Volume 3 ▪ Issue 1 ▪ October 2012

Editor-in-Chief
Professor. Santosh Kumar Nanda

INTERNATIONAL JOURNAL OF

LOGIC AND COMPUTATION (IJLP)

ISSN : 2180-1290

Publication Frequency: 6 Issues / Year

CSC PUBLISHERS
http://www.cscjournals.org

Copyrights © 2012 Computer Science Journals. All rights reserved.

INTERNATIONAL JOURNAL OF LOGIC AND COMPUTATION (IJLP)

VOLUME 3, ISSUE 1, 2012

EDITED BY
DR. NABEEL TAHIR

ISSN (Online): 2180-1290

International Journal of Logic and Computation (IJLP) is published both in traditional paper form and in Internet. This journal is published at the website http://www.cscjournals.org, maintained by Computer Science Journals (CSC Journals), Malaysia.

IJLP Journal is a part of CSC Publishers
Computer Science Journals
http://www.cscjournals.org

INTERNATIONAL JOURNAL OF LOGIC AND COMPUTATION (IJLP)

Book: Volume 3, Issue 1, October 2012
Publishing Date: 25-10-2012
ISSN (Online): 2180-1290

This work is subjected to copyright. All rights are reserved whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illusions, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication of parts thereof is permitted only under the provision of the copyright law 1965, in its current version, and permission of use must always be obtained from CSC Publishers.

IJLP Journal is a part of CSC Publishers
http://www.cscjournals.org

© IJLP Journal
Published in Malaysia

Typesetting: Camera-ready by author, data conversation by CSC Publishing Services – CSC Journals, Malaysia

CSC Publishers, 2012

EDITORIAL PREFACE

It is a great privilege for me as Editor in Chief of International Journal of Logic and Computation (IJLP) to present our readers the current issue of Journal which wraps up its third year and first issue of successful publication. This journal has focused on publishing research that provides information for practitioners, researchers and academicians with a teaching or research interest in engineering and science discipline. The first issue of IJLP is organized to presents articles in a particular area of computer logic and computation to attract readers who are interested in reading papers related to that special field. The first issue of IJLP provides a better chance to fulfill the anticipation of a broader community of our audiences.

The initial efforts helped to shape the editorial policy and to sharpen the focus of the journal. Starting with Volume 4, 2013, IJLP aims to appear with more focused issues. Besides normal publications, IJLP intend to organized special issues on more focused topics. Each special issue will have a designated editor (editors) – either member of the editorial board or another recognized specialist in the respective field.

As EIC of IJLP, I want to encourage contributors to IJLP to submit not only manuscripts addressing basic and applied research articles but also reviewed articles, practitioner oriented papers and other exploratory research projects addressing contemporary issues in such areas as Computational Logic, Knowledge based systems, Application of Logic in Hardware and VLSI, Soft Computing Techniques, Type theory, Natural Language etc. The review process will remain the same for these articles as mentioned in the official website of IJLP.

IJLP editors understand that how much it is important for authors and researchers to have their work published with a minimum delay after submission of their papers. They also strongly believe that the direct communication between the editors and authors are important for the welfare, quality and wellbeing of the Journal and its readers. Therefore, all activities from paper submission to paper publication are controlled through electronic systems that include electronic submission, editorial panel and review system that ensures rapid decision with least delays in the publication processes.

To build international reputation of IJLP, we are disseminating the publication information through Google Books, Google Scholar, Directory of Open Access Journals (DOAJ), Open J Gate, ScientificCommons, Docstoc, Scribd, CiteSeerX and many more. Our International Editors are working on establishing ISI listing and a good impact factor for IJLP. I would like to remind you that the success of the journal depends directly on the number of quality articles submitted for review. Accordingly, I would like to request your participation by submitting quality manuscripts for review and encouraging your colleagues to submit quality manuscripts for review. One of the great benefits that IJLP editors provide to the prospective authors is the mentoring nature of the review process. IJLP provides authors with high quality, helpful reviews that are shaped to assist authors in improving their manuscripts.

Editorial Board Members
International Journal of Logic and Computation (IJLP)

EDITORIAL BOARD

EDITOR-in-CHIEF (EiC)

Professor Santosh Kumar Nanda
Eastern Academy of Science and Technology
India

EDITORIAL BOARD MEMBERS (EBMs)

Professor Madhumangal Pal
Vidyasagar University
India

Dr. Nidaa Abdual Muhsin Abbas
University of Babylon
Iraq

Assistant Professor. Jitendra Kumar Das
Synergy Institute of Engineering and Technology
India

Dr. Eng. Sattar B. Sadkhan
University of Babylon
Iraq

TABLE OF CONTENTS

Volume 3, Issue 1, October 2012

Pages

Novel Parallel - Perfix Structure Binary to Residue Number System Conversion Method

Omar Dajani
Pepe Siy Wayne State Univerisity
Department of Electrical and Computer Engineering

Abstract

In the present world there is always a demand for faster algorithms and techniques that could boost up the speed of the computations. With the help of VLSI fabrication techniques and using residue number system (RNS) arithmetic we can achieve the faster speeds. In this paper we propose a novel parallel prefix binary to residue number system conversion method. The method that we present in this paper utilizes parallel-prefix technique with multiplexers and modulo adders as the main building blocks which makes it practical and suitable for VLSI implementation.

Keywords: Residue Number System, Binary to Residue Conversion, Multiplexer, Modulo Adder.

1. INTRODUCTION

In the last decade, Residue number system (RNS) has received increased attention due to its ability to support high speed concurrent arithmetic applications [1-3] such as Fast Fourier transform (FFT), image processing and digital filters utilize the efficiencies of RNS arithmetic in addition and multiplication. The advancements in very large scale integration (VLSI) technology and demand for parallelism computation have enabled researchers to consider RNS as an alternative approach to high speed concurrent arithmetic.

Several methods are found in literature for binary to RNS conversion. Alia and Martinelli [4] have proposed a method for binary to residue conversion based on powers of 2. A modification to the above method was proposed by Cappocelli and Giancarlo [5]. Anandmohan [6] has proposed a similar method but with difference that his method is based on the cyclic property of power of 2 moduli set. Behrooa[7] proposed a table lookup schemes for binary to Residue conversions.

In this paper, we present a novel binary to Residue Number System conversion method that we used to build Residue Arithmatic logic unit (RALU). RALU has three main units: Binary to Residue unit, ALU and Residue to Binary unit [8]. The organization of this paper is as follows. Section two explains RNS system. In section three we present new conversion from binary to RNS algorithm. Section four and five show illustrative example and implementation selection techniques. Section six is comparison between the new method and pervious work. Conclusion is in section seven.

2. RESIDUE NUMBER SYSTEM

Any n-bit nonnegative integer number X, in the range $0 \le X \le 2^n-1$ is represented in binary number system as $X = 2^{n-1} b_{n-1} + ... + 2^2 b_2 + 2 b_1 + b_0 = \sum_{j=0}^{n-1} 2^j b_j$

where $b_j \in \{0, 1\}$.

Meanwhile in RNS, X is represented by k residue digits x_i as $X = \{x_1, x_2, x_3, \ldots, x_k\}$ where $x_i = X \bmod m_i$ and m_i belong to set of relatively prime moduli; $m_i \in \{m_1, m_2, m_3, ..., m_k\}$ [9]. If

the moduli are relatively prime numbers, there is a unique RNS representation for each integer in range $0 \le X \le \prod_{i=1}^{s} m_i$

3. NEW NOVEL CONVERSION METHOD FROM BINARY TO RESIDUE REPRESENTATION

As shown above an integer number X can be represented in Binary system as

$$X = 2^{n-1} b_{n-1} + ... + 2^2 b_2 + 2 b_1 + b_0 = \sum_{j=0}^{n-1} 2^j b_j$$

And RNS representation of number X is

$$|X|_m = \left|\sum_{j=0}^{n-1} \left|2^j b_j\right|_m\right|_m \qquad \text{for } m > 2 \qquad \text{for } m > 2$$

$$= \left|\sum_{j=0}^{n-1} \left|2^j\right|_m b_j\right|_m \qquad (1)$$

Let $M_{A_1A_0} = (A_1, A_0)[Y_0, Y_1, Y_2, Y_3]$ denotes a 2-bit multiplexer where the 2 control bits (A1, A0) select the inputs (Y_0, Y_1, Y_2, Y_3) to be outputted

Lemma 1: For any pair of bits b_j& b_i for j & i ≥ 0,

$$\left|\left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m = \left|X_{ji}\right|_m$$

can be implemented using 2-bit multiplexer :

$$M_{ji} = (b_j, b_i)[0, \left|2^i\right|_m, \left|2^j\right|_m, \left|\left|2^j\right|_m + \left|2^i\right|_m\right|_m] \qquad (2)$$

Where the control bits (A_1, A_0) equal (b_j, b_i)

Proof:

Rewrite equation $\left|\left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m$ as

$$(0.\bar{b}_j.\bar{b}_i) + (\left|2^i\right|_m.\bar{b}_j.b_i) + (\left|2^j\right|_m.b_j.\bar{b}_i) + (\left|\left|2^j\right|_m + \left|2^i\right|_m\right|_m.b_j.b_i)$$

This is equivalent to 2-bit multiplexer M_{ji} with control bits (A_1, A_0) equal (b_j, b_i). Figure (1) shows the implementation for equation (2) with $b_j = b_1$ and $b_i = b_0$

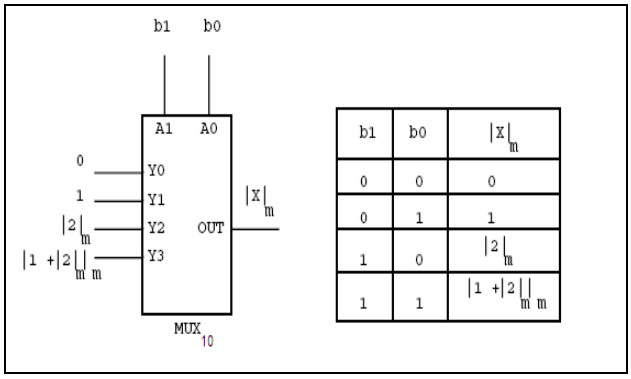


FIGURE 1: Two Bits (b1 & b0) Binary to Residue Number System Conversion

This pre-processing operator M_{ji} is represented in acyclic graph as node " ○ " in figure (2a), where all the inputs are constants and pre-calculated .

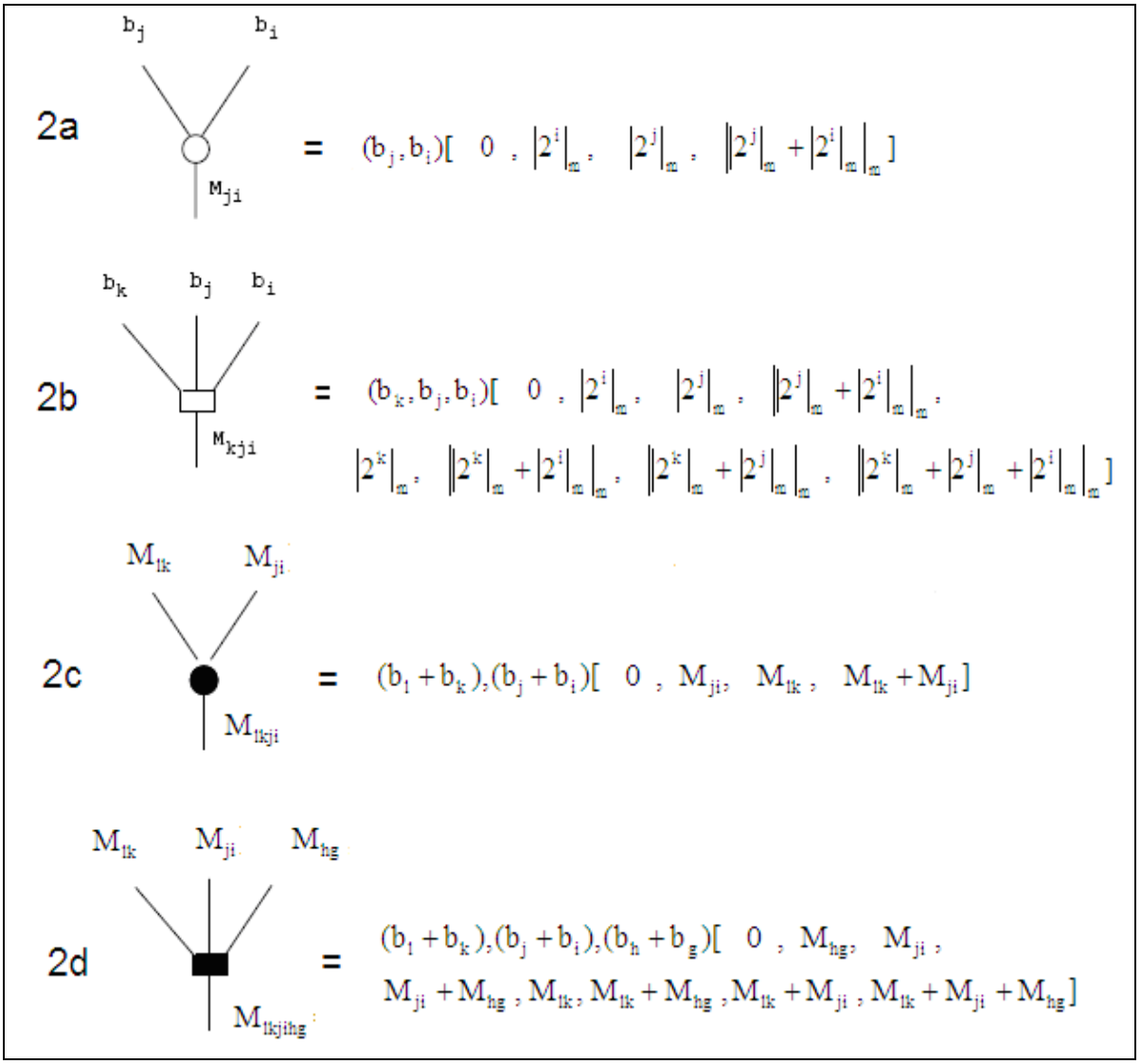


FIGURE 2: Prefix logic operation and their implementation

In three bit system, let $M_{A_2A_1A_0} = (A_2, A_1, A_0)[Y_0,\ \ Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7]$ denotes a 3-bit multiplexer where the 3 control bits (A_2, A_1, A_0) select the inputs ($Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7$) to be outputted.

Lemma 2: For any three bits b_k, b_j & b_i for k, j & i ≥ 0,
$\left\| 2^k \right|_m b_k + \left| 2^j \right|_m b_j + \left| 2^i \right|_m b_i \Big|_m = \left| X_{kji} \right|_m$ can be
implemented using 3-bit multiplexer :

$$M_{kji} = (b_k, b_j, b_i)[0\ , \left| 2^i \right|_m, \left| 2^j \right|_m, \left\| 2^j \right|_m + \left| 2^i \right|_m |_m, \left| 2^k \right|_m, \left\| 2^k \right|_m + \left| 2^i \right|_m |_m,$$
$$\left\| 2^k \right|_m + \left| 2^j \right|_m |_m, \left\| 2^k \right|_m + \left| 2^j \right|_m + \left| 2^i \right|_m |_m] \qquad (3)$$

Where control bits (A_2, A_1, A_0) equal (b_k, b_j, b_i)

Proof:

Rewrite $\left|\left|2^k\right|_m b_k + \left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m$ as

$$(0.\bar{b}_k.\bar{b}_j.\bar{b}_i) + (\left|2^i\right|_m.\bar{b}_k\bar{b}_j.b_i) + (\left|2^j\right|_m.\bar{b}_k.b_j.\bar{b}_i) +$$
$$(\left|\left|2^j\right|_m + \left|2^i\right|_m\right|_m.\bar{b}_k.b_j.b_i) + (\left|2^k\right|_m.b_k.\bar{b}_j.\bar{b}_i) +$$
$$(\left|\left|2^k\right|_m + \left|2^i\right|_m\right|_m, b_k\bar{b}_j.b_i) + (\left|\left|2^k\right|_m + \left|2^j\right|_m\right|_m b_k.b_j.\bar{b}_i) +$$
$$(\left|2^k\right|_m + \left|2^j\right|_m + \left|2^i\right|_m.b_k.b_j.b_i)$$

Above equation is equivalent to 3-bit multiplexer with b_k, b_j & b_i as selection control inputs. Figure (3) shows the implementation for equation 3 with $b_k = b_2$ $b_j = b_1$ and $b_i = b_0$

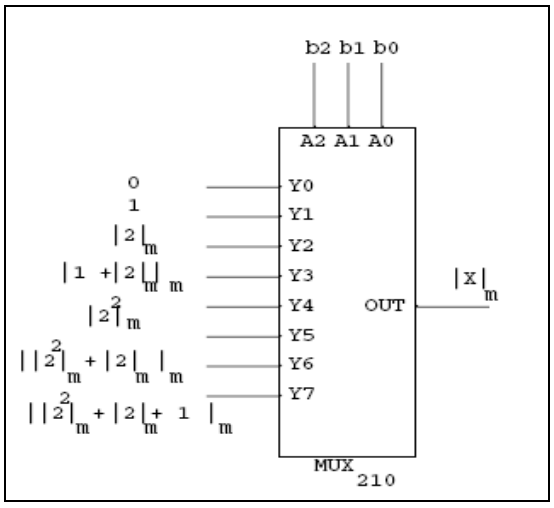


FIGURE 3: Three bits (b_2, b_1, b_0) Binary to Residue Number System Conversion

This pre-processing operator M_{kji} is represented in acyclic graph as node " □ " in figure (2b), where all the inputs are constants and pre-calculated.

Theorem 1: For Any two pairs of bits (b_l & b_k) (b_j & b_i for j, i , l & k ≥ 0 with the given expression

$$\left|\left|2^l\right|_m b_l + \left|2^k\right|_m b_k + \left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m = \left|X_{lkji}\right|_m$$

can be implemented using 2-bit multiplexer

$$M_{lkji} = (b_l + b_k),(b_j + b_i)[0, M_{ji}, M_{lk}, M_{lk} + M_{ji}] \qquad (4)$$

Where control bits (A_1, A_0) equal (b_l,+b_k , b_j,+b_i)

Proof $\left|X_{lkji}\right|_m = \left|\left|2^l\right|_m b_l + \left|2^k\right|_m b_k + \left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m$

$$\left|X_{lkji}\right|_m = \left|M_{lk} + M_{ji}\right|_m \qquad (5)$$

Where $M_{lk} = \left|\left|2^l\right|_m b_l + \left|2^k\right|_m b_k\right|_m$ and $M_{ji} = \left|\left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m$ by Lemma 1

Let $b_{lk} = (b_l + b_k)$ and $b_{ji} = (b_j + b_i)$

Rewrite equation (5) as $(0.\bar{b}_{lk}.\bar{b}_{ji}) + (M_{ji}.\bar{b}_{lk}.b_{ji}) + (M_{lk}.b_{lk}.\bar{b}_{ji}) + (M_{lk} + M_{ji}.b_{lk}.b_{ji})$.

And this is equivalent to 2-bit multiplexer M_{lkji} with control bits (A_1, A_0) equal (b_l+ b_k , b_j+ b_i). Figure (4) shows implementation for two pair bits (b_3, b_2) & (b_1, b_0)

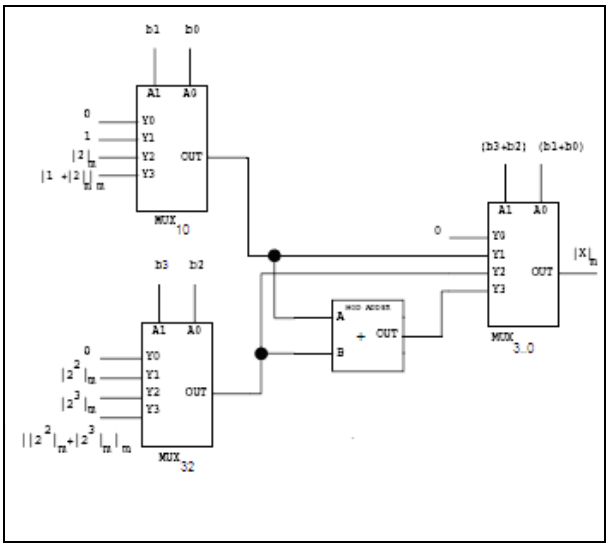


FIGURE 4: Four Bits Binary to RNS $\left|X_{3..0}\right|_m = \left|\left|2^3\right|_m b_3 + \left|2^2\right|_m b_2 + \left|2^1\right|_m b_1 + b_0\right|_m$

Lemma 3:

Combining two pairs of bits (b_l & b_k) (b_j & b_i) requires one 2-bit multiplexer and one 2 input mod adder. The delay time $\tau_{Total} = \tau_{mux_2} + \tau_{modadder_2}$

Proof

Equation (4) and figure (4) show that $(b_l + b_k),(b_j + b_i)[0, M_{ji}, M_{lk}, M_{lk} + M_{ji}]$ is equivalent to one 2-bit multiplexer and one 2-input mod adder; and delay time is equal to $\tau_{mux_2} + \tau_{modadder_2}$.

Figure (2c) represents acyclic graph " ● " for node M_{lkji} where M_{lk} & M_{ji} are inputs.

Lemma 4

The parallel prefix operator ● has the following properties

1) Commutative

$$M_{lk} \bullet M_{ji} = M_{ji} \bullet M_{lk}$$

2) Associative

$$M_{lk} \bullet (M_{hg} \bullet M_{ji}) = (M_{lk} \bullet M_{hg}) \bullet M_{ji}$$

Proof:

$$M_{lk} \bullet M_{ji} = M_{lkji}$$

$$= (b_l + b_k),(b_j + b_i)[0, M_{ji}, M_{lk}, M_{lk} + M_{ji}]$$

$$= \left|\left|2^l\right|_m b_l + \left|2^k\right|_m b_k + \left|2^j\right|_m b_j + \left|2^i\right|_m b_i\right|_m \quad (6)$$

$$M_{ji} \bullet M_{lk} = M_{jilk}$$

$$= (b_j + b_i),(b_l + b_k)[0, M_{kl}, M_{ji}, M_{ji} + M_{lk}]$$

$$= \left|\left|2^j\right|_m b_j + \left|2^i\right|_m b_i + \left|2^l\right|_m b_l + \left|2^k\right|_m b_k\right|_m \quad (7)$$

Both expressions (6) and (7) are the same by commutative property of "+" hence ● operator is commutative

$$M_{lk} \bullet (M_{hg} \bullet M_{ji}) = M_{lk} \bullet M_{hgji}$$
$$= M_{lkhgji}$$
$$= \left\| 2^{l} \right|_m b_k + \left| 2^{k} \right|_m b_k + \left| 2^{h} \right|_m b_h + \left| 2^{g} \right|_m b_g + \left| 2^{j} \right|_m b_j + \left| 2^{i} \right|_m b_i \Big|_m \quad (8)$$

$$(M_{lk} \bullet M_{hg}) \bullet M_{ji} = M_{lkhg} \bullet M_{ji}$$
$$= M_{lkhgji}$$
$$= \left\| 2^{l} \right|_m b_k + \left| 2^{k} \right|_m b_k + \left| 2^{h} \right|_m b_h + \left| 2^{g} \right|_m b_g + \left| 2^{j} \right|_m b_j + \left| 2^{i} \right|_m b_i \Big|_m \quad (9)$$

Both expressions (8) and (9) are the same by associative property of "+" hence ● operator is associative

Theorem 2: For any three pairs of bits (b_l & b_k) , (b_j & b_i) and (b_h & b_g) for l, k, j, i, h & g ≥ 0 with given expression

$$\left\| 2^{l} \right|_m b_l + \left| 2^{k} \right|_m b_k + \left| 2^{j} \right|_m b_j + \left| 2^{i} \right|_m b_i + \left| 2^{h} \right|_m b_h + \left| 2^{g} \right|_m b_g \Big|_m = \left| X_{lkjihg} \right|_m$$

can be implemented using 3-bit multiplexer

$$M_{lkjihg} = (b_l + b_k), (b_j + b_i), (b_h + b_g)[0, M_{hg}, M_{ji}, M_{ji} + M_{hg}, M_{lk}, M_{lk} + M_{hg},$$
$$M_{lk} + M_{ji}, M_{lk} + M_{ji} + M_{hg}] \quad (10)$$

Where control bits (A_2, A_1, A_0) equal (b_l+b_k , b_j+b_i , b_h+b_g,)

Proof:

$$\left| X_{lkjihg} \right|_m = \left\| 2^{l} \right|_m b_l + \left| 2^{k} \right|_m b_k + \left| 2^{j} \right|_m b_j + \left| 2^{i} \right|_m b_i + \left| 2^{h} \right|_m b_h + \left| 2^{g} \right|_m b_g \Big|_m$$

$$\left| X_{lkjihg} \right|_m = \left| M_{lk} + M_{ji} + M_{hg} \right|_m \quad (11)$$

Where $M_{lk} = \left\| 2^{l} \right|_m b_l + \left| 2^{k} \right|_m b_k \Big|_m$,

$M_{ji} = \left\| 2^{j} \right|_m b_j + \left| 2^{i} \right|_m b_i \Big|_m$ and

$M_{hg} = \left\| 2^{h} \right|_m b_h + \left| 2^{g} \right|_m b_g \Big|_m$ by Lemma 1

Let $b_{lk} = (b_l + b_k)$, $b_{ji} = (b_j + b_i)$ and $b_{hg} = (b_h + b_g)$

Rewrite equation (11) as

$$(0.\bar{b}_{lk}.\bar{b}_{ji}.\bar{b}_{hg}) + (M_{hg}.\bar{b}_{lk}\bar{b}_{ji}.b_{hg}) + (M_{ji}.\bar{b}_{lk}.b_{ji}.\bar{b}_{hg}) + ((M_{ji} + M_{hg})\bar{b}_{lk}.b_{ji}.b_{hg}) +$$
$$(M_{lk}.b_{lk}.\bar{b}_{ji}.\bar{b}_{hg}) + ((M_{lk} + M_{hg}).b_{lk}\bar{b}_{ji}.b_{hg}) +$$
$$((M_{lk} + M_{ji}).b_{lk}.b_{ji}.\bar{b}_{hg}) +$$
$$((M_{lk} + M_{ji} + M_{hg})b_{lk}.b_{ji}.b_{hg})$$

This is equivalent to 3-bit multiplexer M_{lkjihg} with control bits (A_2, A_1, A_0) equal (b_l+ b_k , b_j+ b_i , b_h+ b_g)

Figure (5) shows implementation for three pairs of bits (b_5, b_4), (b_3, b_2) & (b_1, b_0)

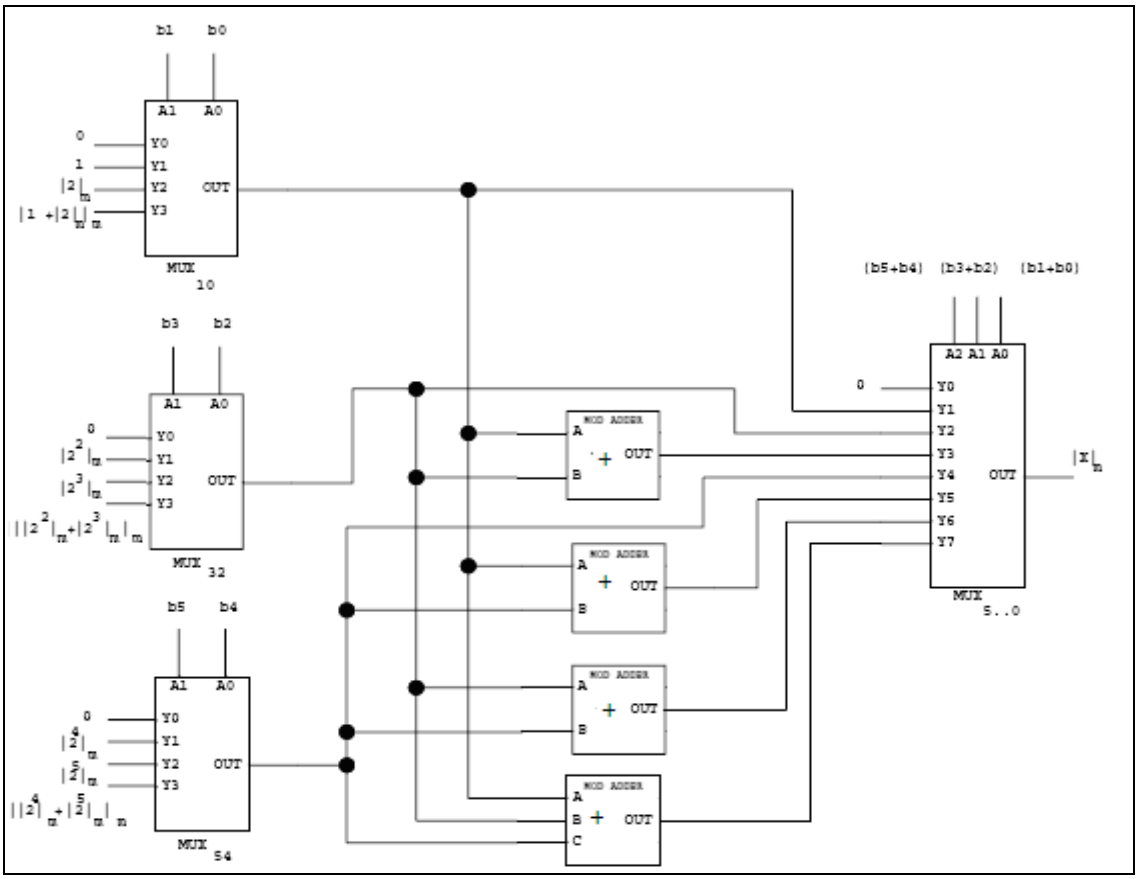


FIGURE 5: Six Bits Binary to Residue Number System Conversion

$$\left|X_{5..0}\right|_m = \left|\left|2^5\right|_m b_5 + \left|2^4\right|_m b_4 + \left|2^3\right|_m b_3 + \left|2^2\right|_m b_2 + \left|2^1\right|_m b_1 + b_0\right|_m$$

Lemma 5:

Combining three pairs of bits (b_l & b_k), (b_j & b_i) & (b_h & b_g) requires one 3- bit multiplexer and three 2-input mod adder and one 3-input mod adder. The delay time equals

$$\tau_{Total} = \tau_{mux_3} + 2\tau_{modadder_2} = \tau_{mux_3} + \tau_{modadder_3}$$

Proof

Equation (10) and figure (5) show that $(b_l + b_k),(b_j + b_i),(b_h + b_g)[0, M_{hg},$ $M_{ji}, M_{ji} + M_{hg}, M_{lk}, M_{lk} + M_{hg}, M_{lk} + M_{ji}, M_{lk} + M_{ji} + M_{hg}]$ is equivalent to one 3-bit multiplexer and three 2-input mod adder and one 3-input mod adder; and delay time is equal to $\tau_{mux_3} + 2\tau_{modadder_2}$

Figure (2d) represents acyclic graph for "■ " for node M_{lkhgji} where $M_{lk,}$ M_{hg} & M_{ji} are inputs

Lemma 6:

The parallel prefix operator ■ has the following properties

1) Commutative

M_{lkjihg} ■ M_{tsrqpo} = M_{tsrqpo} ■ M_{lkjihg}

2) Associative

M_{lkjihg} ■ (M_{tsrqpo} ■ M_{zyxwru}) = (M_{lkjihg} ■ M_{tsrqpo}) ■ M_{zyxwru}

Proof:
The proof is similar to Lemma 4

4. ILLUSTRATIVE EXAMPLE

In this section, we will use illustrate how theorem 1, theorem 2, lemma 1 and lemma 2 can be combined to design a binary to residue convertor. Figure (6) shows how $|X|_m$ for n = 8 is computed.
In the first layer, using pre-processing operator ○ each consecutive pair of bits are group together (b_7, b_6) (b_5 , b_4) (b_3 , b_2) (b_1, b_0) creating nodes M_{76}, M_{54}, M_{32}, M_{10} . In the second layer, using parallel prefix operator ● each consecutive M node are combined (M_{76}, M_{54}) (M_{32}, M_{10}) forming nodes $M_{7..4}$ & $M_{3..0.}$ In the last layer, using parallel prefix operator ● the last 2 M nodes are combined ($M_{7..4}$, $M_{3..0}$) forming node $M_{7..0}$ = $|X_{7..0}|_m$. Figure (7a) shows the actual hardware implementation.

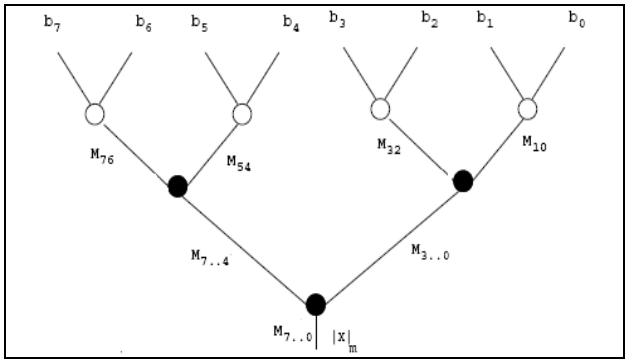


FIGURE 6: Prefix Structure of 8 Bits Binary to RNS

Total delay time for this example is calculated by counting the delay introduce by the operator in each layer
by using lemma 3 as follows

Layer 1: delay time is τ_{mux_2}

pre-processing operator ○ doesn't requires an adder

Layer 2 : delay time is $\tau_{mux_2} + \tau_{modadder_2}$

Layer 3: delay time is $\tau_{mux_2} + \tau_{modadder_2}$

Total delay is the sum of all layers delay time $\tau_{Total} = 3\tau_{mux_2} + 2\tau_{modadder_2}$

To show that hardware works, the signal propagation for binary number $|X|_7 = |(1110 \quad 0110)_2|_7 = |244_{10}|_7 = 6$ is illustrated in figure (7b). Similarly, the reader can try any bit pattern in figure (7b) to check the validity of the design. For example $|X|_7 = |(1111 \quad 1111)_2|_7 = |255_{10}|_7 = 3$ where each multiplexer select line is 3 and the selected output are shown in parenthesis.

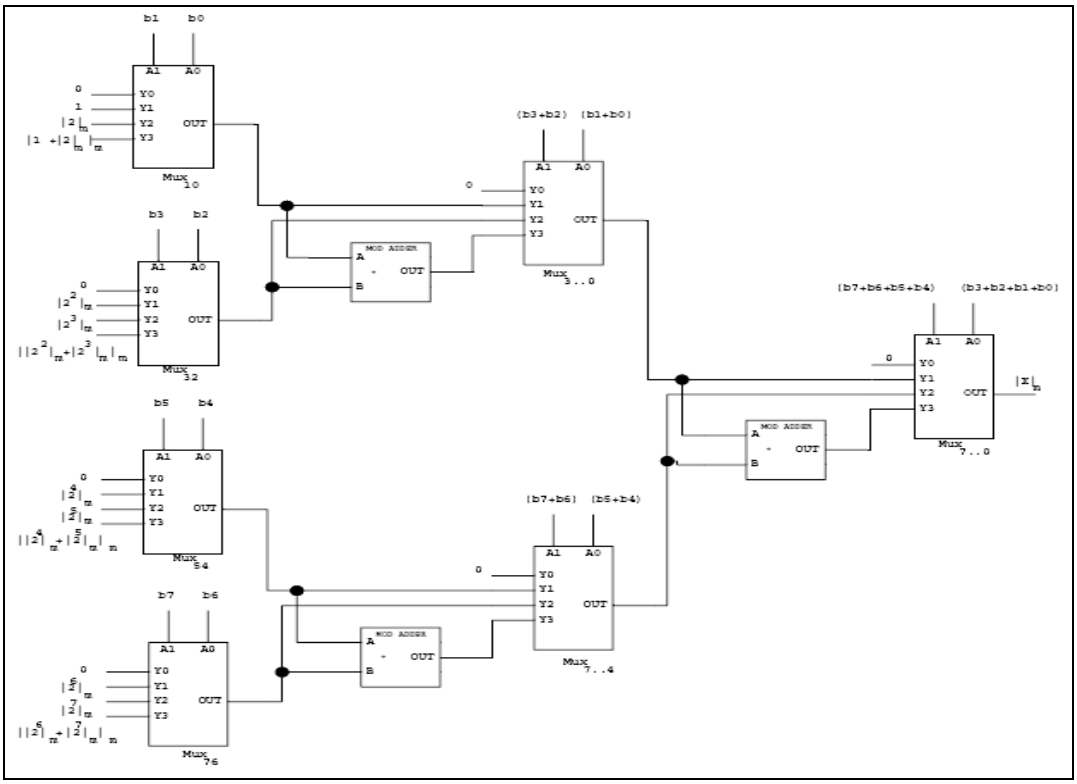


FIGURE 7A: Eight Bits Binary to Residue Number System Conversion

$$\left|X_{7..0}\right|_m = \left|\left|2^7\right|_m b_7 + \left|2^6\right|_m b_6 + \left|2^5\right|_m b_5 + \left|2^4\right|_m b_4 + \left|2^3\right|_m b_3 + \left|2^2\right|_m b_2 + \left|2\right|_m b_1 + b_0\right|_m$$

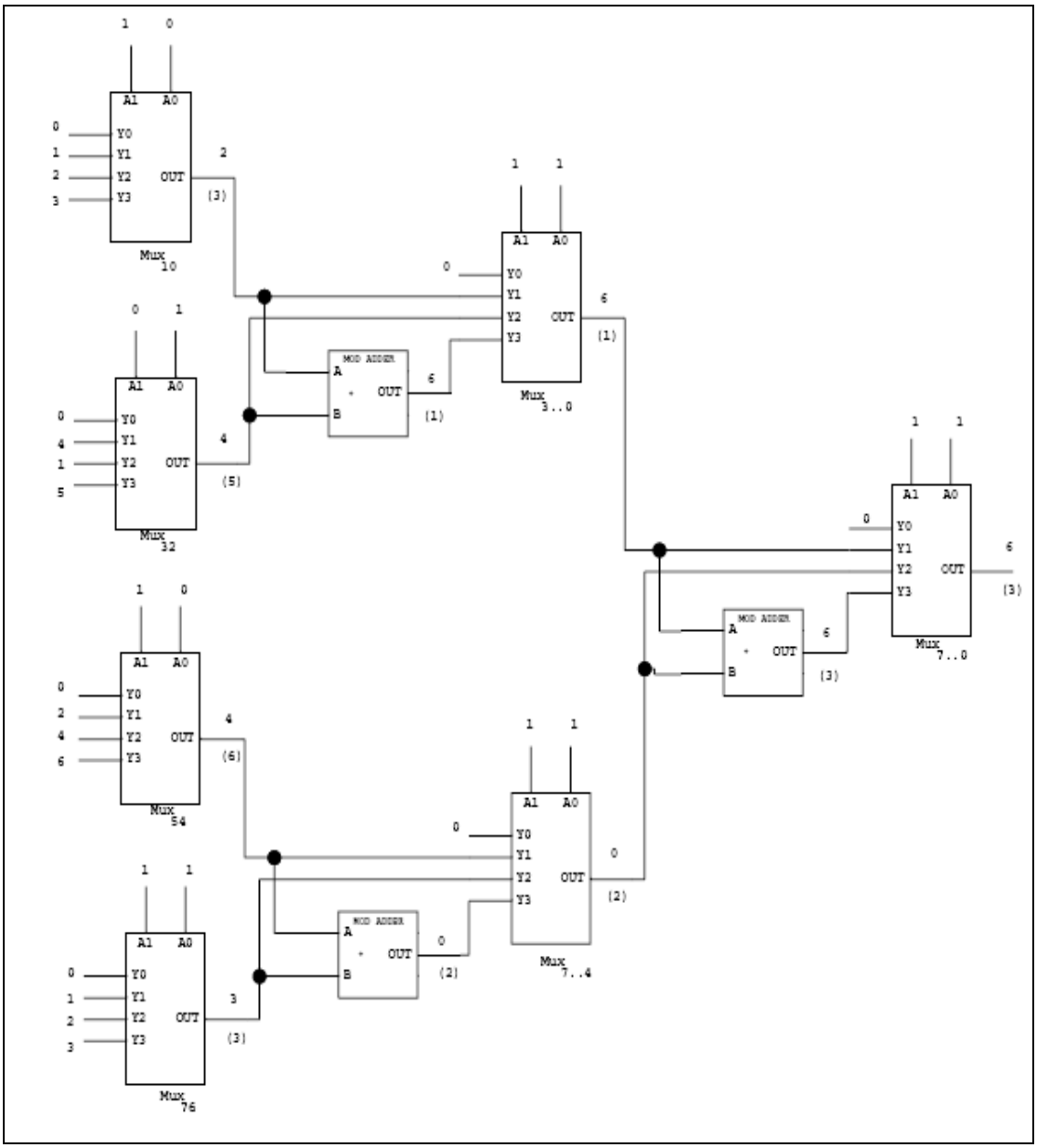


FIGURE 7B: Example for Signal propagation of $\left|(1110 \quad 0110)_2\right|_7$ and $\left|(1111 \quad 1111)_2\right|_7$

5. IMPLEMENTATION SELECTION

There are several possible binary to RNS imp- lementations using a combination of 2-bit and 3-bit multiplexers. Figure (8) shows three different imp- lementations (design 1, design 2 and design 3) for 10 bits binary to residue conversion system.

To simplify comparison, the following reasonable assumptions are made

$\tau_{\text{mux}_2} = \tau_{\text{mux}_3}$; $\tau_{\text{modadder}_3} = 2\tau_{\text{modadder}_2}$

Design 1 uses nine 2-bit multiplexers and four 2-input mod adders with

Layer 1: delay time is τ_{mux_2}

Layer 2: delay time is $\tau_{\text{mux}_2} + \tau_{\text{modadder}_2}$

Layer 3: delay time is $\tau_{mux_2} + \tau_{modadder_2}$

Layer 4: delay time is $\tau_{mux_2} + \tau_{modadder_2}$

Total delay is sum of all layers delay time $\tau_{Total} = 4\tau_{mux_2} + 3\tau_{modadder_2}$

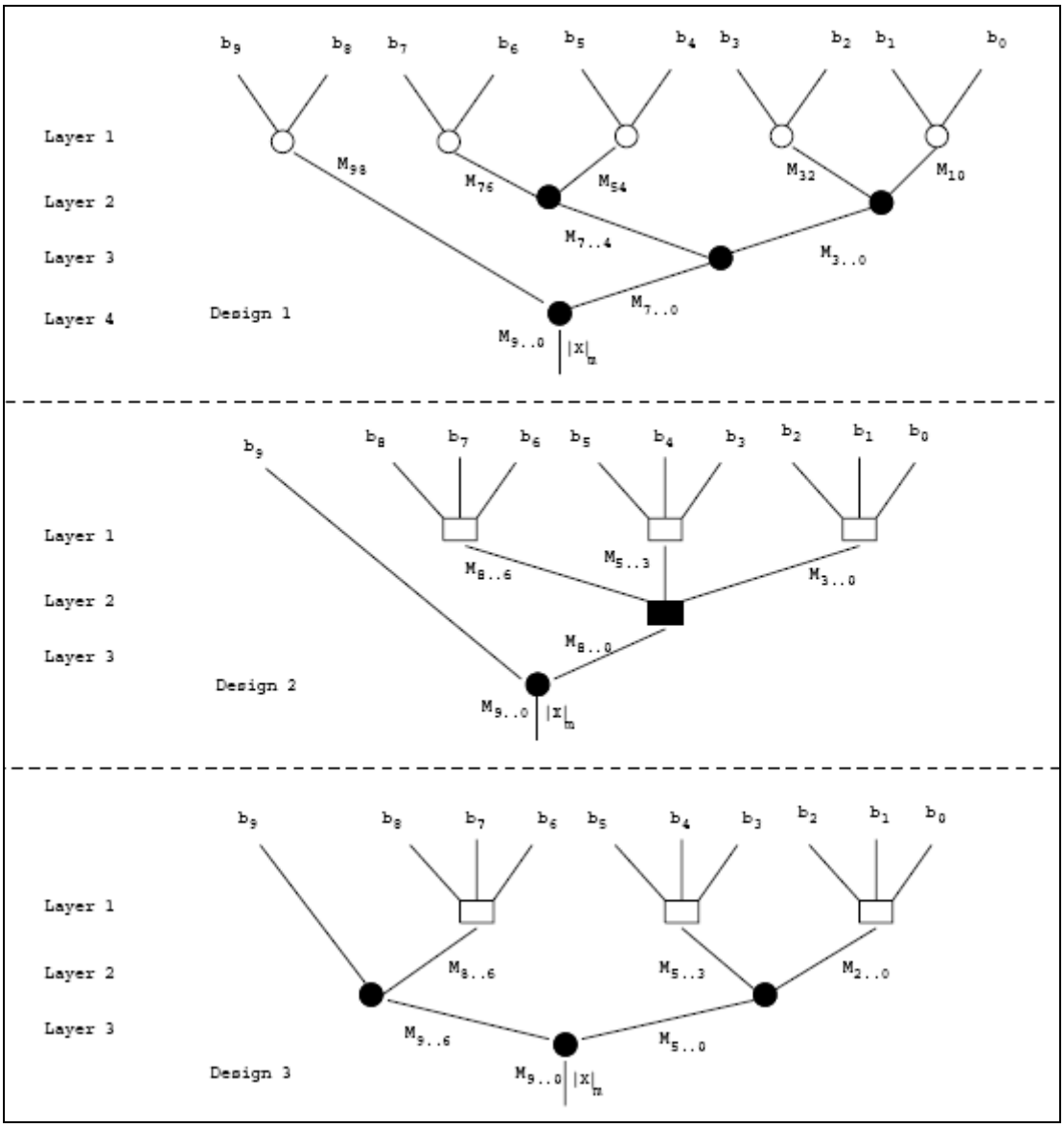


FIGURE 8: Prefix Structure of 10 Bits Binary to RNS

$$|X_{9..0}|_m = \left| |2^9|_m b_9 + |2^8|_m b_8 + |2^7|_m b_7 + |2^6|_m b_6 + |2^5|_m b_5 + |2^4|_m b_4 + |2^3|_m b_3 + |2^2|_m b_2 + |2|_m b_1 + b_0 \right|_m$$

Design 2 uses four 3-bit multiplexers, one 2-bit multiplexers, four 2-input mod adders and one 3-input adder with

Layer 1: delay time is τ_{mux_3}

Layer 2: delay time is $\tau_{mux_3} + 2\tau_{modadder_2}$

Layer 3: delay time is $\tau_{mux_2} + \tau_{modadder_2}$

$$\tau_{Total} = \tau_{mux_2} + 2\tau_{mux_3} + 3\tau_{modadder_2}$$
$$= 3\tau_{mux_2} + 3\tau_{modadder_2}$$

Design 3 uses three 3-bit multiplexers, three 2-bit multiplexer and three 2-input mod adders with

Layer 1: delay time is τ_{mux_3}

Layer 2 : delay time is $\tau_{mux_2} + \tau_{modadder_2}$

Layer 3: delay time is $\tau_{mux_2} + \tau_{modadder_2}$

$$\tau_{Total} = 2\tau_{mux_2} + \tau_{mux_3} + 2\tau_{modadder_2}$$
$$= 3\tau_{mux_2} + 2\tau_{modadder_2}$$

From Table (1), it shows that Design 3 uses less hardware and the fastest

Design		Hardware count			
#	Time Delay	Mux_2	Mux_3	Mod add_2	Mod add_3
1	$4\tau_{mux_2} + 3\tau_{modadder_2}$	9	0	4	0
2	$3\tau_{mux_2} + 3\tau_{modadder_2}$	1	4	4	1
3	$3\tau_{mux_2} + 2\tau_{modadder_2}$	3	3	3	0

TABLE 1: shows comparison between the three designs implementation for n = 10

6. COMPARISON SELECTION TO PERVIOUS WORK

This Novel method has hardware advantages than any competitive converters. In 1984, Alia and Martinelli [3] published binary to RNS conversion design based on power 2 mod m_i . The design uses processing elements (PE) and each PE is associated with two registers. Each of these registers is serially loaded with $\left|2^i\right|_m$ and $\left|2^{i+1}\right|_m$ respectively and it enabled to put their content or zeros' out depending on value 1 or 0 of b_j and b_{j+1} respectively. The two outputs are added in a modular adder. Thus, at first level, n/2 PEs are required. The number of stages in this method is $[\log_2 n]$ and after successive transformation and addition, the residue result is available. Cappocelli and Giancarlo [4] suggested the use of t PEs where $t = n/\log_2 n$, each PE computing the residue corresponding to k- bit binary word where $k = \log_2 n$, the residue 2^{kt} mod m_i is serially fed to $\hat{k}$th PE ($\hat{k} = 0, 1, 2, \ldots, t-1$), Based on these initial residues, the residues corresponding to the next (k-1) powers are computed by first doubling and then weighting according to the input bits in each PE. The partial residues of k-bit words computed over parallel t PEs are then added to yield the final residue. Anandmohan [5] has proposed a similar method but with a difference that X is divided into t sections based on the cyclic property of 2^j mod m_i. Using the fact that, 2^j, 2^{j+lo} and 2^{j+2lo} have the same residues due to periodicity of period *lo* , *lo* bits are first added. The width of the result is confined to *lo* bits by adding the carry bit resulting from previous addition to LSB of the result. The residue results is then determined by using methods given in [3]

7. CONCLUSIONS

In this paper we presented a new novel binary to Residue conversion method that eliminates the need for processing elements (PE) as the above competitive converter designs and doesn't use table lookup as in Behrooa Parhami [6]. The new method that we present here is based on multiplexers concept which makes it practical and suitable for VLSI implementation

8. REFERENCES

[1] M.A. Bayoumi, "Digital filter VLSI systolic arrays over finite fields for DSP applications," in Proc. 6th IEEE annual Phoenix Conf. Computers and Communications, pp 194-199, Feb 1987.

[2] M. A. Soderstrand et al., Eds., "Residue Number System Arithmetic: Modern Applications in Digital Signal Processing" New York: IEEE Press, 1986

[3] K. Konstantinides and V. Bhaskaran, “Monolithic architectures for image processing and compression,”IEEE Computer Graphics & Applications, pp. 75-86, Nov. 1992

[4] G. Alia and E. Martinelli, "A VLSI algorithm for direct and reverse conversion from weighted binary number to residue number system," IEEE Trans. Circuits Syst., vol. 31, pp. 1425–1431, Dec. 1984.

[5] R. M. Capocelli and R. Giancarlo, "Efficient VLSI networks for converting an integer from binary system to residue number system and vice versa," IEEE Trans. Circuits Syst., vol. 35, pp. 1425–1431, Nov. 1988.

[6] A. Mohan, “Novel design for binary to RNS converters,” in Proc. Int.Symp. Circuits and Systems, London, U.K., 1994, pp. 357–360.

[7] Behrooa Parhami, "Optimal Table-Lookup Schemes for Binary-to-Residue and Residue-to-Binary Conversions," IEEE Trans, Circuits Syst., 1993

[8] Mohamed. Akkal and Pepe Siy, "A new Mixed Radix Conversion algorithm", Journal of Systems Architecture, Volume 5, Iusse 9, September 2007, Pages 577-586

[9] N. S. Szabo and R. I. Tanaka, " Residue Arithmetic and Its Applications to Computer Technology". New York: McGraw Hill, 1967.

Design and Simulation of Moore Logic Circuit based SAR Analog to Digital Converter

Osama Q.J Al-Thahab
Engineering Faculity/Electgrical Department *oalthahab@yahoo.com*
University of Babylon
Babylon, Iraq

Hanaa mohsin ali
Engineering Faculity/Electgrical Department *ahanaamohsin@yahoo.com*
University of Babylon
Babylon, Iraq

Abstract

In this work the circuit of 4 bit successive approximation Register Analog to digital converter is designed and simulated by using Multisim 11 program. Here the idea is to design a circuit that give low power consumption. The proposed circuit is designed with the sequential synchronous logic circuit idea by using Moore theory for the first time, so the state diagram is built and then implemented by using J-k flip flop, the proposed circuit is tested by DC and AC input voltages, so that the maximum voltage is 5 v and minimum voltage is 0 v, and so the step voltage will be 0.3125 v. SAR ADC needs Digital to Analog circuit, Latch circuit, and comparator which compare between the input voltage and the voltage results from the DAC, and all these circuits were built in this work with all they needs.

Keywords: ADC, SAR ADC, Logic circuit, Moore Sequential Model, Karnaugh Map.

1. Introduction

Over the past two decades, silicon integrated circuit (IC) technology has evolved so much and so quickly that the number of transistors per square millimeter has almost doubled in every eighteen months. Since the minimum channel length of transistors has been shrunk, transistors have also become faster. The evolution of IC technology has been driven mostly by the industry in digital circuits such as microprocessors and memories. As IC fabrication technology has advanced, more analog signal processing functions have been replaced by digital blocks. Despite this trend, analog-to-digital converters (ADCs) retain an important role in most modern electronic systems because most signals of interest are analog in nature and must to be converted to digital signals for further signal processing in the digital domain [1].

Analog-to-digital converters (ADC's) are often used to collect data from sensors, such as touchscreens, thermometers, camera image sensors and battery meters. To do this, the ADC measures the voltage or current input, and outputs a string of bits, which represents the input voltage or current. There are many applications for analog-to-digital converters, ranging from sensors, audio and data acquisition systems to video, radar and communications interfaces. The applications that require the highest sample-rates in the ADC are typically found in video, radar and communication areas [2,3].

2. Types of ADC

2.1 Flash ADC

Flash ADC conversion is the fastest possible way to quantize an analog signal. In order to achieve N-bit from a flash ADC, it requires (2^N-1) comparators, (2^N-1) reference levels and digital encoding circuits. The reference levels of comparators are usually generated by a resistor string. The high sensitivity of the comparator offset and a large circuit area are the main drawbacks of a flash ADC. For instance, to build a 10-bit ADC based on flash architecture requires more than 1,023 comparators. Therefore, it will occupy a very large chip

area and dissipate high power. Moreover, each comparator must have an offset voltage smaller than $1/2^{10}$, which is quite difficult to build [1].N-bit flash ADC is shown in FIGURE1. Here a resistive ladder drives a bank of (2^N-1) comparators, whose Logic encoded output is converted to a binary code with a binary encoder [4].

2.2 Digital ramp ADC

Also known as the stair step-ramp, or simply counter A/D converter, The basic idea is to connect the output of a free-running binary counter to the input of a DAC, then compare the analog output of the DAC with the analog input signal to be digitized and use the comparator's output to tell the counter when to stop counting and reset. The basic idea can be shown in FIGURE2-a, and the output can be seen in FIGURE 2-b.

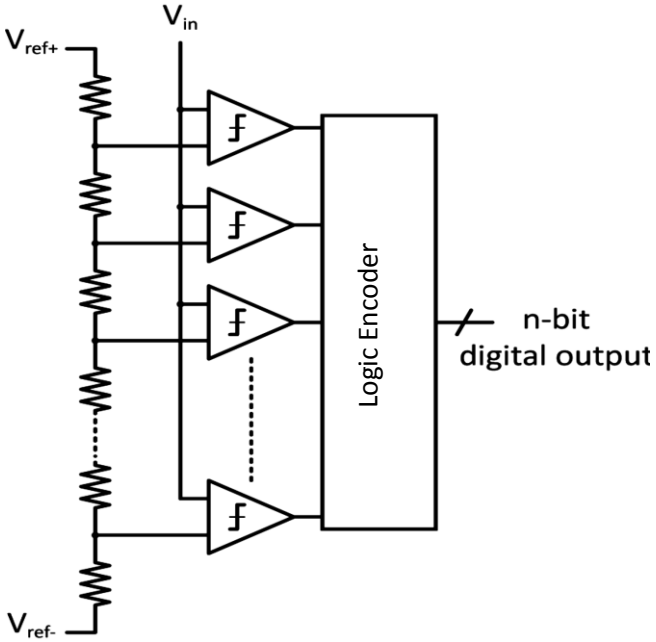
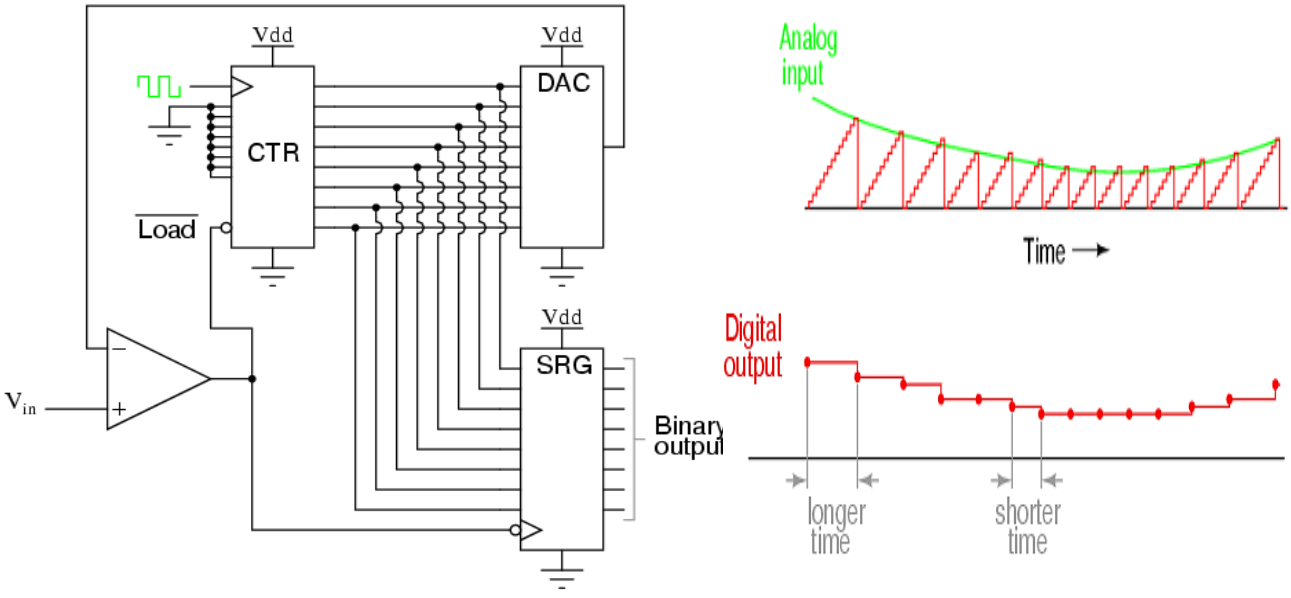


FIGURE1: N-bit flash ADC



(a) (b)

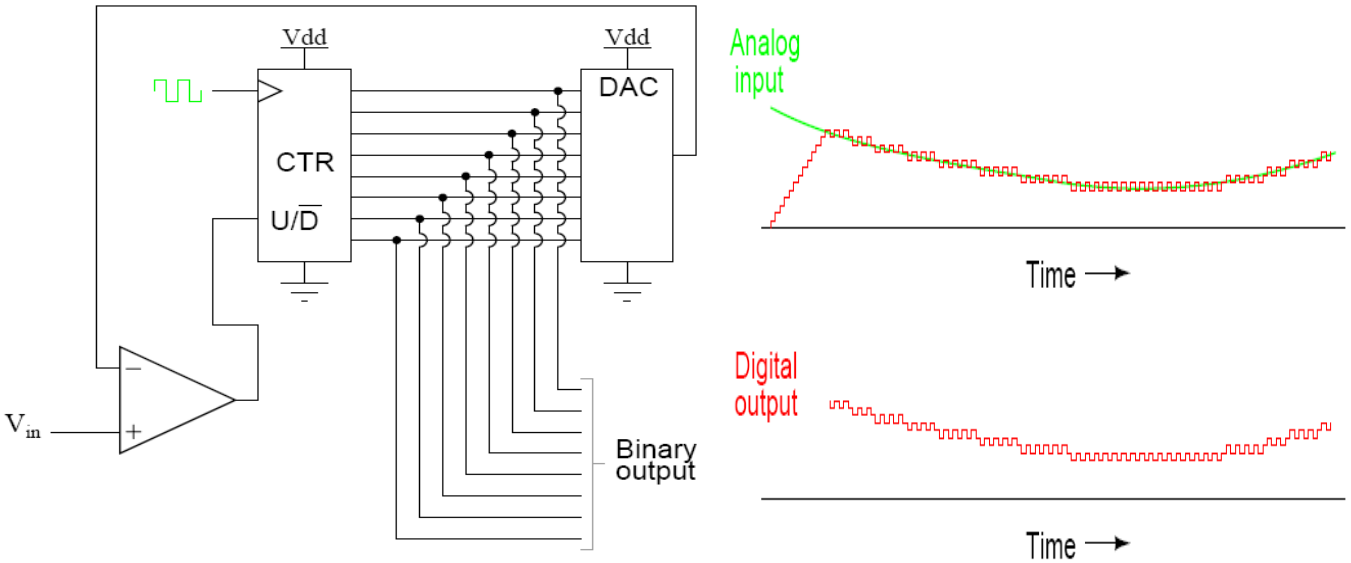
FIGURE2: a- Counter ADC. b- The outputs of counter ADC.

Note how the time between updates (new digital output values) changes depending on how high the input voltage is. For low signal levels, the updates are rather close-spaced. For higher signal levels, they are spaced further apart in time as shown in FIGURE2-b. For many ADC applications, this variation in update frequency (sample time) would not be acceptable. This, and the fact that the circuit's need to count all the way from 0 at the beginning of each count cycle makes for relatively slow sampling of the analog signal, places the digital-ramp ADC at a disadvantage to other counter strategies [5].

2.3 Tracking ADC

Here instead of a regular "up" counter driving the DAC, this circuit uses an up/down counter. The counter is continuously clocked, and the up/down control line is driven by the output of the comparator. So, when the analog input signal exceeds the DAC output, the counter goes into the "count up" mode. When the DAC output exceeds the analog input, the counter switches into the "count down" mode. Either way, the DAC output always counts in the proper direction to track the input signal. The circuit and output are shown in FIGURE3-a and 3-b.

Note that the much faster update time than any of the other "counting" ADC circuits. Also note how at the very beginning of the plot where the counter had to "catch up" with the analog signal, the rate of change for the output was identical to that of the first counting ADC. Also, with no shift register in this circuit, the binary output would actually ramp up rather than jump from zero to an accurate. Perhaps the greatest drawback to this ADC design is the fact that the binary output is never stable: it always switches between counts with every clock pulse, even with a perfectly stable analog input signal. This phenomenon is informally known as bit bobble, and it can be problematic in some digital systems [5].



(a) (b)

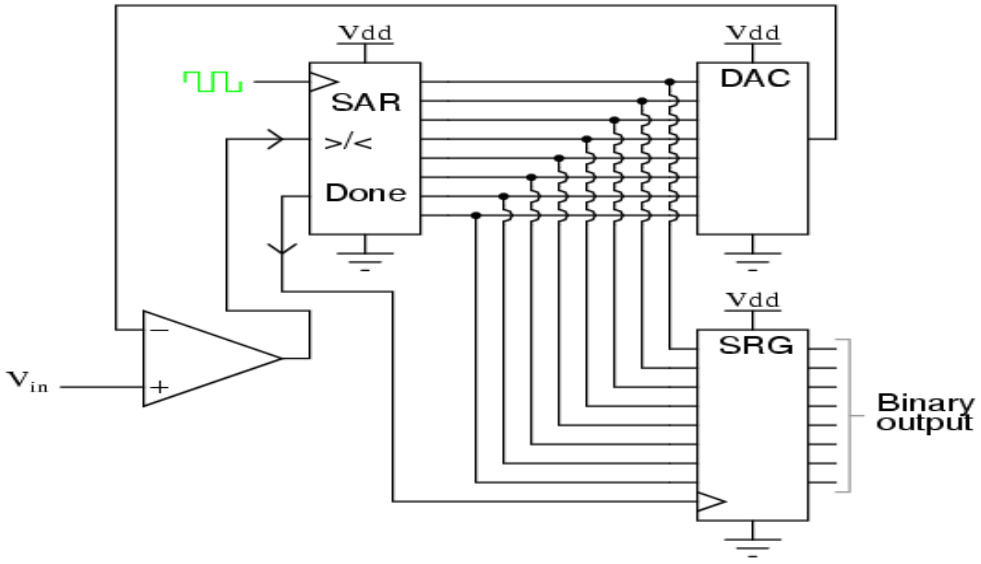
FIGURE3: a- Tracking ADC. b- The outputs of Tracking ADC

2.4 Successive Approximation ADC

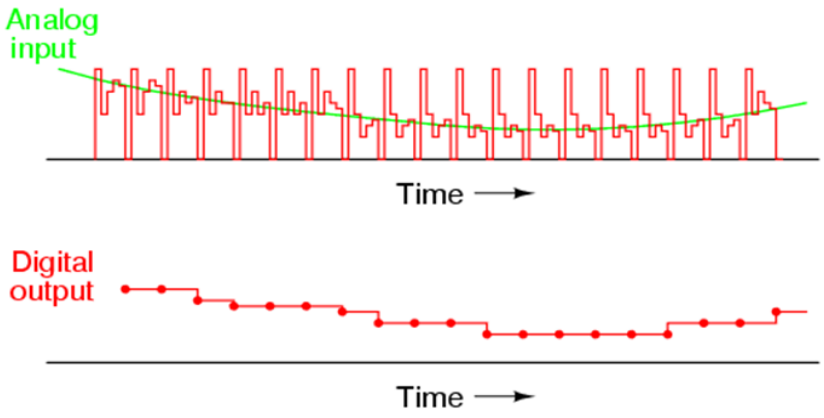
Successive approximation register analog-to-digital converters are common for low power A/D-conversion. This makes them suitable for wireless systems where power consumption limits operation time [6]. The SAR architecture mainly uses the binary search algorithm. The SAR ADC consists of fewer blocks such as one comparator, one DAC and one control logic. The main advantage of SAR ADC is good ratio of speed to power. The SAR ADC has compact design compare to flash ADC, which makes SAR ADC inexpensive. The physical limitation of SAR ADC is it has one comparator throughout the entire conversation process. If

there is any offset error in the comparator, it will reflect on the all conversion bits so you must have a suitable comparator [7].

One method of addressing the digital ramp ADC's shortcomings is the so-called successive-approximation ADC. The only change in this design is a very special counter circuit known as a successive-approximation register. Here the register Instead of counting up in binary sequence, this register counts by trying all values of bits starting with the most-significant bit and finishing at the least-significant bit. Throughout the count process, the register monitors the comparator's output to see if the binary count is less than or greater than the analog signal input, adjusting the bit values accordingly. The circuit, and the outputs are shown in FIGURE4-a, and FIGURE4-b.



(a)



(b)

FIGURE4: a- SAR ADC circuit. b- The operation of SAR ADC

It should be noted that the SAR is generally capable of outputting the binary number in serial (one bit at a time) format, thus eliminating the need for a shift register. From FIGURE 4-b, it can be noted how the updates for this ADC occur at regular intervals, unlike the digital ramp ADC circuit [5].

3. proposed Moore SAR Logic circuit

In general, sequential circuits can be classified into two types: (1) those in which the output or outputs depend only on the present internal state (called Moore circuits) so the outputs can be taken directly from the output of flip flops (y= Q), and (2) those in which the output or outputs depend on both the present state and the input or inputs (called Mealy circuits) [8,9,10].

In this work the SAR is designed by using Moore sequential circuit and implemented by J-K Flip Flop for the first time, and since the SAR operation can be represented in the tree as seen in FIGURE5, then the state diagram can be shown in FIGURE 6. Here Em represents the Analog desire voltage which wanted to be digitize, and ea represents the voltage after DAC.

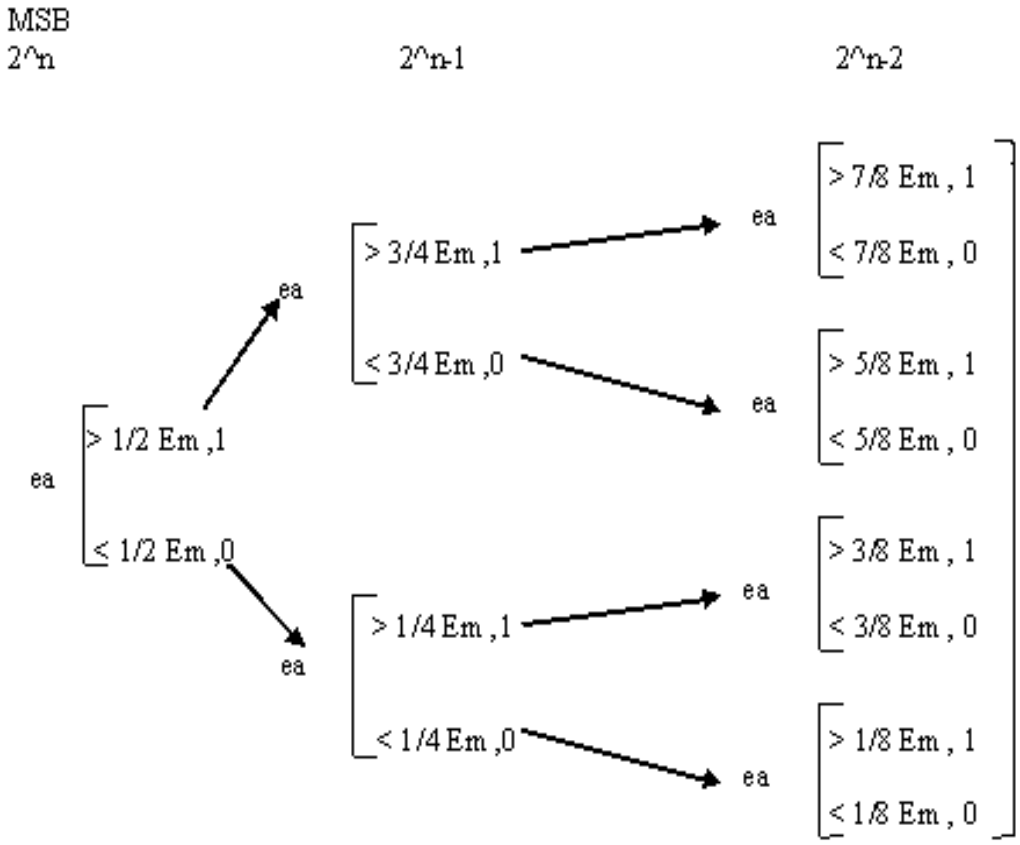


FIGURE 5: The operation of SAR

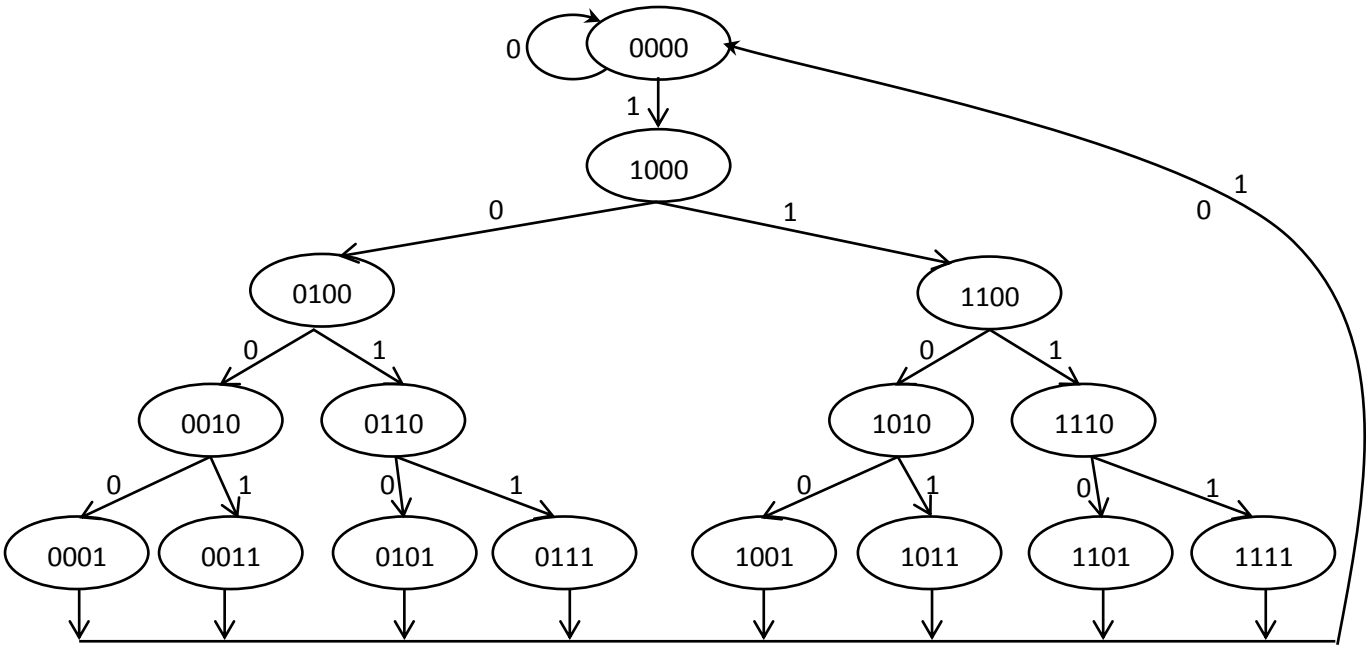


FIGURE 6: The state diagram of Moore SAR .

Then the state table can be shown in TABLE 1, and the J-K excitation table can be shown in TABLE 2.

previous State				Next State	
y4	y3	y2	y1	X= 0	X= 1
0	0	0	0	0000	1000
1	0	0	0	0100	1100
1	1	0	0	1010	1110
0	1	0	0	0010	0110
1	1	1	0	1101	1111
1	0	1	0	1001	1011
0	1	1	0	0101	0111
0	0	1	0	0001	0011
0	0	0	1	0000	0000
0	0	1	1	0000	0000
0	1	0	1	0000	0000
0	1	1	1	0000	0000
1	0	0	1	0000	0000
1	0	1	1	0000	0000
1	1	0	1	0000	0000
1	1	1	1	0000	0000

TABLE 1: State table

Input (X= 0)				Input (X= 1)			
J_4K_4	J_3K_3	J_2K_2	J_1K_1	J_4K_4	J_3K_3	J_2K_2	J_1K_1
0x	0x	0x	0x	1x	0x	0x	0x
x1	1x	0x	0x	x0	1x	0x	0x
x0	x1	1x	0x	x0	x0	1x	0x
0x	x1	1x	0x	0x	x0	1x	0x
x0	x0	x1	1x	x0	x0	x0	1x
x0	0x	x1	1x	x0	0x	x0	1x
0x	x0	x1	1x	0x	x0	x0	1x
0x	0x	x1	1x	0x	0x	x0	1x
0x	0x	0x	x1	0x	0x	0x	x1
0x	0x	x1	x1	0x	0x	x1	x1
0x	x1	0x	x1	0x	x1	0x	x1
0x	x1	x1	x1	0x	x1	x1	x1
x1	0x	0x	x1	x1	0x	0x	x1
x1	0x	x1	x1	x1	0x	x1	x1
x1	x1	0x	x1	x1	x1	0x	x1
x1	x1	x1	x1	x1	x1	x1	x1

TABLE 2: J-K excitation table

For simplification you can use a five bit Karnaugh Map as shown in TABLE 3 where the output of J_1 is determined.

y4y3 \ y2y1X	000	001	011	010	110	111	101	100
00			x	x	x	x	1	1
01			x	x	x	x	1	1
11			x	x	x	x	1	1
10			x	x	x	x	1	1

TABLE 3: Karnaugh map of J1

From TABLE 3, it can be seen that J_1= y3, and so on for all J-k's in all Flip-Flops. So the SAR J-K Flip-Flop hole circuit can be shown in FIGURE 8-a, and its block in FIGURE 8-b, here y1 represent the LSB, and y4 represent the MSB.

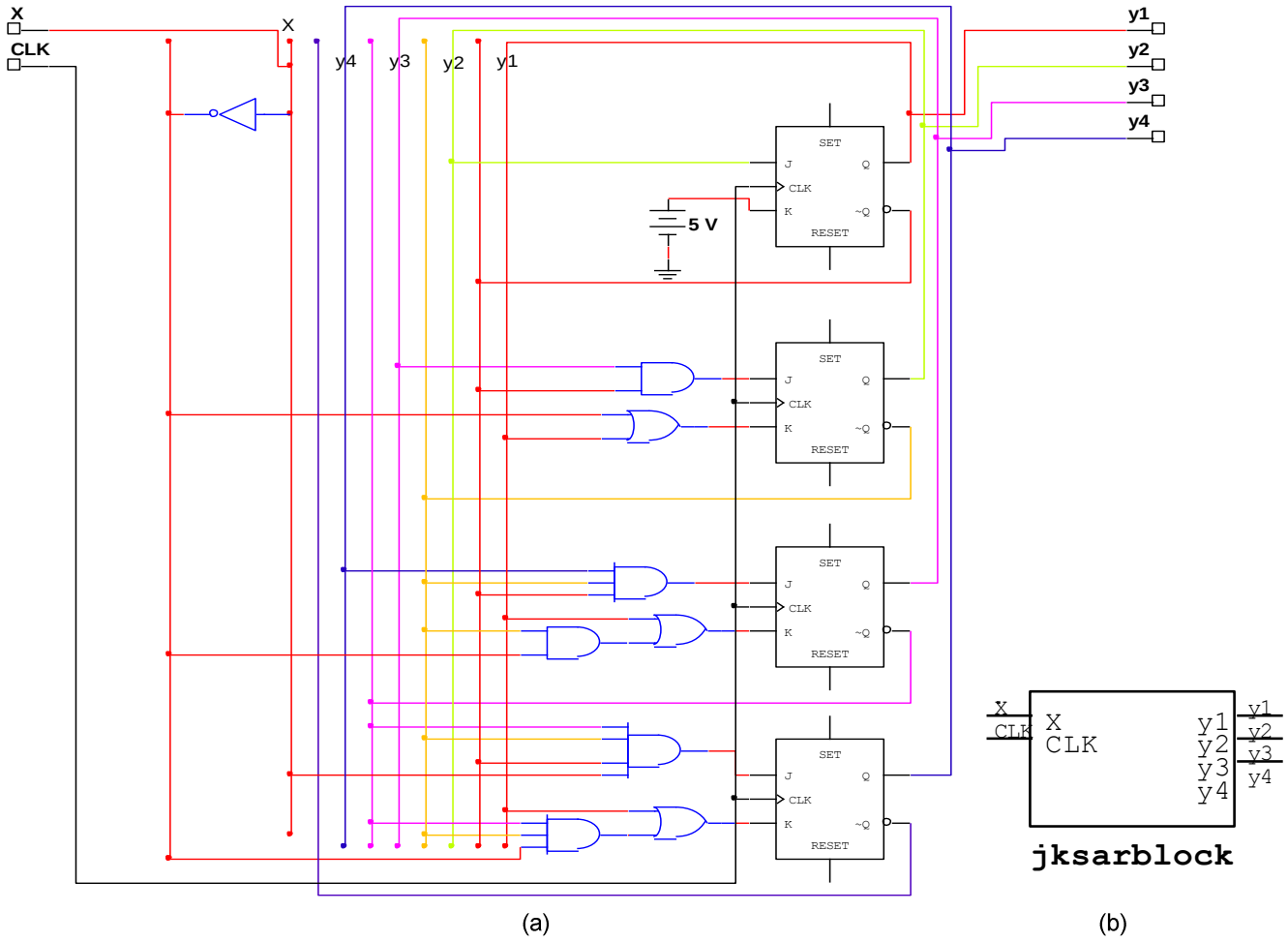


FIGURE 8: a- SAR J-K Flip-Flop. b- Blok of it.

4. Digital To Analog Circuit

The circuit of DAC is shown in FIGURE 9-a with its Block in 9-b, this type is called R/2R (Ladder) DAC which uses fewer unique resistor values. In this work, 5 v is taken as reference voltage, so that $(0000)_2$= 0 v, and $(1111)_2$= 5 v. The step size and output voltage are seen in equation (1), and (2), and since n= 4, then the step voltage (Vs) is 0.3125 v. The output voltage for $(1111)_2$ is equal to 4.6785 v, and by using TTL Logic circuit then VD= 5 v. Here R= 1kΩ, and Ref is equal to 1kΩ (from equation (2)).

$$V_s = V_{ref}/2^n \quad \ldots(1)$$

$$V_o = \frac{Ref}{R} VD\left(\frac{Do}{16} + \frac{D1}{8} + \frac{D2}{4} + \frac{D3}{2}\right) \quad \ldots(2)$$

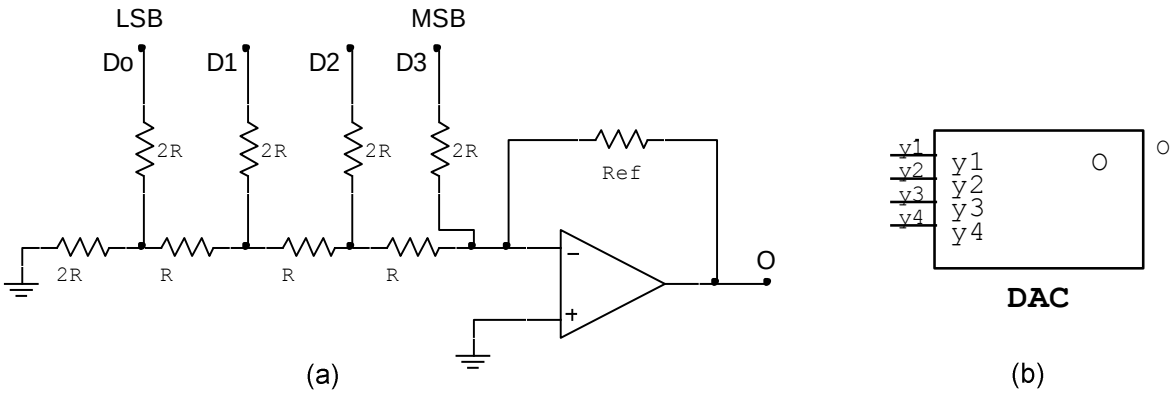


FIGURE 9: a-The circuit of DAC. b- The Block of DAC

5. Comparator

The 741 operational amplifier was used to compare the pervious state of ADC with the input signal . If the input signal is more than pervious state of ADC the OP-Amp give logic-1 ,else; give logic-0.

6. Clock

The 555 multivibrator was used to provide a clock to the circuit as shown in FIGURE (10). Here the two resistance are equal to each other in order to make the duty cycle equal to 50%.

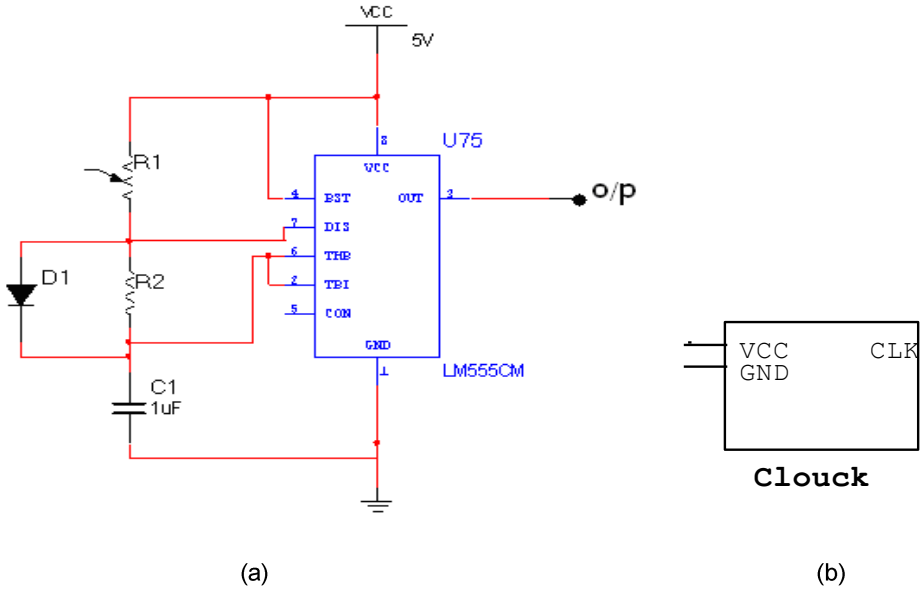


FIGURE 10: a- Block of Clock. b-CLK circuit

7. JK-Counter

In order to view the final result ; the mod-5, D-counter was designed and give the result throw logic circuit as a clock to the latch as shown in FIGURE 11-a, and its block in FIGURE 11-b.

8. Latch

The circuit of parallel in parallel out latch is used as shown in FIGURE (11-c) and its block is shown in FIGURE (11-d).

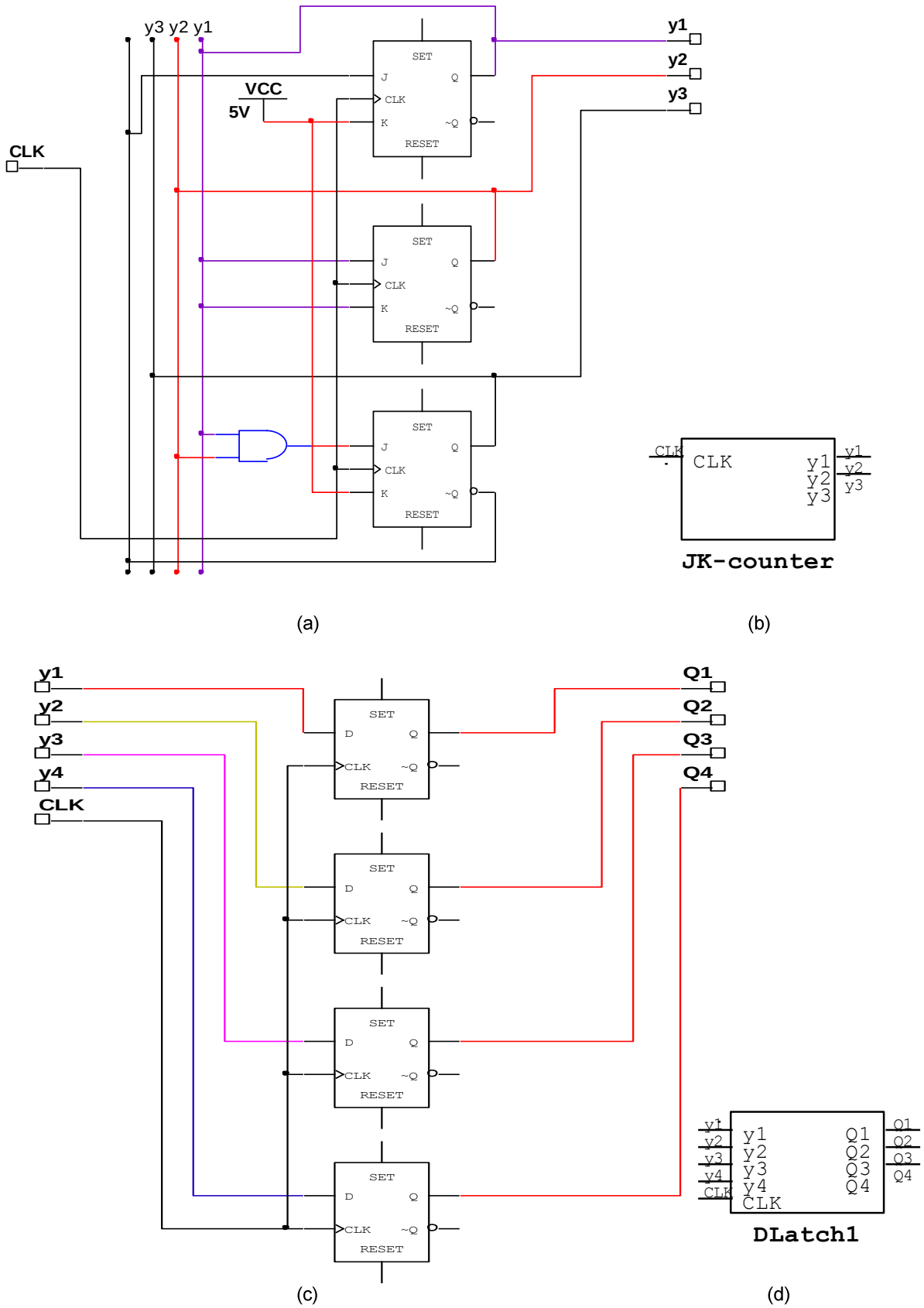


FIGURE 11: a- Jk counter. b- the block of it. c- D-Latch. d- The block of it.

9. Proposed Moore SAR ADC

The Hole circuit is shown in FIGURE 12. The circuit was tested first by DC input voltage, and second by AC input voltage with different voltages (under the range from 0-5 v). Here the results are obtained from Oscilloscope, DCD HEX display, and 4 props.

10. Results and Discussion

First Vref is set to DC voltage Like (3 v) for example as seen in FIGURE13, and the results is seen in FIGURE14.

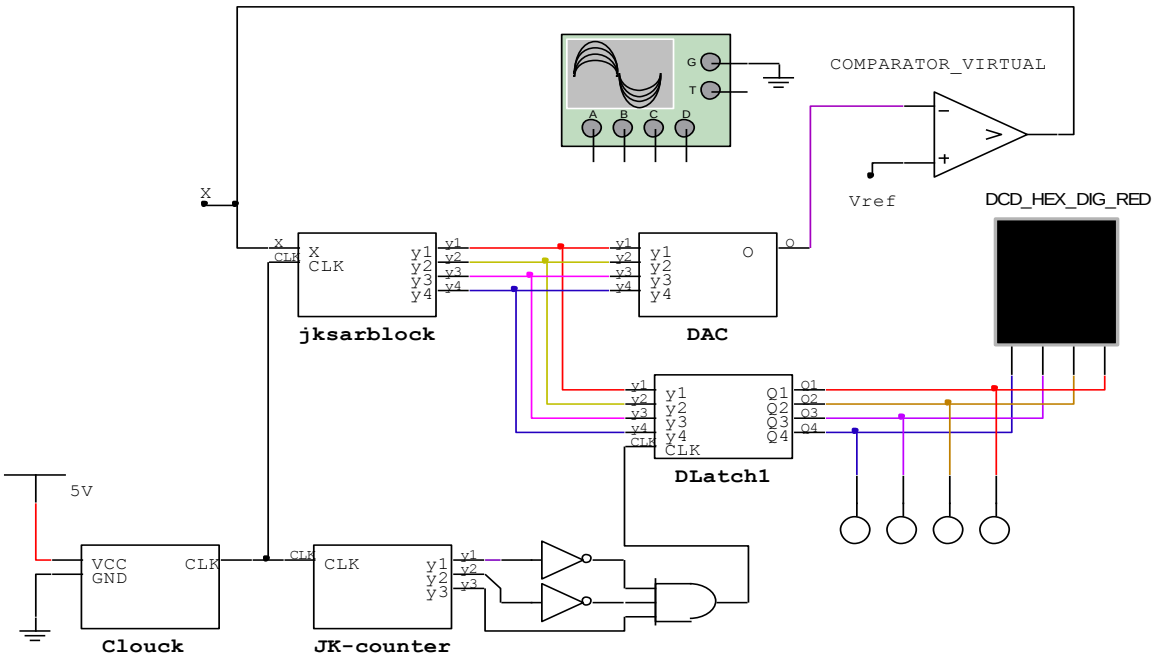


FIGURE 12: Moore SAR ADC circuit

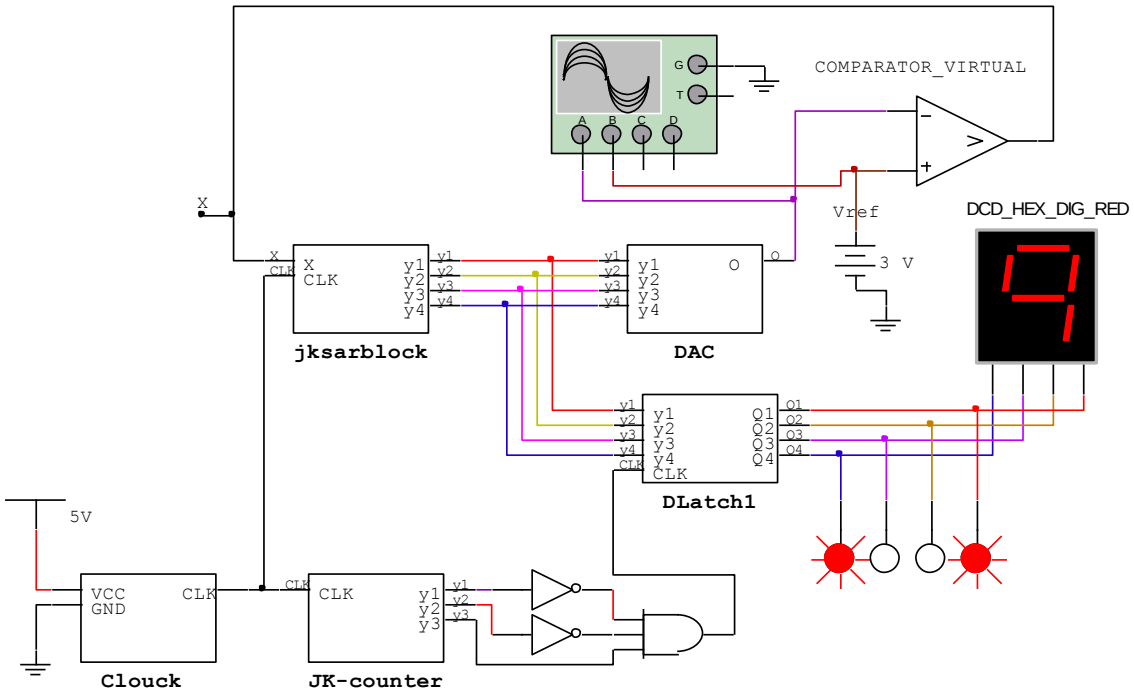


FIGURE 13: DC input voltage to the proposed circuit

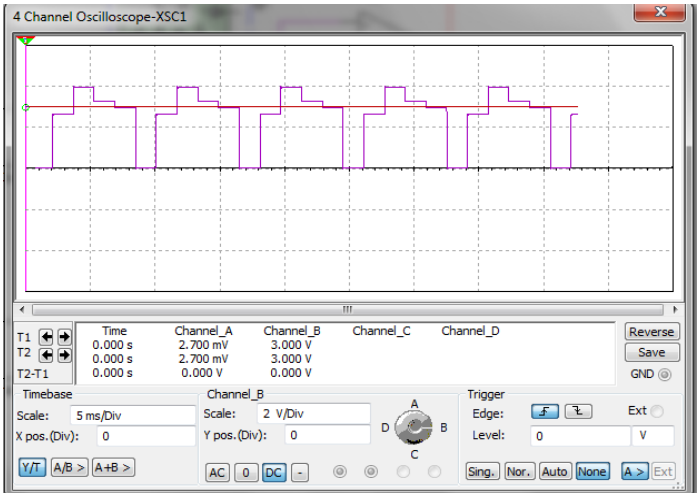


FIGURE 14: response to 3 v input.

You can see from FIGURE 13 that the result from BCD HEX display is (9) which is true since (9*0.3125= 2.8125) so the proposed circuit is indeed work successfully, also from FIGURE 14 it can be shown that the circuit track the input voltage until it reaches to it which exactly satisfy the idea of SAR. Finally the probe response is (1001) which equals to 9 in decimal.

Another tests are done by taking Vref equal to 1.5 v, and 5v, and the results are shown in FIGURE15 and FIGURE 16 respectively.

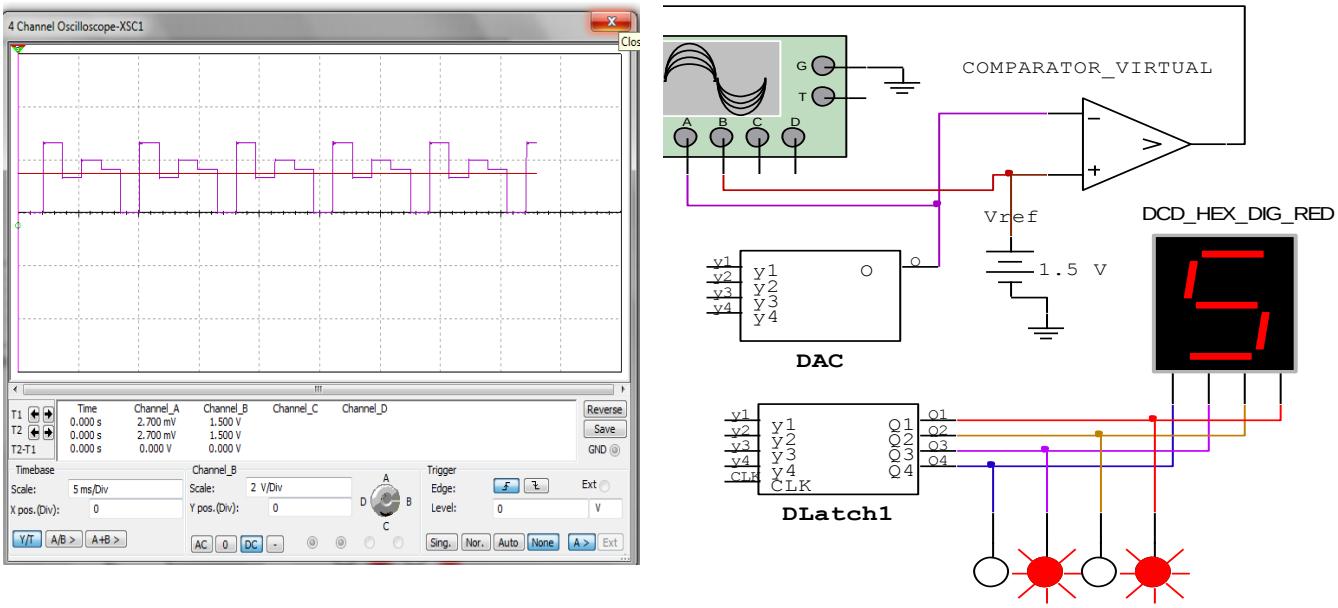


FIGURE 15: the results when Vref= 1.5 v

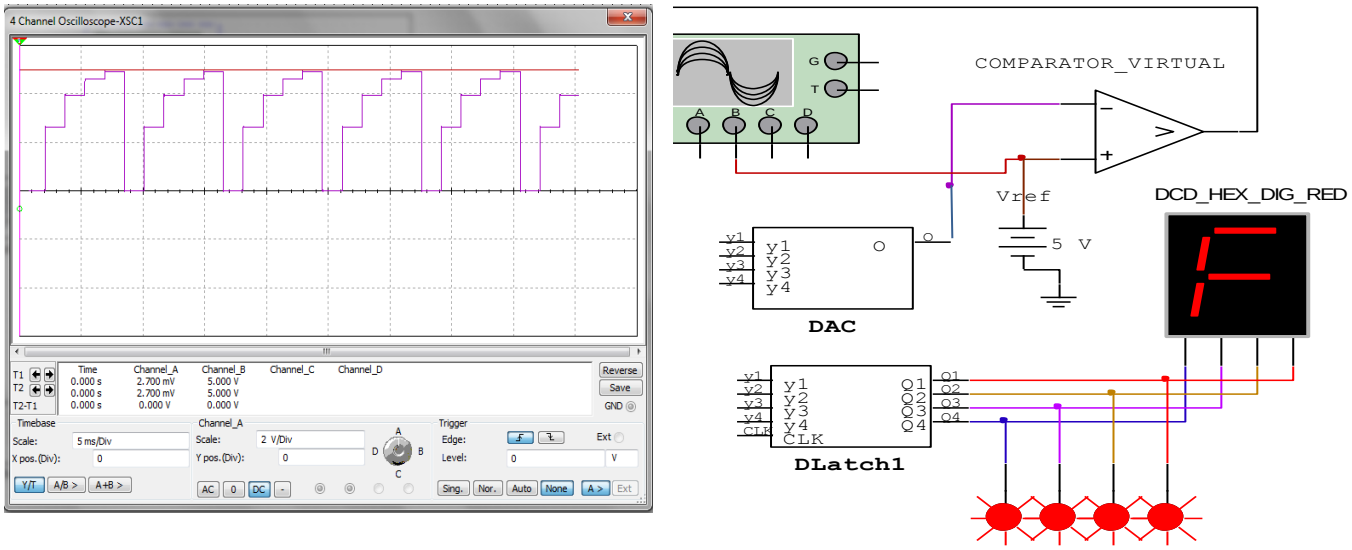


FIGURE 16: the results when Vref= 5 v.

Second test is done by setting Vref to AC voltage like for example Vref= 5sinwt, and the result is shown in FIGURE 17.Offcourse the results seen from the BCD HEX display and the probe are variable with time since Vref is variable. You can see from the reading of OSC. That the output of the proposed circuit is tracking the AC input until it reaches the value of it.

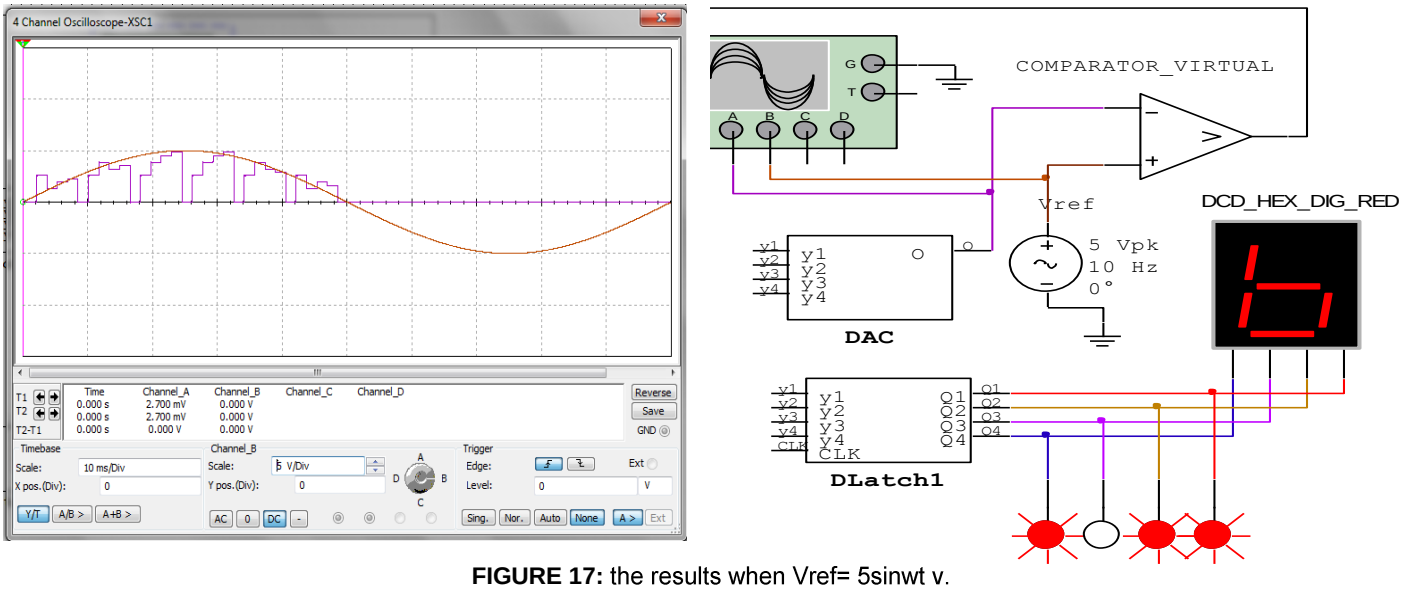


FIGURE 17: the results when Vref= 5sinwt v.

In this work the algorithm is designed with a low consumption power which is 11 m watt from 5 v supply voltage, with a clock frequency up to 8 MHz, and as compared with B. Ginsburg [4] which design an ADC with a 30 m watt of power and also 4 bit resolution, it can be seen that the proposed circuit consume low power, also J. Digel. [6] designs a circuit with 13 m watt consumption power but from 2.6 v source voltage with 6-7 bit and a rate up to 80 Ms/s, whereas the proposed design use 5 v supply voltage and consumes 11 m watt.S. Yan Ng[11] with his team design a 5 bits SAR with 600 MHz circuit and in spite of that it consumes a 30 m watt.

11. Conclusion

In this work a new approach is tested, which is design of Successive Approximation Register ADC with Moore Sequential Logic circuit, and simulated by using JK Flip-Flop with Multisim 11 program. It is found that the design was work successfully and easy to implement and you can digitize any input voltage (here from 0-5 v), offcourse you can increase the input voltage after redesign the DAC, but the step voltage will increase too so the accuracy will be decreased, so the solution is to design 5 or 6 bits SAR ADC where the step voltage will minimize. From the results it is found that the consumption power is 11 m watt with a clock rate up to 8 MHz so it consumes low power but with low bit accuracy, also the proposed circuit is worked with both DC and AC voltage, and it is very fast, accurate and implemented with a traditional logic gates like OR, AND, NOT, and JK Flip-Flop.

12. Future work

As a future work, increasing the bit accuracy from 4 bits to 6 or 7 bits, with high clock frequency and lowering the power consumption by for example using a capacitor array in the DAC circuit are taking into account.

13. REFERENCES

1. C. Cho. "A POWER OPTIMIZED PIPELINED ANALOG-TO-DIGITAL CONVERTER DESIGN IN DEEP SUB-MICRON CMOS TECHNOLOGY". PhD Thesis. Georgia Institute of Technology.2005.
2. K. Kval. "Design of an Analog to Digital Converter with Superior Accuracy/Bandwidth vs. Power Ratio". MSc Thesis. Norwegian University of Science and Technology Department of Electronics and Telecommunications. 2011.
3. T. Sundström. "Design of High-Speed Analog-to-Digital Converters using Low-Accuracy Components". Linköping University Department of Electrical Engineering. Linköping Studies in Science and Technology Dissertations. No. 1367. 2011.
4. B. Ginsburg. "Energy-Efficient Analog-to-Digital Conversion for Ultra-Wideband Radio". PhD Thesis. Massachusetts Institute of Technology Department of Electrical Engineering. 2007.
5. T. Kuphaldt. "Lessons In Electric Circuits". Volume IV-Digital, Fourth Edition. copy right Tony R. Kuphaldt. 2002.
6. J. Digel. and other. "A 6 bit and a 7 bit 80MS/s SAR ADC for an IR-UWB Receiver". Institute of Electrical and Optical Communications Engineering. University of Stuttgart. 2IHP. Frankfurt (Oder). Germany. 2011.
7. N. Barot. "Successive Approximation Analog to Digital Converter". Industrial Sponsor Srenik Mehta. Atheros Comm. 2010.
8. K. Breeding. Digital Design Fundamentals. Prentice Hall, Second Edition, 1992, pp 158-160.
9. E. Hwang. Digital Logic and Microprocessor Design With VHDL. La Sierra University. Books/cole, 2005, pp 202-214.
10. T. Floyd. DIGITAL FUNDAMENTALS. Prentice Hall Ninth Edition Pearson Education Interactional, 2006, pp 447-458.
11. S. Yan Ng. and other "A low-voltage CMOS 5-bit 600MHz 30mW SAR ADC for UWB wireless Receivers", Department of Electrical and Computer Engineering, The Ohio State University Columbus, 2005 IEEE, 187-190.

Automated Education Propositional Logic Tool (AEPLT): Used For Computation in Discrete Mathematics

J. Mbale *mbalej@yahoo.com*
Centre of Excellence in Telecommunications (CoE)
Department of Computer Science
Faculty of Science
University of Namibia
Windhoek, 340, Namibia

Abstract

The Automated Education Propositional Logic Tool (AEPLT) is envisaged. The AEPLT is an automated tool that simplifies and aids in the calculation of the propositional logics of compound propositions of conjuction, disjunction, conditional, and bi-conditional. The AEPLT has an architecture where the user simply enters the propositional variables and the system maps them with the right connectives to form compound proposition or formulas that are calculated to give the desired solutions. The automation of the system gives a guarantee of coming up with correct solutions rather than the human mind going through all the possible theorems, axioms and statements, and due to fatigue one would bound to miss some steps. In addition the AEPL Tool has a user friendly interface that guides the user in executing operations of deriving solutions.

Keywords: Compound proposition, propositional variables, propositional logic, truth table, connective and SEMINT specific parser.

1. INTRODUCTION

This work envisages a solution of automating the calculation of the propositional logic which is user friendly. This paper introduces the Automated Education Propositional Logic Tool (AEPLT) designed for the stated task. The AEPLT automatically calculates the propositional logics of compound propositions of conjuction, disjunction, conditional, and bi-conditional. The AEPLT architecture is composed of the following components: Proposition Process (PP), Operator, SEMINT Specific Parser, Assumption Statements, T/F and Truth Table. Once the user activates the system, the SEMINT Specific Parser automatically extracts the propositional variables from the PP and connectives from Operator. The SEMINT Specific Parser has the ability of forwarding the compound propositions or formulas to the right Assumption Statements. In these right Assumption Statements, the formulas are examined against various statements. After this examination, the system is ready to give results whether Truth False value, which is then recorded in the truth table. Once the results are recorded, the user can access the results from the truth table for the intended application. The work also demonstrates an algorithm that clearly illustrates the stages of calculating and implementing the tool. The system's application is comprehensively demonstrated by its interface, which guides the use of the system and makes this tool user friendly.

1.1 Statement of Problem

During the Discrete Mathematics classes, students struggle to calculate the propositional logics of compound proposition. Yet in this cutting edge era, the Information Communication Technology (ICT) tools have been employed and applied to automate all challenging mathematical, physics, engineering and any other scientific problems. Considering all the theorems, axioms and statements the student has to undergo in order to derive the logic that would help him come up with results, the process tends to be cumbersome. It is in view of this that the AEPLT was introduced to automate the calculation of the propositional logic of the conjuction, disjunction, conditional and bi-conditional.

1.2 Literature Review

The propositional logic is one of the topics under Discrete Mathematics course or discipline. Before tackling propositional logic, it is inevitable to first look at the Discrete Mathematics which is the overall course. In fact, the significance of Discrete Mathematics as the basis for formal approaches to software development has been noted by many scholars such as Dijkstra, Gries, and Schneider [1,2,3]. This position continues to be espoused by the ITiCSE working group [4] among others [5,6,7,8]. The idea that Discrete Mathematics can be viewed as software engineering mathematics has been popularized by the Woodcock and Loomes textbook [9]. In [1] it was emphasized that the application of Discrete mathematics to the software development problem has been the subject of extensive research. He added that much of the initial effort was directed to formal verification, the process of showing the equivalence of two software system presentations. He also indicated that Discrete Mathematics pedagogy has a rich background. Whereas in [10,11,12,13] they pointed out that Discrete Mathematics literature dates back even to the first proposed computing curricula. In [14, 15] they described Discrete Mathematics as the study of mathematical structures and objects that are fundamentally discrete rather than continuous. They gave some examples of objects with discrete values as integers, graphs, or statement in logic. From their description they also pointed out that concepts from Discrete Mathematics were useful for describing objects and problems in computer algorithms and programming languages. They further emphasized that these had applications in cryptography, automated theorem proving and software development. This description is also supported in [16] where they discussed Discrete Mathematics as the study of mathematical structures that do not support or require the notion of continuity. They outlined the topics of Discrete Mathematics to include: logic, sets, numbers theory, combinatorics, graphs, algorithms, probability, information, complexity, computability, etc. Also in [1] it was pointed out Discrete Mathematics underlying modern software engineering theory that included: propositional and first-order predicate logic, reasoning, proof techniques, induction, finite set theory, relations and graphs. the Discrete Mathematics classes, students struggle to calculate the propositional logics of compound proposition.

The research on the propositional logic was envisaged as far back as 1854 by a Mathematician George Boole [17, 18] who established the rules of symbolic logic in his book titled, The Laws of Thought. Since then, a lot of scholars had been researching on the significance of propositions towards building the reasoning capacity of the learners, engineers and other scientists. Many scholars have come up with various definitions to the concepts of propositional logic. Others have separated definitions of propositional and logic. [19] defined logic as the science of reasoning correctly. He further emphasised that this subject has a long history, and narrates that the person generally agreed to have founded formal logic was Aristotle who is method of formal reasoning was called the syllogism. He also pointed out that in nineteen century philosophers and mathematicians like Boole, De Morgan, Frege and others, became interested in modeling the laws of thought. In [14] he classified logic as the branch of philosophy concerned with analyzing the patterns of reasoning by which a conclusion is drawn from a set of premises, without reference to meaning or context. They further emphasized its importance as a formalization of reasoning, a formal language for deducing knowledge from a small number of explicit stated premises, hypotheses, axioms and facts. They also pointed out that logic was a formal framework for representing knowledge. They also defined proposition as the underlying meaning of a simple declarative sentence, which is either true or false. In this definition, the truth or falsehood of a proposition was indicated by assigning it one of the truth values T, for true and F for false. They also cited some examples of propositions as: mammals are warm blooded, the sun orbits the earth, four is a prime number, Joan is taller than John, etc. A practical example was given from a statement: “In 1938 Hitler seized Austria, *(and)* in 1939 he seized former Czechoslovakia *and* in 1941 he attacked the former USSR *while* still having a non-aggression pact with it.” This statement was expressed in atomic propositions as: p = in 1939 Hitler seized Austria; q = 1939 he seized former Czechoslovakia; r = 1941 he attacked the former USSR, and s = in 1949 Hitler had a non-aggression pact with the USSR. This was formalized in propositional logic as: $s \wedge q \wedge r \wedge s$.

[20] pointed out that many mathematical statements were constructed by combining one or more propositions. He further stated that new prepositions called compound propositions are formed from the existing propositions using logical connectives or operators. Whereas [19] defined compound statements as the combination of primitive statements by means of logic connectives. [19, 20] stated that letters were used to denote propositional variables or statement variables. They gave the conventional letters used for propositional variables as: p, q, r, s, etc. The authors also defined the truth table which displays the relationships between truth values that are true (denoted T) if it is true proposition and false (denoted F) otherwise. They classified these propositions as: (i) negation of p and denoted $\neg p$. They defined as the truth value of the negation of p was the opposite of the truth value of p, read as "not p"; (ii) the compound proposition conjunction denoted $p \wedge q$, is true when both p and q are true and false otherwise; (iii) the compound proposition disjunction denoted $p \vee q$ is false when both p and q are false and true otherwise; the compound proposition conditional denoted $p \rightarrow q$, is false when p is true and q is false, and is true otherwise; (iv) the compound proposition exclusive or denoted $p \oplus q$, is true when exactly one of p and q is true and is false otherwise; (v) the compound proposition bi-conditional denoted by $p \leftrightarrow q$, is true when p and q have the same truth value, and is false otherwise.

2. THE AELPT SYSTEM ARCHITECTURE

The Propositional Computation architecture given in Figure 1 is an automated model which is used to calculate the values of compound propositions: conjunction, disjunction, conditional and bi-conditional. The architecture is composed of the following components: Proposition Process (PP), Operator, SEMINT Specific Parser, Assumption Statements, T/F and Truth Table.

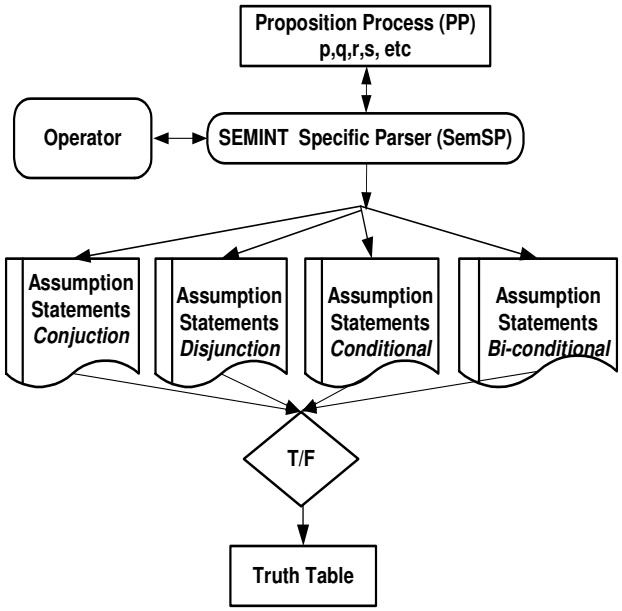


FIGURE 1: Preposition computation architecture.

The user activates the Propositional Process (PP) which holds the propositional variables such as the p, q, r, s, etc. Once the PP is activated, the SEMINT Specific Parser (SemSP) automatically extracts the propositional variables and the logical connectives from the Operator, forming a compound statement or formula such as $p \wedge q$, $p \vee q$, $p \rightarrow q$, $p \oplus q$ and $p \leftrightarrow q$. The SemSP is intelligently designed to automatically pass the compound statement to the right Assumption Statements component. The Assumption Statement has pre-programmed statements that determine the final computed propositional statement whether true or false. Then either the true (T) or false (F) is selected from the T/F decision box and this result is recorded in underlying Truth

Table. This Truth Table then holds the results of the computed compound proposition such as conjuction, disjunction, conditional and bi-conditional. Then the user can pick the results his/her application purposes.

3. IMPLEMENTATION OF AUTOMATED EDUCATION PROPOSITIONAL LOGIC TOOL

The implementation of Automated Education Propositional Logic Tool is done by the algorithm illustrated in Figure 2.

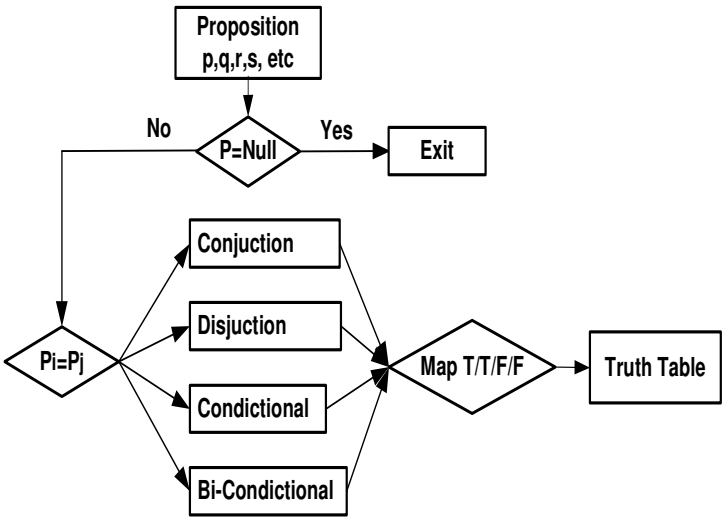


FIGURE 2: Preposition implementation algorithm.

From Figure 2, the propositional variables are activated into the decision box. In the decision box, if the propositional variables are null, then the process is exit. This would imply that the variables are empty and the process can not be continued. At the same instance, if the propositional variables are not null, the pair or set of such statements are passed on to the adjacent decision box. Let P_i be the first propositional variable and P_j be the second one. The pair or the set of propositional variables are mapped to a connective forming a compound statement or formula. Then the mapped propositional variables are forwarded to the right compound proposition where the formula would be computed through a series of statements. These statements make a complete algorithm that determines true and false solution. Below, the statements are discussed:

A. Conjuction ($p \wedge q$):
   - True ***and*** True is True, because both sides of the conjuction are True, then the proposition holds True
   - True ***and*** False is False, because a proposition cannot be both True and False at the same time, hence False
   - False ***and*** True is False, because a proposition cannot be both True and False at the same time, therefore False
   - False ***and*** False is False, because a proposition holds to be False on both sides of conjuction, hence False.

B. Disjunction ($p \vee q$):
   - True ***or*** True is True, because both sides of the disjunction are True, then the proposition holds True
   - True ***or*** False is True, because at least one side of the disjunction is True, therefore, the proposition is True
   - False ***or*** True is True, because at least one side of the disjunction is True, hence, the proposition is True

- False ***or*** False is False, because all the sides of disjunction hold False, then the proposition is False.

C. Conditional ($p \rightarrow q$):
- True ***implies*** True is True, hence the proposition holds True
- True ***implies*** False is False, this result takes precedence to make the proposition False
- False ***implies*** True is True, this result takes precedence to make the proposition True
- False ***implies*** False is False, which is True from the statement, hence the proposition is True.

D. Bi-Conditional ($p \leftrightarrow q$):
- True ***implies*** True and is ***implied*** by true, it gives True, hence the proposition holds True
- True ***implies*** False and False ***implies*** True, therefore the proposition is False because what is False is never True and vice versa
- False ***implies*** True and True ***implies*** False, hence the proposition is False because it is not True that what is False is True and vice versa
- False ***implies*** False and False ***implies*** False, hence the proposition is True, because False is False.

After the formulas examine the four compound statements, then from the decision box the results are produced and recorded in the Truth Table. Therefore, the user can pick the results for the intended application.

E. The Application Interface: Propositional Tool:
The AEPLT has a user friendly interface. It has a pull down menu, where the user can select what he/she wants to calculate such as conjuction, disjunction, conditional, and bi-conditional as illustrated in Figure 3.

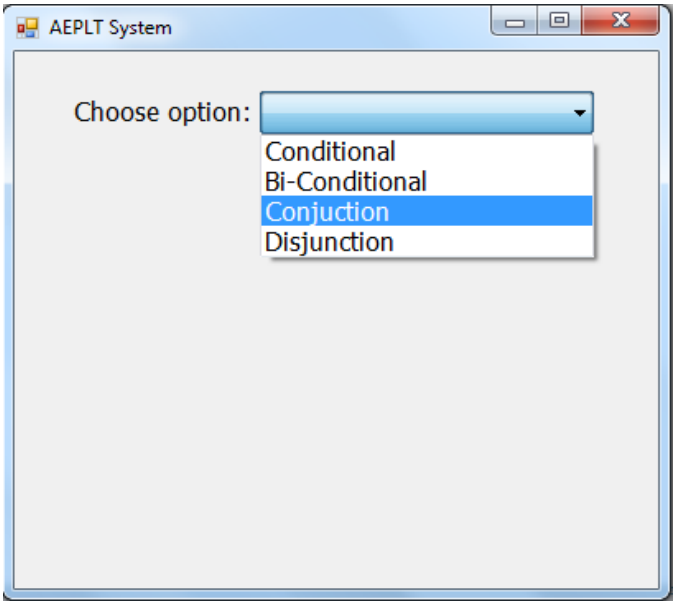


FIGURE 3: Compound propositional input window.

From Figure 3, if you select conjuction from a pull down menu, then the Figure 4 appears. This figure has entry or input spaces for entering the variables of propositions p and q that are True (T) and False (F). Every after entering values True (T) and False (F), one can still view the individual results without checking on the Truth Table by pressing on the button “View Result”. When you press on “View Button”, the system will display Truth Table conjuction results. Similar calculations can be done on others such as the disjunction, conditional and bi-conditional.

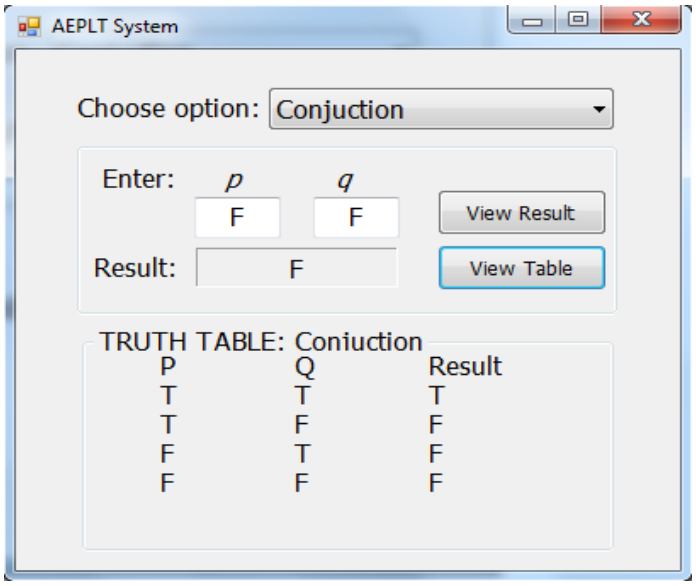


FIGURE 4: The Conjuction Window for Input.

4. CONCLUSION

Development in all sectors of work, require correct planning in order to provide tools that will yield the intended results. As in [19], logic was defined as the science of reasoning correctly. Once the implementers reason correctly in strategizing and planning in executing their tasks, positive results would be achieved. Hence, in this work, the AEPLT was envisaged to come up with automated systems which will always give a precise propositional logic results. The system has an architecture where the user simply enters the propositional variables and whole calculation is done giving accurate results.

The envisaging of this model, Automated Education Propositional Logic Tool (AEPLT) has scored a number of achievements. First it has allowed the users, who are in this case the students to concretely use this automated model, rather than calculating the propositional logic of compound propositions of conjuction, disjunction, conditional and bi-conditional manually. Secondly, the automated model has a user friendly interface where the student enters the propositional variables and then the system automatically maps them with the right connectives to form compound proposition or formula that are calculated to yield the intended results. Thirdly, during the execution, this automated system gives a guarantee of producing correct results rather than when it is done manually whereby due to fatigue or exhaustion, the user may bound to key-in incorrect input and thereafter result into wrong output.

5. REFERENCES

[1] J. P. Cohoon and J. C. Knight. “Connecting Discrete Mathematics and Software Engineering,” 36th ASEE/IEEE Frontiers in Education Conference, San Diego, CA, October 28 – 31, 2006.

[2] E. W. Dijkstra. “On the cruelty of really teaching computing science,” Communications of the ACM, December 1989, pp. 1398-1404.

[3] D. Gries, and F. B. Schneider. “A logical approach to discrete math,” Springer-Verlag, New York, 1993.

[4] V. L. Almstrum, C. N. Dean, D. Goelman, T. B. Hilburn, and J. Smith. “ITiCSE 2000 working group report: support for teaching formal methods,” SIGCSE Bulletin, June 2001.

[5] A. E. Fleury. “Evaluating discrete mathematics exercises”, SIGCSETechnical Symposium on Computer Science Education, 1993, pp. 73-77.

[6] J. W. McGuffee. “The discrete mathematics enhancement project”, Journal of Computing in Small Colleges, 2002, pp. 162-166.

[7] H. Saiedian. “Towards more formalism in software engineering education”, SIGCSE Technical Symposium on Computer Science Education, 1993, pp. 193-197.

[8] K. Heninger. “Specifying Software Requirements Complex Systems: New Techniques and Their Application”, *IEEE* Transactions on Software Engineering, Vol. SE-6, No. 1, January 1980.

[9] J. Woodcock, and M. Loomes. “Software Engineering Mathematics” Software Engineering Institute, Series in Software Engineering, 1988.

[10] A. T. Berztiss. “The why and how of discrete structures”, SIGCSE Technical Symposium on Computer Science Education, 1976, pp. 22-25.

[11] R. E. Prather. “Another look at the discrete structures course”, SIGCSE Technical Symposium on Computer Science Education, 1976, pp. 247-252.

[12] J. P. Tremblay, and R. Manohar. “A first course in discrete structures with applications to computer science,” SIGCSE Technical Symposium on Computer Science Education, 1974, pp. 155-160.

[13] A. Tucker, (editor). “Computing curricula 1991: report of the ACM/IEEE-CS Joint curriculum task force”, *ACM Press*, 1991.

[14] http://www.cs.pitt.edu/

[15] http://gear.kku.ac.pitt.edu/

[16] mason.gmu.edu/~asamsono/teaching/math125/Lecture1.pdf · PDF file

[17] http://docs.google.com/

[18] www-groups.dcs.st-and.ac.uk/history/Mathematicians/Boole.html

[19] Robin Hirsch. www.cs.ecl.ac.uk/staff/r.hirsch//teaching/1b12/

[20] www.coursehero.com/file/2552944/s11propositionallogicBW

Principal Type Scheme for Session Types

Álvaro Tasistro *tasistro@ort.edu.uy*
Universidad ORT Uruguay
11100, Montevideo, Uruguay

Ernesto Copello *copello@ort.edu.uy*
Universidad ORT Uruguay
11100, Montevideo, Uruguay

Nora Szasz *szasz@ort.edu.uy*
Universidad ORT Uruguay
11100, Montevideo, Uruguay

Abstract

Session types as presented in [1] model communication between processes as a structure of dialogues. The dialogues are specified by sequences of types of messages, where each type describes the format and direction of the message. The resulting system imposes a type discipline that guarantees compatibility of interaction patterns between processes of a well-typed program. The system is polymorphic in Curry's style, but no formal treatment of this aspect has been provided yet. In this paper we present a system assigning type schemes to programs and an algorithm of inference of the principal type scheme of any typable program for a significant fragment of the calculus which allows delegation of communication, i.e. transmission of channels. We use classical syntax for variables and channels, i.e. just one sort of names in each case for either bound of free occurrences. We prove soundness and completeness of the algorithm, working on individual terms rather than on α-equivalence classes. The algorithm has been implemented in Haskell and partially checked in the proof assistant Agda.

Keywords: types, principal type scheme, type inference algorithm.

1. INTRODUCTION

Systems of (dyadic) session types allow to structure programs which consist of communicating processes as networks of *dialogues*. Each such dialogue is called a *session* and is carried out through a specific sort of communication entity called a *channel*. Channels are created by a special kind of interaction occurring at ordinary ports, which we explain at once: using syntax close to that in the original presentation of session types [2], we write ***acc*** *a(k).P* to represent a process that is willing to *accept* a session at port *a*. This can interact with concurrent ***req*** *a(k').Q* which can be thought of *requesting* such session. As a consequence of the interaction, a new channel is created that will communicate the continuation processes *P* and *Q*. In these processes, the names *k* and *k'* will (respectively) represent the two *ends* of the newly created channel. Thus, and as a consequence of the dialogue restriction, each channel end in the system belongs to one and only one process.
Once the channel is created, the session takes place, i.e. a sequence of messages is interchanged. The system of types allows characterizing each session as a sequence of *message formats*, where each format specifies the direction and type of contents of the message. Such characterization is a *session type*. A process like *P* or *Q* above can in turn be characterized by the (session) types of its (free) channels, which are determined by the actions performed by the process at each of its channel ends. Let us call the set of channel types of a process its *typing*. Now, in ***acc*** *a(k).P* and ***req*** *a(k).P* the name *k* becomes bound and the process ceases to depend on it; that is to say, the typing of **acc** *a(k).P* shall not mention *k* anymore. The port *a* is, however, assigned the type of *k*. And thus it becomes in principle possible to check whether two processes ***acc*** *a(k).P* and ***req*** *a(k').Q* that expect to establish a session through *a* do indeed hold compatible

interactions. Compatible means actually *dual*, i.e. an output in one process must be mirrored by an input in the other, and with contents of the same type. Thereby, type correctness ensures absence of compatibility errors in communication and freedom from interference of third parties in the dialogues. The safety property can be characterized as freedom from dead-lock in the case that sessions do not overlap.

As already said, session types appeared in [3,2]. Next to that, [4] allowed the transmission of channels in sessions, i.e. the possibility of implementing *delegation*: a process can then delegate a session to another process that takes over the dialogue. It does so by sending the second process the corresponding channel end (which is then definitely lost by the original process.)

The system of types was later refined in [1], following ideas of [5]. That is the system that we shall consider in this paper. The problem to study is that of *type inference*, i.e. of performing type checking even when (some) type declarations are omitted. Actually the system in [1] is polymorphic in Curry's style and admits a definition of *principal type scheme*. The principal type scheme can be computed for each well-formed process, even without any type declaration of variables, channels or ports. Such is the contribution of this paper: a formal treatment of session type polymorphism, in which we give an inference algorithm and prove its soundness and completeness. We here carry out the work for a fragment of the original type system including channel delegation but not choice or recursion. These restrictions are not essential, as we shall indicate. We have implemented our algorithm in Haskell and in the proof assistant Agda [6], in which the proof of soundness has been completely fomalized.

This kind of work has not been done elsewhere, as far as we know. In [5] a type checking (not inference) algorithm is given and its soundness proven, for a system more expressive than the one considered here, since it allows for subtyping in the session types. In [7] a simple version of session types is studied. in which only two implicit channels can be simultaneously used. As a consequence of this, delegation of channels is not possible. The type system is also simplified not allowing recursive types. In this work type safety is proven and an OCaml implementation of an inference algorithm is presented, for which proofs of some basic results are given.

There are some other related works that embed session types in other programming languages. In [8] and [9] session types are implemented in Haskell, making use of its powerful polymorphic type system and type classes with functional dependencies [10]. In the first work only one channel is implemented and soundness of the embedded system is proven. In the second one multiple channels are allowed but no soundness property is given. In [11] a more general technique is given to embed session types with multiple channels, thus earning more portability in the host language. In particular, any polymorphic language can be used as host. Soundness is proven but only for one channel.

The rest of the paper is organized as follows: in the next section we introduce the process language and the type system, adapted from the one in [1]. We notice it is polymorphic and then introduce the notion of type scheme, employing type variables. In section 3 we formulate our algorithm of type inference which computes, for every typable program, its principal type scheme, i.e. a type scheme assignable to the program and from which every other typing of the program can be obtained as a type substitution instance. We give detailed proofs of soundness and completeness of the algorithm. We expose conclusions and remaining work in section 4.

2. SESSION TYPES

Syntax of processes is as follows:

P : − 0 | k!e ; P | k?x. P | k!!k' ; P | k??k'. P | acc a(k). P | req a(k). P | P |Q

We now informally explain their meaning. In what follows *channel* and *channel end* are used interchangeably:
- *0* is the inactive process.
- In the term *k!e ; P*, *k* is a channel end and *e* an expression whose value is data to be sent along *k*. Then the process continues behaving as *P*.
- In *k?x. P*, data is received in the channel end *k*. The variable *x* is bound in the term.
- The term *k!!k' ; P* sends the channel end *k'* along the channel *k* and then becomes *P*.
- Dually, *k??k'. P* receives a channel in the channel end *k*. The name *k'* becomes bound in the term.
- The meaning of *acc a(k). P* and *req a(k). P* is related to session initiation and has already been explained. The name *k* is bound in both terms.
- Finally, *P |Q,* is the parallel composition of processes *P* and *Q*.

As usual, we assume denumerable sets of channel names and of variables. Also, as is evident from the syntax above, we assume a class of data expressions to be specified separately.
The syntax has been chosen so as to include those cases that are essential for the study of compatibility of interaction. In this regard, the only constructs that could be said missing are the choice operators. But consideration of these adds only technical difficulties that lie somehow beside the problem we are interested in. Also for completing a sufficiently expressive language we should include recursion or replication. We shall comment on this later.

We now turn to the consideration of types. We shall assume that an appropriate type system exists for the data expressions, whose properties are to be stated when necessary. Let for the moment δ stand for data types. Then session types are as follows:

$$\alpha,\beta : - 1 \mid \uparrow\delta ; \alpha \mid \downarrow\delta ; \alpha \mid \uparrow\alpha ; \beta \mid \downarrow\alpha ; \beta$$

i.e. they are finite sequences of message formats, each of which specifies the direction (↑ = out, ↓ = in) and type of the contents of the message (type of data or type of a channel being sent or received in delegation). *1* stands for impossibility of communication. Should we consider recursion, we would have to allow for (finite descriptions of) infinite sequences. Also if we considered choice there would have to be a branching construct.

The *dual* $\underline{\alpha}$ of a type α is defined as follows:

$$\underline{1} = 1$$
$$\underline{\uparrow\delta; \alpha} = \downarrow\delta; \underline{\alpha}$$
$$\underline{\downarrow\delta; \alpha} = \uparrow\delta; \underline{\alpha}$$
$$\underline{\uparrow\alpha; \beta} = \downarrow\alpha; \underline{\beta}$$
$$\underline{\downarrow\alpha; \beta} = \uparrow\alpha; \underline{\beta}$$

A typing judgement is of the form $\Gamma; \Pi \vdash P \triangleright \Delta$, where:

- *P* is the program being typed.
- Δ is the channel context, recording the types of the free channels of the program *P*.
- Γ is the data context, containing the declarations of the free (data) variables of *P*.
- Π is the port context, with the declarations of the ordinary ports of P.

Data contexts Γ are finite partial functions from data variables to data types. The application of function *F* to argument *a* will be written *F a*. The *union* of two data contexts Γ, Γ' is still a valid data context when $\Gamma\ x = \Gamma'\ x$ for every variable *x* which is defined (declared) in both Γ and Γ'. Port contexts Π are, similarly, finite partial functions from sort names to session types.
Channel contexts Δ are instead total functions from channel names to channel types, *1* almost everywhere. This choice proves to be convenient and reflects the fact that unused and unusable (inactive) channels are indistinguishable. In particular, ***1*** is the constant function everywhere equal to *1*. Two channel typings Δ and Δ' are *disjoint*, to be written $\Delta \ / \ \Delta'$ iff for every channel *k*, at

least one of Δk and $\Delta' k$ is *1*. The *union* of two disjoint channel typings, to be written $\Delta.\Delta'$, is such that for every channel *k*, $(\Delta.\Delta')k$ is the sum of Δk and $\Delta' k$, where sum has *1* as (left and right) identity element. *Overriding* a function *F* with a pair *(a, b)* is written $F \prec\!+ a \to b$ and gives value *F x* for every $x \neq a$, whereas it gives value *b* for argument *a*. In the case of channel and data contexts, we will write : in overridings instead of the symbol →. When treating channel contexts it will prove sometimes convenient to use a notation for a strong form of overriding to be written · and that can be called *extension*. Specifically, $\Delta \cdot k : \alpha$ means the overriding of Δ with the pair *(k,* α*)* but requiring further $\Delta k = 1$.

The type system is exposed in Figure 1.

$$\textit{inact:}\ \frac{}{\Gamma;\ \Pi\ \vdash\ 0 \rhd \mathbf{1}}$$

$$\textit{snd:}\ \frac{\Gamma \vdash e :\ \delta \qquad \Gamma;\ \Pi \vdash P \rhd \Delta}{\Gamma\ ;\ \Pi \vdash k!e\ ;\ P \rhd \Delta \prec\!+ k{:}\uparrow\delta\ ;\ \Delta k}$$

$$\textit{rcv:}\ \frac{\Gamma \prec\!+ x{:}\ \delta\ ;\ \Pi \vdash P \rhd \Delta}{\Gamma\ ;\ \Pi \vdash k?x.\ P \rhd \Delta \prec\!+ k{:}\downarrow\delta\ ;\ \Delta k}$$

$$\textit{thrw:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta}{\Gamma\ ;\ \Pi \vdash k!!k'\ ;\ P \rhd \Delta \prec\!+ k{:}\uparrow\alpha\ ;\ \Delta k \cdot k'{:}\alpha}$$

$$\textit{ctch:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \cdot k'{:}\alpha}{\Gamma\ ;\ \Pi \vdash k??k'.\ P \rhd \Delta \prec\!+ k{:}\downarrow\alpha\ ;\ \Delta k}$$

$$\textit{acc:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \cdot k{:}\ \Pi a}{\Gamma;\ \Pi \vdash acc\ a(k).\ P \rhd \Delta}$$

$$\textit{req:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \cdot k{:}\ \underline{\Pi a}}{\Gamma;\ \Pi \vdash req\ a(k).\ P \rhd \Delta}$$

$$\textit{conc:}\ \frac{\Gamma\ ;\ \Pi \vdash P \rhd \Delta \qquad \Gamma\ ;\ \Pi \vdash Q \rhd \Delta'}{\Gamma;\ \Pi \vdash P \mid Q \rhd \Delta.\Delta'}\ \Delta/\Delta'$$

FIGURE 1: The Type System

We proceed to explain the rules:

- First, the rule *inact* establishes that any channel is *completed* (no longer usable) in process *0.*
- The next *snd* rule corresponds to the event of *sending* data through the channel *k*. We assume the existence of a type system for data expressions in which it is possible to type these under declarations of its variables, which are of course the variables of our programs. That explains the first premise of the rule. The second premise types the continuation process *P*, and then the conclusion updates the typing of *P* with the new type of *k*, obtained by prefixing $\uparrow\delta$ to the type sequence characterizing *k* in the continuation process.
- The third rule *rcv* corresponds to *receiving* data through a channel. A variable *x* is used and its declaration updates the data context in the typing of the continuation process *P*. The type declared to *x* is of course the type of the data received in the resulting typing in the conclusion of the rule.
- Next comes the rule *thrw* corresponding to sending (*throwing*) a channel end through a channel. The thrown channel end must be named with a fresh identifier, i.e. one not occurring in the continuation process P. This reflects the fact that the channel end will no longer belong to the process that just threw it over. In the rule the condition is imposed by the use of the extension operator · in the conclusion. Notice that the rule can be applied for whatever type is associated to the thrown channel.

- The next rule *ctch* is for receiving (*catching*) a channel sent over by a communicating process. The name *k'* used to represent the received channel end becomes bound, which is reflected in the fact that it cannot appear in the typing Δ. Syntactically, this is enforced by use of the extension operator in the premise. In fact, the names *k* and *k'* could coincide (since after all no restriction should be placed in the choice of the name *k'*). In this case the resulting typing of the process at hand depends on *k* and reflects that it becomes unusable after the catching.
- The rule *acc* is for *accepting* a session. The channel end *k* becomes bound and similar considerations as above apply. But there is a detail to comment, which concerns the type assigned to *k* in order to type the continuation process. This type is the same that the port *a* has in the port context Π. This means two things: firstly, *a* has to be declared in Π, and this ought to be made explicit as a side condition to the rule. The reason why we have omitted this has to do only with brevity of the presentation and will become clearer below. Secondly, the rule reflects that the typing of the ordinary ports is the type of the channel end created by interaction at that port.
- This is to be linked to the next, dual, rule *req* for typing a *request* of session. What we require in this case is that the channel behaves in a manner dual to the type of the port, and that will make it dual to the type of the opposite end of the channel created at the interaction of acceptance and request. That is to say, in a parallel composition of an acceptance and a request of a session the port is typed uniformly in both cases, and it is the channel ends which have to receive dual types.
- Finally, the rule *conc* of *concurrent composition* of processes requires that no channel end belongs to more than one process (disjointness of the channel typings) and that all variables and ports are uniformly typed in both processes.

Notice that no type declaration is required in the syntax of terms for any of the variables, ports or channels. This is coincident with the formulation in [1], of which the system presented here is a slight variant. The system is thus polymorphic à la Curry. Examples of polymorphic terms are: *acc a(k).k?x.k!x; 0* and *acc a(k).acc b(k'). . . . k!!k'; P* .
This motivates the investigation of type schemes. We therefore consider a denumerable set of *type variables* a^t and define the (session) *type schemes* as follows:

$$\alpha : - a^t \mid \underline{a}^t \mid 1 \mid \uparrow\delta\ ;\ \alpha \mid \downarrow\delta;\ \alpha \mid \uparrow\alpha\ ;\ \alpha \mid \downarrow\alpha\ ;\ \alpha$$

The scheme $\underline{a}^t$ is there just to stand as the dual of type variable a^t (and its dual is of course a^t).

We then consider the type system given before with two modifications: first, we assign type schemes in place of types; and, secondly, only for the sake of simplicity of the treatment, we shall consider port contexts as total functions from port names to type schemes. The (new) port contexts shall be built by successive instantiations from an original context in which every port name has associated a different type variable. We call this the *void* or *purely generic* port context and write it Ω. It can be implemented by assuming that each port name *a* can be encoded uniquely as a type variable a^t. We define $dom\ \Pi = \{a \mid \Pi a \neq a^t\}$ for port context Π. This set will always be finite. We further define two port contexts Π and Π' to be *compatible* when for every port *a*, either $\Pi a = \Pi' a$ or one of them is a type variable. For compatible port contexts Π and Π', define the *union* $\Pi.\Pi'$ to take, for each port *a*, the common value at *a* of Π and Π' if such is the case, or the more instantiated one otherwise. We insist in that these considerations are only for simplicity in the treatment to be presented below and are not essential to it. Otherwise, the system with type schemes is exactly like the one above. In particular, the rules are the same as displayed in Figure 1.

We have also to consider *type substitutions*. These are finite partial functions from type variables to type schemes and their action on type schemes is defined in the obvious way. We only have to remark that substitution of type scheme α for a^t in $\underline{a}^t$ yields $\underline{\alpha}$.

Finally, we assume that a similar extension to type schemes can be applied to the system of typing of data expressions. Then the following two basic results are obtained, provided they hold too for the system of data expressions:

Lemma 1 (Weakening). If $\Gamma; \Pi \vdash P \rhd \Delta$ and $\Gamma \subseteq \Gamma'$ then $\Gamma'; \Pi \vdash P \rhd \Delta$.

Proof. Immediate induction on the type system. Use that $\Gamma \prec\!+ x: \delta \subseteq \Gamma' \prec\!+ x: \delta$ if $\Gamma \subseteq \Gamma'$.

Lemma 2 (Closure under type substitution). If $\Gamma; \Pi \vdash P \rhd \Delta$ then for any type substitution θ, $\Gamma\theta; \Pi\theta \vdash P \rhd \Delta\theta$.

Proof. Induction on the type system. Use that for any type substitution σ and type scheme α, $\underline{\alpha}\sigma = \underline{\alpha\sigma}$.

3. TYPE INFERENCE

An inference algorithm for the given type system is displayed in Figure 2.
We make use of the form of judgement $\Gamma; \Pi \leftarrow P \rightarrow \Delta$ with the obvious meaning, i.e. given program P the algorithm infers (if possible) the contexts Γ, Π and Δ. Further, as shall be proven presently, the typing inferred in case of success is the most general that can be assigned to P in the type system, i.e. it is the principal type scheme of P. This means that every other typing of P can be obtained from the one inferred by applying to this a suitable type substitution.
A simple inspection reveals that for each program P the inferred typing is unique up to the choice of the type variables used to construct it. The type variables are introduced in the (conclusions of the) rules rcv_2 and *thrw,* and as will be shown, the choice of particular names is immaterial once certain basic conditions of freshness are ensured, namely that the names are fresh w.r.t. the set of type variables used in each rule's premise. This allows us to make the following convention in order to simplify the presentation: in rules with two premises, no type variable is used in both premises. And in rules in which we introduce type variables, these are fresh w.r.t. the set of type variables used in the premises.

$$inact: \ \frac{}{\varnothing; \Omega \leftarrow 0 \rightarrow \mathbf{1}}$$

$$snd: \ \frac{\Gamma \leftarrow e \rightarrow \delta \qquad \Gamma'; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\theta \cup \Gamma'\theta; \Pi\theta \leftarrow k!e\ ; P \rightarrow \Delta\theta \prec\!+ k: \hat{\ }\delta\theta\ ; (\Delta\theta)k} \ \Gamma \cup^{\theta} \Gamma'$$

$$rcv_1: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma \setminus x\ ; \Pi \leftarrow k?x.\ P \rightarrow \Delta \prec\!+ k:\downarrow\Gamma x\ ; \Delta k} \ x \in dom\ \Gamma$$

$$rcv_2: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\ ; \Pi \leftarrow k?x.\ P \rightarrow \Delta \prec\!+ k:\downarrow \mathsf{a}^{\mathsf{t}}\ ; \Delta k} \ x \notin dom\ \Gamma$$

$$thrw: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\ ; \Pi \leftarrow k!!k'\ ; P \rightarrow \Delta \prec\!+ k:\uparrow \mathsf{a}^{\mathsf{t}}\ ; \Delta k \cdot \mathsf{k}':\mathsf{a}^{\mathsf{t}}}$$

$$ctch: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\ ; \Pi \leftarrow k??k'\ ; P \rightarrow \Delta \setminus k' \prec\!+ k:\downarrow\Delta k'\ ; (\Delta \setminus k')k}$$

$$acc: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\theta; \Pi\theta \leftarrow acc\ a(k).\ P \rightarrow \Delta\theta \setminus k} \ \Pi a \cup^{\theta} \Delta k$$

$$req: \ \frac{\Gamma\ ; \Pi \leftarrow P \rightarrow \Delta}{\Gamma\theta; \Pi\theta \leftarrow req\ a(k).\ P \rightarrow \Delta\theta \setminus k} \ \Pi a \cup^{\theta} \underline{\Delta k}$$

$$conc:\ \frac{\Gamma\ ;\ \Pi \leftarrow P \rightarrow \Delta \qquad \Gamma'\ ;\ \Pi' \leftarrow Q \rightarrow \Delta'}{\Gamma\theta \cup \Gamma'\theta;\ \Pi\theta.\Pi'\theta \leftarrow P \mid Q \rightarrow \Delta\theta.\Delta'\theta}\ \ \Delta/\Delta'\ ,\ (\Gamma,\Pi)\ \cup^{\theta}\ (\Gamma',\Pi')$$

FIGURE 2: The Inference Algorithm

We now explain the rules. The general idea is of course to infer the minimal and most general contexts that fit the given program.
First, the rule *inact* assigns to *0* the void contexts.
In the *snd* (send) rule use is made of inference of type of data expressions –we assume such algorithm to be available– which gives the first premise. The second premise corresponds to the (recursive) inference of typing of the continuation process *P*. Then the condition for success of the rule is that the two inferred contexts Γ and Γ' unify, i.e. that the (data) type schemes at their common variables unify. This is (we assume) standard first order unification, which is decidable and yields in case of success a most general unifier θ. This is what is expressed by the side condition $\Gamma \cup^{\theta} \Gamma'$ to the rule. The conclusion obtains immediately by realizing that every context has to be instantiated by θ and Γ and Γ' have to be put together. Besides, the typing of the process at hand has to be updated with the type inferred for the channel *k*.
For the *rcv* (receive) rule there are two subcases. Once the continuation process *P* has been recursively typed one checks whether the data variable *x* is used in *P* or not. In the first case, the type assigned to *x* in the data context is recorded in the type of the channel as being received. If otherwise the variable is not used in *P*, then any type does since any value can be received. Therefore we update the channel *k* with the mark of input of a fresh type variable. According to the convention given above, this variable can be any one not occurring in the premise of the rule. A situation entirely similar to this last subcase arises in the next rule, in which the thrown out channel *k'* can be typed with any type whatsoever.
In the rule *catch* the point is to delete the bound name *k'* so that it does not occur in the resulting channel context. The rest of the manipulation has to do with considering the case in which the names *k* and *k'* coincide.
In the rest of the rules the novelty is the use of a unification algorithm over session types. This is expressed in the side conditions to the rules. Now session types are also first order trees if data types are and therefore such algorithm exists under the assumptions that we have established. This is actually the point on which all our development rests.

We can now prove the full correctness of our algorithm. For this we have to suppose correct the algorithm of data type inference.

Proposition 3 (Soundness of Type Inference). If $\Gamma;\ \Pi \leftarrow P \rightarrow \Delta$ then $\Gamma;\ \Pi \vdash P \triangleright \Delta$.

Proof. By induction on the rules of the inference algorithm.
Case *inact*: Immediate.
Case *snd*: Assume $\Gamma \vdash e : \delta$ (soundness of expression type inference) and $\Gamma'\ ;\ \Pi \vdash P \triangleright \Delta$ (induction hypothesis). Assume further $\Gamma \cup^{\theta} \Gamma'$ (side condition to the rule in the inference algorithm.) We then know both $\Gamma\theta \vdash e : \delta\theta$ and $\Gamma'\theta;\ \Pi\theta \vdash P \triangleright \Delta\theta$ because of the property of preservation of typing under type substitution in both type systems (expressions and session types). Now, since Γ and Γ' unify under θ, $\Gamma\theta \cup \Gamma'\theta$ is defined and, by weakening of both type systems, we get $\Gamma\theta \cup \Gamma'\theta \vdash e : \delta\theta$ and $\Gamma\theta \cup \Gamma'\theta;\ \Pi\theta \vdash P \triangleright \Delta\theta$. Hence, by rule *snd* of the session type system, $\Gamma\theta \cup \Gamma'\theta;\ \Pi\theta \vdash k!e;\ P \triangleright \Delta \prec\!\!+ k : \uparrow \delta\theta;\ (\Delta\theta)k$, as required.
Case rcv₁: Assume $\Gamma;\ \Pi \vdash P \triangleright \Delta$ (induction hypothesis) and $x \in \mathrm{dom}\ \Gamma$ (side condition to the rule.) We then know $\Gamma = \Gamma \setminus x \prec\!\!+ x : \Gamma x$ and therefore, because of the rule *rcv* of the type system, we have $\Gamma \setminus x;\ \Pi \vdash k?x.P \triangleright \Delta \prec\!\!+ k : \downarrow \Gamma x;\ \Delta k$, as required.
Case rcv₂: Assume $\Gamma;\ \Pi \vdash P \triangleright \Delta$ (induction hypothesis) and the side condition $x \notin \mathrm{dom}\ \Gamma$. *Also,* as indicated before, assume a^t fresh in $(\Gamma,\ \Pi,\ \Delta)$. Then $\Gamma \subseteq \Gamma \prec\!\!+ x : a^t$ and therefore by weakening,

$\Gamma \prec\!\!+ x: a^t ; \Pi \vdash P \rhd \Delta$. Now, using rule *rcv* of the type system, we arrive at the desired $\Gamma; \Pi \vdash k?x.P \rhd \Delta \prec\!\!+ k: \downarrow a^t ; \Delta k$.
Case thrw: Immediate. Notice that the side condition needs not be used.
Case ctch: Immediate once one writes $\Delta = (\Delta \setminus k') \cdot k' : \Delta k'$.
Case acc: Assume $\Gamma; \Pi \vdash P \rhd \Delta$ and side condition $\Pi a \cup^{\theta} \Delta k$. By preservation of typing under type substitutions we know $\Gamma\theta; \Pi\theta \vdash P \rhd \Delta\theta$. Now $\Delta\theta = (\Delta\theta \setminus k) \cdot k: (\Delta\theta)k$. And $(\Delta\theta)k = (\Delta k)\theta =$ (since $\Pi a \cup^{\theta} \Delta k) = (\Pi a)\theta = (\Pi\theta)a$, whence the required $\Gamma\theta; \Pi\theta \vdash acc\ a(k).P \rhd \Delta\theta \setminus k$ by use of the rule *acc* of the type system.
Case req: Identical to the preceding one.
Case conc: Use the unification side condition, preservation of typing under type substitution, and weakening, just the same as in case *snd*.

Proposition 4 (Completeness of Type Inference). $\Gamma; \Pi \vdash P \rhd \Delta$ implies $\Gamma_1; \Pi_1 \leftarrow P \rightarrow \Delta_1$ for contexts $\Gamma_1, \Pi_1, \Delta_1$ and type substitution θ such that $\Gamma_1\theta \subseteq \Gamma$, $\Pi_1\theta = \Pi$ and $\Delta_1\theta = \Delta$.

Proof. By induction on the rules of the type system.
Case inact: Define $\theta a^t = \Pi a$ for every $a \in dom\ \Pi$.
Case snd: Assume $\Gamma_1 \leftarrow e \rightarrow \delta_1$ with $\Gamma_1\theta \subseteq \Gamma$ and $\delta_1\theta = \delta$ for appropriate type substitution θ (this corresponds to completeness of the data expression type inference system.) Assume the induction hypothesis, i.e. $\Gamma_1' ; \Pi_1' \leftarrow P \rightarrow \Delta_1'$ with $\Gamma_1'\theta' \subseteq \Gamma$, $\Pi_1'\theta' = \Pi$ and $\Delta_1'\theta' = \Delta$ for appropriate type substitution θ'. Assume further that the type variables employed in Γ_1 and Γ_1' are disjoint. Hence without loss of generality we can also take θ and θ' to possess disjoint domains. Now the union σ of these two substitutions makes both $\Gamma_1\sigma \subseteq \Gamma$ and $\Gamma_1'\sigma \subseteq \Gamma$, which means that there is a subcontext of Γ that is a type substitution instance of both Γ_1 and Γ_1'. Hence these two have a most general unifier ζ and we can apply rule *snd* of the inference algorithm to obtain $\Gamma_1\zeta \cup \Gamma_1'\zeta ; \Pi_1'\zeta \leftarrow k!e ; P \rightarrow \Delta_1'\zeta \prec\!\!+ k : \uparrow \delta\zeta ; (\Delta_1'\zeta)k$. Also because ζ is the m.g.u. of Γ_1 and Γ_1', we know that there exists ζ' such that $\sigma = \zeta\zeta'$. Further, since θ' is the subset of σ acting on the type variables of Γ_1', Π_1' and Δ_1', we have $\Gamma_1'\theta' = \Gamma_1'\zeta\zeta'$ and similarly for Π_1' and Δ_1'. Therefore in the inference above we have what is required to prove, namely $(\Gamma_1\zeta \cup \Gamma_1'\zeta)\zeta' = \Gamma_1\zeta\zeta' \cup \Gamma_1'\zeta\zeta' = \Gamma_1\theta \cup \Gamma_1'\theta' \subseteq \Gamma$, $\Pi_1'\zeta\zeta' = \Pi_1'\theta' = \Pi$ and $[\Delta_1'\zeta \prec\!\!+ k : \uparrow\delta\zeta ; (\Delta_1'\zeta)k]\zeta' = \Delta \prec\!\!+ k : \uparrow\delta; \Delta k$, where the latter can be easily checked by just distributing the substitution ζ'.
Case rcv: Assume the induction hypothesis, i.e. $\Gamma_1; \Pi_1 \leftarrow P \rightarrow \Delta_1$ with $\Gamma_1\theta \subseteq \Gamma \prec\!\!+ x: \delta$, $\Pi_1\theta = \Pi$ and $\Delta_1\theta = \Delta$ for appropriate type substitution θ.
If now $x \in dom\ \Gamma_1$ then we can apply rule rcv_1 of the inference algorithm to get $\Gamma_1 \setminus x; \Pi_1 \leftarrow k?x.P \rightarrow \Delta_1 \prec\!\!+ k : \downarrow\Gamma_1 x; \Delta_1 k$. Moreover, we have $(\Gamma_1 \setminus x)\theta \subseteq \Gamma$, which follows from $\Gamma_1\theta \subseteq \Gamma \prec\!\!+ x: \delta$, and, by hypothesis, $\Pi_1\theta = \Pi$. Finally, $[\Delta_1 \prec\!\!+ k: \downarrow\Gamma_1 x; \Delta_1 k]\theta = \Delta \prec\!\!+ k: \downarrow\delta; \Delta k$, which can be checked by distributing θ and using $\Delta_1\theta = \Delta$ as well as $(\Gamma_1 x)\theta = (\Gamma_1\theta)x = \delta$.
If otherwise $x \notin dom\ \Gamma_1$, we choose a sufficiently fresh type variable a^t and apply rule rcv_2 to get $\Gamma_1; \Pi_1 \leftarrow k?x.P \rightarrow \Delta_1 \prec\!\!+ k: \downarrow a^t ; \Delta_1 k$ and taking $\theta' = \theta \cdot a^t \rightarrow \delta$, the required conditions $\Gamma_1\theta' \subseteq \Gamma$, $\Pi_1\theta' = \Pi$, and $[\Delta_1 \prec\!\!+ k: \downarrow a^t ; \Delta_1 k]\theta' = \Delta \prec\!\!+ k: \downarrow\delta; \Delta k$ all hold.
Notice that here is a place where we introduce type variables into the inferred type scheme. It should be clear that the procedure works whatever freshness conditions are imposed to type variable a^t besides the basic one we have agreed upon, namely that a^t is fresh in the contexts Γ_1, Π_1 and Δ_1.
Case thrw: Similar to the last case above. A sufficiently fresh type variable is introduced.
Case ctch: Immediate once one notes that $(\Delta \cdot x : \alpha) \setminus x = \Delta$.
Case acc: Assume the induction hypothesis, i.e. $\Gamma_1 ; \Pi_1 \leftarrow P \rightarrow \Delta_1$, with $\Gamma_1\theta \subseteq \Gamma$, $\Pi_1\theta = \Pi$ and $\Delta_1\theta = \Delta \cdot k: \Pi a$. Notice that $(\Pi_1 a)\theta = (\Pi_1\theta)a = \Pi a = (\Delta_1\theta)k = (\Delta_1 k)\theta$. Therefore, $\Pi_1 a \cup^{\zeta} \Delta_1 k$ and $\theta = \zeta\zeta'$. We can then apply rule *acc* of the inference algorithm to obtain $\Gamma\zeta ; \Pi\zeta \leftarrow acc\ a(k).\ P \rightarrow \Delta\zeta \setminus k$. And, besides, $\Gamma_1\zeta\zeta' \subseteq \Gamma$, $\Pi_1\zeta\zeta' = \Pi$ and $(\Delta_1\zeta \setminus k)\zeta' = \Delta\zeta\zeta' \setminus k = \Delta$, as required.
Case req: Identical to the preceding one.
Case conc: Similar to first case *snd*.

Soundness and completeness, together with unicity of inference of typing (up to choice of type variables) give the result on *principal type scheme*. Actually a principal type scheme for P is, by

definition, one that is assignable to P and that satisfies the conditions exposed in the completeness theorem for all the typings $\Gamma; \Pi \vdash P \triangleright \Delta$ assignable to P.

4. CONCLUSION

The classical result of (implicitly) simply typed λ calculus, of existence and effective computability of principal type scheme of any typable term can be extended to session types. This fact has been mentioned in passing in [4] and [1] but only now has it been proven formally. What remains for us to make this result complete is to extend our present development to types of choice (branching) and recursive types. Of these, the latter seem to constitute the interesting problem. But, as pointed out above, the key point on which the given algorithm and proofs rest is the existence of a unification algorithm for session types. And, as remarked out in e.g. [12], this algorithm can be extended to unification of regular trees with all other details of the proof holding without modifications.
We have also formalized a great part of the present development in the proof assistant Agda [6], which implements a version of constructive type theory. Besides, we have implemented the inference algorithm in Haskell. All this is available in [13] and we expect to soon complete the formalization of the whole development. Notice that the treatment presented in this paper does not depend on identifying α-convertible terms and is therefore amenable to direct formalization.

Acknowledgements Ernesto Copello was partially supported by a graduate student scholarship from ANII (Agencia Nacional de Investigación e Innovación), Uruguay.

5. REFERENCES

[1] N. Yoshida and V. T. Vasconcelos. “Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication”. In *1st International Workshop on Security and Rewriting Techniques*, volume 171(4) of *ENTCS*, pages 73–93. Elsevier, 2007.

[2] K. Takeuchi, K. Honda, and M. Kubo. “An interaction-based language and its typing system”. In Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 1994.

[3] K. Honda. “Types for dyadic interaction”. In Eike Best, editor, *CONCUR’93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57208-2_35.

[4] K. Honda, V. T. Vasconcelos, and M. Kubo. “Language primitives and type disciplines for structured communication-based programming”. In *ESOP’98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.

[5] S. J. Gay and M. Hole. “Subtyping for session types in the pi calculus”. *Acta Inf.*, pages 191–225, 2005.

[6] U. Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[7] L. G. Mezzina. “How to infer finite session types in a calculus of services and sessions”. In *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION’08, pages 216–231, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] M. Neubauer and P. Thiemann. “An implementation of session types”. In Bharat Jayaraman, editor, *PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[9] M. Sackman and S. Eisenbach. “Session Types in Haskell: Updating Message Passing for the 21st Century”. Technical report, June 2008.

[10] M. P. Jones. “Type classes with functional dependencies”. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP ’00, pages 230–244, London, UK, 2000. Springer-Verlag.

[11] R. Pucella and J. A. Tov. “Haskell session types with (almost) no class”. *SIGPLAN Not.*, 44(2):25–36, September 2008.

[12] F. Cardone and M. Coppo. “Type inference with recursive types: Syntax and semantics”. *Inf. Comput.*, 92(1):48–80, 1991.

[13] Ernesto Copello. Inferencia de tipos de sesión. Master’s thesis, Universidad ORT Uruguay, 2012.

INSTRUCTIONS TO CONTRIBUTORS

The International Journal of Logic and Computation aims to promote the growth of logic and computing research from the perspectives of logic, mathematics and computer science, but emphasizes semantics of programs, in contrast with the traditional treatment of formal languages as sets of strings. IJLP promote this new field with its comprehensive selection of technical scientific papers and regular contributions such as letters, reviews and discussions for logical systems using classical and non-classical logic, constructive logic, categorical logic, modal logic, type theory, logical issues in logic programming, knowledge-based systems and automated reasoning programing; logical programming issues in knowledge representation, non-monotonic reasoning, logics and semantics of programming and applications of logic in hardware and VLSI.

To build its International reputation, we are disseminating the publication information through Google Books, Google Scholar, Directory of Open Access Journals (DOAJ), Open J Gate, ScientificCommons, Docstoc and many more. Our International Editors are working on establishing ISI listing and a good impact factor for IJLP.

The initial efforts helped to shape the editorial policy and to sharpen the focus of the journal. Starting with Volume 4 2013, IJLP will aim to appear with more focused issues. Besides normal publications, IJLP intend to organized special issues on more focused topics. Each special issue will have a designated editor (editors) – either member of the editorial board or another recognized specialist in the respective field.

We are open to contributions, proposals for any topic as well as for editors and reviewers. We understand that it is through the effort of volunteers that CSC Journals continues to grow and flourish.

IJLP LIST OF TOPICS

The realm of International Journal of Logic and Computation (IJLP) extends, but not limited, to the following:

- Categorical Logic
- Classical and Non-Classical Logic
- Constructive Logic
- Logic Representation Techniques
- Logical Programming Issues in Knowledge Representa
- Modal Logic
- Non-Monotonic Reasoning
- Programming Reasoning Test Collection
- Semantic Representation in Logic Programming
- Challenges in Natural Language and Reasoning
- Computer Logical Reasoning
- Knowledge-Based Systems and Automated Reasoning Pr
- Logical Issues in Logic Programming
- Logics and Semantics of Programming
- Natural Language
- Programming Expressiveness
- Reasoning Systems
- State-Based Semantics

CALL FOR PAPERS

Volume: 4 - Issue: 1

i. Submission Deadline: January 31, 2013 **ii. Author Notification:** March 15, 2013

iii. Issue Publication: April 2013

CONTACT INFORMATION

Computer Science Journals Sdn BhD
B-5-8 Plaza Mont Kiara, Mont Kiara
50480, Kuala Lumpur, MALAYSIA

Phone: 006 03 6207 1607
006 03 2782 6991

Fax: 006 03 6207 1697

Email: cscpress@cscjournals.org

CSC PUBLISHERS © 2012
COMPUTER SCIENCE JOURNALS SDN BHD
M-3-19, PLAZA DAMAS
SRI HARTAMAS
50480, KUALA LUMPUR
MALAYSIA

PHONE: 006 03 6207 1607
006 03 2782 6991

FAX: 006 03 6207 1697
EMAIL: cscpress@cscjournals.org