فصلنامه‌ی علمی و پژوهشی شریف
تابستان ۱۳۸۶، شماره‌ی سی و هشتم، ص. ۵۹-۶۶

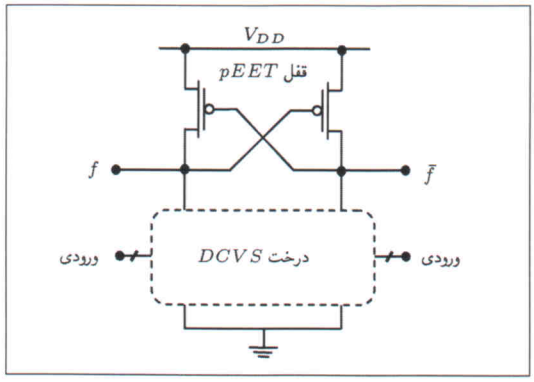
ارائه‌ی الگوریتم نوین ساده‌سازی DCVS بر پایه‌ی قانون جدید تزویج سطوح

امید کاوه‌ای (دانشجوی دکتری)
کیوان ناوی (استادیار)
تورج نیکویین (دانشجوی دکتری)
دانشکده‌ی مهندسی برق و کامپیوتر، دانشگاه شهید بهشتی

در این نوشتار روشی نوین برای ساده‌سازی درخت پایین‌بر که نقش بسیار اساسی در پیاده‌سازی منطق تفاضلی دارد پیشنهاد خواهد شد. این الگوریتم بر قانون جدید تزویج سطوح، که از قابلیت ساده‌سازی بیشتری برخوردار است، مبتنی است. الگوریتم ارائه شده، با استفاده از این قانون و ترکیب آن با قوانین پیشین، روشی نوین برای طراحی شبکه‌ی درخت پایین‌بر معرفی می‌کند. در بیان الگوریتم پیشنهادی، برای نخستین بار از بیان ماتریس‌گونه‌ی رابطه بین ترتیب ورودی‌های و متغیرهای کنترلی در یک نمودار تصمیم‌گیری استفاده شده است. با به‌کارگیری این روش می‌توان به میزان قابل توجهی سرعت و فضای اشغالی را بهبود بخشید. نتایج حاصل از اِعمالِ الگوریتم پیشنهادی روی برخی از مدارات benchmark نشان‌دهنده‌ی بهبود چشم‌گیر در کاهش تعداد گره به‌کار رفته در پیاده‌سازی مدارات است.

۱. مقدمه

شبکه‌ی nFET یا پایین‌بر نقش بسیار مهمی در پیاده‌سازی مدارات منطق تفاضلی ایفا می‌کند. به‌عنوان مثال منطق DCVS[۱] روشی است برای طراحی مدارات CMOS که از مزایای بسیار زیادی برخوردار است از میان عمده‌ترین این مزایا می‌توان به‌کاهش تأخیر مدار، چگالی چیدمان بالاتر، مصرف توان کم‌تر و انعطاف‌پذیری منطقی بیشتر اشاره کرد. درخت nFET پایه‌ی بسیاری از مدارات منطقی دو ریلی از قبیل DDCVS [۲]، SSDL [۳]، ECDL [۴]، DCSL [۵] و ... تلقی می‌شود.[۱-۸] ساختار کلی یک مدخل[۶] بر پایه‌ی منطق DCVS در شکل ۱ نشان داده شده است. این ساختار دو ترانزیستور pFET را شامل می‌شود که به‌صورت ضربدری و از طریق مدخل و زه[۷] به یکدیگر متصل‌اند و تشکیل یک قفل[۸] را می‌دهند که خروجی‌های f و $\overline{f}$ را مهیا می‌سازند. این قفل توسط شبکه‌یی از ترانزیستورهای nFET که از آن به‌عنوان درخت DCVS نام می‌بریم فعال می‌شود. در این شبکه که در ادامه به روش‌های موجود طراحی آن اشاره خواهیم کرد، مسیرهایی باید وجود داشته باشند که بتوانند تابع مورد نظر و مکمل آن را توسط شبکه pFET پیاده‌سازی کنند. به‌عبارت دیگر وقتی مسیر خروجی f ما از طریق مسیری در درخت DCVS به زمین متصل می‌شود، خروجی f ما صفر منطقی خواهد شد. همین عمل باعث فعال شدن pFET مقابل شده و مسیری از تغذیه به خروجی $\overline{f}$ متصل خواهد شد، و در این خروجی یک منطقی دیده می‌شود. به‌عبارت بهتر اگر یک خروجی صفر منطقی شود، عمل قفل‌کردن خروجی متقابل را کنترل خواهد کرد.



شکل ۱. ساختار پایه‌ی یک مدخل در منطق DCVS.

۲. روش‌های طراحی درخت DCVS

۱.۲. منطق AOI/OAI

یکی از روش‌های طراحی درخت nFET روشی است مبتنی بر استفاده از فرم‌های منطقی AOIیاOAI، که در آن به‌منظور پیاده‌سازی یک ساختار دوریلی می‌بایست ابتدا توسط قانون دمورگان هر دو تابع f و $\overline{f}$ را به‌دست آورد و سپس هر یک از آنها را به‌صورت کاملاً جداگانه پیاده‌سازی کرد. در این روش از ۰ و ۱های جدول کارنو به‌منظور پیاده‌سازی توابع مورد نظر بهره می‌بریم. به‌عبارت دیگر در این روش توابع f و $\overline{f}$ ساخته، و سپس به‌طورکاملاً مستقل از یکدیگر پیاده‌سازی می‌شوند. این روش به‌علت در نظر نگرفتن بخش‌های

مشترک و یکسان در طراحی توابع f و $\overline{f}$ از نظر طراحی بسیار ابتدایی و ساده است.[۳]

۲.۲. درخت منطقی ساخت‌یافته

در شیوه‌ی طراحی ساخت‌یافته‌ی درخت nFET [۱و۲] از جدول تابع به‌عنوان پایه‌یی برای طراحی درختی که از قابلیت پیاده‌سازی f و $\overline{f}$ برخوردار است استفاده می‌شود. در روش قبلی چون هر شاخه به‌صورت کاملاً مجزا ساخته می‌شد، اشتراک‌های موجود در پیاده‌سازی اصلاً در نظر گرفته نمی‌شد؛ به‌عبارت دیگر مدار دارای دو بخش جدا اما با اشتراک‌های احتمالی زیادی بود که در نظر گرفته نشده بودند. ولی در روش فعلی طراح می‌تواند در طرح خود امکان اشتراک ترانزیستورها را هم در نظر داشته باشد. این عمل ممکن است به کاهش پیچیدگی در طرح، کاهش فضای اشغالی و افزایش سرعت مدارات بینجامد.[۹و۱۰] این روش را با ارائه‌ی مثالی توضیح خواهیم داد. جدول تابع f را در جدول ۱ مشاهده می‌کنید. در این جدول صحت در سطرها، و متغیرهای ورودی به‌ترتیبِ ارزش از بالا به پایین قرار گرفته‌اند.[۳و۱۱]

متغیرهای ورودی دنباله‌های کاملاً مشخصی دارند. به‌طور منطقی یک ورودی که با «۱» نشان داده شده یعنی خود متغیر ورودی و یک ورودی که با «۰» نشان داده شده است یعنی مکمل متغیر ورودی. به‌منظور پیاده‌سازی این درخت نیز ازnFETهایی با چشمه‌های متصل (شکل ۲ الف) استفاده می‌کنیم و آن را با بلوکی که در شکل ۲ ب آمده نشان می‌دهیم.

یکی از ترانزیستورها توسط ورودی، و دیگری توسط مکمل آن کنترل می‌شود. هر ترانزیستور زه‌های مجزایی دارد که به a و b متصل است. aو b هرکدام می‌توانند صفر یا یک، و یا خروجی طبقات بالاتر باشند. این ساختار تنها یک خروجی به‌نام u دارد. طرز محاسبه‌ی u برحسب کنترل‌ها هم در شکل ۲الف آمده است. به‌عبارت بهتر، ساختار شکل ۲ به‌طورکلی بیانگر یک تقویت‌کننده‌ی ۲ در ۱ است. همان‌طورکه در شکل ۲ ب مشاهده می‌شود، سمتی که علامت آن (-) است توسط $\overline{x}$، و سمتی که علامت آن (+) است توسط x کنترل می‌شود.[۱۲-۱۴]

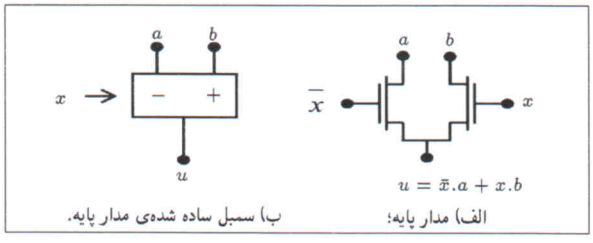
حال تابع f را با این ساختار پیاده‌سازی می‌کنیم. شکل ۳ نمایانگر پیاده‌سازی این تابع توسط دنباله‌یی از این بلوک‌ها است؛ از بالا به پایین مسیرهای تابع و مکمل تابع در نظر گرفته شده‌اند. در سطح اول که با متغیر ورودی A کنترل می‌شود، تمامی زوج‌های کنترلی ۱ ۰ در نظر گرفته شده‌اند، که ۴ زوج کنترلی از این نوع را شامل می‌شوند. سطح دوم نیز که توسط B کنترل می‌شود، دارای دو زوج ۱ ۰ است که هرکدام به دو زوج سطح اول متصل‌اند؛ در سطح سوم نیز تنها یک زوج ۱ ۰ وجود دارد و آن هم به‌عنوان ورودی دو گره[۹] سطح دوم اختیار می‌شود؛ این سطح توسط متغیر ورودی C کنترل می‌شود. ساختار مذکور پیاده‌سازی تابع موردنظر را نشان می‌دهد. در نتیجه در سطح اول درخت ورودی‌ها (همان ۰ و ۱های تابع) در جدول صحت خواهند بود. همان‌طور که در شکل ۳ ملاحظه می‌کنید، این ساختار با در نظر گرفتن وضعیت کنترل‌ها و ورودی‌ها از قابلیت کاهش تعداد بلوک نیز برخوردار است.

این قابلیت طی دو قانونی که در ادامه آمده است بررسی خواهد شد. قانون اول ساده‌سازی درخت nFET براساس رابطه‌ی (۲-۱) تشریح، و در شکل ۴ نمایش داده شده است. طبق این قانون، گرهی که هر دو ورودی آن یکسان باشد، گرهی است که تأثیری در عملکرد درخت ندارد و قابل حذف است.[۱و۲]

$$u = \bar{A}.a + A.a = (A + \bar{A}).a = a \qquad (۱-۲)$$

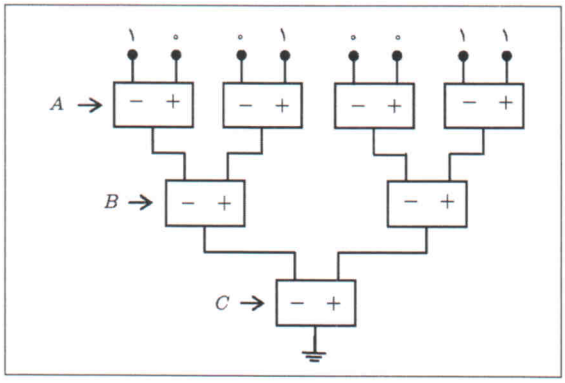
جدول ۱. جدول صحت تابع خاص $f(f = \bar{A}.\bar{B}.\bar{C} + A.B + B.C)$.

f	۱	۰	۰	۱	۰	۰	۱	۱
A	۰	۱	۰	۱	۰	۱	۰	۱
B	۰	۰	۱	۱	۰	۰	۱	۱
C	۰	۰	۰	۰	۱	۱	۱	۱

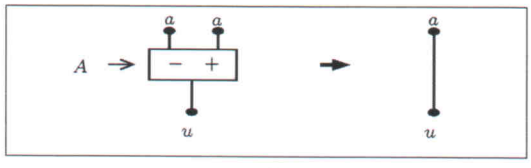


ب) سمبل ساده شده‌ی مدار پایه. الف) مدار پایه؛

شکل ۲. ساختار بلوک پایه در درخت DCVS.



شکل ۳. ساختار عمومی درخت تابع سه ورودی (و در اینجا تابع خاص).



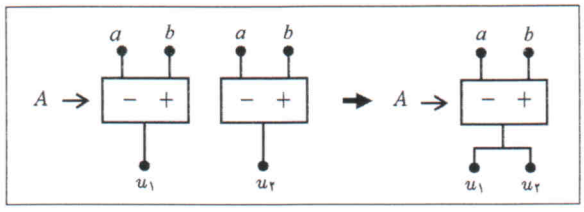
شکل ۴. حذف یک جفت nFET.

به‌عبارت دیگر، ترانزیستورها در این حالت هیچ‌گونه عملیات منطقی انجام نمی‌دهند (غیر از عملیات عبور)، و درنتیجه می‌توان یکی از ورودی‌ها را به خروجی وصل کرد. این عمل را اصطلاحاً «اتصال کوتاه» نیز می‌نامند.

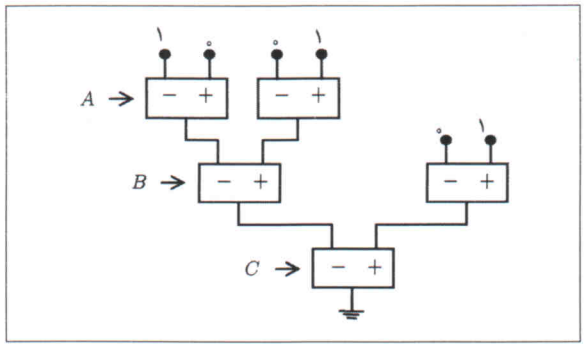
حال اگر دو گره در درخت باشند که ورودی‌های آنها نظیر به نظیر یکسان و کنترل آنها نیز کاملاً مشابه باشد، می‌توان آن دو گره را معادل در نظر گرفت و طبق شکل ۵ با در نظر گرفتن یکی، دیگری را حذف کرد. این عمل را «قانون دوم ساده‌سازی» می‌نامند.[۱و۲]

طبق شکل ۵، $u_1 = u_2$ است. با اجرای این قوانین روی درخت شکل ۳ می‌توان شاهد ساده‌سازی انجام گرفته بود (شکل ۶).

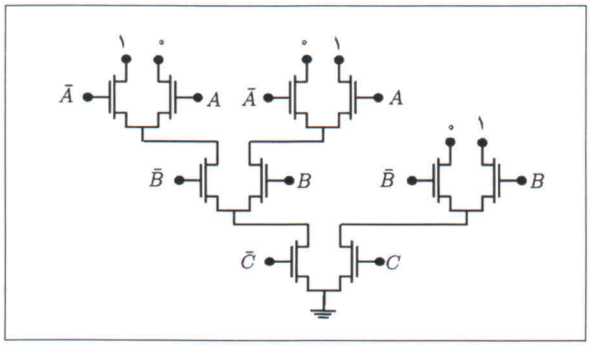
حال اگر تمامی بلوک‌ها را با معادل ترانزیستوری آنها جایگزین کنیم به شکل ۷ خواهیم رسید. پیاده‌سازی کامل این درخت، با منطق DCVSرا هم در شکل ۸ با اضافه شدن قفل به درخت nFET



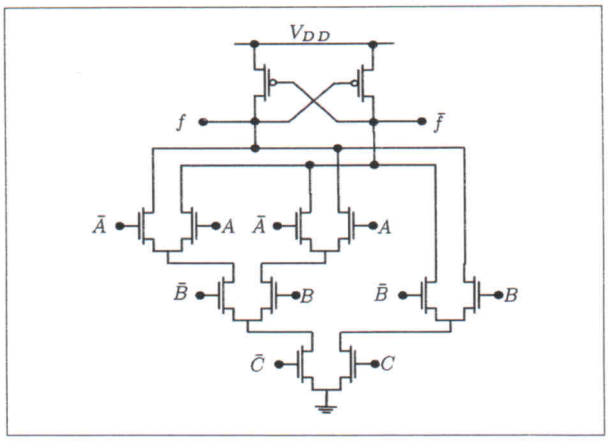
شکل ۵. حذف گره اضافی در درخت.



شکل ۶. ساده‌سازی درخت پس از اجرای قوانین اول و دوم.



شکل ۷. جایگزینی بلوک‌های پایه با ساختار ترانزیستوری.



شکل ۸. ساختار کامل یک تابع پیاده‌شده توسط منطق DCVS.

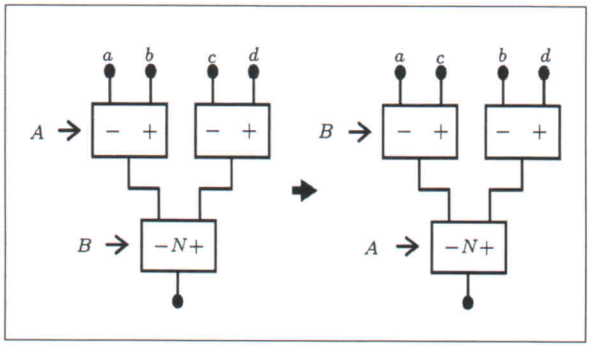
مشاهده می‌کنید. چنان که در این ساختار نشان داده شده، ورودی‌های «۰» درخت، به خروجی f و ورودی‌های «۱» درخت به خروجی $\overline{f}$ متصل می‌شوند؛ و این به‌خاطر عمل قفل‌کردن یا شبکه‌ی pFET است.

۳. الگوریتم نوین ساده‌سازی درخت

۱.۳. مفاهیم اولیه‌ی به‌کار رفته در الگوریتم

در این قسمت به ارائه‌ی الگوریتمی نوین در طراحی انواع مدارات منطق تفاضلی یا دوریلی خواهیم پرداخت، که نتیجه‌ی آن ساده‌تر شدن مدارات و درنتیجه کاهش فضای اشغالی و افزایش سرعت آنها است. در این الگوریتم از قانون جدیدی استفاده کرده‌ایم که در ابتدای این قسمت به آن اشاره خواهیم کرد. این قانون راکه قانون تزویج نام نهاده‌ایم، به‌دلیل ترکیب با قوانین اول و دوم ساده‌سازی «قانون سوم ساده‌سازی» نیز می‌نامند. شکل ۹ به همراه رابطه‌ی (۳-۱) صراحتاً به چگونگی عملکرد این قانون روی وضعیت گره‌ها اشاره دارد.

A، B، C، D هر کدام می‌توانند نتیجه‌ی یک زیردرخت، یا در ساده‌ترین حالت ۰ و ۱ باشند. به‌عبارت دیگر جمله‌ی فوق بیانگر این



شکل ۹. تزویج یک شاخه‌ی پایه از درخت DCVS نسبت به گره N.

نکته است که عمل تزویج فقط شاخه‌ی دودویی پایه را پوشش نمی‌دهد بلکه آن را روی هر گره و در هر سطحی (به‌جز اولین سطح) می‌توان اعمال کرد.

$$u = \bar{B}(\bar{A}.a + A.b) + B(\bar{A}.c + A.d) = \bar{B}.\bar{A}(a) + \bar{B}.A(b) + \bar{A}.B(c) + A.B(d) = \bar{A}(\bar{B}.a + B.c) + A(\bar{B}.b + B.d) \quad (۱-۳)$$

به‌طور مثال اعمال قانون تزویج روی گرهی در سطح سه، طبق رابطه‌ی (۳-۲) قابل ملاحظه است.

$$\begin{aligned} u_۱ &= \bar{C}(\bar{B}(\bar{A}.a + A.b) + B(\bar{A}.c + A.d)) \\ &+ C(\bar{B}(\bar{A}.e + A.f) + B(\bar{A}.g + A.h)) \\ u_۲ &= \bar{A}(\bar{B}(\bar{C}.a + C.e) + B(\bar{C}.c + C.g)) \\ &+ A(\bar{B}(\bar{C}.b + C.f) + B(\bar{C}.d + C.h)) \\ u_۳ &= \bar{B}(\bar{C}(\bar{A}.a + A.b) + C(\bar{A}.e + A.f)) \\ &+ B(\bar{C}(\bar{A}.c + A.d) + C(\bar{A}.g + A.h)) \end{aligned} \quad (۲-۳)$$

بسیار ساده قابل اثبات است که:

$$u_۱ = u_۲ = u_۳ = u$$

البته هرکدام از آنها به یک زیردرخت کاملاً متفاوت با دیگری اشاره دارد. اگر تابع سوئیچ ما به‌صورت $F(A,B,C)$ تعریف شود، آنگاه این تابع دارای سه درخت کاملاً مستقل خواهد بود. به‌عبارت دیگر، اگر f یک تابع mمتغیره (بیتی) باشد، آنگاه تعداد m درخت کاملاً مستقل از این درخت می‌توان داشت، طوری که همگی تابع F را پیاده‌سازی می‌کنند. طریقه‌ی پیدا کردن ترتیب قرارگیری کنترل‌ها در این درخت‌ها (که باعث تفاوت می‌شود) به‌صورت ماتریسی آمده است:

$$\begin{bmatrix} x_۱ & x_۲ & x_۳ & \dots & x_m \\ x_۲ & x_۳ & \ddots & x_m & x_۱ \\ x_۳ & \ddots & x_m & x_۱ & x_۲ \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_m & x_۱ & x_۲ & \dots & x_{m-۱} \end{bmatrix}_{m\times m}$$

اگر به سطر و ستون این ماتریس که ما آن را «ماتریس کنترل» نام نهاده‌ایم توجه کنید، خواهید دید که در هر سطر و ستون تنها یکبار از یک کنترل خاص استفاده شده است و در ضمن تمامی ستون‌ها و سطرها، دارای تمامی کنترل‌ها هستند. به‌بیان دقیق‌تر، این ماتریس نمایشی دیگر از مجموعه مرزی[۱۰] است. ثابت شده است که برای هر تابع $f(x)$ همیشه پیاده‌سازی بهینه توسط یکی از ترتیب‌های مجموعه‌مرزی حاصل خواهد شد.[۱۵] البته بر این اثبات و سایر الگوریتم‌های گوناگون ارائه شده اصلاحاتی نیز صورت گرفته است.[۹،۱۶و۱۷] این ماتریس نسبت به قطر اصلی متقارن است. کمی پیش‌تر انواع درخت‌های موجود و مستقل را برای تابعی با سه کنترل بیان کردیم، در اینجا به روش بیان ماتریسی آنها خواهیم پرداخت:

$$\begin{pmatrix} A & B & C \\ B & C & A \\ C & A & B \end{pmatrix}_{۳\times۳}$$

به‌منظور بررسی ساده‌سازی در درخت می‌توان سطرها یا ستون‌های ماتریس کنترل را به‌عنوان کنترل در نظر گرفت.

ویژگی‌های عمده‌یی که ماتریس کنترل برای ما ایجاد می‌کند، عبارت‌اند از:

۱. یک کنترل خاص به تمام سطوح اعمال می‌شود.

۲. از تکرار اعمال یک کنترل خاص به یک سطح جلوگیری می‌شود.

۳. درخت‌هایی کاملاً مستقل ایجاد می‌شوند که بهترین ترتیب اعمال کنترل‌ها را در بین خود دارند.

در کنار ماتریس کنترل ماتریس حالت را قرار می‌دهیم که شامل تمامی حالات ترکیبی کنترل‌ها است. از تأثیر ماتریس حالات روی ماتریس کنترل می‌توان به ماتریس ورودی‌ها دست یافت.

در زیر ماتریس حالات راکه یک ماتریس $m \times ۲^m$ است، ملاحظه می‌کنید.

$$\begin{pmatrix} ۰ & ۰ & \dots & ۰ \\ ۱ & ۰ & \dots & ۰ \\ ۰ & ۱ & \dots & ۰ \\ ۱ & ۱ & \dots & ۰ \\ ۰ & ۰ & \dots & ۰ \\ \vdots & \vdots & & \vdots \\ ۰ & ۱ & \dots & ۱ \\ ۱ & ۱ & \dots & ۱ \end{pmatrix}_{۲^m\times m}$$

اثری که از آن صحبت کردیم دارای شکل ریاضی عملکرد قانون تزویج در مدار است. به‌عبارت دیگر، کار اصلی قانون تزویج سطوح جابه‌جا کردن ترتیب قرارگیری کنترل‌ها است. از تأثیر ماتریس حالات در ماتریس کنترل می‌توان به‌ترتیب قرارگرفتن ورودی‌های در هر کدام از درخت‌های مستقل (ترتیب کنترل‌ها) دست یافت. به‌بیان دقیق‌تر، نتیجه‌ی این اثر که ما آن را ماتریس ورودی‌ها نام نهاده‌ایم، در ستون iام

خود دارای ترتیبی از ورودی‌هاست که دقیقاً متناظر با درخت مستقلی است که ترتیب کنترل‌های آن با ستون iام ماتریس کنترل یکسان است. شایان ذکر است که اثری که از آن صحبت شد دارای دو عملگر تعریف شده است: عملگر ضرب ($\vec{\square}$):

$$\circ \vec{\cdot} x_i = \bar{x}_i$$

$$\text{۱} \vec{\cdot} x_i = x_i$$

و عملگر جمع ($\hat{+}$):

$$x_\text{۱} \hat{+} x_\text{۲} \hat{+} \ldots \hat{+} x_m = x_\text{۱} x_\text{۲} \ldots x_m = \text{۱۱}\ldots\text{۱} = I_{\text{۲}^m - \text{۱}}$$

و برای تابع سه متغیری داریم:

$$A \hat{+} \bar{B} \hat{+} \bar{C} = A.\bar{B}.\bar{C}. = \text{۱}\circ\circ = I_\text{۱}$$

$$\bar{A} \hat{+} B \hat{+} C = \bar{A}.B.C. = \circ\text{۱۱} = I_\text{۶}$$

در اینجا نمونه‌یی از این عملیات، برای $m = \text{۳}$ آمده است:

$$\begin{pmatrix} \circ & \circ & \circ \\ \text{۱} & \circ & \circ \\ \circ & \text{۱} & \circ \\ \text{۱} & \text{۱} & \circ \\ \circ & \circ & \text{۱} \\ \text{۱} & \circ & \text{۱} \\ \circ & \text{۱} & \text{۱} \\ \text{۱} & \text{۱} & \text{۱} \end{pmatrix}_{\text{۲۳}\times\text{۳}} \cdot \begin{pmatrix} A & B & C \\ B & C & A \\ C & A & B \end{pmatrix}_{\text{۳}\times\text{۳}} = \begin{pmatrix} I_\circ & I_\circ & I_\circ \\ I_\text{۱} & I_\text{۲} & I_\text{۴} \\ I_\text{۲} & I_\text{۴} & I_\text{۱} \\ I_\text{۳} & I_\text{۶} & I_\text{۵} \\ I_\text{۴} & I_\text{۱} & I_\text{۲} \\ I_\text{۵} & I_\text{۳} & I_\text{۶} \\ I_\text{۶} & I_\text{۵} & I_\text{۳} \\ I_\text{۷} & I_\text{۷} & I_\text{۷} \end{pmatrix}_{\text{۲۳}\times\text{۳}}$$

چنان که مشاهده می‌کنید، ستون اول ماتریس ورودی‌ها بیان‌گر همان چینش ابتدایی و ذاتی ورودی‌ها به‌ازاء اعمال چینش کنترل اولیه (ستون اول ماتریس کنترل) است. ماتریس ورودی به‌طور ساده بیان می‌کند که اگر ستون iام ماتریس کنترل به مدار اعمال شد، آنگاه ستون iام ماتریس ورودی هم به‌عنوان ورودی به مدار اعمال خواهد شد؛ و به این ترتیب تمامی خصوصیات درخت‌های مستقل را در اختیار داریم.

در این روش مدل عملکرد را برای یک m پیدا می‌کنیم؛ مثلاً درمورد تابعی که در بخش قبلی بررسی شد در جدول ۲ می‌توان فرم‌های مختلف ورودی را به ازاء اعمال ترتیب‌های مختلفی از کنترل‌ها مشاهده کرد. رابطه‌ی زیر نیز طریقه‌ی پیدا کردن یک ترتیب مستقل از کنترل‌ها، از روی ترتیب مستقل دیگری از کنترل‌ها را، توسط قانون تزویج نشان می‌دهد.

$$u = \bar{C}\big(\bar{B}\big(\bar{A}.a + A.b\big) + B\big(\bar{A}.c + A.d\big)\big)$$
$$+C\big(\bar{B}\big(\bar{A}.e + A.f\big) + B\big(\bar{A}.g + A.h\big)\big) \Rightarrow F(A,B,C)$$

$$(*)\ \big(\bar{\Phi}\big(\bar{\Psi}.\alpha + \Psi.\beta\big) + \Phi\big(\bar{\Psi}.\gamma + \Psi.\varphi\big)\big)$$
$$= \big(\bar{\Psi}\big(\bar{\Phi}.\alpha + \Phi.\gamma\big) + \Psi\big(\bar{\Phi}.\beta + \Phi.\varphi\big)\big)$$
$$u = \bar{A}\big(\bar{C}\big(\bar{B}.a + B.c\big) + C\big(\bar{B}.e + B.g\big)\big)$$
$$+ A\big(\bar{C}\big(\bar{B}.b + B.d\big) + C\big(\bar{B}.f + B.h\big)\big)$$
$$\Rightarrow F(B,C,A)$$

در ادامه به ارائه‌ی الگوریتمی برای استفاده از این قانون خواهیم پرداخت. هدف ما در این الگوریتم اثر دادن قانون تزویج روی درخت اولیه، و درنتیجه رسیدن به ترتیب‌های گوناگونی از کنترل‌ها است، به گونه‌یی که پس از عمل‌کردن قوانین اول و دوم ساده‌سازی بتوان به بهینه‌ترین پیاده‌سازی دست یافت. مراحل انجام الگوریتم در شکل ۱۰ نشان داده شده است. درخت ورودی به این الگوریتم همانند الگوریتم قدیمی[۱و۲] درخت دودویی اولیه‌یی است که توسط ترتیب کنترلی BBD [۱۱] به‌دست آمده است، و چینش کنترل‌های آن هم طبق یکی از ستون‌های ماتریس کنترل است. تعداد سطوحی که می‌توان عمل تزویج را روی آنها انجام داد، طبق رابطه‌ی زیر بیان می‌شوند (m برابر است با تعداد متغیرها (بیت‌ها)ی تابع).

(تعداد سطوح با قابلیت تزویج =) $(m - \text{۱})$

تعداد گرهی هم که عمل تزویج در سطح iام نسبت به آنها انجام می‌گیرد برابر است با:

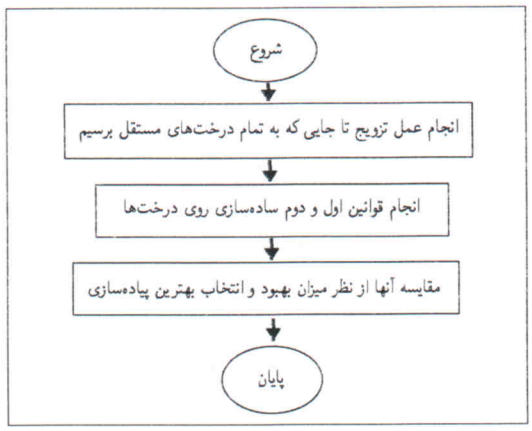
(تعداد گره‌های مزدوج شده در سطح iام =) (۲^{m-i})

در رابطه‌ی فوق $o > \text{۱}$ است، زیرا عمل تزویج برای گره‌های اولین سطح تعریف نمی‌شود. مقدار نهایی i هم برابر با تعداد سطوح تعریف شده در بالاست.

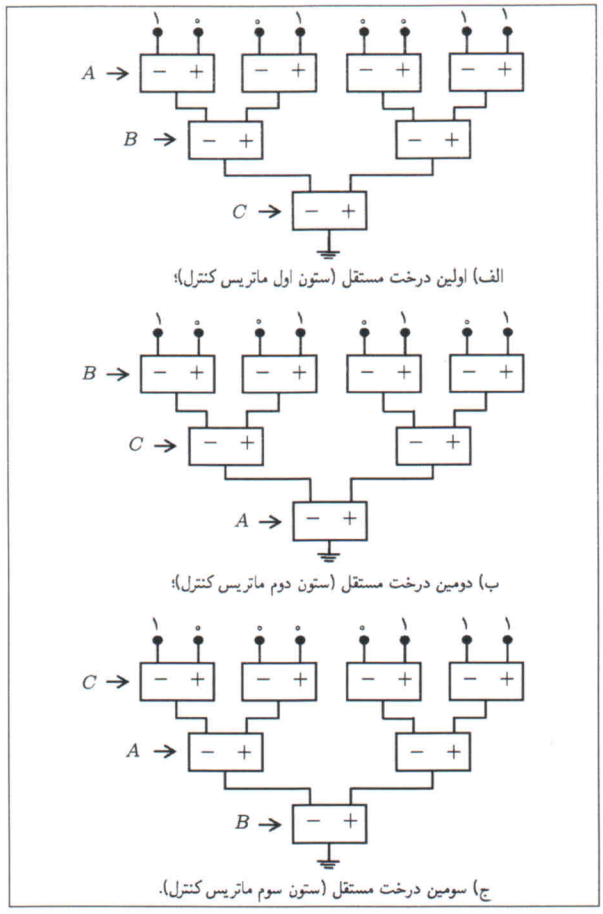
همانطورکه در شکل ۱۰ ملاحظه می‌کنید، پس از به‌دست آوردن تمامی درخت‌های مستقل، قوانین اول و دوم ساده‌سازی را روی آنها پیاده می‌کنیم. سپس نتایج به‌دست آمده از انجام این عملیات را با یکدیگر مقایسه و ساده‌ترین آنها را به‌عنوان نتیجه‌ی نهایی الگوریتم

جدول ۲. چینش ورودی‌ها به‌ازاء ترتیب ورودی‌ها.

کنترل / جایگاه ورودی	A, B, C	B, C, A	C, A, B
$In\circ$	$I_\circ = \text{۱}$	$I_\circ = \text{۱}$	$I_\circ = \text{۱}$
$In\text{۱}$	$I_\text{۱} = \circ$	$I_\text{۲} = \circ$	$I_\text{۴} = \circ$
$In\text{۲}$	$I_\text{۲} = \circ$	$I_\text{۴} = \circ$	$I_\text{۱} = \circ$
$In\text{۳}$	$I_\text{۳} = \text{۱}$	$I_\text{۶} = \text{۱}$	$I_\text{۵} = \circ$
$In\text{۴}$	$I_\text{۴} = \circ$	$I_\text{۱} = \circ$	$I_\text{۲} = \circ$
$In\text{۵}$	$I_\text{۵} = \circ$	$I_\text{۳} = \text{۱}$	$I_\text{۶} = \text{۱}$
$In\text{۶}$	$I_\text{۶} = \text{۱}$	$I_\text{۵} = \circ$	$I_\text{۳} = \text{۱}$
$In\text{۷}$	$I_\text{۷} = \text{۱}$	$I_\text{۷} = \text{۱}$	$I_\text{۷} = \text{۱}$



شکل ۱۰. الگوریتم نوین.



شکل ۱۱. درخت‌های مستقل (مرحله اول الگوریتم).

معرفی می‌کنیم. حال الگوریتم جدید را روی تابع f (که در قسمت دوم پیاده‌سازی شده بود) پیاده می‌کنیم. در شکل ۱۱ که مراحل تشکیل درخت‌های مستقل نشان داده شده است، هرکدام از اشکال آن یک درخت کاملاً مستقل از دیگری است.

با نگاهی به این درخت‌ها می‌توان دریافت که پس از ساده‌سازی توسط قوانین اول و دوم از میان درخت‌های مستقل فوق تنها شکل ۱۱ ب ساده‌ترین حالت را پیشنهاد می‌کند. نتیجه‌ی نهایی اجرای الگوریتم را در شکل ۱۲ مشاهده می‌کنید. همانطور که ملاحظه می‌کنید در پیاده‌سازی جدید نسبت به پیاده‌سازی قبلی ۲۰٪ بیشتر ساده‌سازی انجام شده است. وضعیت پیاده‌سازی جدید از نظر فضای اشغالی و سرعت نیز نسبت به خروجی الگوریتم قدیمی به‌مراتب بهتر است.

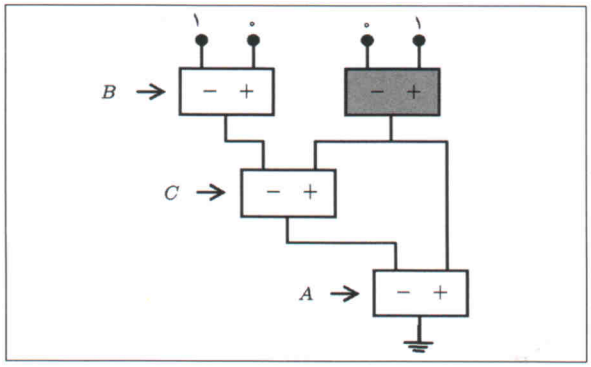
در این پیاده‌سازی، دو ترانزیستور کم‌تر از پیاده‌سازی پیشین صرف شده است. در شیوه‌یی که ۱۲۳dd نام نهاده‌اند[۱۴] میزان بهبود در تعداد ترانزیستورهای مصرفی در درخت DCVS (بدون در نظر گرفتن جفت pFET) برای تابع

$$F = ۱۰۱۱\ ۰۰۰۰\ \ ۱۰۱۱\ ۰۰۱۱\ \ ۱۰۱۱\ ۱۰۰۱\ \ ۱۰۱۱\ ۰۰۰۱$$

بیش از ۱۵٪ (از ۲۶ ترانزیستور به ۲۲ ترانزیستور) ذکر شده است، در صورتی که اگر تابع F را با روش نوین پیاده‌سازی کنیم، به ۱۶ ترانزیستور بیشتر نیاز نداریم، که این خود بهبودی نزدیک به ۳۴٫۵٪، نسبت به پیاده‌سازی اولیه و ۲۷٫۲۷٪ (از ۲۶ ترانزیستور به ۱۶ ترانزیستور)، نسبت به پیاده‌سازی معرفی شده در مرجع ۱۴ را نشان می‌دهد. در مثالی دیگر می‌توان به یک پیاده‌سازی نمونه اشاره‌کرد.[۷] که اگر آن را نیز طبق روش جدید پیاده کنیم به بهبودی معادل ۲۰٪ در تعداد ترانزیستور مصرفی دست خواهیم یافت. شایان ذکر است که این الگوریتم نیز همانند سایر الگوریتم‌های معرفی شده[۱۱،۱۸-۲۰] در توابع متقارن ساده‌سازی را بیش از آنچه که در الگوریتم‌های اولیه انجام می‌گرفت انجام نمی‌دهد.

در جدول ۳ نتیجه‌ی آزمون و مقایسه‌ی الگوریتم پیشنهادی با الگوریتم‌های موجود در بسته‌ی CUDD(بسته دیاگرام تصمیم‌گیری دانشگاه کلورادو) نسخه‌ی ۱، ۳، ۲[۲۱] روی سیستمی با پردازنده Pentium۴، حافظه ۵۱۲ مگابایتی و سیستم عامل UNIX، آورده شده است.

مدارات انتخاب شده برای انجام این مقایسه اغلب جز مدارهای benchmark [۲۲] هستند. تابع F نامبرده شده در بالا را نیز نمونه‌ی ما



شکل ۱۲. نتیجه مرحله نهایی اجرای الگوریتم.

جدول ۳. نتایج برای مدارات انتخابی benchmark.

مدارهای benchmark	الگوریتم‌های CUDD			الگوریتم پیشنهادی	بهبود
	Random	Sym.Sift	Window		
	تعداد گره‌ها	تعداد گره‌ها	تعداد گره‌ها	تعدادگره‌ها	
نمونه‌ی ما	۱۱	۱۰	۱۱	۸	٪۲۰
5xp1	۹۸	۹۴	۸۹	۷۴	٪۲۰
9sym	۲۵	۲۵	۲۵	۲۵	-
alu2	۲۷۴	۱۹۹	۲۴۹	۱۵۸	٪۲۰
b12	۱۰۵	۹۴	۱۱۱	۸۷	٪۷
c8	۱۳۶	۱۳۹	۱۳۶	۸۳	٪۳۹
cc	۱۰۸	۹۵	۱۰۸	۶۰	٪۳۶
con1	۱۷	۱۸	۱۹	۱۶	٪۵
rd53	۱۷	۱۷	۱۷	۱۷	-
rd84	۴۲	۴۲	۴۲	۴۲	-
squar5	۵۵	۵۶	۶۴	۳۵	٪۳۶
t481	۲۱	۲۱	۲۱	۲۱	-

(our_test) نامیده‌ایم. در ستون بهبود، در پیاده‌سازی برخی مدارات با هیچ‌کدام از الگوریتم‌ها بهبودی مشاهده نمی‌شود؛ به‌بیان دقیق‌تر به‌عبارت بهتر تعداد گره‌ها در تمام پیاده‌سازی‌ها با هم برابرند که این امر نشان‌دهنده‌ی متقارن بودن آن تابع است.

به‌عنوان مثال تابع جمع یا رقم‌نقلی در یک تمام‌جمع‌گر را درنظر بگیرید. از آنجاکه مقدار این توابع در ۴ میان‌ترم یک و در ۴ میان‌ترم باقی‌مانده صفر است، در پیاده‌سازی با هر الگوریتمی از نظر تعداد گره مورد استفاده نتیجه‌ی یکسانی حاصل خواهد شد.

بررسی پیچیدگی زمانی الگوریتم

هدف ما در این قسمت بررسی میزان هزینه‌ی زمانی این الگوریتم است. اگر هزینه‌ی زمانی هر عمل تزویج τ_μ را فرض کنیم، هزینه زمانی عملیات تزویج برای سطح iام برابر است با:

$$\left(۲^{m-i}\right).\tau_\mu$$

در ضمن اگر عملیات تزویج یک‌بار و به‌طورکامل روی درختی اجرا شود، هزینه‌ی زمانی آن برابر است با:

$$\sum_{i=۲}^{(m-۱)} \left(\left(۲^{m-i}\right).\tau_\mu\right)$$

به‌راحتی قابل درک است که به منظور رسیدن به تمامی درخت‌های مستقل لازم است درخت اصلی چندین بار به‌طورکامل مزدوج شود. این تعداد برای رسیدن به تمامی ترتیب‌ها برابر با $(m-۱)$ بار است. در نتیجه هزینه‌ی زمانی لازم برای رسیدن به تمامی درخت‌های مستقل برابر است با:

$$(m-۱)\left(\sum_{i=۲}^{(m-۱)} \left(\left(۲^{m-i}\right).\tau_\mu\right)\right)$$

در رابطه‌ی فوق مشخص است که بخش جمع یک صورت از سری هندسی با قدر نسبت ۲ است، با در نظرگرفتن این خاصیت:

$$(m-۱)\left(۲^{m-۱}-۳/۵\right).\tau_\mu$$

حال اگر هزینه‌ی زمانی ساده‌سازی توسط قوانین اول و دوم برای هر درخت را ثابت، و برابر با τ_θ فرض کنیم، آنگاه هزینه‌ی زمانی عمل ساده‌سازی نیز برابر خواهد بود با:

$$m.\tau_\theta$$

اگر از زمان مقایسه در برابر این زمان‌ها صرف‌نظر کنیم، آنگاه هزینه‌ی زمانی عمل ساده‌سازی بدین صورت خواهد بود:

$$(m-۱)\left(۲^{m-۱}-۳/۵\right).\tau_\mu + m.\tau_\theta$$

این در حالی است که هزینه‌ی زمانی الگوریتم ارائه شده توسط محققان[۱۱] به‌صورت زیر است.

$$\sum_{i=۱}^{m}(m-i).۲^{m-i}$$

پیچیدگی زمانی این الگوریتم و الگوریتم ما برابر با $O(m.۲^m)$ است. در یک روش ارائه شده[۷] پیچیدگی زمانی برابر با $O(m!.۲^m)$ است و در روشی دیگر[۱۳] پیچیدگی زمانی آن برابر با $O(m^۲.۳^m)$ می‌باشد. الگوریتم ارائه شده در این نوشتار ضمن برابری پیچیدگی زمانی با الگوریتم بهینه‌ی موجود، امکان ساده‌سازی بیشتر را نیز فراهم آورد.

۴. نتیجه‌گیری

در این نوشتار پس از بررسی روش‌های متداول در پیاده‌سازی درخت پایین‌بر، الگوریتمی جدید بدین منظور پیشنهاد شد. این الگوریتم به‌دلیل استفاده از قانون تزویج، که در این نوشتار ارائه شد، توانست به ساده‌سازی بیشتر در پیاده‌سازی مدارات کمک کند (جدول ۳). این قانون که از آن به‌عنوان قانون سوم ساده‌سازی نیز نام برده‌ایم، امکان ساده‌سازی بیشتر را توسط قوانین اول و دوم فراهم می‌آورد. ساده‌سازی در تابع سه‌متغیره‌یی که به‌عنوان نمونه پیاده شد، ٪۲۰ بیشتر از الگوریتم قبلی مشاهده شد. در سایر مقایسه‌های انجام گرفته با روش‌های معمول استفاده شده در بسته‌ی CUDDبه حداکثر بهبودی معادل ٪۳۹ نیز دست یافتیم. این درحالی است که الگوریتم معرفی شده در این نوشتار از نظر هزینه (پیچیدگی) زمانی با بهترین الگوریتم موجود برابری دارد و هر دو از پیچیدگی زمانی $O(m.۲^m)$ برخوردارند.

پانوشت

1. Differential Cascode Voltage Switch(DCVS)
2. dynamic DCVS
3. sample-set differential logic
4. enable/disable CMOS differential logic
5. differential current switch logic
6. gate
7. drain
8. latch
9. node (نقطه‌ی تماس شاخه‌ها با درخت)
10. bound-set
11. binary decision diagram

منابع

1. K.M. Chu, D.L. Pulfrey, "A comparison of CMOS circuit techniques: differential cascode voltage switch logic versus conventional logic", *IEEE J. Solid-State Circuits,* **SC-22**(4), pp. 528-532 (Aug 1987).
2. K.M. Chu, D.L. Pulfrey, "Design procedures for differential cascode voltage switch logic circuits", *IEEE J. Solid-State Circuits,* **SC-21**(4), pp. 1082-1087 (Dec 1986).
3. Uyemura, CMOS Logic Circuit Design, Kluwer, ISBN 0-7923-8452-0 (1999).
4. O. Kavehie, K. Navi, "A novel 54x54-bit scalable multiplier architecture", Zanjan, Iran 13th *Iranian Conf. Electrical Engineering,* pp. 367-371 (May 2005).
5. F.S. Lai, W. Hwang, "Design and implementation of differential cascode voltage switch with pass-gate (DCVSPG) logic for high-performance digital systems", *IEEE J. Solid-State Circuits,* **32**(4), pp. 563-573 (April 1997).
6. O. Kavehie, K. Navi, "A new design for 27:2 compressor", 10th *Annual Computer Society of Iran on Computer Conf.,* Tehran, Iran (15-17 Feb. 2005).
7. T. Karoubalis, G.Ph.Alexiou, N. Kanopoulos, "Optimal synthesis of differential cascode voltage switch (DCVS) logic circuits using ordered binary decision diagrams (OBDDs)", *Proc. of Euro-DAC-95 with Euro-VHDL-95,* pp. 282-287, Brighton, UK (Sep 1995).
8. T.A. Grotjohn, B. Hoefflinger, "Sample-set differential logic (SSDL) for complex high- speed VLSI", *IEEE J. Solid-State Circuits,* **SC-21**(2) pp. 367-369 (April 1986).
9. R.E. Bryant, "Graph-Based algorithm for boolean function manipulation", *IEEE Trans. on Computers,* **35**(8), pp. 677-691 (Aug 1986).
10. S.W. Jeong, "Binary decision diagrams and their applications to implicit enumeration techniques in logic synthesis", PhD thesis, University of Colorado, (1992).
11. N. Kanopoulos, N. Vasanthavada, "Testing of differential cascode voltage switch (DCVS) circuits", *IEEE J. Solid-State Circuits,* **SC-25**(3), pp. 806-812 (June, 1990).
12. R. Nair, D. Brand, "Construction of optimal DCVS trees", *IBM T.J. Watson Research Center,* NY, Tech. Rep. RC11863 (1986).
13. S.J. Friedman, K.J. Supowit, "Finding the optimal variable ordering for binary decision diagrams", *IEEE Trans. on Computers,* **39**, Issue. 5, pp. 710-713 (May 1990).
14. A. Jaekel, "Constructing testable differential pass-transistor logic circuits", *Journal of Electronic and Computer Engineering,* **27**(2), PP. 83-87 (April 2002).
15. R.L. Ashenhurst, "The decomposition of switching functions", *Proceedings of the International Symposium on the Theory of Switching,* Part I (vol. XXIX, Ann. Computation Lab. Harvard), Harvard University Press, Cambridge, pp. 75-116 (1959).
16. M. Teslenko, A. Martinelli, E. Dubrova, "Bound-Set Preserving ROBDD Variable Orderings May Not Be Optimum", *IEEE Trans. on Computers,* **54**(2), pp. 236-237 (Feb 2005).
17. E. Dubrova, L. Macchiarulo, "A comment on graph-based algorithm for boolean function manipulation", *IEEE Trans. on Computers,* **49**(11), pp. 1290-1292 (Oct 2000).
18. L.G. Heller, et al., "Cascode voltage switch logic: a differential CMOS logic family", *ISSCC84 Digest,* pp. 16-17 (Feb. 1984).
19. D. Somasekhar, K. Roy, "Differential current switch logic: a low power DCVS logic family", *IEEE J. Solid-State Circuits,* **31**(7), pp. 981-991 (July 1996).
20. S.H. Lu, "Implementation of iterative networks with CMOS differential logic", *IEEE J. Solid-State Circuits,* **23**(4), pp. 1013-1017 (August 1988).
21. Fabio Somenzi, (CUDD v.2.3.1 Package) ftp://vlsi.colorado.edu/pub/.
22. Kettle N., King A., "An anytime symmetry detection algorithm for ROBDDs", *Asia and South Pacific Design Automation Conference,* , pp.243-248,UK (24-27 Jan. 2006).