Ensuring consistency between cycle-accurate and instruction set simulators

Fatma Jebali, Dumitru Potop-Butucaru

Ensuring consistency between cycle-accurate and instruction set simulators

Fatma Jebali
Inria
Paris, France
fatma.jebali@inria.fr

Dumitru Potop-Butucaru
Inria
Paris, France
dumitru.potop_butucaru@inria.fr

***Abstract*—The xMAS micro-architecture modeling language has been introduced by Intel to facilitate the formal representation and analysis of on-chip interconnect fabrics. In this paper, we introduce xMAStime, a new domain-specific language inspired by xMAS. xMAStime allows the modeling of full micro-architectures comprising certain classes of CPU pipelines, caches, and RAM. Given an in-order pipeline model in xMAStime, we automatically generate both a Cycle-Accurate, Bit-Accurate (CABA) hardware simulator and a timed instruction set simulator where time is accounted with safe upper bounds, as in the pipeline analysis step of Worst-Case Execution Time (WCET) analysis. The approach relies on the theory of endochronous systems, which allows us to ensure functional equivalence and timing consistency between the two generated simulators, using a delay-insensitivity argument. xMAStime is implemented over Lucid Synchrone – a dataflow synchronous language featuring a higher order type system and type inference, which facilitate the definition of our DSL. We use the new DSL to model and synthesize simulation code for a full-fledged MIPS32-based architecture.**

I. Introduction

One key difficulty in embedded systems design is the existence of multiple models of the same hardware system, developed separately, at different abstraction levels, and used in various phases of the design flow. In the design of real-time embedded systems, we can identify:

- Cycle-accurate system models [C$^+$03] used to perform fine-grain hardware simulation, mostly during HW and driver design phases. These models provide an exact functional and temporal representation of system execution.
- Microarchitectural models used for pipeline simulation during WCET (*Worst-Case Execution Time*) analysis [HCFR13], [HFC14], [HRP17]. These models are used to compute safe over-approximations of the duration of a sequential piece of code, i.e., one function running without interruption on a processor core). To provide precise results, these models preserve much of the microarchitectural detail of processor pipelines and memory hierarchy (e.g. cache states, data transfer latencies).
- System-level models used for WCRT (*Worst-Case Response Time*) analysis [HP07]. Assuming that the contribution of processor pipelines and caches is taken into account by WCET analysis, which provides safe upper bounds on the duration of sequential tasks, WCRT models provide the information allowing to derive system-level timing guarantees. Important information here includes the structure of the interconnect and the way resources (caches, memory banks, *etc.*) are shared.

In this paper, we are concerned with the first two models, which are both hardware simulators with a cyclic activation pattern. Establishing semantic consistency between these two simulation models is challenging for several reasons. First, the activation pattern, which is the logical time base of the simulation, depends on the abstraction level. In cycle-accurate models, simulation cycles correspond to hardware clock ticks, whereas in WCET analysis models they correspond to changes in the program counter of the sequential program. Second, data abstractions are different in the two simulation models. Cycle-accurate simulators are often also *bit-accurate*, *i.e.* provide exactly the same results as the actual hardware. By comparison, pipeline simulators in WCET analysis abstract away most data types and related operators, typically retaining only Booleans, which can be exploited at analysis time. Last, but not least, the simulators are usually pieces of C/C++ code manually written by different teams or obtained through complex translation processes from high-level Architecture Description Languages (ADLs) that may not have a clear semantics. Formally relating such pieces of code is difficult.

Contribution. This paper proposes a method to ensure the semantic consistency between the two HW models we consider, focusing on time abstraction issues. Our method relies on *desynchronization* theory [PBCB06], which defines sufficient properties ensuring that a synchronous model can be seen as an asynchronous Kahn Process Network (KPN). When a synchronous HW model satisfies these properties, any scheduling of its computations that is compatible with data dependencies will produce the same result (a property known as scheduling-independence). We show how to control scheduling through changes of the logical time base of the model prior to code generation using a synchronous language compiler. In particular, a careful choice of the logical time base allows us to produce, from the same model, either a cycle-accurate simulator, or the one needed for WCET analysis. In conjunction with some data abstraction, this logical time manipulation allows the synthesis of semantically consistent simulators from a single model.

Furthermore, we can ensure by construction that synchronous models satisfy the properties required by desynchronization theory. To this end, we introduce a new hardware

modelling language, named xMAStime, allowing the compositional modeling of systems satisfying the required properties.

Outline. The remainder of the paper is organized as follows. Section II introduces both the problem and the principles of the solution we propose. Section III introduces the Lucid Synchrone language used to implement xMAStime. Section IV presents the xMAS formalism which inspired the choice of primitives of xMAStime. In Section V, we define xMAStime. We present our MIPS32 case study in Section VI. Section VIII concludes the paper and gives directions for future work.

II. Motivation and method overview

A. Motivating example

To illustrate the desired behaviour of cycle-accurate and WCET analysis pipeline simulators, we consider the two-stage pipeline of Fig. 1(a). The pipeline is used to execute sequential programs with no branching. We do not detail here the function of the pipeline stages, nor the form or function of the instructions. We assume that each pipeline stage consists of exactly one unit (U1 and U2) and that the units are sequential resources, meaning that only one instruction can use a given unit at any given time. We also assume that all instructions use both pipeline stages.

Fig. 1(c,d) provides graphical representations of the pipeline simulations we want to obtain for the two-instruction program in Fig. 1(b). Cycle-accurate simulators, like those implemented using the SoCLib platform [C+03], describe the exact behavior of the modeled hardware. A cycle-accurate simulation is driven by the advancement of the hardware clock (clk in Fig. 1(c)). For each instruction, the pipeline units are used in order. For instance, I1 uses U1 on cycles 0 and 1 and U2 on cycles 2 and 3. Execution proceeds in an ASAP (as soon as possible) fashion. As soon as I1 leaves U1, it is succeeded by I2. Pipeline parallelism means that different units can execute different instructions at the same time.

The execution time of a unit may depend on the instruction and on the data. For instance, the duration of I1 on U1 is smaller than the worst-case duration of U1, which is assumed to be equal to 3 clock cycles. The worst-case duration of U2 is assumed to be 2 clock cycles.

The precision of cycle-accurate simulation, which requires the traversal of the pipeline model (to make decisions and advance simulation) at each clock cycle, has a high computational cost. To avoid this cost, *instruction set simulators* only traverse the model once per instruction, and may additionnally make simplifying assumptions. We are interested here in instruction set simulators that keep an accounting of time, like those used in WCET analysis tools such as Heptane [HRP17].

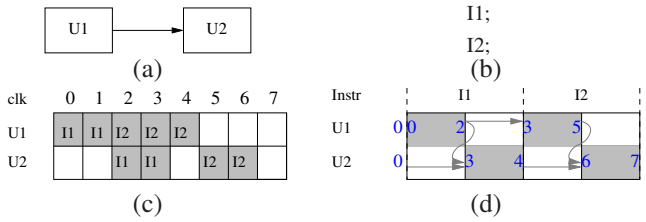


Fig. 1: Two-stage pipeline (a), two-instruction program (b), cycle-accurate simulation (c), instruction set simulation (d)

The functioning of such simulators is illustrated in Fig. 1(d). The use of each pipeline stage/unit is tracked with one counter initialized at 0 (the 0 figures on the left). During the simulation of one instruction, the counter of each stage used by the instruction is modified twice, to record the first and last clock cycles where the instruction uses the stage. These are respectively the left and right numbers inside each greyed rectangle. These figures are computed in a classical max-plus fashion. Executing an instruction on stage Ui can only start after the previous instruction has completed execution on Ui, and after the data dependencies of the current instruction on Ui have been solved. These dependencies are represented with arrows, and determine why, for instance, the start date of I2 on U2 is $6 = 1 + max(4, 5)$. The last cycle of I2 on U2 is simply computed as the first cycle plus the worst-case duration of U2.

The use of worst-case stage durations means that the start and end cycles of an instruction on a stage do not exactly match between the cycle-accurate and instruction set simulations. However, for the instruction simulation figures to be used in a WCET analysis tool, they must always be greater than or equal to the corresponding values produced through cycle-accurate simulation. *In this paper, our objective is to ensure that the simulators satisfy this timing abstraction property, in addition to the functional semantics preservation due to scheduling-independence.* For instance, in the cycle-accurate simulation, the end cycle of I2 on U2 is 6, whereas it is 7 in the instruction set simulation.

B. Method overview

In stark comparison with the previous example, formally stating and ensuring the functional semantics preservation and timing abstraction is challenging for complex, real-life pipelines featuring multiple stages, bypasses, register banks and hazards, *etc.* We propose to derive both simulators from the same hardware model in a way that ensures consistency. As modelling formalism, we rely on dataflow synchronous languages. This is natural, given that both cycle-accurate and instruction set simulations comply with the synchronous execution model [BCE+03]. Synchronous language compilers allow the automatic translation of our models into simulation code. Furthermore, synchronous languages have well-defined semantics allowing the precise description of the concurrent semantics of pipelined hardware.

1) The two simulation levels: To illustrate the construction of two simulators starting from the same synchronous program, consider the following Lucid Synchrone program (syntax presented in Section III), meant to provide a (first, incomplete) representation of the pipeline in Fig. 1(a):

```
1  let node simple () = () where
2    rec x = u1( () )
3    and (_) = u2(x)
```

Listing 1: First representation of the two-stage pipeline

Lucid Synchrone being a dataflow synchronous language, programs are structured as hierarchical dataflow *nodes*. This node represents a full system, so it has no inputs, nor outputs. The unique variable `x` is used to transfer data from U1 to U2. The type of `x` is inferred from the types of U1 and U2.

Producing two simulators from this dataflow program is done by using two versions of the U1 and U2 units. To produce a cycle-accurate simulator, where execution cycles correspond to the advancement of the HW clock, we use cycle-accurate versions of the units. For instance, to obtain the trace of Fig. 1(c), unit U2 takes two cycles between receiving an input and completing execution. Note that even units with no input or no output in the graphical representation have at least one input and one output in textual form. Drawing from functional programming, we use the special type `unit` and its unique value `()` to represent computation triggering and synchronization.

To produce an instruction set simulator, the implementations of U1 and U2 are different. Upon reception of an input they produce the corresponding output *in the same execution cycle*. Thus, all computations associated with an instruction are performed in exactly one execution cycle of the synchronous program. To perform the worst-case timing accounting, all dataflow variables are tagged with integer dates, which are propagated in a max-plus fashion, using the dedicated constructs presented in Section V. Note that type inference allows us to use the same program, unmodified. However, *to ensure functional semantics preservation between the cycle-accurate and instruction set simulations, the two versions of each unit and the system model (the node) instantiating them must satisfy the properties defined in Sections II-B3 and II-B4*.

2) In-order pipelines: The construction of the instruction set simulator, as detailed above, does not always produce a correct simulator. Consider the example of Fig. 2. Here, the pipeline has 4 units, and each of the two instructions of the program use only a sub-set of these units (U0, U1, U2 for I1 and U0, U3, U2 for I2). The pipeline also contains a demultiplexer which sends the output of U0 to either U1 or U3, and a multiplexer which takes the results from U1 and U3 and sends them to U2 *in the order in which they arrived at the multiplexer*. Coupled with the hypothesis that the duration of U2 is variable, this means that the order in which I1 and I2 are executed on U2 depends on the actual duration of I2 on U3. In Fig. 2(c) this duration is 3 cycles, and I1 is executed before I2. In Fig. 2(d) this duration is one cycle, and I2 is executed before I1.

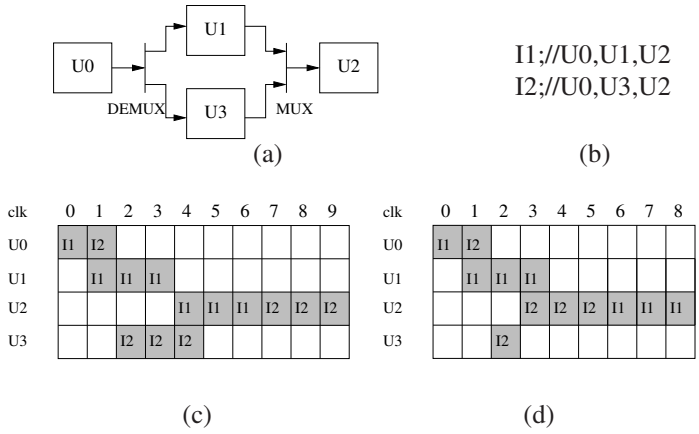


Fig. 2: Delay-sensitive, out-of-order pipeline (a) with a program (b) and two possible execution traces (c,d). Program comments provide the units used by each instruction.

We say that the behavior of this pipeline is delay-sensitive, and that it allows *out-of-order execution*. Clearly, instruction set simulation taking into account timing aspects is not possible for out-of-order pipelines, where the execution of an instruction may depend on subsequent instructions. The simulation code can be generated, but the simulation will be incorrect for the case of Fig. 2(d).

Out-of-order execution is a key feature of most of today's high-performance processors [HP07]. This feature is usually coupled with branch prediction and multi-core cache coherency logic, making static timing analysis virtually impossible. For this reason, our work here targets simpler, more predictable processor cores, like those used in embedded processors and in many-cores (e.g. Kalray MPPA [dDvAPL14]).

3) Endochronous components: To determine when synchronous pipeline models can be given cycle-accurate and instruction set simulators which are consistent with each other, we rely on the theory of endochronous systems [PBCB06].

Endochrony is a property of synchronous components, like those forming our pipelines. An endochronous component is delay-insensitive. Its behavior does not depend on the interleaving of inputs arriving on different input channels, but only on the sequences of inputs received separately on each channel. For instance, the units U0-3 and the demultiplexer of Fig. 2(a) are each endochronous, but the multiplexer is not, because the sequence of outputs depends on the interleaving of inputs arriving from U1 and U3.

Delay-insensitivity means that endochronous components can be seen as asynchronous components that are deterministic as functions from streams of inputs to streams of outputs. Thus, a pipeline formed only of endochronous components will be a *Kahn Process Network (KPN)* [G.74], with deterministic, delay-insensitive asynchronous semantics. Furthermore, replacing one component with another representing the same function from streams of inputs to stream of outputs does not change the (asynchronous) behavior of the KPN. For instance, replacing the cycle-accurate implementations of U1 and U2 with the instruction set simulation ones in the two-stage pipeline model does not change the KPN semantics of the model if these implementations represent the same stream functions.

Endochrony is compositional, allowing the incremental construction of endochronous components (and can also be checked, albeit at a high computational cost).

4) Model-level correctness properties: In addition to the endochrony of components, our pipeline models must satisfy three supplementary propeties ensuring the correctness of the simulators and their semantic consistency.

a) Isochrony: The cycle-accurate and instruction set simulation models must be correct synchronous programs. Consider the pipeline model in Fig. 3, and assume that U2 needs one input on each of its inputs to perform an execution cycle. While describing a correct asynchronous KPN, the corresponding instruction set simulation model is rejected by the synchronous language compiler because the inputs of U2 have different clocks (cf. Section III).

Requiring the correction of the instruction set simulator program ensures the *isochrony* of the model, the companion property of endochrony [PBCB06] needed to ensure correct desynchronization.

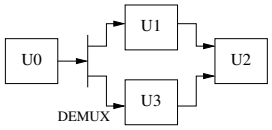


Fig. 3: Non-isochronous pipeline model

b) Boundedness: In the original definition of KPN, communication channels are lossless unbounded FIFOs that can accumulate incoming values waiting consumption. However, synchronous models and real hardware do not allow such an accumulation process. To ensure the absence of accumulation, processor pipelines rely on both timing arguments and on feedback mechanisms. For instance, the model in Fig. 1(a) is not bounded. If the duration of U1 is 2 and that of U2 is 4, then in an ASAP execution model U1 produces twice as much values as U2 can consume, the rest being lost/overwritten.

We require that our HW models are 1-bounded *i.e.* that in the asynchronous KPN interpretation of the model, regardless of the durations and triggering policies of the pipeline units, no dataflow variable may need to store more than one value at a time. The pipeline model in Fig. 1(a) can be completed to a 1-bounded model as pictured in Fig. 4. The elements

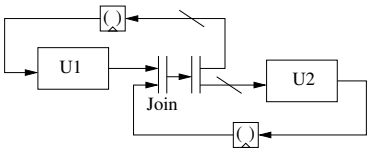


Fig. 4: Two-stage pipeline, 1-bounded model

of this model will be formally introduced in Section V. The key elements of the feedback loop ensuring 1-boundedness are the Join element that waits for one value on each input before producing them both as a pair on the output, and the two 1-place buffers, initially filled with synchronization values `()`. The token passing mechanism implemented by these nodes ensure that U1 and U2 are used as sequential resources, and that after producing a value `v`, U1 can only start producing another one after U2 has consumed `v`.

The 1-boundedness property can be ensured by enforcing structural properties. In systems without conditional execution like the one in Fig. 4, a necessary and sufficient property ensuring 1-boundedness is that each variable is part in a dataflow cycle containing exactly one 1-place buffer that is initially filled.

c) In-order execution: The final property we have to ensure is in-order execution, as introduced above. Checking that a model does not allow out-of-order execution can be done either statically, through sufficient structural properties, or dynamically by tagging all values with the instruction index and then checking at runtime that the tags never decrease.

III. Lucid Synchrone

Lucid Synchrone [Pou06] is a dialect of the dataflow synchronous language Lustre/Scade [HCRP91]. It has been built on top of OCaml [LDF+14], inheriting its higher-order type system with type inference. This allows us to use the same source code for the top-level node of both the cycle-accurate and the instruction set simulators, even though the type of the variables changes.[1]

A Lucid Synchrone program is formed of *equations*, each equation assigning values to zero or more variables. At each execution cycle of the program, a variable can be *absent*, in which case it cannot be used in computations during the cycle, or *present*, in which case it is computed once before being used with the same value in all use points during the cycle. Some variables are called *clocks*. They have Boolean type and are used to determine when other variables are computed or used. We say that two variables have the same clock if they are both present or both absent at each cycle.

All Lucid Synchrone statements can be derived starting from OCaml function calls[2] and three dataflow primitives: `merge` (deterministic multiplexer), `when` (sub-sampling), and `fby` (memory between execution cycles). The inputs of a function call must all have the same clock. When the inputs are present, the function call is performed, and the outputs assigned a value. If variables `a` and `c` have the same clock and `c` is a clock variable, "`a when c`" represents a value that is present (and equal to `a`) only at instants where `c` is present with value true. Conversely, "`merge c a b`" has the same clock as `c` and the same data type as both `a` and `b`. It can be defined when `c` is a clock variable, and when `a` is present only at cycles where `c` is true, and when `b` is present only at cycles where `c` is false. The value output by `merge` is that of the present argument `a` or `b`. The value produced by "`k fby a`" has the same clock and type as `a`. The expression stores values from one cycle where `a` is present to the next. At the first cycle where `a` is present, the output value is the constant `k`.

Of the derived constructs of Lucid Synchrone, we shall make use of only one: the *valued signals*, which allow for an event-driven programming style. Each signal `s` can be seen a pair formed of a clock variable `c_s` and a data variable `d_s` having clock `c_s`. Signals are produced and read using the constructs `emit` and `present`. When executed, `emit o = exp` will set the clock of `o` to true and its value to that of `exp` for the current cycle. The `present`

[1]For instance, in instruction set simulation, all variables are tagged with an integer date, which is not needed in cycle-accurate simulation.

[2]Tuple operations can also be seen as function calls.

construct performs pattern-matching on the clocks of signals much like the `match` statement of OCaml performs it on regular variables.

IV. The xMAS Language

A. Syntax and intuitive semantics

The core of the xMAS formalism [CKO12] is a graphical dataflow language. The eight primitive dataflow operators of the language are depicted in Fig. 5, and their intuitive semantics is the following:

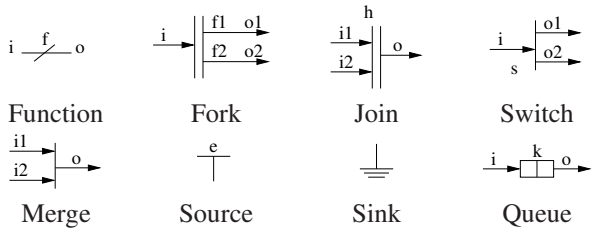


Fig. 5: The xMAS primitives

- Upon arrival of an input x on input port i, a *function call* operator outputs $f(x)$ on o.
- Upon arrival of an input x on i, a *fork* operator produces on each output port one value, obtained by applying to x the function associated to the port (f_1 and f_2 in Fig. 5).
- A *join* operator requires inputs to arrive at the same time on all input ports. When this happens, the operator sends on the output port a value computed from the input ones using the function h.
- Upon arrival of an input x on i, a *switch* operator sends x on the output $s(x)$, where $s(x)$ belongs to $\{0, 1\}$.
- The *merge* operator is a multiplexer. It transfers inputs to the output using a user-definable arbitration policy.
- The *source* and *sink* operators are the inputs and outputs of an xMAX specification. They respectively produce and consume messages following a user-definable policy representing the environment behavior, and the ability of the xMAS model to accept or provide a value.
- The *queue* is the only xMAS operator able to store values between execution cycles. A queue is a lossless FIFO with limited, specified capacity. A queue is initially empty. When full, it cannot accept inputs. When empty, it cannot produce outputs. A queue can accept input and produce output at the same time, but a value cannot be output immediately upon reception.

Operators are connected through point-to-point channels, which are the dataflow arcs. Channels are reliable: they cannot create, duplicate, or loose messages.

B. Operational semantics

Semantics borrows from synchronous formalisms while introducing some nondeterminism. Execution is cyclic. At each cycle, some values can be read from source operators and from queues, some computation performed, and some values output to sink operators or stored into queues.

Execution cycles are sequences of operator actions performed according to the operator semantics defined above. However, not all sequences of actions built in this way are valid execution cycles. In addition to the obvious *causality* constraint that a value is produced on a channel before it can be consumed, each cycle must satisfy a *global consistency* constraint: Between cycles, no value can be stored outside of the FIFO queues (neither on other operators, nor on channels).

This semantics can be seen as transactional, and is difficult to implement. Existing xMAS implementations rely on a *low-level encoding* of channels based on systematic use of back-pressure signaling, whose operational event-driven semantics detects valid propagation paths for data messages [CKO12].

C. Imperative xMAS

The original presentation of xMAS focuses on its dataflow part – the graphical language presented above – which is close in form and semantics to dataflow synchronous languages such as Lustre.

But xMAS is not complete without the definition of the policies governing the behavior of source, sink, and merge operators. The specification of these policies follows a very different presentation style:

- They are specified under the form of *imperative* code.
- Their code refers to the low-level encoding of channels mentioned above.
- They can make decisions based on the *absence* of dataflow values, not only on their presence. Absence can be determined with tests on the variables forming the low-level encoding of the channels.

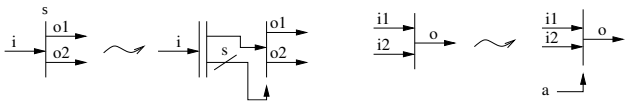
Testing the absence of dataflow values means that the behavior of an xMAS specifications may depend on the order in which inputs arrive through source operators and/or the order in which internal computations are performed, which in turn may depend on the relative speeds of the various operators. We say that, in general, xMAS specifications are *scheduling-dependent* or *delay-sensitive*.

D. Simplifications on xMAS

For presentation purposes and without loss of generality, we consider in this paper the following restrictions on xMAS:

- When an operator may have more than one input or more than one output, the number of inputs/outputs is restricted to two.
- In fork operators, the functions associated with outputs are both identity. Thus, a fork simply duplicates its input on all outputs. The combinator function of a join operator produces the pair of the two input values. Note that transformations on outputs, if needed, can always be performed outside the fork or join operator by using function call operators.

For the sake of clarity, we shall also proceed to a refactoring of operators switch and merge to ensure a better orthogonality between xMAS primitives. As pictured in Fig. 6(a) we separate the routing function from the switch operator, and represent it with a separate function operator that outputs a Boolean used



(a) Switch (b) Merge

Fig. 6: Refactoring xMAS primitives

by the refactored switch to make the routing decision (0/false routes the input to the top output, and 1/true to the bottom one).

As pictured in Fig. 6(b), we separate the arbitration policy from the merge operator. The arbitration decision is represented as a Boolean input to Merge, determining which input must be transmitted to the output. Unlike in switch operators, the arbitration decision cannot always be computed using a simple function operator. For instance, the representation of classical Round-Robin arbiters [CKO12] requires making decisions based on the absence of values on inputs, which is done with tests on the state of the low-level channel encoding. In such cases, we encapsulate the low-level decisions into source operators whose input values and policy correspond to the computation of the arbitration decisions, as pictured in Fig. 7.

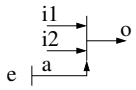


Fig. 7: Refactored fair arbitration using a source

V. From xMAS to xMAStime

Previous work on xMAS showed that its set of primitives provides the good mix of asynchronous and synchronous features for the description of complex interconnect fabrics for verification purposes. Our experiments showed that the xMAS primitives also allow the description of full architectures, including CPU pipelines. They allow the natural representation of asynchronous pipeline interlocking (as in Fig. 4), instruction dispatch to functional units (as in Fig. 2) and the complex behavior of register files, as in the MIPS32 example of Section VI.

However, the non-deterministic and transactional semantics of xMAS does not provide good support for the synthesis of efficient simulation code. We introduce here a new language based on xMAS but with synchronous semantics, that facilitates the specification of complex pipelines and at the same time allows efficient code generation using an existing synchronous language compiler.

The new formalism is called xMAStime. As earlier explained, from each xMAStime model we derive two synchronous programs corresponding to the two simulation levels we target. The simulators are obtained by compilation of the two synchronous programs, when the correctness properties of Sections II-B3 and II-B4 are respected. To attain this workflow, we implemented xMAStime on top of the Lucid Synchrone language presented in Section III. More precisely, xMAStime is an API with two implementations. The xMAStime models are developed using only primitives of this API. Then, the synchronous models corresponding to the cycle-accurate or instruction set simulators are obtained by using the corresponding API implementation. We describe here the API and its two implementations.

A. The xMAStime API

An xMAStime model is a Lucid Synchrone program—a dataflow *node*. Its inputs and outputs are represented in the *interface* of the node, and not (like in xMAS) with dedicated primitives. The use of Lucid Synchrone's module system, which xMAS does not have, facilitates the modular construction of complex architecture models. But the difference is not merely syntactical. The sink and source xMAS primitives allow the specification of production and consumption policies, which cannot be directly expressed in xMAStime. This is not an issue, because xMAStime is meant to represent full hardware architectures evolving in isolation, and not parts thereof. Like in Listing 1, the main node of an xMAStime model must have no inputs and no outputs, as it describes a CPU accessing its own RAMs to perform computations *without interrupts from the environment*. Interaction with the environment can be modeled as the sampling of memory-mapped devices.

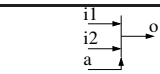
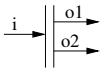
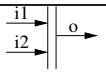
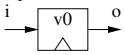
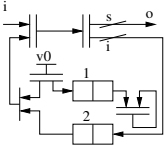
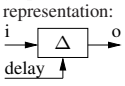
xMAStime primitive	xMAS (refactored) correspondent	Lucid Synchrone intuitive correspondent
`xt_fcall`		`o = f(i)`
`xt_switch`		`o1 = i when s` `o2 = i when not s`
`xt_merge`		`o = merge s i1 i2`
`xt_fork`		`o1 = i` `o2 = i`
`xt_join`		`o = (i1, i2)`
`xt_fby` graphical representation: 		`o = v0 fby i`
`xt_time_delay` graphical representation: 	none	none

Fig. 8: xMAStime API and corresponding xMAS primitives

The API proper contains seven primitives, listed in Fig. 8 along with their intuitive xMAS and dataflow synchronous correspondent. For the first five primitives, the correspondence with both the xMAS counterpart (in the simplified version we introduced in Section IV-D) and a dataflow synchronous

statement is straightforward. When graphically representing xMAStime models we shall use for these five primitives the corresponding xMAS representation.

The queue construct of xMAS is replaced in xMAStime with the `fby` statement specific to dataflow synchronous languages, hence its classical graphical representation. This will allow the direct use of Lucid Synchrone's synchronous semantics to interpret the resulting specifications. It is important to note that the behavior of the synchronous `fby` can be implemented in xMAS using the construction of Fig. 8. Note the complex initialization circuitry needed to bring the $v0$ value from a source to the initially empty queues.

The last construct is the time accounting primitive, with no direct counterpart in either xMAS or Lucid Synchrone. To preserve the close syntactic proximity with xMAS, we require that each variable (including node inputs) is used as input in at most one primitive. In other words, the use of broadcast is replaced with that of `xt_fork`.

```
1 (* cycle-accurate implementation *)
2 let node xt_fcall (static f) i = o where
3   present
4   | i(v) -> do emit o = f(v) done
5   end
6 (* instruction set simulation implementation *)
7 let node xt_fcall (static f) i = o where
8   rec (i_val,i_tag) = i     (* separate tag from value *)
9   and o = (f(i_val),i_tag) (* tag remains unchanged *)
```

Listing 2: Implementation of xt_fcall

B. API implementation

The implementation of the xMAStime API is provided in Listings 3-5 and Listings 7-10. Recall from the motivating example that time advancement is encoded in a different fashion in cycle-accurate and instruction set simulations. In instruction set simulation, time must be explicitly represented with tags annotating the values transmitted through dataflow variables. This is the case in Listing 2 (lines 7-9), where the input value is first decoded as a pair formed of the data value and its time tag. The output is constructed from the computed data value `f(i_val)` and the tag received as input. Note that, in line with synchronous language modeling, function computations do not take time.

Tag modification follows a max-plus approach. Each primitive receiving multiple inputs (such as `xt_join`, in Listing 3, lines 11-14) will compute the output tags as the maximum of the input ones. Accounting for operation durations is done with the `xt_time_delay` primitive (Listing 5, lines 25-27), which adds to the tag a value computed from the input data value using the provided `delay` function. To model a functional unit and its duration, one needs to use an `xt_fcall` primitive for the functional part, and an `xt_time_delay` primitive to account for the duration.

```
1 (* cycle-accurate implementation *)
2 let node xt_join (i1, i2) = o where
3   rec (o1, v1) = buffer1 (i1, c)
4   and (o2, v2) = buffer1 (i2, c)
5   and clock c = (?i1 || (v1 != None)) && (?i2 || (v2 != None))
6   and emit o = match (o1 when c, o2 when c) with
7                | (Some x1, Some x2) -> x1, x2
8              | _ -> failwith "xt_join"
9             end
10 (* instruction set simulation implementation *)
11 let node xt_join (i1, i2) = o where
12   rec (i1_val, i1_tag) = i1
13   and (i2_val, i2_tag) = i2
14   and o = ((i1_val, i2_val), (max i1_tag i2_tag))
```

Listing 3: Implementation of xt_join

```
1 (* cycle-accurate implementation *)
2 let node xt_fby (static def) i = o where
3   rec emit init_sig = def
4   and o = init_sig fby i
5 (* instruction set simulation implementation *)
6 let node xt_fby (static def) i = o where
7   o = def fby i
```

Listing 4: Implementation of xt_fby

Note that Lucid Synchrone nodes can have static parameters, which are not dataflow variables. Static parameters are fixed and do not change during node execution, like the function executed by `xt_fcall`.

```
1 (* cycle-accurate implementation *)
2 let node xt_time_delay (delta, i) = o where
3   rec inp = present
4             | (xt_join (delta,i))(d,x) -> let a = (d,x) in Some a
5             | _ -> None
6            end
7   and (osample, nxt) =
8       match inp, prev with
9       | Some a, Some b -> let (di,xi) = a and (dp,xp) = b in
10                          if dp = 0 then (Some xp, Some a)
11                          else failwith "xt_time_delay overflow"
12      | Some a, None | None, Some a ->
13          let (d,x) = a in
14          if d = 0 then (Some x,None)
15          else let b = (d-1,x) in (None, Some b)
16      | None, None -> None,None
17      end
18  and prev = None fby nxt
19  and clock c = (osample <> None)
20  and emit o = match osample when c with
21               | Some x -> x
22               | None -> failwith "xt_time_delay unreachable"
23              end
24 (* instruction set simulation implementation *)
25 let node xt_time_delay (delta, i) = o where
26   rec (i_val,i_tag) = i
27   and o = (i_val, i_tag + (delta i))
```

Listing 5: Implementation of xt_time_delay

In cycle-accurate simulation, the simulation clock is the clock cycle of the architecture and it counts the advancement of time. We no longer need time tags associated with dataflow values. The difficulty here comes from the fact that the operations of primitives such as `xt_join` or `xt_time_delay` are no longer instantaneous, and their duration may depend on data values (*e.g.* `delta` in Listing 5). The Lucid Synchrone compiler is no longer capable of automatically inferring the relations between activation conditions of the primitives, and we need to resort to a more asynchronous programming style. We do this using the *signals* of Lucid Synchrone, which are *emitted* and whose *presence* can be tested using

dedicated primitives. In cycle-accurate simulation, all dataflow variables are signals. In Listing 2 (lines 2-5), the cycle-accurate implementation of `xt_fcall` will emit a value `f(v)` on the output `o` whenever `i` is present with value `v`.

We are now ready to provide (in Listing 6) the xMAStime textual source of the two-stage pipeline of Fig. 4. The listing contains the main node `simple_timed` and the auxiliary node `xt_resource` that models a resource whose functionality is provided by function `f` and duration is provided by function `delay`. The data processing and duration computation functions, which can be defined in OCaml, are not provided. Note how `xt_resource` uses the `delay` function to aliment the `xt_time_delay` primitive which accounts for the duration of the resource computation.

```
1  let node xt_resource (static f) (static delay) i = o where
2    rec (i1, i2) = xt_fork i
3    and x = xt_fcall f i1
4    and o = xt_time_delay (xt_fcall delay i2, x)
5
6  let node simple_timed () = () where
7    rec x = xt_resource f1 d1 (xt_fby () (xt_fcall snd sync1))
8    and sync2 = xt_resource f2 d2 (xt_fcall fst x2)
9    and (sync1, x2) = xt_fork x1
10   and x1 = xt_join (x, (xt_fby () sync2))
```

Listing 6: Two-stage pipeline, 1-bounded model

C. Correction considerations

1) Method-related artefacts: Notice that the cycle-accurate implementation of certain xMAStime primitives contains error conditions, implemented with the `failwith` construct of OCaml, which raises a `Failure`. The use of exceptions may be surprising in a "correct-by-construction" modeling method. However, the reason for this is simple: the asynchronous programming style based on signals means that the overall correction of the application, in the sense of Sections II-B3 and II-B4, cannot be inferred by the Lucid Synchrone compiler. For instance, the compiler cannot infer the 1-boundedness of the application, and therefore requires the programmer to provide code covering buffer overflow cases *e.g.* when a `xt_join` receives two values on one input and none on the other.

For specifications that satisfy the required correctness properties, the error handling code is unreachable. Simplifying the programming of the primitives and the generated code by removing this dead code would require a dedicated compiler, able to verify the respect of the xMAStime-specific correctness properties. Our current solution, where error handling code still exist, but only *inside* the primitives (and not in their interfaces), seems a good compromise between ease of implementation (for language prototyping) and time investment into development tools.

2) Functional semantics preservation and timing abstraction: For each xMAStime primitive, the two implementations are endochronous and correspond to the same function from streams of inputs to streams of outputs. According to II-B4, if the compiler can generate code for both simulators, and if no run-time errors occur in the cycle-accurate simulator[3] then the functional semantics is preserved. If the specification is also in-order, then the start and end dates of each instruction on each pipeline unit are *equal* in the two simulations.

Note in the definition of `xt_resource` that the duration of pipeline resources may depend on their input (it is computed by applying function `delay` to the input data). To perform timing accounting in the worst case, as needed during WCET analysis, this function must be replaced with the constant function returning the worst-case value. Data abstraction can be handled in the same way, by replacing the data types and data processing function with abstracted versions.

VI. The MIPS32 case study

We applied our modelling approach on a full-fledged architecture featuring a MIPS32 CPU core. The architecture is of Harvard type, with separate instruction and data memory banks and separate L1 caches, both with a LRU (least recently used) replacement policy. The resulting model is used to generate a cycle-accurate simulator and an instruction set simulator. Simulation is performed in isolation: the memory is precharged with the sequential program to run and with the values of initialized data, and execution proceeds without interaction with the environment.

The design takes advantage of the modularity of Lucid Synchrone. The top level node of our model is pictured in Fig. 9. Like in previous figures, named boxes correspond to instantiations of other nodes. For instance, Fig. 10 provides the implementation of the memory access node.

Our MIPS32 model uses a classical 5-stage pipeline implementation [HP07] (fetch/decode/execute/memory access/write-back). Execution is fully in-order. The interlocking protocol implemented using `xt_fby` primitives ensures 1-boundedness. The construction of the interlocking protocol is simplified on the MIPS32, due to the existence of delay slots.

The register file is placed inside the node implementing the decode stage, but it is also accessed by the write-back stage. Pipelined stages run concurrently, so there may be situations where the content of the registers to be read in the decode stage depends on the result produced by a preceding instruction that is still in the pipeline. Such a situation is called a *data hazard*. The sequential execution semantics of the program is preserved by allowing later pipeline stages to freeze earlier stages until the preceding instruction has finished updating the registers in the register file. In Fig. 9, the protocol allowing this is implemented using a single feedback line from the write-back node to the instruction decode node.

All data types and data handling routines of the model are defined in OCaml. Particularly complex pieces of this code implement instruction decoding and cache policies. Of the MIPS32 ISA we did not implement the optional floating point unit (FPU) instructions. Also in OCaml are implemented all the functions computing the duration of the various units.

[3]This can be statically ensured through 1-boundedness.

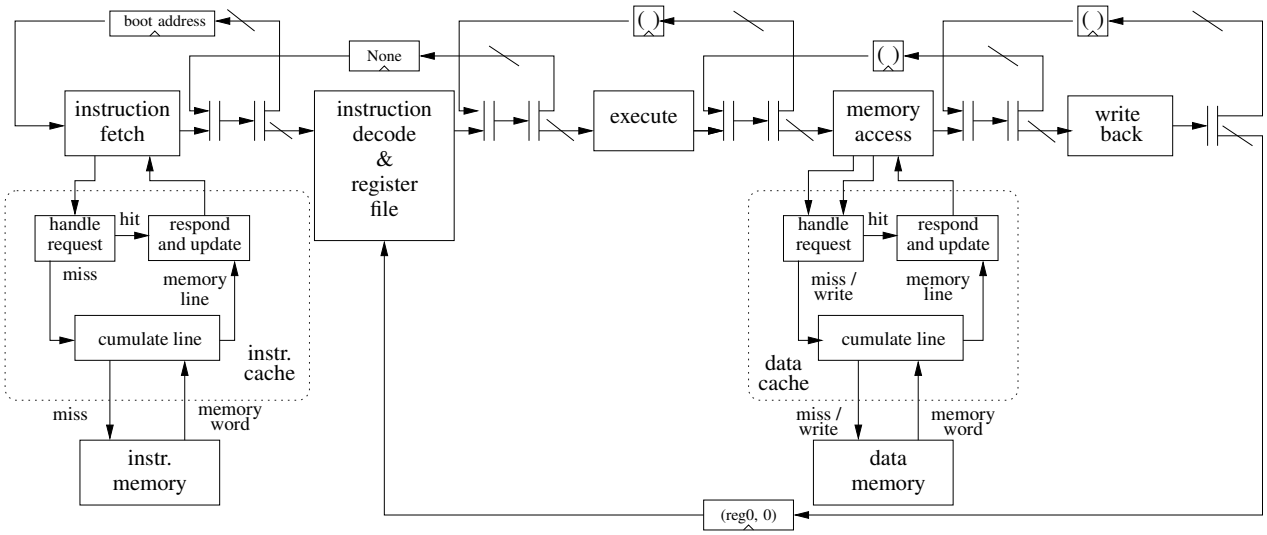


Fig. 9: Top-level view of the MIPS32 model

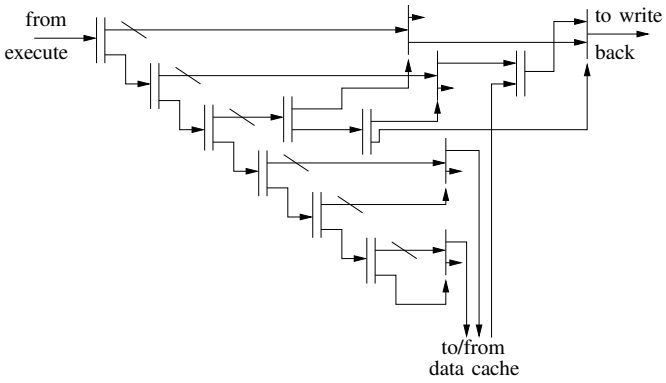


Fig. 10: Implementation of the "memory access" node

Overall, the data-handling part of the model, written in OCaml consists in around 9700 lines of code (loc). Lucid Synchrone code, including both the implementation of the xMAStime primitives (fully provided in this paper) and the nodes of the MIPS32 model comprise around 800 loc.

VII. Related work

Architecture modeling for the real-time is not a new subject ([HFC14], [CB13] provide a good overview). Traditional approaches are based on the use of Architecture Description Languages (ADLs) which allow the modeling of modern architectures, but do not have clear semantics. The semantics of the artefacts (simulators, analysis models, synthesizable RTL) obtained from these models is largely dependent on the translation tool, making it difficult to establish formal semantics preservation between artefacts.

More recently, [HFC14] gave formal semantics to an ADL tailored for timing analysis. This is done by means of constraint programming, allowing the use of constraint solvers to compute the WCET of basic blocks on complex architectures with out-of-order execution [HCFR13]. Given that the formal timed semantics is basically that of the WCET analysis model and is not operational, it is not clear how easy it is to relate it with the semantics of simulation models of the same architecture. By comparison, we target simpler architectures, with the objectives of 1) highlighting how theoretical principles allow establishing formal semantics preservation results for both functional and timing aspects and 2) embodying these principles in easy-to-use DSLs allowing experimentation.

The xMAS formalism has a formal operational semantics, but the transactional form of this semantics mainly supports the formal modeling and verification of functional properties of communication fabrics [CKO12]. The synthesis of Verilog and C++ executable code is also possible [ZL17], allowing cycle-accurate simulation, and network calculus has been used to compute performance bounds on the communication fabric models. By comparison, our xMAStime formalism gives up non-determinism and is used to represent full-fledged architectures. It explicitly covers timing aspects and allows establishing semantics preservation guarantees between two different simulation levels.

Our work is also strongly related to the large corpus of results on high-level synthesis (a good overview in provided in [TKE17]). When the high-level specification has a well-defined formal semantics that is preserved in the implementation, the equivalence between different synthesis outputs (*e.g.* synthesizable RTL which is amenable to cycle-accurate simulation and functional simulation code, as is it possible using synchronous languages [BCE$^+$03]) is acquired by construction. The originality, in our case, is that the semantics of the implementations can be transposed in the high-level model, through the change in logical simulation clock, which changes scheduling. Also, our semantics preservation guarantees cover timing aspects.

VIII. Conclusion

We proposed an approach for the construction of mutually consistent cycle-accurate and timed instruction set simulators

of full micro-architectures comprising CPU pipeline, caches, and RAM. The method is theoretically founded on desynchronization theory. It is fully tooled, and has been showcased on a full-fledged MIPS32-based architecture.

On the application side, ongoing work aims at integrating the generated simulators with the SoCLib cycle-accurate simulator and WCET analysis tool used in [PP13]. The current method is restricted to single-core in-order architectures that can be given delay-insensitive models. In future work, we will extend our approach to allow *limited* recourse to speculative execution and out-of-order execution, by safely approximating delay-sensitive timing behaviors with a delay-insensitive ones. We will also investigate the representation of timing compositional multi-core architectures.

References

[BCE+03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

[C+03] SoCLib Consortium et al. The SoCLib project: An integrated system-on-chip modelling and simulation platform. Technical report, CNRS, 2003. http://www. soclib. fr, 2003.

[CB13] F. Cassez and J. Béchennec. Timing analysis of binary programs with uppaal. In *Proceedings ACSD*, pages 41–50. IEEE, 2013.

[CKO12] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras. xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification. *IEEE Design Test of Computers*, 29(3):80–88, June 2012.

[dDvAPL14] B. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager. Time-critical computing on a single-chip massively parallel processor. In *Proceedings DATE*, March 2014.

[G.74] Kahn G. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[HCFR13] H. Herbegue, H. Cassé, M. Filali, and C. Rochange. Hardware architecture specification and constraint-based WCET computation. In *Proceedings SIES*, Porto, Portugal, 2013.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[HFC14] H. Herbegue, M. Filali, and H. Cassé. Formal Architecture Specification for Time Analysis. In *Proceedings ARCS*, Lubeck, Germany, 2014.

[HP07] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.

[HRP17] D. Hardy, B. Rouxel, and I. Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *Proceedings WCET*, Dubrovnik, Croatia, 2017.

[LDF+14] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02, 2014. https://caml.inria.fr/distrib/ocaml-4.02/ocaml-4.02-refman.pdf.

[PBCB06] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in Synchronous Systems. *Formal Methods in System Design*, 28(2):111–130, Mar 2006.

[Pou06] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: `www.lri.fr/~pouzet/lucid-synchrone`.

[PP13] D. Potop-Butucaru and I. Puaut. Integrated Worst-Case Execution Time Estimation of Multicore Applications. In *Proceedings WCET*, Paris, France, 2013.

[TKE17] R. Townsend, M.A. Kim, and S.A. Edwards. From functional programs to pipelined dataflow circuits. In *Proceedings CC*, Austin, TX, USA, 2017.

[ZL17] X. Zhao and Z. Lu. A Tool for xMAS-Based Modeling and Analysis of Communication Fabrics in Simulink. *ACM Trans. Model. Comput. Simul.*, 27(3):16:1–16:26, 2017.


```
let node buffer1 (i, c) = (o, pre_reg) where
  rec iloc = present | i (v) -> Some v | _ -> None end
  and (reg, o) = match (iloc, pre_reg) with
                 | (Some iv, Some rv) ->
                   if c then (Some iv, Some rv)
                   else failwith "buffer1 overflow"
                 | ((Some x, None) | (None, Some x)) ->
                   if c then (None,Some x)
                   else (Some x, None)
                 | (None, None) ->
                   if c then failwith "buffer1 underflow"
                   else (None, None)
    end
  and pre_reg = None fby reg
```

Listing 7: Lossless one-place buffer in xMAStime. Only used in cycle-accurate models, and must be inlined.

```
(* cycle-accurate and instruction set simulation implementations *)
(* coincide *)
let node xt_fork i = (o1, o2) where
  rec o1 = i and o2 = i
```

Listing 8: Implementation of xt_fork

```
(* cycle-accurate implementation *)
let node xt_switch (i, c) = (o1, o2) where
  rec o = xt_join (i, c)
  and present
      | o (v, c) ->
        let clock clk = c in
          do emit o1 = v when clk and emit o2 = v whenot clk done
      end
(* instruction set simulation implementation *)
let node xt_switch (i, c) = (o1, o2) where
  rec (i_val, i_tag) = i and (c_val, c_tag) = c
  and o1 = ((i when c_val), i_tag)
  and o2 = ((i whenot c_val), i_tag)
```

Listing 9: Implementation of xt_switch

```
(* cycle-accurate implementation *)
let node xt_merge (i1, i2, c) = o where
  rec (o1, v1) = buffer1 (i1, c1)
  and (o2, v2) = buffer1 (i2, c2)
  and (clk, vc) = buffer1 (c, cc)
  and cl = present | c (v) -> Some v | _ -> None end
  and c1 =  (?i1 || (v1 != None)) &&
            (((cl = Some true) && vc = None) || (vc = Some true))
  and c2 =  (?i2 || (v2 != None)) &&
            (((cl = Some false) && vc = None) || (vc = Some false))
  and clock cc = ((((cl = Some true) && (vc = None)) ||
                  (vc = Some true)) && c1) ||
                 ((((cl = Some false) && (vc = None)) ||
                  (vc = Some false)) && c2)
  and emit o = match o1 when cc, o2 when cc, clk when cc with
               | Some x, _, Some true -> x
               | _, Some x, Some false -> x
               | _ -> failwith "xt_merge"
               end
(* instruction set simulation implementation *)
let node xt_merge
  (i1, i2, c) = o
where
  rec (i1_val, i1_tag) = i1
  and (i2_val, i2_tag) = i2
  and (c_val, c_tag) = c
  and o = (o_val, o_tag)
  and o_val = merge c i1_val i2_val
  and o_tag = max i1_tag i2_tag
```

Listing 10: Implementation of xt_merge