicting transitive dependencies, such as, for example, dependencies $a \to d$ and $b \to d$ in partial order P_{ADD}.

C. From instructions to instruction sets

The following notions are introduced to formally define specification and composition of instruction sets.

An *instruction* is a pair $\mathsf{I} = (\psi, P)$, where $\psi \in \{0,1\}^{|X|}$ is a vector assigning a Boolean value to each variable in X, and $P = (V, \prec)$ is a partial order defined on a set of atomic actions V. Semantically, ψ represents the instruction opcode[2], while the precedence relation $\prec$ of the partial order captures behaviour of the instruction[3]. We assume that V and X belong to the corresponding universes shared by all the instructions of the processor: $V \subseteq U_V$ and $X \subseteq U_X$.

An *instruction set* (denoted by IS) is a set of instructions with unique opcodes, i.e. for any $\mathsf{IS} = \{\mathsf{I}_1, \mathsf{I}_2, \ldots, \mathsf{I}_n\}$, such that $\mathsf{I}_k = (\psi_k, P_k)$, all opcodes ψ_k must be different.

Given a CPOG $H = (V, E, X, \rho, \phi)$ there is a natural correspondence between its projections and instructions: an opcode $\psi = (x_1, x_2, \ldots, x_{|X|})$ induces a partial order $H|_\psi$, and paired together they form an instruction $\mathsf{I}_\psi = (\psi, H|_\psi)$ according to the above definition. This leads to the following formal link between CPOGs and instruction sets.

A CPOG $H = (V, E, X, \rho, \phi)$ is a *specification* of an instruction set $\mathsf{IS}(H)$ defined as a union of instructions $(\psi, H|_\psi)$ which are allowed by the restriction function ρ:

$$\mathsf{IS}(H) \stackrel{\text{df}}{=} \{(\psi, H|_\psi),\ \rho(\psi) = 1\}. \tag{1}$$

Using this definition we can formally state that the graph in Fig. 3 specifies the instruction set from Table I. Section III shows how to obtain and efficiently manipulate such CPOG specifications.

III. Transformations

In this section we describe CPOG transformations which allow to systematically manipulate instruction sets. The transformations facilitate the following stages of the ISA design flow shown in Fig. 1:

- *compositional and modular construction* of instruction sets from smaller subsets and/or individual instructions (Subsection III-A);
- *global ISA modifications*, that is modifications of all the instructions at once, for example, re-encoding, retargeting for a different hardware platform, refinement for hardware synthesis (Subsection III-B);
- *local and incremental ISA modifications*, which usually apply only to a subset of all the instructions and are heavily relied on in various ISA optimisation algorithms (Subsection III-C);
- *hardware synthesis*, i.e., transformation of an instruction set into a microcontroller by mapping a given CPOG into Boolean equations (Subsection III-D).

An important feature of all the discussed transformation procedures is their higher efficiency in comparison to the conventional approaches. In particular, we will demonstrate that the algorithmic complexity of all the procedures does not depend on the number of instructions in a given ISA.

[2]In this section the instruction operands are implicit and the opcode completely defines the instruction. We elaborate on this in Section V.

[3]We incorporate the notion of a *microprogram* [28] (the behaviour of the instruction) into the definition of the instruction.

A. Composition

Compositionality is a key concept in modern system design: a realistic system can only be designed and analysed by breaking it down into smaller pieces. A typical instruction set of a modern processor contains hundreds of base instruction classes and various ISA extensions, and usually is a result of several design iterations. Therefore, it is necessary to be able to compose large instruction sets from smaller ones to enable modularisation, reuse, and incremental development.

A CPOG can be deconstructed by means of projections, as was demonstrated in Fig. 3. The opposite operation, that is constructing a CPOG out of given parts, is called *composition*. This subsection describes how it can be used to build large instruction sets from smaller ones.

Formally, *composition* of two instruction sets IS_1 and IS_2 is simply defined as their union $\mathsf{IS}_1 \cup \mathsf{IS}_2$; it is required that the union does not contain two instructions with the same opcode. Due to the commutativity and associativity properties of set union $\cup$, one can compose more than two instruction sets by performing their pairwise composition in arbitrary order, for instance, $\mathsf{IS}_1 \cup \mathsf{IS}_2 \cup \mathsf{IS}_3 = (\mathsf{IS}_1 \cup \mathsf{IS}_2) \cup \mathsf{IS}_3 = \mathsf{IS}_1 \cup (\mathsf{IS}_2 \cup \mathsf{IS}_3)$.

Note that if instructions in given sets IS_k are represented individually (e.g., by listing them one after another as in conventional methods), then the complexity of the composition operation is linear with respect to the total number of instructions: $\Theta(|\mathsf{IS}|)$, where $\mathsf{IS} = \bigcup_k \mathsf{IS}_k$. This is because we have to iterate over all of them to generate the result. It may be unacceptably slow for those applications which routinely perform various operations on large instruction sets. By using the CPOG model for the compact representation of instruction sets, one can perform most of the operations much faster, as demonstrated below.

Let instruction sets IS_1 and IS_2 be specified with graphs $H_1 = (V_1, E_1, X_1, \rho_1, \phi_1)$ and $H_2 = (V_2, E_2, X_2, \rho_2, \phi_2)$, respectively, as in (1). Then their composition has CPOG specification $H = (V_1 \cup V_2, E_1 \cup E_2, X_1 \cup X_2, \rho_1 + \rho_2, \phi)$, where the vertex/arc conditions ϕ are defined as

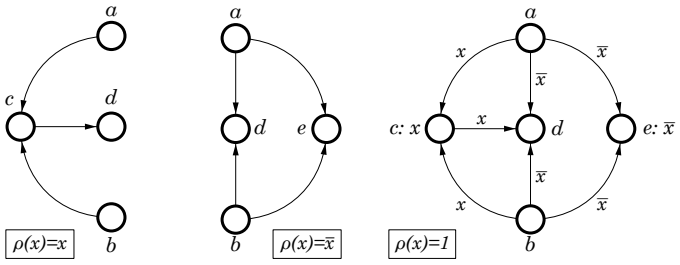
$$\forall z \in V_1 \cup V_2 \cup E_1 \cup E_2,\ \phi(z) \stackrel{\text{df}}{=} \rho_1\phi_1(z) + \rho_2\phi_2(z).$$

We call H the *CPOG composition* of H_1 and H_2 and denote this operation as $H = H_1 \cup H_2$. Note that if $\rho_1 \cdot \rho_2 \neq 0$ then the composition is undefined, because $\mathsf{IS}(H_1)$ and $\mathsf{IS}(H_2)$ contain instructions with the same opcode ψ allowed by both restriction functions: $\rho_1(\psi) = \rho_2(\psi) = 1$. It is possible to formally prove that $\mathsf{IS}(H) = \mathsf{IS}(H_1) \cup \mathsf{IS}(H_2)$ using algebraic methods[4] [32], deriving the following important result:

$$\mathsf{IS}(H_1 \cup H_2) = \mathsf{IS}(H_1) \cup \mathsf{IS}(H_2).$$

Crucially, the complexity of computing a CPOG composition does not depend on the total number of instructions $|\mathsf{IS}_1 \cup \mathsf{IS}_2|$. It depends only on the sizes of graph specifications H_1 and H_2: $\Theta(|V_1|+|E_1|+|V_2|+|E_2|)$. Since the number of arcs $|E_k|$ is at most quadratic with respect to $|V_k|$ and $|V_k| \leq |U_V|$ (all vertices are contained in universe U_V), we have the following upper bound on CPOG composition complexity: $O(|U_V|^2)$.

[4]The proof follows from Theorems 1 and 2 of [32] which concern a more restrictive operation – CPOG addition.



(a) H_{ADD} (b) H_{XCHG} (c) $H_{ADD} \cup H_{XCHG}$

Figure 4: Graph composition

Note that $|U_V|^2$ is potentially much smaller than the number of different instructions[5], which can be exponential with respect to $|V|$, in particular the total number of partial orders on set U_V is greater than $2^{\frac{1}{4}|U_V|^2}$ [10]. To conclude, we can operate on the CPOG representations of instruction sets faster than on the instruction sets themselves.

Let us demonstrate composition of instruction sets on the aforementioned processing unit example. Fig. 4(a,b) shows two graphs H_{ADD} and H_{XCHG} specifying singleton instruction sets $\mathsf{IS}(H_{ADD}) = \{(1, P_{ADD})\}$ and $\mathsf{IS}(H_{XCHG}) = \{(0, P_{XCHG})\}$, respectively. Since their restriction functions are orthogonal $\rho_{ADD} \cdot \rho_{XCHG} = x \cdot \overline{x} = 0$, we can compose them into the graph shown in Fig. 4(c). It specifies the composition $\mathsf{IS}(H_{ADD} \cup H_{XCHG}) = \{(1, P_{ADD}), (0, P_{XCHG})\}$ as intended (see Fig. 3 as a proof).

B. Global transformations

Consider a graph $H = (V, E, X, \rho, \phi)$. Since elements of the quintuple are shared by all instructions in $\mathsf{IS}(H)$, we can make global modifications of the instruction set without iterating over all the instructions. For example, we can add a new action go at the beginning of every instruction by setting $V' = V \cup \{go\}$, $\phi(go) = 1$, and $\phi(go \to v) = 1$ for all $v \in V$. The cost of this global modification is only $\Theta(|V|)$; we call transformations of this type *event insertions*.

It is possible to introduce a global *concurrency reduction* between actions a and b, by setting $E' = E \cup \{a \to b\}$ and $\phi(a \to b) = 1$. As a result, action b will always be scheduled after a in *all* the instructions. The cost of this transformation is $O(1)$, but it is not safe in general: it can introduce deadlocks if action a is scheduled to happen after b in one of the instructions (forming a cyclic dependency). To ensure deadlock freeness verification algorithms from [29] must be employed.

Another basic transformation with the global effect is *variable substitution*. For instance, by replacing every occurrence of x with $\overline{x}$ in all conditions ϕ and function ρ, we flip the corresponding bit in all instruction opcodes. To perform this operation we need to change $\Theta(|V|^2)$ Boolean functions. Variable substitution is a powerful transformation, it can affect not only a single bit, but all the opcodes; care must be taken to ensure that the resultant opcodes do not clash.

The above transformations are *global*. It is possible, however, to apply them only to a subset of selected instructions using the operations of *set extraction* and *decomposition* defined below.

C. Local transformations

Instead of looking at the whole instruction set of a processor one may need to focus attention on its smaller part. As an example, consider the *MMIX processor* instruction set [26] containing 256 different opcodes. 16 of them, starting with bits 0010, are dedicated to addition/subtraction operations, and a designer wants to manipulate them separately from the others.

Let graph $H = (V, E, X, \rho, \phi)$ specify the whole instruction set $\mathsf{IS}(H)$ of the processor and 8-bit opcodes be encoded with variables $\{x_1, \ldots, x_8\}$. Function $f = \overline{x_1} \cdot \overline{x_2} \cdot x_3 \cdot \overline{x_4}$ enumerates all Boolean vectors starting with 0010 and its conjunction with ρ enumerates all wanted opcodes. Thus, graph $H' = (V, E, X, f \cdot \rho, \phi)$ specifies the required part of $\mathsf{IS}(H)$. There is a dedicated operation in the CPOG algebra, called *scalar multiplication*, specifically intended for this task: $H' = f \cdot H$ [32]. Its main feature is that

$$\forall f,\ \mathsf{IS}(f \cdot H) \subseteq \mathsf{IS}(H)$$

In our context, f can be considered an *instruction property* and operation $f \cdot H$ can be called a *set extraction*: it extracts a subset of a given instruction set according to a required property.

A generalisation of this operation is called *decomposition*. It is easy to see that $H_1 = f \cdot H$ and $H_0 = \overline{f} \cdot H$ together contain all instructions from $\mathsf{IS}(H)$: the instructions with opcodes satisfying property f are put into H_1, and all the rest are put into H_0. Thus, any instruction set can be decomposed into two disjoint sets according to a given property. This is formally captured by the following statement:

$$\forall f,\ \mathsf{IS}(H) = \mathsf{IS}(f \cdot H) \cup \mathsf{IS}(\overline{f} \cdot H)$$

Set extraction and decomposition are very cheap operations: they only require computation of a conjunction of two Boolean functions f and ρ.

Returning back to the MMIX example, we can decompose $\mathsf{IS}(H)$ into two disjoint sets: addition/subtraction operations $\mathsf{IS}_1 = \mathsf{IS}(f \cdot H)$, and all the rest $\mathsf{IS}_0 = \mathsf{IS}(\overline{f} \cdot H)$. Then we can apply a transformation, e.g., an event insertion, to IS_1 resulting in IS_1^t. Finally, we can compute composition $\mathsf{IS}^t = \mathsf{IS}_1^t \cup \mathsf{IS}_0$ which contains all the instructions from the original instruction set $\mathsf{IS}(H)$, but with a *local* transformation applied only to addition/subtraction operations.

D. Mapping to logic gates

Finally, the refined CPOG can be mapped into Boolean equations. The mapping procedure is a purely structural operation, i.e., it merely transforms conditions of a given CPOG into logic gates of the corresponding microcontroller without iterating over the instruction set itself.

[5]Although this statement does not hold for our simplistic examples, e.g., $|V| + |E| = 5 + 7 = 12$ and $|\mathsf{IS}| = 2$ in Fig. 4, it does hold in practice. For example, our implementation of Intel 8051 microprocessor (see Section VI) has 244 instructions but its CPOG representation contains only 17 vertices and 47 arcs. Also, if we do not use abstraction and treat instructions *ADD A,B* and *ADD C,D* as different ones, the number of instructions of a modern 32-bit processor can easily grow to 2^{32} while its CPOG will remain compact.

Consider a functional unit represented by vertex $v \in V$ in a given graph $H = (V, E, X, \rho, \phi)$. It is *enabled* to be executed if the following two conditions are met:

- vertex v belongs to the partial order of the currently active instruction, i.e., its condition is satisfied by the current opcode: $\phi(v) = 1$;
- all the functional units corresponding to its predecessors in the graph have already been executed.

This is captured in terms of Boolean equations as follows:

$$enabled(v) = \phi(v) \cdot \prod_{u \in V} \big(\phi(u) \cdot \phi(u \to v) \Rightarrow executed(u)\big),$$

where $a \Rightarrow b$ stands for *Boolean implication* indicating 'b if a' relation, while predicates $enabled(v)$ and $executed(v)$ refer to the physical signals responsible for activation and completion detection of the functional unit corresponding to vertex v; see details of the microcontroller realisation in [31]. The obtained equations should undergo the conventional logic minimisation [28][32] and technology mapping [18] procedures.

It is interesting to note that size of the microcontroller does not depend on the number of instructions directly. There are $\Theta(|V|^2)$ conditions in all the resultant equations; the average size of these conditions is difficult to estimate, but in practice we found that the overall size of the microcontroller never grows beyond $\Theta(|V|^2)$.

IV. Functional correctness

In this section we discuss a formalism called Event-B [7] and its application to formal verification of correctness of CPOG-based representations of instructions. Event-B belongs to a family of state-based modelling languages that represent a design as a combination of state (a vector of variables) and state transformations (computations updating variables). In general, a design in Event-B is abstract: it relies on data types and state transformations that are not directly realisable. This permits terse models abstracting away from insignificant details and enables one to capture various phenomena of a system with a varying degree of detail. Crucially, each statement about the effect of a certain computation is supported by a formal proof. In Event-B, one is able to make statements about safety (this incorporates the property of functional correctness) and progress. Safety properties ensure that a system never arrives at a state that is deemed unsafe (i.e., a shaft door is never open when a lift cab is on a different floor). Progress properties ensure that a system is able to achieve its operational goals.

A. General Event-B methodology

An Event-B development starts with the creation of an abstract specification. A cornerstone of the Event-B method is the stepwise development that facilitates a gradual design of a system implementation through a number of correctness-preserving *refinement* steps. The general form of an Event-B model (or *machine*) is shown in Fig. 5. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called *events*) are defined by a list of new local variables (parameters) vl, a state predicate g called *event guard*, and a next-state relation S called *substitution* (see the **EVENTS** section in Fig. 5).

MACHINE M
  SEES Context
  VARIABLES v
  INVARIANT $I(c, s, v)$
  INITIALISATION $R(c, s, v')$
  EVENTS
    E_1 = **any** vl **where**
      $g(c, s, vl, v)$
    then
      $S(c, s, vl, v, v')$
    end
    ...
END

Figure 5: Event-B model structure

The **INVARIANT** clause contains the properties of the system (expressed as state predicates) that should be preserved during system execution. These define *safe states* of a system. In order for a model to be consistent, invariant preservation should be formally demonstrated. Data types, constants and relevant axioms are defined in a separate component called *context*.

Model correctness is demonstrated by generating and discharging *proof obligations* – theorems in first order logic. The proof obligations demonstrate model consistency, such as the preservation of the invariant by the events, and refinement links to other Event-B models. A collection of automated theorem provers attempts to discharge proof obligations; typically only 3%-5% of proofs require user intervention.

If a model possesses rich control flow properties (e.g., a computational algorithm) the control flow aspect of a model is defined in a separate view called the flow of a model [25]. The flow aspects introduces further verification obligations to ensure that all specified event ordering are found among event traces of a specification. In this work we apply the flow aspect to obtain structured programs – programs that use concepts like sequential composition, choice and loop.

B. Modelling instructions

Our goal is the verification of an instruction, that is, explaining how it is assembled from smaller blocks and whether such an assembly always delivers right results. Before one may attempt such verification, it is requisite to obtain a formal specification of what an instruction is expected to do. In other words, what is the expected effect of an instruction execution on system memory, registers and flags. Such a specification must capture both the normal and abnormal cases. A normal case is a successful execution of an instruction until the completion; this happens when an instruction is called in a right state and with appropriate parameters. For some instructions, there are side conditions that must be satisfied or an instruction execution is aborted. One may also want to foresee (and, possibly, try to mask) abortive execution attempts due to transient hardware faults.

For a refinement-based approach such as Event-B the conventional way to obtain a specification is to gradually develop it from a high-level abstraction of a computing platform: memory that may be written and read, and a device acting upon

it [14][17]. Several specifications have been developed recently, e.g., for XMOS architecture [40], that employ Event-B to formalise instruction sets of real-life CPUs. A CPU is treated as a black-box so that a specification ends with a characterisation of normal and abnormal instruction behaviours. We take such a specification as our starting point, open the black box and explain how each instruction is realised.

Let us first examine what constitutes an instruction specification. The relevant ingredients are state variables (capturing concepts like memory, stack and registers), invariant and the pre- and postconditions of normal and abnormal instruction cases. Model variables v abstractly characterise memory and CPU state. An invariant $I(v)$ defines a set of safe states $S = \{v \mid I(v)\}$ that includes all the reachable model states; it is guaranteed that no chain of instruction execution could lead to a state outside S. Predicate $R(c, s, v')$ defines the set of vectors of initial variable values.

Let predicate families $P^i_N(v)$ and $Q^i_N(v, v')$ denote pre- and postconditions of normal instruction cases, where v and v' correspond to the current and the next states. Correspondingly, $P^i_A(v)$ and $Q^i_A(v, v')$ define abnormal cases.

For instruction preconditions $P_N(v)$ and $P_A(v)$ it holds that whenever an instruction is invoked and the system is in a safe state the instruction is ready to run:

$$I(v) \Rightarrow \bigvee_i P^i_N(v) \vee \bigvee_i P^i_A(v).$$

At the same time, there must be a definite way to tell which case applies in a current state and there should not exist a state where both normal and abnormal cases may be executed. Formally, the normal and abnormal preconditions of an instruction must partition the set S of safe states:

$$S = \{v \mid P_N(v)\} \oplus \{v \mid P_A(v)\}.$$

A postcondition expresses the set of states that may be reached via an instruction execution (an instruction specification may be non-deterministic) and the relationship to the original state. An instruction must terminate in a safe state; that is, re-establish the invariant condition $I(v)$:

$$\forall i, t \cdot t \in \{N, A\} \wedge I(v) \wedge P^i_t(v) \wedge Q^i_t(v, v') \Rightarrow I(v').$$

The condition may be satisfied by simply choosing a pair of $P^i_t(v)$ and $Q^i_t(v, v')$ such that the left-hand side is always false. To counteract this, it is required that an instruction is always able to deliver some result:

$$I(v) \wedge P^i_t(v) \Rightarrow \exists v' \cdot Q^i_t(v, v').$$

The condition also captures the cases where a contradiction is present only for a subset of states characterised by $I(v) \wedge P^i_t(v)$, e.g., a pair of predicates $(y > 0, y' * y' = y)$ where $y \in \mathbb{N}$ do not define a valid instruction case.

In a general case, an instruction specification is formed of a number of normal and abnormal cases.

instruction name **is**
state v
invariant $I(v)$
behaviour
$P^1_N(v) \rightarrow Q^1_N(v, v')$
$\dots$
$P^k_N(v) \rightarrow Q^k_N(v, v')$
$P^1_A(v) \rightarrow Q^1_A(v, v')$
$\dots$
$P^k_A(v) \rightarrow Q^k_A(v, v')$
end

An instruction implementation explains how each case of an instruction specification is implemented by a deterministic program comprising of primitive functional blocks.

To formally relate an operation specification to an implementation we construct a separate Event-B development for each case of an operation. An abstract machine of such development is based on the following template.

MACHINE op
VARIABLES m, r, f, c
INVARIANT
$I_m(m, r, f)$
$c \in \mathbb{B}$
$c = \text{FALSE} \Rightarrow P(m, r, f)$
$c = \text{TRUE} \Rightarrow Q(m, r, f)$
INITIALISATION
$m, r, f, c : |I_m(m', r', f') \wedge P(m', r', f') \parallel c := \text{FALSE}$
EVENTS
op = **when**
$c = \text{FALSE}$
then
$m, r, f, c \ : | \ Q(m', r', f') \parallel c := \text{TRUE}$
end
END

Here, I_m is the state model of an instruction, c is an auxiliary control variable. The model defines a single step automata. The automata is initialised into state when $c = \text{FALSE}$ and atomically transitions into a terminal state where $c = \text{TRUE}$. The invariant properties $c = \text{FALSE} \Rightarrow P(m, r, f)$ and $c = \text{TRUE} \Rightarrow Q(m, r, f)$ explain the meaning of the automata states in relation to the operation definitions: initially, the state satisfies the operation precondition; upon termination it satisfies the operation postcondition. A single transition, defined in event *op*, takes the automata from a state satisfying the precondition to a state satisfying the postcondition. Thus, the specification is trivially convergent.

We use the standard Event-B refinement to gradually replace event *op* with a convergent, deterministic program. The determinacy of a final specification is established at the syntactic level (only deterministic variable updates are used in event specifications). The preservation of convergence is a part of the refinement method.

There is a small semantic mismatch. While we speak about operations in the terms of preconditions and postconditions, Event-B events are defined in the terms of guards and postconditions. The difference is that a guard may not be weakened during refinement while a precondition may not be strengthened. The solution is to insist that an abstract event guard is always refined in such a way that abstract states characterised by the guard are all accounted for by the guards of concrete events. In other words, the collective precondition of an implementation is not more restrictive than in the abstract

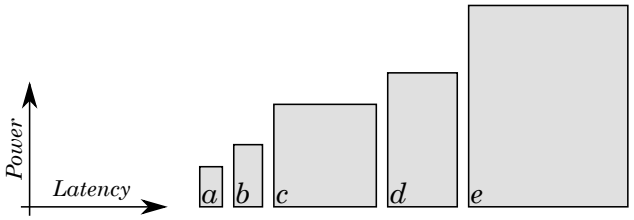


Figure 6: Datapath components for *DP3* implementation

model:

$$I(v) \wedge G(v) \Rightarrow H_1(v) \vee \cdots \vee H_n(v),$$

where G is a guard of some abstract event and H_i are the guards of a subset of concrete events. The condition states that whenever an event is refined, for every state of the event guard there is always something to do in the refined machine.

An illustration to the described modelling approach is provided in Section V.

V. Case study

In this section we study a common low-level GPU instruction, called *DP3*, which given two vectors $\mathbf{x} = (x_1, x_2, x_3)$ and $\mathbf{y} = (y_1, y_2, y_3)$, computes their dot product $\mathbf{x} \cdot \mathbf{y} = x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$. There are many ways to achieve the required functionality in hardware; consider the following datapath components (denoted by $a \ldots e$) which can be used to fulfil this task:

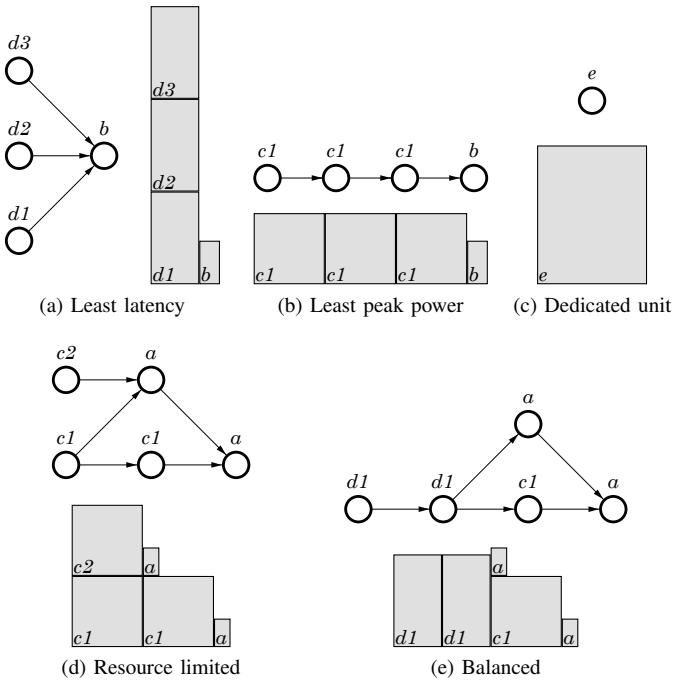
a) 2-input adder;
b) 3-input adder;
c) 2-input multiplier;
d) fast 2-input multiplier;
e) dedicated DP3 unit.

Similar to the *Energy Token model* [35], we associate two attributes, *execution latency* and *power consumption,* with every component. Fig. 6 depicts them as labelled boxes with dimensions corresponding to their attributes; the area of a box represents *energy* required for the computation.

Depending on the current operation mode and availability of the components, a processor has to schedule their activation in the appropriate partial order. Fig. 7 lists several possible partial orders together with their power/latency profiles.

Least latency implementation: the fastest way to implement the instruction is to compute multiplications $tmp_k = x_k \cdot y_k$ concurrently using three fast multipliers $d1$-$d3$ and then compute the final result $tmp_1 + tmp_2 + tmp_3$ with a 3-input adder b; see Fig. 7(a). This implementation is the costliest in terms of peak power and thus may not always be affordable.

Least peak power implementation: a directly opposite scheduling strategy is shown in Fig. 7(b). Three multiplications are performed sequentially on the same slow multiplier $c1$, followed by 3-input addition b. This strategy has the largest latency among all the presented because it is completely sequential and uses slow power-saving components. On a positive side, this implementation requires only two basic functional blocks, which are likely to be reused by other instructions, so its *resource utilisation* is high.



(a) Least latency (b) Least peak power (c) Dedicated unit

(d) Resource limited (e) Balanced

Figure 7: Different implementations of *DP3* instruction

Use of a dedicated component: it is possible that the chosen hardware platform contains a dedicated computation unit capable of computing dot product of two vectors, e.g. *Altera Cyclone III* FPGA board allows building a functional block called *ALTMULT_ADD*(3) with three multipliers connected to a 3-input adder. We can directly execute this block without any scheduling – see Fig. 7(c). While being convenient and potentially very efficient due to custom design, such solution is not always justified because of low resource utilisation: it is impossible to reuse the built-in multipliers for implementing other instructions and if *DP3* is rarely used by software then this dedicated component will be wasting area and power (due to the leakage current) most of the time. Moreover, such implementation does not allow any dynamic reconfiguration thereby being less flexible.

Fast implementation with limited resources: if there are only two available multipliers $c1$ and $c2$ (either because of hardware limitations or because other multipliers are busy at the moment) then the fastest possible scheduling strategy is as follows. At first, two multiplications should be performed in parallel. Then their results are fed to 2-input adder a, while $c1$ is restarted for computing the third multiplication. Finally, the obtained results are added together by the same adder a as shown in Fig. 7(d).

Balanced solution: Fig. 7(e) presents a balanced strategy, which aims to spread power consumption evenly over time, while being relatively fast. This schedule may be advantageous for the best *energy utilisation* and in security applications.

A. Derivation of the instruction set

We could devise more implementations of this instruction, but this is not the point of the case study. The goal is to

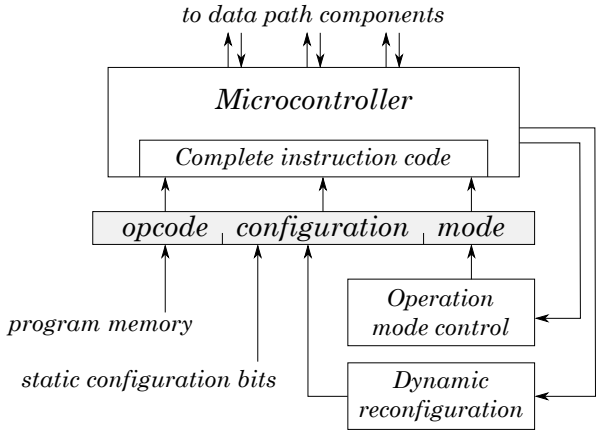


Figure 8: Complete instruction code

demonstrate that even such a basic instruction as *DP3* has a lot of valid scheduling strategies with distinct characteristics. Importantly, it is not possible to select the best strategy because a priori it is not known which one is better. Therefore including only one of them into a processor instruction set is a serious compromise which should not be done at this early and abstract stage of the design process. We propose to include as many different implementations into the instruction set as possible, and, if needed, reduce the behavioural spectrum at the later design stages when more information is at hand (some final decisions can even be made during runtime by dynamic processor reconfiguration). The CPOG model is well suited for this task: it can represent a multitude of different implementations of the same instruction in a compact overlaid form. If the instruction is intended to have only one opcode, we can distinguish between its different implementations using *mode* and *configuration variables*. They are not part of the opcode (which is fetched from the program memory during software execution), but can be dynamically changed by the power/latency runtime control mechanisms [38] or be statically set to constants according to the limitations of the actual hardware platform, as shown in Fig. 8.

We can specify all the discussed implementations of *DP3* instruction using a single CPOG. To do that we first have to encode all of them. If there are no requirements on the mode/configuration codes, then a designer is free to assign them arbitrarily, however it may affect CPOG complexity and, as a consequence, complexity of the resultant microcontroller. In this case it is possible to resort to the help of automated[6] optimal encoding methods [30], which generate codes $\psi_1 = 001$, $\psi_2 = 011$, $\psi_3 = 000$, $\psi_4 = 111$, and $\psi_5 = 101$ for the five partial orders depicted in Fig. 7 (note that these optimal codes are far from trivial sequence of binary codes 000-100). If we compose all of them into a single CPOG using the method from Subsection III-A, we obtain the graph shown in Fig. 9(a). The mode/configuration variables are denoted as $X = \{x, y, z\}$, and two intermediate variables $\{p, q\}$ are derived from them to simplify other graph conditions; as a result only seven 2-input gates are required to compute all graph conditions. The obtained graph is a superposition of the given partial orders, i.e. all of them can be visually identified in it – see, for example, Fig. 9(b), which shows the balanced implementation generated by code ψ_5, and compare it with partial order in Fig. 7(e). For a designer this gives a useful higher-level picture which brings out interaction between the components much better than separate partial order diagrams (this is similar to a metro map which represents a set of metro lines in a compact understandable form).

[6]We used WORKCRAFT framework [5] for CPOG modelling and encoding.

B. Verification of correctness

We now demonstrate application of the Event-B modelling and verification approach described in Section IV to the above example. Due to space constraints we limit ourselves to the consideration of the least latency implementation of DP3 instruction, as shown in Fig. 7(a). We show with a formal approach that our chosen implementation does indeed compute the dot product of two vectors. The following is a simple *DP3* instruction specification that defines only one normal case.

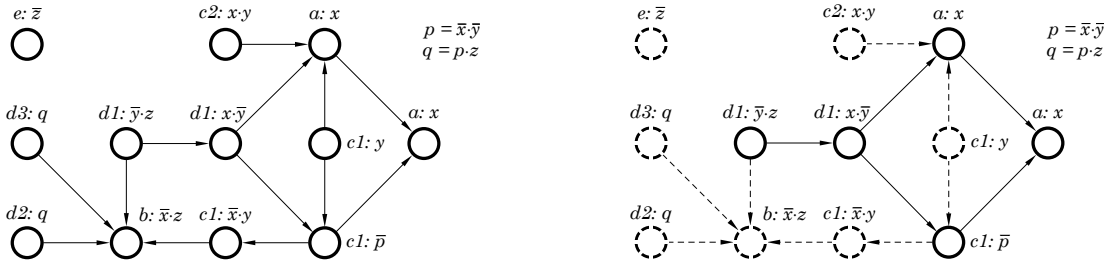
instruction dotp **is**
$c = \text{TRUE} \rightarrow r = x(1) * y(1) + x(2) * y(2) + x(3) * y(3)$
end

As the first step, we obtain an abstract Event-B state model of the instruction by instantiating the model template given above. The properties of the dot product operation are substituted in the place of abstract predicates P and Q. The result is the following Event-B machine. Note that the specification is generalised to an arbitrary vector length. This does not affect proofs and the model may be reused should there be a need for a differing vector length:

MACHINE dotp
VARIABLES x, y, r, c
INVARIANT
$x \in 1..n \rightarrow \mathbb{Z}$
$y \in 1..n \rightarrow \mathbb{Z}$
$r \in \mathbb{Z}$
$c \in \mathbb{B}$
$c = \text{TRUE} \Rightarrow r = \Sigma\{x(i) * y(i) \mid i \in 1..n\}$
INITIALISATION
$x :\in 1..n \rightarrow \mathbb{Z}$
$y :\in 1..n \rightarrow \mathbb{Z}$
$r :\in \mathbb{Z}$
$c := \text{FALSE}$
EVENTS
dotp = **when**
$c = \text{FALSE}$
then
$r := \Sigma\{x(i) * y(i) \mid i \in 1..n\}$
$c := \text{TRUE}$
end
END

The machine is refined into an implementation that makes use of n parallel multipliers and one n-input adder; this is a generalised version of the least latency implementation. The result is the model shown in Fig. 10.

All the consistency and refinement proof obligations are discharged by autonomous theorem provers. Once a concrete model of an instruction is developed and verified it must be, somehow, transformed into a graph to feed it into the CPOG synthesis routines. For this we construct a graph expressing possible event orderings (called the flow aspect of a model). This additional model must be proven consistent with the Event-B machine in a sense that all the paths in such a graph are also possible event sequences in the history of a



(a) Composition H of all implementations

(b) Projection $H|_{\psi_5}$, $\psi_5 = (1, 0, 1)$

Figure 9: CPOG specification of *DP3* instruction

MACHINE least_latency
refines dotp
VARIABLES x, y, r, c, m
INVARIANT
$m \in 1..n \nrightarrow \mathbb{Z}$
$\forall i \cdot i \in \mathrm{dom}(m) \Rightarrow m(i) = x(i) * y(i)$
INITIALISATION $\ldots \parallel m := \emptyset$
VARIANT $1..n \setminus \mathrm{dom}(m)$
EVENTS
mul2 = **any** i **where**
$i \in 1..n$
$i \notin \mathrm{dom}(m)$
then
$m(i) := x(i) * y(i)$
end
addn **ref** dotp = **when**
$c = \mathrm{FALSE}$
$\mathrm{dom}(m) = 1..n$
then
$r := \Sigma(m)$
$c := \mathrm{TRUE}$
end
END

Figure 10: Machine for the least latency implementation

machine execution. The relevant proof obligations are generated automatically by the Event-B modelling tool [6]. The following flow aspect is constructed for a trivial specialisation of least_latency where $n = 3$ with parametrised event mul3 split into three separate events, one for each $i \in \{1, 2, 3\}$; the n-input adder becomes 3-input adder:

The shaded boxes are assertions — elements aiding in the construction of a proof; these do not contribute to the output control graph. Single and double circles are the initialisation and termination actions; the rounded boxes are the events of a machine. The input for CPOG synthesis is a graph obtained by removing assertion elements and dropping all the edge and node annotations. Other implementations of the *DP3* instruction can be verified in a similar way.

VI. Design of Intel 8051 microcontroller

In this section we discuss application of the presented CPOG-based methodology to design of Intel 8051 instruction set [36] and automated synthesis of the corresponding microcontroller. Intel 8051 (also referred to as Intel MCS-51) is a popular CPU introduced back in 1980; although Intel officially discontinued it in 2007 it is still widely available from other vendors in various compatible configurations.

The 8051 ISA is not as complex as modern Intel or ARM ones, however, it is still a serious and practically useful (due to available legacy software) benchmark for the presented methodology. Our implementation supports 244 instructions; to design an ISA of that scale it was essential to make use of the discussed compositional methods in order to split the whole instruction set into manageable subsets, develop them separately, and later merge the intermediate results using the CPOG composition procedure, as discussed in Section III.

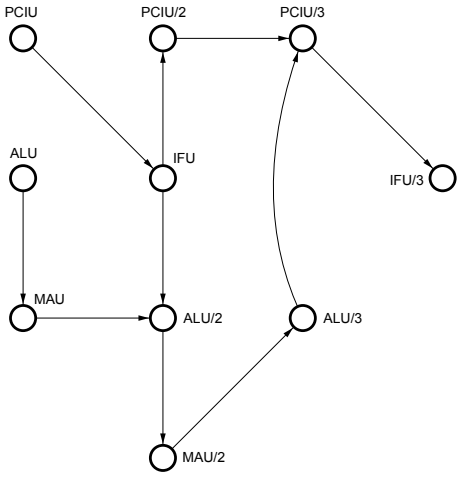
A. Architecture and functional units

Our architecture generally follows the standard Intel 8051 design style [36], which is based on the Harvard architecture with separate data and program memory [8]. There are two on-chip RAMs (a register bank and data storage); the program is stored in a reprogrammable off-chip ROM.

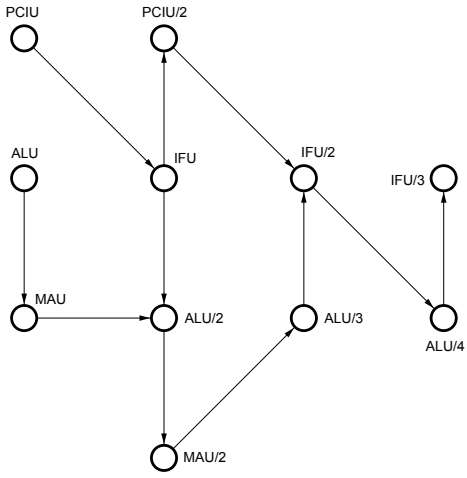
There are five functional units that are considered atomic blocks in our top-level CPOG model of the ISA:

- *Program Counter Increment Unit* (PCIU), as the name suggests, is responsible for incrementing the PC register throughout a program execution.
- *Instruction Fetch Unit* (IFU) fetches instruction opcodes and immediate instruction operands from the program memory using the PC as the address pointer.
- *Arithmetic Logic Unit* (ALU) performs all arithmetic, logic, and other data processing tasks; it contains all the required functional units at the lower hierarchical level hidden from the other top-level components:
  - arithmetic: adders, multipliers, dividers;
  - bitwise operations: Boolean algebra, shifters;
  - data transfer, comparators and the flag register.
- *Memory Access Unit* (MAU) is used to access the register and data memory banks, as well as the stack.
- *Stack pointer Increment Decrement Unit* (SIDU) provides functionality for the stack pointer manipulations.

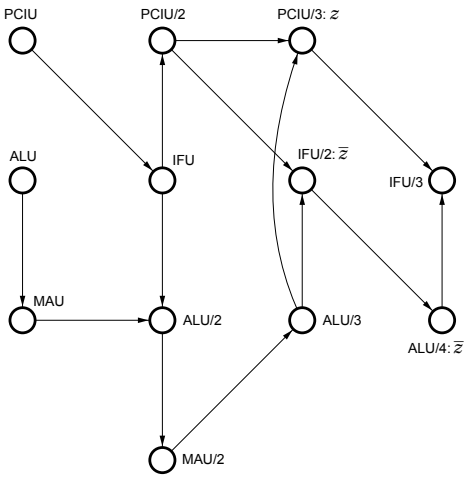
Note that ALU can also be considered a microcontroller: it has its own set of functional units that have to be activated in certain partial orders according the the current instruction opcode. Therefore the whole design is hierarchical: one can abstract from ALU internals and consider it simply as an atomic functional unit whose implementation is an independent design objective and may, for example, be reused. Hierarchical CPOGs are outside the scope of this work, see [29].



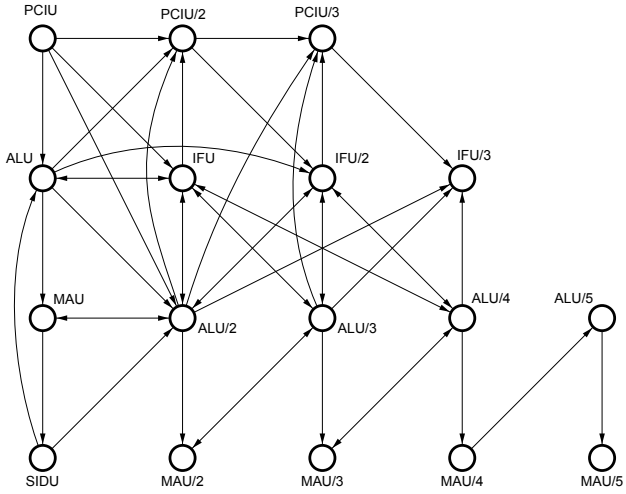
(a) Specification of the first scenario of CJNE ($z = 1$)



(b) Specification of the second scenario of CJNE ($z = 0$)



(c) Complete specification of CJNE instruction



(d) Compete Intel 8051 instruction set (conditions are not shown)

Figure 11: CPOG specifications of CJNE instruction and complete 8051 microcontroller instruction set

The original 8051 architecture was based on an 8-bit ALU, however, we decided to implement a more ambitious 16-bit version to achieve a higher performance and have a unified 16-bit width for both data and address buses. This was a late design decision, but thanks to compositionality of our approach it was possible to quickly identify all the affected instructions, isolate them by extracting the corresponding instruction subset, and perform an appropriate batch modification, as explained in Section III-C.

Another important ISA transformation concerned introduction of two versions for each of the basic computation components (such as adders, multipliers, etc.) with different power/latency characteristics to allow dynamic reconfiguration of the microprocessor for different application-specific requirements in the spirit of Section V.

B. Compositional approach to CJNE instruction

In this subsection we demonstrate CPOG specification of one of the most complicated 8051 instructions, namely, the conditional branch CJNE instruction [36]. Specifically, we consider the following addressing mode:

CJNE @Rn, #immediate, offset

In this mode the CJNE instruction compares contents of the memory location whose address is provided in the specified register with a given immediate constant, and branches to the specified destination (by adding the given address offset to the PC) if their values are not equal. Otherwise, execution continues with the next instruction. CJNE is a good example to demonstrate compositionality of CPOGs: the complete behaviour of the instruction is split into two scenarios, which are easier to specify separately; the scenarios are then composed together resulting in the complete instruction specification.

Fig. 11(a) shows a graph describing the order of activation of functional units in the first CJNE scenario when the branch is not taken because the compared values are equal. This scenario begins with two concurrent sequences of actions: $PCIU \rightarrow IFU$ is executed to fetch the constant

stored immediately after the instruction opcode, while actions $ALU \rightarrow MAU$ are performed to fetch the contents of Rn from the internal memory. After that, another similar sequence is performed, $ALU/2 \rightarrow MAU/2$, to look up the contents of the memory at the address loaded from Rn[7]. Finally, $ALU/3$ is performed to compare the obtained values; the corresponding status flags are set according to the result, in particular, if the values are equal the flag z is set to 1. In this scenario we assume that the values are indeed equal, therefore, the processor may proceed with the next instruction, that is, the program counter is incremented twice (skipping the branch offset) and the next instruction opcode is fetched (actions $PCIU/2 \rightarrow PCIU/3 \rightarrow IFU/3$).

The second scenario, see Fig. 11(b), is identical to the first one until the moment when comparison is performed by $ALU/3$ and it is determined that the compared values are different. At this point, the execution continues as follows. The branch offset is loaded by performing $IFU/2$ straight after $PCIU/2$. Then the actual branch operation is executed by adding the offset to the current PC value ($ALU/4$) and fetching the next instruction opcode. Note that action $PCIU/3$ is skipped in this scenario.

The two scenarios are sufficiently complex even when considered separately; they require a formal proof of correctness using the methodology presented in Section IV. After it is done, one can merge them into one instruction by using the CPOG composition which is a correct by construction operation and does not require any further verification. The result of the composition is shown in Fig. 11(c). One can see that the composition has only three conditional elements, namely, $\phi(PCIU/3) = z$ and $\phi(IFU/2) = \phi(ALU/4) = \overline{z}$. All the other vertices and all the arcs are unconditional due to high similarity between the two scenarios.

C. Complete instruction set and implementation details

Similarly to the CJNE instruction, the rest of the 8051 ISA was formally specified with CPOGs. The instruction opcodes were derived using the optimal encoding methodology presented in [30]. Encoded instructions were then composed together to obtain the complete CPOG specification shown in Fig. 11(d). Note that vertex and arc conditions are not shown on the diagram for clarity; the arcs therefore illustrate interdependencies between different functional units occurring in all 244 supported instructions.

The final CPOG was mapped into logic gates (see Section III-D) generating the enabling signals for all the functional units. We validated the design on an *Altera Cyclone III* family FPGA, and then prepared an ASIC implementation in *STMicroelectronics 130nm* technology; the design passed the 8051 ISA testbenches and is to be fabricated in 2013. To estimate the complexity of the generated control logic, we counted the number of cells used for the top-level control (326) and the internal ALU control (220). It should be noted that in the used technology a cell can correspond to a logic gate with up to 9 inputs. To sum up, the overall area use was only 546 logic gates for the whole microcontroller except the functional units. For comparison, we took three publicly available Intel 8051 implementations, namely [1], [2], and [3], and synthesised their central controllers in the same technology library. The final gate counts were, respectively: 1545, 472 (without the ALU/interrupt control), and 825. The ALU and interrupt control logic from [2] was scattered across datapath modules for optimisation, hence we could not extract it and it was not included in the count of 472. However, we can still conclude that our implementation is very efficient in terms of area. Moreover, unlike [1-3], our 8051 microcontroller supports two different modes of operation (low latency and low power) which can be dynamically chosen at runtime [31].

[7]We use $/k$ notation to distinguish between different executions of the same functional unit in the course of an instruction.

VII. Conclusions

In this paper we demonstrated that the Conditional Partial Order Graph model is a very convenient and powerful formalism for specification of processor instruction sets. It is possible to efficiently describe many different 'microcode' implementations of the same instruction as a single mathematical structure and perform its refinement, optimisation, and encoding using formal CPOG transformations. Crucially, these transformations operate on a CPOG specification rather than on the instruction set itself and thus their complexity does not depend on the number of different instructions.

The overall number of CPU instructions is often quite large although the majority of them are of a fairly trivial nature. To free a designer from the tedium of attending to the minute details of instruction logic we plan to implement a procedure to automatically construct a collection of correct instruction specifications. A number of such procedures were studied within the constructive logic where the proof of a specification statement is given in terms that permit an automatic extraction of an executable program. Although the search space for a proof is potentially very large, the application of proof planning techniques, such as rippling and abstraction, reduce it considerably to make possible the discovery of non-trivial programs with loops and branching [13].

Another direction of the future work includes development of a software toolkit for integration of the presented methodology into the standard processor design flow.

Acknowledgement

We would like to thank the reviewers for their helpful comments and suggestions. This work was supported by EPSRC grants EP/I038357/1 (eFuturesXD, project PowerProp) and EP/J008133/1 (TrAmS-2).

References

[1] Dalton project 8051 controller. http://www.cs.ucr.edu/~dalton/i8051.
[2] Opencores.org 8051 core project. http://opencores.org/project,8051.
[3] Oregano Systems 8051 core. http://www.oreganosystems.at/?page_id=172.
[4] International Technology Roadmap for Semiconductors: Design, 2009.
[5] The Workcraft framework homepage. http://www.workcraft.org, 2009.
[6] The RODIN toolset. http://www.rodintools.org, 2012.
[7] J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
[8] H. H. Aiken and G. M. Hopper. The automatic sequence controlled calculator. *Electrical Engineering*, Vol. 65; No. 8-9: pp. 384-391 (Aug 1946); No. 10: pp. 449-454 (Oct 1946); No. 11: pp. 522-528 (Nov 1946).
[9] S. Baranov. *Logic Synthesis for Control Automata*. Kluwer Academic Publishers, 1994.

[10] G. Birkhoff. *Lattice Theory*. Third Edition, American Mathematical Society, Providence, RI, 1967.

[11] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proc. of the 2002 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 262–269. ACM, 2002.

[12] P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proc. of the 41st Design Automation Conference (DAC)*, pages 395–400. ACM, 2004.

[13] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[14] D. Cansell, D. Mery, and C. Proch. System-on-chip design by proof-based refinement. *Int. J. Softw. Tools Tech. Transfer*, 11:217–238, 2009.

[15] N. Clark, H. Zhong, and S. A. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the IEEE/ACM Int'l Symposium on Microarchitecture (MICRO)*, pages 129–140, 2003.

[16] J. Cocke and V. Markstein. The evolution of risc technology at ibm. *IBM J. Res. Dev.*, 44:48–55, January 2000.

[17] J. Colley. *Guarded Atomic Actions and Refinement in a System-on-Chip Development Flow: Bridging the Specification Gap with Event-B*. PhD thesis, University of Southampton, 2010.

[18] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.

[19] J. A. Fisher. Very long instruction word architectures and the eli-512. *SIGARCH Comput. Archit. News*, 11:140–150, June 1983.

[20] A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving (ITP)*, pages 243–258, 2010.

[21] S. Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2000.

[22] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of ineffciency in general-purpose chips. *Commun. ACM*, 54(10), 2011.

[23] J. Harrison. Formal verification of IA-64 division algorithms. In *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), LNCS 1869*, pages 234–251. Springer, 2000.

[24] S. Heath. *Microprocessor architectures RISC, CISC and DSP*. Butterworth-Heinemann Ltd., 2nd edition, 1995.

[25] A. Iliasov. Use case scenarios as verification conditions: Event-B/Flow approach. In *Proceedings of 3rd International Workshop on Software Engineering for Resilient Systems*, September 2011.

[26] D. Knuth. *MMIXware, A RISC Computer for the Third Millennium*, volume 1750 of *Lecture Notes in Computer Science*. Springer, 1999.

[27] J.-E. Lee, K. Choi, and N. Dutt. Energy-efficient instruction set synthesis for application-specific processors. In *Proc. of Int'l Symposium on Low Power Electronics and Design (ISLPED)*, pages 330–333, 2003.

[28] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[29] A. Mokhov. *Conditional Partial Order Graphs*. PhD thesis, Newcastle University, 2009.

[30] A. Mokhov, A. Alekseyev, and A. Yakovlev. Encoding of processor instruction sets with explicit concurrency control. *IET Computers & Digital Techniques*, 5(6):427–439, 2011.

[31] A. Mokhov, D. Sokolov, M. Rykunov, and A. Yakovlev. Formal modelling and transformations of processor instruction sets. In *Int'l Conf. on Formal Methods and Models for Codesign (MEMOCODE)*, pages 51–60, 2011.

[32] A. Mokhov and A. Yakovlev. Conditional Partial Order Graphs: Model, Synthesis and Application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[33] M. O. Myreen. Verified just-in-time compiler on x86. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of Principles of Programming Languages (POPL)*, pages 107–118. ACM, 2010.

[34] N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *Proc. of the 2010 Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 707–712. IEEE Press, 2010.

[35] D. Sokolov and A. Yakovlev. Task scheduling based on energy token model. In *Workshop on Micro Power Management for Macro Systems on Chip (uPM2SoC)*, 2011.

[36] C. Steiner. *The 8051/8052 Microcontroller: Architecture, Assembly Language, And Hardware Interfacing*. Universal Publishers, 2005.

[37] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for asips. In *Proc. of the Int'l Symposium on High-Level Synthesis*, pages 11–16, 1994.

[38] F. Xia, A. Mokhov, Y. Zhou, Y. Chen, I. Mitrani, D. Shang, D. Sokolov, and A. Yakovlev. Towards power-elastic systems through concurrency management. *IET Computers and Digital Techniques*, 6(1):33–42, 2012.

[39] F. Yuan and K. Eder. A Generic Instruction Set Architecture Model in Event-B for Early Design Space Exploration. Technical Report CSTR-09-006, University of Bristol, September 2009.

[40] F. Yuan, S. Wright, K. Eder, and D. May. Managing complexity through abstraction: A refinement-based approach to formalize instruction set architectures. In *13th International Conference on Formal Engineering Methods*, pages 585–600, 2011.

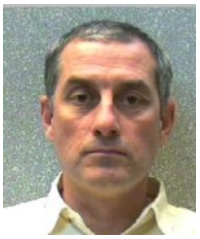
Andrey Mokhov is a Lecturer in the School of Electrical and Electronic Engineering, Newcastle University. In 2000-2005 he studied Computing Science at Kyrgyz-Russian Slavic University, Kyrgyzstan. After graduation he joined the Microelectronic Systems Design group at Newcastle University as a PhD student and in 2009 he successfully defended his PhD dissertation. His research concerns different levels of electronic design automation: from formal models for system specification and verification to logic synthesis and application-specific optimisation.

Alexei Iliasov is a Research Associate at the School of Computing Science of Newcastle University. He got his PhD in Computer Science in 2008 in the area of modelling artefacts reuse in formal developments. His research interests include agent systems, formal methods for software engineering and verification.

Danil Sokolov is a researcher in the School of Electrical and Electronic Engineering, Newcastle University. His first degree is in Computing Science received in 1999 from Kyrgyz-Russian Slavic University. In 2001 he joined the Microelectronic System Design group at Newcastle University as a research student and received his PhD in Electrical and Electronic Engineering in 2006. His research interests are modelling and design automation for power-efficient and heterogeneous systems.

Maxim Rykunov is a PhD student in Newcastle University. He received his BSc and Msc degrees from Saint-Petersburg State Polytechnical University, Russia. His current research interests are development of asynchronous microprocessors and design of power efficient systems.

Alex Yakovlev is a Professor in Computer Systems design; he leads the Microelectronic Systems Design Group at Newcastle. He is an international authority in self-timed and concurrent systems, and low-power computing. He pioneered a model of signal transition graphs, based on Petri nets, which is a de factor standard for asynchronous controllers and interfaces. He currently holds a prestigious EPSRC Dream Fellowship on energy-modulated computing.



Alexander Romanovsky is a Professor in the Centre for Software Reliability. He is a co-investigator of the TrAmS EPSRC/UK platform grant on Trustworthy Ambient Systems (2007-2011) and the Principle investigator of the new EPSRC/RSSB research project Safecap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety (2011-2014).