School of Electrical, Electronic & Computer Engineering

Formal modelling and transformations of processor instruction sets

Andrey Mokhov, Danil Sokolov, Maxim Rykunov, Alex Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2011-169

March 2011

Contact:

Andrey.Mokhov@ncl.ac.uk

Danil.Sokolov@ncl.ac.uk

Maxim.Rykunov@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grants EP/G037809/1 and EP/F016786/1

NCL-EECE-MSD-TR-2011-169

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

`http://async.org.uk/`

Formal modelling and transformations of processor instruction sets

Andrey Mokhov, Danil Sokolov, Maxim Rykunov, Alex Yakovlev

Abstract

Instruction sets of modern processors contain hundreds of instructions defined on a relatively small set of datapath components and distinguished by their codes and the order in which they activate these components. Optimal design of an instruction set for a particular combination of available hardware components and software requirements is crucial for system performance and is a challenging task involving a lot of heuristics and high-level design decisions. The overall design process is significantly complicated by inefficient representation of instructions, which are usually described individually despite the fact that they share a lot of common behavioural patterns.

This paper presents a new methodology for compact graph representation of processor instruction sets, which gives the designer a new high-level perspective for reasoning on large sets of instructions without having to look at each of them individually. This opens the way for various transformation and optimisation procedures, which are formally defined and explained on several examples, as well as practically evaluated on an FPGA platform.

1 Introduction

Modern microprocessors become increasingly diversified in terms of power modes, heterogeneous hardware platforms, requirements for legacy software reuse, etc. This is amplified by the rapidly growing demand for low power consumption, high performance and small area of the produced circuits. As a result, under the pressure of time to market constraints, a computer architect faces a productivity gap: the capacity of modern CAD tools is insufficient for exploring the variety of possible architectural solutions and for identifying the optimal instruction set, which is a large part of a microprocessor design.

There are several criteria which determine the choice of a processor microarchitecture and the generation of an efficient instruction set:

Functionality. Each instruction is associated with a sequence of atomic *actions* (usually acyclic) to complete the 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 the key dilemma of computer architecture – it determines the tradeoff between the complexity of microarchitecture implementation and the software code it executes. Historically there were different views on this dilemma [10]. Initially, the Complex Instruction Set Computer (CISC) architecture with its semantically rich instruction set dominated the microprocessor market. CISC instructions could access their operands in several addressing modes and could execute complex multi-cycle operations without storing the intermediate results, which was advantageous for slow and expensive memory. The major disadvantage of CISC was the complexity of the instruction decoding logic – it had to distinguish among many instructions and their addressing modes. This problem has

been resolved in the Reduced Instruction Set Computer (RISC) architecture, where the simplicity of instruction decoding and pipelining was achieved at the cost of decreased code density. In RISC a relatively small set of basic instructions was employed to build complex functionality at the level of software [4]. The microarchitecture complexity has been further reduced in the Very Long Instruction Word (VLIW) architecture, where the scheduling for Instruction Level Parallelism (ILP) is performed statically during the program compilation [7].

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, Intel 80386 processor [9] can operate in real and protected modes. In real mode the instruction set is backward compatible with the previous generations of 16-bit x86 architecture and can access only 1MB of memory through 20-bit segmented address space. In protected mode the processor switches to 32-bit instruction set, addresses up to 4GB of memory and also features special instructions for multitasking, process level security and memory management (virtualisation and paging).

Similarly, in the ARM architecture [8], 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 performance as less data needs to be fetched from the memory. In the Jazelle mode the instruction set is changed to natively execute Java Bytecode and to support JIT compilation [18].

In Razor-II architecture [6] the conservative variation margins on the clock period are cut and potentially incorrect computation results are handled in error correction mode, where preemption mechanism is employed, similar to that of speculative computations [13].

Resources. At least one computation resource needs to be available for each type of atomic action comprising the instructions. The availability of resources has two aspects: *static* and *dynamic*. The static aspect is addressed at the stage of system synthesis and is mostly constrained by the circuit area and timing requirements. The dynamic aspect arises at the runtime when the same resource is needed for several actions – such a conflict has to be resolved through scheduling which may also involve resource arbitration. It is advantageous to optimise the quantity of each resource type at the synthesis stage targeting a trade-off between resource idle time and the number of conflicts to resolve. This can be achieved by the statistical analysis of potential resource utilisation and careful adjustment of the instruction set.

Usually a designer tries to balance the load on CPU, memory and communication buses at the design time. However, it is often not possible to fine tune the circuit for all execution scenarios at the design time and one of the circuit components becomes a bottleneck limiting the performance of the whole system. In this situation a dynamic reconfiguration of the system brings advantages, e.g. the critical path can be sped up to improve circuit latency and the non-critical paths can be slowed down to save power.

Modern microprocessors, while often referred to as RISC-like, also exhibit the features of CISC and VLIW architectures. For example, they have multi-clock instructions with high-level execution semantics (e.g. if-then-else, DSP and multimedia instructions), which is typical for CISC. They also combine the compile-time scheduling of VLIW architecture with dynamic arbitration of resources to employ ILP for instruction pipelining, out-of-order and speculative execution. Being combined with various operation modes and resource restrictions, such a diversity of instruction functionality presents a real challenge to the efficient design of microprocessors.

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

- compact description of individual instruction functionalities as partial orders of atomic actions;
- efficient representation of complete instruction sets to allow their transformations (optimisation of encoding, re-targeting for different hardware platforms, etc.);
- capturing of processor operation modes as explicit parameters of the instruction sets;
- possibility to express the resource availability constraints;
- encoding of instruction set for different optimisation criteria (code length minimisation, complexity of decoding logic, legacy software compatibility, etc.);

We propose to address these requirements using a graph model, called Conditional Partial Order Graphs (CPOGs) [17]. This model is particularly convenient for composition and representing large sets of partial orders in a compact form. It can also be equipped with a set of mathematical tools for the refinement, optimisation, encoding and synthesis of the control hardware which implement the required instruction set, similar in spirit to the approach based on *control automata* [2].

This paper presents a significant contribution to the relatively new concept of CPOGs. The previous CPOG-related publications, e.g. [15][16][17], focused on algebraic CPOG properties, controller synthesis, verification and optimal encoding of partial orders, while this work brings all these methods to the area of formal specification of processor instruction sets and introduces *CPOG transformations* as an efficient way of instruction set management.

The organisation of the paper is as follows. Section 2 gives the background of the CPOG model and shows how to use it for specification and composition of processor instruction sets. It is followed by Section 3, where we describe several transformations defined on CPOGs and discuss issues of a physical microcontroller implementation. Case study in Section 4 demonstrates how CPOGs can be used for capturing different hardware configurations and operation modes. The paper is concluded with experiments, Section 5, where we specify an instruction set and study its FPGA implementation.

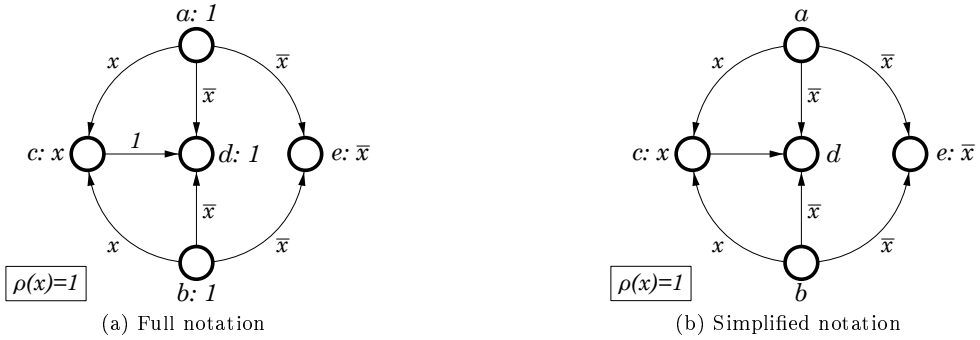
2 Formal model for instruction sets

This section presents the basic definitions behind the CPOG model and demonstrates how it can be applied to the efficient specification of processor instruction sets.

2.1 CPOG essentials

A *Conditional Partial Order Graph* [17] (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.



(a) Full notation (b) Simplified notation

Figure 1: Graphical representation of CPOGs

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, semicolon 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(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 Figure 1(b).

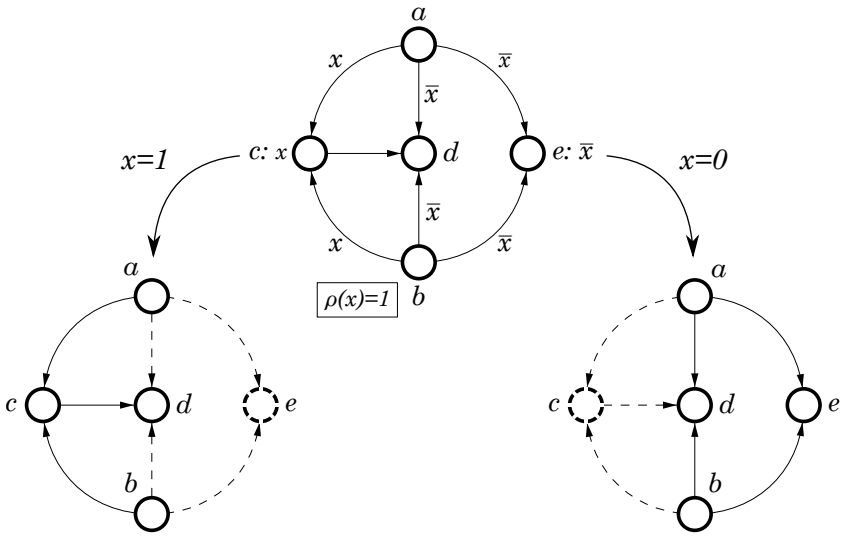


Figure 2: CPOG projections: $H|_{x=1}$ (left) and $H|_{x=0}$ (right)

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 Figure 2. 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.

2.2 Specification and composition 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 $a \dots e$) 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 1 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 lower 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 the table[1]. Section 4 will address joint specification of different scheduling strategies of an instruction.

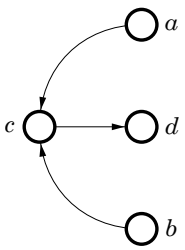
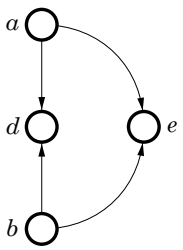
Instruction	Addition	Exchange
Action sequence	a) Load A b) Load B c) Add B to A d) Save A	a) Load A b) Load B d) Save A e) Save B
Partial order with maximum concurrency	 P_{ADD}	 P_{XCHG}

Table 1: Two instructions specified as partial orders

[1] In this paper we describe partial orders using *Hasse diagrams* [3], i.e. without depicting transitive dependencies, such as, for example, dependencies $a \to d$ and $b \to d$ in partial order P_{ADD}.

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 1 appear to be the two projections shown in Figure 2, 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/Signal Transition Graph [5] would have an explicit choice place, a Finite State Machine [14] – an explicit choice state, and a specification written in a Hardware Description Language [14] would describe the two instructions by two separate branches of a conditional statement `if` or `case`).

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:

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

Using this definition we can formally state that the graph in Figure 2 specifies the instruction set from Table 1. In the rest of this section we show how to obtain such CPOG specifications.

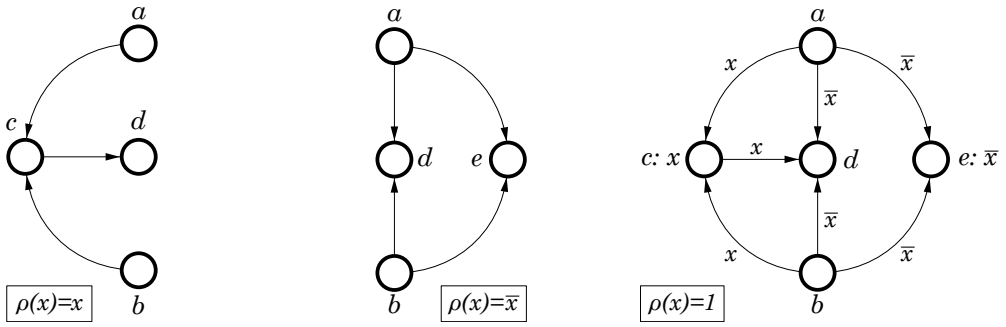
Composition of two instruction sets IS_1 and IS_2 is their union $\mathsf{IS}_1 \cup \mathsf{IS}_2$. Composition is not defined if the union contains two instructions with the same opcode (otherwise, the result would not be an instruction set by the above definition). Due to the commutativity and associativity properties of set union $\cup$ we can compose more than two instruction sets by performing their pairwise composition in arbitrary order.

Note that, if instructions in given sets IS_k are represented individually (as they are 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. Using the CPOG model for the compact representation of instruction sets allows most of the operations to be performed 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. Then their composition has CPOG specification $H = (V_1 \cup V_2, E_1 \cup$

[2] In this section the instruction operands are implicit and the opcode completely defines the instruction. We elaborate on this in Section 4.

[3] We incorporate the notion of a *microprogram* [14] (the behaviour of the instruction) into the definition of the instruction.



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

Figure 3: Graph composition

$E_2, X_1 \cup X_2, \rho_1 + \rho_2, \phi)$, 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] [17], hence we can derive the following important equation:

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

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: $O(|U_V|^2)$. Note that $|U_V|^2$ is potentially much smaller than the number of different instructions, which can be exponential with respect to $|V|$, in particular the total number of partial orders on set U_V is greater than $2^{\frac{1}{4}|U_V|^2}$ [3]. To conclude, we can operate on the CPOG representations of instruction sets faster than on the instruction sets themselves.

Let us demonstrate specification and composition of instruction sets on the aforementioned processing unit example. Figure 3(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 Figure 3(c). It specifies compositional instruction set $\mathsf{IS}(H_{ADD} \cup H_{XCHG}) = \{(1, P_{ADD}), (0, P_{XCHG})\}$ as intended (see Figure 2).

3 Transformations

In this section we describe several CPOG transformations which allow efficient management of instruction sets. We also discuss the issues associated with physical controller implementation and possible signal-level refinements of the model for capturing synchronous and self-timed control interfaces.

3.1 Basic graph transformations

Consider a graph $H = (V, E, X, \rho, \phi)$. Since elements of the quintuple are shared by all instructions from $\mathsf{IS}(H)$, we can make global modifications of the instruction set without iterating over all the instructions.

[4] The proof follows from Theorems 1 and 2 of [17] which concern a more restrictive operation – CPOG addition.

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 [16] 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 to apply them to a subset of selected instructions using the operations of *set extraction* and *decomposition* defined below.

3.2 Set-theoretic operations

Instead of looking at the whole instruction set of a processor we may need to focus our attention on its smaller part. As an example, consider the *MMIX processor* instruction set [12] containing 256 different opcodes. 16 of them, starting with bits 0010, are dedicated to addition/subtraction operations, and we want 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 CPOG algebra, called *scalar multiplication*, intended for this task: $H' = f \cdot H$ [17]. 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 $\mathsf{IS}(H)$: all 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 obtaining IS_1^t. Finally, we can compute composition $\mathsf{IS}^t = \mathsf{IS}_1^t \cup \mathsf{IS}_0$ which contains all instructions from the original instruction set $\mathsf{IS}(H)$, but with a *local* transformation applied to addition/subtraction operations.

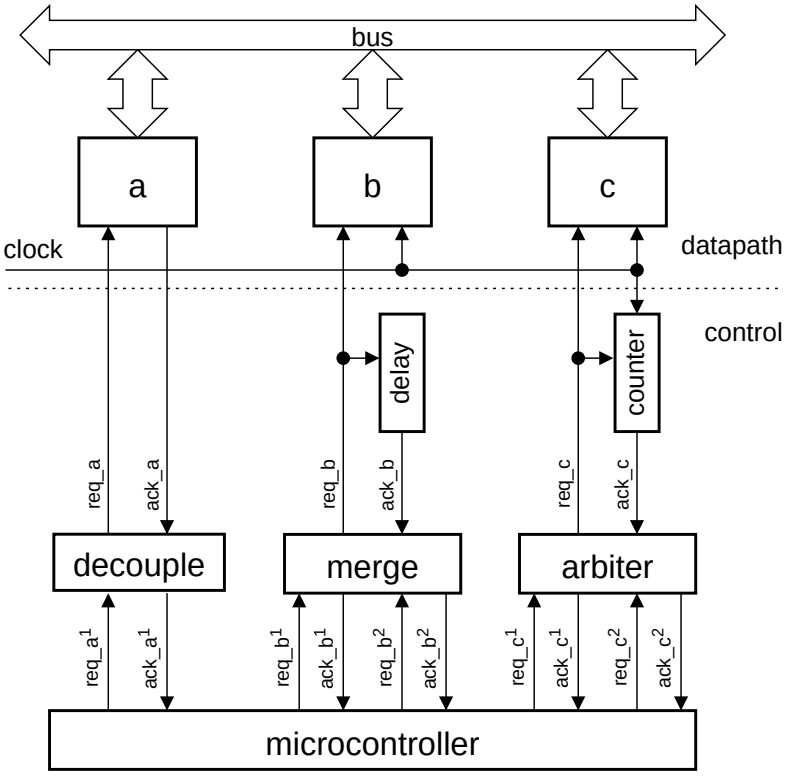


Figure 4: Datapath interface architecture

3.3 Refinements for control synthesis

As soon as all the intended manipulations with the instruction set are performed, we can proceed to the stage of mapping the resultant CPOG into Boolean equations and produce a physical implementation of the specified microcontroller. In order to descend from the abstract level of atomic actions to the physical level of digital circuits the signal-level refinements are necessary.

To interface with an asynchronous datapath component a it is possible to use the standard request-acknowledgement handshake (req_a, ack_a), as shown in Figure 4. In case of a synchronous component b the request signal is used to start the computation but, as there is no completion detection, the acknowledgement signal has to be generated using a matched delay [20]. Also, there are cases when a matched delay has to be replaced with a counter connected to the *clock* signal to provide an accurate multi-cycle delay – see the interface of component c in the same figure. Note that we do not explicitly show *synchronisers* [11] in the diagram; it is assumed that components b and c are equipped with the necessary synchronisation mechanisms to accept asynchronous requests from the microcontroller.

To explicitly specify handshake signals it is possible to perform a graph transformation explained in Figure 5. Every atomic action a^1 is split into a pair of events req_a^1+ and ack_a^1+ standing for rising transitions of the corresponding handshake signals. If there are two occurrences of an atomic action, e.g. b^1 and b^2, then both vertices are split[5], etc. Semantically, when an atomic action a^1 is ready for execution, the controller should issue the request signal req_a^1 to component a; then the high value of the acknowledgement signal ack_a^1 will indicate completion of a.

Notice that the microcontroller does not reset handshakes until all of them are complete. This leads to a potential problem: a component cannot be released until the instruction execution is finished. To deal with the problem it is necessary to *decouple* the microcontroller from the component, see box 'decouple' in Figure 4 and its gate-level implementation in Figure 6(a). Also, when a component b is used twice in an instruction we have to combine two handshakes $(req\_b^{1,2}, ack\_b^{1,2})$ into one using the *merge* controller,

[5] We use superscripts to distinguish different occurrences of the same event.

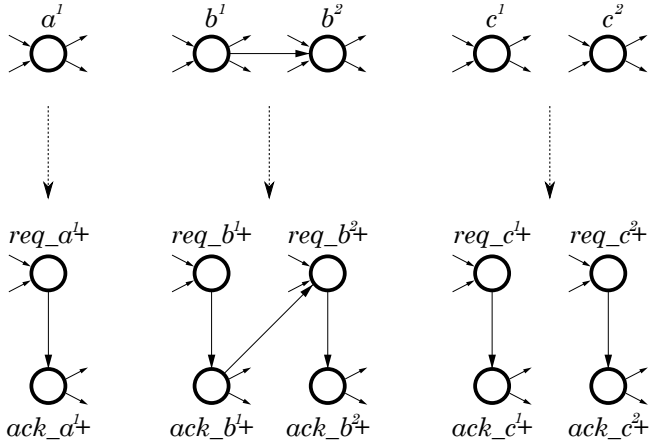


Figure 5: Signal-level refinement

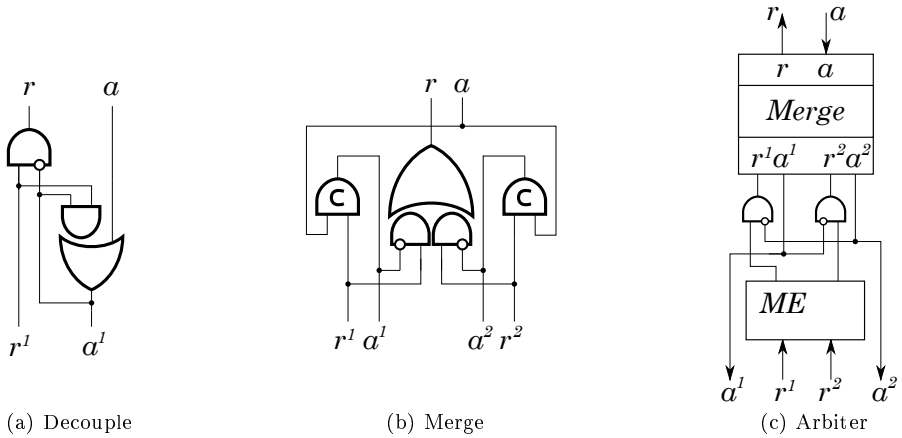
see Figure 6(b). Merge controllers can only be used if the requests are mutually exclusive[6]. If this is not the case, as e.g. for concurrent actions c^1 and c^2, then we have to set an *arbiter* guarding access to the component. Its implementation consists of the merge controller and the *mutual exclusion (ME) element* [11], see Figure 6(c).

Finally, the refined graph can be mapped into Boolean equations. An event associated with vertex $v \in V$ is enabled to fire (req_v+ is excited) when all the preceding events $u \in V$ have already fired (ack_u have been received) [17]:

$$req\_v = \phi(v) \cdot \prod_{u \in V} \big(\phi(u) \cdot \phi(u \to v) \Rightarrow ack\_u\big)$$

where $a \Rightarrow b$ stands for Boolean implication indicating 'b if a' relation. Mapping is a simple structural operation, however the obtained equations may not be optimal and should undergo the conventional logic minimisation [14][17] and technology mapping [5] procedures.

It is interesting to note that the size of the microcontroller does not depend on the number of instructions directly. There are $\Theta(|V|^2)$ conditions ϕ 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 $\Theta(|V|^2)$.



(a) Decouple (b) Merge (c) Arbiter

Figure 6: Handshake controllers

[6] It is possible to formally verify if two events in a CPOG are mutually exclusive using CPOG verification techniques from [16].

4 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 $a \dots e$) 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* [19], we associate two attributes, *execution latency* and *power consumption,* with every component. Figure 7 visualises them as labelled boxes, whose dimensions correspond to their attributes; the area of a box represents *energy* required for the computation.

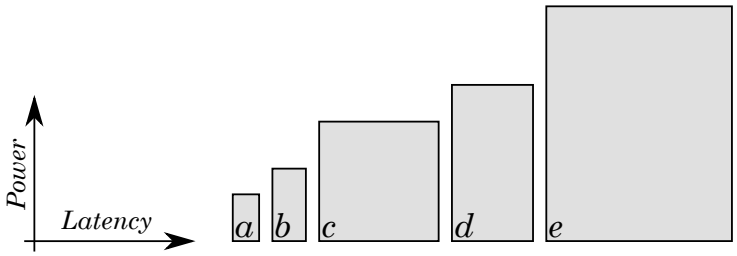


Figure 7: Datapath components for *DP3* implementation

Depending on the current operation mode and availability of the components, the processor has to schedule their activation in the appropriate partial order. Figure 8 lists several possible partial orders together with their power/latency profiles.

Fastest 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 Figure 8(a). This implementation has a very high cost in terms of peak power and thus may not always be affordable.

Least peak power implementation: a directly opposite scheduling strategy is shown in Figure 8(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 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 *component utilisation* is high.

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 Figure 8(c). While being convenient and potentially very efficient due to custom design, such solution is not always justified because of low component 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 real-time rescheduling 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 Figure 8(d).

Balanced solution: Figure 8(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.

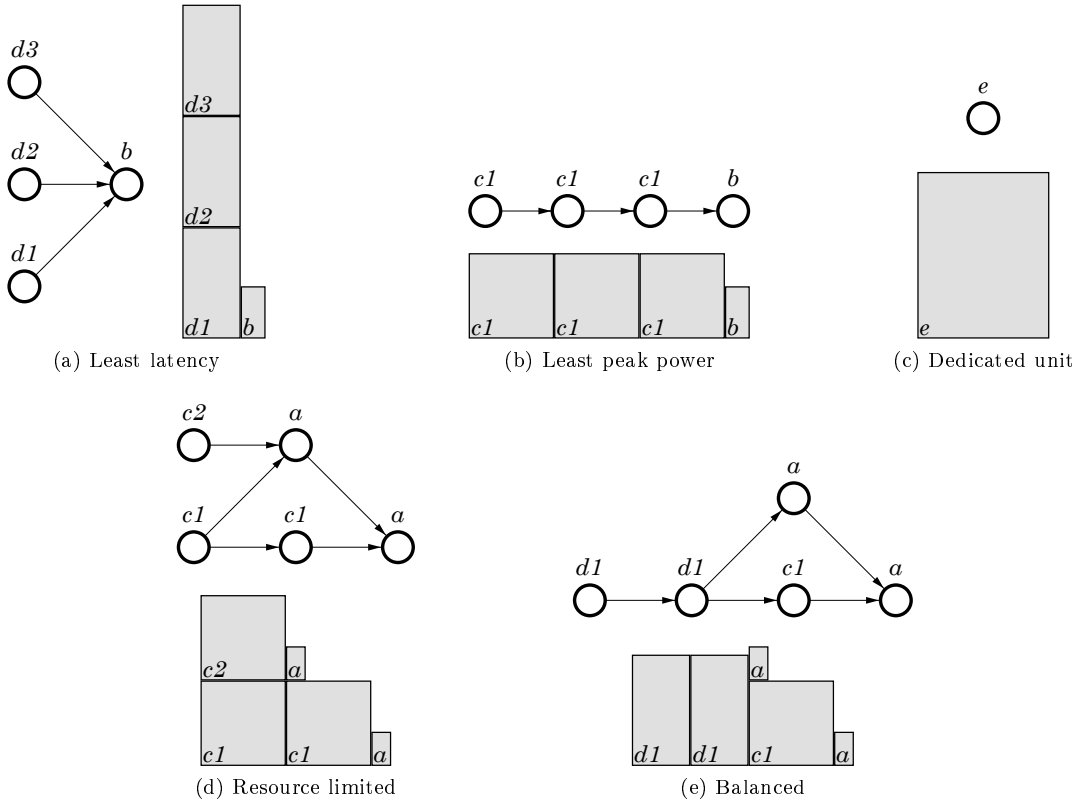


Figure 8: Different implementations of *DP3* instruction

We could continue listing different possible implementations of this instruction, but this is not the point of the case study. The point is to demonstrate that even such 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 none of them is the best. 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 perfectly suited for this task: it can represent multitude of different implementations of the same instruction efficiently. 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 [21] or be statically set to constants according to the limitations of the actual hardware platform, as shown in Figure 9.

We can specify all 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[7]

[7] We used Workcraft framework [1] for CPOG modelling and encoding.

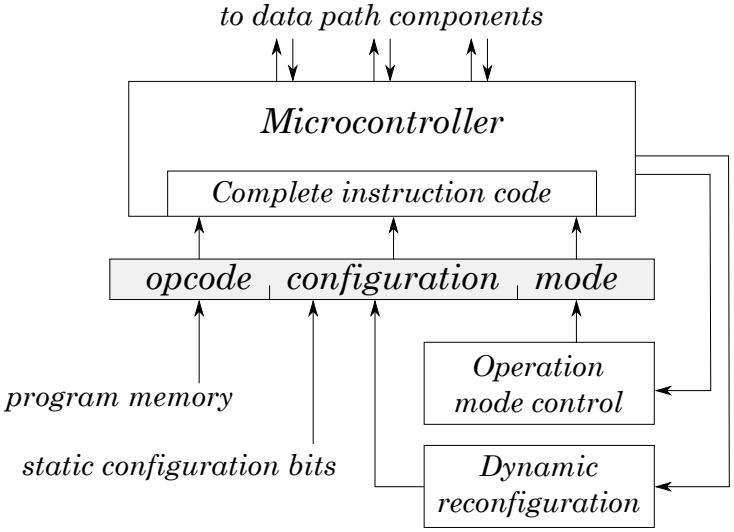
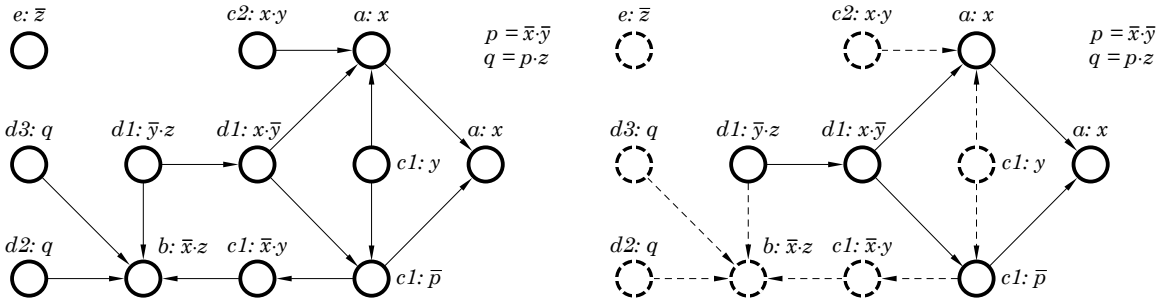


Figure 9: Complete instruction code

optimal encoding methods [15], 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 Figure 8 (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 Section 2, we obtain the graph shown in Figure 10(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, Figure 10(b), which shows the balanced implementation generated by code ψ_5, and compare it with partial order in Figure 8(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).



(a) Composition H of all implementations

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

Figure 10: CPOG specification of *DP3* instruction

In the next section we apply this approach to specification of a simple processing unit containing three instructions. Each of them is described in four implementation variants which correspond to two different modes and two different hardware configurations.

5 Experiments

This section demonstrates specification of an instruction set with three instructions running under two different operation modes and on two different hardware platforms.

In addition to instruction *DP3* described in the previous section, we consider two more instructions, namely *ADD* and *MAD*, which are traditionally supported by most GPUs at the assembly level. Instruc-

tion *ADD* computes 4-component vector sum $\mathbf{x}+\mathbf{y}=(x_1+y_1, x_2+y_2, x_3+y_3, x_4+y_4)$, i.e. it executes four addition operations $add_k = x_k + y_k$ independently, while *MAD* performs a more sophisticated task, executing four independent multiplications followed by additions: $mad_k = x_k \cdot y_k + z_k$.

Figure 11 shows the complete instruction set split into four mode/configuration sets. Hardware configuration 0 contains four multiplication and four addition components (denoted by $m1$-$m4$ and $a1$-$a4$), while configuration 1 contains only a pair of each component type ($m1$, $m2$, $a1$, and $a2$), thus being less flexible in terms of possible scheduling strategies but better in terms of component utilisation.

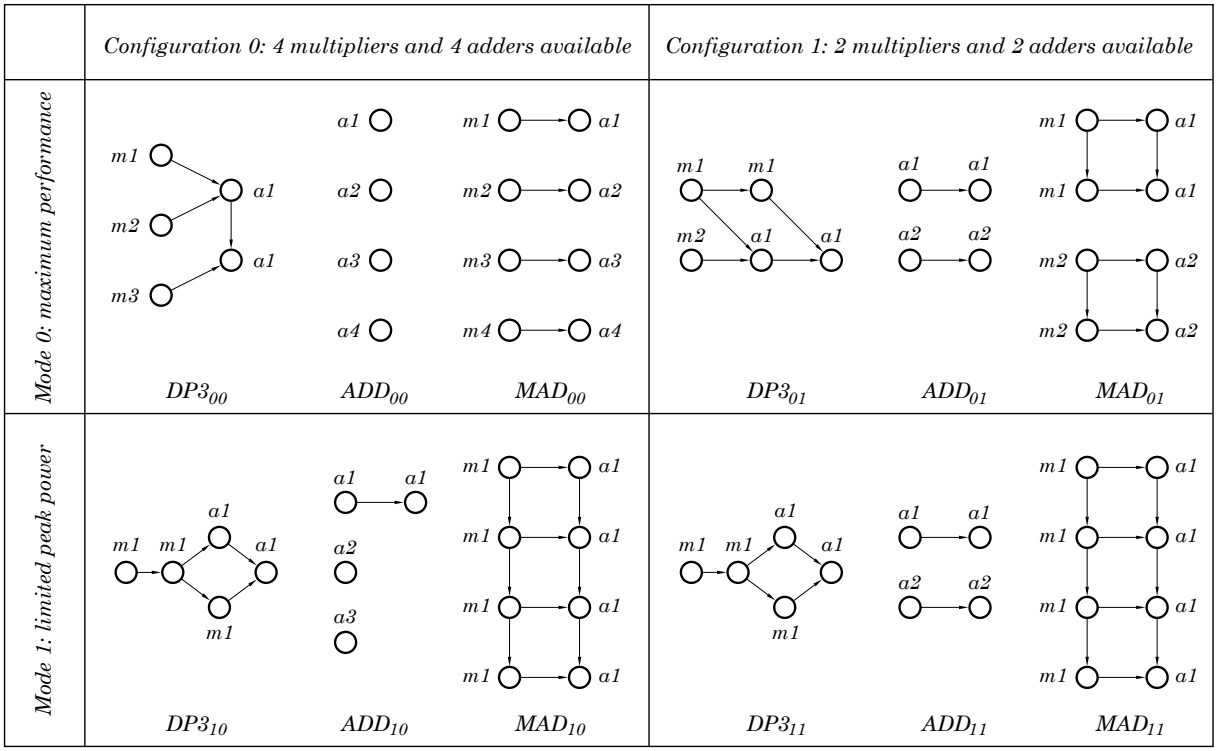


Figure 11: Complete instruction set used for experiments

There are two operation modes: mode 0 aims to maximise performance of the processor by high parallelism, while mode 1 executes the instructions under the limited power availability. The power limit was set to allow concurrent execution of a multiplier and an adder, or concurrent execution of three adders.

We use a subscript to denote the mode/configuration code of an instruction, e.g. $DP3_{01}$ stands for implementation of instruction *DP3* intended for use in mode 0 and configuration 1.

All the instructions have been composed into a single instruction set IS; the corresponding CPOG H is shown in Figure 12(a). Opcodes $\psi_{DP3} = 01$, $\psi_{ADD} = 00$, and $\psi_{MAD} = 11$ have been automatically generated by the optimal encoding procedure [15]. There are four variables in the complete instruction code: $X = \{x, y, m, c\}$, where x and y are the opcode bits, while m and c stand for the mode and configuration bits, respectively; for example, instruction MAD_{01} has code 1101. Since opcode $(x, y) = (1, 0)$ is not used, function $\rho = \overline{x} + y$ forbids it.

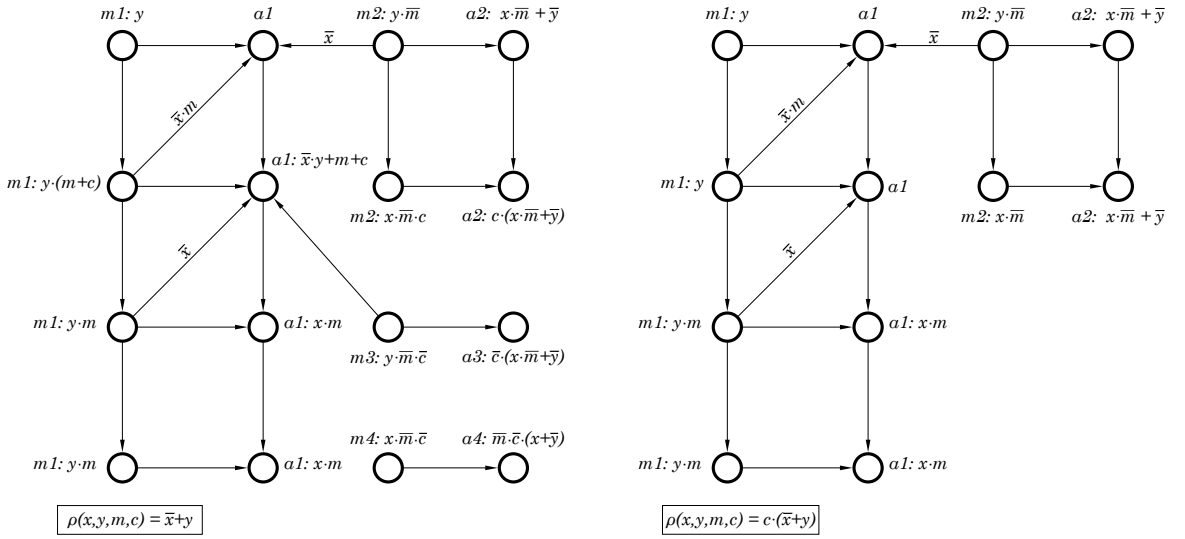
We synthesised a microcontroller for each configuration by using decomposition into $\mathsf{IS}_0 = \mathsf{IS}(\overline{c} \cdot H)$ and $\mathsf{IS}_1 = \mathsf{IS}(c \cdot H)$, followed by mapping of the obtained instruction sets into Boolean equations, as explained in Section 3. These equations were imported into *Altera Quartus II* design kit for logic minimisation and technology mapping into an FPGA board from the *Cyclone III* family; we used 32-bit multipliers and 64-bit adders in our design. Figure 12(b) shows the reduced instruction set IS_1 after logic minimisation [17] (conditions containing variable c were minimised).

Both microcontrollers have been tested to confirm the correct implementation of each instruction in terms of its functionality and the proper activation order of the datapath components. Latency and

Configuration	Mode	Instruction	Latency, ns	Peak power, mW
0	0	$DP3_{00}$	22.3	0.54
		ADD_{00}	8.0	0.32
		MAD_{00}	17.0	**0.72**
	1	$DP3_{10}$	35.7	0.26
		ADD_{10}	14.0	0.24
		MAD_{10}	44.0	0.26
1	0	$DP3_{01}$	24.8	0.36
		ADD_{01}	13.0	0.16
		MAD_{01}	27.0	0.52
	1	$DP3_{11}$	34.9	0.26
		ADD_{11}	13.6	0.16
		MAD_{11}	42.0	0.26

Table 2: Latency and peak power of instructions

peak power of each instruction have been measured using Quartus II analysis tools and are reported in Table 2. As expected, in mode 0 all the instructions are executed faster but at the expense of higher peak power (up to 0.72 mW); in mode 1, on the other hand, the peak power never gets higher than 0.26 mW. In configuration 1 the difference between the modes is smaller, because there are not enough hardware components to take advantage of maximum parallelism. Note that we were unable to perform peak power measurements with a good accuracy using *PowerPlay Analyzer* (a part of Quartus II toolkit), therefore the figures in Table 2 should be considered as a guide only.



(a) Complete instruction set $\mathsf{IS}(H)$ (b) Instruction set $\mathsf{IS}_1 = \mathsf{IS}(c \cdot H)$ after logic minimisation

Figure 12: CPOG specifications of instruction sets used in experiments

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

Acknowledgement

This work was supported by EPSRC grants EP/G037809/1 and EP/F016786/1.

References

[1] The Workcraft framework homepage. http://www.workcraft.org, 2009.

[2] Samary I. Baranov. *Logic Synthesis for Control Automata*. Kluwer Academic Publishers, 1994.

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

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

[5] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic synthesis of asynchronous controllers and interfaces*. Advanced Microelectronics. Springer-Verlag, 2002.

[6] Shidhartha Das, Carlos Tokunaga, Sanjay Pant, W. H. Ma, Sudherssen Kalaiselvan, Kevin Lai, David M. Bull, and David T. Blaauw. Razor ii: in situ error detection and correction for pvt and ser tolerance. *IEEE Trans. Solid-State Circuits*, 44:32–48, 2009.

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

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

[9] Patrick Gelsinger. Design and test of the 80386. *IEEE Design and Test of Computers*, 4:42–50, 1987.

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

[11] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*. John Wiley and Sons, 2008. ISBN: 978-0-470-51082-7.

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

[13] Mikko H. Lipasti and John Paul Shen. Superspeculative microarchitecture for beyond ad 2000. *Computer*, 30:59–66, September 1997.

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

[15] Andrey Mokhov, Arseniy Alekseyev, and Alex Yakovlev. Automated synthesis of instruction codes in the context of micro-architecture design. In *Proceedings of International Conference on Applicatioon of Concurrency to System Design (ACSD'10)*, pages 3–12, 2010.

[16] Andrey Mokhov and Alex Yakovlev. Verification of Conditional Partial Order Graphs. In *Proc. of 8th Int. Conf. on Applicatioon of Concurrency to System Design (ACSD'08)*, 2008.

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

[18] Magnus 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.

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

[20] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, 2001.

[21] Fei Xia, Andrey Mokhov, Yu Zhou, Yuan Chen, Isi Mitrani, Delong Shang, Danil Sokolov, and Alex Yakovlev. Towards power elastic systems through concurrency management. Technical report, Newcastle University, July 2010.