論 文

高級 하드웨어 記術 言語 設計에 관한 硏究

正會員 金 泰 憲* 正會員 李 康 煥* 正會員 鄭 周 洪* 正會員 安 致 得*

A Study on Design of a High Level Hardware Description Language

Tae Hun Kim*, Kang Whan Lee*, Ju Hong Jeong*, Chie Teuk Ahn* *Regular Members*

要 約

本 論文에서는 高 水準 論理 合成을 위한 새로운 高級 하드웨어 記述言語인 ASPHODEL(Algorithm Synthesis Pascal Hardware for Optimal Design and Efficient Language)과 알고리듬 컴파일러를 提案한다.

ASPHODEL은 VLSI의 最上位 設計 레벨인 알고리듬 레벨에서 하드웨어 特性을 효율적으로 表現할 수 있다. VLSI 複雜度를 效率的으로 處理하기위해 入出力 포트와 階層的 처리기들로 하드웨어를 모델화하고, 記述의 效率性을 위해 프로그래밍 언어인 Pascal의 構文을 基本으로 採擇하여 高級하드웨어 記述言語로서 高 水準 論理 合成 시스템에 利用될 수 있도록 하였다.

알고리듬 컴파일러는 ASPHODEL 記述을 入力으로하여 語彙 分析과 構文 分析을 거쳐 中間 레벨의 設計 表現으로 變換한다.

提案된 ASPHODEL과 알고리듬 컴파일러에 實際 設計 예를 適用, 說明함으로써 그 效用性을 보인다.

ABSTRACT

A new high level Hardware Description Language, ASPHODEL(Algorithm Synthesis Pascal Hardware for Optimal Design and Efficient Language), and its algorithm compiler for high level synthesis are described in this paper.

The new HDL, appropriated to the description of algorithmic level and lower level, models VLSI circuits as an abstracted block which is consisted of input /output ports and hierachical processors to control VLSI complexities with efficiency. Also, in order to improve the descriptive power, popular Pascal programming language is modified to build ASPHODEL syntax rules.

ASPHODEL algorithm complier generates an intermediate form through lexical and syntax analysis from ASPHODEL source codes.

To show the validation of presented language and its compiler, those are applied to practical design examples.

*韓國電子通信硏究所
ETRI
論文番號 : 93-63

Ⅰ. 序 論

IC 칩의 集積度가 증가함에 따라 設計 自動化에 대한 要求가 크게 증가하고 있다. 從前의 設計 經驗에 의한 manual 設計에 비해 設計의 정확도와 시간 短縮을 통해 설계 費用이 크게 줄어드는 魅力 때문에 빠른 속도로 基本 槪念들이 定立되면서 우수한 연구 결과가 많이 發表되고 있다. 최근의 動向은 論理合成에 많은 연구 結果가 소개되고 있으며, 그 이유는 레이아웃 분야의 상당한 實用性에 近接함에 起因한다고 사료된다.[1]

Dave Johannsen이 1979년 "Bristle Blocks"[2]를 發表하면서 소개된 실리콘 컴파일러에 대한 魅力은 高 水準 論理 合成의 槪念이 확고히 確立되면서 더욱 커지고있는 趨勢이다. 무엇보다 실리콘 컴파일러의 문제점은 얼마나 實用性에 近接할 수 있는지에 硏究 方向이 歸着되는데, 最近 VHDL[3]의 標準化에 따라 지금까지 散在해져있던 硏究 結果들이 하나로 集約되면서 조만간 이 문제도 말끔히 解決되리라 여겨진다. 하지만 현재까지는 하드웨어 記述言語와 實際 具現 문제사이의 最適의 trade off에 대한 解答을 구하는데 많은 試行錯誤를 격고있는 실정이다.[4]

本 論文에서는 이를 위해 먼저 設計者가 設計 飼養으로 부터 곧 바로 직접 記述할 수 있는 high level abstraction 槪念을 갖고 또한 이의 다음 段階의 translation을 고려한 새로운 하드웨어 記述言語인 ASPHODEL을 設計하고, 實際 變換 可能性을 보이기위해 알고리듬 컴파일러를 具現한다. 이때 알고리듬 컴파일러는 일반적인 高水準 論理 合成 시스템의 入力 形態인 中間 그래프에 相應하는 table을 生成하는 前 處理과정으로서 意味를 지닌다. 제안된 하드웨어 記述言語와 컴파일러에 實際 設計 예를 適用하여 DFG(Data Flow Graph)[25]와 比較 檢討를 行함으로서 각각의 有用性을 보인다.

Ⅱ. 하드웨어 記述 言語의 槪要

1. 旣存의 硏究 分析

하드웨어 記述 言語에 대한 硏究는 1960년대 말 CDL[6], DDL[7], AHPL[8] 등을 筆頭로 設計레벨에 따라 다양한 硏究가 進行되어 오고 있다.[9]-[15]

本 절에서는 기존에 제안된 記述 言語 가운데 대표적인 몇가지 言語들의 分析 結果를 소개한다. 比較 分析의 가늠자는 [16]에서 指摘한 바와 같이 記述의 容易性, 解讀性, 表現性, 계층적 設計 支援 가능성 및 變換 가능성을 중심으로 比較 分析하여 새로 設計할 하드웨어 記述 言語가 지녀야 할 바람직한 形態를 定立하고 그들간의 절충점을 摸索하는 基本 자료로 活用함을 目的으로 하고자 한다. 다음에 이를 정리하였다.

分析된 하드웨어 記述 言語 대부분이 디지틀 設計 및 시뮬레이션을 목표로 開發되었지만, 실제 高 水準 論理 合成 言語로 쓰이기에는 부족한 점이 많았다. 우선 CDL, DDL, AHPL 등은 記述이 어렵고 행위 단계의 記述은 행하기 어려운 것으로 판단되며, MacPitts[17]의 경우는 실리콘 컴파일러의 入力 言語로 선정되어 合成은 가능하나 관련 연산자나 제어문이 풍부하지못하고, Lisp에 가까운 構文 構造로서 readability가 다소 떨어진다.

한편, DoD에 의해 발표된 VHDL[3][20]은 매우 다양한 設計 方式으로의 接近을 허용한다는점과 technology에 독립적인 장점을 갖고 있어 高 水準 論理 合成 시스템의 入力 言語로 폭 넓게 응용될 수 있을 것으로 사료된다. 하지만 言語 자체가 標準 언어로서 開發 目的을 두어 凡用的으로 쓰일 수 있는 多樣性이 오히려 構文의 과대한 복잡함을 줄 수 있으므로 構文의 완전 熟知의 용이성에 問題点을 露出하고 있고, 그 결과 컴파일러(합성 시스템)의 具現이 매우 어려운 短點이 지적되고 있다. 현재로는 전체 構文을 완전히 지원하는 VHDL관련 設計 道具는 주로 시뮬레이터(예를들면 Vantage사의 spreadsheet, Viewlogic사의 Viewsim 등)에 局限되어 있으며, 高 水準 合成 시스템과 VHDL 전체 構文의 실용성있는 結合은 차후 해결되어야할 難題로 남아있다.[26]

2. 高 水準 合成用 하드웨어 記述 言語의 開發 方式

本 論文에서는 高 水準 合成 시스템을 위한 專用 言語 開發을 위해 記述 言語를 먼저 高 水準 論理 合成 시스템들의 일반적인 合成 흐름과 고려된 최적 합성 요소들(예를들면 complilation process와 intermediate form처리 기법)을 土臺로 하드웨어 記述 言語의 구문과 어의를 다음과 같은 목표를 기본으로 設定하여 高 水準 合成에 실질적으로 應用될 수 있는 HDL을 設計하려하였다.

첫째, 合成의 용이성을 최대로 反映하되 記述의 容易性, 解讀性도 함께 지원한다.

둘째, 高 水準 즉, 알고리듬 레벨 記述의 效率性을 지원하기위해, data와 그들간의 순서적인 制御로 實現되는 procedural language 形態로 開發한다.

세째, 앞의 目標에 프로그래밍 環境을 첨가하여 設計하고자하는 하드웨어와 이의 表現을 위한 프로그래밍 環境의 양 側面을 同時에 고려하여 構文을 設定한다. 이를위해 기존의프로그래밍 言語 가운데 배우기 쉽고 解讀하기 쉬운 Pascal[21]을 擴張하여 構文을 定義하고, 아울러 行爲 段階의 시뮬레이터 具現에 용이성을 갖도록 配慮하도록 한다.

Ⅲ. 하드웨어 記述 言語의 設計

1. 하드웨어의 모델링

本 論文의 ASPHODEL은 VLSI 複雜度를 효율적으로 處理할수 있도록 high abstraction 槪念을 갖는 하드웨어 모델을 選定했다. 즉, 入力 데이타를 위해 clear defined bit length를 갖는 入力 포트와 入力 데이타의 실제 計算, 貯藏, 制御의 procedural 處理를 위한 階層性을 지닌 假象 機能 블록들과 그 결과를 出力하는 出力 포트로 全體 하드웨어의 main module을 모델화했고, 假象 블록들의 階層性은 submodule들로 모델화했다. 이때 이와같은 狀況의 ASPHODEL 표현은 시스템의 入出力 관계는 main module宣言部에서 main入出力포트로 宣言되며, 계층적 機能 블록들은 sub_module(즉, procedure)들에서 入出力 wire포트 變數로, 기능 블록의 세부적인 표현은 statement들로 자연스럽게 매칭되면서 나타난다.

2. ASPHODEL의 syntax表現 規則

먼저 終端 記號(terminal symbol)는 英文字 혹은 ' '(single quotation mark)를 사용하여 表記하고, 非終端記號(nonterminal symbol)는 < >(angular brackets) 내에 英文字와 _(under line)으로 表記한다. Syntax rule은 S::=E의 形態로 나타나며 S는 非終端 記號를, E는 S에 對應되는 構文 表現式(syntax expression)을 나타낸다. 아울러 E를 表現함에 있어 選擇, 反復, 存在 有無를 | (or), { }(brace), [ ](square bracket)을 사용한다. 한편, 앞의 세 記號를 構文 表現의 構成員으로 사용할때는 반드시 ' '(single quotation mark)를 사용한다.

3. ASPHODEL의 基本 語彙

ASPHODEL의 基本 語彙에는 豫約語, 演算子, 英文字, 숫자 등이 있는데 說明의 편이를위해 英文字와 숫자에 대한 構文 表現을 앞에서 언급한 表現 規則을 사용하여 설명한다. 豫約語와 演算子에 대한 세부적인 定義는 附錄을 參照하기를 바란다.

3.1 Comment

Comment는 { } 안에 記入하여 記述하는 어느 位置에서나 參考도록하여 documental design 方式을 支援하도록 했다. 다음은 이의 예이다.

(사용 예)

```
{
This is an example of ASPHODEL comment.
}
```

3.2 Identifier

設計者가 임의대로 사용할 수 있는 이름은 英文字와 숫자, . (period), _(under line) 등으로 구성되며 構文 定義는 다음과 같다.

(구문 정의)

```
<identifier> ::= <letter>{<letter_or_num>}
<letter_or_num> ::= <letter>|<num>
<letter> ::= <character>|'_'|'.'
<character> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|
                K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<num> ::= 0|1|2|3|4|5|6|7|8|9
```

3.3 Constants

이에는 크게 numeric contants와 alphanumeric contants의 두 종류가 있으며 構文은 다음과 같다.

(구문 정의)

```
<contants> ::= <unsigned_number>
             |<sign><unsigned_number>
             |<contant_identifier>
             |<sign><constant_identifier>
             |<string>
<unsigned_number> ::= <unsigned_integer>
                    |<unsigned_real>
<unsigned_integer> ::= <digit>
<sign> ::= '+'
         |'-'
<constant_identifier> ::= <identifier>
<string> ::= "<character>{<character>}"
<digit> ::= {<binary_digit>}"B
          |{<octal_digit>}"O
          |{<decimal_digit>}
          |{<hexadecimal_digit>}"H
<binary_digit> ::= 0|1|<mask_digit>
<mask_digit> ::= 'f'
<octal_digit> ::= 0|1|2|3|4|5|6|7
<decimal_digit> ::= 0|1|2|3|4|5|6|7|8|9
<hexadecimal_digit> ::= 0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f
```

예로서 2진수 11'B는 십진수로는 3을, a'H는 십진수 10을 나타내며, 〈mask_digit〉는 don't care 위치의 假定을 위해 쓰일수 있다.

3.5 Termination

終結 文字는 세미콜론과 마침표가 있다. 세미콜론은 宣言部에서 하나의 宣言이 끝날때와, 文章部에서 한문장의 終結을 위해 쓰이며, 마침표는 全體 記述의 完全終了를 나타낼때 사용한다.

4. ASPHODEL의 構文과 語義

4.1 ASPHODEL의 構文의 概要

ASPHODEL로 하드웨어를 記述할 때는 먼저 program이란 豫約語로 시작하며, 名稱(identifier) 그리고 사용할 라이브러리에 대한 記述을 한다. 다음에, ASPHODEL의 하드웨어 모델인 假象박스의 入出力 및 저장 要素등에 대한 宣言을하고, 宣言部에서 宣言된 하드웨어 資源을 갖고, 실제 問題 解法에 대한 절차적인 順序를 文章部에서 記述한다. 예를들면 다음과 같은 개략적인 記述 예를 가정할 수 있다.

(사용 예)

```
program example (ASPHODEL.lib);{ASPHODEL program head part}
var a : integer  sign(16)  main iport ;{declaration part}
    temp1 : integer sign(16) temp;
    b : integer sign(16)  main oport;
begin              {statement part begin}
       temp1 := a -1;
       b := temp1;  {statement elements}
end.               {statement and program end}
```

4.2 宣言部

宣言部에는 하드웨어 資源의 形態와 情報가 하드웨어 特性 예를들면, bit特性등에 맞게 명확히 定義되고 宣言되어야 하며, 이를위한 構文은

(구문 정의)

```
<declaration_part> ::=  <global_document_declaration>
                        <label_declaration>
                        <const_declaration>
                        <type_declaration>
                        <var_declaration>
                        <procedure_and_function_declaration>
```

와 같다. 이들에 대한 세부적 構文 정의는 附錄을 보기바란다. 여기서 〈global_ducument_declaration〉은 option으로서 생략해도 無妨하지만, 設計의 전반적인 特定 情報나 事項등을 記入하여 문서화 槪念을 지원하도록하고 있으며, 〈label_declaration〉은 文章에서 goto문을 쓸경우만 나타나는 option 部分으로서 狀態 遷移에의한 FSM의 記述에 應用될 수 있는 構文構造이다.

〈const_declaration〉은 重複 記述을 事前에 방지하고, 記述의 편리성과 解讀의 용이성을 위해 static 한 情報를 미리 定義해두는 곳으로서 言語적인 觀點에서 보면 一種의 마크로에 해당한다. 이는 특히 〈bit_length_definition〉(附錄 參照)등의 記述에 유용하다. 예를들어,

(예)

```
.....
const &c = a + b; { c,a,b all must be temp variable.}
      falsepulse = 0;
      truepulse = 1;
.....
```

와 같이 사용하면, 此後에 文章部 記述이 훨씬 쉽고, 理解가 簡便해 진다.

〈type_declaration〉은 크게 〈scalar_type_declaration〉과 〈structured_type_declaration〉으로 나뉘어 지며 자세한 構文 정의는 역시 附錄에 나타냈다.

〈variable_declaration〉은 하드웨어 特性에 맞도록 〈port_variable〉〈temp_variable〉 등의 變數가 있다. 포트 變數에는 入出力 特性에 맞게 入力 포트, 出力 포트, 入出力 포트등이 있으며, 부수적으로 main, wire 등의 속성(attribution)을 가지며 main은 하드웨어 모델의 가상 박스의 入出力을 나타내며, wire는 계층적 處理器들간의 communication은 wire를 통해서 行爲 段階의 parameter passing方式으로 실행된다.

〈Temp_variable〉은 臨時로 데이타를 저장할 경우 사용되며 하드웨어 接近 槪念으로 보면, storage element에 해당한다. 예를 들어,

(사용 예)

```
.....
var variable1 : integer(3) temp; {부호비트를 포함하지않는 정수형의 3 비트 길이를
                                  갖는 temp 변수 선언}
    variable2 : one temp; {1 비트 길이를 갖는 부수 data type을 갖는 temp 변수예 }
    variable3 : integer sign(9) main iport; {부호비트를 포함하고 길이가 9 비트인 main
                                  입력포트 변수 선언}
.....
```

를 가정할 수 있다.

〈procedure_and_function_declaration〉은 하나의 모듈 單位에서 動作하는 一種의 subprogram으로서 방대한 VLSI 시스템 設計의 動作을 모듈 單位로 정의해 두고, 이들을 body部에서 계층적으로 實現함으로서 보다 효율적인 記述과 檢證의 용이성을 지원하는 중요한 槪念 중의 하나이다. 이때 入出力 port의 속성은 wire로 宣言되어야하며 入力과 出力 port만 許容된다.

Procedure와 function의 차이점은 return value의 有無에 있으며, 예를 들어,

(사용 예)

```
program .....; {program start}
.....          ; {some declarations}
function compute_delay(delay:one wire in ;strength:one wire in)one wire out;
const norminal_strength = 1;
      begin
      if (strength >= norminal_strength)
      then compute_delay := delay;
      end; { the end of function definition}
begin
.....          ; {some statements}
result := comput_delay(0, add); {function call and its assignment statement}
.....          ; {some statements}
end. {the end of program}
```

와 같이 사용가능하다.

4.3 ASPHODEL의 文章部

ASPHODEL의 文章部는 宣言部에서 宣言된 하드

웨어 資源을 實際 具現하는 問題에 대한 節次를 記述하는 곳으로서 이의 효과적인 記述을 위해서는 우선 계층적 記述이 가능해야하고, 아울러 다양한 制御 構造가 뒷받침이 되어야한다.

ASPHODEL의 制御 構造로는 when문, if문, case문, exit문, goto문등이 지원되며, 反復 構造를 위해서는 for, while, repeate문 등이 제공된다. 아울러 順次 複合文과 竝列 複合文을 두어 하드웨어 特性과 構造的과 文章 實現이 가능하도록 배려했다. 여기서 竝列 複合文이란 앞에서 언급한 하드웨어 記述 言語 開發을 위한 절충 問題에 대한 하나의 解答으로서, 엄밀히 말해서 順次와 竝列 실행이 言語의 표현상으로는 同一하여 그 정보가 隱匿(hiding)되어야 함이 좋겠지만, 設計者의 의도를 정확히 設計에 반영하고, compilation processor의 과대한 負荷를 抑制하기위해 導入된 竝列處理의 표현에 쓰이는 構文이다. 예를 들어서, 文章部에서

(사용 예)

```
.....
parbegin
      a := a + 1;
      b := b + a;
parend;
.....
```

로 記述한 결과는 a, b : = (b + a), (a + 1) ; 로 記述한 竝列 割當文과 똑같은 효과를 나타내며 竝列로 實行할 文章이 많을 경우에는 竝列 複合文을 사용함을 原則으로한다. 竝列 實行은 일반적인 順次 實行文과는 현격한 차이를 가진다. 예를 들면, 初期 값으로 a와 b가 각각 V1, V2 값을 갖는다면, 마지막 文章이 수행되고 난후의 b의 最終 값은 V1 + V2인 반면에 順次的으로 實行하면, V2 + V1 + 1이 되므로 주의하여야 한다. 보다 구체적으로 竝列 實行을 하드웨어 側面에서 설명하면, 合成時 同一 사이클(same cycle)에서 竝列 複合文내의 알고리듬을 合成 시켜야함을 設計者의 觀點에서 反映한 것이라 볼 수 있다.

4.4 ASPHODEL의 expression

ASPHODEL의 expression으로는 基本 數式, 演算 數式, boolean論理 數式, 비트 論理 數式, shift 및 rotate數式, 關係 數式등이 있으며 附錄 I에 자세히 나타냈다.

먼저, 基本 數式은 비트 參照등을 위한 간단한 割當文으로 예를 들어, pc가 8비트로 宣言되고, pc : = 11111111'B ; pc(3 : 0) : = 111'B ; pc(5) : = 0'B ; 등으로 문장부에 記述된다는 가정을 하면, 첫째 예는 pc의 모든 비트 參照를, 둘째 예는 最上位부터 最下位 4비트 參照를, 마지막 예는 最下位 부터 計算하여 5번째 비트만 參照함을 나타낸다.

演算 數式은 算術演算子를 사용하는 數式이며, 이에 사용되는 演算子는 單一 演算子로 '-', '+'와 二重 演算子로 '-', '+', '*', 'mod', 'div'등이 있다. 이때 單一 演算子인 '-'는 operand를 2의 보수화함을 의미하며, '+'는 일반적으로 省略되어도 알고리듬 컴파일러는 이를 default로 認識한다.

論理 數式에 쓰이는 論理 演算子는 boolean 論理 演算子와 비트 論理 演算子의 두 種類가 있으며, shift 및 rotate 數式을 위한 演算子는 左右 特性에 따라 shl(shift_left), shr(shift_right), rotl(rotate_left), rotr(rotate_right)등이, operand는 正數 값만 갖는 數式이 許容된다.

한편, 關係 數式에 쓰이는 關係 演算子로는 '='(等價), '< >'(非等價), '<', '<=', '>', '>='(大小 比較) 등이 있으며 그 結果 형은 항상 論理 형이다.

5. 타 하드웨어 記述 言語와의 비교

本 論文의 ASPHODEL과 다른 HDL과의 比較는 일반 프로그래밍 言語의 比較와 같이 각 言語간에 응용 目的에 따라 一長一短이 존재하므로 절대적이라 斷言할 수 없으나, 본 著者들이 타 문헌을 통한 객관적인 比較와 실제 記述을 통해 주관적 평가에 따라 행하였으며, 다음 표에 이를 要約하였다.

표를 살펴보면, 기존에 提案된 HDL 대부분이 특정 프로그래밍 言語를 하드웨어 特性과 設計 레벨에 의거 修訂 보완한 構文을 채택하고 있으며, 이는 바람직한 것으로 보인다. 그 이유는 프로그래밍 言語 자체를 HDL로 채택할 경우에는 하드웨어와 소프트웨어 간의 特性 차이(예를들면, 병렬성이나 비트 特性등)으로 표현 범위가 좁게 되는 문제가 發生하며, 하드웨어 特性에만 치중하면 記述과 判讀이 어려워지는 문제가 발생한다.

本 論文에서는 기존의 프로그래밍 言語로 알고리듬 記述이 용이하며, 기술한 것의 해독도 용이한 Pascal을 하드웨어 병렬성을 고려하고, 다양한 비트 특성, 연산자의 다양성, 라이브러리 支援, 중복 記述의 배제(매크로 개념)등을 支援하고 무엇보다 高水準의 論理에 적합하도록 變換 가능하고, 정확히 정의된 構文과 語意를 갖고 設計 사양으로 부터 곧 바로 直接 記述이 가능한 장점을 가지는 ASPHODEL을 개발하였다.

표 1. 타 하드웨어 기술 언어와의 비교
Table 1. Comparision with other HDLs

평가 항목 \ 평가된 HDL들	CDL	DDL	AHPL	ZEUS	ISPS	MACPITTS	VHDL	ASPHODEL
digital design?	○	○	○	○	○	○	○	○
계층적 설계?	×	□	□	○	○	□	○	○
구조적 설계?	×	×	×	□	□	□	○	○
documental design?	□	□	□	□	□	□	○	○
제어문의 다양성?	×	×	×	□	□	×	○	○
구문의 복잡도?	×	×	×	□	□	□	○	□
기술의 용이성?	○	□	×	□	□	□	□	○
컴파일러구성용이?	○	○	○	□	○	○	□	○
고수준 논리합성에 응용가능한가?	×	×	×	□	□	□	○	○
병렬 기술 가능?	□	□	□	□	○	○	○	○
주목적?	교육	교육	교육	구조설계	시뮬레이터	실리콘 컴파일러	표준화	고수준 합성
관련 프로그래밍 언어	알골	자체	APL	modula2	Lisp	자체	Ada	파스칼

○ 좋음, □ 보통, × 좋지못함

Ⅳ. ASPHODEL 알고리듬 컴파일러

1. 中間 形態의 重要性

高水準論理 合成 과정에서 中間 形態의 표현으로는 그래프 형태가 主流를 이루고 있으며, 이는 기존의 소프트웨어 컴파일러의 데이타 흐름 解析 [22]의 基本 理論에 근거하며, 이의 하드웨어 合成으로의 應用[23]이 최근 일반적인 흐름으로 되고 있는 趨勢이다. 예를들면, [25]의 경우 DFG(Data Flow Graph)를, [24]에서는 CDFG(Control Data Flow Graph)를 사용한다. 이때 이러한 中間 表現法을 [25]에서는 手動 變換하며, 본 논문의 알고리듬 컴파일러는 구체적인 合成의 첫번째 phase로서 CDFG에 相應하는 데이타와 構造를 지닌 中間 코우드를 生成하는 高水準論理 合成의 前 處理器로서 意味를 지닌다.

2. 알고리듬 컴파일러의 構成

CDFG에 相應하는 데이타 구조를 生成하는 알고리듬 컴파일러는 ASPHODEL로 표현된 設計 情報로부터 먼저 語彙 分析과 構文 分析 그리고 語意, 分析 등을 수행하게되며, 本 論文에서는 UNIX 上의 language 개발 道具인 LEX와 YACC를 사용하여 알고리듬 컴파일러를 具現하였다. 알고리듬 컴파일러의 흐름도는 다음 그림 1과 같다.

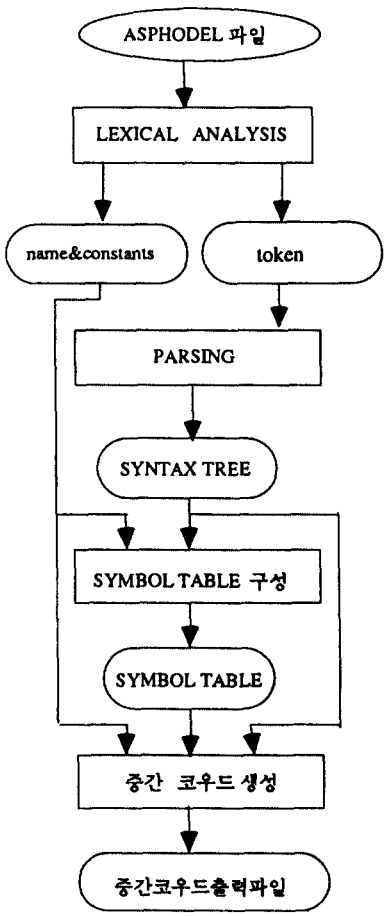

그림 1. 알고리듬 컴파일러의 흐름도
Fig. 1. a flow diagram of algorithm compiler

2.1 語彙 分析

語彙 分析 과정은 ASPHODEL로 記述된 設計 情

報를 token으로 區別하고, 名稱과 常數들에 대해서는 따로 table에 記錄한다. Token들에는 豫約語, 名稱, 常數, 區分子등이 있으며, 이때 認識된 각 token들은 해당되는 코드로 바뀌어 構文 分析의 入力으로 들어가게 된다. 아울러 lexical error가 發生하면, 해당 行 번호를 出力하고, comment는 無視한다.

2.2 構文 分析과 語意 分析

構文 分析 과정은 語彙 分析 과정의 결과인 token code를 받아서 ASPHODEL 構文 規則에 따라 파싱(parsing)하는 과정으로서, 本 論文에서는 YACC를 사용하기위해 먼저 ASPHODEL 構文 規則들을 LALR(1) 文法으로 바꾼다음, 이를 통해 構文 規則들이 認識되었을때 呼出되는 수행 코드를 프로그램하여 構文 分析器를 具現했다. 이로부터 얻어지는 결과는 構文 트리(tree)이며, 이로부터 語意 分析이 진행된다.

語意 分析 과정에서는 먼저 앞에서 生成된 構文 트리를 재귀적(recursive)으로 探索하면서 각각의 文法 生成에 관련된 정보를 수집하여 宣言部에 나타난 정보들의 意味를 分析하여 심볼 테이블을 만들고, 이를 토대로 文章部에 대해 構文 트리를 이용하여 中間 code를 生成한다.

심볼 테이블은 다음 그림 2와 같은 資料 構造 형식을 갖도록 하고, 變數들의 scope 깊이를 정확히 支援하고, 과대한 메모리占有를 피하기위해 스택(stack) 構造로 실현하였다.

명칭	깊이	속성	포인터

그림 2. 심볼 테이블의 基本 資料 構造
Fig. 2. a basic data structure of symbol table

V. 實 驗

이상에서 提案한 ASPHODEL과 알고리듬 컴파일러의 效用性을 보이기위해 실제 高水準 하드웨어 設計 예제를 갖고 實驗을 행하였다.

알고리듬 컴파일러의 實驗을 위해서 기존의 合成 예제 가운데 任意로 몇개의 예들을 選定하여 實驗을 행했다. 이때 사용된 예들은 대부분 벤치마크 예제[4]들로서 기존의 合成 시스템에서 사용되었던 대표적인 設計 예이다.

이에대한 實驗 결과의 한 예를 다음에 소개한다. 實驗에 사용된 예제는 [17]에서 사용된 예로서 이를 本 論文의 ASPHODEL로 記述하면 다음 그림 3과 같이 표현된다.

```
program example1 (ASPHODEL.lib);
{This description is to solve the equation
        √(a^2 + b^2) = max [7/8 max( |a|,|b|) + 1/2 min(a,b),max(a,b)] }

const &bit = 16; { The length of integer in this description is 16 }
var a,b : integer main iport; { System input ports are a and b; }
    aab,bab,less,great,sqs,t1,t2,t3: integer temp; { Temporary variables; }
    result: integer main oport; { System output port is result;}
{The begin of statement part;}
 begin
        if a < 0 then
                        aab := - a;
                else
                        aab := a;

        if b < 0 then
                        bab := -b;
                else
                        bab := b;
```

```
if aab > bab then
        begin
                great := aab;
                less := bab;
        end;
            else
                begin
                        less := aab;
                        great := bab;
                end;
                t1 := great div 8;
                t2 := great - t1;
                t3 := less div 2;
                sqs := t2 + t3;
                        if great > sqs then
                                                result := great;
                                        else
                                                result := sqs;
end. { system end;}
```

그림 3. ASPHODEL로 記述된 MacPitts의 예
Fig. 3. An ASPHODEL design example for MacPitts example

한편, 위의 陰數 處理 알고리듬 대신, 마스크 常數를 도입하고 비트 and 시키는 알고리듬을 사용한다면, 그림 4와 같이 또 다르게 나타난다.

```
program example2 (ASPHODEL.lib);
const sbit = 8;
{ bit length is 8 bits }
var a,b: integer main iport:
    ain,bin,a_abs,b_abs,max,min,
    mxtemp,mxout,miout,out_temp:integer temp:
    result: integer main oport:
begin { begin of system behavior description }
        ain := a;{ The op ":=" is move operation.}
        bin := b;
        a_abs := ain;
        b_abs := bin;
        if(ain band 17777777'B) = 10000000'B then
                        {if ain is a negative}
                                a_abs := bnot ain;
        if(bin band 17777777'B) = 10000000'B then
                        {if bin is a negative}
                                b_abs := bnot ain;
        max := a_abs;
        min := b_abs;
                if(b_abs > a_abs) then
                                begin
                                  max := b_abs;
                                  min := a_abs;
                                end;
                mxtemp := max shr3;
                miout := min shr1;
                mxout := max - mxtemp;
                out_temp := mxout + miout;
                        if max > out_temp then
                                        result := max;
                                {output the max value;}
end. { the end of system description;}
```

그림 4. 實驗을 위해 알고리듬 컴파일러에 入力된 ASPHODEL 記述의 한 예
Fig. 4. an ASPHODEL description for inputting of algorithm compiler

```
/*.....*/
/*   OUTPUT                    FILE      */
/*.....*/

{*****************************************}
{*  CONTROL AND DATA FLOW GRAPH DATA FILE *}
{*****************************************}
56  1  40
#1   $const  a(_8_int)  on    0
#2   $const  11111111'B  off  0
#3   $const  10000000'B  off  0
#4   $const  b(_8_int)  on    0
#5   $const  3  off  0
#6   $const  1  off  0
#7   'mv'  <-  #1  on  a_abs(_8_int)  0
#8   'band'  <-  #1  #2  on  t1(_8_int)  0
#9   'band'  <-  #4  #2  on  t2(_8_int)  0
#10  'mv'  <-  #4  on  b_abs(_8_int)  0
#11  '='  <-  #8  #3  on  bool  0
#12  '='  <-  #9  #3  on  bool  0
#13  %imerg  <-#7  #11  c1  off  0
#14  %imerg  <-#7  #11  c0  off  0
#15  'bnot'  <-#13  on  t3(_8_int)  0
#16  %selec  <-#1511  #1410  off  0
#17  %imerg  <-#10  #12  c1  off  0
#18  %imerg  <-#10  #12  c0  off  0
#19  'bnot'  <-#17  on  t4(_8_int)  0
#20  %selec  <-#1911  #1810  off  0
```

*노드 번호 20번 까지만 출력한 결과임

그림 5. 알고리듬 컴파일러의 出力 結果
Fig. 5. output result of algorithm compiler

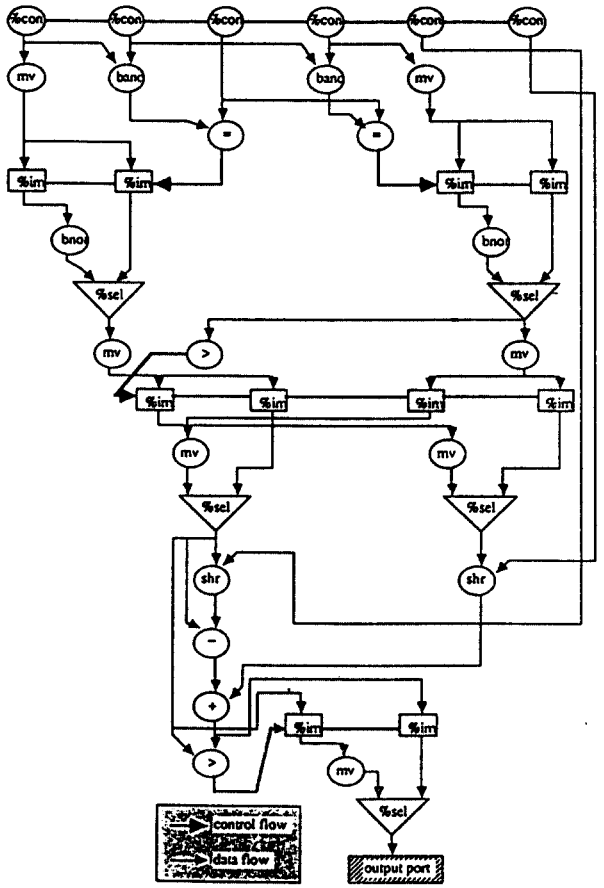

그림 6. 그림 6에 相應하는 [25]의 DFG
Fig. 6. DFG of [25] correspond with Fig.4

이를 알고리듬 컴파일러에 入力하여 最終的으로 얻은 出力 파일은 그림 5와 같으며, 이 결과는 [25]에서 manual로 그린 DFG와 比較 分析하여 올바른 中間 표현이 生成되었는지를 檢討하기위하여 심볼 테이블과 3주소 表現法을 變形하여 確認用으로 노드 番號 20번 까지만을 出力한 결과이다. 이에 대한 자세한 表現 規則은 附錄Ⅱ를 參照하기를 바란다.

이에 相應하는 [25]의 DFG는 그림 6에 나타냈고, ASPHODEL로 記述된 設計 情報가 高水準論理 合成을 위한 意味있는 表現으로 정확히 變換 되었음을 알 수 있다.

Ⅵ. 結果 및 硏究 課題

本 論文에서는 알고리듬 레벨의 하드웨어 特性을 효율적으로 表現하며, 合成에 용이한 새로운 하드웨어 記述言語로 ASPHODEL고 효과적인 最終 論理 合成을 위한 前處理 段階로서 意味를 갖는 알고리듬 컴파일러를 提案했다.

實際 設計 예와 記述 예를 통한 實驗으로 개발된 하드웨어 記述 言語의 效用性과 다음 合成 段階로의 意味있는 變換에 대해 論하였다.

앞으로의 硏究 課題로는 실제로 合成器를 具現하는 作業과 시뮬레이터에 대한 硏究가 필요하다. 아울러 標準化를 지원하기위해서 알고리듬 레벨에서의 ASPHODEL to VHDL translator 開發도 要求된다. 한편, image 處理나 HDTV 등을 위해 ASPHODEL의 DSP용 버젼에 대한 硏究가 1차로 完了된 상태이며, 체계적인 檢證과 檢討를 통해 追後 發表할 豫定이다.

```
{*********************************}
{* 부록 I : ASPHODEL SYNTAX  *}
{*********************************}

<program> ::= <program_head><block>'.'
<program_head> ::= program <identifier>['('<file_identifier>{','<file_identifier>}')'];'
<file_identifier> ::= <identifier>  {  Standard cell lib. identifier is ASPHODEL.lib in the
current
version of algorithmic compiler }
<block> ::= <declaration_part><statement_part>
<declaration_part> ::= <global_document_declaration>
                       <label_declaration>
                       <const_declaration>
                       <type_declaration>
                       <var_declaration>
<global_document_declaration> ::= <empty>
 |'{' <identifier> ',' {<identfier>}';"}'
<label_declaration> ::= <empty>
                        | label <identifier>{',' <identifier>}';'
<const_declaration> ::= <empty>
                        | const <const_definition>{';'<const_definition>}';'
<const_definition> ::= <bit_length_definition>
                       |<value_definition>
<bit_length_definition> ::= &bit '=' <unsigned_integer>
                            |&bit '=' sign <unsigned_integer>
<value_definition> ::= <value_number>
                       |<mask_number>
<value_number> ::= <identifier> '=' <constant>
<mask_number> ::= &mask<unsigned_integer> '=' {<binary_digit>}'"B
<type_declaration> ::= <empty>
                       | type <type_definition> {';'<type_definition>}';'
<type_definition> ::= <identifier> '=' <hardware>
<hardware> ::= <simple_hardware>
               |<structured_hardware>
<simple_hardware> ::= <scalar_form>
                      |<subrange_form>
                      |<predefineed_form>
<scalar_form> ::= '('<identifier>{','<identifier>')'
<subrange_form> ::= <constant> '..' <constant>
<predefined_form> ::= integer
                      | one
                      | character
                      | boolean
                      | real
<structured_hardware> ::= <array_form>
                          | <record_form>
<array_form> ::= array '[' <simple_hardware>[','<simple_hardware>]']' of <hardware>
<record_form> ::= record <field_list> end
<field_list> ::= <record_section> {';'<record_section>}
<record_section> ::= <empty>
                     |<field_identifier>{','<field_identifier>}';'<hardware>
```

```
<field_identifier> ::= <identifier>
<var_declaration>:= var <hardware_var_def>{';'<hardware_var_def>}';'
<hardware_var_def> ::= <identifier>{','<identifer>}':'<type>
<type> ::= <hardware>[<bit_spec.>][var_spec.>]
<bit_spec.> ::= '('<unsigned_integer>')'
              / sign '('<unsigned_integer>')'
<var_spec.> ::= <system_port>
               /<subsystem_port>
               /<temp>
<system_port> ::= <input_port_type>
                 /<output_port_type>
                 /<ioport_type>
<input_port_type> ::= iport / main iport
<output_port_type> ::= oport / main oport
<ioport_type> ::= ioport / main ioport
<subsystem_port> ::= <wire_input_port>
                    /<wire_output_port>
<wire_input_port> ::= wire_in
<wire_output_port> ::= wire_out
<temp> ::= <empty>
         / temp
<procedure_and_function_declaration> ::= {<procedure_or_function>';'}
<procedure_or_function> ::= <procedure_declaration>
                           /<function_declaration>
<procedure_declaration> ::= <procedure_head><block>
<procedure_head> ::= procedure <identifier>';'
                    / procedure <identifier>'('<formal_parameters>
                          {';'<formal_parmameters>}')' ';'
<formal_parameters> ::= <parameter_group>
                        / var <parameter_group>
                        / function <parameter_group>
                        / procedure <identifier> {','<identifier>}
<parameter_group> ::= <identifier> {','<identifier>}':'<type>
<function_declaration> ::= <function_head><block>
<function_head> ::= function <identifier> ':' <result_type> ';'
                  / function <identifier>
                   '('<formal_parameters> {';'<formal_parameters>}')":'<result_type>';'
<result_type> ::= <type>
{ start of statements }
<statement_part> :: begin <statement> {';'<statement>}';'end
<statement> ::= <unlabeled_statement>
              / <label>':'<unlabeled_statement>
<unlabeled_statement> ::= <simple_statement>
                         /<structured_statement>
<simple_statement> ::= <assignment_statement>
                      /<procedure_statement>
                      /<empty_statement>
<assignment_statement> ::= <serial_assignment_statement>
                          / <par_assignment_statement>
<serial_assignment_statement> ::= <variable> ':='<expression>
                                 / <function_designator> ':='<expression>
```

<par_assignment_statement>::=<variable> {','<variable>} ':=' <expression>{','<expression>}
<variable> ::= <entire_variable>
| <component_variable>
<entire_variable> ::= <identifier>[<bit_spec.>]
<component_identifier> ::= <indexed_variable>
| <field_designator>
<indexed_variable> ::= <array_variable> '['<simple_expression>
{','<simple_expression>}']'
<array_variable> ::= <variable>
<field_designator> ::= <record_variable>'.'<field_identifier>
<record_variable> ::= <variable>
<field_identifier> ::= <identifer>
<procedure_statement> ::= <procedure_identifier>
| <procedure_identifier> "('<actual_parm>{','<actual_parm>}')'
<procedure_identifier> := <identifier>
<actual_parm> ::= <expression>
| <variable>
| <procedure_identifier>
| <function_identifier>
<empty_statement> ::= <empty>
<structured_statement> ::= begin <statement> {';'<statement>}';'end
| parbegin <statement> { ';'<statement>}';'parend
| <when_statement>
| <if_statement>
| <case_statement>
| <exit_statement>
| <continue_statement>
| <go_to_statement>
| <for_statement>
| <while_statement>
| <repeat_statement>
<when_statement> ::= when <expression> whendo <statement>
<if_statement> ::= if <expression> then <statement>
| if <expression> then <statement> else <statement>
<case_statement> ::= case <expression> of <case_list_element> {';'<case_list_element> end
<case_list_element> ::= <case_label_list> ':' <statement>
| <empty>
<case_label_list> ::= <case_label>{','<case_label>}
<exit_statement> ::= exit
<continue_statement> ::= continue
<go_to_statement> ::= goto <label>
<while_statement> ::= while <expression> do <statement>
<repeat_statement> ::= repeat <statement>{';'<statement>} until <expression>
<for_statement> ::= for <control_variable> ':=' <for_list> do <statement>
<for_list> ::= <initial_value> to <final_value>
| <initial_value> downto <final_value>
<control_variable> ::= <identifier>
<initial_value> ::= <expression>
<final_value> ::= <expression>
<expression> ::= <simple_exp>
| <arithmetic_exp>

```
                    | <boolean_logical_exp>
                    | <bit_logical_exp>
                    | <shift_rotate_exp>
                    | <relation_exp>
<simple_exp> ::= <constant>
                    | <hardware_identifier> [<bit_ref>]
                    | <array_identifier>'['<expression>']'
                    | <record_identifier>'.'<element_identifier>
                    | <function_designator>
                    | '('<expression>')'
<bit_ref> ::= '('<bit_position>')'
              | '('<msb>':'<lsb>')'
<bit_position> ::= <unsigned_integer>
<msb> ::= <usigned_integer>
<lsb> ::= <unsigned_integer>
<arithmetic_exp> ::= <un_arithmentic_op><expression>
                    | <expression> <bi_arithmetic_op> <expression>
<un_arithmetic_op> ::= '-' | '+'|<empty>
<bi_arithmetic_op> ::= '-'|'+'|'*'|mod|div
<boolean_logical_exp> ::= <un_logical_op> <expression>
                          | <expression> <bi_logical_op> <expression>
<un_logical_op> ::= not
<bi_logical_op> ::= and | or | xor | nand | nor
<bit_logical_exp> ::= <bit_un_logical_op> <expression>
                          | <expression> <bit_bi_logical_op> <expression>
<bit_un_logical_op> ::= bnot
<bi_logical_op> ::= band | bor | bxor | bnand | bnor
<shift_and_rotate_exp> ::= <expression><sh_or_rot_op> <unsigned_integer>
<sh_or_rot_op> ::=  shl | shr | rotl | rotr
<relation_exp> ::= <expression> <rel_op> <expression>
<rel_op> ::= '=' | '<>' | '<' | '<=' | '>' | '>='



{*****************************************}
{* 부록II.  ASPHODEL intermediate file syntax *}
{*****************************************}
<dfg_file> ::= <file_number> <the_last_node_number> {<data_control_flow>}
<file_number> ::= <unsigned_integer>
<the_last_node_number> ::= <unsigned_integer>
<data_control_flow> ::= <node_number> <node_type_definition>
                        [<previouse_information>] [<hardware>] <flag_on_off>
                        [<jump_value_expression>] <outgoing_information>
                        <end_mark>
<node_number> ::= '#'<unsigned_integer>
<node_type_definition> ::= <const_node_def.> | <op_node_def.> | <flow_mark_def.>
                           |<sub_program_processing_node_def.>
<hardware> ::= <hardware_type_descriptor> | <design_information_descriptor>
<flag_on_off> ::= < data_synthesis_or_not>
<data_synthesis_or_not> ::= on | off
<end_mark> ::= 0
```

참 고 문 헌

1. D.D. Gajski et al., "Silicon Compilation : A Tutorial," NATO Advanced Study Institute on Logic Synthesis and Silicon Compilation for VLSI Design, July 1986.
2. D.Johanssen, "Bristle Blocks : A Silicon Compiler," Proc. 16th Design Automation Conference, pp.310-313, 1979.
3. M.Shahad et al., "VHSIC Hardware Description Language," IEEE Computer, pp.94-103, Feb.1985.
4. G.Borriello and E.Detjens, "High-Level Synthesis : Current Status and Future Directions," 25th Design Automation Conference, pp.477-482, 1988.
5. Y.Chu, "Intoducing the Computer Design Language," Proc. IEEE Computer Conference, COMPCON 72, pp.215-218, Sept. 1972.
6. D.L.Dietmeyer, "Introducing DDL," IEEE Computer, pp.34-38, Dec. 1974.
7. F.Meskinpour and M.D. Ercegovac, "A functional language for description and design of digital systems : sequential constructs," 22nd Design Automation Conference, pp.238-244, Jun. 1985.
8. F.J.Hill, "Introducing AHPL," IEEE Computer, pp.28-30, 1974.
9. M.R.Barbacci, "Instruction Set Processor Specifications(ISPS) : The Notation and its Applications," IEEE Trans. on Computers, pp.23-40, Jun. 1981.
10. M.B.Pedro et al., "HARPA : A Hierachical Multi-Level Hardware Description Language," 21st Design Automation Conference, pp.59-65, 1984.
11. W.H.Evans et al., "ADL : An Algorithmic Design Language for Integrated Circuit Synthesis," 21st Design Automation Conference, pp.66-72, 1984.
12. A.V.Goldberg et al., "Approaches Toward Silicon Compilation," IEEE Circuit and Device Magazine, pp.29-38, May 1985.
13. G.L.Granjean et al., "Utilization of a Hardware Description Language," IEEE International Conference on Computer Aided Design, pp.159-161, 1983.
14. M.C.McFarland and A.C.Parker, "An Abstract Model of Behavior for Hardware Descriptions," IEEE Trans. on Computers, pp.621-637, July 1983.
15. P.Robinson and J.Dion, "Programming Language for Hardware Description," 20th Design Automation Conference, pp.12-16, 1983.
16. D.E.Thomas and D.P.Siewiorek, "Measuring Designer Performance to Verify Design Automation Systems," IEEE Trans. on Computers, pp.48-61, Jan. 1981.
17. J.R.Southard, "MacPitts : An Approach to Silicon Compilation," IEEE Computer, pp. 74-82, 1983.
18. K.J.Lieberherr and S.E. Knudsen, "Zeus : A Hardware Description Language for VLSI," 20th Design Automation Conference, pp.17-23, Jun. 1983.
19. A.V.Goldberg et al., "Approaches Toward Silicon Compilation," IEEE Circuits and Devices Magazine, pp.29-38, May 1985.
20. C.H. Cho, N.D. Dutt et al., "Development of Logic Synthesizer from Register Transfer Level VHDL," 과학 기술처 '89 특정 연구 결과 발표회 논문집, pp.54-61, 1990.
21. K.Jenson and N.Wirth, Pascal User Manual and Report, Spinger_Verlag, 1987.
22. A.V.Aho and J.D.Ullman, Principle of Compiler Design, 3rd Edition, Addison Wesley Publishing Co.
23. R.Camposano, "Synthesis Techniques for Digital System Design," 22nd Design Automation Conference, pp.475-481, 1985.
24. A.Orailoglu and D.D.Gajski, "Flow Graph Representation," 23rd Design Automation Conference, pp.503-509, 1986.
25. V.K.Raj, "Translating Data Flow Graphs to Architectures," Ph.D. Thesis, Univ. of Iilinois at Urbana Champaign.
26. N.D.Dutt and J.R.Kipps, "Bridging High-Level Synthesis to RTL Technology Libraries," 28th

Design Automation Conterence, pp.526-529, 1991.

金 泰 憲(Tae Hun Kim) 正會員
1964年 5月 16日生
1987年 2月 : 한양대학교 공과대학 전자공학과 졸업(공학사)
1989年 2月 : 한양대학교 대학원 전자공학과(컴퓨터 공학 전공)(공학석사)
1989年 6月 ~ 1989年 12月 : 금성반도체 연구소 연구원
1990年 1月 ~ 1991年 8月 : 금성중앙 연구소 연구원
1991年 9月 ~ 1993年 2月 : 한국전자통신연구소 연구원
1993年 3月 ~ 現在 : 포항공대 전산학과 박사과정
※주관심분야 : VLSI synthesis, CAD, 영상처리, 회로및 시스템등

李 康 煥(Kang Whan Lee) 正會員
1964年 10月 6日生
1987年 2月 : 한양대학교 전자공학과 졸업
1989年 9月 : 중앙대학교 대학원 전자공학과 졸업
1989年 8月 ~ 現在 : 한국전자통신연구소 영상통신연구실 근무
※주관심분야 : VLSI 설계, 신호처리, CAD

鄭 周 洪(Ju Hong Jeong) 正會員
1957年 11月 3日生
1984年 2月 : 고려대학교 공대 전자공학과(학사)
1986年 2月 : 한국과학기술원 전기 및 전자공학과(석사)
1993年 3月 : 한국과학기술원 전기 및 전자공학과 박사과정
1986年 2月 ~ 現在 : 한국전자통신연구소 선임연구원

安 致 得(Chie Teuk Ahn) 正會員
1956年 8月 15日生
1980年 2月 : 서울대학교 공대 전자공학과(학사)
1982年 2月 : 서울대학교 공대 전자공학과(석사)
1991年 8月 : 미국 University of Florida 전기공학과(박사)
1982年 12月 ~ 現在 : 한국전자통신연구소 영상통신연구실장, 선임연구원