M. Baclet, R. Pacalet and A. Petit

Register Transfer Level Simulation

Research Report LSV–04–10, May 2004

Laboratoire Spécification et Vérification

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

http://www.lsv.ens–cachan.fr/Publis/
Research Report LSV–04–10, Lab. Spécification et Vérification, CNRS & ENS de Cachan, France, May 2004

Register Transfer Level Simulation

Manuel BACLET[1], Renaud PACALET[2], and Antoine PETIT[1]

[1] LSV - UMR 8643 - CNRS/ENS de Cachan
61, Av. du Président Wilson - 94235 Cachan Cedex - France
{baclet,petit}@lsv.ens-cachan.fr
phone number:+33 (0) 1 4740 7542

[2] GET/ENST/COMELEC/LabSoC
Institut Eurecom BP 193
2229, Route des Cretes - 06904 Sophia Antipolis Cedex - France
pacalet@enst.fr

Abstract. As the complexity of designed Systems on Chip (SoC) increases, due to the ever growing number of transistors that can be integrated on a chip, methods have been developed to validate a design before it is actually manufactured. Currently, the method that is the most used is based on software simulation. Hence, finding methods to easily provide accurate and fast simulation would be very useful. In this paper, we propose an automaton-based formalism that allows to model the sequences of input/output of an on-chip component, abstracting the labels and the data values. This formalism can be used, on one hand, to decide whether two components, which are supposed to implement the same function, have the same input/output behaviors so that one can be replaced by the other, and on the other hand, to obtain automatically, from a high level description of the chip, a program that simulates the component accurately (at the register transfer level). The paper is throughout illustrated with the detailed example of IDCT, a signal processing component.

Keywords: SoC, Simulation, FSM, Automata

Introduction

Systems on Chips A system on chip is an on-chip assembly of hardware components which can be processors, memories, digital signal processors, buses, multiple interfaces, etc. Current systems can integrate more than ten millions of transistors on a single silicon substrate. The complexity of such systems confronts designers with difficulties:

- most tools and methods used to design and validate SoCs are the same tools and methods that were used some years ago for much simpler Application Specific Integrated Circuits (ASICs),
- an increasing designing time in which a large part is devoted to verification/validation.

In order to overcome these problems, designers try to reuse complex components (Intellectual Property) and to apply formal methods of verification to ensure that the component will work properly. However, the validating method is mostly the test which relies on either an expensive hardware emulator or on a program that simulates it.

Simulation It is the most commonly used method to validate a complete on-chip system, since formal methods (such as model-checking) are confronted by the state explosion problem. Systems are often simulated during millions or billions of cycles. Efficiency of the simulation is then a critical parameter and the ideal model of each subcomponent of the system is both bit and cycle precise and must be implementable in a very efficient manner. Unfortunately, those criteria are most of the time conflicting: as precise models are more detailed, they are commonly slower. In order to accelerate the simulations, a possible approach is to separate the processing and the control so that a part of the code computes the result by a classical software approach (without taking into account the internal structure of the component) and another part is devoted to the control of the input/output of the component.

In the sequel, we describe this approach on an IDCT component which is an hardware accelerator for data flow oriented signal processing algorithms. Its complexity is about the same as other signal processing components such as Inverse Quantization, Motion Compensation, Huffman decoders components or communication dedicated components such as Fast Fourier Transform, Digital filters, Viterbi or Turbo Decoders components.
In the IDCT case, it is easy to obtain a precise model by describing each of the seven subcomponents and their assembling. Designers confronted with time limits for simulation can try to create an abstract model of the component. This is done in subsection 4.2 with the pipe-lined model. However, this method is error-prone: we proved that the detailed model and the hand-made abstract model are not equivalent. Such mistakes can have important consequences, for example from an industrial point of view, leading to longer design time and incorrect prototyping. This shows the necessity to use formal methods to prove the equivalence of two models and the need of an automated method for building the abstract model.

Outline In this paper, we propose a formalism, the *io-automata*, in order to model the sequences of input/output of a component. More precisely, we are only interested in events like «a sample enters the component», «no sample comes out», etc. This is a high level abstraction where not only real values of the data are forgotten but also the labels of the input/output events are omitted. It leads to automata that keep a reasonable size and that model precisely the input/output sequences taking into account the internal state of the component. Indeed, we do not only model the interface of a component but also the way it is managed.
In section 1, we give a detailed description of the IDCT component that will be used throughout the paper as an example. The important point is that it is a real component and not a simplified example. Then, we describe the io-automata formalism used to model the input/output sequences of a component. This formalism is a special case of finite state automata but is different from the Input/Output automata proposed by N. A. Lynch and M. R. Tuttle in [LT89]. We define the product of io-automata in such a way that if $\mathcal{A}$ is the model of component A and $\mathcal{B}$ the model of component B, the product of $\mathcal{A}$ and $\mathcal{B}$ is the model of the component obtained by connecting A and B. This product is some sort of wreath product on finite automata, hence it differs from the classical product as defined in [AN82]. This is detailed in section 2. Section 3 is devoted to the applications of io-automata. We detail how our formalism is used to compare components, provide automatically accurate simulations and evaluate interesting parameters of a component. In section 4, we explain how IDCT is hand-modeled by means of io-automata. An abstract model of IDCT called pipelined IDCT is described and we prove that this abstraction is not valid, showing that such hand-made abstractions are hazardous. Finally, we study the io-automata associated to IDCT to show that the simulation method we propose is efficient.

1 An IDCT component

1.1 Context

The Discrete Cosine Transform [RY90,MJ91] is a processing commonly used in signal compression applications. Its bidimensional version is the core of most image compression algorithms (JPEG, MPEG, H26x, etc.). The computation power needed is such that most of video compression systems use dedicated hardware accelerators to compute it. The hardware component we study here computes the inverse transformation, which is used for image decoding. This component can be found in video decompression systems, like DVD players, satellite or cable set-top boxes.

1.2 Interface

The IDCT component is a synchronous device: all operations (inputs, outputs,etc.) are synchronized with the rising edge of a unique clock. We recall that a clock is a periodic signal with crenel shape. Rising edges are times where the signal goes from low level to high level.
IDCT has an input interface from which it can receive samples at the maximum rate of one per clock cycle. A simple hand-shake protocol allows it to accept or refuse a new sample,

depending on its internal state. Similarly, IDCT has an output interface which produces samples of the resulting computation at the maximum rate of one per cycle. A simple hand-shake protocol is used so that the target component can accept or refuse a new sample. In such a system, when a sample is refused by the target system, the sender keeps trying to send it, until it is accepted by the target. Hence, no data can be lost but a deadlock is possible if a component stops accepting new samples, indefinitely.
The function computed by IDCT is a double matrix product: when it has received 64 samples, corresponding to a 8×8-matrix, X, the IDCT component will produce on its output interface, after some latency cycles, 64 samples corresponding to the product $A.X.^tA$, where A is some constant 8×8-matrix.

1.3 Internal architecture

The internal architecture of IDCT is the linear connection of seven subcomponents which communicate by mean of the same hand-shake protocol used for input and output interfaces. IDCT is represented in Fig. 1. Each component receives samples from the previous one (or from the environment for INR_1) and produces samples intended for the following component (or for the environment in the case of $POST_2$) under the control of the hand-shake protocol. The arrow labels indicate the width of each interconnection, i.e. the number of samples that can be transmitted per clock period. For example, in case of a data exchange between $ACCU_1$ and $POST_1$, eight samples are transferred at a time from $ACCU_1$ to $POST_1$.

There are four kinds of components involved in IDCT, each of them having characteristic values of input width, output width, storage capacity (maximum number of samples the component can accept when its output is frozen), minimum latency (minimum number of clock periods between the input of a sample i and the output of a sample o that depends on i) and of I/O dependency ($o[a] = f(i[a0], i[a1], ..., i[an])$, the dependency relation between an output sample and one or more input samples). Since we are only interested in input/output sequences, and not in the function implemented in the components, we do not describe the treatment realized by each component.

Fig. 1. An IDCT component

Now, we describe the different subcomponents of IDCT.

INR inputs one sample per clock period and outputs one sample per clock period. Its storage capacity is 8 samples, that is, once it received 8 input samples it cannot accept any new input sample unless it is allowed to produce an output sample at the same time. The I/O dependency relation shows that the input flow is split in sequences of 8 consecutive samples $(i[8 * n], ...i[8 * n + 7])$. The output flow is split in sequences of 8 consecutive samples too $(o[8 * n], ...o[8 * n + 7])$. Any output sample in a sequence depends on each of the 8 samples in the corresponding input sequence. The minimum latency is 1, that is, if INR inputs a sample i on rising edge of clock $\#N$, it is sometimes possible that an output sample o depending on i be sampled by the next component on the next rising edge of clock $\#(N + 1)$. Considering the I/O dependency and the minimum latency of INR, the minimum latency from $i[8 * n]$ to $o[8 * n]$ is 8, the minimum latency from $i[8 * n + 1]$ to $o[8 * n]$ is 7, etc. and the minimum latency from $i[8 * n + 7]$ to $o[8 * n]$ is 1.

Input width	1 sample
Output width	1 sample
Storage capacity	8 samples
I/O dependency	$o[8*n+k] = f(i[8*n], ...i[8*n+7])_{0 \leqslant k \leqslant 7}$
Minimum latency	1 cycle

The other components are much simpler:

ACCU inputs one sample per clock period and outputs 8 samples per clock period. Its storage capacity is 8 samples.

Input width	1 sample
Output width	8 samples
Storage capacity	8 samples
I/O dependency	$o[8*n+k] = f(i[8*n], ...i[8*n+7])_{0 \leqslant k \leqslant 7}$
Minimum latency	1 cycle

POST inputs 8 sample per clock period and outputs 1 samples per clock period. Its storage capacity is 8 samples.

Input width	8 samples
Output width	1 sample
Storage capacity	8 samples
I/O dependency	$o[8*n+k] = f(i[8*n], ...i[8*n+7])_{0 \leqslant k \leqslant 7}$
Minimum latency	1 cycle

TRSP is a component that computes the matrix transposition. It inputs one sample per clock period and outputs one sample per clock period. Its storage capacity is 64 samples. An input is possible if less than sixty four samples are stored in the component and an output is possible if the next needed sample is currently stored in the component.

Input width	1 sample
Output width	1 sample
Storage capacity	64 samples
I/O dependency	$o[64*n+8*k+l] = f(i[64*n+8*l+k])_{0 \leqslant k,l \leqslant 7}$
Minimum latency	1 cycle

1.4 2-IDCT

In order to explain how the modeling was computed, we will need a simplified version of the component, which is obtained by setting all parameters with value 8 (resp. 64) to value 2 (resp. 4). This component will be denoted by 2-IDCT.

We now describe the formalism that we chose to model the sequences of input/output an on-chip component.

2 IO Automata

We introduce here an automaton-based formalism to represent the sequences of input/output of a component. Those automata are essentially classical finite state automata over a particular alphabet. The important point is that we define a wreath automata product that corresponds to the linking of two components.

2.1 Alphabet

In the sequel, we consider the finite alphabet $\Sigma = \{(i,o),(i,\overline{o}),(\overline{i},\overline{o}),(\overline{i},o)\}$. At each rising edge of the clock, a letter must be read. The meaning of the alphabet is that the letter:
- (i,o) is read when an input and an output occur simultaneously,
- $(i,\overline{o})$ is read when an input occurs but no output occurs,
- $(\overline{i},o)$ is read when an input occurs but no output occurs and
- $(\overline{i},\overline{o})$ is read if no output and no input occur.

A finite sequence of letters is a *word* over Σ and the set of all words is denoted by Σ^*.

2.2 IO-Automata

Definition 1. *An io-automaton is a triple $\mathcal{A} = (Q, q_0, T)$ where*
- *Q is a finite set of states,*
- *q_0 is the initial state,*
- *T is a function from $Q \times \Sigma$ into $Q \cup \{\perp\}$, the transition function.*

If $T(q,l) = \perp$, no transition labeled by l can be fired from state q. ($\perp$ can be viewed as a sink state and in the following, those transitions will not be represented in figures.)
A word $u = (l_1, l'_1)\dots(l_n, l'_n)$ is *accepted* by the automaton if there exists $q_1 \dots q_n \in Q$ such that:
- $T(q_0, (l_1, l'_1)) = q_1$,
- $\forall j > 1, T(q_{j-1}, (l_j, l'_j)) = q_j$

This will be denoted by:

$$q_0 \xrightarrow{(l_1,l'_1)} q_1 \xrightarrow{(l_2,l'_2)} q_2 \dots q_{n-1} \xrightarrow{(l_n,l'_n)} q_n.$$

If it is not the case, there exists $1 \leqslant k \leqslant n$ and a sequence of states such that:

$$q_0 \xrightarrow{(l_1,l'_1)} q_1 \xrightarrow{(l_2,l'_2)} q_2 \dots q_{k-1} \xrightarrow{(l_k,l'_k)} \perp .$$

Such a path through the automaton is called the *run* of the automaton over the word u.
The set of all words accepted by $\mathcal{A}$ will be denoted by $\mathcal{L}(\mathcal{A})$.

Remark 1. An io-automaton $\mathcal{A} = (Q_{\mathcal{A}}, q_{0\,\mathcal{A}}, T_{\mathcal{A}})$ can be viewed as a classical deterministic complete finite state automaton with:
- $Q \cup \{\perp\}$ as the set of states,
- Q as the set of final state.

2.3 An example

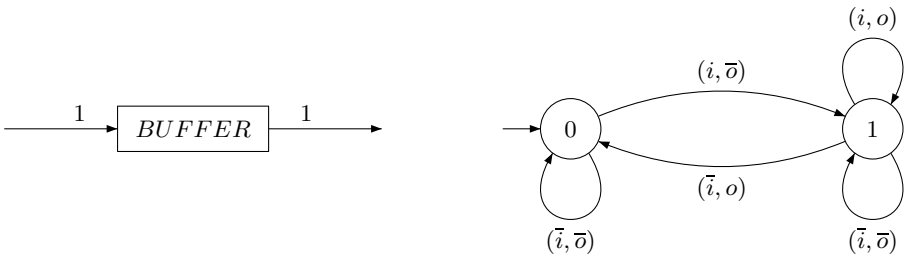


Fig. 2. A one-data buffer and its io-automaton

A one-data buffer (Fig. 2) is a component that inputs one sample per clock cycle and outputs one sample per clock cycle. Its storage capacity is one sample. When it is empty (i.e. the component does not store any sample), one input is possible but no output can happen. When one sample is stored in the component, an output is possible and in this case, an input is also possible.

Input width	1 sample
Output width	1 sample
Storage capacity	1 sample
I/O dependency	$o[n] = f(i[n])$
Minimum latency	1 cycle

The associated io-automaton is represented in Fig. 2.
The state named 0 corresponds to the empty buffer and the state named 1 to the buffer with one sample stored. When in state 0, no output is possible, so the only enabled transitions are $(i, \overline{o})$ and $(\overline{i}, \overline{o})$, leading respectively to the state 1 and 0. When in state 1, three transitions are possible, namely (i, o), $(\overline{i}, \overline{o})$ and $(\overline{i}, o)$ leading respectively to states 1, 1 and 0. Transition $(i, \overline{o})$ is not allowed since the component can only store one sample.

2.4 IO-Automata Product

We now present an automata product that allows a modular description of a complex component: we can model each subcomponent with an automaton and obtain the model of the complete component by computing the product of all subcomponent automata. The product we define here is not the classical automata product [AN82], hence leading to a formalism that is different from the input/output automata introduced by [LT89].

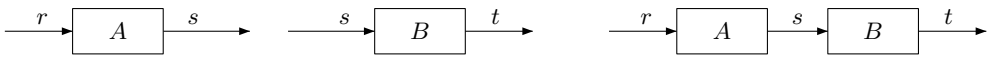


Fig. 3. Components A and B and their connection

Let A and B be the components of Fig. 3. We suppose that the output width of A is equal to the input width of B, so that those components can be connected like in Fig. 3.
Let $\mathcal{A} = (Q_{\mathcal{A}}, q_{0\mathcal{A}}, T_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, q_{0\mathcal{B}}, T_{\mathcal{B}})$ be automata that model respectively the components A and B. We want to define the product, $\mathcal{A} \times \mathcal{B}$, of $\mathcal{A}$ and $\mathcal{B}$ so that it models the component obtained by linking A to B.
We want to synchronize outputs of A with inputs of B so that when a data transfer between A and B is possible then this transfer must happen.
This leads to the following definition of the product of $\mathcal{A}$ and $\mathcal{B}$, over the same alphabet Σ:

$$\mathcal{A} \times \mathcal{B} = (Q, q_0, T)$$

where:
- $Q = Q_{\mathcal{A}} \times Q_{\mathcal{B}}$,
- $q_0 = (q_{0\mathcal{A}}, q_{0\mathcal{B}})$,
- T is defined in the following way:
  Let (q_1, q_2) be in Q and (l_1, l_2) in Σ. If there exist $q_1' \in Q_1$ and $q_2' \in Q_2$ such that

$$T_1(q_1, (l_1, o)) = q_1' \text{ and } T_2(q_2, (i, l_2)) = q_2',$$

  we set:

$$T((q_1, q_2), (l_1, l_2)) = (q_1', q_2'),$$

  else, if there exist $q_1' \in Q_1$ and $q_2' \in Q_2$ such that

$$T_1(q_1, (l_1, \overline{o})) = q_1' \text{ and } T_2(q_2, (\overline{i}, l_2)) = q_2',$$

we set:

$$T((q_1, q_2), (l_1, l_2)) = (q_1', q_2'),$$

else, we set:

$$T((q_1, q_2), (l_1, l_2)) = \perp .$$

2.5 Example

The connection of two one-data buffers like in Fig. 2 is modeled by the io-automaton of Fig. 4.

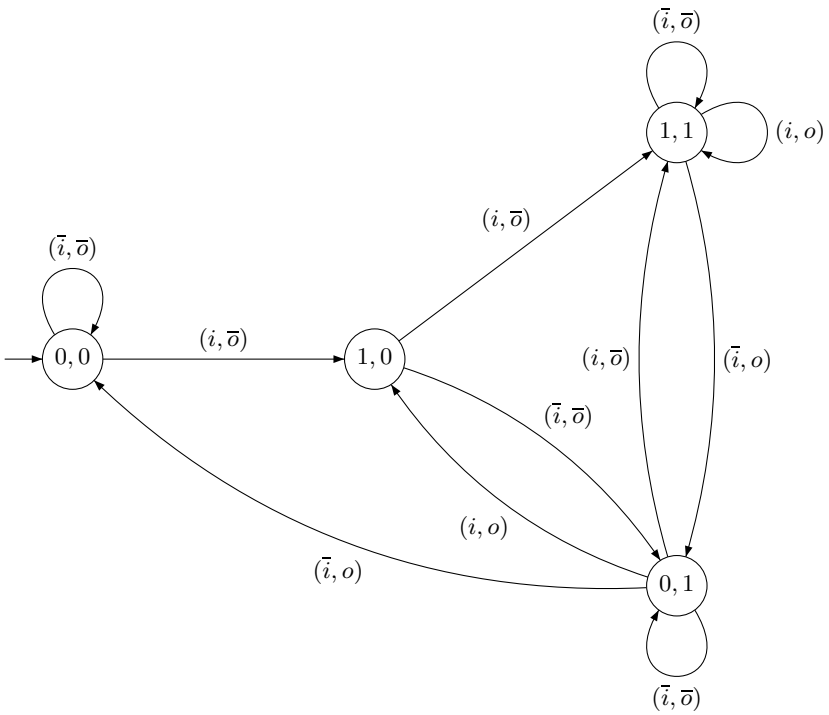


Fig. 4. Io-automaton of two one-data buffers

We now study the basic properties of the previously defined product.

2.6 Product properties

In this subsection, we give basic properties, that will be used implicitly in the sequel, of the io-automata product showing that our formalism is sound.

Definition 2. *Let $\mathcal{A}$ and $\mathcal{B}$ be two io-automata. $\mathcal{A}$ and $\mathcal{B}$ are isomorphic, which will be denoted by $\mathcal{A} \simeq \mathcal{B}$, if they are identical, up to the names of the states.*

Let $\mathcal{A}$, $\mathcal{A}'$, $\mathcal{B}$ and $\mathcal{B}'$ be io-automata. If $\mathcal{A}$ and $\mathcal{A}'$ are isomorphic and $\mathcal{B}$ and $\mathcal{B}'$ are isomorphic, then it is clear that:

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \text{ and } (\mathcal{A} \times \mathcal{B}) \simeq (\mathcal{A}' \times \mathcal{B}')$$

Proposition 1. *i. (Associativity)* $(\mathcal{A} \times \mathcal{B}) \times \mathcal{C} \simeq \mathcal{A} \times (\mathcal{B} \times \mathcal{C})$

ii. (Compatibility)

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}') \textit{ and } \mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}') \Rightarrow \mathcal{L}(\mathcal{A} \times \mathcal{B}) = \mathcal{L}(\mathcal{A}' \times \mathcal{B}')$$

We give the proof of the proposition:
Let $\mathcal{A} = (Q_\mathcal{A}, i_\mathcal{A}, T_\mathcal{A})$, $\mathcal{B} = (Q_\mathcal{B}, i_\mathcal{B}, T_\mathcal{B})$ and $\mathcal{C} = (Q_\mathcal{C}, i_\mathcal{C}, T_\mathcal{C})$ be io-automata.
Let $q_\mathcal{A}$ be in $Q_\mathcal{A}$, $q_\mathcal{B}$ in $Q_\mathcal{B}$, $q_\mathcal{C}$ in $Q_\mathcal{C}$ and $(l, l') \in \Sigma$.
We suppose that $T_{(\mathcal{A}\times\mathcal{B})\times\mathcal{C}}(((q_\mathcal{A}, q_\mathcal{B}), q_\mathcal{C}), (l, l')) = ((q'_\mathcal{A}, q'_\mathcal{B}), q'_\mathcal{C})$ and we prove that

$$T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C})) :$$

- If $T_{\mathcal{A}\times\mathcal{B}}((q_\mathcal{A}, q_\mathcal{B}), (l, o)) = (q'_\mathcal{A}, q'_\mathcal{B})$ and $T_\mathcal{C}(q_\mathcal{C}, (i, l')) = q'_\mathcal{C}$
  - if $T_\mathcal{A}(q_\mathcal{A}, (l, o)) = q'_\mathcal{A}$ and $T_\mathcal{B}(q_\mathcal{B}, (i, o)) = q'_\mathcal{B}$, we have

    $$T_{(\mathcal{B}\times\mathcal{C})}((q_\mathcal{B}, q_\mathcal{C}), (i, l')) = (q'_\mathcal{B}, q'_\mathcal{C}),$$

    and

    $$T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C})).$$

  - else, $T_\mathcal{A}(q_\mathcal{A}, (l, \overline{o})) = q'_A$ and $T_\mathcal{B}(q_\mathcal{B}, (\overline{i}, o)) = q'_\mathcal{B}$, we have

    $$T_{(\mathcal{B}\times\mathcal{C})}((q_\mathcal{B}, q_\mathcal{C}), (\overline{i}, l')) = (q'_\mathcal{B}, q'_\mathcal{C}),$$

    and

    $$T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C})).$$

- Else, $T_{\mathcal{A}\times\mathcal{B}}((q_\mathcal{A}, q_\mathcal{B}), (l, \overline{o}) = (q'_\mathcal{A}, q'_\mathcal{B})$ and $T_\mathcal{C}(q_\mathcal{C}, (\overline{i}, l')) = q'_\mathcal{C}$
  - if $T_\mathcal{A}(q_\mathcal{A}, (l, o)) = q'_\mathcal{A}$ and $T_\mathcal{B}(q_\mathcal{B}, (i, \overline{o})) = q'_\mathcal{B}$, we have

    $$T_{(\mathcal{B}\times\mathcal{C})}((q_\mathcal{B}, q_\mathcal{C}), (i, l')) = (q'_\mathcal{B}, q'_\mathcal{C}),$$

    and

    $$T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C})).$$

  - else, $T_\mathcal{A}(q_\mathcal{A}, (l, \overline{o})) = q'_A$ and $T_\mathcal{B}(q_\mathcal{B}, (\overline{i}, \overline{o})) = q'_\mathcal{B}$, we have

    $$T_{(\mathcal{B}\times\mathcal{C})}((q_\mathcal{B}, q_\mathcal{C}), (\overline{i}, l')) = (q'_\mathcal{B}, q'_\mathcal{C}),$$

    and

    $$T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C})).$$

We show in a similar way that if $T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C}))$, then we have

$$T_{(\mathcal{A}\times\mathcal{B})\times\mathcal{C}}(((q_\mathcal{A}, q_\mathcal{B}), q_\mathcal{C}), (l, l')) = ((q'_\mathcal{A}, q'_\mathcal{B}), q'_\mathcal{C}).$$

And this shows that:

$$\begin{aligned}\forall (q_\mathcal{A}, q_\mathcal{B}, q_\mathcal{C}) \in (Q_\mathcal{A} \times Q_\mathcal{B} \times Q_\mathcal{C}), \forall (q'_\mathcal{A}, q'_\mathcal{B}, q'_\mathcal{C}) \in (Q_\mathcal{A} \times Q_\mathcal{B} \times Q_\mathcal{C}), \forall (l, l') \in \Sigma,\\ \Big[T_{(\mathcal{A}\times\mathcal{B})\times\mathcal{C}}(((q_\mathcal{A}, q_\mathcal{B}), q_\mathcal{C}), (l, l')) = ((q'_\mathcal{A}, q'_\mathcal{B}), q'_\mathcal{C})\\ \Leftrightarrow T_{\mathcal{A}\times(\mathcal{B}\times\mathcal{C})}((q_\mathcal{A}, (q_\mathcal{B}, q_\mathcal{C})), (l, l')) = (q'_\mathcal{A}, (q'_\mathcal{B}, q'_\mathcal{C}))\Big].\end{aligned}$$

Hence, we have:

$$(\mathcal{A} \times \mathcal{B}) \times \mathcal{C} \simeq \mathcal{A} \times (\mathcal{B} \times \mathcal{C})$$

□

We now detail the proof of property ii.:
Let $u = (l_1, l'_1)...(l_n, l'_n) \in \Sigma^*$ be a word accepted by $\mathcal{A} \times \mathcal{B}$. There exists a sequence of states $(q_{\mathcal{A}_0}, q_{\mathcal{B}_0})...(q_{\mathcal{A}_n}, q_{\mathcal{B}_n})$ such that:

$$(q_{\mathcal{A}_0}, q_{\mathcal{B}_0}) \xrightarrow{(l_1,l'_1)} (q_{\mathcal{A}_1}, q_{\mathcal{B}_1}) \xrightarrow{(l_2,l'_2)} (q_{\mathcal{A}_2}, q_{\mathcal{B}_2}) \dots (q_{\mathcal{A}_{n-1}}, q_{\mathcal{B}_{n-1}}) \xrightarrow{(l_n,l'_n)} (q_{\mathcal{A}_n}, q_{\mathcal{B}_n}),$$

with $(q_{\mathcal{A}_0}, q_{\mathcal{B}_0}) = (q_{0_\mathcal{A}}, q_{0_\mathcal{B}})$.
For all i, the transition labeled (l_i, l'_i) from state $(q_{\mathcal{A}_i}, q_{\mathcal{B}_i})$ is either a *transfer transition* (i.e. that comes from a synchronization between letters (l_i, o) and (i, l'_i)) or a *non-transfer transition* (i.e. that comes from a synchronization between letters $(l_i, \overline{o})$ and $(\overline{i}, l'_i)$). Using this, we define

two words $u_{\mathcal{A}}$ and $u_{\mathcal{B}}$: $u_{\mathcal{A}} = (l_i, o_i)_{1 \leqslant i \leqslant n}$ with $o_i = o$ if the transition labeled (l_i, l'_i) from state $(q_{\mathcal{A}_i}, q_{\mathcal{B}_i})$ is a transfer transition and $o_i = \overline{o}$, otherwise and $u_{\mathcal{B}} = (i_i, l'_i)_{1 \leqslant i \leqslant n}$ with $i_i = i$ if the transition labeled (l_i, l'_i) from state $(q_{\mathcal{A}_i}, q_{\mathcal{B}_i})$ is a transfer transition and $i_i = \overline{i}$, otherwise. By this construction, $u_{\mathcal{A}} \in \mathcal{L}(\mathcal{A})$ and $u_{\mathcal{B}} \in \mathcal{L}(\mathcal{B})$. Hence $u_{\mathcal{A}} \in \mathcal{L}(\mathcal{A}')$ and $u_{\mathcal{B}} \in \mathcal{L}(\mathcal{B}')$. So there exists two sequences of states in $\mathcal{A}'$ and $\mathcal{B}'$ such that:

$$q_{\mathcal{A}'0} \xrightarrow{(l_1,o_1)} q_{\mathcal{A}'1} \xrightarrow{(l_2,o_2)} q_{\mathcal{A}'2} \cdots q_{\mathcal{A}'(n-1)} \xrightarrow{(l_n,o_n)} q_{\mathcal{A}'n},$$

$$q_{\mathcal{B}'0} \xrightarrow{(i_1,l'_1)} q_{\mathcal{B}'1} \xrightarrow{(i_2,l'_2)} q_{\mathcal{B}'2} \cdots q_{\mathcal{B}'(n-1)} \xrightarrow{(i_n,l'_n)} q_{\mathcal{B}'n},$$

with $q_{\mathcal{A}'0} = q_{0\mathcal{A}'}$ and $q_{\mathcal{B}'0} = q_{0\mathcal{B}'}$.
We now show by induction that for all $1 \leqslant i \leqslant n$,

$$\ddagger : T_{\mathcal{A}' \times \mathcal{B}'}((q_{\mathcal{A}'i-1}, q_{\mathcal{B}'i-1}), (l_i, l'_i)) = (q_{\mathcal{A}'i}, q_{\mathcal{B}'i}).$$

i. If $i = 1$, if $o_1 = o$ and, hence, $i_1 = i$, $\ddagger$ is true, else, $o_1 = \overline{o}$ and $i_1 = \overline{i}$. If we suppose that there exists a transition labeled with (l_1, o) from $q_{\mathcal{A}'0}$ and a transition labeled with (i, l'_1) from $q_{\mathcal{B}'0}$, since one of these transition is not possible in either $\mathcal{A}$ or $\mathcal{B}$, there would exist a word accepted by $\mathcal{A}'$ not accepted by $\mathcal{A}$ or a word accepted by $\mathcal{B}'$ not accepted by $\mathcal{B}$, since no transfer synchronization can be realized between $\mathcal{A}$ and $\mathcal{B}$ from state $q_{\mathcal{A}0}$ and $q_{\mathcal{B}0}$ with letter (l_1, l'_1). Hence, the synchronization of $\mathcal{A}'$ and $\mathcal{B}'$ with letter (o_1, i_1) from state $(q_{\mathcal{A}'0}, q_{\mathcal{B}'0})$ is a non-transfer transition and $\ddagger$ is also true in this case.

ii. Let i be an integer such that $1 \leqslant i \leqslant (n-1)$ and let suppose that $\ddagger$ is true for all integers k with $1 \leqslant k \leqslant i$. Then, using exactly the same scheme of proof as in case i., we show that $\ddagger$ is true for integer $i + 1$.
We have then proved that $u \in \mathcal{L}(\mathcal{A}' \times \mathcal{B}')$, and this shows that:

$$\mathcal{L}(\mathcal{A} \times \mathcal{B}) \subset \mathcal{L}(\mathcal{A}' \times \mathcal{B}')$$

For symmetry reasons, the converse inclusion holds, which ends the proof. □
We now detail the possible applications of our formalism, showing how it can be used to test whether two SoCs are equivalent, provide automatically simulation and estimate performance of a component.

3 Applications

3.1 Comparison of SoCs

Let A and A' be two on-chip components that implement the same function. Components A and A' are *equivalent* if they cannot be distinguished by their external behaviors.
Deciding whether component A and A' are equivalent is possible with our formalism. Indeed, A and A' are equivalent if and only if their associated io-automata $\mathcal{A}$ and $\mathcal{A}'$ accept the same languages:

$$A \text{ and } A' \text{ are equivalent} \Leftrightarrow \mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$$

From the classical theory of finite state automata, it is possible to decide whether two automata accept the same language and if it is not the case, to obtain a word (that can be chosen of minimal length) that is accepted by one automaton but not by the other. Obtaining a sequence exhibiting divergent behaviors can be useful to decide which component behaves incorrectly and to help solving the problem. This approach is illustrated in subsection 4.2 on IDCT.

3.2 Simulation synthesis

Let A be a component and $\mathcal{A}$ its associated io-automaton. From the classical theory of finite state automata, we know that there exists a unique deterministic and complete automaton with minimal number of states that accepts the same language. This automaton is the *minimal automaton* associated to $\mathcal{L}(\mathcal{A})$. It can be computed efficiently in $O(|\Sigma||Q| \ln |Q|)$ time [Wat95].

We believe that, in practice, the size of the minimal automaton representing a component will be sufficiently small so that managing the input/output of a component in a simulation can be done by going through the automaton. This automaton can be represented by an array that would encode the transition function and since this array and the automaton have the same size, we think that the simulation obtained by this method would be efficient.

This approach is described in the IDCT case in subsection 4.3. On this example, it can be seen that the product has a smoothing effect on the input/output sequences since the minimal automaton representing IDCT is much smaller than the automaton obtained by computing the product of minimal automata of its subcomponents. That means that after the linking of the components, not all internal states are reachable and this is taken into account by our formalism.

3.3 Performance evaluation

Another interesting point is the ability to evaluate performance parameters such as the best computation time or the worst computation times in an environment (or by making assumptions on the way output samples are accepted and input samples are presented on the interface). Hence, since we are not able to verify that the function computed by the component is the function it is supposed to implement, we can nevertheless verify that the component furnishes the result of the computation under some time constraint. This point was not yet studied in the IDCT case.
We now illustrate these applications on IDCT.

4 A complete example: IDCT

Our first goal was to compare the component with a hand made abstraction which was implemented in a SystemC program since implementing each subcomponent would lead to very slow simulation. We were able to find that this abstraction is not correct as explained in subsection 4.2. In subsection 4.3, we describe the results obtained on the minimal io-automaton of IDCT. Now, we detail the modeling of 2-IDCT and we recall here that parameter 2 is chosen so that the automata represented in the sequel keep a reasonable size: we wrote programs that generate models for the real components and results of the following subsections were obtained using these models.

4.1 Modeling of 2-IDCT

In order to generate the automata that model IDCT components, we had to find parameters that allow to describe the internal state of each component. Pertinent parameters are, for example, the number of samples inside the component or the number of samples ready for output. The io-automata are then recursively created from the initial state, using this set of parameters. In the following, we describe the parameters chosen for each component and the io-automata modeling 2-IDCT are represented. We highlight the fact that it is the same set of parameters that are used to model the real component.

2-INR The internal states of 2-INR can be characterized by two parameters: the number of samples which are stored in the component and the number of samples ready for output. This leads to the io-automaton modeling 2-INR which is represented in Fig. 5.

2-ACCU The io-automaton modeling 2-ACCU is represented in Fig. 6. The internal states of 2-ACCU can be characterized by the number of samples which are stored inside the component.

2-POST The io-automaton modeling 2-POST is represented in Fig. 7. The internal states of 2-POST can be characterized by the number of samples which are stored inside the component.

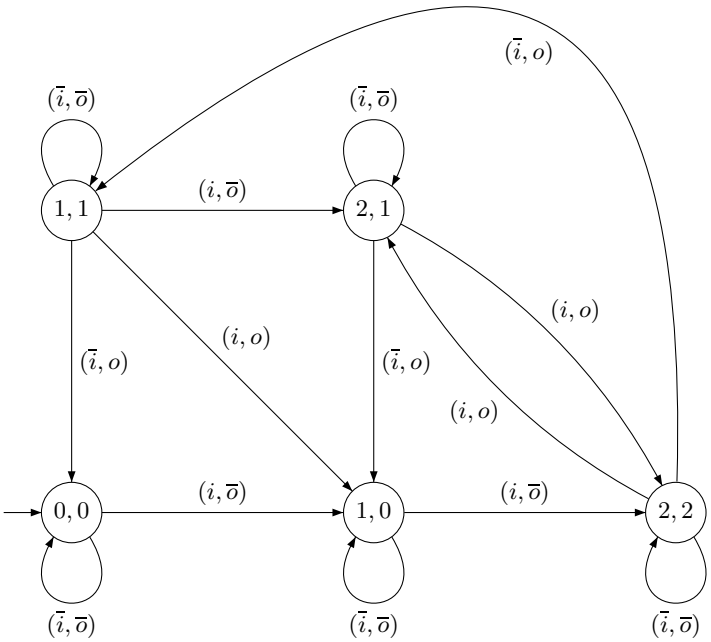


Fig. 5. 2-INR io-automaton

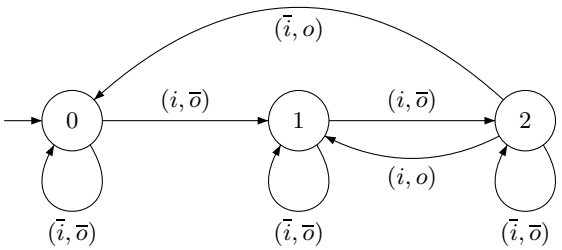


Fig. 6. 2-ACCU io-automaton

2-TRSP The io-automaton modeling 2-TRSP is represented in Fig. 8. Since this component can store up to four samples, parts of two different matrices can be stored at a time. Each state of the automaton is indexed by three integers. The first integer is the number of samples inside the component, the second is the number of samples inside the component of the first matrix and the third is the number of samples ready for output.

4.2 A pipelined model

SystemC[1] is a set of C++ classes dedicated to hardware simulation. We were provided a SystemC program that was supposed to implement an abstract version of IDCT.
The SystemC model separates the control and the processing. A first part is devoted to the temporal behavior (is an input or an output possible at the next rising edge) and a second one computes the transformed matrices of the input matrices.

[1] **http** : //**www.systemc.org/**

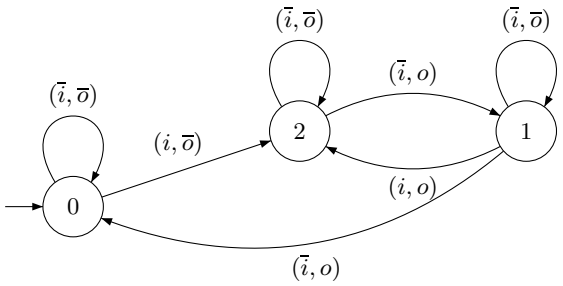


Fig. 7. 2-POST io-automaton

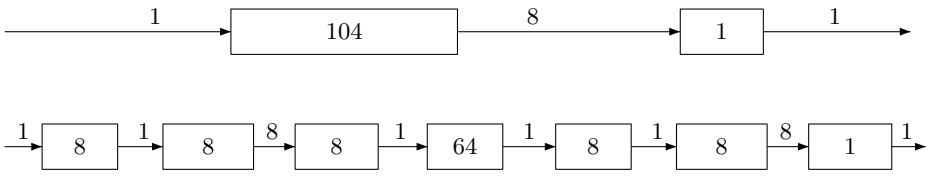
For the first part of the model, the manual abstraction consists in merging the first six components in one single, much simpler which is represented in Fig. 9. This component (called $PIPELINE$) can store a maximum of 104 samples. The minimum latency between the input of the last sample of an input matrix and the output of the first sample of the corresponding output matrix (20 cycles) is modeled in this component too. The output samples are transfered to the second component, eight at a time, like in the more detailed model:

Input width	1 sample
Output width	8 sample
Storage capacity	104 samples
I/O dependency	$o[64*n+k] = f(i[64*n], ...i[64*n+63])_{0 \leqslant k \leqslant 63}$
Minimum latency	20 cycles

We produced a model $\mathcal{A}_{PIPELINE}$ of the $PIPELINE$ component which we do not describe here since $\mathcal{A}_{PIPELINE}$ has 1527 states. We set $\mathcal{A}_{IDCT_a} = \mathcal{A}_{PIPELINE} \times \mathcal{A}_{POST}$. Using the automata library FSM[2], we proved that:

$$\mathcal{L}(\mathcal{A}_{IDCT}) \subsetneq \mathcal{L}(\mathcal{A}_{IDCT_a})$$

This shows that IDCT and pipelined IDCT are not equivalent, so that the abstraction is not correct. A trace of the divergence can be found easily with the help of the library and indeed, the word $(i,\overline{o})^{105}.(i,o)^8$ is a word of minimal length that demonstrates this mismatch. After the run with label $(i,\overline{o})^{105}.(i,o)^7$, the components are in the following states:



At the next rising edge, the pipelined component can accept the following input/output: (i,o),$(\overline{i},o)$ and $(\overline{i},\overline{o})$ whereas the original component can only accept $(\overline{i},o)$ and $(\overline{i},\overline{o})$.

We want to point out the facts that finding such a word by hand is a difficult task and that it can be easily obtained using the io-automata formalism. Moreover time required for all the computations is less than thirty seconds on a $G4$ processor. We now detail results on $\mathcal{A}_{IDCT}$.

[2] **http : //www.research.att.com/sw/tools/fsm/**

4.3 IDCT model study

First, we summarize the results concerning the sizes of different automata involved in IDCT:

Automaton	Reachable states	|Minimal Automaton|
$\mathcal{A}_{INR}$	44	44
$\mathcal{A}_{ACCU}$	9	9
$\mathcal{A}_{POST}$	9	9
$\mathcal{A}_{INR} \times \mathcal{A}_{ACCU} \times \mathcal{A}_{POST}$	477	308
$\mathcal{A}_{TRSP}$	2641	2641
$\mathcal{A}_{INR} \times \mathcal{A}_{ACCU} \times \mathcal{A}_{POST} \times \mathcal{A}_{TRSP}$	86466	5446
$\mathcal{A}_{IDCT}$	457815	6979

The important fact is that the minimal io-automaton of IDCT, denoted by $\mathcal{A}IDCT_m$, is much smaller (almost 70 times) than the automaton $\mathcal{A}_{IDCT}$, although it was computed from the minimal io-automata of INR, $ACCU$, $POST$ and $TRSP$. This illustrates the fact that product of io-automata has a smoothing effect on the language accepted, since the size of the minimal automaton can be viewed as a measure of the complexity of the accepted language. Indeed, $\mathcal{A}_{IDCT_m}$ has only 6979 states, so that it is possible to use it directly in the simulation program to manage the input/output of IDCT efficiently. $\mathcal{A}_{IDCT_m}$ can be represented, in the SystemC code, by an array of length 6979 that represents the transition function and input/output can then easily be handled by traversals the automaton.

Another point is that all computation times were about thirty seconds on a $G4$ processor, which shows that realistic examples can be treated by our methods efficiently and that we should be able to deal with more complex components, since $\mathcal{A}_{IDCT_m}$ remains a small automaton.

Conclusion

We propose methods that help comparing SoCs, implementing simulations of systems on chip and measuring performance parameters, by using an adapted formalism, namely the io-automata. Those methods are applied here to a particular component, IDCT which is a complex signal processing component with a linear topology. Generalizing our approach to components with several input/output interfaces seems easy, by extending the alphabet, and that would allow to integrate between components more elaborate communication protocols such as the VCI protocol [VSI01,BP03]. Moreover, since the size of the minimal automata modeling IDCT is quite small, we believe that modeling a complete MPEG video decoder or a Baseband Digital Processor should be possible.
Our approach allows a modular building of the component model: we model each subcomponent and then, we obtain the model for the complete component by computing the product of sub-models. However, even if this is a simpler task, each subcomponent has to be hand-modeled and in our future works, we plan to study how the model can be generated from the characteristic parameters of a components so that synthesis of a simulation program will become completely automatical.
Another idea we want to explore is to generate SystemC code from the model, by representing the automaton by a set of bounded integer variables (e.g. number of samples in the component, a latency counter, etc.) such that the transition function can be easily encoded with these variables. This would lead to faster simulation since the code of the program would be much shorter and hence, could be loaded in the cache memory of the processor it is executed on.

References

[AN82] A. Arnold and M. Nivat. Comportements de processus. In *Colloque AFCET "Les Mathématiques de l'Informatique*", pages 35–68, 1982.

[BBL+01] B. Bérard, M. Bidoit, F. Laroussinie, A. Petit, and Ph Schnoebelen. *Model-Checking Techniques and Tools*. Springer, 2001.

[BM97] G. Barret and A. McIsaac. Model checking in a microprocessor design project. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 1997.

[BP03] M. Baclet and R. Pacalet. Vérifications du protocole VCI. In *Actes 4ième Colloque sur la Modélisation des Systèmes réactifs (MSR'2003)*, pages 431–445. Hermès, 2003.

[LT89] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MJ91] Z. J. Mou and F. Jutand. A high-speed low-cost dct architecture for hdtv applications. In *Proceedings of International Conference on Acoustisc, Speech, and Signal Processing (ICASSP'91)*, pages 1153–1156, 1991.

[RY90] K. R. Rao and P. Yip. *Discrete cosine transform: algorithms, advantages, applications*. Academic Press Professional, Inc., 1990.

[SSG02] N. Savoiu, S. Shukla, and R. Gupta. Automated concurrency re-assignment in high level system models for efficient system-level simulation. In *Proceedings of Design, Automation and Test in Europe (DATE'2002)*, pages 875–889, 2002.

[TTP97] A. L. Turk, Probst S. T., and G. J. Powers. Verification of a chemical process leak test procedure. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 84–94. Springer, 1997.

[VSI01] VSI Alliance(TM). *Virtual Component Interface Standard, Version 2 (OCB 2 2.0)*. april 2001.

[Wat95] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, September 1995.

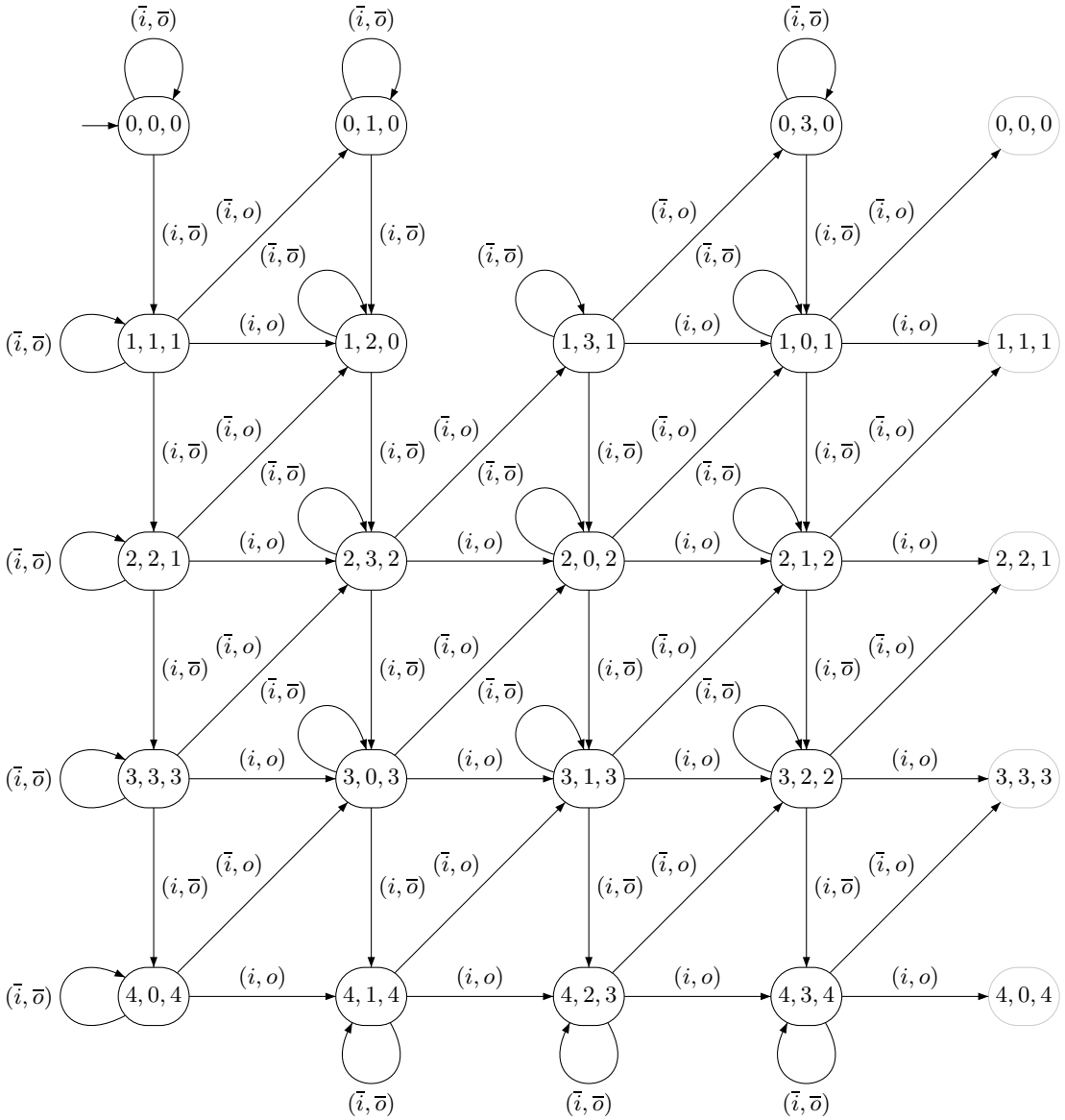


Fig. 8. 2-TRSP io-automaton

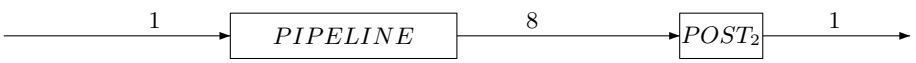


Fig. 9. Pipelined IDCT