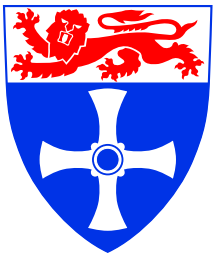
School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE



Cost-aware Synthesis of Asynchronous Datapath based on Partial Acknowledgement

Y. Zhou, D.Sokolov and A. Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2006-113

February 2006

Contact:

yu.zhou@ncl.ac.uk

danil.sokolov@ncl.ac.uk

alex.yakovlev@ncl.ac.uk

EPSRC supports this work via GR/S81421 (SCREEN)

NCL-EECE-MSD-TR-2006-113

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

http://async.org.uk/

Cost-aware Synthesis of Asynchronous Datapath based on Partial Acknowledgement

Y. Zhou, D.Sokolov and A. Yakovlev

February 2006

Abstract

We present in this report a novel method to design the asynchronous datapath circuit. The robustness of the implementation is based on the concept of partial acknowledgement. In each of the two design flows in this report, we introduce the method to implement the library as well as the algorithm directed by the cost functions. The example in the case study shows the advantages of our methods compared to previous ones.

1 Introduction

In the past few years, designers were trying to build up a bridge between synchronous and asynchronous circuit design. Rather than synthesizing the circuits from the "truly" asynchronous specifications such as communicating processes [8] or signal transition graph [18], the aim was to introduce the benefits of asynchronous techniques at a lower cost by reusing synchronous design tools like those for logic synthesis.

The asynchronous circuits built by these methods are distinctly robust in fighting against the delay variations which were difficult to control in deep-submicron technology. However, it had severe penalties of large overhead and high power consumption. For instance, the complex dual-rail implementations consumed much more than 200% of the power of the equivalent single-rail ones. NCL-D [7] was such a design method where each gate in the original single-rail netlist was replaced by the *input-complete* (**IC**)[4] (or *strongly-indicating* (**SI**), [11]) module encoded using dual-rail code, a delay-insensitive (DI) code [16]. In NCL-D implementation, all the fork wires existed in the original netlist were allowed to have arbitrary delays and were therefore non-critical. Only critical fork wires within the **IC** functional module needed to be verified for the timing closure, which was a relatively easy task through careful routing of the library elements. Whilst robust, the datapath implemented this way was bulky and slow as well because of the synchronization existed everywhere in the circuit. To reduce the overhead of NCL-D, another method was proposed in [5] (known as NCL-X) where each gate in the netlist was replaced by the more economical *early-propagative* (**EP**) functional module. By doing so, the previous non-critical forks in the NCL-D implementation became critical whose delays had to be bounded by the critical path delay in the completion detection (**CD**) circuitry. NCL-X used less area in the functional part but had the extra burden of the **CD** circuitry whose implementation style decided its actual overhead compared with NCL-D.

In this paper, we also design asynchronous datapath by reusing synchronous synthesis tools. Our method does not exclude the existence of the non-critical fork wires in the final implementation and is hence more

conservative than NCL-X. What distinguishes our method from the previous ones is that we pursue the minimum cost of an implementation to maintain the robustness by guiding the design procedures by explict cost functions. In this paper, we develop the *partial acknowledgement* as the key concept by which the requirement of robustness is guaranteed. We distribute the partial acknowledgement of every input and internal variable of the circuit in a cost-aware way. Two design flows are proposed in this paper on the understanding that the design objective can be transformed into different design procedures given different libraries. Our implementations are purely made up of the functional modules without CD circuitry, diminishing the verification task for timing closure compared with NCL-X. We observe from the case study that our design flows, especially the second one, notably reduce the area of an implementation while keeping the robustness at a reasonable level.

2 Background

2.1 Monotonic data flow in asynchronous circuit and the hazard-free conditions

Asynchronous circuits are free from using the clock for computation coordination. Instead, the request and acknowledgement (handshaking) signals are used in them [14]. In this paper, we adopt the dual-rail code, a DI code [16], where each 1-bit signal is represented using two rails: $rail^0$and $rail^1$. In a system using 4-phased protocol, *null* (or *spacer*) is represented by the combination of $rail^0$=$rail^1$=*0* whereas the data word "1" ("0") by $rail^0$=*0,* $rail^1$=1 ($rail^0$=*1,* $rail^1$=*0*), respectively. The data and *null* wavefronts flow consecutively where the variables in a circuit change gradually from *nulls*(valid code words) to valid code words(nulls). These monotonic transitions aim to eliminate the *hazard*, a dangerous phenomenon in asynchronous circuit resulting from the lack of a global clock. *Hazard* is also viewed as a consequence of insufficient acknowledgement [5], a notion describing that a change of signal *a* indicates that of another signal *b*. If we can make sure that every transition in both phases is acknowledged by another transition, the two consecutive data wavefronts would not interweave producing a malfunction. Equivalently, two criteria were proposed in [4] to ensure a *hazard-free*, 4-phased system:

1. *Completeness of Inputs,* which required that the outputs of a circuit may not transition from *null(data)* to *data(null)* until all inputs have transitioned from *null(data)* to *data(null)*. It allowed the *"weak conditions"* of signaling defined by Seitz [11].

2. *Observability,* which ruled out the propagation of an *orphan* passing through a gate. An *orphan* is defined as a wire that transitions during the current *data* wavefront but is not used in the determination of the outputs.

2.2 Timing assumptions in asynchronous datapath implementation

It is very difficult to design an asynchronous circuit without any timing assumption using standard logic gates, i.e., the Delay-Insensitive (DI) circuit [14] where all the gate elements and fork wires are allowed to have arbitrary delays. The category of circuits with an inferior level of robustness is known to be Quasi-Delay-Insensitive (QDI) where there exist both critical and non-critical forks. The wires after the non-critical forks are allowed to have arbitrary delay while those after the critical forks are required to satisfy the timing assumption, i.e., the isochronic fork assumption [1].

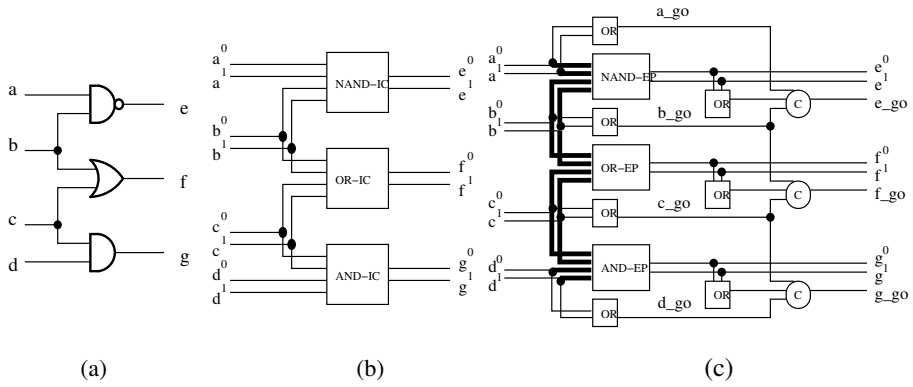


Figure 1: Timing assumptions in the NCL-D and NCL-X approaches

NCL-D is the practice in designing the dual-rail QDI circuit that reuses synchronous synthesis tools. It replaces each gate in the netlist with the equivalent **IC** functional module that satisfies the *Completeness of Inputs* introduced in 2.1. As an example, Figure 1 (b) implements the netlist in Figure 1(a) using NCL-D. The suffix **IC** stands for the *input-complete* implementation of a gate. Figure 2 illustrates two different styles to implement an **IC** functional module - those based on the Muller-C elements, i.e., DIMS [15] and the threshold logic, i.e., the Null Convention Logic (NCL) [13]. Each **IC** functional module in Figure 1 represents an "equipotential region" [11] outside which the fork wires are allowed to have arbitrary delay without disturbing the correct functioning. Inside an **IC** functional module, however, the wire delays are required to satisfy the timing bounds. We mark those wires in Figure 2 to be verified for the timing closure when *a* and *b* changing from *nulls* to the valid code words "1".

The third category of the asynchronous circuit is known to be Speed-Independent (SI) where all the fork wires in the circuit are critical. NCL-X is the method to implement it. It consists of two parts: the functional parts and the completion detection circuitry. In the functional part, each gate is replaced by the equivalent *early-propagative* (**EP**) functional module that does not satisfy the *Completeness of Inputs*. Figure 2 demonstrates the different implementations of an **EP** module of the AND function. We can see that either implementation will output a valid data "0" when only one of the inputs transitions from *null* to the valid data "0". The completion detection circuitry made up of OR gates and C-element tree, on the other hand, checks the variables' states in each wavefront. Figure 1(c) illustrates the implementation of the netlist in Figure 1(a) using NCL-X, where the **EP** suffix stands for *early-propagative* implementation introduced above. In the NCL-X implementation, the delays of all the fork wires fan-outing to the **EP** modules in the functional part of the circuit must be bounded by the critical-path delay in the completion detection circuitry in order to exclude the hazards. In the implementation in Figure 1(c), the fork wires to be verified for timing closure are shown in bold.

As shown in Figure 2 there exists library-dependent implementations for both the **IC** and **EP** functional modules of the same simple gate (AND gate in our example). [12] compared their areas and speed.

3 Partial acknowledgement

For a dual rail encoded variable *n* in a circuit with the 4-phased protocol, we use the notation $n\uparrow$ and $n\downarrow$ to denote the *rising phase transition* of *n* from *null* (n_{null}) to a valid code word belonging to n_{cw} and the

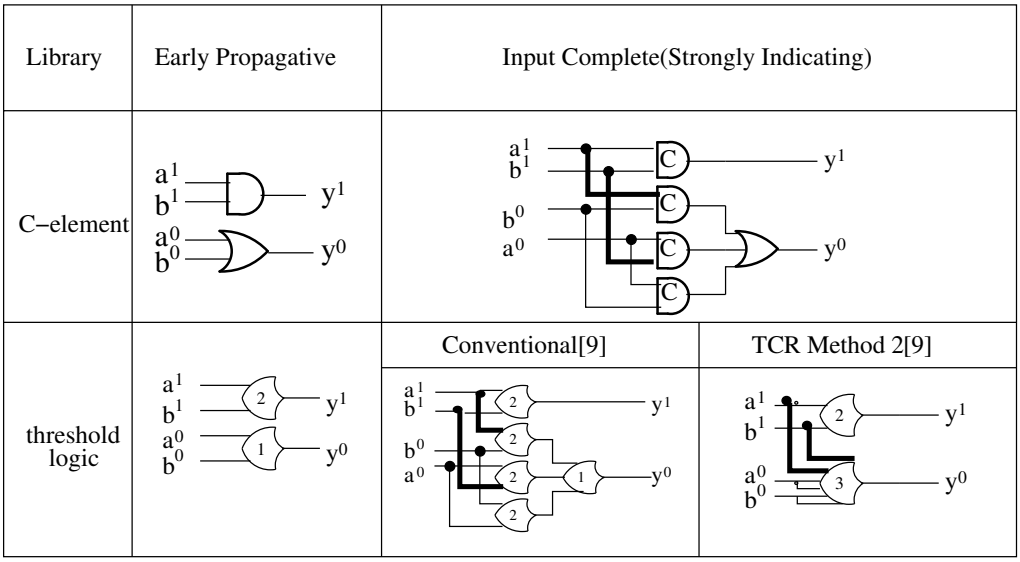


Figure 2: Different implementations of the functional modules for an AND gate

falling phase transition from a valid code word to n_{null}, respectively. n_{cw}, the set of valid code words, includes the valid code words "1" and "0".

We define the ***partial acknowledgement*** of the rising phase transition of a variable *n* by the rising phase transition of another variable *m*∈*direct-fan-out(n)* iff:

$$(\forall n \uparrow \in n_{null\rightarrow cw} : \exists m \uparrow \in m_{null\rightarrow cw} : n \uparrow \rightarrow m \uparrow) \tag{3.1}$$

Similarly, we define the ***partial acknowledgement*** of the falling phase transition of a variable *n* by the falling phase transition of another variable *m*∈*direct-fan-out(n)* iff:

$$(\forall n \downarrow \in n_{cw\rightarrow null} : \exists m \downarrow \in m_{cw\rightarrow null} : n \downarrow \rightarrow m \downarrow) \tag{3.2}$$

From the definitions 3.1 and 3.2, it is obvious that by observing the transition of variable *m* from *null* (valid word) to valid word (*null*) we can determine whether variable *n* has already transitioned from *null* (valid data) to valid data (*null*), if the rising (falling) phase transition of *n* is partially acknowledged by that of *m*. The acknowledgement is defined to be partial because variable *n* can have other fan-outs besides *m.*

We define that a variable *n* is ***partially acknowledged*** if both its rising and falling phase transitions are partially acknowledged. Particularly, if they are partially acknowledged by the corresponding phase transitions of the same variable *m*∈*direct-fan-out(n)*, we say that variable *n* is ***partially acknowledged*** by *m*, i.e. ,

$$\begin{aligned}(\forall n \quad \uparrow \quad &\in n_{null\rightarrow cw} : \exists m \uparrow \in m_{null\rightarrow cw} : n \uparrow \rightarrow m \uparrow) \\ \wedge \quad &(\forall n \downarrow \in n_{cw\rightarrow null} : \exists m \downarrow \in m_{cw\rightarrow null} : n \downarrow \rightarrow m \downarrow)\end{aligned} \tag{3.3}$$

Finally we define that a variable *n* is ***acknowledged*** iff it is partially acknowledged by every variable belonging to *n*'s direct fan-outs.

From our definitions, the output of an **IC** functional module partially acknowledges all its input variables. On the contrary, the output of an **EP** functional module partially acknowledges none of its input variables.

Therefore, every input and internal variable in NCL-D implementation is acknowledged. In NCL-X implementation, however, each input and internal variable is only partially acknowledged.

4 Designing asynchronous datapath based on partial acknowledgement

In this section, we present two design flows to implement the asynchronous datapath where every input and internal variable of the circuit is at least *partially acknowledged*. Both design flows have the same objective and work with the structural single-rail netlist that can be generated using the high-level and logic synthesis tools. The gates in the single-rail netlist are converted to the appropriate types of dual-rail functional modules existing in a design library, according to the specific algorithm aiming to minimize the circuit's overhead. No explicit completion detection circuitry, like that used in NCL-X, are adopted in our design flows. This avoids the extra routing of the wires in the completion detection circuitry and reduces the verification task as well. The design objective is elaborated in the following part:

Design Objective: Given the single-rail netlist of a synchronous circuit, implement its dual-rail counterpart, i.e., a dual-rail encoded netlist with an equivalent function where every gate in the original netlist is replaced by the appropriate type of its dual-rail encoded functional module. Find out such an implementation with *minimum area* under the requirement that both the rising and falling phase transitions of each input and internal variable are partially acknowledged.

The first design flow represents a quick remedy to the previous methods. The library associated with it contains only the **IC** and **EP** functional modules of the basic gates. In this case, **IC** functional modules are the only candidates to partially acknowledge a circuit variable. We develop the algorithm in sections 4.1.1 and 4.1.2. The second design flow is more flexible in that it provides a library of the functional modules of the basic gates with the tuned ability to partially acknowledge the input variables. Besides the **EP** and **IC** functional modules, the design library in design flow 2 has other functional modules that can partially acknowledge any arbitrary combination of the input variables. The flexibility in the second design flow delivers a further reduction of the implementation that is as robust as the first one. We introduce its library design in section 4.2.1 and the algorithm in 4.2.3.

4.1 Design flow 1

In this design flow, the responsibility for partial acknowledgement of the input and internal variables is taken exclusively by the **IC** functional modules in the circuit. Section 2.2 introduced different techniques to implement the **IC** and **EP** functional modules. Our design algorithm is independent of the implementation techniques.

We revisit the example in Figure 1(a). We intuitively implement gate e and gate g with their **IC** functional modules whereas the gate f the **EP** one in the target circuit in Figure 3(b). By our definition, the input variables a and d are acknowledged while b and c partially acknowledged. 92 transistors (the DIMS implementation of the **IC** functional modules) are used in our method compared with 102 used in NCL-X (the C-element implementation of the **CD** circuitry). Our method also avoids the extra routing of the interconnections used in the **CD** circuitry in NCL-X. In terms of robustness, our design has less number of wire forks required for timing verification compared with NCL-X, which are marked out by bold lines in Figure 3(b).

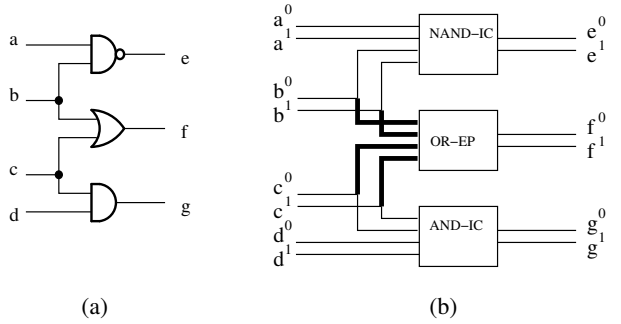


Figure 3: Intuitive implementation of the exemplar circuit

The limitations of the design library in the first design flow are twofold. Firstly, the rising and falling phase transitions of a circuit variable are partially acknowledged by the same functional module(s). Secondly, circuit variables can only be partially acknowledged by the **IC** functional module and a **IC** functional module will partially acknowledge all its inputs. Thus we can transform the design objective into the problem of finding out a set of gates in the synchronous netlist, after being converted to their **IC** functional modules, that can partially acknowledge all the input and internal variables with the minimum cost in area. ***Partial acknowledgement matrix (*PAM*)*** is developed in 4.1.1 to facilitate the design task followed by the specific algorithm for design flow 1 based on the **PAM**.

4.1.1 Defining the design problem based on PAM

A **PAM** is a 2-dimensional matrix constructed from the single-rail netlist. The size of a **PAM** is m by n, where m is the number of input and internal variables in the netlist while n is the number of gates. Each row in a PAM stand for an input or internal variable in the circuit whereas a column for one gate. A value "1" is assigned to the element $\mathbf{PAM}_{ij}$ if variable i is a direct input to gate j and otherwise a value "0". For example, the **PAM** of the circuit in Figure 3(a) is demonstrated in Table 1. A column j is said to cover the row i iff

variable	gate e	gate f	gate g
a	1	0	0
b	1	1	0
c	0	1	1
d	0	0	1

Table 1: PAM of the exemplar circuit in Figure 3(a)

$\mathbf{PAM}_{ij}$=1.

According to the design objective in 4.1, we need to find out the set of gates in a netlist covering all the rows in its PAM with a minimum cost of the final implementation. An exact solution to it needs to firstly search all the possible sets of gates covering the PAM and then compare their implementation costs to find out the most economical one. During the cost estimation, a gate is evaluated using its **IC** functional module's area if it belongs to the cover set or otherwise the **EP** one. However, this exact solution involves the full exploration of the search space and requires a long computation time. In the following, we introduce a quick search method that enables some reduction strategies. We will aim the minimum cost of all the **IC** functional

modules that cover all the variables because the **IC** modules take the dominant place in the overall overhead compared with the **EP** ones.

In this method, cost c_j is associated with the *j*-th column in the **PAM** representing the area of the **IC** functional module of gate *j*. We use the number of transistors in a functional module as an indication of its area in this paper. In the case of the **PAM** in Table 1, all the columns have the same cost under the same implementation technique. It is natural that the costs of the same type of gate can be different when using different implementation techniques. E.g., the **IC** implementation of the **AND2** gate by TCR method is much more economical in area than that by DIMS [12].

We refine our design problem based on **PAM**: given the **PAM** of a circuit, find out the set of columns that covers all the rows with a minimum total cost. Then replace the gates in this set with their **IC** functional modules while others the **EP** ones. The problem can be formulated as a special form of the Integer Linear Program (ILP), i.e., finding out $min(\sum c_j s_j)$ subject to the constraint $PAM \cdot \mathbf{s} \geq \mathbf{1}$, where

$\mathbf{s} = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}$ ($s_j = 1\, if\, column\, j\, is\, selected$, $or\, s_j = 0\, if\, column\, j\, is\, not\, selected$) and $\mathbf{1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$.

After finding out the minimum cost solution $\mathbf{s}$, implement the gate *j* with its **IC** functional module if $s_j = 1$or its **EP** functional module if $s_j = 0$.

We can recognize without difficulty that gate *e* and gate *g* are the only gates in an optimal solution that will be converted to their **IC** functional modules. But for the larger circuit it is necessary to develop some formal procedures to facilitate the design process.

4.1.2 Formalization of the design procedure

In this section we will first discuss how to simplify the **PAM** without interfering the possibility of finding out the optimal solution. Three different reduction techniques are proposed with their applications to the specific circuit examples. They originated from the reduction methods to the more general problem, the unate covering problem (UCP) [2].

	.....	gate g	.....
⋮	.....	⋮	.....
Variable m	0 ···· 0	1	0 ···· 0
⋮	.....	0	.....
Variable n	.....	1	.....

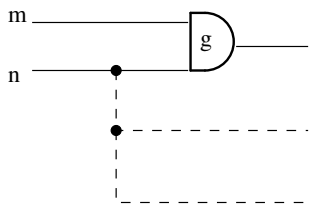


Figure 4: Essential gate g

- Essential Column (Gate)

If there is only one non-zero value in the row r_i of a PAM, i.e., variable *i* has only one fanout gate, then the column (gate) with a value 1 on row *i* is call the essential column (gate) and must be included in the covering set because its **IC** functional module is the only one that can partially acknowledge variable *i*. Figure 4 shows the circuit example where *m* has only one fan-out gate *g*. In this case *g* must be mapped to its **IC** functional module in the final implementation. After the essential column is identified, all the rows covered by it are eliminated from the **PAM** because they have been partially acknowledged by the essential gate.

- Row (Variable) Dominance

A variable *i* in the circuit dominates another variable *j* if the row corresponding to r_i in the PAM has 1s in the same columns as row r_j. In terms of a circuit, this means that the set of fan-out gates of the variable *j* is a subset of that of the variable *i*. We know that variable *i* will be partially acknowledged if *j* is partially acknowledged but not the other way around. Consequently, we can safely remove the dominating row (r_i) from the **PAM** because its constraint condition is superfluous. In the example of Figure 5, variable *m* dominates *n* and the row standing for variable *m* can be withdrawn from our consideration.

- Column (Gate) Dominance

A gate *i* dominates another gate *j* if the column *i* in the *PAM* has 1s in the same rows as column *j*. Provided c_i is no higher than c_j, we can discard the dominated gate without affecting the final results purely because the dominating gate partially acknowledges more variables than the dominated one with smaller cost. In the example of Figure 6, gate *r* dominates gate *s* and therefore we can eliminate the column (gate) *s* in the *PAM*.

	...	gate r	gate s	gate t	...
⋮	⋮	⋮	⋮	⋮	⋮
Variable m	⋮	1	1	1	⋮
⋮	⋮	⋮	⋮	⋮	⋮
Variable n	⋮	1	0	1	⋮

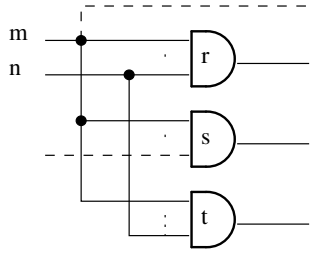


Figure 5: Dominance of variable m over variable n

	...	gate r	gate s	...
⋮	⋮	⋮	⋮	⋮
Variable l	⋮	1	1	⋮
Variable m	⋮	1	0	⋮
Variable n	⋮	1	1	⋮

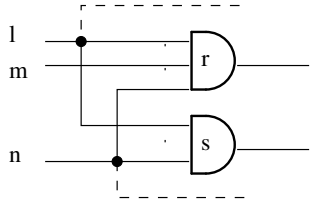
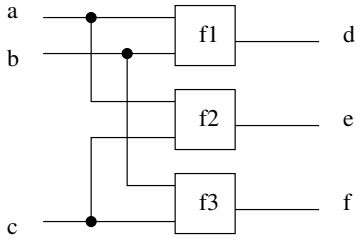


Figure 6: Dominance of gate r over gate s

We simplify the **PAM** of the target circuit using above reduction schemes until no more partially unacknowledged variables left. However we cannot always rely on the three rules because there are situations where none of them can be used. It was called the "cyclic" situation as illustrated in Figure 7 where there is no obvious reason to choose one gate rather than another as part of the optimal solution.

We could use the algorithm to find a minimum-cost solution of the cyclic PAM by searching the design space making up of all possible combinations of its columns. But its computational complexity is exponential to the number of gates in a circuit and therefore very costly. Alternatively, we can use the branch-and-bound algorithm to reduce the search space without sacrificing the optimal solution. In this branch-and-bound algorithm, we trace the branches in a binary search tree where each node in the tree represents a gate that will be either chosen or not in an optimal solution. At the beginning of the algorithm, we



(a)

Variable	f1	f2	f3
a	1	1	0
b	1	0	1
c	0	1	1

(b)

Figure 7: A examplar circuit (a) with a cyclic PAM (b)

estimate the lower bound which is the lowest possible cost to cover all the variables left in the **PAM** (Some quick estimation method is introduced in [2]). During the search process, we will stop at a branch whose cost is equal to the lower bound no matter whether other optimal solutions exist or not. Besides the lower bound, we keep the upper bound of a PAM that is equal to the lowest cost of the branch in the detected part so far. The upper bound will be dynamically updated by the newly found lower ones. We will discard a branch if its cost estimation is higher than the upper bound and thus reduce the computation complexity. During the branch and bound algorithm, we can use the three reduction methods introduced above to iteratively simplify the PAM until all the variables are crossed out, according to our design objective in 4.1. The general branch and bound algorithm can be found in [2] and a more recent one in [3]. We summarize in algorithm 1 the design procedures used in design flow 1.

Suppose in Figure 7 we chose *f2* to be included in the optimal solution. By removing the variables it covers and using the column (gate) dominance, we come to the optimal solution whose **IC** set includes *{f1*, *f2}* or *{f2*, *f3}*. A more complex example will be studied in section 5.

4.1.3 Timing assumptions

In the final implementation, a fork is non-critical if all the gates it fans into are implemented by their **IC** functional modules. For the fork wires fanning into **EP** functional modules, we need to ensure their timing closure to avoid a potential hazard.

4.2 Design flow 2

One limitation of design flow 1 is its narrow spectrum of functional modules in the design library. It includes either the **IC** functional modules that partially acknowledge all the inputs or the **EP** ones that partially acknowledge none. Consequently, the partial acknowledgement of the circuit variables in an implementation is congregated at certain IC functional modules determined by algorithm1. The first design flow, however, may not lead to the globally optimal solution wherethe partial acknowledgement of circuit variables is freely distributed. By free distribution, we mean, first, to allow a functional module to partially acknowledge only a fraction of its input variables and, second, to allow separate functional modules for the partial acknowledgement of the rising and falling phase transitions of a particular variable. This requires setting up of the modules with the same function but tuned ability to partially acknowledge the designated phase transitions of certain input(s). Martin's adder [9] is an example to to distribute the evaluation of the *nulls* and valid code words of the inputs among *sum* and *carry-out*. In [10], Nielsen proposed the general rules to build the

Algorithm 1 Algorithm of design flow 1

```
Reduction of PAM (PAM)
 {
  IC-Set=empty-set;
  // essential gate reduction
  Essential_gate (PAM);
  // reduction of the dominating variable
  for each r_i∈PAM
     {
     for each r_j∈PAM | j≠i
           {
           v_ij=r_i-r_j;
           if (every element ∈v_ij≥0)
           cross out r_i from the PAM;
           Essential_gate (PAM);
           }
     }
  // reduction of the dominated gate
  for each c_i∈PAM
     {
     for each c_j∈PAM | j≠i
           {
           v_ij=c_i-c_j;
           if (every element ∈v_ij≥0)
           cross out c_j from the PAM;
           Essential_gate (PAM);
           }
     }
     if (cyclic PAM)
           {
           branch-and-bound(PAM);
           }
     return (PAM, IC-Set);
 cost_estimation(IC-Set);
 }

subroutine Essential_gate (PAM)
  {
  for each row r_i in the PAM
     if (∑ PAM_ij = 1);
     IC-Set=IC-Set∪Gate_j|(PAM_ij=1);
     cross out {row_k|PAM_kj=1} from the PAM;
  }
```

reduced direct logic for a similar distribution in [9]. In this paper, we use Boole's expansion theorem in dual rail context to synthesize the functional modules. Then we discuss the cost functions and algorithms used to distribute the partial acknowledgement of a variable in a systematic approach.

4.2.1 Functional modules synthesis

In this section we design the dual-rail encoded modules with different functions that can partially acknowledge certain phase transition(s) of the input variable(s). We name these functional modules using the con-

vention of *module name_(input name and its phase transitions)**, where the *phase transitions* of an input include the rising phase transition (↑), falling phase transition (↓) and both phases transitions (⋆). For example, **AND3_a↑** denotes the 3-input-AND functional module that partially acknowledges the rising phase transition of its input *a* whereas **MAJ3_a⋆b⋆** refers to a 3-input-majority-voting functional module with the boolean function $f = ((a \wedge b) \vee (a \wedge c) \vee (b \wedge c))$ where both the rising and falling phase transitions of *a* and *b* are partially acknowledged by it.

The circuit style used to implement the functional modules is pseudo-static that comprises the pull-up-network (PUN), pull-down-network (PDN), and the output inverters. PDN is made up of n-typed transistors that will partially acknowledge all the rising phase transitions of the designated inputs to a functional module. PUN consists of p-typed transistors that will partially acknowledge all the falling phase transitions of the prescribed inputs. The weak feedback inverter is used to fight against charge leakage problem. Alternatively we can use the cross-coupled p-typed transistors for this purpose. Figure 8 (a) illustrates a prototype of the pseudo-static functional modules with f^0 and f^1 its dual output rails. The PDNs of both output rails share as many as possible n-typed transistors and thus are presented in one block.

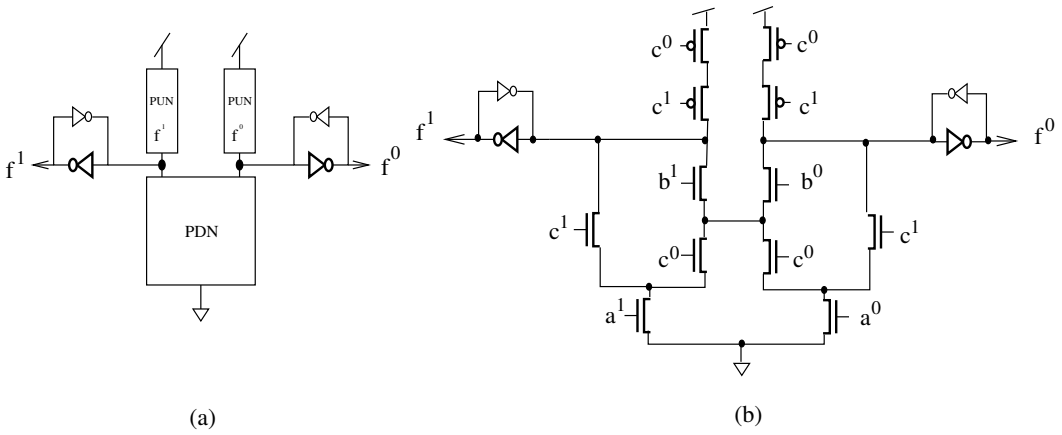


Figure 8: (a) Prototype of the functional module (b) implementation of **MUX2:1_a↑c⋆**

We introduce the design procedures of a general functional module through the example of **MUX2:1_a↑c⋆**.

Step 1: Derive the dual-rail functions, i.e., f^0 and f^1 , from the boolean function *f*. f^1 is converted from *f* by replacing the uncomplemented variables in *f* with their *rail-1s* whereas the complemented ones with their *rail-0s*. f^0 is simply the dual of f^1. E.g., the boolean function of module **MUX2:1_a↑c⋆** is $f = ca + c'b$. We can derive its dual-rail boolean functions of $f^1 = c^1a^1 + c^0b^1$ and $f^0 = c^1a^0 + c^0b^0$.

Step 2: Apply the Boole's Expansion Theorem to f^1 and f^0 with respect to the selected inputs whose rising phase transitions are to be partially acknowledged. If no inputs are to be partially acknowledged for their rising phase transitions, we will keep f^1 and f^0as they have been derived in the first step. In this step we first construct the *dual-rail minterms* of the selected input variables whose rising phase transitions are to be partially acknowledged. For instance, if *a* and *c* are the selected inputs as in our example, then the *dual-rail minterms* would be a^0c^0, a^0c^1, a^1c^0 and a^1c^1. After this we decompose f^1 and f^0 in terms of the *dual-rail minterms*. The decomposition is of the general form:

$$f^1 = m_0 \cdot f^1_{m_0=1} + m_1 \cdot f^1_{m_1=1} + \cdots + m_{2^{n-1}} \cdot f^1_{m_{2^n-1}=1}$$

$$f^0 = m_0 \cdot f^0_{m_0=1} + m_1 \cdot f^0_{m_1=1} + \cdots + m_{2^n-1} \cdot f^0_{m_{2^n-1}=1} \tag{4.1}$$

, where m_0, ..., m_{2^n-1} are the dual-rail minterms of the selected *n* input variables.

Such a decomposition ensures that the rising phase transitions of the selected input variables are partially acknowledged by the rising phase transition of the module's output. It is because the sum of the products in (4.1) ensures there will always exist a minterm in any path from ground to the module's output rails. A minterm is implemented by the cascade of n-typed transistors controlled by its factors, as will be introduced in step 3. We demonstrate the expansion of f w.r.t. its inputs a and c in the example of **MUX2:1_a↑c⋆**, where

$$\begin{aligned} f^1 &= a^0c^0 \cdot f^1_{a^0=c^0=1} + a^0c^1 \cdot f^1_{a^0=c^1=1} + a^1c^0 \cdot f^1_{a^1=c^0=1} + a^1c^1 \cdot f^1_{a^1=c^1=1} \\ &= a^0c^0 \cdot (b^1) + a^0c^1 \cdot (0) + a^1c^0 \cdot (b^1) + a^1c^1 \cdot (1) \end{aligned}$$

$$\begin{aligned} f^0 &= a^0c^0 \cdot f^0_{a^0=c^0=1} + a^0c^1 \cdot f^0_{a^0=c^1=1} + a^1c^0 \cdot f^0_{a^1=c^0=1} + a^1c^1 \cdot f^0_{a^1=c^1=1} \\ &= a^0c^0 \cdot (b^0) + a^0c^1 \cdot (1) + a^1c^0 \cdot (b^0) + a^1c^1 \cdot (0) \end{aligned} \tag{4.2}$$

Step 3: Connect the n-typed transistors in PDN according to the dual-rail expansion in (4.1). The connection rules are similar to that of the complementary static CMOS complex gates: a product term is implemented as the cascade of the n-typed transistors controlled by its factors and then the product terms are connected in parallel to generate a particular output rail. Figure 8(b) shows the final implementation of the module **MUX2:1_a↑c⋆**, where the n-transistors in the PDN are shared to save the total transistors number.

Step 4: Design the PUN of a functional module. For a functional module to partially acknowledge an input's falling phase transition, we need cascade two p-typed transistors controlled by both rails of the variable in the paths from VDD to both the output rails. This is similar to the requirement in [10]. Finally according to [10], a p-typed transistor in the PUN can be removed if it is controlled by an input rail that does not appear in any path from ground to the same output rail. This further simplification is not applicable to our example in Figure 8(b) because both c^1 and c^0 are present in the PDNs. However, we must be cautious in some scenarios where this reduction would be unsafe. For instance, the removal of a p-typed transistor according to the above rule could be dangerous if the input variable is only partially acknowledged for its falling phase transition but not the rising one. In this case, it may fail to acknowledge the rising phase transitions of all the input patterns. It is always safe to apply the reduction rule in a functional module where both the rising and falling phase transitions of an input is partially acknowledged. Figure 9 shows the implementation of **AND2_a⋆** after the removal of a^0 in the PUN of f^1.

If a functional module is not responsible for the partial acknowledgement of the falling phase transitions of any inputs, its PUNs can be implemented in either static or dynamic style. In the former case, the PUNs are complementary to the PDNs and the feedback inverter can be removed because it's free from the dangling state. In the later case, only one p-typed transistor exists in each of the PUNs and it is controlled by the global

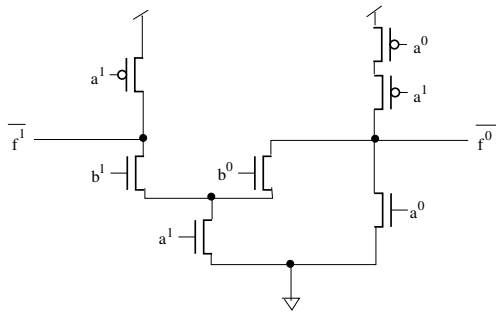


Figure 9: Implementation of **AND2_a⋆**

resetting signal. The resetting signal acts as the precharge signal used in the dynamic logic styles [6] [17] but only controls the foregoing functional modules. Figure 10 illustrates the two ways to implement the functional module **AND2_a↑**. The precharge signal only acts as parallel resetting to some part of the circuit

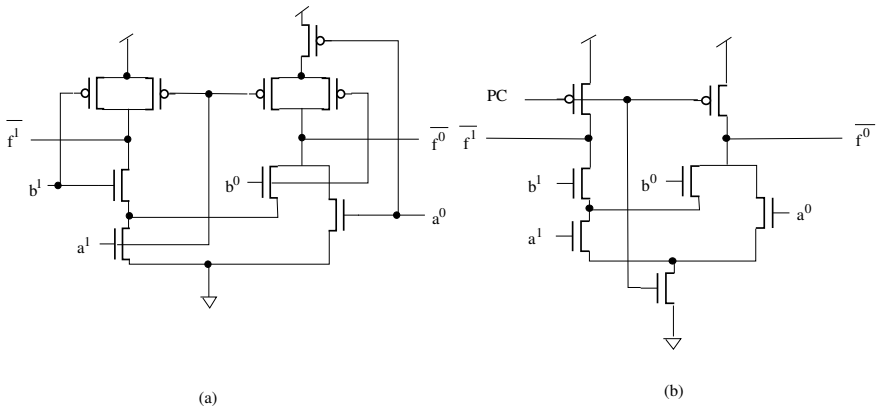


Figure 10: Implementation of **AND2_a↑** by complementary static logic (a) and dynamic logic (b)

and we still rely on the partial acknowledgement of the falling phase transitions of the circuit variables for the separation between two valid data wavefronts. Consequently we need not worry about the skew of the precharge signal.

4.2.2 Determination of the design library

In theory we can design the functional modules that partially acknowledge all the possible combinations of their input variables' phase transitions. It provides a wide spectrum of design elements but involves great design complexity as well. We restrict the combinations in this paper by requiring that the falling phase transition of a circuit variable can only be partially acknowledged by the functional module that acknowledging the variable's rising phase transition. This restriction would not degrade the final performance because we know that the rising phase transition is acknowledged by the p-typed transistors in the PUNs whose number is generally constant (4 p-typed transistors per input). In addition, we can safely apply the reduction rules if the rising and falling phase transitions of a variable are acknowledged by the same module, according to step 4 in section 4.2.1.

The fundamental modules designed according to the procedure in 4.2.1 and satisfying the above restric-

tion are summarized in Table 2 for their costs estimation. The numbers of inputs being partially acknowledged are increasing from left of the table to right. The **EP_DR** column represents the dual-rail encoded, early propagative functional modules. The estimations we make for the **EP_DR** ones are based on the complementary static implementation while we can use the dynmic implementation as well (see Figure 10 for reference).

We can see the the biggest difference of the second design flow, compared with the first one, is that we can have modules partially acknowledging any number, including none, of the inputs. Asymmetries exist in some functional modules in that they use different numbers of transistors to partially acknowledge the same number of inputs. For instance, the transistors counts are different for **AO21_a⋆_b⋆** and **AO21_a⋆_c⋆.**

4.2.3 Measuring the cost of partial acknowledgement

If an input fans into several different modules, we know from Table 2 that generally these modules have different costs to partially acknowledge the input. However, we cannot rely on their absolute values to determine the best candidate to partial acknowledge the input. Imagine the part of a circuit illustrated in Figure 11. The input *a* is the only partially unacknowledged input of *f1* and *f2*, whose other inputs have been acknowledged elsewhere in the circuit. From the Table 2 we know that *f1* uses less transistors (**OR3_a⋆**)

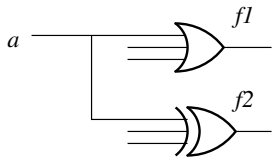


Figure 11: Different costs in partial acknowledging *a*

than *f2* (**XOR3_a⋆**) to partially acknowledge input a. But the choice to implement *f1* as **OR3-a⋆** and *f2* as the **EP_DR** type is much more expensive than the other way around, *f1* as **EP_DR type** and *f2* as **XOR3_a⋆** (49 v.s. 34 transistors).

We realize that the above choice should be based on the comparison of the increased costs of a functional module for its partial acknowledgement of a particular input. That's why the *incremental cost*, $\triangle n$, is introduced. It denotes the number of the increased transistors of a functional module for its *n*-th partially acknowledged input variable. Table 3 summarizes the incremental costs of the different functional modules. In table 3, $\triangle 1$ refers to the increased number of transistors of a functional module for the first partially acknowledged input compared with its **EP_DR** implementation. Different incremental costs exist in $\triangle 1$ ($\triangle 2$, $\triangle 3$) for the same type of gate because of asymmetry.

In the circuit illustrated in Figure 11, the incremental cost to partially acknowledge a by *f1* is $\triangle 1$(*f1*)=$\triangle 1$(OR3)=5 while by *f2* is $\triangle 1$(*f2*)=$\triangle 1$(XOR3)=-10. Therefore, we choose *f2* as the functional module to partially acknowledge the input a.

4.2.4 Optimal distribution of partial acknowledgement using local search algorithm

In the second design flow, we try to find the optimum solution to the design objective stated in the beginning of this section. The library of this design flow consists of the functional modules in Table 2 and those whose functions are not listed in Table 2 but developed according to the procedures in section 4.2.1 under the restrictions of 4.2.2.

Functional Module	**EP_ DR**	**_a⋆(_b⋆)**			**_c⋆**			**_a⋆b⋆**			**_a⋆c⋆(_b⋆c⋆)**			**_a⋆b⋆c⋆**		
		PUN	PDN	Tot	PUN	PDN	Tot	PUN	PDN	Tot	PUN	PDN	Tot	PUN	PDN	Tot
AND(OR)2	8	3	4	15	-	-	-	6	6	20	-	-	-	-	-	-
XOR2	16	4	6	18	-	-	-	8	6	22	-	-	-	-	-	-
AND(OR)3	12	3	6	17	3	6	17	6	8	22	6	8	22	9	10	27
AO21(OA21)	12	4	8	20	3	6	17	8	8	24	7	8	23	11	10	29
MAJ3	20	4	8	20	4	8	20	8	8	24	8	8	24	12	12	32
XOR3	32	4	10	22	4	10	22	8	10	26	8	10	26	12	10	30
MUX2:1	20	4	10	22	4	6	18	8	10	26	8	8	24	12	14	34

Table 2: Transistors count of the exemplar library used in design flow 2

Acknowledgement

Functional Module	△1		△2			△3	
	a(b)	c	a(b)+b(a)	a(b)+c	c+a(b)	ab+c	a(b)c+b(a)
AND(OR)2	7	-	5	-	-	-	-
XOR2	2	-	4	-	-	-	-
AND(OR)3	5	5	5	5	5	5	5
AO21(OA21)	8	5	4	3	6	5	6
MAJ3	0	0	4	4	4	8	8
XOR3	-10	-10	4	4	4	4	4
MUX2:1	2	-2	4	2	6	8	10

Table 3: Incremental cost for an input variable to be partially acknowledged

We construct the **PAM** of a circuit in the same way as introduced in the first design flow. However in this case, a gate (a column in a **PAM**) in the netlist is no longer an "atomic" entity where only the entire inputs can be partially acknowledged. Instead, any number of the module's inputs can be partially acknowledged according to the implementation algorithm. Therefore, we cannot apply the simplification techniques introduced in section 4.1.2 except for the *essential column (gate)* because they do not guarantee a cost-non-increasing reduction. We will take the *row (variable) dominance* as an example. In second design flow, variable *m* can not be withdrawn because it is not necessarily partially acknowledged by the functional module acknowledging *n* (E.g., gate *r* or(and) *t*).

The local search algorithms, instead, are adopted in this design flow. It is directed by the costs table in Table 3 and is locally optimal at each search step because the decision at each step is dependent on the previous one. Therefore, the "seeds" used to initiate the searching algorithms are of great importance.

We use the *modified essential gate* to find out the initial seeds in a **PAM**. If an input (internal) variable in the circuit has only one fanout gate, then the gate's dual-rail implementation must partially acknowledge this variable because there is no other gates to do this job. After all the essential gates have been identified and their acknowledged inputs removed from the **PAM**, we decide to choose those unacknowledged inputs to the foregoing located essential gates as the next-step seeds. These unacknowledged variables have high credits in leading to an overall optimal solution because some of their fan-outs (the essential gates) have been determined unambiguously. All we need to decide then is which fanout gates are chosen to partially acknowledge those unacknowledged variables. The cost table in Table 3 is used for this task where we will choose the one with the lowest incremental cost. If several gates have the same incremental cost, we shall choose the one with the highest number of unacknowledged inputs as it will introduce more future "seeds". We repeat the searching procedure untill all the input and internal variables in the circuit have been partially acknowledged. Finally we can estimate the overall cost by summing the transistors number used in each gate according to Table 2. Figure 2 shows this local search algorithm.

4.2.5 Timing assumptions

The design flow 2 has the same level of robustness as the first one. It has both critical and non-critical forks in the final implementation. In addition, it is very easy to locate the non-critical fork wires as they are the wires fanning in to the functional modules that does not *partially acknowledged* it, i.e., to the modules without the $\star$ notation in their corresponding input ports.

Algorithm 2 Local search algorithm in design flow 2

local_search (PAM)
{
({acked_vars}, {acking_gates}):=modified-essential-gate (PAM);
cross_out({acked_vars});
initial_seeds={unacked inputs ∈{acking_gates}};
  while (more unacked vars in PAM)
  {
    while (more vars in the initial-seeds)
    {
    variable i∈initial-seeds;
    lowest-cost-gate(i):=gate g | △(g)=min({incremental-cost(fanout(i))});
    acked_inputs(g)∪=i;
    initial_seeds∪={unacked input∈g };
    cross_out(i);
    }
  initial_seeds∪=rand_var(PAM);
  }
cost(PAM):=∑cost(g);
return (cost(PAM), {acked_inputs(g∈column(PAM))});
}

5 Case study: a 2-bit Carry-Skip adder

5.1 2-bit Carry-Skip adder

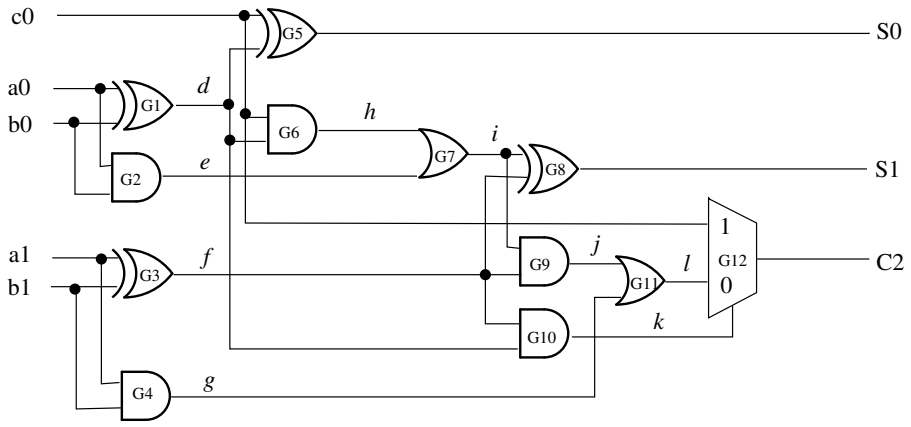


Figure 12: A 2-bit Carry-Skip adder

We demonstrate the two design flows by the example of a 2-bit carry-skip adder (Figure 12) . We construct its **PAM** , a 14 by 12 matrix as shown in Table 4. Each row in the **PAM** represents an input (internal) variable in the circuit whereas each column a gate. $\mathbf{PAM}_{ij}$ is assigned a value "1" if variable *i* is a direct input to the gate *j*.

	G1	*G2*	*G3*	*G4*	*G5*	*G6*	*G7*	*G8*	*G9*	*G10*	*G11*	*G12*
a0	1	1										
a1			1	1								
b0	1	1										
b1			1	1								
c0					1	1						1
d					1	1				1		
e							1					
f								1	1	1		
g											1	
h							1					
i								1	1			
j											1	
k												1
l												1

Table 4: PAM of the 2-bit Carry-Skip adder

5.2 Demonstration of design flow 1

From the *Essential_gate* in algorithm 1, we know that IC-Set={*G7*, *G11, G12}* should be implemented as their **IC** functional modules. The input variables to the IC-Set, *c0, e, g, h*, *j*, *k* and *l,* are removed from the PAM because they have been partially acknowledged. The row representing variable *f* dominates *i* and therefore can be removed according to *row dominance*. Then we apply the *column dominance* which will add *G2, G4, G6* and *G9* to the IC-Set because the **IC** implementation of AND2 is cheaper than that of XOR2, according to Table 2. Finally we have EP-Set={G1, G3, G5, G8, G10} that will be implemented by their **EP** functional modules.

We implement the gates belonging to IC-Set and EP-Set according to Table 2, where a gate's **IC** functional module is replaced by the one partially acknowledging all its input variables while a gate's **EP** one by its **EP_DR** module. The circuit uses 226 transistors in total.

5.3 Demonstration of design flow 2

According to the Algorithm 2, we apply the *modified essential gate* to the *PAM* to find out the *acking_gates*={*G7, G11, G12*} and the *acked_vars*={*e, g, h, j, k, l*}. We know that the gate *G7, G11,* and *G12* must be implemented by the functional modules that can partially acknowledge *e* and *h, g* and *j*, and *k* and *l*, respectively. The variable *c0* is chosen as the first seed for the local search because it is the only unacknowledged input belonging to *acking_gates*. We will choose the gate to partially acknowledge *c0* with the lowest cost, i.e., *lowest_cost_gate(c0),* by comparing the incremental costs of *c0*'s fan-outs. According to Table 3 we know that the incremental cost of *G5*, the $\triangle 1(G5)$, equals to $\triangle 1$(XOR2) which is 2 because *c0* would be the first input of *G5* being partially acknowledged if chosen. Similarly we have $\triangle(G6)=\triangle 1$(AND2)=7 and $\triangle 3(G12)=\triangle 3$(MUX2:1)=10. Therefore we select *G5* as *lowest_cost_gate(c0)* that partially acknowledges *c0*. The variable *d* is the next seed and we have *lowest_cost_gate(d)=G5*. We repeat this procedure until all the input and internal variables in the **PAM** are partially acknowledged. Table 5 lists the whole decision process.

In the final implementation, *G1, G3, G5, G7, G8, G11* are replaced by the corresponding functional modules in table 2 that partially acknowledge all their input variables. *G12* is replaced by **MUX2:1_b⋆c⋆**.

No.	**Seed**	**Incr. cost of the seed's fan-outs**	**lowest_cost_gate**
1	*c0*	△1*(G5)=2,* △1*(G6)=7,* △3*(G12)=10*	*G5*
2	*d*	△2*(G5)=4,* △1*(G6)=7,* △1*(G10)=7*	*G5*
3	a0	△1*(G1)=2,* △1*(G2)=7*	*G1*
4	b0	△2*(G1)=4,* △1*(G2)=7*	*G1*
5	*a1*	△1*(G3)=2,* △1*(G4)=7*	*G3*
6	*b1*	△2*(G3)=4,* △1*(G2)=7*	*G3*
7	*f*	△1*(G8)=2,* △1*(G9)=7,* △1*(G10)=7*	*G8*
8	*i*	△2*(G8)=4,* △1*(G9)=7*	*G8*

Table 5: Unfolding of the decision process in design flow 2

All the rest gates are implemented by their **EP_DR** functional modules according to Table 2. The total cost used in design flow 2 is 192 transistors.

5.4 Results

Table 6 compares the areas used in our design flows and that in NCL-D and NCL-X. The **IC** functional modules in NCL-D and the **EP** functional modules in NCL-X are implemented using the corresponding functional blocks of Table 2 for a fair comparison. The C-elements trees used in the CD circuitry of NCL-X are implemented in a similar way to that of Figure 1 (c). The percentage of the area increase of a particular implementation is compared to the area of the 2-bit single-rail carry-skip adder which uses 84 transistors.

We can see from the table that the dual-rail implementation of the adder, without any variable being partially acknowledged, has a 67% increase in area. The overhead penalty is not 100% as someone may think because the inverters in a single-rail circuit are replaced by purely swapping the dual rails of its input variable. NCL-D is the implementation with the highest level of robustness because all the inter-modules interconncetions in it are non-critical in terms of delay. Its robustness comes at a price of 212% increase of a single-rail netlist's area. NCL-X increase the number of fork wires to be verified for the timing closure (as listed in the *Verification Demand* column) but increases the area as well. It is a little controvertial as it actually leads away from the optimal Area-Verification curve in our example. However, as we have explained, the area of NCL-X is dependent on the way how the CD circuitry is implemented. If we don't limit the number of inputs fanning in to a C-element in the CD circuitry, we will reduce the transistor numbers notably.

It is demonstrated in the table that our design flows reduce the area of an asynchronous implementation compared to that of NCL-D and NCL-X while maintaining a better level of robustness compared with NCL-X. In our two design flows, the second one is more prominent in achieving this goal by introducing the design library with wider spectrum of ability to partially acknowledge the input variables.

2-bit CS Adder	Area(transistors #)	Area Increase	Verification Demand
DR-static-CMOS	140	67%	N/A
NCL-D	262	212%	0
NCL-X	334	298%	50
Design Flow 1	226	169%	20
Design Flow 2	192	129%	22

Table 6: Comparison of the implementations of a 2-bit Carry-Skip adder

6 Conclusion

We propose in this paper two design flows that explore the possibility to design asynchronous datapath where the reliability of the circuit is introduced in a cost-aware manner. In each design flow, we develop the algorithms to implement the circuit given the corresponding design library. The study of a 2-bit carry-skip adder shows our method is successful in striking a balance between the overhead reduction of an asynchronous implementation and the maintenance of the robustness at a reasonable level.

References

[1] C. H. van Berkel. Beware the isochronic fork. Nat. Lab. Unclassified Report UR 003/91, Philips Research Lab., Eindhoven, The Netherlands, 1991.

[2] Gary D.Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[3] E.Goldberg, L.P.Carloni, T. Villa, R.K.Brayton, and A.L. Sangiovanni-Vincentelli. Negative thinking in branch-and-bound: the case of unate covering. *IEEE Transactions on Computer-Aided Design*, March 2003.

[4] Karl M. Fant and Scott A. Brandt. NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis. In *International Conference on Application-specific Systems, Architectures, and Processors*, pages 261–273, 1996.

[5] Alex Kondratyev and Kelvin Lwin. Design of asynchronous circuits using synchronous CAD tools. *IEEE Design & Test of Computers*, 19(4):107–117, 2002.

[6] L.Heller, W.Griffin, J.Davis, and N.Thoma. Cascode voltage switch logic: a differential cmos logic family. In *International Solid State Circuits Conference*, pages 16–17, 1984.

[7] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, and Alex Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, April 2000.

[8] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[9] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[10] Christian D. Nielsen. Evaluation of function blocks for asynchronous design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 454–459. IEEE Computer Society Press, September 1994.

[11] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[12] S. C. Smith, R. F. DeMara, J. S. Yuan, D. Ferguson, and D.Lamb. Optimization of null convention self-timed circuits. *Journal of Systems Architecture*, 37(3):135–165, August 2004.

[13] S. C. Smith, R. F. DeMara, J. S. Yuan, M. Hagedorn, and D. Ferguson. Delay-insensitive gate-level pipelining. *Integration, the VLSI journal*, 30(2):103–131, 2001.

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

[15] Jens Sparsø and Jørgen Staunstrup. Design and performance analysis of delay insensitive multi-ring structures. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 349–358. IEEE Computer Society Press, January 1993.

[16] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[17] Ted E. Williams. Latency and throughput tradeoffs in self-timed asynchronous pipelines and rings. Technical Report CSL-TR-90-431, Stanford University, August 1990.

[18] Alexandre Yakovlev. Designing self-timed systems. *VLSI Systems Design*, 6:70–90, September 1985.