Formal Definition for Instruction-Level Abstraction

Bo-Yuan Huang Pramod Subramanyan Hongce Zhang Aarti Gupta† Sharad Malik
Departments of Computer Science† and Electrical Engineering, Princeton University

Draft Working Document: February 28, 2017

Instruction-level abstraction (ILA) provides an interface between different components in heterogeneous systems. It is an abstraction that formally models firmware-visible behaviors of a hardware module, such as a programmable processor or application specific accelerator. Similar to the instruction-set architecture (ISA) that defines the operation of a processor, an ILA models the operation of a hardware module as a set of instructions, where each instruction represents a certain operation over the states.

Instruction-Level Abstraction

ILA is an abstraction of the hardware and serves as the interface between the hardware and other components. Therefore, only states that are visible and persistent across instructions need to be modeled. We let S be the vector of those state variables, where each element can be either a Boolean variable, a bitvector variable, or a memory variable. In the ILA of a processor, S would contain the architectural registers, flag bits, data memory, and the instruction memory. As for accelerators, S could contain memory-mapped registers, internal buffers, output ports to on-chip interconnect, and so on. In addition to state variables, an ILA also contains a vector of input variables W that models the input ports of the hardware module, such as the interrupt signals for processors and command inputs for accelerators.

In an ILA, hardware operations are abstracted into a set of instructions following the fetch/decode/execute model as processors. An ILA fetches the *opcode* from its states S and inputs W. The opcode is a bitvector that will later be decoded to determine which instruction should be issued. The fetch function indicating how is the opcode fetched is represented by $F : (S \times W) \mapsto bvec_w$ where w is the width of the opcode. Take programmable cores without interrupts for example, the opcode is fetched from the instruction memory pointed by the program counter, i.e. $F(S, W) \triangleq read(\text{IMEM}, \text{PC})$. In cases where interrupt matters, the fetch function could possibly be concatenating the interrupt signals with the value read from instruction memory.

Unlike programmable cores that always have an instruction to fetch, decode, and execute, accelerators may be event-driven and only start executing instructions when a certain trigger occurs. We let $V : (S \times W) \mapsto \mathbb{B}$ be the valid function that indicates when an ILA can execute. For example, if an accelerator executes instructions only when the signals AddrInRange and StateIdle are asserted, then $V(S, W) \triangleq \text{AddrInRange} \wedge \text{StateIdle}$. For programmable cores, there is always an instruction and the valid function is always true.

Each instruction is associated with a decode function $\delta_i : bvec_w \mapsto \mathbb{B}$ that takes the fetched opcode as input and tells whether the instruction is *issued*. Take 8051 microcontroller as an

example, the opcode of the *JZ* instruction is `0x60`, therefore $\delta_{JZ}(\text{opcode}) \triangleq \text{opcode} = \texttt{0x60}$. The set of all decode functions in the ILA is denoted as $D = \{\delta_i \mid 0 \leq i \leq C\}$, where C is the number of instructions.

Besides the decode function, every instruction is also associated with a next state function $N_i : (S \times W) \mapsto S$ to represent the state updates when an instruction is executed. For example, the *INC* instruction (opcode `0x4`) in the 8051 microcontroller increments the accumulator, therefore the next state function of the accumulator is $N_4[\text{ACC}] = ACC + 1$. Note that the operation defined in N_i is atomic and an instruction can be executed only if it has been issued. The set of all next state functions in the ILA is denoted as $N = \{N_i \mid 0 \leq i \leq C\}$, where C is the number of instructions.

To summarize, ILA provides a uniform interface between hardware components in heterogeneous systems by formally abstracting firmware-visible behaviors of hardware into sets of instructions under the fetch/decode/execute model. An ILA is defined as follow:

$$
\begin{aligned}
\text{A} \quad &= \quad \langle S, W, F, V, D, N \rangle, \text{ where} \\
&\quad S \quad \text{ is the vector of state variables,} \\
&\quad W \quad \text{ is the vector of input variables,} \\
&\quad V \quad : (S \times W) \mapsto \mathbb{B} \text{ is the valid function,} \\
&\quad F \quad : (S \times W) \mapsto bvec_w \text{ is the fetch function,} \\
&\quad D \quad = \{\delta_i : bvec_w \mapsto \mathbb{B}\} \text{ is the set of decode functions, and} \\
&\quad N \quad = \{N_i : (S \times W) \mapsto S\} \text{ is the set of next state functions.}
\end{aligned}
$$

Hierarchical ILA

While ILA serves as the interface between different components by abstracting hardware behaviors into sets of instructions, there are cases where we would like to model the hardware in different levels of abstraction. For example, in the specification of Intel x86 architecture, the string copy instruction `REP STOS` involves a sequence of bytes moves. The instruction causes the associated `STOS` be repeated until the count in register `ECX` is decremented to 0 [1]. Such operation should not be represented as one single state update if we want to model the behavior more precisely. Another example is the `START_ENCRYPT` instruction that starts the encryption operation of an AES accelerator, of which the implementation contains message loading, encryption computing, and output storing.

In hierarchical ILA, we allow an ILA to contain *child-ILAs* for complex instructions to help a manage different level of abstraction. Similar to an ILA, a child-ILA follows the fetch/decode/execute model and contains a set of *child-instructions* to help represent complex instructions. Take `START_ENCRYPT` for example, the operation can now be represented by a child-ILA containing one child-instruction for message loading, one for doing encryption, and another for storing the results. The ILA containing the child-ILA is called the *parent-ILA*.

To distinguish between specified behavior and possible implementation, every child-ILA is associated with an auxiliary variable $u \in \mathbb{B}$. $u = 1$ means that the behavior modeled by the child-instructions are required in the specification. We call such child-instructions *sub-instructions*. On the other hand, $u = 0$ means that the child-instructions are representing one possible implementation of the complex instruction, and we call them *micro-instructions*.

A child-ILA is defined similarly as an ILA. It has its own state variables, valid function, fetch function, decode functions, and next state functions denoted as S^μ, V^μ, F^μ, D^μ, and N^μ respectively. S^μ can contain shared states in S of its parent ILA, and every update to the shared states are synchronized. The valid function $V^\mu : S^\mu \mapsto \mathbb{B}$ indicates when the child-ILA can execute. The fetch function $F^\mu : S^\mu \mapsto bvec_l$, where l is the width of the opcode, models how the opcode is fetched from the states. The decode function D^μ is defined exactly the same as an ILA to tell if any child-instruction is issued. State update of each child-instruction is modeled by next state functions $N^\mu_j : S^\mu \mapsto S^\mu$, and N^μ is the set of next state functions for all child-instructions.

With the extension of hierarchical ILA, ILA can now model the hardware at different levels of abstraction. A hierarchical ILA is defined as $A = \langle S, W, V, F, D, N, L\rangle$, where $L = \{(u^{\mu_p}, A^{\mu_p}), \dots\}$ is the set of child-ILAs. u^{μ_p} is the auxiliary variable showing if A^{μ_p} is composed of sub-instructions or micro-instructions. Similarly, child-ILAs can also contain other child-ILAs and are defined exactly the same, except that micro-ILAs can only contain micro-ILAs.

Micro-program and sub-program

Once a child-ILA starts executing, it then steps through a sequence of child-instructions where the sequencing is implicitly determined by the state update itself. This can be understood as a *child-program* even though the control location is not defined on a specific variable as usual. Take the string copy instruction for example, the child-program representing the instruction contains a loop whose loop body is the child-instruction that moves a byte at a time. We call a child-program *sub-program* if it is composed of sub-instructions, and call it *micro-program* if it is composed of micro-instructions.

A child-ILA A^μ starts executing when the valid function V^μ changes from 0 to 1, indicating that there is one child-instruction being issued. This could happen when a certain instruction in its parent-ILA, the parent instruction, is executed and modifies the shared states so that V^μ becomes true. For example, the `START_ENCRYPT` instruction of the AES accelerator updates the shared state *aes_state*, which then starts the child-ILA with the child-instruction for message loading being issued. It is also possible that V^μ becomes true at the same time as the parent instruction is issued. Take string copy for example, the child-instruction for the first byte move is issued at the same time as the string copy instruction is issued. In such cases, the execution of the parent instruction is trivial.

Execution Semantics

The ILA and its child-ILAs execute asynchronously and communicate through shared state variables. Updates of the shared states are synchronized. When an instruction is executed, the shared states in child-ILAs will also be updated; when a child-instructions is executed, the shared states in the parent ILA will also be updated.

The execution of an instruction in the ILA is defined by rule (1).

$$\frac{V(S,W) \quad \delta_i(F(S,W)) \quad S' = N_i(S,W)}{S \rightsquigarrow S'} \tag{1}$$

The rule says that an ILA can transition from state S to S' if the following conditions are satisfied:

1. An instruction is being triggered: $V(S, W) = 1$.
2. The i-th instruction type is issued: $\delta_i(F(S, W)) = 1$.
3. State update of each variable in the vector S' is as defined by $N_i(S, W)$.

The execution of a child-instruction in child-ILA is defined by rule (2).

$$\frac{V^\mu(S^\mu) \quad \delta_i^\mu(F^\mu(S^\mu)) \quad S'^\mu = N_i^\mu(S^\mu)}{S^\mu \rightsquigarrow S'^\mu} \tag{2}$$

The rule says that a child-ILA can transition from state S^μ to S'^μ if the following conditions are satisfied:

1. The child-ILA is activate: $V(S^\mu) = 1$.
2. The j-th child-instruction is issued: $\delta_j^\mu(S^\mu) = 1$.
3. State update of each variable in the vector S'^μ is as defined by $N_j^\mu(S^\mu)$.

There are two implications of these two rules. First, each state update defined in N_i or N_j^μ is atomic. Second, the decode function only indicates whether an instruction is issued, it does not say anything about execution.

The synchronization of the shared states is defined by rules (3) and (4).

$$\frac{S \rightsquigarrow S'}{S^\mu \overset{S'}{\rightsquigarrow} S'^\mu} \tag{3}$$

$$\frac{S^\mu \rightsquigarrow S'^\mu}{S \overset{S'^\mu}{\rightsquigarrow} S'} \tag{4}$$

The notation $S \overset{S'^\mu}{\rightsquigarrow} S'$ means that states in the ILA change from S to S' with all the shared states between S and S^μ updates to the corresponding value in S'^μ while other states in S remain unchanged. The notation $S^\mu \overset{S'}{\rightsquigarrow} S'^\mu$ is defined in a similar way.

Instruction Concurrency

ILA not only serves as the interface between hardware modules but also allows us to model the concurrency behavior across instructions. Modeling such concurrency is important and necessary in verifying heterogeneous systems due to the parallel interaction between hardware components. For example, consider a scenario where a firmware utilizes the AES accelerator to encrypt a block of message. It first uses the string copy instruction to move the message into a certain range of memory; then uses memory-mapped I/O (MMIO) to trigger the encryption operation, which depends on the message; finally, it reads back the result after checking the status of the accelerator. Here both the string copy and the encryption are non-blocking instructions. They allow other instructions to be issued and executed even if

they are not finished yet. Precisely modeling the concurrent interaction between non-blocking instructions is necessary for correct verification.

With the help of hierarchical ILA, non-blocking operations of complex instructions can be represented by child-instructions. The execution semantics of the ILA can model the scenario which issues a new instruction even when other child-ILAs are still executing. Note that the signals on input ports, which are usually used to trigger instructions, are not required to remain unchanged during the execution of child-ILAs. The state updates of these instructions and child-instructions are atomic, as defined by the next state functions, and can be interleaved with others. Therefore we can then utilize the works on instruction interleaving to model and verify the concurrency interaction.

Note that ILA is not restricted to representing non-blocking operations of complex instructions. Other cases can have corresponding abstractions with different decode functions and next state functions depending on the underlying hardware behavior. For example, a simple processor, where new instructions cannot be issued until previous instructions are complete, can be modeled by ensuring the program counter be updated at the last child-instruction.

References

[1] Intel Corporation. Intel Architecture Software Developer's Manual. Volume 2: Instruction Set Reference. `http://download.intel.com/design/intarch/manuals/24319101.pdf`, 1999.